C/C++ セキュアコーディング

整数

2010年3月23日
JPCERTコーディネーションセンター

はじめに　～整数オーバーフローの脅威～

「CとC++言語のプログラムにおいて、整数型の取り扱いに起因する脆弱性はこれまで軽視されており、現在その数を増やしつつある」

2005年、『C/C++セキュアコーディング』 Robert C. Seacord

2007年5月にMITREが公表したレポートによると

「整数オーバーフローは過去数年のあいだトップ10に入る脆弱性であったが、今日、OSベンダのアドバイザリにおいてバッファオーバーフローに続き2番目に多い脆弱性である」

"Vulnerability Type Distributions in CVE"

URL: http://cwe.mitre.org/documents/vuln-trends/index.html

「整数」セクションのゴール

- コンピュータ上の整数の取り扱いについて理解を深める
- Cの「整数変換のルール」をマスターする
- 整数演算の仕組みとエラー条件を理解する
- 整数に関する脆弱性の発生メカニズムを理解し、脅威の緩和方法を身につける

典型的な整数の脆弱性

JPCERT CC®

```
u_int nresp;

nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp*sizeof(char*));
    for (i = 0; i < nresp; i++) {
        response[i] = packet_get_string(NULL);
    }
}

/* OpenSSH 3.3 における整数オーバーフローの脆弱性 */
```

nresp の値が1073741824のとき乗算の結果オーバーフローが発生！

整数に関する脆弱性の例 1

GNU の Bourne Again Shell (**bash**) は、Bourne Shell (**/bin/sh**)と互換性のあるプログラム

- 標準のシェルと同じ構文で、ジョブ管理、コマンド行編集、履歴などの追加機能を提供
- Linux 上で最も広く普及

バージョン 1.14.6 以前の **bash** には、任意のコマンドを実行させてしまう脆弱性が存在

脆弱性のある bash のソースコード

```
static int yy_string_get() {

    register char *string;
    register int c;

    string = bash_input.location.string;
    c = EOF;

    /* If the string doesn't exist, or is empty, EOF found. */
    if (string && *string) {
        c = *string++;
        bash_input.location.string = string;
    }
    return (c);
}
```

*string は char(plain) として宣言されている

コマンドラインから入力文字を1文字づつポインタで取り出し、int型変数 c に代入

文字コード255の値が stringから入ると…

符号拡張前: 255(0xFFFF)

符号拡張後: -1(0xFFFFFFFFFFFF)

bash のソースコードの解説

このポインタ経由で1文字ずつ取り出された文字列は、**int** 型の変数 c に格納される。

char 型をデフォルトで **signed char** に設定するコンパイラでは、この値が **int** 変数に代入されると符号拡張が発生する。

10進文字コードの 255（2の補数形式では -1）の場合、符号拡張の結果 **int** 型変数に -1 が代入される。

-1 は構文解析の別の部分でも使用され、コマンドの終わりを表す。

攻撃のシナリオ

10 進文字コードの 255(8進の 377)は、-c オプションを通じて bash にコマンドとして渡される場合、意図せずコマンド区切り文字と解釈される。

例:

- **bash -c 'ls¥377who'**

このコマンドは、ls と who の 2 つのコマンドを実行する。(**¥377** は、10 進 255 の値を持つ文字を表す)

コマンドインジェクション攻撃につながる

とられた対策

ゼロで符号拡張されるよう型宣言を変更:

```
register unsigned char *string;
```

C++ コンパイラでは代入文 `c = *string++;` でコンパイルエラーになるため、型宣言は変えず代入文で次のように対策してもよい:

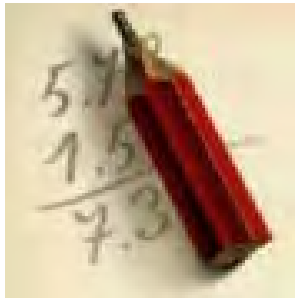
```
c = *(unsigned char*)string++;
```

整数にかかわるセキュリティ

ほとんどの C/C++ ソフトウェア開発では、整数の範囲検査が体系的に実施されていない。

- 整数に起因するセキュリティ上の弱点が存在する
- その一部が脆弱性となる可能性がある

予期せぬ整数値



予期せぬ値

机上の計算で期待されていた値とは異なる値

プログラムが正常に動作していても予期せぬ値が発生することがあり、それが脆弱性の原因となることが多い。

実行される puts()はどれ？

JPCERT CC®

```
signed char x, y;
x = -128;
y = -x;

if (x == y) puts("1");
if ((x - y) == 0) puts("2");
if ((x + y) == 2 * x) puts("3");
if (((char)(-x) + x) != 0) puts("4");
if (x != -y) puts("5");
```

JPCERT CC®

“the Spirit of C”

プログラマを信頼する。

プログラマが必要である事柄を行おうとすることを妨げない。

*JIS X 3010-1993 (ISO/IEC 9899:1990)
「プログラミング言語 C」解説より抜粋

もくじ

第一部

整数のメカニズム

⇒ 整数表現

型

型変換

演算子

- 加算/減算
- 乗算
- 除算
- 左シフト、右シフト

第二部

整数のエラー条件と脆弱性

脅威の緩和方法

まとめ

JPCERT CC®

符号付き絶対値 (signed magnitude)

1 の補数

2 の補数

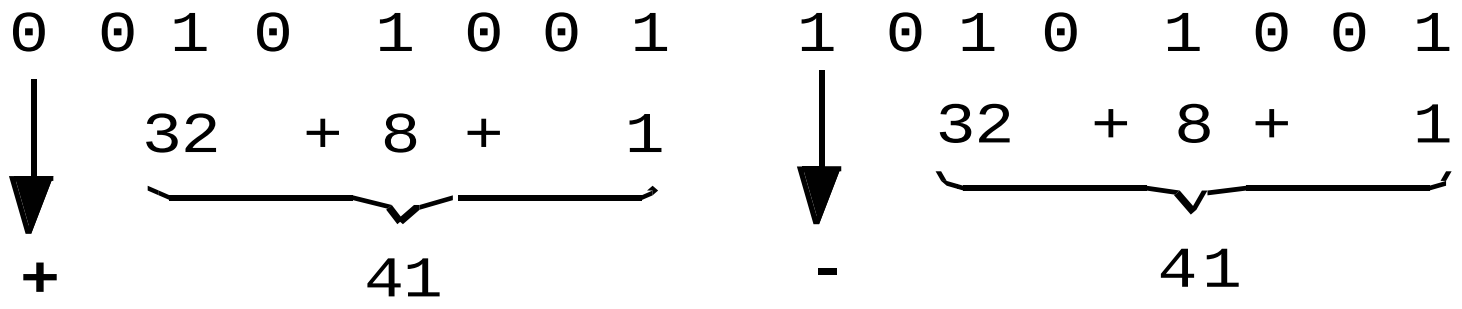
⇒負数を表現する仕組みがそれぞれ異なる

符号付き絶対値表現

符号付き絶対値表現では、最高位ビットで符号を表現

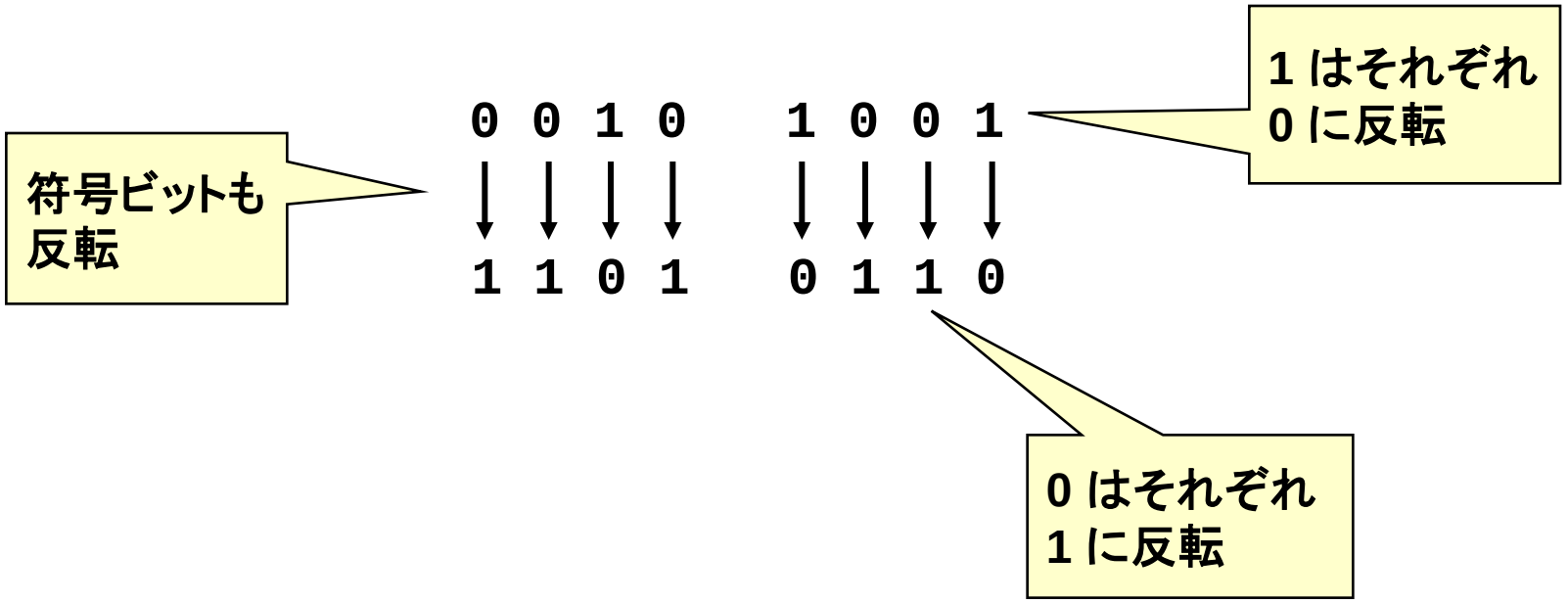
- 0は**正**
- 1は**負**
- 残りの下位ビットは、値の**絶対値**

符号付絶対値で表現した +41 と -41



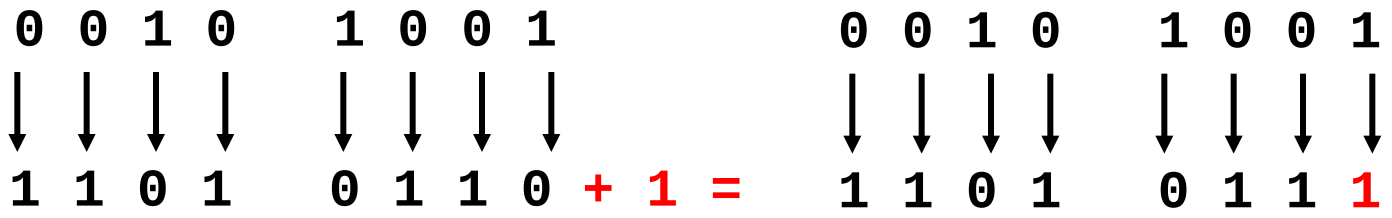
1の補数

負の数は各ビットを反転させて表現する。



2の補数

負の整数は1の補数表現に1を加えて作る。



- 2の補数表現では、単一(正)の値で0を表現
- 符号は最高位ビットで表現
- 正の数の表記は符号付き絶対値表現と同じ

もくじ

JPCERT CC®

第一部

整数のメカニズム

- 整数表現
- ⇒ 型
- 型変換
- 演算子
  - 加算/減算
  - 乗算
  - 除算
  - 左シフト、右シフト

第二部

整数のエラー条件と脆弱性

脅威の緩和方法

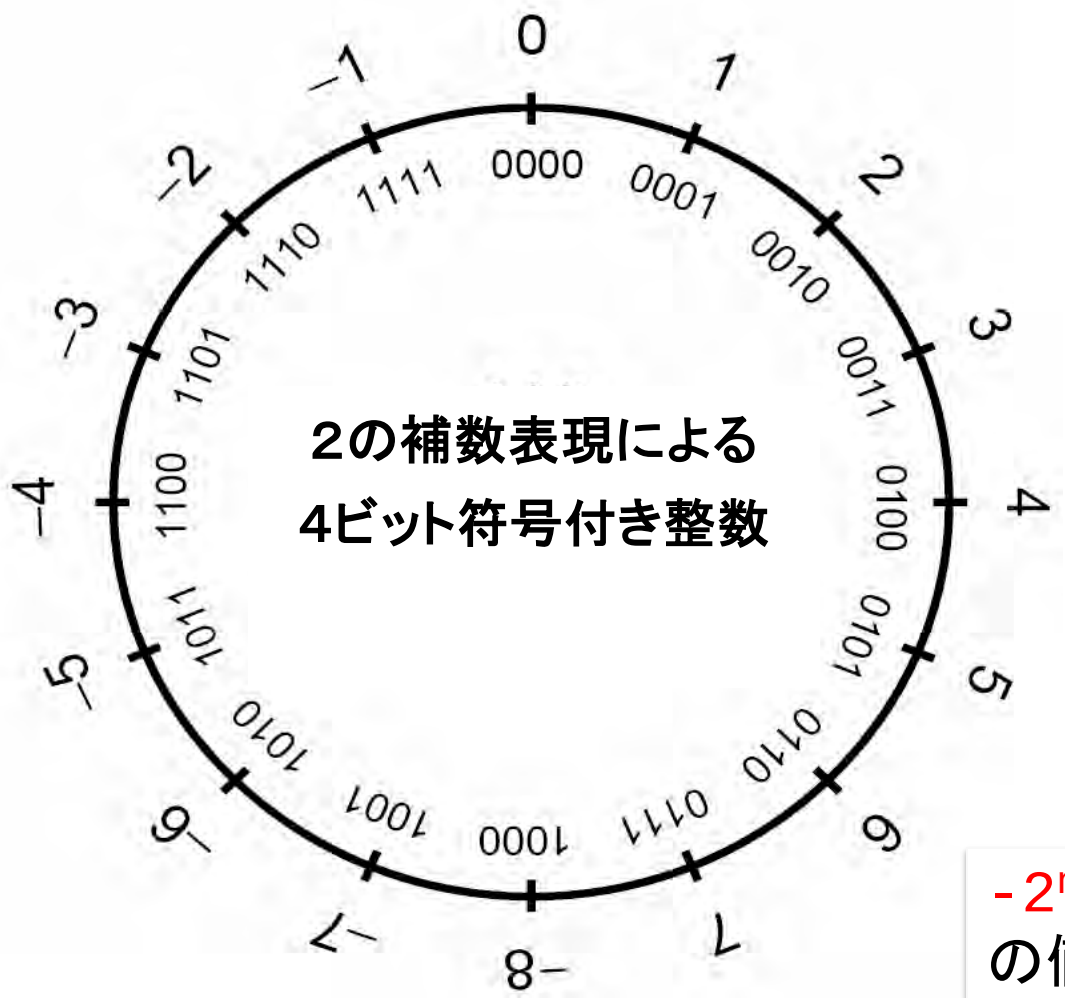
まとめ

符号付きと符号なし整数型

C/C++では、整数は **signed** か **unsigned** かのどちらか。

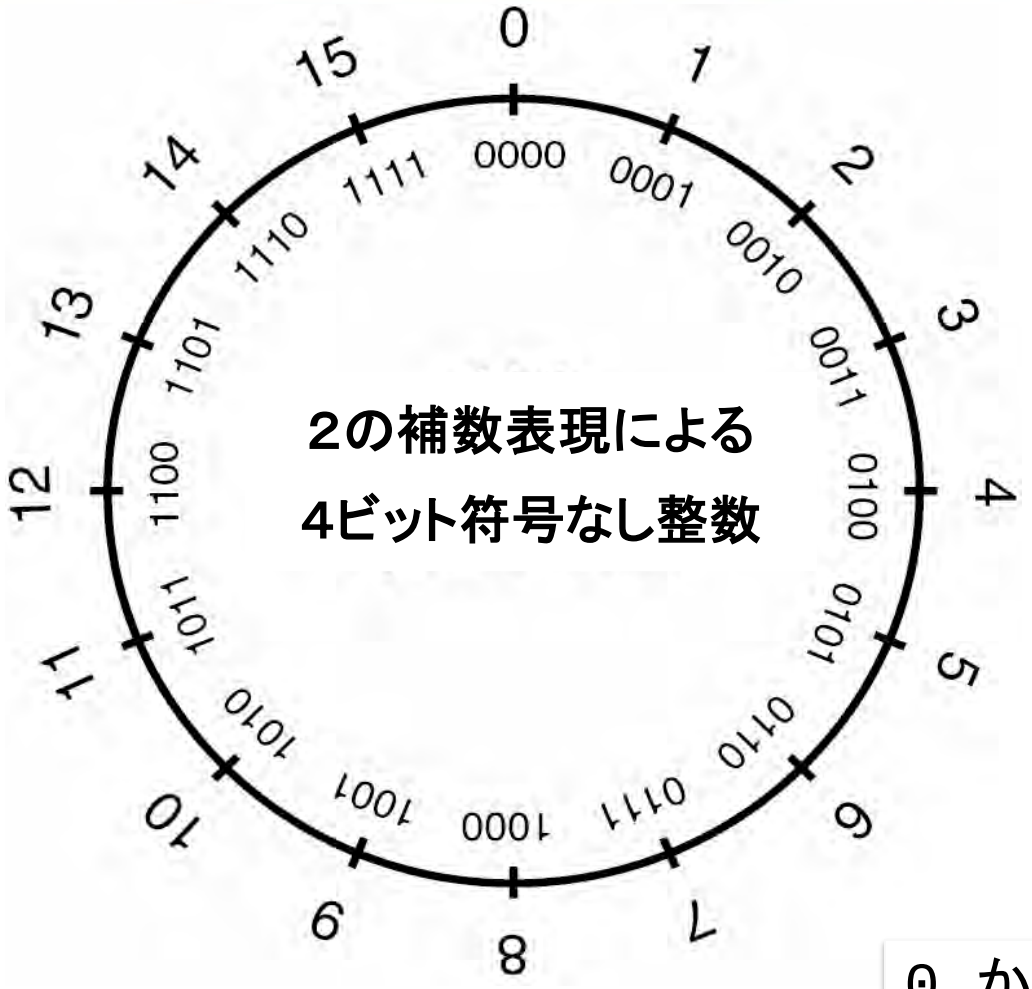
signed 型には、それに対応する **unsigned** 型がある。

符号付き整数表現



-2^{n-1} から $2^{n-1}-1$ までの値をとる（2の補数）

符号なし整数表現



0 から 2^n-1の値を表現

標準整数型

標準整数型

- signed char
- short int
- int
- long int
- long long int

注:**long long int** 型は

- ISO/IEC 9899:1999 (C) で定義
- ISO/IEC 14882:2003 (C++) では定義されていない
- 2006-04-21 のワーキングドラフト と多くの C++ の実装で定義

その他の C99 標準整数型

特別な用途で使用する型:

- **ptrdiff_t**:2つのポインタの差を表す符号付き整数型
- **size_t**:**sizeof** 演算子の結果を保持するための符号なし整数型
- **wchar_t**:サポートしているロケール中で最大の拡張文字集合のすべての要素に対し、一意なコードを表現できる範囲の値を持つ整数型

プラットフォーム依存の整数型

コンパイラベンダーがプラットフォーム依存の整数型を定義することがある。

Microsoft Windows API における定義:

- __int8、__int16、__int32、__int64
- ATOM
- BOOLEAN、BOOL
- BYTE
- CHAR
- DWORD、DWORDLONG、DWORD32、DWORD64
- WORD
- INT、INT32、INT64
- LONG、LONGLONG、LONG32、LONG64
- など

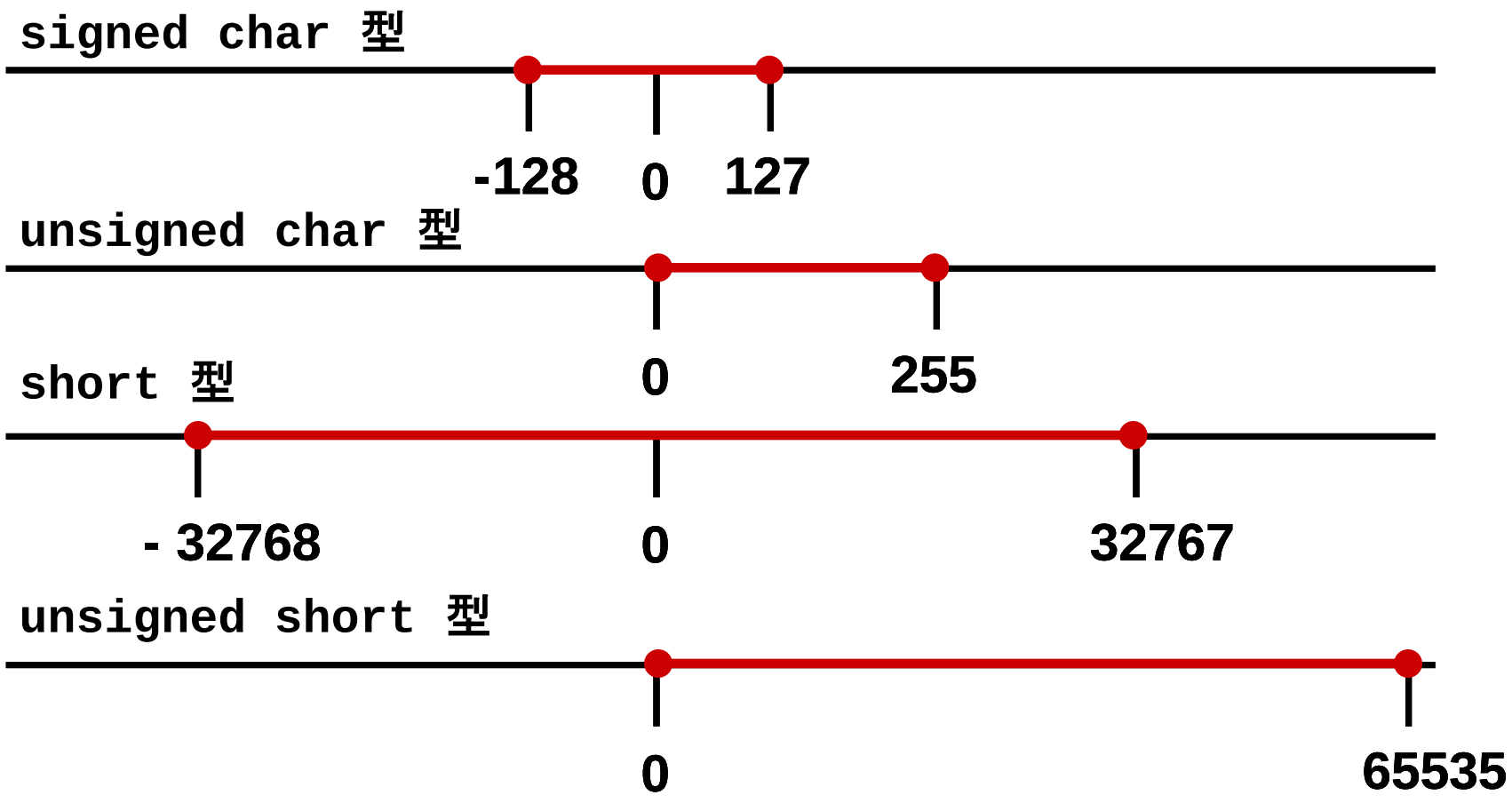
整数の範囲

整数型の最小値と最大値は、以下に依存する。

- 型の表現法
- 符号の有無 (signed か unsigned か)
- 割り当てられるビット数

C99 はこれらの範囲の最小限の必要条件を定める。

整数の範囲の例



データモデル

標準データ型に割り当てられたサイズを定義。

データモデルの名称は、XXXn というパターン。Xは型を、nはそのサイズ(一般的には32か64) を指す。

- ILP64: **int**, **long**, **pointer** 型が64 bit 幅
- LP32: **long**, **pointer** が32 bit 幅
- LLP64: **long long**, **pointer** が64 bit 幅

今日の(インテル) PC プロセッサ向け標準的なデータモデルは**ILP32**。

標準データモデル

データ型 / データモデル	LP32	ILP32	ILP64	LLP64	LP64
char	8	8	8	8	8
short	16	16	16	16	16
int	16	32	64	32	32
long	32	32	64	32	64
long long※1	N/A	N/A	64	64	64
ポインタ	32	32	64	64	64
プラットフォーム	Windows3.1	Windows 32bit	Alpha, Cray	Windows 64bit	Mac OS X, Sun, SGI

※1 long long はC99で新たに導入された型

もくじ

JPCERT CC®

第一部

整数のメカニズム

- 整数表現
- 型
- ⇒ 型変換
- 演算子
  - 加算/減算
  - 乗算
  - 除算
  - 左シフト、右シフト

第二部

整数のエラー条件と脆弱性

脅威の緩和方法

まとめ

左辺値(Lvalue) と 右辺値(Rvalue)

代入式 **E1=E2** において、左オペランド **E1** は 左辺値式 でなくてはならない。

- 左辺値 はオブジェクト を指し示す式
- オブジェクト とは記憶領域の一部

代入が有効であるためには、左オペランドがオブジェクトを指し示していなければならない(つまり 左辺値)

右辺値 は代入演算子の右側に出現することができるが、左側はだめ。

次に示すのは全て 右辺値

- 数値リテラル
- 文字リテラル
- 列挙型定数

右オペランドはどのような式であってもよい。

右辺値 は左辺には出現できないが、左辺値 は左辺、右辺のどちらに現れてもよい。

整数の型変換

JPCERT CC®

明示的(キャスト) あるいは 暗黙的に行われる

- 暗黙的な型変換が起きるのは、Cの仕様が混在する型に対する演算を許すから

データの欠損や誤った解釈につながる

C99 はコンパイラが従うべき変換規則を定める

- 整数の格上げ (integer promotion)
- 整数変換の順位 (integer conversion rank)
- 通常の算術型変換 (usual arithmetic conversion)

整数の格上げ※1

Integer Promotions

※1 JISでは「整数拡張」と呼ぶ

整数の格上げ

JPCERT CC®

int より小さい整数型〇〇〇〇と〇〇〇〇は
int 型に格上げされる

整数の格上げの例

```
char cresult, c1, c2, c3;

c1 = 100;

c2 = 90;

c3 = -120;

cresult = c1 + c2 + c3;
```

c1 と c2 の合計は signed char 型の最大サイズを超える。

しかし、c1、c2、c3 はそれぞれintに格上げされ、式全体は正しく評価される。

c1 の値が c2 の値に加算される。

合計は切り捨てられ、データを失うことなく cresult に格納される。

格上げの目的は、計算途中の値がオーバーフローして算術エラーが起こるのを防ぐことにある

整数の格上げ対象

- 通常の算術変換
- 引数の式
- 単項演算子(+、-、~)のオペランド
- シフト演算子の両方のオペランド

整数の格上げの意味

- int より小さな整数型の演算は、常に **signed int** 型か **unsigned int** 型に整数格上げされ、実際の演算はこの型で実行される。

注意

- 「ビットごとの補数演算子」(~) を (IA-32 の) **unsigned char** に適用すると、結果はかならず **signed int** 型の負の値になる。これは値が 32 ビットに 0 拡張されるため。

整数の格上げの結果一覧

JPCERT CC®

もとの型	格上げ後の型	格上げの根拠
unsigned char	int	格上げする。もとの型の「整数変換の順位」がint 型より低いから。
char	int	格上げする。もとの型の「整数変換の順位」がint 型より低いから。
short	int	格上げする。もとの型の「整数変換の順位」がint 型より低いから。
unsigned short	int	格上げする。もとの型の「整数変換の順位」がint 型より低いから。
unsigned int: 24	int	格上げする。unsigned int のビットフィールドだから。
unsigned int: 32	unsigned int	格上げする。unsigned int のビットフィールドだから。
int	int	格上げしない。もとの型と int 型の順位が同じだから。
unsigned int	unsigned int	格上げしない。もとの型と int 型の順位が同じだから。
long int	long int	格上げしない。もとの型の順位が int 型より高いから。
float	float	格上げしない。もとの型が整数型ではないから。
char *	char *	格上げしない。もとの型が整数型ではないから。

整数変換の順位

Integer Conversion Rank

整数変換の順位

すべての整数型は**整数変換の順位**をもつ

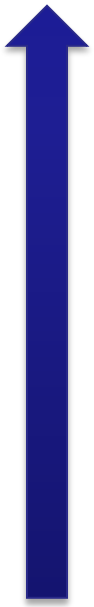
順位にもとづいて型変換を行う

整数変換の順位の定義

- 2つの符号付き整数型は、同じ表現を持つ場合であっても、同じ順位を持ってはならない
- 符号付き整数型の順位は、より精度の低い符号付き整数型の順位よりも高い
- 符号なし整数型の順位は、対応する符号付き整数型の順位と同じ

6.3.1.1 論理型，文字型及び整数型（JIS X 3010）

整数変換の順位と型



順位	型
高い	long long int, unsigned long long int
	long int, unsigned long int
	unsigned int, int
	unsigned short, short
	char, unsigned char, signed char
低い	_Bool

通常の算術型変換

Usual Arithmetic Conversions

通常の算術型変換

共通の型をもたらす仕組みを提供する一連のルール。

- 2項演算子の両方のオペランドを共通の型に合わせる
- 条件演算子 **(? :)** の第2、第3引数を共通の型に合わせる

2つのオペランドの型が異なると、共通の型に合わせる変換が生じる。

どちらか一方、もしくは両方のオペランドが変換される。

通常の算術型変換のルール

両方のオペランドが同じ型を持つ場合、さらなる型変換は行わない。

両方のオペランドが同じ整数型（符号付き、または符号なし）を持つ場合、整数変換の順位の低い方の型を、高い方の型に変換する。

符号なし整数型を持つオペランドが、他方のオペランドの整数変換の順位より高いまたは等しい順位を持つ場合、符号付き整数型を持つオペランドを、符号なし整数型を持つオペランドの型に変換する。

符号付き整数型を持つオペランドの型が、符号なし整数型を持つオペランドの型のすべての値を表現できるならば、符号なし整数型を持つオペランドを、符号付き整数型を持つオペランドの型に変換する。

それ以外の場合、両方のオペランドを符号付き整数型を持つオペランドの型に対応する符号なし整数型に変換する。

両方のオペランドが同じ型を持つ場合、さらなる型変換は行わない。

ex)

```
int + int ⇒ ok!

unsigned long int * unsigned long int
⇒ ok!
```

両方のオペランドが同じ整数型（符号付き、または符号なし）を持つ場合、整数変換の順位の低い方の型を、高い方の型に変換する。

ex)

順位が高い　　順位が低い

```
unsigned long + unsigned int
⇒ unsigned long + unsigned long
```

ルール3

符号なし整数型を持つオペランドが、他方のオペランドの整数変換の順位より高いまたは等しい順位を持つ場合、符号付き整数型を持つオペランドを、符号なし整数型を持つオペランドの型に変換する。

順位が等しい

符号なし整数型のオペランドの型に変換

ex)

```
unsigned int + int ⇒ unsigned int + unsigned int
```

符号付き整数型を持つオペランドの型が、符号なし整数型を持つオペランドの型のすべての値を表現できるならば、符号なし整数型を持つオペランドを、符号付き整数型を持つオペランドの型に変換する。

ex)

符号つきオペランドが

符号なし整数の全ての値を表現できる

```
    long long int + unsigned int
⇒ long long int + long long int
```

それ以外の場合(符号付き整数の型が符号なし整数より順位が高いが、符号なし整数の値をすべて符号付き整数で表現できない場合)、両方のオペランドを符号付き整数型を持つオペランドの型に対応する符号なし整数型に変換する。

longもintも同じビット幅の場合、long int で unsigned int のすべての値を表現できない

ex)

```
    unsigned int + long int
⇒ unsigned long int + unsigned long int
```

クイズ：通常の算術型変換

JPCERT CC®

左オペランド	右オペランド	変換後の共通の型
int	float	
unsigned int	int	
unsigned char	unsigned short	
unsigned int	long int	
unsigned int	long long int	
unsigned int	unsigned long long int	

符号なし整数の変換 1

小さい符号なし ⇒ **大きい**符号なし

- 常に安全
- 通常、値を0拡張

大きい符号なし ⇒ **小さい**符号なし

- 大きな値は切り捨てられる
- 下位ビットの値は保存される

符号なし整数の変換 2

符号なし ⇒ 対応する符号付き (ex. unsigned int ⇒ int)

- ビットパターンは保存、データは欠損しない
- 最高位ビットは符号ビットになる
- 符号ビットが立っていると、符号と絶対値の両方が変化

From unsigned	To	変換方法
char	char	ビット配列を保存し最上位ビットを符号ビットに
char	short	0 拡張する
char	long	0 拡張する
char	unsigned short	0 拡張する
char	unsigned long	0 拡張する
short	char	下位の１バイトを保存する
short	short	ビット配列を保存し最上位ビットを符号ビットに
short	long	0 拡張する
short	unsigned char	下位の１バイトを保存する
long	char	下位の１バイトを保存する
long	short	下位の１ワードを保存する
long	long	ビット配列を保存し最上位ビットを符号ビットに
long	unsigned char	下位の１バイトを保存する
long	unsigned short	下位の１ワードを保存する

データの欠損　データ解釈の誤り

符号付き整数の変換 1

値が負でない符号付き整数 ⇒ より大きな符号なし整数

- 値は変わらない
- 符号拡張される

符号付き整数 ⇒ より小さい符号付き整数

- 高位ビットが切り捨てられる

符号付き整数の変換 2

符号付き整数型 ⇒ 対応する符号なし整数型

- ビット配列は保存され、データは欠損しない
- 最高位ビットは、**符号ビット**としての機能を**失う**

符号付き整数が負の値でない ⇒ 値は変わらない。

値が負の場合、大きい符号なし整数の値として評価。

From signed	To	変換方法
char	short	符号拡張する
char	long	符号拡張する
char	unsigned char	ビット配列を保存し、高位ビットは符号ビットの機能を失う
char	unsigned short	short に符号拡張する。short を unsigned short に変換
char	unsigned long	long に符号拡張する。long を unsigned long に変換
short	char	下位の１バイトを保存する
short	long	符号拡張する
short	unsigned char	下位の１バイトを保存する
short	unsigned short	ビット配列を保存し、高位ビットは符号ビットの機能を失う
short	unsigned long	long に符号拡張する。long を unsigned long に変換
long	char	下位の１バイトを保存する
long	short	下位のワードを保存する
long	unsigned char	下位の１バイトを保存する
long	unsigned short	下位の１ワードを保存する
long	unsigned long	ビット配列を保存し、高位ビットは符号ビットとしての機能を失う

データの欠損　　データ解釈の誤り

符号付き整数の変換の例

```
unsigned int ui = UINT_MAX;
char c = -1;
if (c == ui) {
 printf("-1 = 4,294,967,295?¥n");
}
```

c の値と ui の値を比較

整数の格上げが行われ、c は値が符号無し整数の値 **0xFFFFFFFF** つまり 4,294,967,295 に変換される

型変換のまとめ

次の結果をもたらす変換は避ける。

- **値の欠損**：値の絶対値を表現できない型への変換
- **符号の欠損**：符号付きの型から符号なしの型へ変換し、符号が欠損する

安全が保証されている唯一の整数型変換は、符号の有無が同じで、より大きいサイズの型への変換のみ。

もくじ

第一部

整数のメカニズム

- 整数表現
- 型
- 型変換
- ⇒演算子
  - 加算/減算
  - 乗算
  - 除算
  - 左シフト、右シフト

第二部

整数のエラー条件と脆弱性

脅威の緩和方法

まとめ

整数演算

整数演算の結果、エラーと予期せぬ値が発生する可能性がある。

整数演算の多くは、オーバーフローにつながる可能性がある。

オーバーフローにつながる演算子

JPCERT CC®

Op	Overflow	Op	Overflow	Op	Overflow
+	✓	*=	✓	&	
-	✓	/=	✓	|	
*	✓	%=	✓	^	
/	✓	<<=	✓	~	
%	✓	>>=	✓	!	
++	✓	&=		un +	
--	✓	|=		un -	✓
=		^=		<	
+=	✓	<<	✓	>	
-=	✓	>>	✓	Etc.	

もくじ

JPCERT CC®

第一部

整数のメカニズム

- 整数表現
- 型
- 型変換
- 演算子
  - ⇒ 加算/減算
  - 乗算
  - 除算
  - 左シフト、右シフト

第二部

整数のエラー条件と脆弱性

脅威の緩和方法

まとめ

整数の加算

- 2つの算術演算のオペランドの合計を求めたり、ポインタに整数を加えたりする。
- 両方のオペランドが算術型であれば、通常の算術型変換が行われる。
- 整数の加算がオーバーフローを引き起こすのは、和が割り当てられたビット数で表現できない場合。

加算命令

IA-32 の **add** 命令:

add destination, source

第1オペランド (destination) に第2オペランド (source) を加算し、

- 結果を destination オペランドに格納
- destination オペランドにはレジスタかメモリ領域を指定できる
- source オペランドには即値、レジスタ、メモリ領域のいずれかを指定できる

符号付きと符号なしのオーバーフロー条件が検出・報告される。

JPCERT CC®

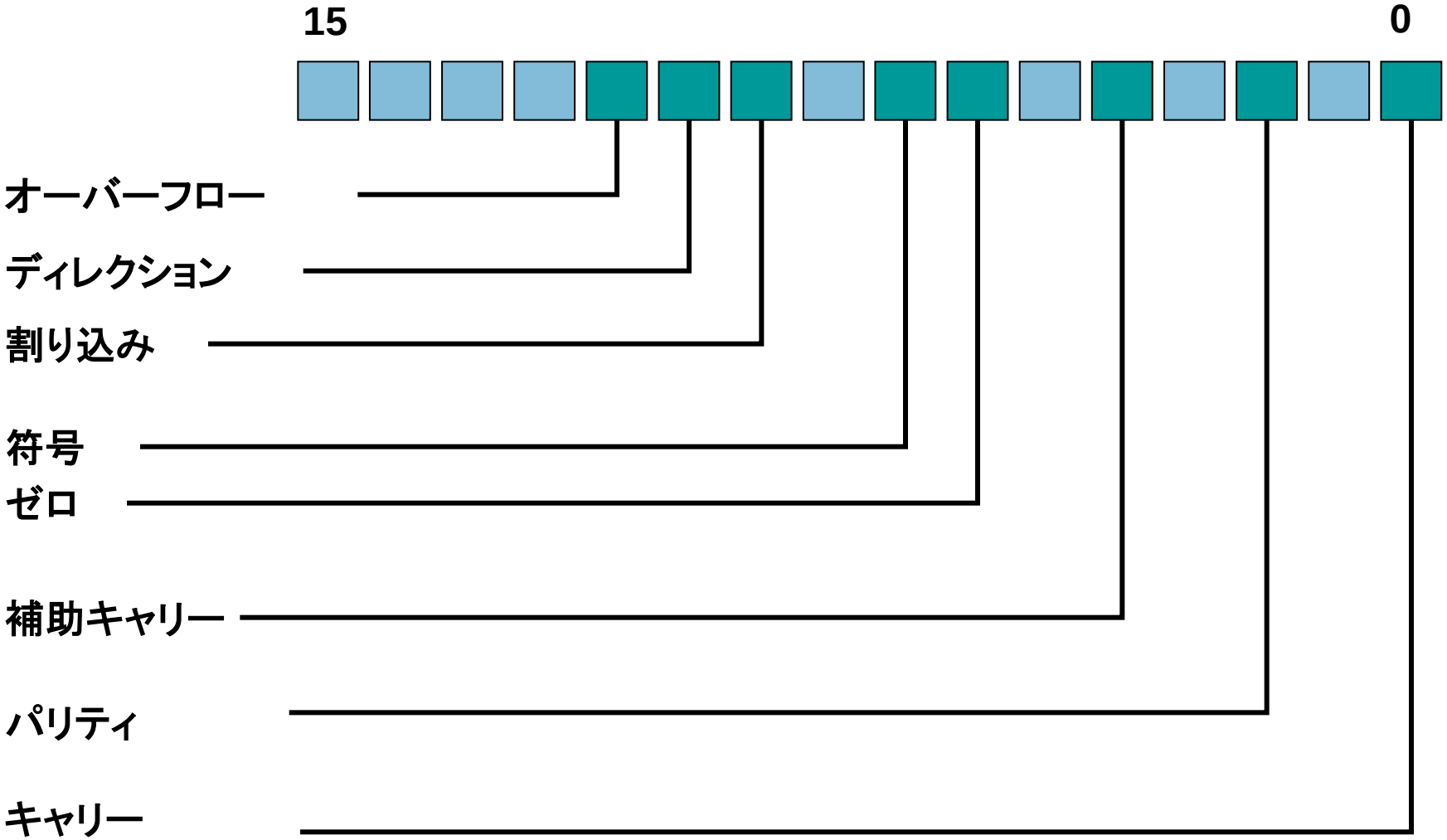
add eax, ebx

- 32ビットの ebx レジスタを32 ビットの eax レジスタに加算
- その合計を eax レジスタに格納

add 命令はフラグレジスタにフラグをセットする。

- オーバーフローフラグ：符号付きの算術オーバーフローを示す
- キャリーフラグ：符号なしの算術オーバーフローを示す

フラグレジスタのレイアウト



フラグの解釈

ハードウェアレベルでは、符号付き整数と符号なし整数の加算との間に違いはない。

オーバーフローフラグとキャリーフラグは、前後関係によって解釈されなくてはならない。

signed int と unsigned int の加算

signedの値もunsignedの値も同じ命令で加算される

```
si1 + si2
mov         eax, dword ptr [si1]
add         eax, dword ptr [si2]
ui1 + ui2
mov         eax, dword ptr [ui1]
add         eax, dword ptr [ui2]
```

注: アセンブリコードは Visual C++ で生成

符号なしオーバーフローの検出

キャリーフラグは符号なし算術オーバーフローを表す。

符号なしオーバーフローは次の命令を用いて検出できる。

- **jc** 命令(キャリーがあればジャンプ)
- **jnc** 命令(キャリーがなければジャンプ)

条件付きジャンプ命令は次の命令の後に配置される。

- 32 ビットでは **add** 命令
- 64 ビットでは **adc** 命令

符号付きオーバーフローの検出

JPCERT CC®

オーバーフローフラグは、 符号付き算術オーバーフローを表す。

符号付きオーバーフローは次の命令を用いて検出できる。

- **jo** 命令(オーバーフローがあればジャンプ)
- **jno** 命令(オーバーフローがなければジャンプ)

条件付きジャンプ命令は次の命令の後に配置される。

- 32 ビットでは **add** 命令
- 64 ビットでは **adc** 命令

整数の減算

IA-32 の減算命令:

- **sub**（減算）
- **sbb**（ボロー付き減算）

  オーバーフローフラグやキャリーフラグをセットし、符号付きあるいは符号なし演算におけるオーバーフローを示す。

もくじ

JPCERT CC®

第一部

整数のメカニズム

- 整数表現
- 型
- 型変換
- 演算子
  - 加算/減算
  - ⇒ 乗算
  - 除算
  - 左シフト、右シフト

第二部

整数のエラー条件と脆弱性

脅威の緩和方法

まとめ

特徴

- 乗算はオーバーフローを起こしやすい。
- 比較的小さいオペランドを乗算した場合でも、オーバーフローする可能性がある。

乗算によるオーバーフローを防ぐには？

2つのオペランドのうち大きい方の2倍のサイズを、結果を格納する領域に割当てる。

符号なし整数の最大値は、$2^n - 1$

$$(2^n-1) \times (2^n-1) = 2^{2n} - 2^{n+1} + 1 < 2^{2n}$$

符号付き整数の最小値は -2^{n-1}

$$-2^{n-1} \times -2^{n-1} = 2^{2n-2} < 2^{2n}$$

乗算命令

IA-32 の乗算命令:

- mul (符号なし乗算) 命令
- imul (符号付き乗算) 命令

mul 命令は、

- 第1(destination)オペランドと第2(source)オペランドを符号なしで乗算し、
- 結果を destination オペランドに格納する。

符号付き･符号なし int 型の乗算

アセンブリレベルでは

```
si_product = si1 * si2;
ui_product = ui1 * ui2;

mov  eax, dword ptr [ui1]
imul eax, dword ptr [ui2]
mov  dword ptr [ui_product], eax
```

アップキャスト

両方のオペランドを、少なくとも2倍のビット数を持つ整数にキャストしてから乗算する。

符号なし整数の場合、

- 一段階大きい整数の上位ビットを確認する。
- 上位ビットが １ つでもセットされていればエラーを発行する。

符号付き整数の場合、上位ビットの全ビットと下位ビットの符号ビットがすべて0あるいはすべて1であれば、オーバーフローは発生していない。

アップキャストの例

```
void* AllocBlocks(size_t cBlocks) {
  // ブロックを割り当てていない場合はエラー
  if (cBlocks == 0) return NULL;
  // 十分なメモリを割り当てる
  // 結果を 64 ビットの整数にアップキャストし、
  // 32 ビットの UINT_MAX と照合して、
  // オーバーフローが発生していないことを確認する
  unsigned long long alloc = cBlocks * 16;
  return (alloc < UINT_MAX)
    ? malloc(cBlocks * 16)
    : NULL;
}
```

乗算の結果は 32 ビットの値となる。結果は unsigned long long に代入されるが、この演算はすでにオーバーフローを起している可能性がある。

C99に準拠するためには32ビットで演算

C99に準拠するためには、このコンテキストにおける32ビット同士の数の乗算の結果は、32ビットでなければならない。

- 精度拡張の乗算命令のないアーキテクチャでは負担が大きくなるため、言語規格の修正は行われなかった。

オペランドのどれか1つをキャストすれば、正しい結果が得られる。

修正したアップキャストの例

JPCERT CC®

```
void* AllocBlocks(size_t cBlocks) {

 // ブロックを割り当てていない場合はエラー
  if (cBlocks == 0) return NULL;

  // 十分なメモリを割り当てる
  // 結果を 64 ビットの整数にアップキャストし、
  // 32 ビットの UINT_MAX と照合して、
  // オーバーフローが発生していないことを確認する
  unsigned long long alloc;

  alloc = (unsigned long long)cBlocks * 16;
  return (alloc < UINT_MAX)

    ? malloc(cBlocks * 16)
    : NULL;
}
```

もくじ

JPCERT CC®

第一部

整数のメカニズム

- 整数表現
- 型
- 型変換
- 演算子
  - 加算/減算
  - 乗算
  - ⇒ 除算
  - 左シフト、右シフト

第二部

整数のエラー条件と脆弱性

脅威の緩和方法

まとめ

整数の除算

32ビット(64ビットも)の整数の最小値を-1で割ると、整数オーバーフロー条件が発生する。

- 2,147,483,648 / -1 = 2,147,483,648

– 2,147,483,648は符号付き32ビット整数として表現できないので、演算結果の値は不正。

エラー検出

IA-32 の **div**, **idiv** 命令

div 命令:

- **ax**, **dx:ax** あるいは **edx:eax** レジスタ(被除数)の(**unsigned**)整数の値を、source オペランド(除数)で割り
- 結果を **ax** (**ah:al**)、**dx:ax**、あるいは **edx:eax** レジスタに格納する

idiv 命令:

- **signed** 値に対して **div** と同じ処理を実行

signed 整数の除算

```
si_quotient = si_dividend / si_divisor;
mov  eax, dword ptr [si_dividend]
cdq
idiv eax, dword ptr [si_divisor]
mov  dword ptr [si_quotient], eax
```

cdq 命令は、**eax** レジスタの値の符号（ビット 31）を、**edx** レジスタの値のすべてのビット位置にコピーする。

注：Visual C++ によって生成されたアセンブリコード

unsigned 整数の除算

```
ui_quotient = ui1_dividend / ui_divisor;

mov eax, dword ptr [ui_dividend]

xor edx, edx

div eax, dword ptr [ui_divisor]

mov dword ptr [ui_quotient], eax
```

注: Visual C++ によって生成されたアセンブリコード

エラー検出

Intel の **div** と **idiv** はオーバーフローフラグをセットしない。

次の場合には除算エラーが発行される:

- source オペランド(除数)が0
- 商が格納用レジスタよりも大きい

除算エラーは、割り込みベクタ0のフォルトを発生させる。

フォルトが通知されると、プロセッサはハードウェアの状態を、フォルトの命令の実行される前の状態に復元する。

プログラム実行に割り込みをかけるプロセッサのメカニズム

割り込み(interrupt)

非同期なイベント。I/Oデバイスがトリガとなる。

例外(exception)

同期イベント。instructionを実行中に発生。

IA-32の場合、fault, trap, abort の3種類がある。

割り込み, 例外発生時の動作

どちらも基本、おなじように処理される

発生すると、プロセッサは現在実行しているプログラムを中断(halt)、その割り込み・例外を処理するためのハンドラプロシージャに処理を移す。

- そのときプロセッサは、interrupt descriptor table (IDT)のエントリーをみて、どのハンドラプロシージャを呼ぶか判断
- IDTには vector (ベクタ)と呼ばれる番号がふられている
- Vector 0は、DIVやIDIV命令で発生したDivide Error(除算エラー)に対応する

Microsoft Visual Studio

C++ 言語の例外処理では、次の状況を復旧できない。

- ハードウェアの例外
- 次のようなフォルト
  - アクセス違反
  - 0除算

Visual Studio は次の仕組みを提供する。

- ハードウェア等の例外を扱うための構造化例外処理 (SEH: structured exception handling)
- Win32 構造化例外の処理を可能にする C 言語のための一連の拡張機能

構造化例外処理は、オペレーティングシステムが提供する仕組みで、C++言語における例外処理とは異なるもの。

C 言語における構造化例外処理

JPCERT CC®

```
int x, y;
__try {
  x = INT_MIN;
  y = -1;
  x = x / y;
}
__except (GetExceptionCode() ==
          EXCEPTION_INT_OVERFLOW ?
          EXCEPTION_EXECUTE_HANDLER :
          EXCEPTION_CONTINUE_SEARCH) {
  printf("Integer overflow during division.¥n");
}
```

C++ 言語の例外処理

```
int operator /(unsigned int divisor) {
    try {
        return ui / divisor;
    }
    catch (...) {
        throw SintException(ARITHMETIC_OVERFLOW);
    }
}
```

Visual C++ で書かれた C++ の例外処理は、構造化例外処理を使って実装されているため、このプラットフォームで C++ の 例外処理を使用できる。

Linux のエラー処理 1

Linux 環境においては、除算エラーなどのハードウェア例外はシグナルによって処理される。

source オペランド（除数）が0であるか、商が格納用レジスタよりも大きい場合には、**SIGFPE**（浮動小数点例外）が生成される。

プログラムの異常終了を回避するために、次のようにシグナルハンドラを設定することができる。

```
signal(SIGFPE, signal_handler);
```

JPCERT CC®

signal() 関数は 2 つの引数を取る

- シグナル番号
- シグナルハンドラのアドレス

フォルトを発生する命令が戻りアドレスに設定されるため、シグナルハンドラを単純に終了すると、その命令とシグナルハンドラが交互に呼び出され、無限ループに陥ってしまう。

もくじ

第一部

整数のメカニズム

- 整数表現
- 型
- 型変換
- 演算子
  - 加算/減算
  - 乗算
  - 除算
  - ⇒ 左シフト、右シフト

第二部

整数のエラー条件と脆弱性

脅威の緩和方法

まとめ

ビット単位のシフト

左シフト演算:

シフト式 << 加算式

右シフト演算:

シフト式 >> 加算式

整数の格上げがオペランドに対して行われる。

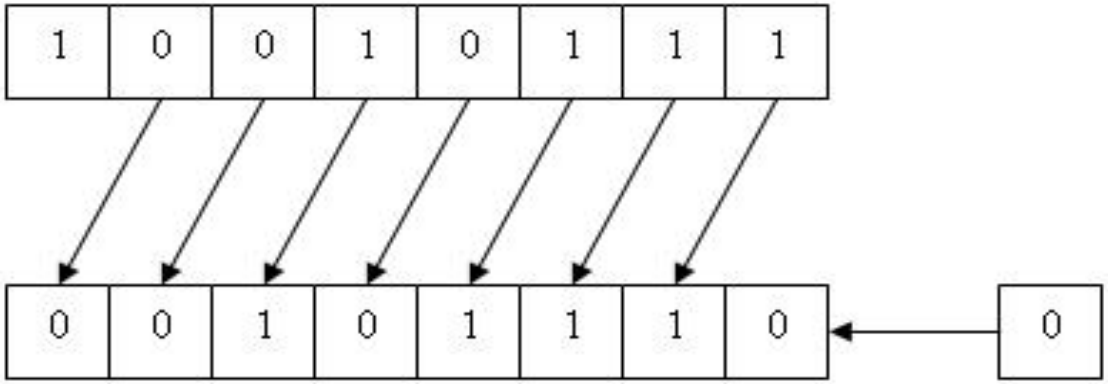
結果の型は、左オペランドを格上げした後の型。

右オペランドの値 が以下の場合、動作は未定義:

- 負の数
- 格上げした左オペランドの幅以上

左シフト

E1 << E2 の結果は、**E1** を **E2** ビット分左にシフトした値とする。空いたビットにはゼロを詰める。



E1 が **signed** 整数型かつ非負の値をもち、**E1 * 2^{E2}** が結果の型で表現できる場合、それが結果の値となる。それ以外の場合、動作は未定義。

符号付き左シフトを安全に行うには

```
signed int の左シフト
int si1, si2, sresult;
if ( (si1 < 0) || (si2 < 0) ||
     (si2 >= sizeof(int)*CHAR_BIT) ||
      si1 > (INT_MAX >> si2) ) {
  /* エラー */
}
else {
  sresult = si1 << si2;
}
```

左右のオペランドは非負の値であること

シフトは左オペランドのビット数より小さいこと

結果が表現可能かチェック

C99において、**CHAR_BIT** マクロは ビットフィールドでない最小のオブジェクトのビット数を定義する。

符号なし左シフト

E1 が式 **E1 << E2** において **unsigned** 型 を持つならば、結果の値は、**E1 * 2^{E2}** の、結果の型で表現可能な最大値より1 大きい値を法とする剰余とする。

- C99 では **unsigned** 整数に対して modulo される
- **unsigned** 整数がラップアラウンドすることで、予期せぬ値が生まれ、セキュリティ上の脆弱性につながることが多い

符号なし左シフトを安全に行うには

```
unsigned int ui1, ui2, uresult;
unsigned int mod1, mod2;
// wrap する場合
if ( (ui2 >= sizeof(unsigned int)*CHAR_BIT) ||
     (ui1 > (UINT_MAX >> ui2))) ) {
  /* エラー条件をハンドル */
}
else uresult = ui1 << ui2;
// modulo する場合
if (mod2 >= sizeof(unsigned int)*CHAR_BIT) {
  /* エラー条件をハンドル */
}
else uresult = mod1 << mod2;
```

シフトは左オペランドのビット数より小さいこと

結果の値が表現可能かどうかチェック

ラップアラウンドが許される modulo の場合

右シフト

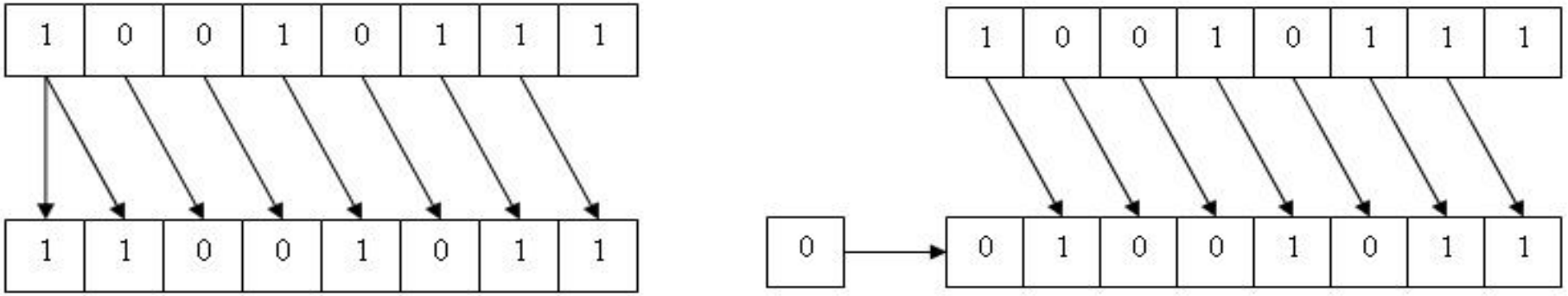
E1 >> E2 の結果は **E1** を **E2** ビット分右にシフトした値とする。

E1 が **unsigned** 型を持つ場合、又は **E1** が **signed** 型でかつゼロ以上の値をもつ場合、結果の値は、$E1/2^{E2}$ の商の整数部分とする。

符号付き右シフト

E1 が **signed** 型でありかつ負の値である場合、シフト後の値は処理系定義であり、下図のいずれかとなる

- 算術 (**signed**) シフト
- 論理 (**unsigned**)シフト



「C89 標準化委員会は、符号付き右シフト演算に符号の拡張を要求しないという点において、K&Rが認めた『処理系に自由度を持たせる』という考えを支持した。その理由は、符号の拡張を要求することで、高速なコードのスピードが低下する可能性があり、また符号を拡張するシフトを採用することの有用性が小さいから」

[C99 Rationale]

右シフトの脆弱性

右シフトが算術シフトである処理系では、符号ビットが伝播(propagate)する。

```
int stringify = 0x80000000;
char buf[sizeof("256")];
sprintf(buf, "%u", stringify >> 24);
```

stringify >> 24 の結果は **0xFFFFFF80** になる

sprintf() 呼び出しで バッファオーバーフローが発生

ビットを抽出したいのならば、

((stringify >> 24) & 0xff)

右シフトを安全に行うには

```
int si1, si2, sresult;
unsigned int ui1, ui2, result;
if ( (si2 < 0) ||
     (si2 >= sizeof(int)*CHAR_BIT) ) {
  /* エラー */
}
else sresult = si1 >> si2;
if (ui2 >= sizeof(unsigned int)*CHAR_BIT) {
  /* エラー */
}
else uresult = ui1 >> ui2;
```

右オペランドは非負であることをチェック

シフトは左オペランドのビット数よりも小さいこと

unsignedについても同様

もくじ

JPCERT CC®

第一部

整数のメカニズム

- 整数表現
- 型
- 型変換
- 演算子
  - 加算/減算
  - 乗算
  - 除算
  - 左シフト、右シフト

第二部

⇒整数のエラー条件と脆弱性

- 脅威の緩和方法
- まとめ

JPCERT CC®

セキュリティポリシーの明示的あるいは暗黙的な違反を許してしまう、一連の状態。

- ハードウェアレベルの整数エラー条件や、整数を伴うロジック上の誤りなどが原因
- これらが他の条件と組み合わされると、脆弱性が発生

整数の脆弱性の種類

JPCERT CC®

1. 整数オーバーフロー
   - 符号付き整数オーバーフロー *(signed integer overflow)*
   - 符号無し整数のラップアラウンド *(unsigned integer wrapping)*
2. 符号エラー *(signedness error)*
3. 切り捨て *(truncation)*
4. 非例外的(単純なロジック上のエラー)
5. 「未定義の動作」としてコンパイラに最適化されてしまう問題

それではひとつずつ問題を見ていきましょう

整数オーバーフロー

(integer overflow)

整数オーバーフローとは

整数オーバーフロー：値がその型の最大値を超えて増加する、あるいは最小値を超えて減少する時に発生。

- 符号の有無に関係なく起こる。
- 実行時に起こる(ランタイムエラー)。
- 最小値を超えて減少することをアンダーフロー(underflow)と呼ぶこともある。

符号付きのオーバーフローは、値を符号ビットに繰り越す時に発生する。

符号なしオーバーフローは、 内部表現がもはやその値を表現できなくなった時に発生する。

オーバーフローの例 1

JPCERT CC®

```
int i;
unsigned int j;
i = INT_MAX;  // 2,147,483,647
i++;
printf("i = %d¥n", i);
j = UINT_MAX; // 4,294,967,295;
j++;
printf("j = %u¥n", j);
```

i = -2,147,483,648

j = 0

オーバーフローの例 2

```
int i = INT_MIN; // -2,147,483,648;
i--;
printf("i = %d¥n", i);

unsigned int j = 0;
j--;
printf("j = %u¥n", j);
```

i = 2,147,483,647

j = 4,294,967,295

GOCR PNM イメージサイズの整数オーバーフロー脆弱性

```
struct{
  unsigned char *p;
  int x;  /* xsize */
  int y;  /* ysize */
  int bpp;
}
typedef struct pixmap pix;
…
void readpgm(char *name, pix *p) {
  /* read pgm */…
  png_readpaminit(fp, &inpam);
  p->x=inpam.width;
  p->y=inpam.height;
  if(!(p->=(char *)malloc(p->x*p->y)))
    F1("Error at malloc");
  for(i=0; i<inpam.height; i++{
    pnm_readpamrow(&inpam, tuplerow);
    for(j=0; j<pnam.width; j++)
      p->p[i*inpam.width+j]=sample;
  }
}
```

符号付き整数オーバーフローの実例

攻撃シナリオ

- 攻撃者は inpam.width と inpam.height に大きな整数値を与える
- malloc()の引数がオーバーフローし、誤って小さなメモリ領域が確保される
- p->pで境界外書き込みが発生し、任意のコード実行につながる

JPEG 処理ルーチンの脆弱性1

JPEGのフォーマット

コメントフィールドには、コメントフィールドの長さを示す 2 バイトの領域 **Length** がある。

Lengthの値には**Length**フィールド自身の2バイト分も含まれる。

したがって、**Data** の長さは Length – 2byte した値になる。

JPEG 処理ルーチンの脆弱性$_2$

問題の関数は、メモリ確保のため、コメント文字列のみの長さを求めるため
JPEGファイルのコメントフィールドから Length の値を読み取り、

終端の NULL バイトの1バイトを加えたメモリ領域を確保する。

JPEG処理ルーチンの脆弱なコード

```
void getComment(unsigned int len, char *src) {
  unsigned int size;
  size = len - 2;
  char *comment = (char *)malloc(size + 1);
  memcpy(comment, src, size);
  return;
}

int main(void) {
  getComment(1, "Comment ");
  return 0;
}
```

0 バイトの malloc() が成功する。

サイズが大きな正の値
(0xffffffff) として解釈される。

コメント長フィールドの値として 1 を持つ画像を作成することで、オーバーフローを引き起こす可能性がある。

符号無し整数のラップアラウンドの実例

メモリ割り当てと整数オーバーフロー

calloc() やその他のメモリ割り当て関数は、メモリ領域の大きさを計算する際、整数オーバーフローを起こす可能性がある。

要求したサイズよりも小さいバッファが返されると、バッファオーバーフローにつながる可能性がある。

次のようなコードは、脆弱性を引き起こす可能性あり ：

- C: `p = calloc(sizeof(element_t), count);`
- C++: `p = new ElementType[count];`

calloc() ライブラリ関数は2つの引数をとる

- 要素型の格納サイズ
- 要素数

C++ の **new** 演算子を使う場合、要素型のサイズは明示的には指定しない。

必要なメモリ領域を計算するために、格納サイズと要素数の積を求める。

オーバーフロー条件

乗算の結果が符号付き整数で表現できない値になると、割り当て関数は成功したように見えるが、実は必要とするよりも少さな領域しか確保できていない。

プログラムは、割り当てられたバッファの終端を越えて書き込みを行い、ヒープオーバーフローが発生する可能性がある。

符号エラー

(signedness error)

次の場合に発生する

符号付き整数を符号無しとして解釈

符号無し整数を符号付きとして解釈

2の補数表現において、最上位ビット(MSB)が符号の意味を持ったり、失ったりする

IA-32アーキテクチャでは、-1 と $2^{32}-1$が誤解釈されうる

符号エラーの例

```
int i = -3;
unsigned short u;
u = i;
printf("u = %hu¥n", u);
```

より小さい符号なし整数への暗黙的な変換

値を表現できる十分なビット数があるため切り捨てエラーは生じない。　ただし 2 の補数表現は大きな符号付きの値と解釈される。このため、u = 65533となる。

符号エラーの例

```
signed char cresult;
c1 = 100;
c2 = 90;
cresult = c1 + c2;
```

c1 と c2 を加算すると、signed char の最大値である(+127)を超えてしまう

cresult = -66
結果として得られるビットパターンは2の補数の8ビットとして表現しうるため、データのロスは発生せず、結果の和をリカバーすることができる。

int より小さな型の整数は、演算が行われる前に格上げされ、**int** 型もしくは **unsigned int** 型になる。

Linux kernel XDR における符号エラー

JPCERT CC®

```
static inline u32 *
decode_fh(u32 *p, struct svc_fh *fhp) {
    int size;
    fh_init(fhp, NFS3_FHSIZE);
    size = ntohl(*p++);
    if (size > NFS3_FHSIZE)
        return NULL;
    memcpy(&fhp->fh_handle.fh_base, p, size);
    fhp->fh_handle.fh_size = size;
    return p + XDR_QUADLEN(size);
}
```

*p は攻撃者が制御できる XDR データに由来する

sizeの値が負の時、上限値チェックをスルーしてしまう

sizeがunsignedの非常に大きな値として評価される

符号エラーが脆弱性につながった実例

符号エラーの対策

負の長さは巨大な正の整数として解釈され、バッファオーバーフローにつながることが多い。

これは、整数 **`size`** が有効な値を持つように制限できれば回避できる:

- 範囲確認をより厳密にし、**`size`** が`0`より大きくかつ **`NFS3_FHSIZE`** より小さいことを保証する
- **`size`** を **`unsigned`** として宣言する

  `memcpy()` 呼び出し時に、符号付き型から符号なし型への変換が起こらないようにする

  そうすれば符号エラーも発生しない

切り捨て

(truncation)

切り捨てエラー (truncation error)

JPCERT CC®

切り捨てエラーが発生するのは、

- ある整数をそれより小さい整数型に変換し
- 元の整数の値が小さい型の範囲に収まらない場合

元の値の下位ビットの値は保存されるが、上位ビットの値は消失する。

切り捨てエラーの例

```
unsigned char cresult, c1, c2;
c1 = 200;
c2 = 90;
cresult = c1 + c2;
```

c1とc2を加算するとunsigned char の最大サイズ(+255)を超える

cresult = 34
十分なサイズをもたない型に結果の値を代入する場合、切捨てエラーが発生する

int よりサイズの小さい整数は、演算前にint またはunsigned int に格上げされる

脆弱な実装

```
bool func(char *name, long cbBuf) {
  unsigned short bufSize = cbBuf;
  char *buf = (char *)malloc(bufSize);
  if (buf) {
    memcpy(buf, name, cbBuf);
    if (buf) free(buf);
    return true;
  }
  return false;
}
```

cbBuf は **bufSize** を初期化するために使われている。また **bufSize** は、**buf** のためのメモリを確保するために使われる

long 型として宣言された **cbBuf** が **memcpy()**関数に長さとして渡される

脆弱性 1

cbBuf は **unsigned short bufSize** に一時的に格納される。

IA-32 上では、GCC でも Visual C++ コンパイラでも、**unsigned short** の最大値は 65,535 である。

同じプラットフォームにおける **signed long** の最大値は 2,147,483,647 である。

したがって 2行目の代入では、65,535 から 2,147,483,647 の間の **cbBuf** の値すべてについて切り捨てエラーが発生する。

脆弱性 2

bufSize が **malloc()** と **memcpy()** の呼び出しのために使われていただけならば、単なるエラーで脆弱性ではない。

bufSize をバッファを確保するために使い、かつ **cbBuf** を **memcpy()** に対する呼び出し時のサイズとして使用すると、1 バイトから 2,147,418,112 (2,147,483,647 - 65,535) バイトの範囲で、**buf** をオーバーフローさせる可能性がある。

非例外的(単純なロジック上のエラー)

負のインデックス値

```
int *table = NULL;
int insert_in_table(int pos, int value){
  if (!table) {
    table = (int *)malloc(sizeof(int) * 100);
  }
  if (pos > 99) {
    return -1;
  }
  table[pos] = value;
  return 0;
}
```

ヒープ領域から配列の記憶領域を確保

pos は99を超えない

value を配列中の指定された場所に格納

非例外的な整数エラー

JPCERT CC®

pos の範囲検査が正しくない

- **pos** 変数は符号付き整数として宣言されており、関数に渡される値は正と負のどちらもありうる。
- 上限を超える正の値は **if(pos > 99)**で捕捉できるが、負の値は検査をすり抜けてしまう。

例外条件（オーバーフローなど）が発生しなくても、整数関連のエラーが起こる可能性がある

JPCERT CC®

「未定義の動作」としてコンパイラに最適化されてしまう問題

```
#include <assert.h>
#include <limits.h>

int foo(int a) {
    assert(a + 100 > 1);
    printf("%d %d¥n", a +
   100, a);
    return a;
}
int main(void) {
    foo(100);
    foo(INT_MAX);
}
```

- 整数オーバーフローは「未定義の動作」
- 処理系は未定義の動作をどう扱ってもよい
  - 完全に無視する
  - 予測不能な値を返す
  - あらかじめ規定された方法で処理する
  - 実行を中断する(診断メッセージを出力)
- コンパイラは未定義の動作に対応するコードを生成しなくてもよく、最適化されてしまうことがある
- GCC4.2.1で-O2以上でコンパイルするとassert文が最適化により削除される

もくじ

JPCERT CC®

第一部

整数のメカニズム

- 整数表現
- 型
- 型変換
- 演算子
  - 加算/減算
  - 乗算
  - 除算
  - 左シフト、右シフト

第二部

整数のエラー条件と脆弱性

⇒脅威の緩和方法

まとめ

型の範囲検査

型の範囲検査

型の値域検査を行うことで整数の脆弱性の多くを取り除くことができる。

Pascal や Ada のような言語では、すべてのスカラ型に対して、とり得る値の範囲を制限したサブタイプを定義することができる。

Ada では、基となる型に **range** キーワードを用いることで、範囲制限を行うことができる。

```
type day is new INTEGER range 1..31;
```

範囲制限は、ランタイムで実施される。

C/C++ にはこれに相当するメカニズムはない。

型の範囲検査の例

```
#define BUFF_SIZE 10
int main(int argc, char* argv[]){
  unsigned int len;
  char buf[BUFF_SIZE];
  len = atoi(argv[1]);
  if ((0<len) && (len<BUFF_SIZE) ){
    memcpy(buf, argv[2], len);
  }
  else
    printf("Too much data¥n");
}
```

符号なし整数として宣言し、暗黙的な型検査

上限と下限の両方に対する明示的な検査

範囲検査の説明

len を **unsigned** として宣言するだけでは、値が **0 から UINT_MAX** の間にあることしか保証しないので範囲制限として不十分。

上限と下限の境界を両方とも検査することで、範囲外の値が **memcpy()** に渡されないことを保証する。

暗黙的な検査と明示的な検査の両方を実施するのは冗長かもしれないが、よりセキュアなコードにするためにはどちらも実施したほうが良い。

整数の入力値検査、範囲検査

外部からの入力値について、上限と下限が定義可能かどうかを評価する。

- インタフェースでその制限を強制
- 内部エラーが発生してから不正な入力値を探し出すより、入力時に問題を発見・修正する方が簡単

過度に大きいあるいは小さい整数の入力を制限。

コーディング規約を定めて以下を徹底する。

- 定数と変数を区別する
- 外部からの影響を受ける変数と、厳密に範囲制限された内部変数とを区別する

入力値検査はセキュアコーディングで最も重要な概念

強い型付け

強い型付け

より良い型検査を実現するためには、型自体が優れたものでなければならない。

符号なしの型を使用することで、変数が負の値を含まないことは保証できる。

- オーバーフローは防ぎきれない

範囲に関する問題をコンパイラによって効果的に検出するためにも、強い型付けを利用すべき。

問題: オブジェクトサイズの表現

悪い例 :

```
short total = strlen(argv[1])+ 1;
```

良い例 :

```
size_t total = strlen(argv[1])+ 1;
```

さらに良い例 :

```
rsize_t total = strlen(argv[1])+ 1;
```

size_t で注意すること

負の数は、**size_t** のような符号なしの型に変換されると、非常に大きな正の数として表現される。

極端に大きなオブジェクトサイズは、オブジェクトのサイズが間違って計算されたことを示す場合が多い。

rsize_t

rsize_t は **RSIZE_MAX** より大きくならない。

アドレス空間の大きな計算機を対象とするアプリケーションでは、
RSIZE_MAX を次のいずれか小さい方の値として定義する:

- サポートされている最大オブジェクトのサイズ
- **(SIZE_MAX >> 1)** （この上限が、正当な、しかし非常に大きなオブジェクトのサイズよりも小さい場合であっても）

rsize_t は **size_t** と同じ型、バイナリレベルで互換性がある

強い型付けの例

水の温度を(華氏)で格納する整数を次のように宣言:

```
unsigned char waterTemperature;
```

waterTemperatureは0から255までの値を表現する符号なし8ビット値。

unsigned char は、

- 水(液体)の温度を表現するには合理的。水温は、華氏32度(氷点)から華氏212度(沸点)までの範囲に収まる
- オーバーフローは防止できない
- 無効な値(1～31 と 213～255)を許可してしまう

抽象データ型

解決法のひとつ:

- 抽象データ型を作り、`waterTemperature` をプライベート変数にして、ユーザが直接アクセスできないようにする
- この抽象データ型のユーザは、パブリックメソッドの呼び出しを通じてのみ、値のアクセスや更新、演算を行える
- これらのメソッドは、`waterTemperature` の値が有効な範囲を逸脱しないことを保証することで、型安全性を提供しないとダメ

適切に実装されれば、整数型の値域エラーは発生しない。

コンパイラによる検査

Visual C++ コンパイラによる検査

Visual C++ .NET 2003 のコンパイラは、整数の値をそれよりも小さい整数型に代入しようとすると警告(C4244)を発する。

- 警告レベル1では、__int64 型の値をunsigned int 型の値に代入しようとすると警告が発行する。
- 警告レベル3もしくは4では、整数の型がそれよりも小さい整数の型に変換しようとすると「possible loss of data（データ欠損の可能性あり）」という警告が発行される。

たとえば、レベル4では、次の代入式は警告の対象となる。

```
int main() {
    int b = 0, c = 0;
    short a = b + c;   // C4244
}
```

Visual C++ の実行時検査

Visual C++ .NET 2003 の**/RTCc** コンパイラフラグ

- 整数がより短い変数に代入されデータ欠損が生じる時に、切り捨てエラーをキャッチする実行時検査の機能

Visual C++ には、**/RTC** 設定を無効にしたり復活させたりするための **runtime_checks** プラグマも用意されている。

オーバーフローをはじめとするその他の実行時エラーをキャッチするフラグは用意されていない。

実行時エラー検査は、性能上の理由からリリース（最適化）ビルド時には無効になる。

GCC の -ftrapv オプション

- 限定的ながら実行時に整数の例外を検出 （**signedのみ**）
- **加算**、**減算**、**乗算**の演算時に符号付きオーバーフローが発生したら、トラップ(例外)を生成
- 既存のライブラリ関数呼び出しを生成する

GCC の実行時検査は、事後条件に基づいて行われる。

つまり、演算が実行された後にその結果の有効性が検査される。

事後条件

符号なし整数の場合、加算の結果がどちらかのオペランドよりも小さければオーバーフローが発生している。

符号付きの整数の場合は、`sum = lhs + rhs` を例とすると

- `lhs` が負の数でなく、かつ `sum < rhs` であるなら、オーバーフローが発生している
- `lhs` が負で、かつ `sum > rhs` であるなら、オーバーフローが発生している
- これ以外の場合は、オーバーフローしていない

符号付き整数の加算

符号付き 16 ビット整数の加算によって発生するエラーを検出するために用いる **gcc** の実行時システム関数

```
Wtype __addvsi3 (Wtype a, Wtype b) {
  const Wtype w = a + b;
  if (b >= 0 ? w < a : w > a)
    abort ();
  return w;
}
```

加算演算を実行し、エラーが発生したかどうかを判断するために加算結果をそのオペランドと比較する

abort() は次の場合に呼び出される
- **b** が非負で、かつ **w < a**
- **b** が負で、かつ **w > a**

“GCC Internals” の arithmetic functions に詳細
http://gcc.gnu.org/onlinedocs/gccint/

安全な整数演算

安全な整数演算 1

整数演算の結果、エラー条件が発生したり、データが消失する危険性がある。

整数の脆弱性に対して最初にとるべき防御策は、範囲検査を徹底すること。

- 明示的
- 暗黙的 - 強い型付けによる

安全な整数演算 2

もうひとつのアプローチは、整数演算を一つ一つ保護すること。

このアプローチには膨大な労力がかかり、パフォーマンスにコストがかかる可能性がある。

信頼できないソースの影響を受ける可能性のある入力が 1 つでもあるすべての整数演算について、安全な整数ライブラリを使用する。

C言語と互換性のあるライブラリ

IA-32特有のメカニズムを利用することで整数オーバーフロー条件を検出する。

unsigned 加算関数

JPCERT CC®

```
bool UAdd(size_t a, size_t b, size_t *r) {
  __asm {
    mov eax, dword ptr [a]
    add eax, dword ptr [b]
    mov ecx, dword ptr [r]
    mov dword ptr [ecx], eax
    jc  short j1
    mov al, 1 // 1 is success
    jmp short j2
j1:
    xor al, al // 0 is failure
j2:
  };
}
```

unsigned 加算関数の例

```
int main(int argc, char *const *argv) {
  unsigned int total;
  if (UAdd(strlen(argv[1]), 1, &total) &&
      UAdd(total, strlen(argv[2]), &total)) {
    char *buff = (char *)malloc(total);
    strcpy(buff, argv[1]);
    strcat(buff, argv[2]);
  else {
    abort();
  }
}
```

結合した文字列の長さは、エラー条件に対する適切なチェックを行う **UAdd()** を使って計算する

IntegerLib

IntegerLib は、**整数エラー条件**を起こさない C プログラムを記述するためにソフトウェア開発者が利用できるユーティリティ関数群。

- ISO/IEC TR 24731 で定義されている実行時における制約処理のメカニズムを使用
- 『Hacker's Delight』(Henry S. Warren 著。邦題:『ハッカーのたのしみ』) で解説されている高性能のアルゴリズムを使用

IntegerLib は CERT/CC によって開発され、次のサイトから自由に入手できる

http://www.securecoding.cert.org

(「CERT C Programming Language Secure Coding Standard」の「Integers」を参照)

IntegerLib の例

ライブラリを使って2つの **signed long** 整数値を加算する方法:

```
long retsl, xsl, ysl;
xsl = LONG_MAX;
ysl = 0;
retsl = addsl(xsl,ysl);
```

短い整数型

短い整数型(**char** や **short**)の場合、用意されている安全な変換関数を用いて加算の結果を切り捨てる必要がある。

```
char retsc, xsc, ysc;
xsc = SCHAR_MAX;
ysc = 0;
retsc = si2sc(addsi(xsc, ysc));
```

SafeInt クラス

SafeInt は、David LeBlanc が作成した C++ 用のテンプレートクラス。

事前条件の手法を実装しており、処理を実行する前にオペランドの値を検査することで、エラーが発生するかどうかを判断する。

クラスはテンプレートとして宣言されるため、どの整数型に対しても適用できる。

関係する演算子は、添字演算子 [] を除き、すべてこの実装のもので置き換えられている。

SafeInt の例

```
int main(int argc, char *const *argv) {
    try{
        SafeInt<unsigned long> s1(strlen(argv[1]));
        SafeInt<unsigned long> s2(strlen(argv[2]));
        char *buff = (char *) malloc(s1 + s2 + 1);
        strcpy(buff, argv[1]);
        strcat(buff, argv[2]);
    }
    catch(SafeIntException err) {
      abort();
    }
 }
```

変数 **s1** と **s2** が SafeInt 型として宣言されている

この + 演算子が呼び出される時には、SafeInt クラスの一部として実装されている安全なバージョンの演算子が用いられる

加算

符号なし整数の加算演算の左側 (LHS: left-hand side) と右側 (RHS: right-hand side) の合計が、次を超える場合、整数オーバーフローになる。

- **UINT_MAX: unsigned int** 型の加算の場合
- **ULLONG_MAX: long long** 型の加算の場合

事前条件の例

lhs と rhs が unsigned int であり、かつ次の場合に、オーバーフローが発生する。

```
lhs + rhs > UINT_MAX
```

加算によるオーバーフローを回避するために、演算子 +を用いて以下を検査できる。

```
lhs > UINT_MAX - rhs
```

あるいは、

```
~lhs < rhs
```

SafeInt の特徴

SafeInt が優れている点

- **移植性が高い** - アセンブリ言語命令に依存しない
- **使い勝手がよい**
  - 演算子をインラインの式で使用できる
  - C++ の例外処理を利用する
  - exception handler クラスがテンプレートの引数

Microsoft Public License(MS-PL) でだれでも使える

- http://www.codeplex.com/SafeInt
- **gcc**でもコンパイルできる(もちろんVCも)

Safe Integer Library を使うべき場合の例

以下のような整数値が、信頼できないソースによって操作される場合、Safe Integer Library を使用する。

- **int StructSize**（構造体のサイズ）
- **int HowMany**（割り当てる構造体の数）

```
void* CreateStructs(int StructSize, int HowMany) {
   SafeInt<unsigned long> s(StructSize);
   s *= HowMany;
   return malloc(s.Value());
}
```

構造体のサイズと数の積を求めて、割り当てるメモリのサイズを決定する

乗算では整数オーバーフローが発生する可能性があり、バッファオーバーフローの脆弱性につながる

Safe Integer Library を使うべきでない場合の例

例外条件が発生しない場合は、Safe Integer Library を使用しない。

- タイトなループ
- 外部からの影響を受けない変数

```
…
char a[INT_MAX];
for (int i = 0; i < INT_MAX; i++)
a[i] = '¥0’;
…
```

検証と確認

JPCERT CC®

入力値を検査しても…

- その先で行われる整数演算がオーバーフローなどエラー条件を引き起こさない保証はない。

検証を行っても…

- 完全な品質を保証することは困難
- よほど単純なプログラムでないかぎり、可能性のあるすべての入力範囲を網羅するのは非現実的
- ただし適切に検証すれば、コードの安全性を向上させることができる

検証 2

整数の脆弱性検証では、すべての整数型変数に対して境界条件をチェックすべき。

コード中に型の範囲検査が実装されていれば、上限と下限の境界について正しく動作することを検証

- 境界検査が実装されていない場合、使用されている様々な整数サイズについて、その最大値と最小値で検証

ホワイトボックステストをおこない、整数変数の型を特定する。

ソースコードが手に入らない場合は、すべての型について最大値と最小値を用いて検証する。

ソースコード監査

整数の値域エラーが起きないかどうか、ソースコードを監査する。

監査でチェックすること:

- 整数の型範囲が適切に検査されていること
- 入力値が使用目的の範囲に制限されていること

負の値をとらない整数は、

- 符号なしの型として宣言されていること
- 値がとる上限、加減の範囲チェックが適切にされていること

信頼できない入力ソースからの整数を使ったすべての演算は、安全な整数ライブラリをつかって処理。

整数オーバーフローを防ぐ方法

整数オーバーフローを防ぐ方法

「こうすれば絶対に整数オーバフローを防げる」という 決定打はない！

1.unsigned 型をつかう

- 特にメモリを割り当てたり、インデックスするコードでは unsigned 型を使えば、レンジチェックは上限だけすれば良い

2. よからぬ事態を想定する

- int, char, size_t など標準データ型は、処理系によって定義が異なることを想定する（bit幅、charがデフォルトでsignedかunsignedかなど）
- とくに、コードが32-bitと64-bitの両方でコンパイルされる場合に注意

3. ユーザからの数値入力を制限する

– 数値入力の範囲を制限する

4. メモリの割り当て・参照に使う値はチェックしてから使う

- 最終チェック後に値をいじらない（演算しない）
- 特に signed 型をつかったメモリ割り当てなど、値が暗黙的に unsigned 型に変換される場合に注意

5. コンパイラの警告メッセージはしっかり吟味

5.1.Miscrosoft Visual Studio CL

警告レベルは最大でコンパイル /W4

整数関連の警告にはとくに注意

- C4018 - signed 型と unsigned 型の数値を比較するには、コンパイラで signed 型の値を unsigned 型に変換する必要がある
- C4244 – 整数型がより小さな整数型にされ、データが失われた可能性がある
- C4389 - 演算で符号付きの変数と符号なしの変数が使用され、データが失われた可能性がある

開発ビルドでは /RTCc を使う。

- 値が小さなデータ型に代入されて切り捨てが発生する場合をランタイムエラーにできる。
- オーバーヘッドが大きく、リリースビルドや /O 最適化と一緒につかえない

5.2.GCC

整数関連のオプションをつけてコンパイルする

- -Wconversion – 問題のありそうな型変換について警告する。signed型定数とunsigned型の暗黙的変換など。
- -Wsign-compare – signed値がunsigned値に変換されて比較演算が誤った結果を生み出す場合に警告

-ftrapv でランタイムエラーチェックをオンにする。

6. 整数変換のルールを理解する

ルールは複雑だが覚える価値はある。少なくとも次の2点は暗記。

- 1つ以上の型が含まれる演算では、精度の低い方がアップキャストされ、値の表現が変化することがある
- unsigned 型はより大きな signed 型に暗黙的にキャストされることがある

7. オーバーフローが発生しうる演算については、事前条件と事後条件をチェックする

- 演算子にはオーバーフローを引き起こすものとそうでないものがある。(ex. +, -, *, /, ++, --, +=, -=, *=, /=, <<=, >>=, <<, >>, unary-)
- 自分でチェックできない場合、SafeInt や IntSafe などのライブラリを使う

もくじ

JPCERT CC®

第一部

整数のメカニズム

- 整数表現
- 型
- 型変換
- 演算子
  - 加算/減算
  - 乗算
  - 除算
  - 左シフト、右シフト

第二部

- 整数のエラー条件と脆弱性
- 脅威の緩和方法
- ⇒まとめ

まとめ

整数の脆弱性を回避するポイントは、計算機における

整数の振る舞い、取り扱いを理解する

⇒インデックス（またはその他のポインタ算術演算）、長さ、サイズ、ループカウンタとして使われている整数に注意

- `Safe Integer Library` を用いて例外条件を取り除く
- インデックスとして使われる整数値は範囲検査
- すべてのサイズと長さに `size_t` や`rsize_t` を使用

参考情報

参考情報

オンラインの資料

Michael Howard. *Reviewing Code for Integer Manipulation Vulnerabilities.*
http://msdn.microsoft.com/en-us/library/ms972818.aspx

「Michael Howard's Web Log」のエントリ *Safe Integer Arithmetic in C.*
http://blogs.msdn.com/michael_howard/archive/2006/02/02/523392.aspx

Secure C Library. http://std.dkuug.dk/jtc1/sc22/wg14/www/docs/n1031.pdf

Safe Integer Operations.
https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/coding/312-BSI.html

blexim. *Basic Integer Overflows.* Phrack Magazine.
http://www.phrack.org/issues.html?issue=60&id=10#article

Oded Horovitz. *Big Loop Integer Protection.* Phrack Magazine.
http://www.phrack.org/issues.html?issue=60&id=9#article

整数のセキュリティに関するCERT/カーネギーメロン大学のポータル
http://www.cert.org/secure-coding/integralsecurity.html

この資料のベースとなっている書籍

『C/C++ セキュアコーディング』
著者：Robert C. Seacord
翻訳：JPCERT/CC, 歌代和正監訳
アスキー・メディアワークス
368ページ
2006年11月発売

C/C++ セキュアコーディングだけでなく、Cのコードとスタックやヒープメモリの関係など、マシンアーキテクチャに関する知識を深めるのにも役立ちます。

整数に関する CERT C のルール
INT30-C. 符号なし整数の演算結果がラップアラウンドしないようにする
INT31-C. 整数変換によってデータの消失や解釈間違いが発生しないことを保証する
INT32-C. 符号付き整数演算がオーバーフローを引き起こさないことを保証する
INT33-C. 除算および剰余演算がゼロ除算エラーを引き起こさないことを保証する
INT34-C. 負のビット数、あるいはオペランドのビット数以上シフトしない
INT35-C. 整数式をより大きなサイズの整数に対して比較や代入をする際には、事前に演算後のサイズで評価する

CERT C セキュアコーディングスタンダード

JPCERT CC®

整数に関する CERT C のリコメンデーション
INT00-C. 処理系のデータモデルについて理解する
INT01-C. オブジェクトのサイズを表現するすべての整数値に rsize_t もしくは size_t を使用する
INT02-C. 整数変換のルールを理解する
INT03-C. セキュアな整数ライブラリを使用する
INT04-C. 信頼できない入力源から取得した整数値は制限する
INT05-C. 可能性のあるすべての入力を処理できない入力関数を使って文字データを変換しない
INT06-C. 文字列トークンを整数に変換するには strtol() 系の関数を使う
INT07-C. 数値には符号の有無を明示した文字型のみを使用する
INT08-C. すべての整数値が範囲内にあることを確認する
INT09-C. 列挙定数が一意の値に対応することを保証する
INT10-C. % 演算子を使用する際、結果の剰余が正であると想定しない
INT11-C. ポインタ型から整数型への変換やその逆の変換は注意して行う
INT12-C. 式中で使用される単なるintのビットフィールドの型について想定しない
INT13-C. ビット単位の演算子は符号無しオペランドに対してのみ使用する
INT14-C. 同じデータに対してビット単位の演算と算術演算を行わない
INT15-C. プログラマ定義の整数型に対する書式指定入出力には、intmax_t もしくは uintmax_t を使用する
INT16-C. 符号付き整数の表現形式を想定しない
INT17-C. 処理系に依存しない方法で整数定数を定義する

CERT C セキュアコーディングスタンダード

ルールの最新版はCERT/CCのWikiでアップデートされています
https://www.securecoding.cert.org (英語)

日本語の情報は、JPCERT/CCのホームページか書籍『CERT C セキュアコーディングスタンダード』をご覧下さい
http://www.jpcert.or.jp/sc-rules/

『CERT C セキュアコーディングスタンダード』
著者：Robert C. Seacord
翻訳：JPCERT/CC 久保正樹，戸田洋三
アスキー・メディアワークス
368ページ
2009年9月29日発売

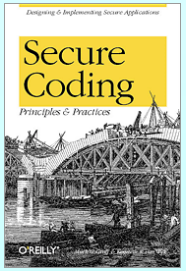
参考図書

Brian Chess& Jacob West.Secure Programming with Static Analysis, Addison-Wesley, 2007 	静的ソースコード分析がどのようなメカニズムで脆弱性につながるエラーやセキュリティ上の欠陥を発見するか詳説。手法の理論と実践、ソフトウェア開発プロセスにどのようにして取り入れるか、ツールを使った効率的コードレビューの方法について分かりやすく解説している。著者のBrian Chessは静的解析ツールFortify社の創業者兼主任研究員。
Gary McGrow. Software Security: Building Security In, Addison-Wesley, 2006 	ソフトウェアセキュリティを実践する方法を詳しく解説。論旨が明快で英語でも読みやすい。BlackHat(攻撃者)とWhiteHat(セキュリティ)の両面から問題を論じている。ソフトウェア開発ライフサイクルの「7つのタッチポイント」(http://www.swsec.com/resources/touchpoints/)について解説している。

参考図書

Michael Howard. The Secure Development Lifecycle, Microsoft Press, 2006 	開発の各フェーズでセキュリティ対策を行うマイクロソフトの Security Development Lifecycle (SDL) について詳しく説明している。 SDL についての詳しい説明は http://www.microsoft.com/japan/msdn/security/general/sdl.aspx
Robert C. Seacord. 『C/C++セキュアコーディング』 アスキー, 2006 	C/C++セキュアコーディングに関して日本語で手に入る書籍としては一番よくまとまっている。翻訳はJPCERT。 C/C++の文字列操作、ポインタ偽装、動的メモリ管理、整数演算、書式指定出力、ファイル入出力などに関するコーディングエラーがいかに脆弱性につながるか、脆弱性を防ぐにはどうコーディングすればよいか解説している。 C/C++プログラマ必読の書！

参考図書

GregHogland& Gary McGraw. Exploiting Software: How to Break Code, Addison-Wesley, 2004 	攻撃が実際にどのように行われるのかを解説することで、開発者がセキュアなプログラムを書くことの重要性を説く。クライアント・サーバ製品に対する攻撃の実際、攻撃を成立させるための入力値はどのように作られるか、などのトピックを取り上げる。特に、バッファオーバーフロー攻撃の解説とルートキットに関する解説は分かりやすい。
Mark G. Graff & Kenneth R. Van Wyk. Secure Coding: Principles and Practices, O'Reilly, 2003 	セキュアコーディングの実践方法について書いた本ではなく、セキュアコーディングについて書かれた本。簡潔かつポイントをしぼって書かれている。従って、各論の深い内容を期待してはダメ。

Michael Howard & David LeBlanc.『Writing Secure Code 第2版』, 日経BPソフトプレス, 2004	上下2巻本。マイクロソフトの開発環境における実践的コーディングの話は具体的で分かりやすい。マイクロソフトの開発環境で開発するプログラマにおすすめ。
John Viega& Gary McGraw. Building Secure Software: How to Avoid Security Problems the Right Way, Addison-Wesley, 2001 	セキュアなソフトウェア開発に必要な事柄を多角的に論じた好著。
Mark Dowd, John McDonald, and Justin Schuh. The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities 	脆弱性の百科事典。サンプルコードを交えてどのようなコードが脆弱性を引き起こすのか触れられている。C/C++の話も多数。Cに関する問題の章はダウンロード可能: http://www.informit.com/content/images/0321444426/samplechapter/Dowd_ch06.pdf

参考ウェブサイト

JPCERT CC®

過去の脆弱性事例データベースなど

CWE (Common Weakness Enumeration)
`http://cwe.mitre.org/`

CVE (Common Vulnerabilities and Exposures)
`http://cve.mitre.org/`

Making Security Measurable (セキュリティの定量化に関する取り組みへのリンク集)
`http://measurablesecurity.mitre.org/`

JVN (JPCERT, IPAが運営する日本の脆弱性情報サイト)
`http://jvn.jp/`

BugTraq
`http://www.securityfocus.com/`

milw0rm (過去の攻撃事例。危険なサイトなので、ウェブブラウザのJavascriptはオフにしてご利用下さい。)
`http://www.milw0rm.com/`

2009 CWE/SANS Top 25 Most Dangerous Programming Errors
`http://cwe.mitre.org/top25/`

参考ウェブサイト

JPCERT CC®

ソフトウェア開発関連

OWASP Source Code Flaws Top 10 Project
http://www.owasp.org/index.php/Category:OWASP_Source_Code_Flaws_Top_10_Project

Software Assurance Maturity Model
http://opensamm.org/Home.html

Gary McGraw. “Automated Code Review Tool for Security.”
http://www.cigital.com/papers/download/dec08-static-software-gem.pdf

ソフトウェアとセキュリティに関する様々な記事をダウンロードできる
http://www.cigital.com/papers/

Chris Wysopal. “We’ve Reached the Application Security Tipping Point.”
http://www.veracode.com/blog/2008/11/we’ve-reached-the-application-security-tipping-point/

Secure Programming for Linux and Unix HOWTO（日本語訳）
http://www.linux.or.jp/JF/JFdocs/Secure-Programs-HOWTO/index.html

参考

64-Bit プログラミングモデル

1つまたはそれ以上のCのデータ型のビット幅が変更されることで、プログラムは明白な影響を受けもすれば、それほど明確ではない影響も受ける。

基本的に、次の2つの問題が存在する:

- 64-bit データ型のいずれかで定義されたデータオブジェクトは、16 あるいは 32-bit システム上で同様に宣言されたデータオブジェクトとサイズが異なる。
- (C標準で規定されたものではなく、あるコードの開発者がいずれにしても利用する)基本データ型間の関係にたいする想定はもはや通用しない。

こうした想定に依存したプログラムは、多くの場合、適切に動作しなくなる

[**64-Bit Programming Models: Why LP64?**
http://www.unix.org/version2/whatsnew/lp64_wp.html]

LLP64

LLP64 は、**int** と **long** を32-bitデータ型のままにすることで、両者の間の関係を保存している。

Objects not containing ポインタを含まないオブジェクトのサイズは、32-bitシステムのそれと同じ。

64-bitスカラーデータのサポートは、**long long** **が提供する。**

LLP64 は実のところ、64-bitアドレスを備えた32-bitのデータモデルである。

データ型のサイズを仮定することで発生する実行時の問題の大半は、ポインタが **int** に収まるだろうという誤った推測に関連する。

これらの問題を解決するためには、 **int** もしくは **long** 型の変数を **long long** 型に変更する。

LLP64

LLP64 なオペレーティングシステムは、システムプログラミングインターフェイス（API）のデータ型定義の多くを変更するか、64-bit幅の新たなインターフェイスを導入することを余儀なくされる。

これらインターフェイスの大半は、25年もの間、すべてのUNIXオペレーティグシステムにわたり安定して利用されてきた。

従って、LLP64 は既存の仕様を広範囲にわたって変更してはじめて、64-bit幅をサポートすることができる。

ILP64

ILP64 は、**int, long, ポインタ**のサイズをすべて同じにすることで、これらの型の関係性を保存する。(ILP32 の場合のように)

ポインタを **int** や **long** 型の変数に代入してもデータの欠損にはつながらない。

一方で、このデータモデルではデータの移植性が無視されるか、あるいは、**int32** や **__int32** といった 32-bitのデータ型を追加することに依存する。

これによって既存のデータ型と潜在的に競合することになり、特にC言語における開発のスピリット(Cの基本データ型に型のサイズ記述を埋め込むことは避ける)に反する。

サイズとデータのアラインメントをどうしても保持しなくてはならないプログラムは、非標準のデータ型を使用することを余儀なくされ、ポータブルでなくなるかもしれない。

LP64

LP64 は中庸をとる。 8, 16, 32-bitのスカラー型 (**char**, **short**, および **int**) が提供されており、 32-bitシステム上でそのサイズとアラインメントを保持したオブジェクトを宣言することができる。

64-bit型 (**long**) が提供されることで算術機能がフルサポートされ、かつポインタ演算とあわせて利用することができる。

スカラー型オブジェクトにアドレスを代入するプログラムでは、そのオブジェクトを **int** ではなく **long** に指定してやる必要がある。

移植性の問題

32-bitシステムから移行するほとんどすべてのアプリケーションは、64-bitポインタを扱うには何らかの小さな変更を強いられる。特に、 **int** とポインタの相対的サイズに関して仮定している部分は変更が必要。

LP64 モデルでは問題を引き起こさない **int, char, short** および **float** (これらのデータ型のサイズは 32-bitシステムのそれと同じなので)の相対的サイズに関する仮定は、ILP64 モデルでは問題になる。

LP64 でも ILP64 でも 32-bitシステムからのポーティングが煩雑になることには違いないが、条件が同じならば、小さなデータ型のほうがアプリケーションのパフォーマンスは良い。

互換性(interoperability)の問題

ILP64 モデルには 32-bit データ型を自然に記述するすべがない。従って、そのような型を記述するには、例えば __int32 のような移植性の悪い識別子を利用しなくてはならない。

これはしかし、#ifdef 文を使わなくても 32-bit, 64-bit どちらのプラットフォームでも実行できるコードを生成する上で、実際的な問題を引き起こすかもしれない。

既存のプログラム中の int はおおむね、64-bit 環境でも 32-bit のままで問題ない ; ポインタ や long と同じサイズであることを期待されているのはごくわずか。

ILP64 だと int の大半は、__int32 に変更する必要がある。

- __int32 は 32 bit の int のようには動作しない
- __int32 はすべての操作が int (64-bits, 符号拡張)されねばならず、64-bit で算術演算される点で short と同じ。

従って、ILP64 における __int32 は ILP32 における int とも異なれば、LP64 における int とも異なる。

こういった差異は、案の定、見つけることの難しいバグを生む。

性能の特徴 1

2つのカテゴリーの違い:

- 定義されたモデルを適切に実装するための命令サイクルへのコスト
- 必要な場所に移動するのに要するメモリシステムへのコスト(すべてのメモリ階層において)

命令サイクルペナルティは、意図したプログラミングモデルのセマンティックを適切に実装するために追加の命令サイクルが必要とされるたびに、発生する。

例えば、LP64 では、`long` を含む型の混ざった状態では、 `int` 符号拡張するだけで済む。また、大抵の整数式には `long` が含まれない。

性能の特徴 2

商用アプリケーションにおけるより大きな現実的影響には、余分にメモリを消費すること、またそのメモリをシステム内で運搬するコストがあげられる。

- 64-bit 整数は 32-bit 整数の2倍のスペースを必要とする
- さらに、レイテンシのペナルティーは膨大になる可能性があり、特にディスクに対しては 1,000,000 CPU cycles を超える可能性も
- C/C++のプログラムでは、**int** が最も頻繁に用いられるデータ型
- ソフトウェアベンダの中にはLP64システム上で **int** をすべて **long** をおきかえることで ILP64 データモデルを近似し実験した

結論として ILP64 は用いないと判断されたが、その理由は **int** の幅が広がることの恩恵がなかったのと、余分にメモリを使用することで発生するパフォーマンス上のペナルティを支払いたくなかったから。

BACKUP

JPCERT CC®

int より小さい整数型(char, short)は、int 型に格上げされてから演算する。

K&R が出版されてから C89 仕様が決まるまでの間、整数格上げルールの実装について意見の大きな相違が発生した。

処理系は大きく次の2つの陣営に分かれた

- unsigned preserving
- value preserving

Unsigned and Value Preserving

unsigned preserving アプローチでは、int より小さな2つの符号無し型(**unsigned char** と **unsigned short**)を **unsigned int** に拡張するよう要求する。

value preserving アプローチでは、元の型の値がすべて **int** で表現できるのなら

- **int** より小さい型は **int** に変換される
- それ以外の場合、**unsigned int** に変換される

The C89 標準化委員会は、value preserving ルールを選んだ。

JPCERT CC®

整数の格上げの目的は、**計算途中の値**が**オーバーフロー**して算術エラーが起こるのを防ぐこと。

整数の格上げにより、変数（**c1** と **c2**）の値は **int** のサイズにまず格上げされる。

```
char c1, c2;
c1 = c1 + c2;
```

2つの **int** 型を加算し、その結果が **char** 型に切り詰められる。

符号なし乗算

```
if (OperandSize == 8) {
  AX = AL * SRC;
else {
  if (OperandSize == 16) {
    DX:AX = AX * SRC;
  }
  else {  // OperandSize == 32
    EDX:EAX = EAX * SRC;
  }
}
```

8 ビットのオペランドの結果が 16 ビットの destination レジスタに格納される

16 ビットのオペランドの結果が 32 ビットの destination レジスタに格納される

32 ビットのオペランドの結果が 64 ビットの destination レジスタに格納される

signed long long int の加算

JPCERT CC®

add 命令は、下位 32 ビットを加算する

```
sll1 + sll2
mov         eax, dword ptr [sll1]
add         eax, dword ptr [sll2]
mov         ecx, dword ptr [ebp-98h]
adc         ecx, dword ptr [ebp-0A8h]
```

adc 命令は、上位 32 ビットとキャリービットの値を加算する

符号付き整数への変換

符号なし整数から符号付き整数への変換において、

- 同じサイズへの変換の場合 - ビットパターンは保存される。最上位ビットは符号ビットになる。
- 大きなサイズへの変換の場合 - 値は 0 拡張されてから変換される。
- 小さなサイズへの変換の場合 - 下位ビットは保存される。

符号なし整数の最上位ビットが、

- セットされていない場合 - 値は変わらない
- セットされている場合 - 結果は負の値

符号なし整数への変換

符号付き整数から符号なし整数への変換において、

- 同じサイズへの変換の場合 - 元の整数のビットパターンは保存される。
- 大きなサイズへの変換の場合 - 符号拡張されてから変換される。
- 小さなサイズへの変換の場合 - 下位ビットは保存される。

符号付き整数の値が、

- 負でない場合 - 値は変わらない
- 負の場合 - （通常は大きな）正の値になる

符号の有無があいまいになるケース

処理系が2の補数でありかつ符号付き整数のオーバーフローが暗黙にラップアラウンドする場合、以下の2つ条件を満たすならば結果の値に符号が付くかどうかは分からない

- 式に **unsigned char** あるいは **unsigned short** が含まれ、演算の結果が符号ビットの立った **int** 幅の値になる場合

  (例: **unsigned char**に対する補数演算)
- その式の結果が、符号が重要な意味を持つ以下に示すようなコンテクストで使用される場合
  - **sizeof(int) < sizeof(long)** において結果が **long** 型に拡張されなければならない場合
  - 右シフトが算術シフトで行われる処理系における、右シフト演算子の左オペランドである場合
  - **/**, **%**, **<**, **<=**, **>**, or **>=** のいずれかのオペランドである場合

同様のあいまいさが発生するケースとして、**unsigned int** と **signed int** の負の値が演算される場合がある。

符号付き/符号なし「文字」

JPCERT CC®

char 型は符号付きもしくは符号なしの型を持ちうる。

最高位ビットがセットされている **signed char** を **int** 型で保存すると、結果は負の数になる。

127(**0x7f**)より大きな値を持つ可能性のある文字データを扱う場合、バッファ、ポインタ、およびキャストには **unsigned char** を使う。