

セキュリティホールの種類

◆ 最近の流行は……

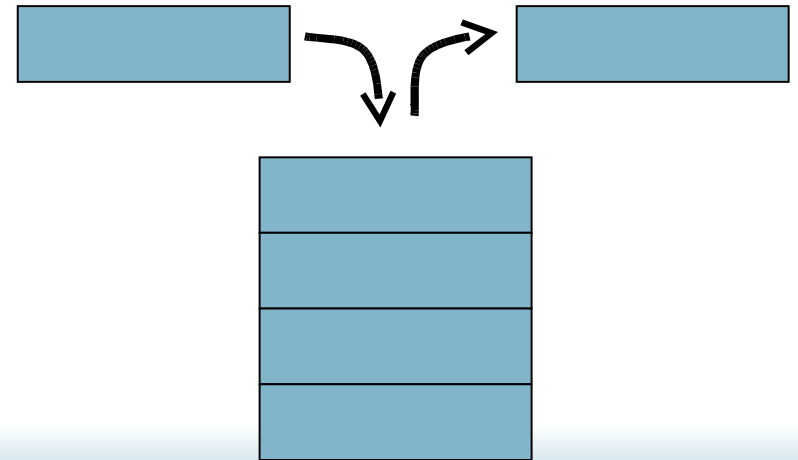
- Web でのサニタイジングの不備を突くもの
 - XSS(クロスサイトスクリプティング)
 - SQLインジェクション
- スタックオーバーフローによるもの
 - 「リモートから任意のコードを実行できるアプリケーションの脆弱性」といえばだいたいこれ
- 境界条件を利用したもの

C言語での関数呼び出し

- ◆ 関数を呼び出すときは、現在のアドレスを「スタック」に積んでから関数のプログラムがある場所にジャンプする

スタック

- ◆ スタック＝積み重ね・山
- ◆ データを記録する構造の一つで、上に重ねて上から持っていく



スタックの例

◆ 現実でのスタックの例

// ∩ Λ_Λ
 C^(('·ω·`)
 _つ^/∩c
 () ()

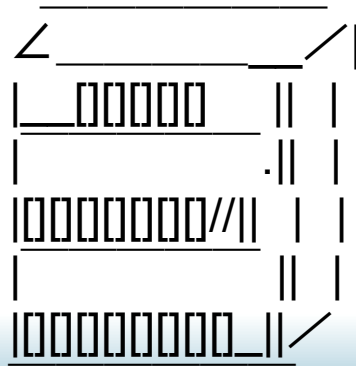
1・読書中

Λ_Λ
 C^(('·ω·`)
 _つ^/∩c
 () ()

??

2・わからない言葉が出てくる

Λ_Λ
 ('·ω·`)
 (o ..o)
 `u—u'

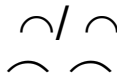

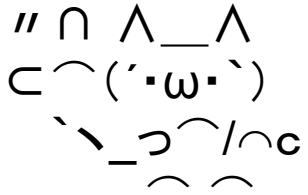

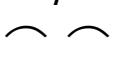
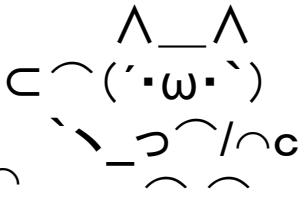





3・その本は置いて、その言葉について書いてある本を読む

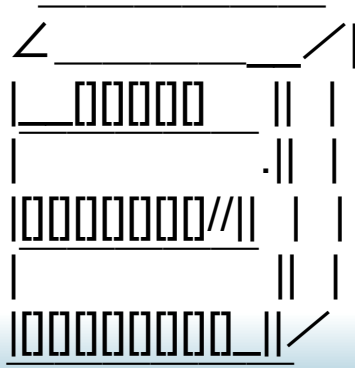
() ()

スタックの例

◆ 現実でのスタックの例




 4・さらに読む
 


 ??
 5・またわからない言葉が出てくる




 3・さらに今読んでた本を上重ねて、その言葉について書いてある本を読む



スタックの例

◆ 現実でのスタックの例

〃 ∩ Λ_Λ
C^(('ω'))
/_っ^/c
〇/〇
〇〇

7・さらに読む

Λ_Λ
C^(('ω'))
/_っ^/c
〇/〇
〇〇

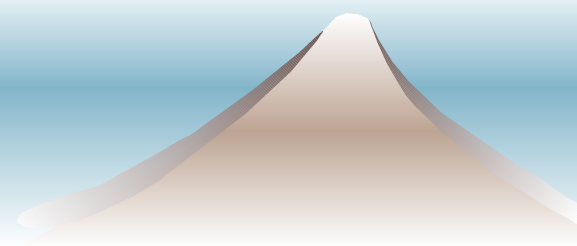
!!

8・言葉の意味が
わかった

Λ_Λ
(ω)
(o..o)
'u—u'

9・読んでいた本を返し、
積んである本の一番上を取る

〇/〇
〇〇



スタックの例

◆ 現実でのスタックの例

〃 ∩ ∧_∧
C^(('ω'))
/_っ^/c
〇/〇
〇

10・続きを読む

∧_∧ !!
C^(('ω'))
/_っ^/c
〇/〇
〇

11・最初の言葉の意味がわかった

∧_∧
(ω)
(o..o)
'u—u'

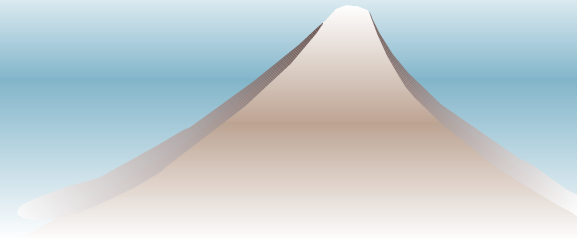
12・読んでいた本を返し、
積んである本の一番上を取る

〃 ∩ ∧_∧
C^(('ω'))
/_っ^/c
〇/〇
〇

13・最初の読書に戻る

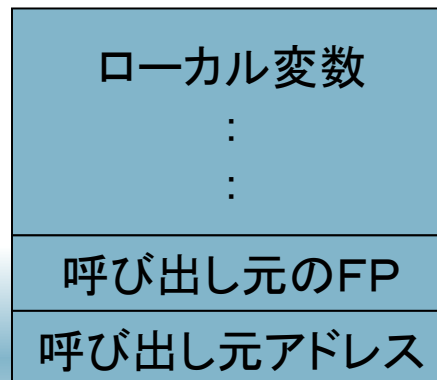
C言語での関数呼び出し(再掲)

- ◆ 関数を呼び出すときは、現在のアドレスを「スタック」に積んでから関数のプログラムがある場所にジャンプする



関数呼び出しとスタック

- ◆ 関数呼び出しがされたとき、スタックはどのようなになっているか？
- ◆ ローカル変数、呼び出しもとのフレームポインタ、呼び出し元アドレス(関数が終了したときに戻る先)が入っている

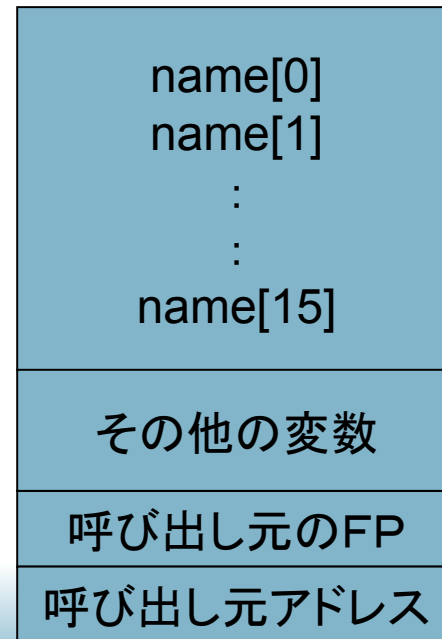


セキュリティホールのあるコード

◆ 問題が起こる代表的なコード

```
int getbuf()  
{  
    char name[16];  
    printf("Input your name: ");  
    scanf("%s", name);  
  
    return 0;  
}
```

右の関数が呼ばれたときのスタック



セキュリティホールのあるコード

- ◆ さっきのコードをコンパイルしてみました

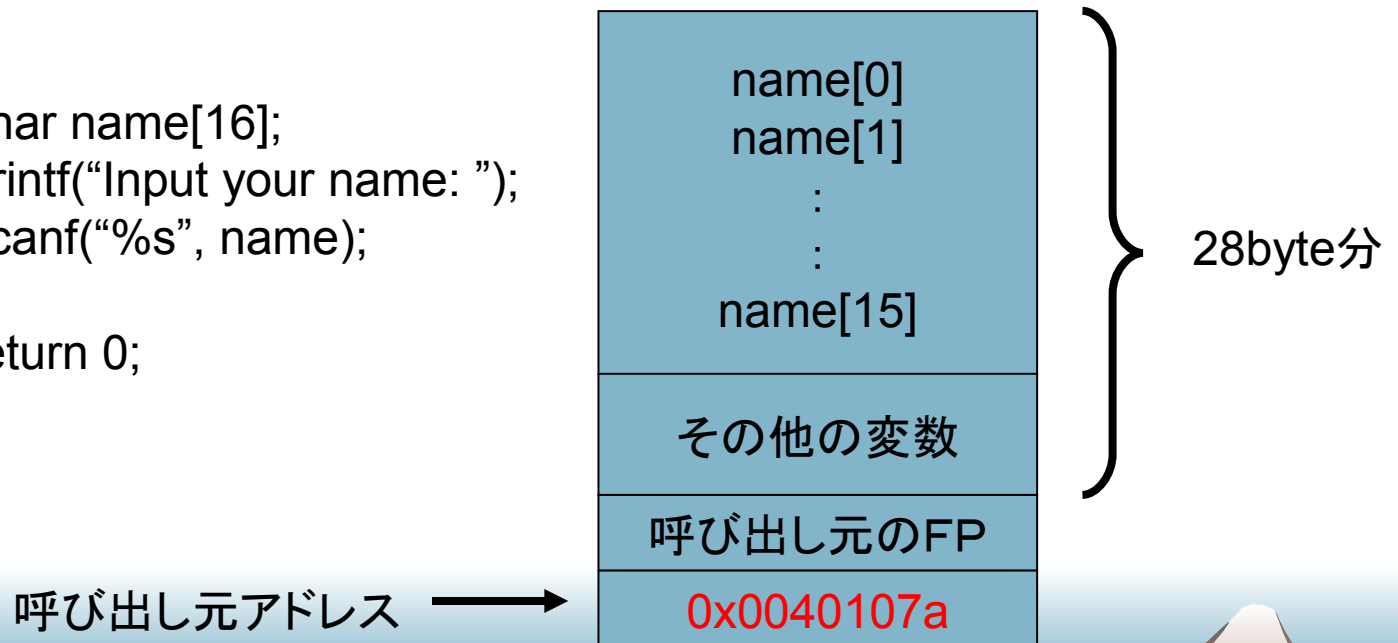
```
0x0040107a <main+42>: call 0x401086 <test>
:
0x00401086 <test+0>:  push %ebp
0x00401087 <test+1>:  mov  %esp,%ebp
0x00401089 <test+3>:  sub  $0x28,%esp
0x0040108c <test+6>:  lea  0xfffffe8(%ebp),%eax
0x0040108f <test+9>:  mov  %eax,0x4(%esp)
0x00401093 <test+13>: movl  $0x402000,(%esp)
0x0040109a <test+20>: call 0x401150 <scanf>
0x0040109f <test+25>: leave
0x004010a0 <test+26>: ret
```

セキュリティホールのあるコード

◆ 問題が起こる代表的なコード

```
int getbuf()  
{  
    char name[16];  
    printf("Input your name: ");  
    scanf("%s", name);  
  
    return 0;  
}
```

右の関数が呼ばれたときのスタック



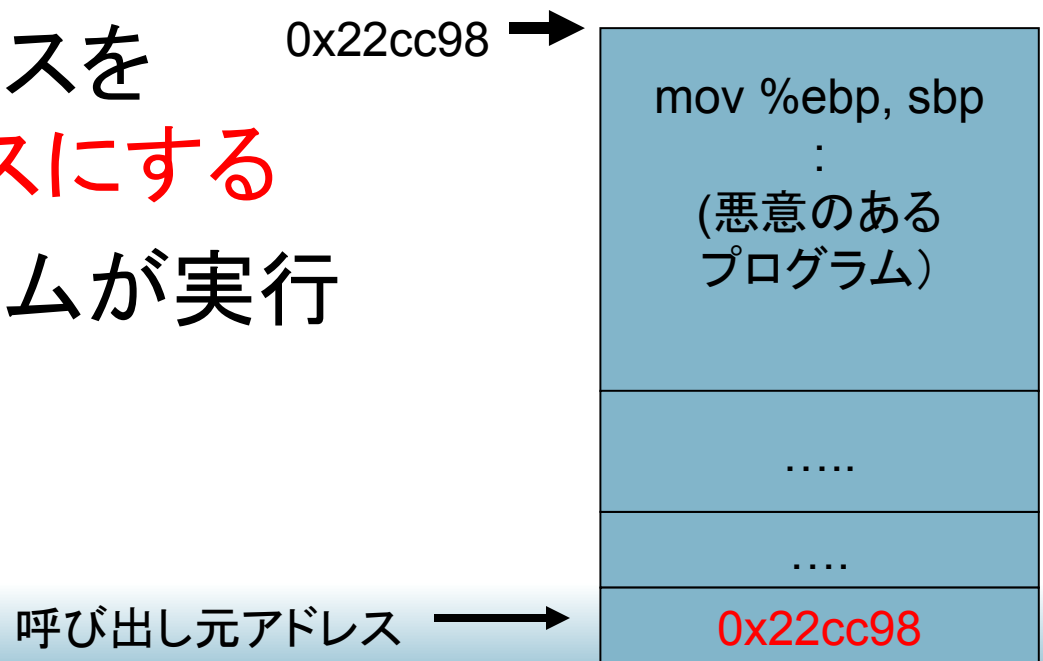
```
0x00401089 <test+3>: sub $0x28,%esp
```

セキュリティホールのあるコード

- ◆ 画面上では Input your name: と出ている
- ◆ ここでおもむろに16文字以上の名前を入れると……

悪用する場合……

- ◆ もっと長いバッファがあるとして……
- ◆ 入力で直接プログラムをバッファに書き込む
- ◆ 呼び出し元アドレスを $0x22cc98$ →
バッファのアドレスにする
- ◆ 入力したプログラムが実行されてしまう！



実際の場合には

- ◆ シェルコード(シェルを呼び出す)や、スタートアップファイル書き換え、外部からプログラムをダウンロードして実行など
- ◆ 一旦乗っ取ってしまえばあとはどうとでも

まとめ

- ◆ ノイマン式アーキテクチャはプログラムもデータもメモリ上で扱う
- ◆ ただのデータ入力をさせたつもりで、プログラムを書き込まれてしまう
- ◆ 不用意なバッファ・関数の使用がセキュリティホールを作る

おまけ

- ◆ これってOS・CPUとかで防げないの？
 - 最近のOSでは対策もいくつかある
 - OpenBSD / Solaris
 - W^X: 書き込めるセグメントにあるプログラムは実行しない (実行しようとする例外で落ちる)
 - もともとCPUにはメモリ保護機能がある、保護が甘いだけ
 - AMD64 / Sparc には W^X と同じ事を CPU レベルでやる機能もある
 - ただ互換性に問題がある、一部のアプリケーションは動かなくなってしまう (例: 動的にコードを生成するアプリ・難解読化コード等)

おまけ

プロセスの構造、CPUのメモリ保護

- ◆ OSによるが、4つの部分に分けられている
- ◆ コード部は読み込み専用
 - 普通にコード部を書き換えようとしてもうまくいかない

コード部
データ部
ヒープ部
スタック部