C/C++ セキュアコーディングセミナー
2010年度版

文字列

JPCERT コーディネーションセンター

依然、脅威の高いバッファオーバーフロー

MITREの2007年のレポートによると、

バッファオーバーフローは過去数年No.1の脆弱性

2005年以降ウェブアプリの台頭により順位を下げたが、2007年のデータではNo.2の脆弱性とその脅威は衰えず

OSベンダの脆弱性としては依然No.1

“Vulnerability Type Distributions in CVE”

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

バッファオーバーフロー脆弱性を突くBlaster

W32.Blaster.Worm

- Windows RPCサービスのバッファオーバーフロー脆弱性を突き、ネットワーク上の脆弱なシステムに感染。感染したシステムをDDoS攻撃の踏み台に悪用。
- 800万台の Windows システムが感染
- 想定被害総額 500億円以上

☞Zotob, Sasser, Slammer, Nimda, Code Red などのワームやウィルスもバッファオーバーフローの脆弱性を攻撃するもの。プログラマのうっかりミスが甚大なインパクトを与えた実例。

バッファオーバーフロー脆弱性を攻撃する
W32.Blaster.Worm

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

感染攻撃

感染攻撃

感染攻撃

⑥リモートでワームのダウンロードと実行を指示

感染攻撃

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）

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**

W32.Blaster.Wormに攻撃されたWindowsのRPCコード

```
error_status_t _RemoteActivation(
   /* ... */, WCHAR *pwszObjectName, ... ) {
  *phr = GetServerPath(pwszObjectName,&pwszObjectName);
    /* ... */
}

HRESULT GetServerPath(
  WCHAR *pwszPath, WCHAR **pwszServerPath ){
  WCHAR *pwszFinalPath = pwszPath;
  WCHAR wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1];
  hr = GetMachineName(pwszPath,wszMachineName);
  *pwszServerPath = pwszFinalPath;
}
HRESULT GetMachineName(
  WCHAR *pwszPath,
  WCHAR
wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1]){
  pwszServerName = wszMachineName;
  LPWSTR pwszTemp = pwszPath + 2;
  while ( *pwszTemp != L'¥¥' )
         *pwszServerName++ = *pwszTemp++;
  /* ... */
```

- GetMachineName()のループ処理の境界チェックに不備
- pwszTempが参照する文字型配列の先頭からMAX_COMPUTERNAME_LENGTH_FQDN + 1文字にバックスラッシュが入っていないと、最後にポインタの終端を超えて参照してしまう
- 攻撃可能な未定義の動作
- コードインジェクションが可能

修正コードを考えてみる

セキュアな実装にするには、どのような修正が考えられるか?

修正案: バックスラッシュもしくはnull終端文字(L'¥0')もしくはバッファの終端に達した場合にループが終了するように修正する？

```
HRESULT GetMachineName(
  wchar_t *pwszPath,
  wchar_t wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1]){
  wchar_t *pwszServerName = wszMachineName;
  wchar_t *pwszTemp = pwszPath + 2;
  wchar_t *end_addr    = pwszServerName + MAX_COMPUTERNAME_LENGTH_FQDN;
  while ( (*pwszTemp != L'¥¥')
     &&  ((*pwszTemp != L'¥0'))
     && (pwszServerName < end_addr) )
  {
    *pwszServerName++ = *pwszTemp++;
  }
  /* ... */}
```

Cにおける文字列の問題

- 文字列を表現するデータ構造(null終端文字配列)はエラーを生みやすい
  - 文字列処理のための配列操作でエラーに
  - メモリ領域確保のための長さ計算でエラーに
- 一貫性のないCの標準関数が事態を悪化させる要因
  - null終端する/しない、サイズ引数を取る/とらない
- バッファオーバーフロー

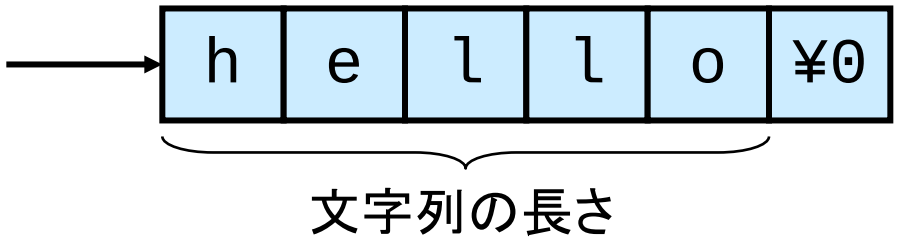
☞脆弱なコード例に学び、注意のポイント、セキュアな文字列処理の方法を習得しよう！

もくじ

Cの文字列に関する基礎知識のおさらい

null終端バイト文字列 (NTBS)

- null終端バイト文字列 (Null Terminated Byte String) 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。
- 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。
- 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(バイト数)を乗じた値。

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、単なる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”;
```

3種類の char 型

符号修飾宣言の違いで、異なる型として扱われる。

- **signed char**
- **unsigned char**
- **char**
  - 処理系定義であり、signed または unsigned 型として取り扱われる

文字データを取り扱う場合は「単なる」char を使う

- 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

文字データには「単なる」char を使う

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

len = strlen(pc_str); /*warning出ない*/

len = strlen(sc_str); /*signed,warning出る*/

len = strlen(uc_str); /*unsigned,warning出る*/
```

strlenの関数プロトタイプ：
`size_t strlen(const char *s);`

※ C++コンパイラではエラーになる。

☞STR04-C. 基本文字集合にある文字を表すには単なる char を使用する

std::basic_string

C++ の標準化により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
- バ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**

C++ における文字列

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++ プログラムでは NTBS と basic_stringを組み合わせるなど、複数の文字列型を混在して利用する状況が多い。

- 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

文字列に関する
セキュアコーディング

文字列に関するセキュアコーディング

- ここから文字列に関するCERT C セキュアコーディングルールを1つひとつ見ていきます
- 文字列処理に関するコーディングエラーにはどのようなものがあるのか、その対策はどうすればよいか、理解を深めましょう

注意

- 背景が赤のコード…脆弱なコード例
- 背景が青のコード…セキュアなコード例

とりあげるルール一覧

1. 文字列はnull終端させる
2. 文字データとnull終端文字を格納するために十分な領域を確保する
3. ワイド文字の文字列サイズは正しく求める
4. 文字列リテラルを変更しない
5. 長さに制限のないコピー元から固定長配列へデータをコピーしない
6. コンテナの変更により無効になったイテレータを使用しない
7. 文字列リテラルで初期化された文字配列の範囲を指定しない
8. 要素へのアクセスは範囲チェックして行う

セキュアコーディングルール1

文字列はnull終端させる

STR32-C Null-terminate byte strings as required

結果の文字列がnull終端されない可能性があるコード

```
char ntbs[NTBS_SIZE];

ntbs[sizeof(ntbs)-1] = '¥0';
strncpy(ntbs, source, sizeof(ntbs));
```

ntbs < source であり，かつ sourceの最初のsizeof(ntbs)文字までにnull文字が含まれない場合，strncpy()の結果 ntbs は null 終端されない．

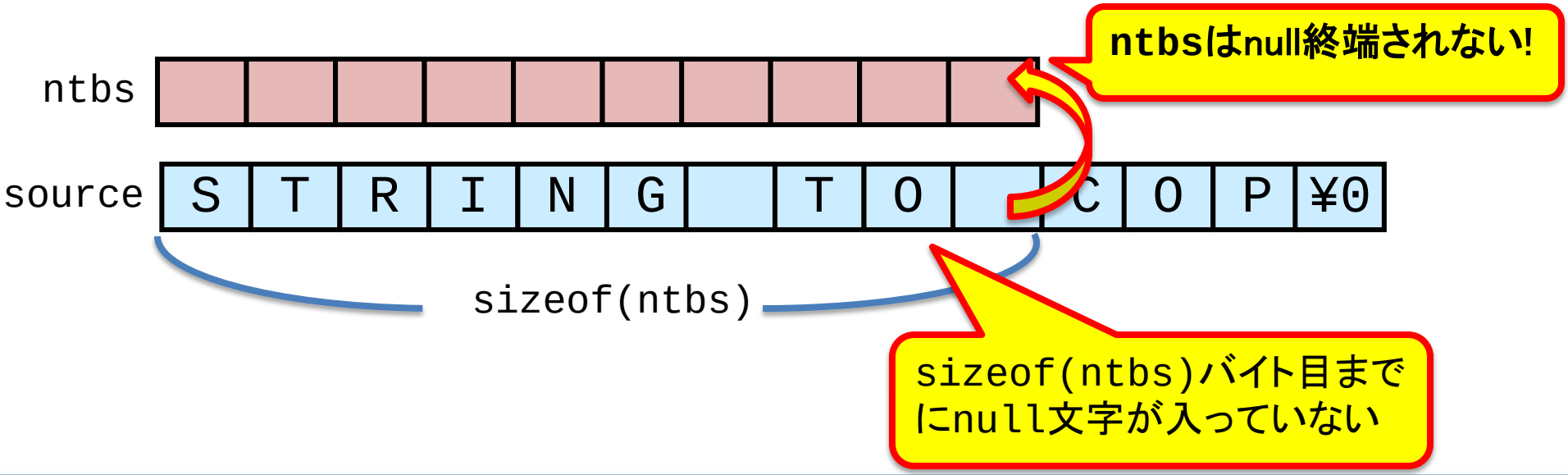
```
char ntbs[NTBS_SIZE];

memset(ntbs, 0, sizeof(ntbs)-1);
strncpy(ntbs, source, sizeof(ntbs)-1);
```

memset()の第三引数の指定が誤り．strncpy()の結果ntbsはnull終端されない可能性がある．

strncpy()

```
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，最大n文字をコピーする．null文字より後の文字はコピーされない.
- したがって，**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文字で終端されない。



null終端されていない文字列がなぜ危険なのか？

文字列がnull終端されていないと、どのようなセキュリティ上の問題につながることがあるのか？

- 文字列のサイズを正しく計算できない
- 文字列の文字数を正しく計算できない
- null終端文字を前提に動作する標準関数が未定義の動作をする

修正案1: 切り捨て

何が正しい解決法かはプログラマの意図に依存する.

文字列を切り捨ててでも, 結果を必ず null 終端文字列にすることを意図した修正案.

```
char ntbs[NTBS_SIZE];

strncpy(ntbs, source, sizeof(ntbs)-1);
ntbs[sizeof(ntbs)-1] = '¥0';
```

修正案2: 切り捨てずにコピー

```
char *source = "0123456789abcdef";
char ntbs[NTBS_SIZE];
/* ... */
if (source) {
  if (strlen(source) < sizeof(ntbs)) {
    strcpy(ntbs, source);
  }
  else {
   /* コピー元文字列がコピー先のサイズに収まらない場合のエラー処理 */
  }
}
else {
  /* NULL だった場合のエラー処理 */
}
```

切り捨てが発生しない
source < ntbs の場合しかコピー操作を行わない

修正案3: strncpy_s()を使う

strncpy_s() 関数はコピー元の配列からコピー先の配列に最大n文字コピーする.

コピー元配列からnull文字がコピーされなければ、コピー先配列のn番目の位置にnull文字が書き込まれ，結果の文字列は確実にnull終端される.

```
char *source;
char a[NTBS_SIZE];
/* ... */
if (source) {
  errno_t err = strncpy_s(a, sizeof(a), source, 5);
  if (err != 0) {
    /* エラー処理 */
  }
}else {
  /* NULL だった場合のエラー処理 */
}
```

STR07-C. Use TR 24731 for remediation of existing string manipulation code

ISO/IEC セキュリティ TR 24731-1

- より安全でセキュアなプログラミングを推進するため、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) で規定されたライブラリ
- 間違いを犯しにくいC標準関数群を定義
  - bounds-checking interfaces

TR 24731-1で定義された関数	置き換え対象の標準関数
strcpy_s()	strcpy()
strcat_s()	strcat()
strncpy_s()	strncpy()
strncat_s()	strncat()

TR 24731-1 関数の特徴

バッファオーバーフローに対するリスクを緩和

- 必ずnull終端する
- 書き込み先の領域が足りない場合はエラーにする．文字列の予期せぬ切捨てを行わない．

エラーが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++ 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る
- 誤用によりバッファ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）

strcpy_s()

終端 null 文字を含むコピー元文字列を、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** 型の引数を余分に取る。
- 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、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。

strcpy_s() の使用例

```
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);
}
```

切り捨てが発生するため strcpy_s() は失敗し，実行時制約エラーを引き起こす．

実行時制約 (runtime constraints)

各 _s 関数には実行時制約が規定されている

- **set_constraint_handler_s()** 関数で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

デフォルトのハンドラの動作は処理系定義．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()** を呼ぶ
- **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

null終端文字を確保する戦略

Cでは文字列を確実にnull終端することが重要

- これが保証されないとサイズ計算できない

しかしnull終端は簡単に壊れる

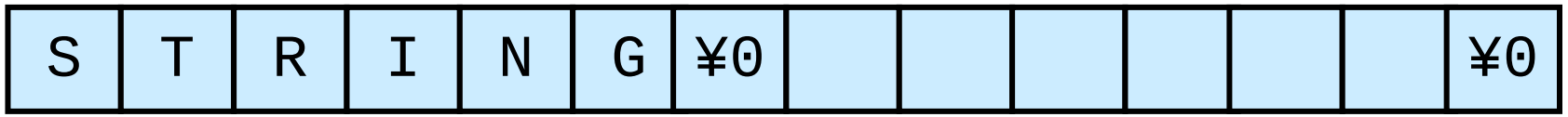
- 文字列がnull終端されていると想定したアプローチは破綻する

自分で「null終端してから使う」というアプローチ

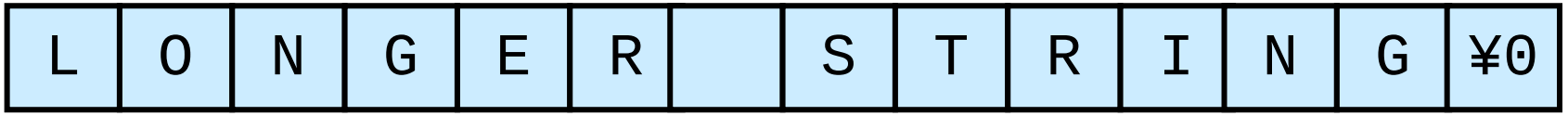
- 特にopen()やunlink()で有効
- では文字列のどこをnull終端するのがよいか？

割り当てたメモリの最後のバイトに強制的に書き込む

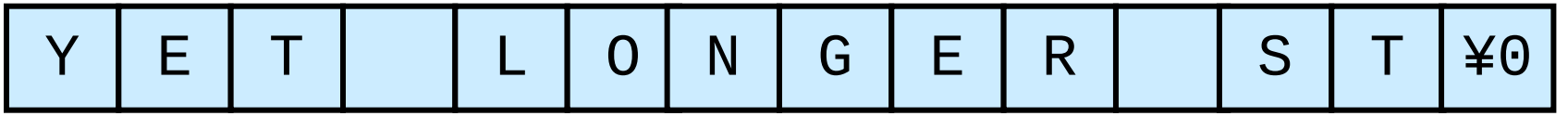
手作業でnull終端してから文字列を使う



- 文字列がバッファより短く、適切にnull終端されている場合、バッファの最後にnull文字を追加しても影響なし



- 文字列がちょうどバッファに収まっている場合、元のnull文字を上書きするだけで問題ない



- 文字列が長過ぎて適切にnull終端されていない場合、null文字を書き込むことで文字列長計算エラーやオーバーフローのリスクを軽減できる

まとめ–文字列はnull終端させる

- null終端されない文字列は様々な問題を引き起こす
- 結果の文字列がnull終端されるような関数呼び出し
- 文字列を使う側でnull終端してから使うというアプローチ

セキュアコーディングルール2

文字データとnull終端文字を格納するために十分な領域を確保する

STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator

オフバイワンエラー(off-by-one)を引き起こすコード例1

```
char dest[ARRAY_SIZE];
char src[ARRAY_SIZE];
size_t i;
/* ... */
for (i=0; src[i] && (i < sizeof(dest)); i++) {
  dest[i] = src[i];
}
dest[i] = '¥0';
/* ... */
```

forループでsrcからdestへデータをコピーしている.
しかし, dest に追加する必要のあるnull終端文字を考慮しておらず、null終端文字がdestの終端よりも1 バイトうしろに誤って書き込まれる可能性がある.

オフバイワンエラーを修正したコード例1

```
char dest[ARRAY_SIZE];
char src[ARRAY_SIZE];
size_t i;
/* ... */
for (i=0; src[i] && (i < sizeof(dest)-1); i++) {
  dest[i] = src[i];
}
dest[i] = '¥0';
/* ... */
```

destに追加されるnull終端文字を考慮して，ループの終了条件を修正(マイナス1)する.

オフバイワンエラー(off-by-one)を引き起こすコード例2

```
char lastname[20];
char firstname[20];
char name[40];
char fullname[40];

strncpy(name, firstname, sizeof(name));
strncat(name, lastname, sizeof(name)-strlen(name));
snprintf(fullname, sizeof(fullname), "%s", name);
```

```
char *strncat(char *restrict s1, const char *restrict s2, size_t n);
```
null終端された文字列s2をnull終端された文字列s1に最大n文字コピーし、最後にnull文字を追加する

同様に，strncat()，snprintf()を使用してバッファにデータをコピーしようとしているが，null文字を考慮していない.

オフバイワンエラーを修正したコード例2

```
char lastname[20];
char firstname[20];
char name[40];
char fullname[40];

strncpy(name, firstname, sizeof(name)-1);
strncat(name, lastname, sizeof(name)-strlen(name)-1);
snprintf(fullname, sizeof(fullname)-1), "%s", name);
```

バッファの終端に1バイト空き領域を残すことでnull文字を追加できるよう修正．これによりオフバイワンエラーは発生しない．

C標準の文字列処理関数の使用上の注意

- `strcat(char *dst, const char *src)`
  `strcpy(char *dst, const char *src)`
  - dstよりsrcの方が長いと、バッファーオーバーフローが発生しうる
- `sprintf(char *str, const char *format, ...)`
  - 入力文字数に対してstrの長さが足りないと、バッファオーバーフローの可能性
- `strncat(char *dst, const char *src, size_t n)`
  - dstの残りの文字数がn以下の場合、バッファオーバーフローが発生しうる
- `strncpy(char *dst, const char *src, size_t n)`
  - srcがdstより長い場合，n文字までdstに書き込むためオーバーフローしうる．結果のnull終端も保証されない．
- `snprintf(char *dst, const char *src, size_t n)`
  - srcがdstより長い場合，n文字までdstに書き込むためオーバーフローしうる．結果のnull終端も保証されない．

セキュアコーディングルール3

ワイド文字の文字列サイズは正しく求める

STR33-C. Size wide character strings correctly

サイズの判定: sizeof演算子とstrlen()関数

```
sizeof("abcdef");
sizeof("");
strlen("abcdef");
strlen("");
```

これらの演算結果、わかりますか？

sizeof 演算子

- オペランドの大きさ(バイト数)を返す
- 配列型オペランドに適用した場合の結果は，その配列中の総バイト数

size_t strlen(const char *s);

- s が指す文字列の長さを計算する
- 返り値: 終端ナル文字に先行する文字の個数

char型配列のサイズ判定

配列のサイズ判定を正しくおこなうには、

```
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．

ワイド文字の間違ったサイズ計算

```
wchar_t wide_str1[] = L"0123456789";
wchar_t *wide_str3 = (wchar_t *)malloc(wcslen(wide_str1) + 1);

if (wide_str3 == NULL) {
  /* Handle Error */}
/* ... */
free(wide_str3);
wide_str3 = NULL;
```

ワイド文字列の長さを計算するのにwcslen()を使っているが，正しい計算を行っていない．どこが間違ってる？

1ワイド文字が1バイトとは限らない．wcslen()は文字数を返すが，malloc()の引数はバイト数．
求めたい値は，(文字数 x 1ワイド文字のバイト数)

ワイド文字の正しいサイズ計算

```
wchar_t wide_str1[] = L"0123456789";
wchar_t *wide_str2 =
   (wchar_t *)malloc((wcslen(wide_str1) + 1) * sizeof(wchar_t));
if (wide_str2 == NULL) {
  /* Handle Error */
}
/* ... */
free(wide_str2);
wide_str2 = NULL;
```

ワイド文字のサイズで正しくスケールしており，正しいサイズのメモリを割り当てている.

セキュアコーディングルール4

文字列リテラルを変更しない

STR30-C. Do not attempt to modify string literals

文字列リテラルとは

JIS X 3010 6.4.5 文字列リテラル

単純文字列リテラル（character string literal）は，二重引用符で囲まれた0個以上の多バイト文字の並びとする（例えば"xyz"）．ワイド文字列リテラル（wide string literal）は，文字L という接頭語をもつことを除いて，単純文字列リテラルと同一とする.

...

プログラムがこれらの配列を変更しようとした場合，その動作は未定義とする.

```
char p1[] = "Always writable";
char *p2 = "Possibly not writable";
const char p3[] = "Never writable";

p1[0] = 'x';  // これらの
p2[0] = 'x';  // コードは
p3[0] = 'x';  // どのような結果になる？
```

6.7.3 型修飾子 脚注

処理系は，volatileでないconstオブジェクトを，読取り専用記憶域に置いてもよい。さらに，処理系はそのアドレスが使われないならば，そのようなオブジェクトに記憶域を割り付けなくてもよい。

文字列リテラルの誤った操作

```
const char* get_dirname(const char* pathname) {
  char* slash;
  slash = strrchr(pathname, '/');
  if (slash)
    *slash = '¥0'; /* 未定義の動作 */
  return pathname;
}
int main() {
  puts(get_dirname(__FILE__));
  return 0;
}
```

strrchr()関数の返り値である char* を使って pathname が参照するオブジェクトに変更を加えようとしている．ポインタが参照する先は文字列リテラルなので、これを書き換えようとするとシグナル(例えば SIGSEGV)があがるだろう．なぜなら，オブジェクトは読み込み専用メモリ領域に格納されているかもしれないから．

const修飾されたオブジェクトは変更しない

```
char* get_dirname(char* pathname) {
  char* slash;
  slash = strrchr(pathname, '/');
  if (slash)
    *slash = '¥0';
  return pathname;
}
int main() {
  char pathname[] = __FILE__;
  /* calling get_dirname(__FILE__) may be diagnosed */
  puts(get_dirname(pathname));
  return 0;
}
```

get_dirname()関数の引数を単なるchar*に変更することで，誤ってconst char* が渡されるとコンパイルエラーになる．文字列リテラルをconst修飾されていないchar*に変更することは言語仕様上は許される動作であるが、コンパイラによってはwarningを出すものもある．

セキュアコーディングルール5

長さに制限のないコピー元から固定長配列へデータをコピーしない

STR35-C. Do not copy data from an unbounded source to a fixed-length array

無制限文字列コピーによるバッファオーバーフロー

```
char *FixFilename(char *filename, int cd, int *ret) {
…
char fn[128], user[128], *s;
…
s = strrchr(filename, '/');
if(s) {
   strcpy(fn, s+1);
```

PHP/FI 2.0beta10 の php.cgiから抜粋した脆弱なコード

```
char * strcpy(char *dst, char *src);
```
― 文字列dstに文字列srcの内容をコピー

sの参照先であるfilenameがユーザからの入力値である場合，その長さは任意の長さとなりfn[128]に収まらずスタックでバッファオーバーフローが発生．httpdの子プロセスの権限で任意のコードが実行可能．

http://insecure.org/sploits/php.cgi.ncaa.overflow.pattern_restrict.html

動的メモリを確保するように修正

```
char *FixFilename(char *filename, int cd, int *ret) {
…
char user[128], *s;
char *fn = NULL;
…
s = strrchr(filename, '/');
if(s) {
   const size_t len = strlen(s);
   fn = malloc(len+1);
   if(fn) {
     strncpy(fn, s, len+1);
   } else {
      /* メモリー割り当てエラーの処理 */
   }
} else {
   /* エラー処理 */
}
free(fn);
fn = NULL;
```

セキュアコーディングルール6

コンテナの変更により無効になったイテレータを使用しない

ARR32-CPP. Do not use iterators invalidated by container modification

basic_string の利点

サイズの判定にはメンバ関数**size()**を使う

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

連結はオペレータ ‘+’ でおこなう

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

Cの文字列操作のような間違いは起きにくい

basic_string のイテレータ

イテレータを用いることで、文字列の内容に対する反復処理が可能。

```
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、イテレータが、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。

無効なイテレータ利用エラー

```
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++, ' ');
  }
}
```

イテレータ loc は最初の insert() 呼び出しの後に無効になる

有効なイテレータを利用する

```
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;
}
```

イテレータ loc の値は，挿入操作のたびに更新する

セキュアコーディングルール7

文字列リテラルで初期化される文字配列のサイズを指定しない

STR36-C. Do not specify the bound of a character array initialized with a string literal

JIS X 3010: 6.7.8 初期化

文字型の配列は，単純文字列リテラルで初期化してもよい。それを波括弧で囲んでもよい．単純文字列リテラルの文字（空きがある場合又は配列の大きさが分からない場合，終端ナル文字も含めて）がその配列の要素を前から順に初期化する．

```
const char s[3] = "abc";
```

配列のサイズは3，一方文字列リテラルのサイズは4．
この代入の結果得られるsはnull終端されていないので使うと危険．

```
const char s[] = "abc";
```

文字列リテラルを格納するために十分な領域をコンパイラが確保してくれる．

```
char s[3] = { 'a', 'b', 'c' }; /* NOT a string */
```

null終端文字ではなく文字の配列を作成する場合のコード．

セキュアコーディングルール8

要素へのアクセスは範囲チェックして行う

STR39-CPP. Range check element access

安全であるとは限らない要素へのアクセス

添字演算子(subscript operator) **[]** を使用した場合、境界チェックは行われない。

```
string bs(“01234567”);
size_t i = f();

bs[i] = '¥0’;
```

より安全な要素へのアクセス

a.at(n) は添字演算子 **[]** と同様の動作をするが、**n >= a.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;
}
```

配列の境界を越えた書き込み

```
int main(int argc, char *argv[]) {
  int t 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。

脅威の緩和方法

コンパイラオプションを活用して脅威を緩和

GCC: -fstack-protector でスタックを保護

JPCERT CC®

- 以下の防止、検出を行う
  - バッファオーバーフローによるローカル変数(ポインタ)の改ざん
  - バッファオーバーフローによるreturn address, saved ebpの改ざん
- ローカル変数とsaved ebpの間にcanaryを挿入し、この値が上書きされていないかチェックする
- GCC extension for protecting applications from stack-smashing attacks

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

GCC: Mudflap

- GCC4で新規追加されたC/C++対応のデバッグ補助機能
  - バッファオーバーフロー
  - メモリリーク
  - ヌルポインタ参照、ポインタの誤用
  - などを検知する
- `-g -fmudflap` をつけてコンパイル、リンク時に`-lmudflap`
- ポインタアクセスを行うコードの周辺にアクセスの正当性をチェックする命令を挿入し、チェックしてくれる
- Valgrindと組み合わせて使うと効果的にメモリ関連の脆弱性を検出できる

  `http://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging`

GCC: -D_FORTIFY_SOURCE

- GCCの”Automatic Fortification (自動要塞化)”機能

  gcc –O1 –D_FORTIFY_SOURCE=1 foo.c
- コンパイル時に明らかなオーバーフローのチェック
- ランタイムのオーバーフロー時にメッセージを出力してabort
- -D_FORTIFY_SOURCE=2 でより厳しいチェック

  printf, vfprintf, syslogに”%n”が渡されるとabort

- Red Hat Enterprise Linux5 では全てのプログラムのコンパイルに利用されている

  Defensive Programming for Red Hat Enterprise Linux (and What To Do If Something Goes Wrong)
  http://people.redhat.com/drepper/defprogramming.pdf

VC++: /GS

- リターン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知するメカニズム
  バッファとリターンアドレスの間にcookieを挿入し、上書きを検知する

- MSDN「コンパイラ セキュリティの徹底調査」
  http://msdn.microsoft.com/ja-jp/library/aa290051(VS.71).aspx

安全な文字列操作ライブラリを使用して脅威を緩和

SafeStr

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、文字列の内容を移動する。

したがって、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/

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。

XXLによるエラー処理

XXLとは

- 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
- 呼び出し側が例外処理しなくてはならない
- 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
  - メッセージを **stderr** に出力する
  - **abort()** を呼ぶ

アプリケーションはSafeStrとXXLの2つの外部ライブラリに依存しなくてはならなくなる。

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

まとめ

まとめ：基本対策

- null終端文字の取扱いに関するよくある間違いのパターンの把握と回避
- 仕様に一貫性のない文字列処理関数群を適切に使う
- プログラムの外部から取得した信頼できない文字列データは、入力値検査を行った上で適切に処理する
- コンパイルオプションの活用

まとめ：応用対策

バッファオーバーフローの検知・予防に有用なライブラリやツールの使用を検討する

- TR24731関数、**strlcpy(), strlcat()**
- C++ではなるべく **basic_string** で済ます
- Checked STLが使える環境では利用を検討
- 静的解析ツールを利用し、脆弱なコードを検出

CERT C Secure Coding Standardなどを参考に、コーディング規約を定め、みんなで守る

CERT C: 07. 文字と文字列 (STR)

- STR00-C. 文字の表現には適切な型を使用する
- STR01-C. 文字列の管理は一貫した方法で行う
- STR02-C. 複雑なサブシステムに渡すデータは無害化する
- STR03-C. null 終端バイト文字列を不注意に切り捨てない
- STR04-C. 基本文字集合にある文字を表すには単なる char を使用する
- STR05-C. 文字列リテラルの参照には const へのポインタを使用する
- STR06-C. strtok() が分割対象文字列を変更しないと想定しない
- STR07-C. TR 24731 を使用し、文字列操作を行う既存のコードの脅威を緩和する
- STR08-C. 新たに開発する文字列処理するコードには managed string を使用する
- STR30-C. 文字列リテラルを変更しない
- STR31-C. 文字データと null 終端文字を格納するために十分な領域を確保する
- STR32-C. 文字列は null 終端させる
- STR33-C. ワイド文字の文字列サイズは正しく求める
- STR34-C. 文字データをより大きなサイズの整数型に変換するときは事前に unsigned 型に変換する
- STR35-C. 長さに制限のないコピー元から固定長配列へデータをコピーしない
- STR36-C. 文字列リテラルで初期化される文字配列のサイズを指定しない
- STR37-C. 文字処理関数への引数は unsigned char として表現できなければならない

https://www.jpcert.or.jp/sc-rules/

もくじ

1. C/C++における文字列
2. 文字列使用時に犯しやすい誤りと基本対策
3. 脅威の緩和方法
4. まとめ
5. **バッファオーバーフローの脆弱性**
   - バッファオーバーフロー概要
   - 攻撃例の解説

バッファオーバーフローの脆弱性

JPCERT CC®

基礎知識

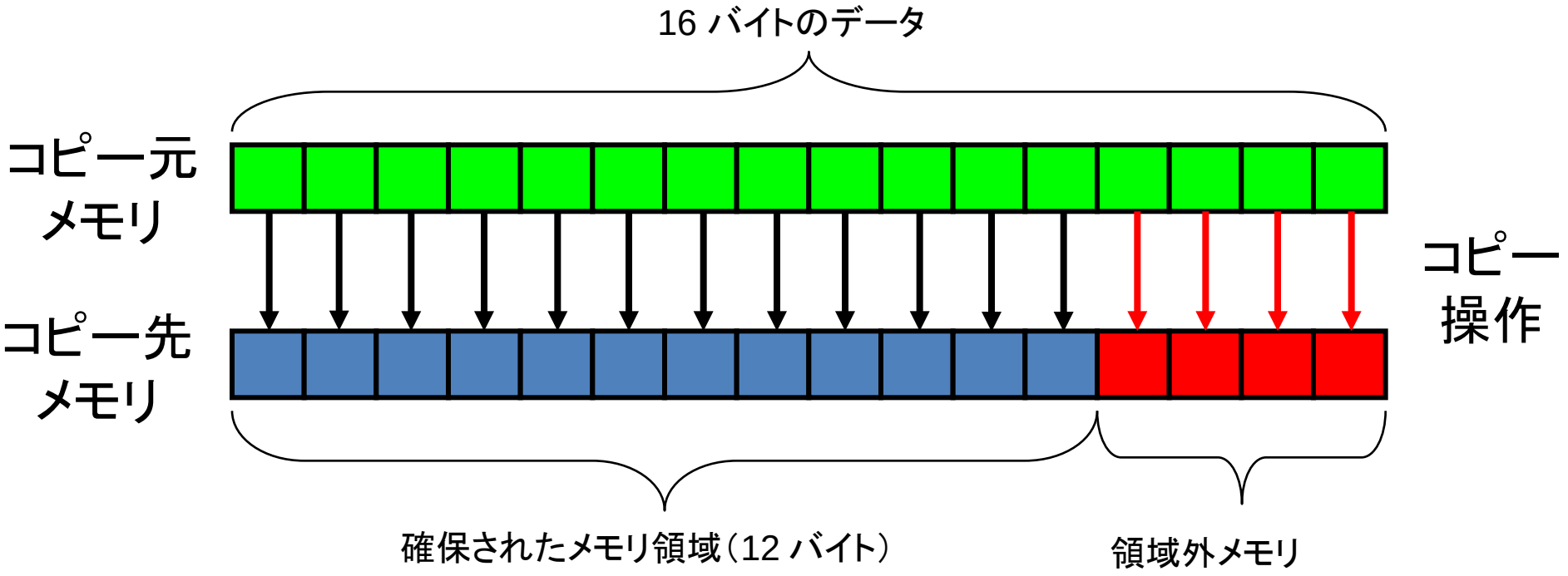
- バッファオーバーフローの概要
- スタックの理解

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

バッファオーバーフローとは？

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。



なぜバッファオーバーフローが発生するのか？

言語仕様的側面

- 誤りを犯しやすい `null` 終端バイト文字列
- 境界検査を強制しない文字処理標準ライブラリの存在
- そもそも文字列をプログラマがマネージしなくてはならない言語仕様上、セキュリティはプログラマの腕次第

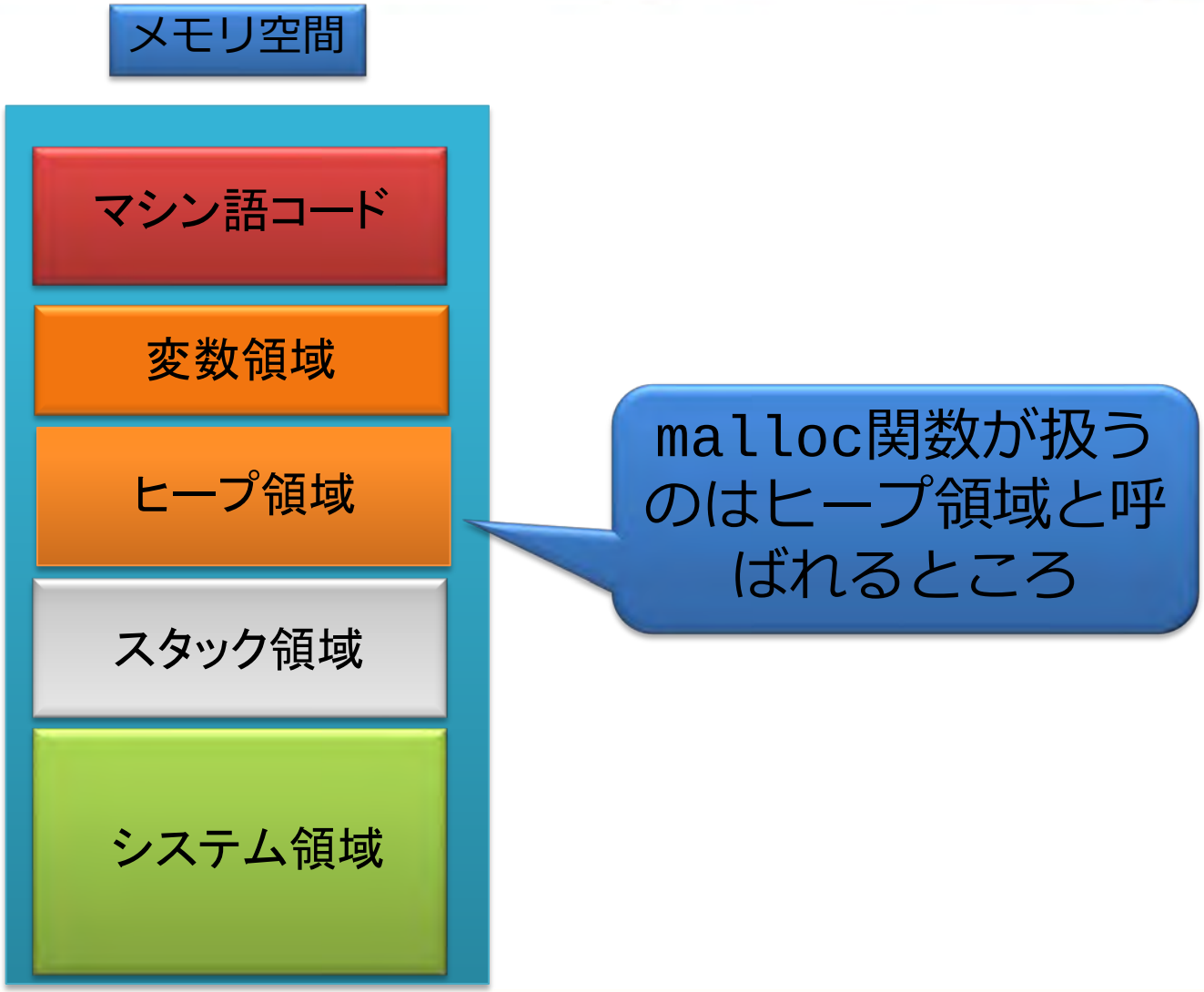
人的側面

- 仕様の認識が不十分で、意図せず関数を誤用
- 入力文字列の検証が不十分(不在)、やり方がマズい
- そもそも脅威に対する認識が不十分

バッファオーバーフローの特徴

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

プロセスがアクセスできるメモリ空間



JPCERT CC®

基礎知識

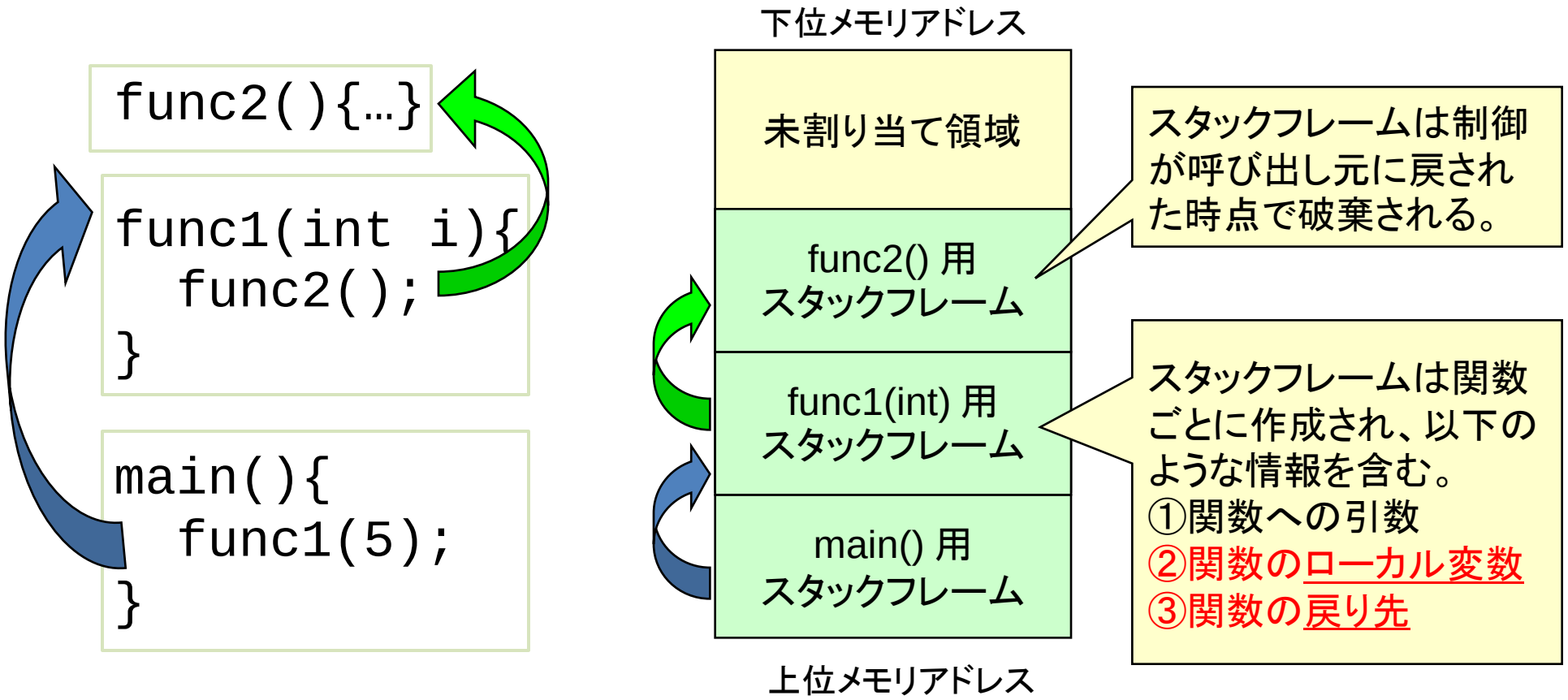
- バッファオーバーフローの概要
- スタックの理解

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

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。



スタックが更新されるタイミング

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破棄される（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。

アセンブラで見る関数の呼び出しと、スタックの状態遷移

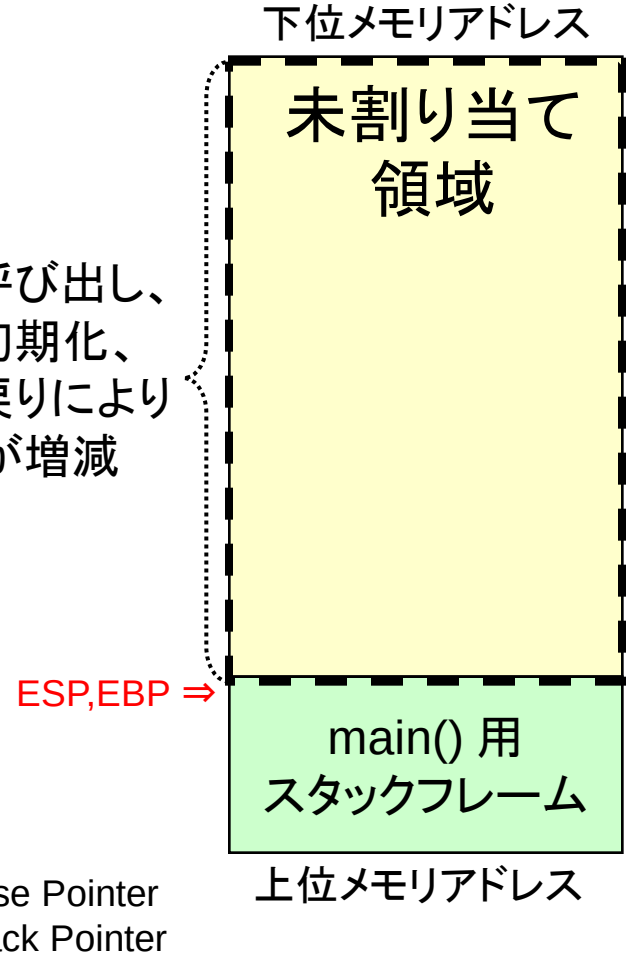
JPCERT CC®

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
}
```



EBP : Extended Base Pointer
ESP : Extended Stack Pointer

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

```
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
```

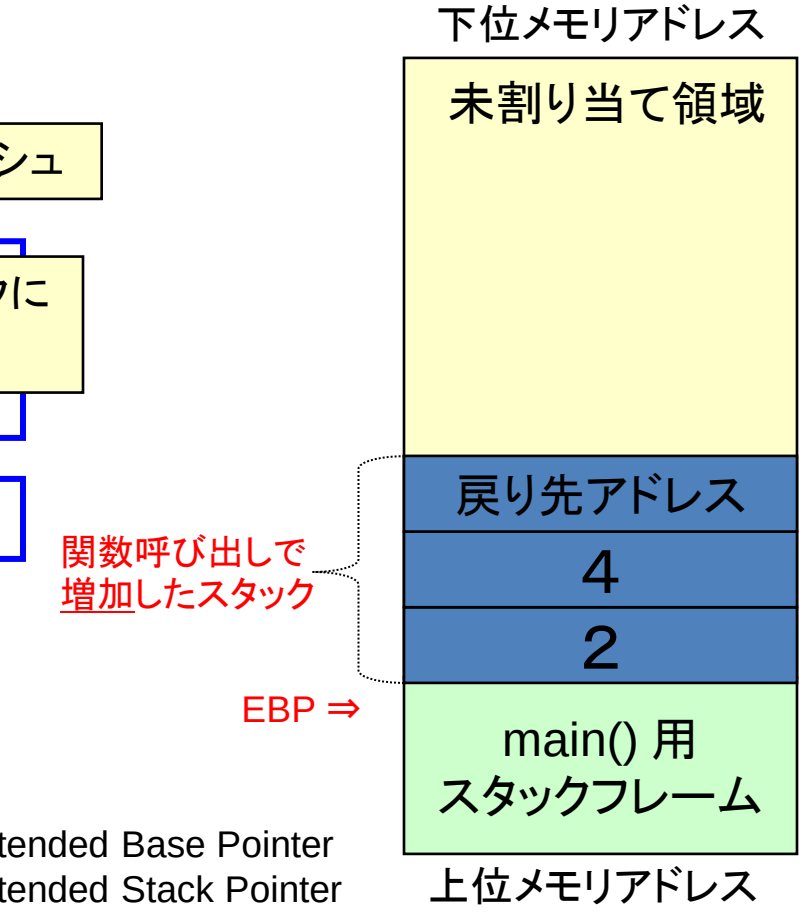
最初の引数をスタックにプッシュ

```
push 4
```

```
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



EBP : Extended Base Pointer
ESP : Extended Stack Pointer

Intel表記

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];
```

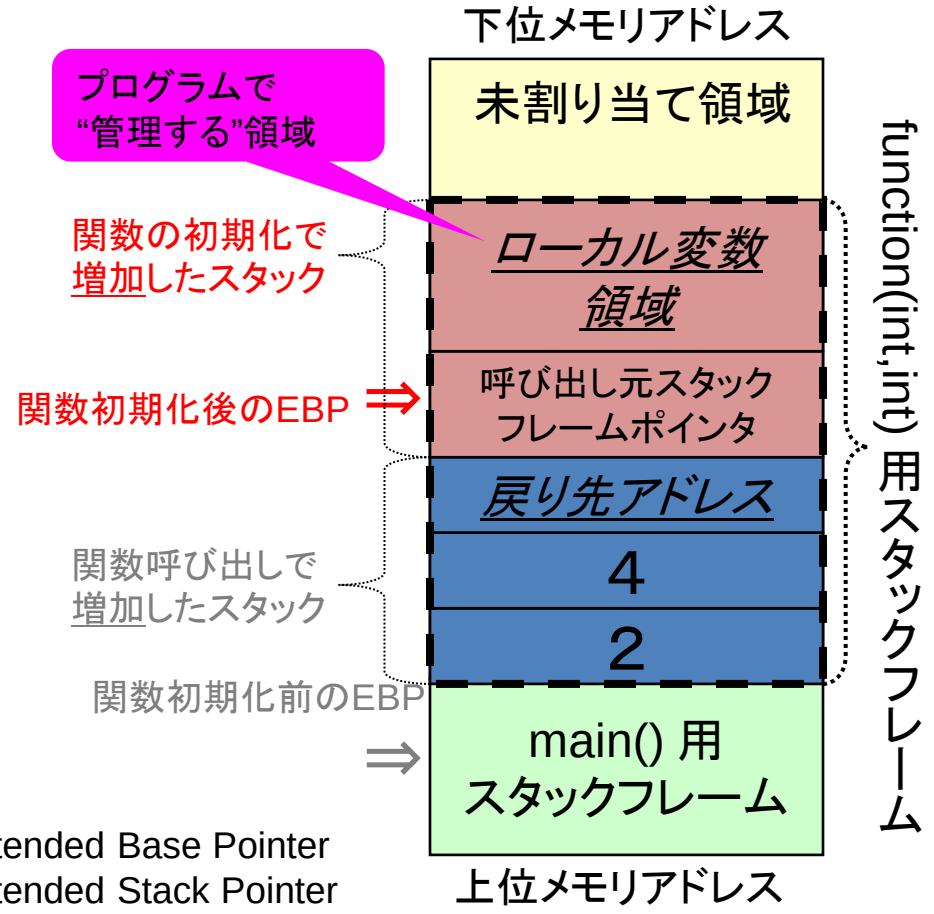
```
push ebp
mov ebp, esp
sub esp, 44h
```

フレームポインタを保存する。

現在のスタックポインタを呼び出された関数のフレームポインタに設定する。

ローカル変数のための領域（44h=68バイト）を確保する。

※“ESP”は常に最上位の有効なスタックを示す



EBP : Extended Base Pointer
ESP : Extended Stack Pointer

関数の戻り(1/2) 戻り先の取り出しと制御の戻し

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

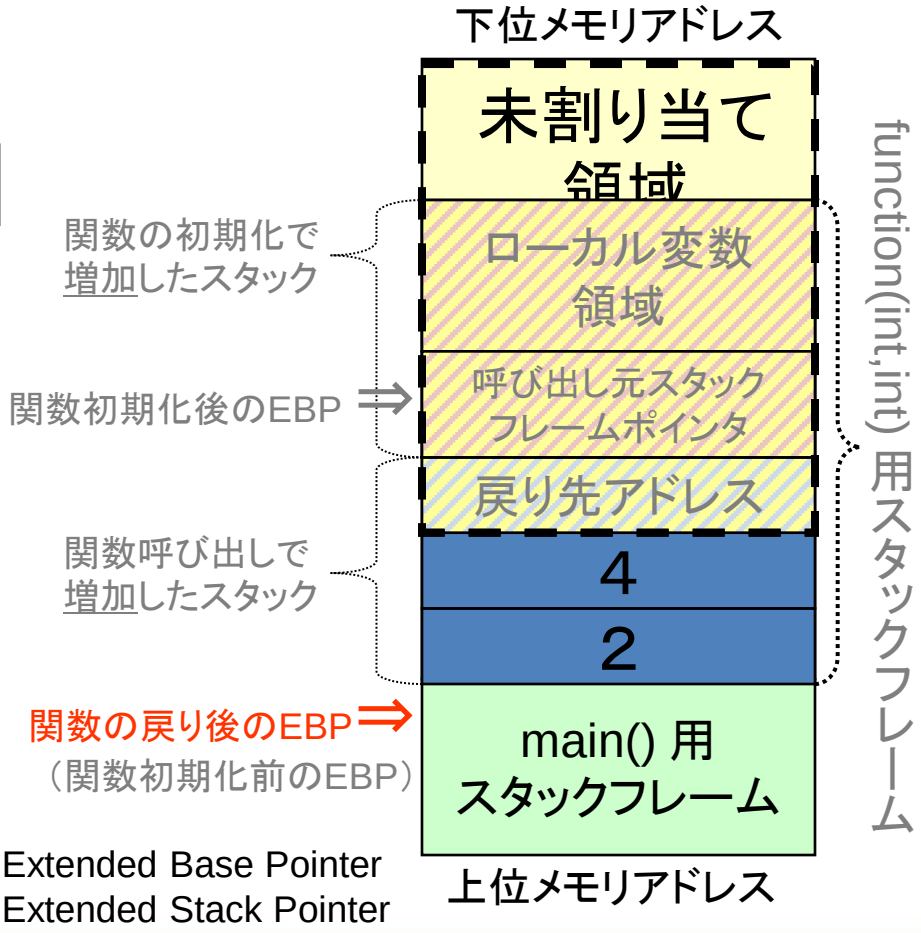
スタックポインタを復帰する。

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

フレームポインタを復帰する。

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。

※"ESP"は常に最上位の有効なスタックを示す



EBP : Extended Base Pointer
ESP : Extended Stack Pointer

関数の戻り(2/2) 関数の引数領域の解放

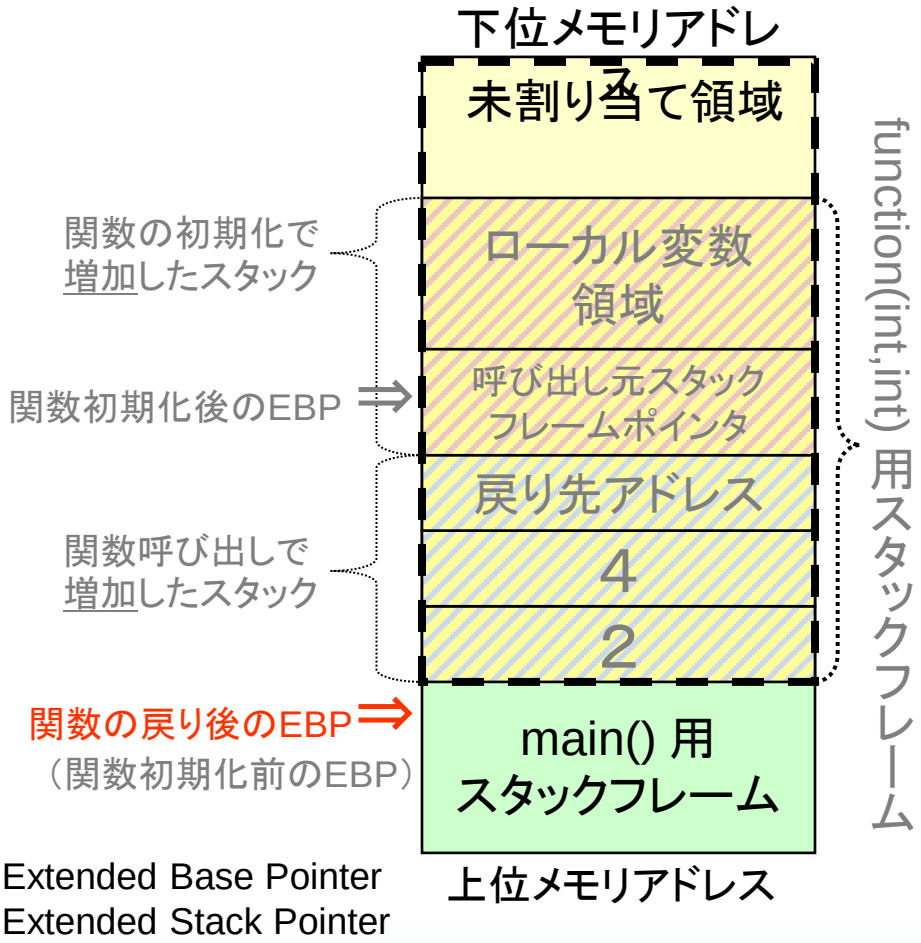
```
main(){
  function(4, 2);
```

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。

※“ESP”は常に最上位の有効なスタックを示す



EBP : Extended Base Pointer
ESP : Extended Stack Pointer

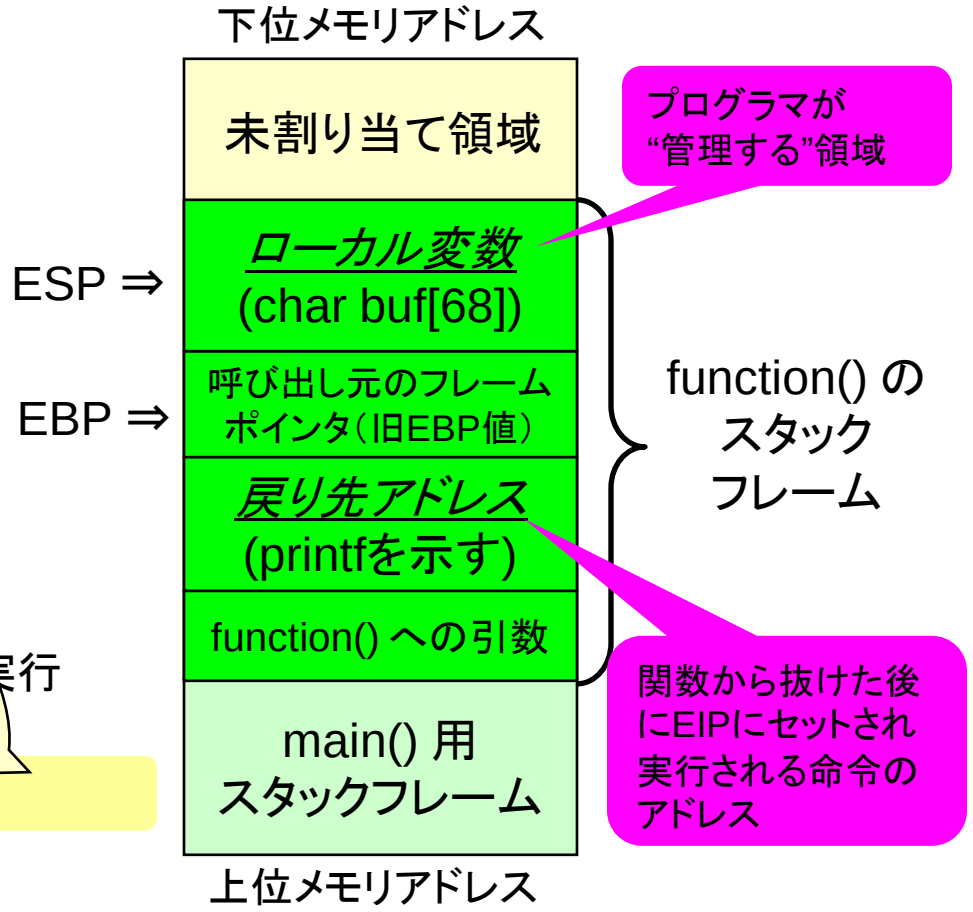
まとめ：呼び出された関数が実行中のスタックの状態

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



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）

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

脆弱なサンプルプログラム：BufferOverflow.c

JPCERT CC®

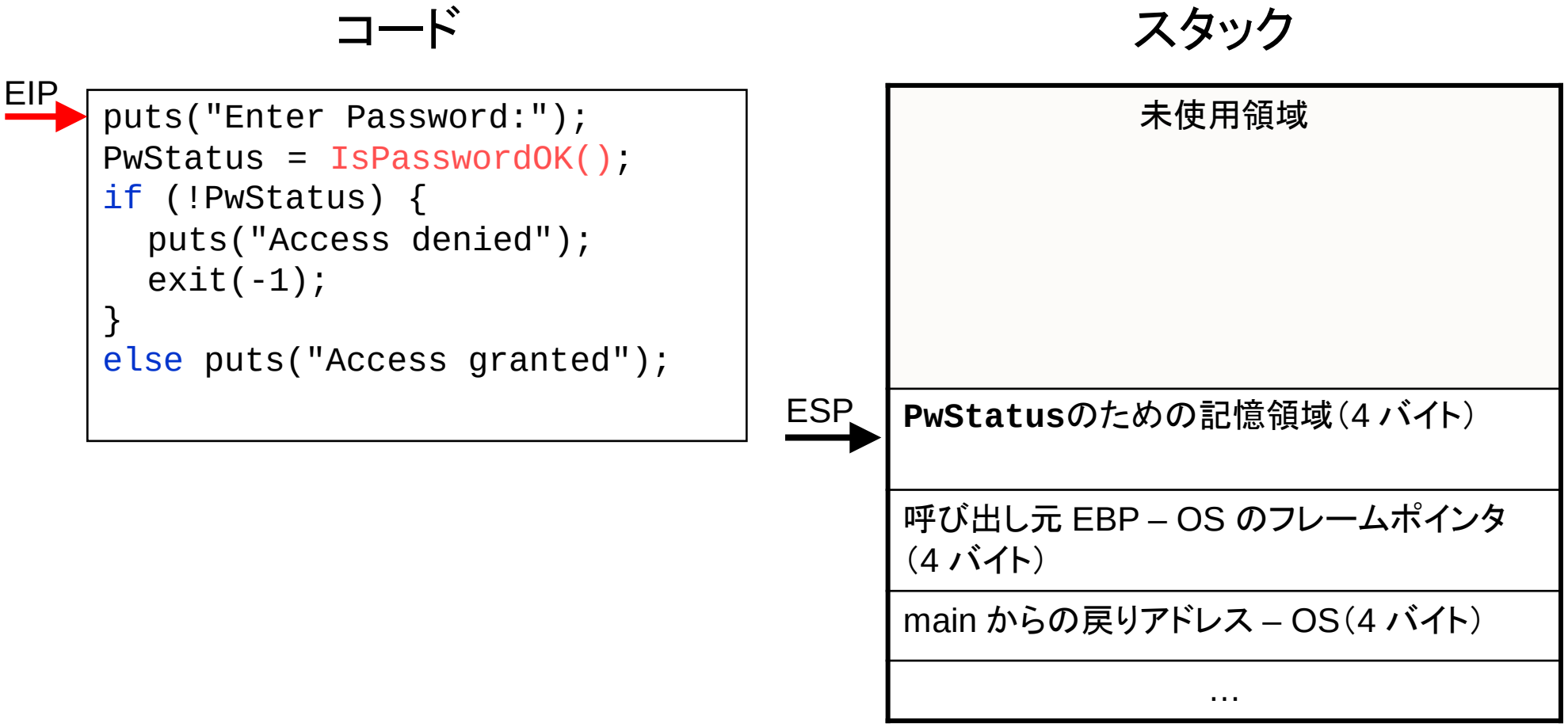
```
bool IsPasswordOK(void) {
  char Password[12];
  gets(Password); //stdin(キーボード)から入力を取得
  if (!strcmp(Password,"goodpass")) return(true); //パスワードが正しい
  else return(false); //パスワードが無効
}

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
  if (!PwStatus) {
    puts("Access denied");
    exit(-1); //プログラムを終了
  }
  else puts("Access granted");
}
```

サンプルプログラムの実行

実行例 1　正しいパスワード

実行例 2　不正なパスワード

IsPasswordOK()呼び出し前のスタック



IsPasswordOK()呼び出し中のスタック

JPCERT CC®

コード

EIP

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

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

スタック

ESP

Passwordのための記憶領域（12 バイト）
呼び出し元 EBP – main のフレームポインタ（4 バイト）
呼び出し元の戻りアドレス – main（4 バイト）
PwStatusのための記憶領域（4 バイト）
呼び出し元 EBP – OS のフレームポインタ（4 バイト）
mainからの戻りアドレス – OS（4 バイト）
…

IsPasswordOK()呼び出し後のスタック

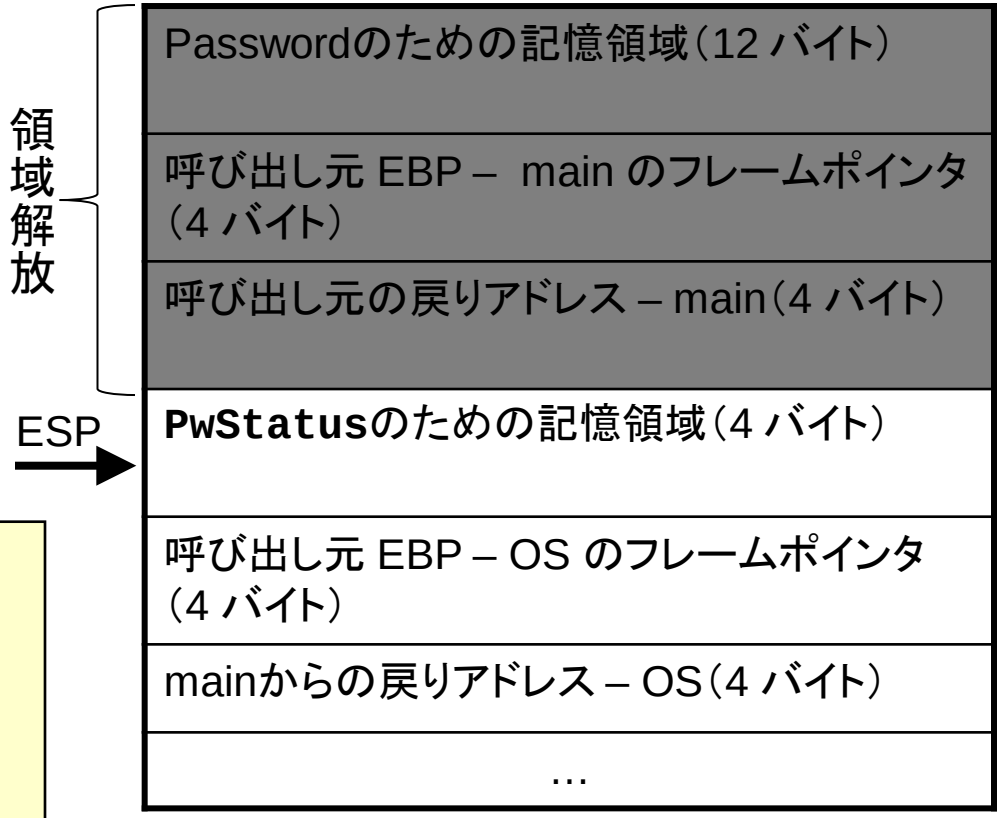
コード

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



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

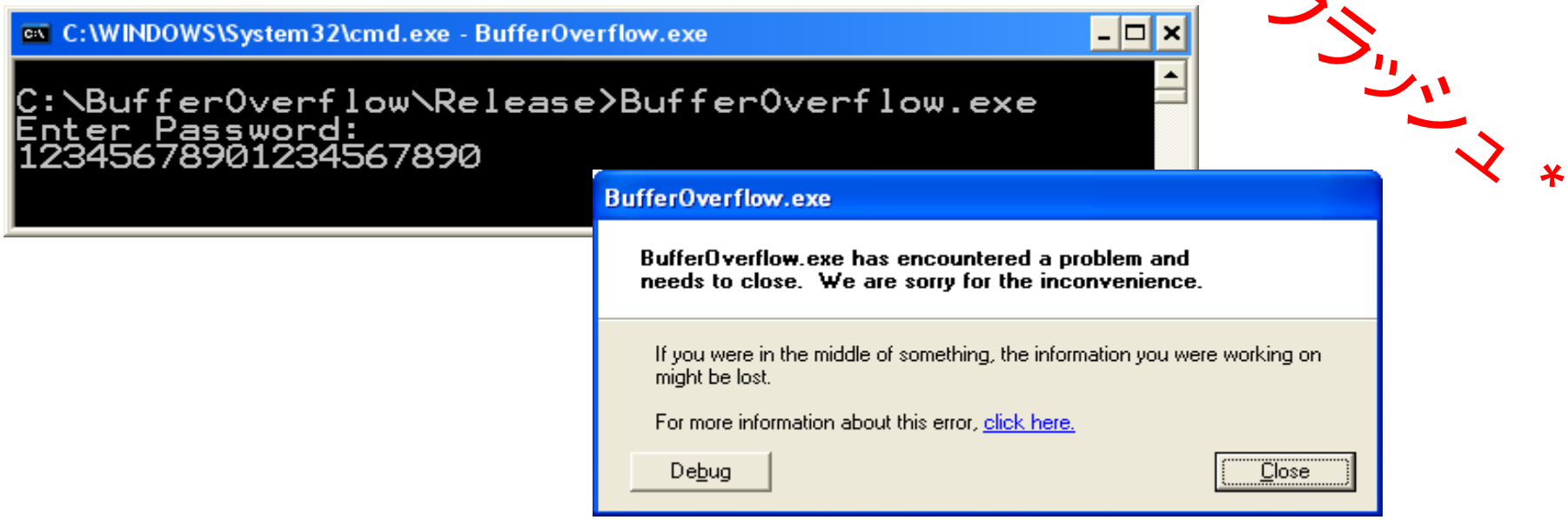
サンプルによる解説

- 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

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・・・



スタック破壊の仕組み

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

EIP

ESP

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。

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

バッファオーバーフローを利用した実行パス操作の攻撃例

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？

何が起きたのか？

“123456789012345 6j▸*!” 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からの戻りアドレス – OS（4 バイト）

書き換え

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）

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

コードインジェクションの概要

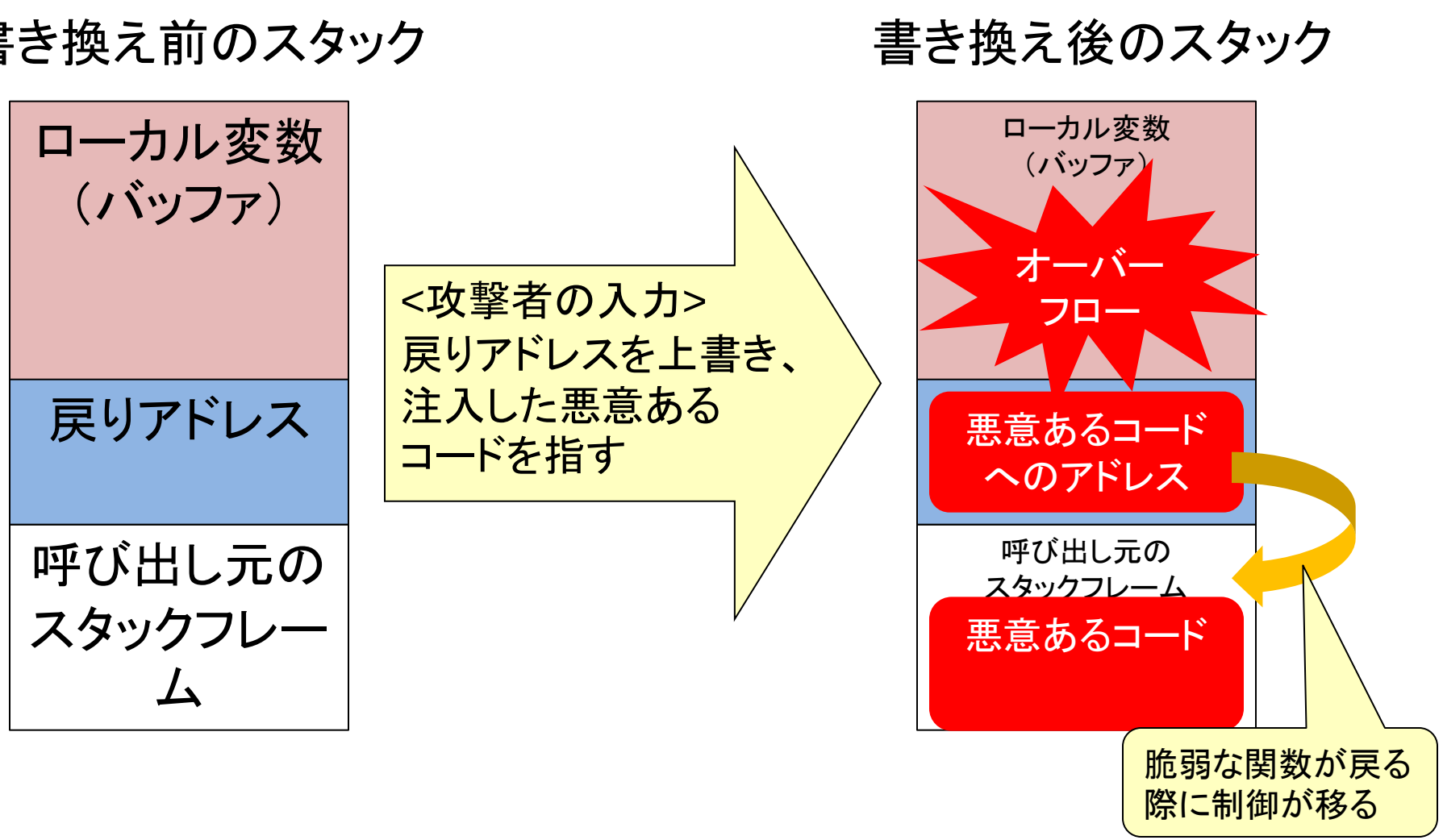
JPCERT CC®

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る時に、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は攻撃価値が高い。）

コードインジェクションの概要図



./BufferOverflow < exploit.bin

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  "12345678901234567"
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。

悪意のある引数の条件

1. 脆弱なプログラムに正当な入力として受け入れられるものであること

2. 制御可能な他の入力と組み合わせ、結果として悪意のあるコードを実行できること

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ないこと

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  "789012345678à· +"
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。

悪意のある引数の解析2

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）

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）

悪意のあるコード

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）

シェルコードの例

```
xor %eax,%eax                  #eaxに 0 を設定
mov 0xbffff9ff,%eax            #null ワードに設定
mov %al,$0xb          #execveのコードを設定
mov %ebx,$0xbffffa03           #第 1 引数のポインタ
mov %ecx,$0xbffff9fb           #第 2 引数のポインタ
mov %edx,0xbffff9ff            #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”
```

シェルコード：ゼロの生成

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 0xbffff9ff,%edx        #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。

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 0xbffff9ff,%eax        #null ワードに設定
mov %al,$0xb       #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。

シェルコード：システムコールへの引数設定

…

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

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。

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。

シェルコード：システムコールの実行

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

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

アークインジェクション（return-into-libc）

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

脆弱なプログラム

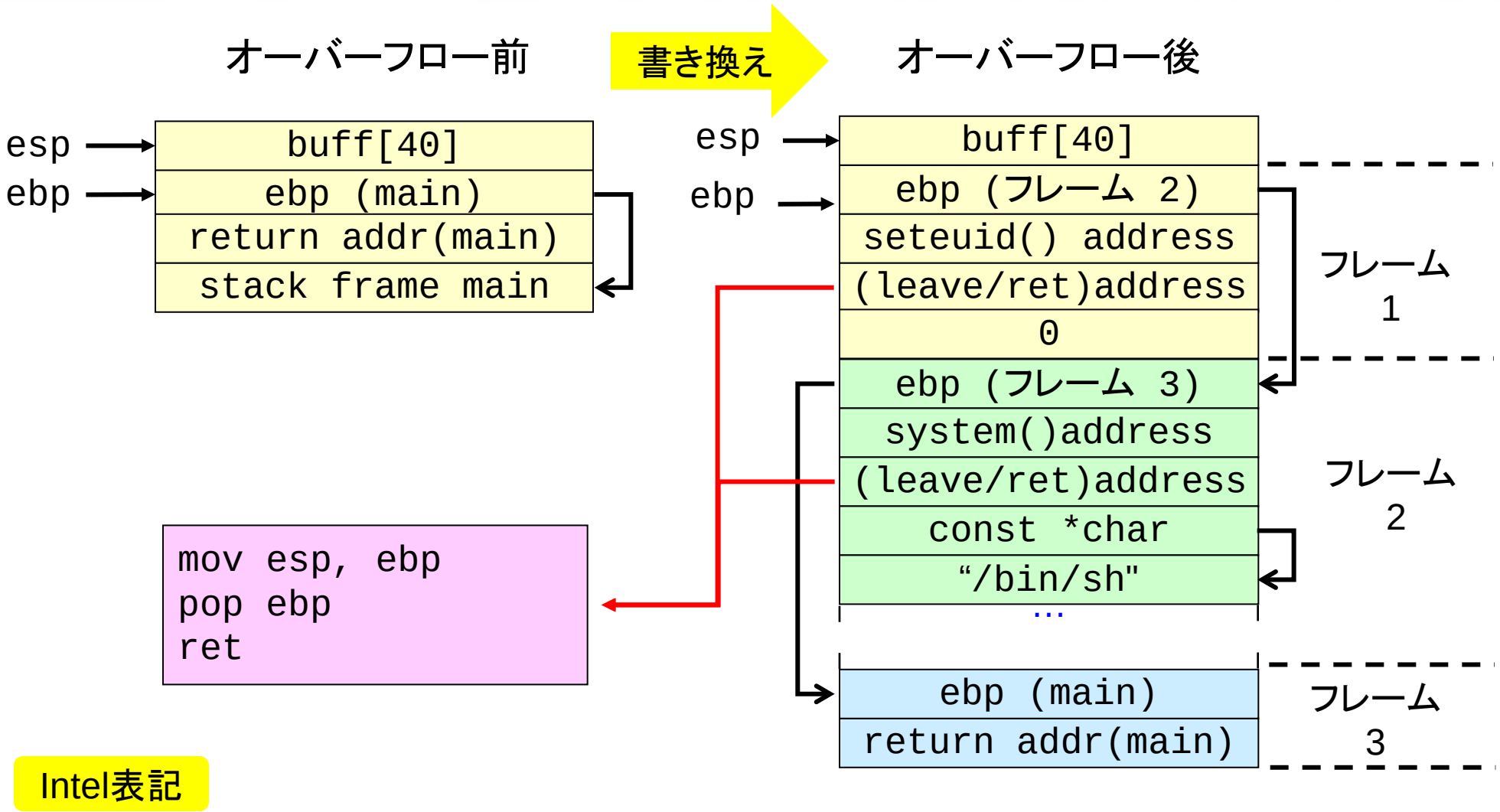
JPCERT CC®

```
#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;
}
```

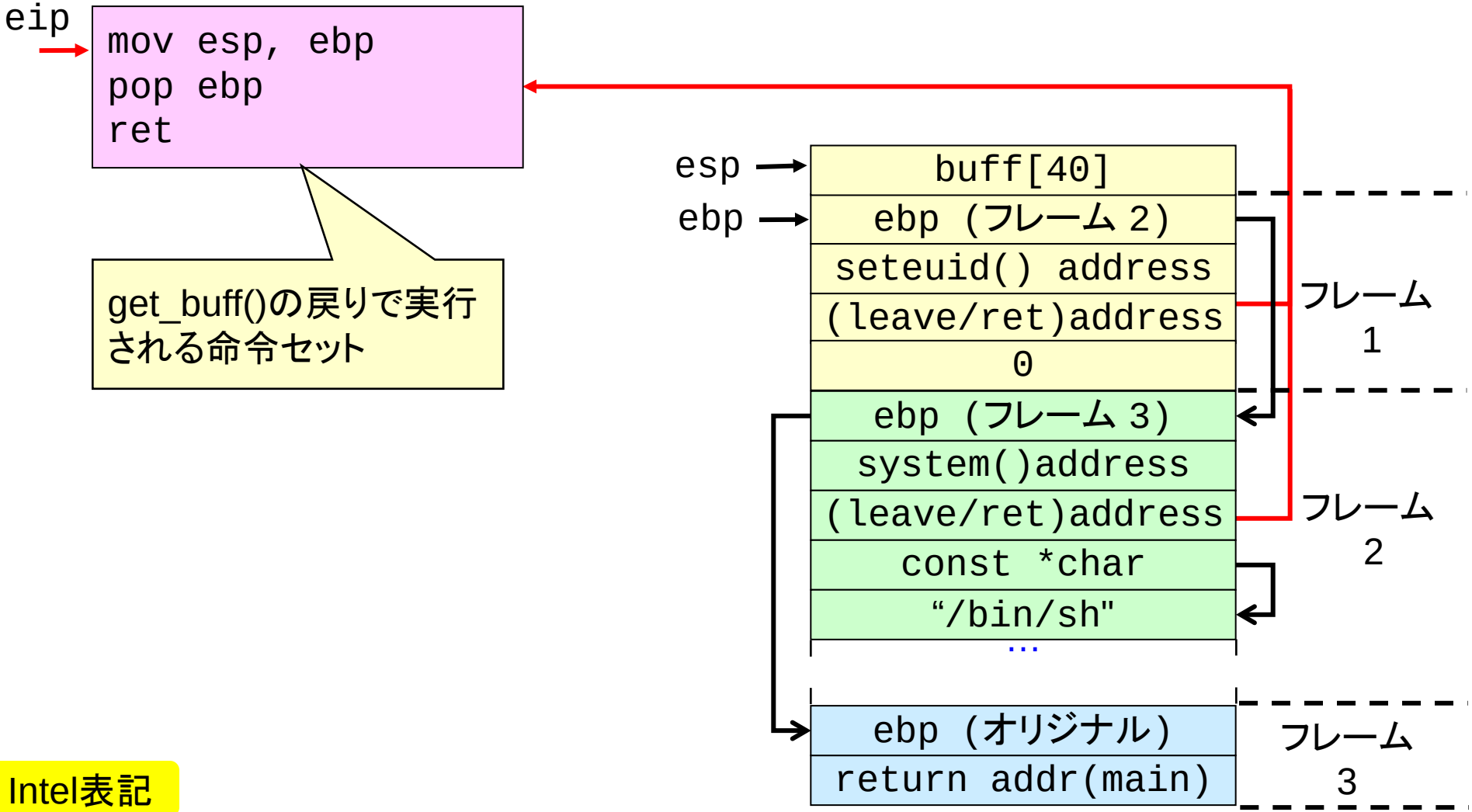
アークインジェクションによる攻撃コード

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

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

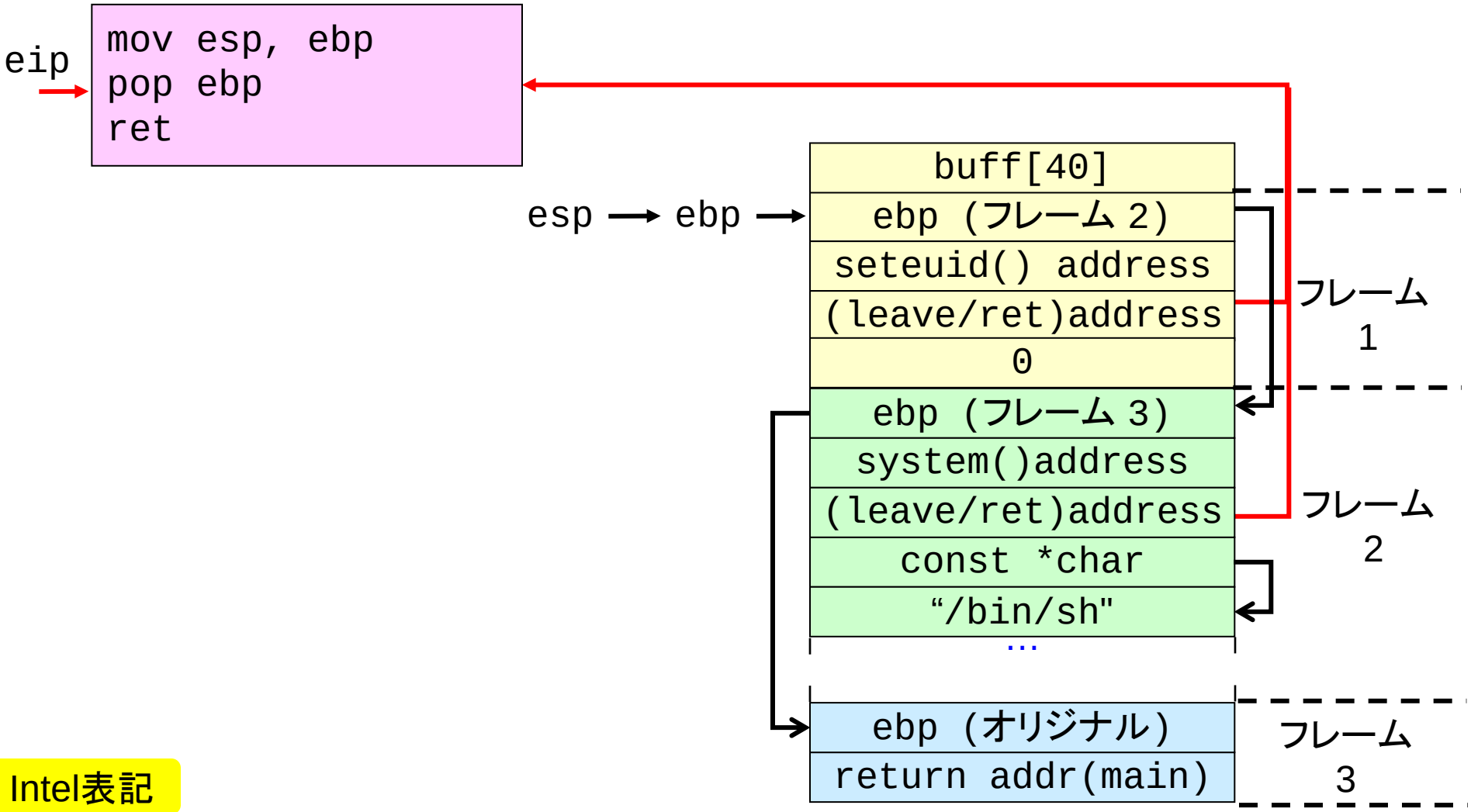


get_buff() の戻り



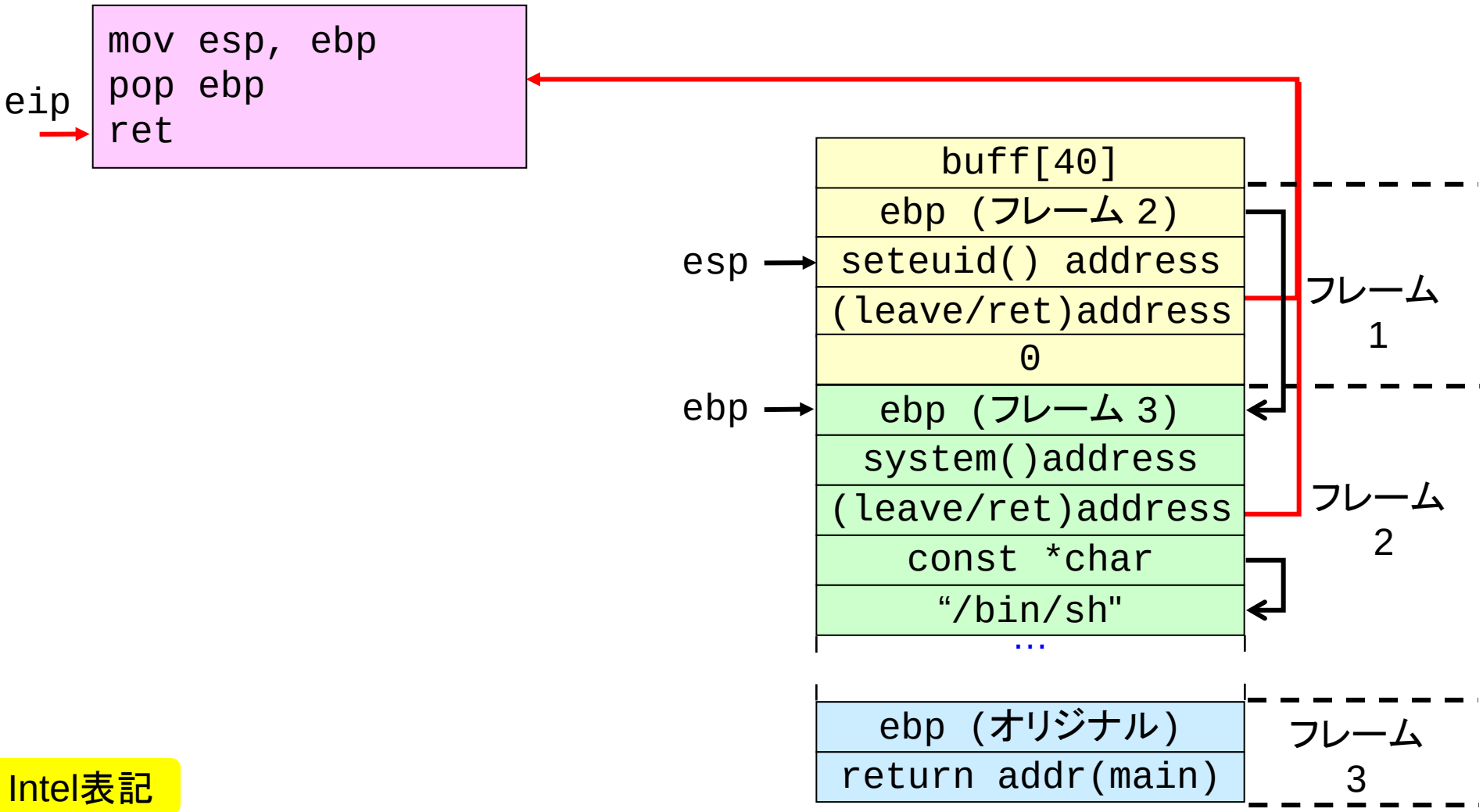
Intel表記

get_buff() の戻り



Intel表記

get_buff() の戻り

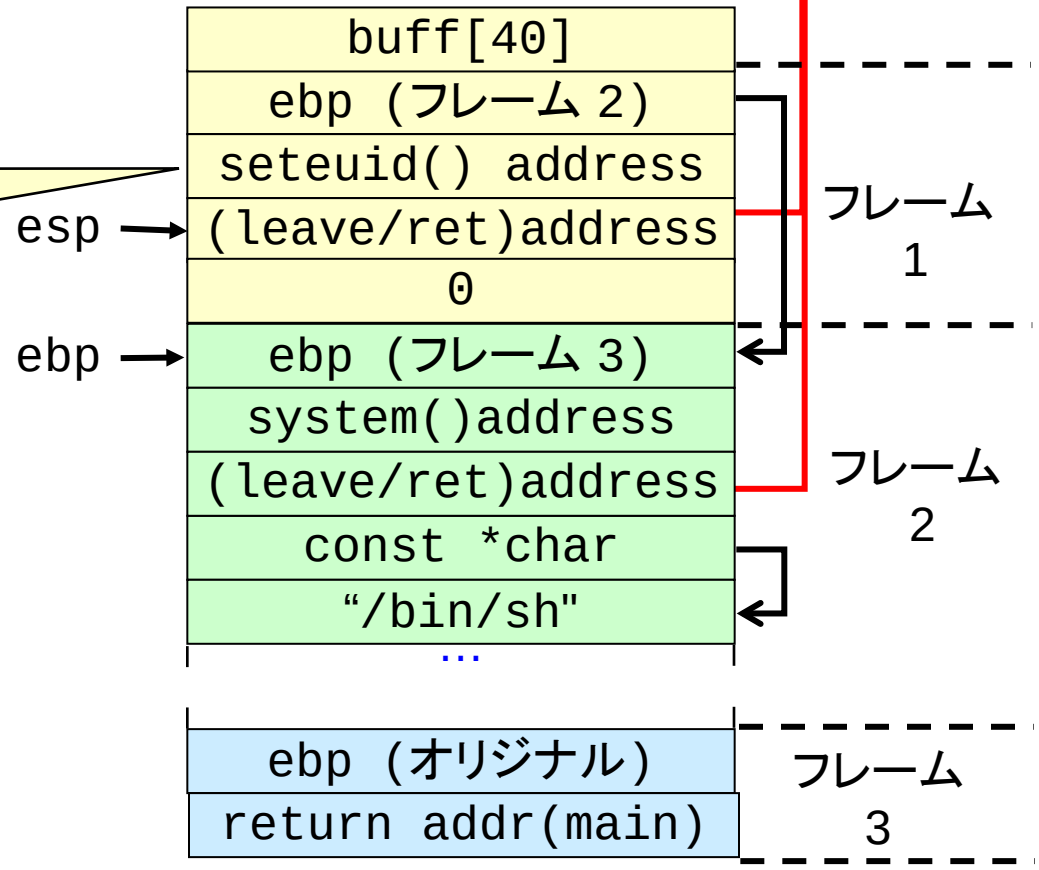


Intel表記

get_buff() の戻り

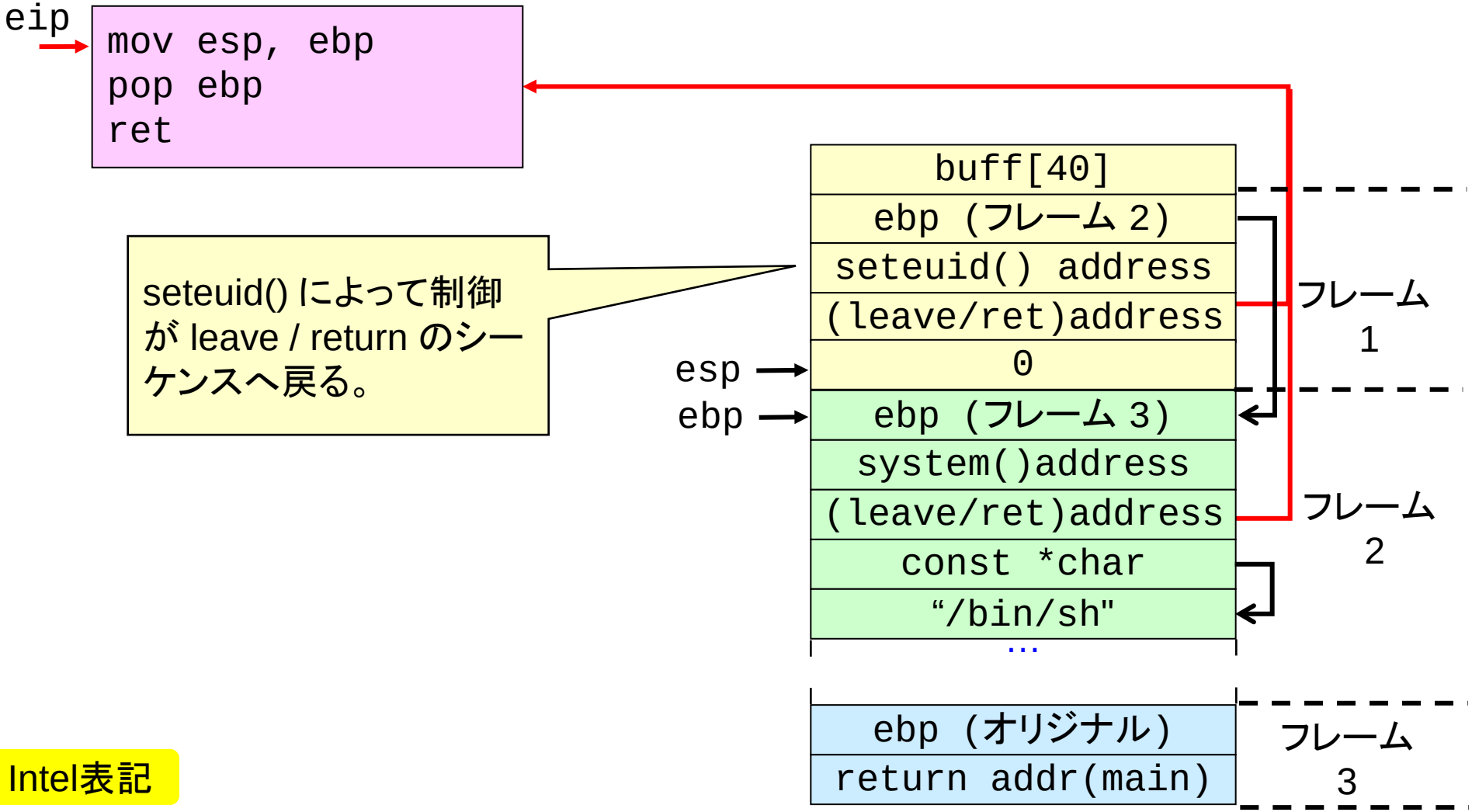
```
mov esp, ebp
pop ebp
ret
```

ret 命令によって制御が seteuid() へ移る。



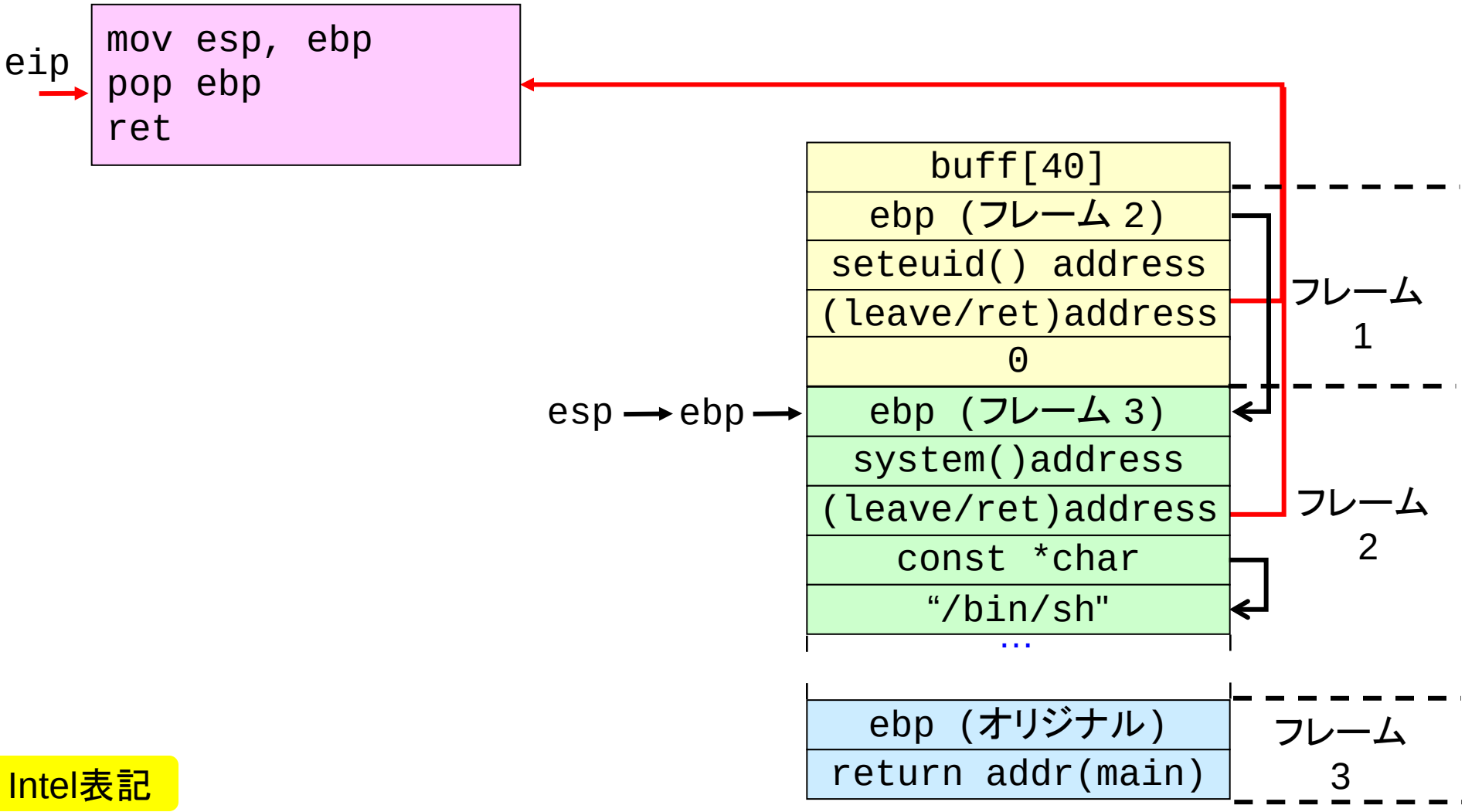
Intel表記

seteuid() の戻り



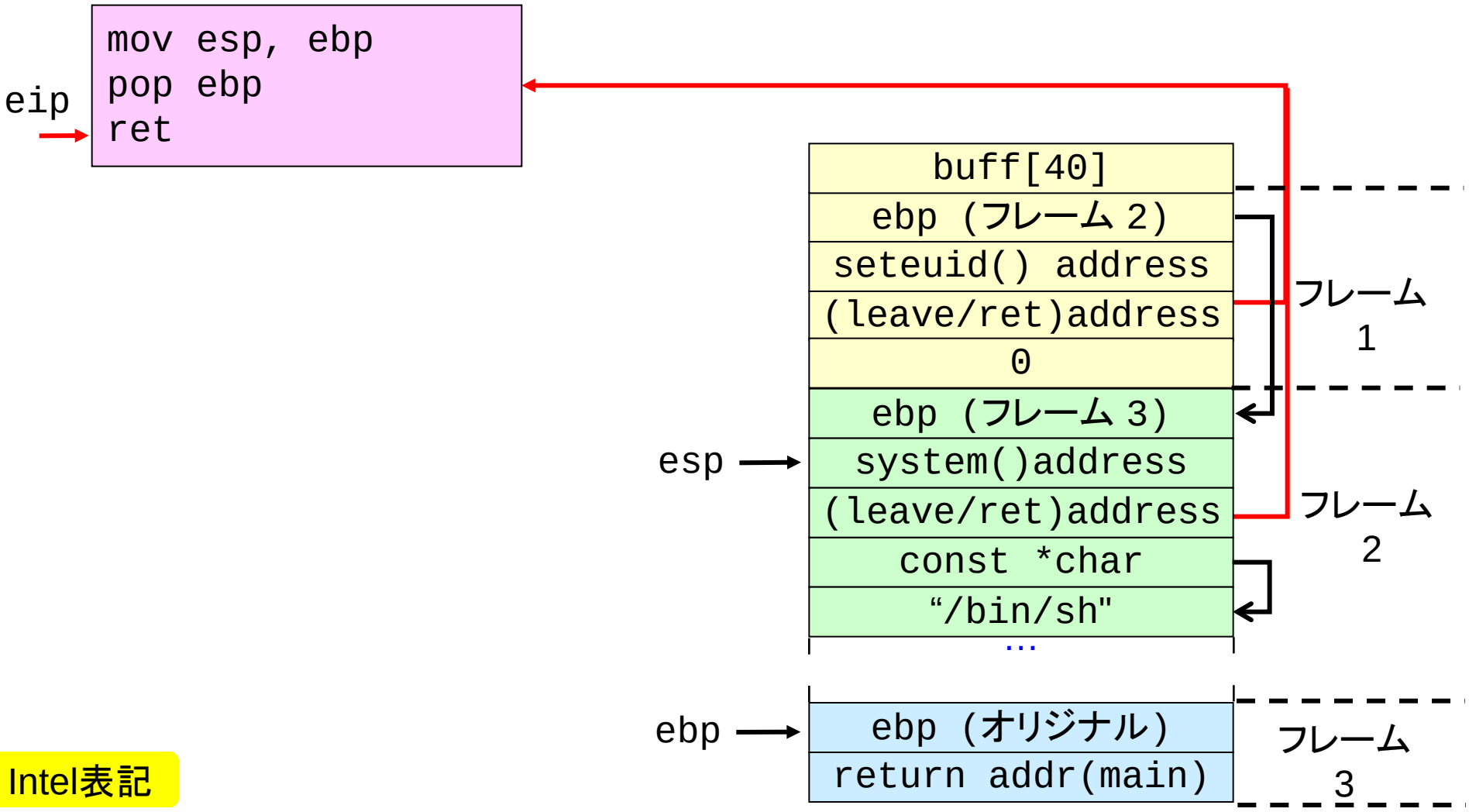
Intel表記

seteuid() の戻り



Intel表記

seteuid() の戻り

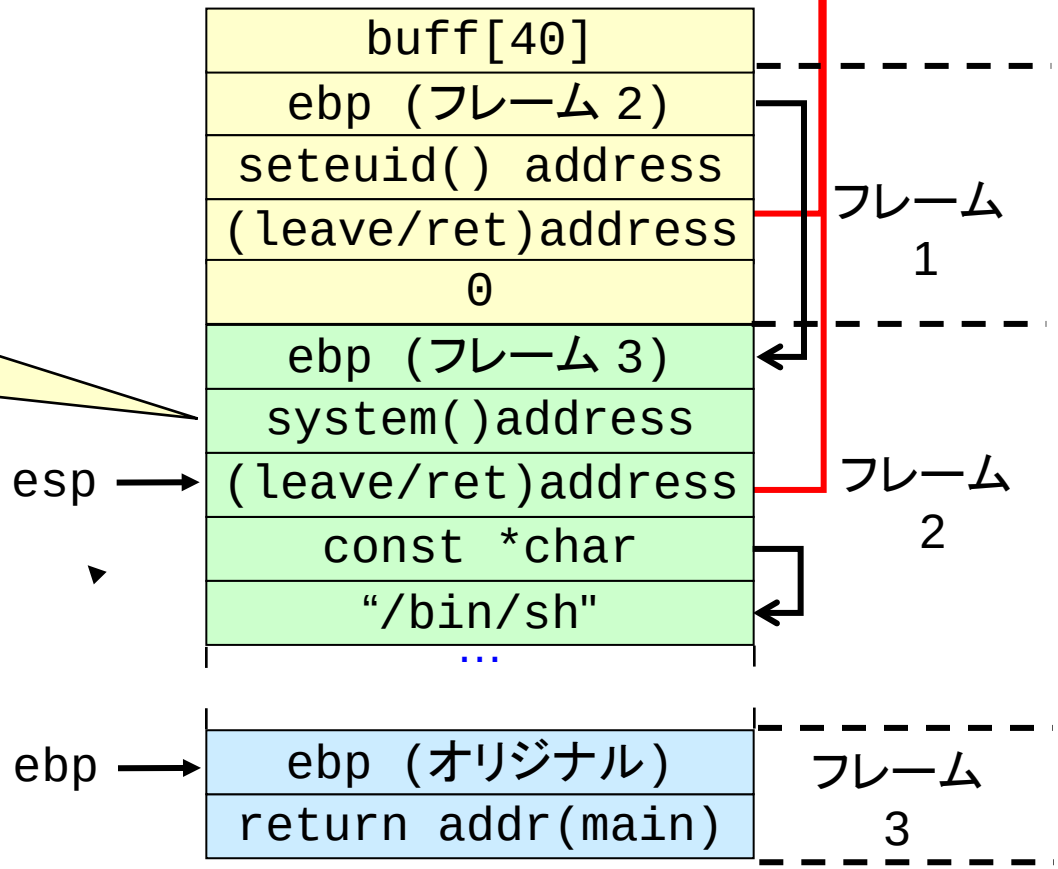


Intel表記

seteuid() の戻り

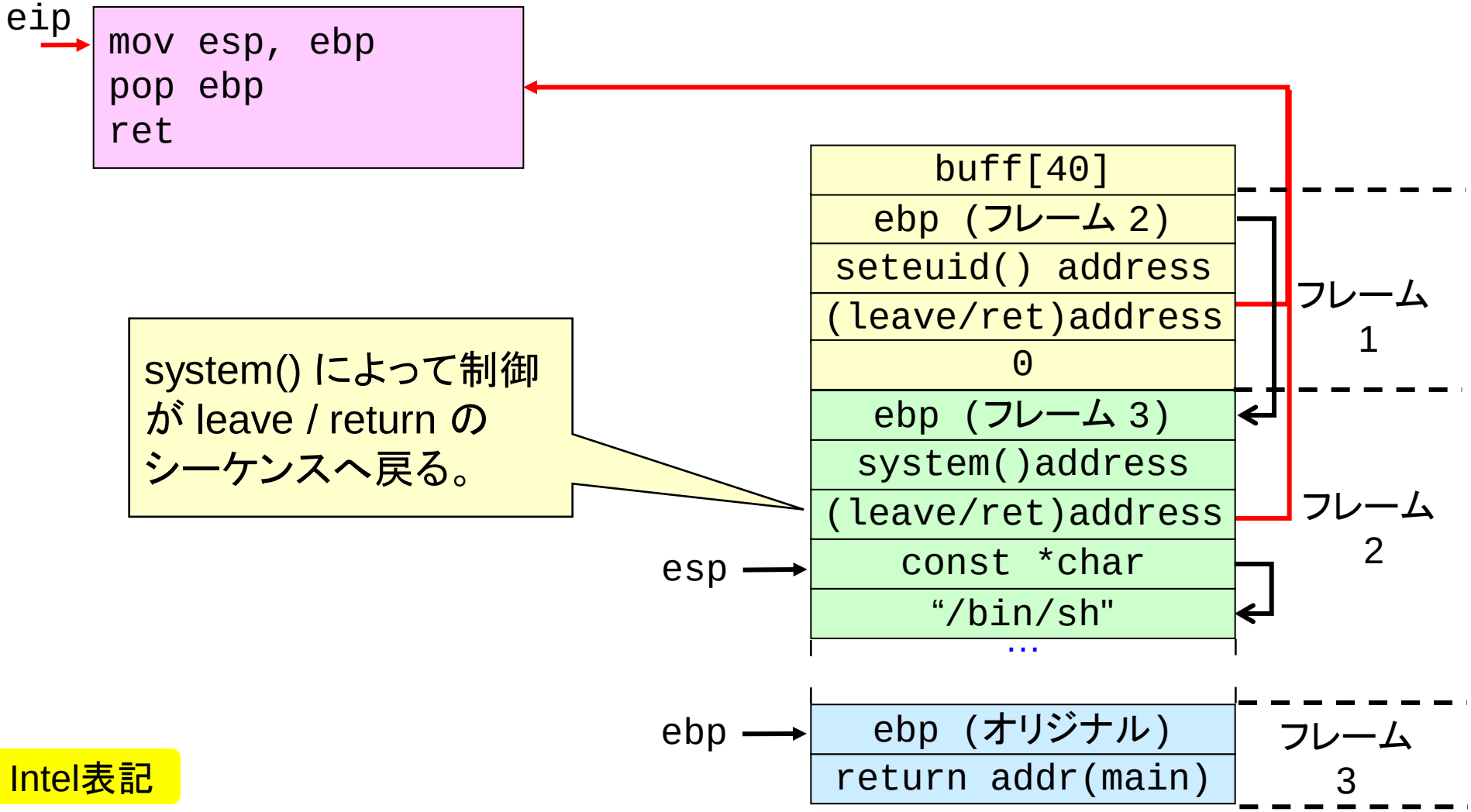
```
mov esp, ebp
pop ebp
ret
```

ret 命令によって制御が system() へ移る。

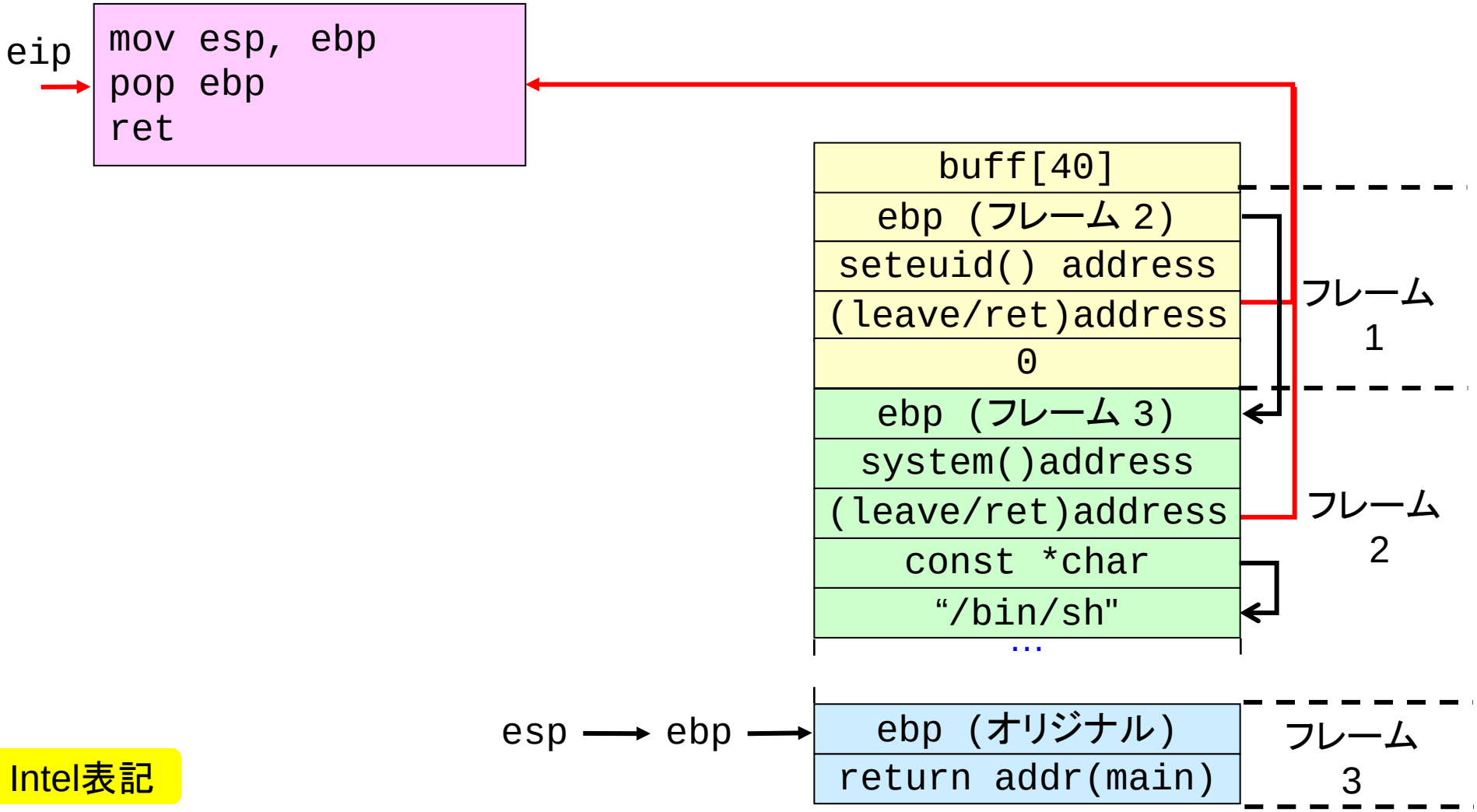


Intel表記

system() の戻り

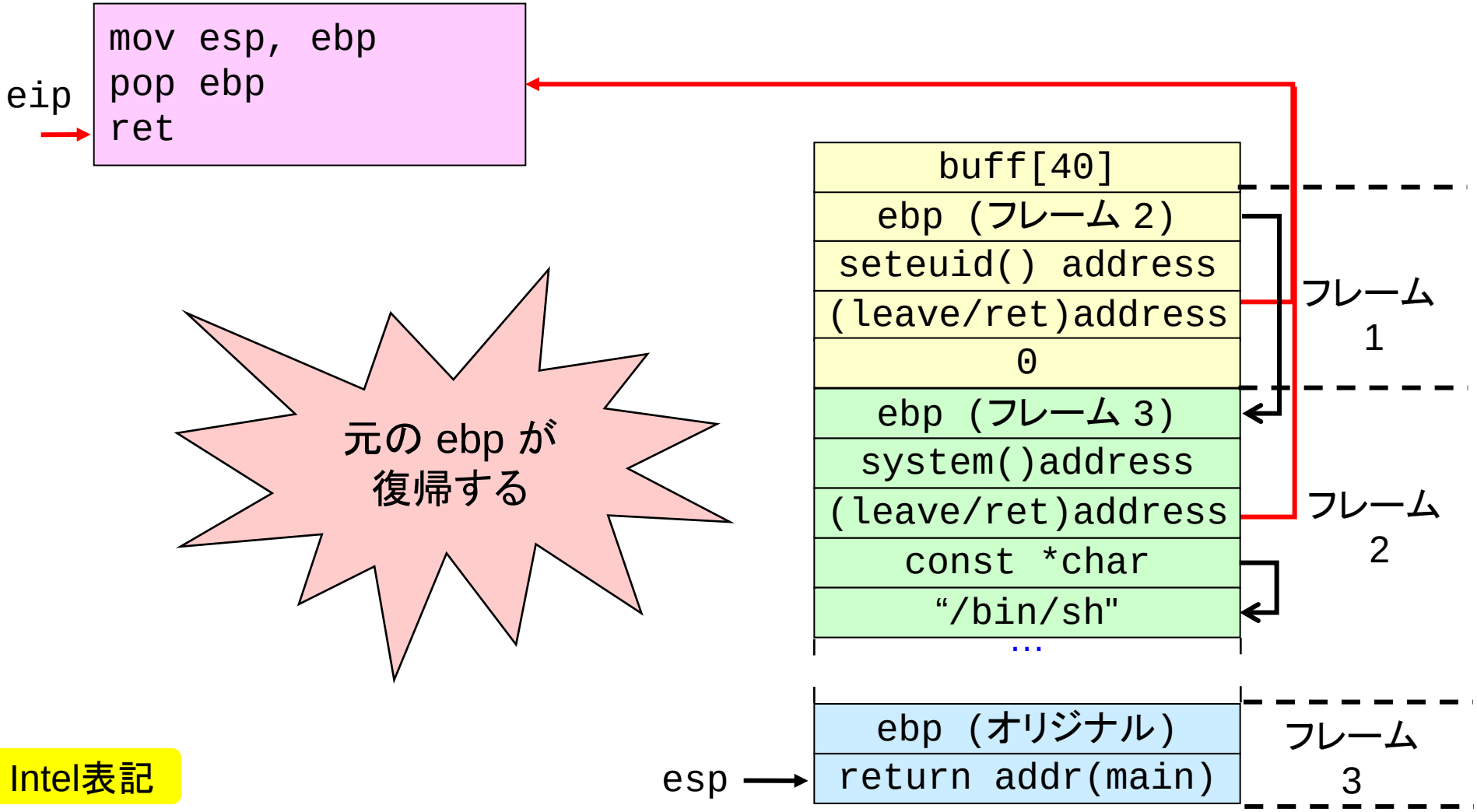


system() の戻り



Intel表記

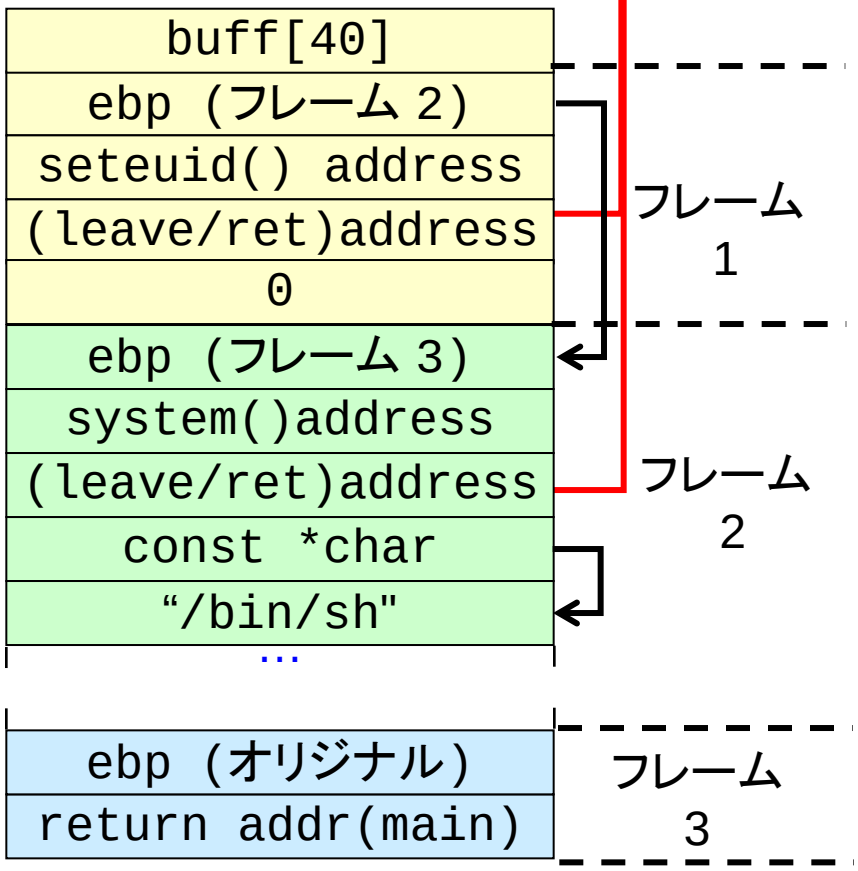
system() の戻り



Intel表記

system() の戻り

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



Intel表記

なぜこれが問題なのか？

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。

参考

「無制限」文字列処理関数

- 「無制限」 – unbounded
  - データコピー時に格納先バッファサイズを考慮しない
  - ソースデータの文字列長が格納先バッファより大きいとバッファオーバーフロー

- これらは、利用すべきでない関数群
  - コーディング規約等で使用を禁じるべき
  - 静的解析ツールはシステマチックにフラグをあげる

scanf()系

- int scanf(cont char *format, …);
  - stdinから文字を読み、制御文字列 format に従って変換

```
char v[128], val[15*1024], …, boundary[128], buffer[15*1024];
…
for(;;) {
  …
  // if the variable is followed by ‘; filename=“name”’ it is a file
  inChar = getchar();
  if (inChar == ‘;’) {
    …
    // scan in the content type if present, but simply ignore it
    scanf(“ Content-Type: %s “, buffer);
```

w3-msql 2.0.11から抜粋した脆弱なコード

sprintf()系

- int sprintf(char *str, const char *format, …);
  - 結果を文字列strに格納し、最後にヌル文字を追加する
- 格納先バッファがすべての引数の変換結果を格納できるようプログラマが保証しなくてはならない

```
char speed[128];
…
sprintf(speed, “%s%d”, (cp = getenv(“TERM”)) ? CP : “”,
       (def_rspeed > 0) ? def_rspeed : 9600);
```

Kerberos 5 v1.0から抜粋した脆弱なtelnet daemonのコード

strcpy()系

- char * strcpy(char *dst, char *src);
  - 文字列dstに文字列srcの内容をコピー

```
char *FixFilename(char *filename, int cd, int *ret) {
…
char fn[128], user[128], *s;
…
s = strrchr(filename, '/');
if(s) {
   strcpy(fn, s+1);
```

PHP/FI 2.0beta10 の php.cgiから抜粋した脆弱なコード

「制限付き」文字列処理関数

- 「制限付き」– bounded
  - 格納先バッファサイズを引数にとる
  - サイズ引数の値を適切に指定しない限り、バッファオーバーフローは発生しうる
  - なまじサイズを指定しているだけに、正しいサイズを指定しているかのコードレビューしないとダメ。しかし、レビューはシステマチックに行える

☞制限付き(サイズ引数をとる)関数を使えば自動的にコードがセキュアになるわけではない！

snprintf()

- int snprintf(char *dst, size_t n, char *fmt, …)
  - WindowsとUNIXで挙動が違う
  - Windows: バッファのサイズが足りない場合、-1を返す. null終端は保証されない
  - UNIX: null終端を保証する. 返り値は、十分なバッファ領域が確保されていたと仮定した書き込み文字数

strncpy()

JPCERT CC®

対策ポイント
- 格納先バッファサイズに基づく安全な「制限」
- strncpy()呼び出し後、手作業でヌル終端

```
if (strncmp(status, “200 OK”, 6))
{
   /* It’s not valid. Kill this off. */
   char temp[32];
   const char *c;

   /* Eww */
   if ((c = strchr(status, ‘¥r’)) || (c = strchr(status, ‘¥n’)) ||
       (c = strchr(status, ‘¥0’)))
   {
       strncpy(temp, status, c – status);
       temp[c – status] = ‘¥0’;
   }
   gaim_debug_error(“msn”, “Received non-OK result: %s¥n”, temp);

   slpcall->wasted = TRUE;

   /* msn_slp_call_destroy(slpcall); */
   return slpcall;
}
```

Gaim インスタントメッセンジャークライアントの V.80から抜粋した脆弱なコード

strncat()

```
strncpy(record, user, MAX_STRING_LEN – 1);
strcat(record, “:”);
strncat(record, cpw, MAX_STRING_LEN – 1);
```

Apache 1.31 の修正パッチ案

```
strncpy(record, user, MAX_STRING_LEN – 1);
strncat(record, “:”, MAX_STRING_LEN – strlen(record) - 1);
strncat(record, cpw, MAX_STRING_LEN – strlen(record) - 1);
```

今度は文字列の切り捨てが発生する可能性あり

- Apache httpdの修正パッチ．プログラマはstrXXX()をすべてstrnXXX()で置き換えた．
- しかし新しい問題を作り込んでしまった．
- userとcpwのサイズによってはrecordがオーバーフローしてしまう．MAX_STRING_LENはバッファ全体のサイズであって、recordには既にuserが書き込まれている．
- さらなる修正コードでも、userの長さによっては切り捨てが発生する．

strlcpy() と strlcat()

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 **src** から **dst** に最大 **(size-1)**文字数分コピーし、null終端する。

strlcat() は、null 終端文字列 **src** から**dst** の末尾に最大 **(size-strlen(dst)-1)** 文字数分追加し、null終端する。

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、コピー先バッファ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。

文字列の切り捨て

strlcpy() と **strlcat()** は、生成すべき文字列の全長(null文字を含まない)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、

- 文字列を格納するために必要なバイト数がわかる
- 返り値**+1**でメモリを再確保し、コピーし直す

strlcpy() と strlcat() のまとめ

JPCERT CC®

*BSD や Solaris ではシステムライブラリに取り込まれている。GNU/Linux では、glib(glibcではない)に g_strlcpy(), g_strlcat() という名称で取り込まれている。

- 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〈VOLUME 1〉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。