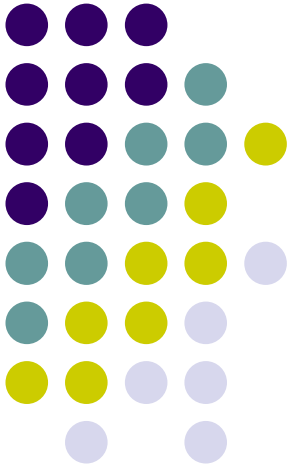
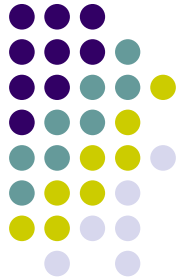
JPNIC・JPCERT/CC Security Seminar 2004
For Advanced

ぜい弱性キーワードを読み解く

古賀 洋一郎
NEC Corporation

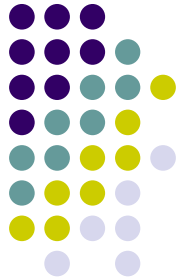


《CONTENTS》



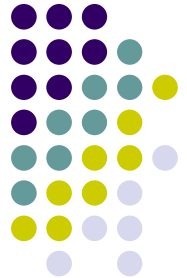
- はじめに
- 基礎知識
- Cプログラムのぜい弱性原理
  - Buffer Overflow (Stack-based / Heap-based / off-by-one)
  - Double free Bug
  - Format String Bug
- まとめ

《はじめに》



- ぜい弱性情報の特徴
- 本講義のねらい

ぜい弱性情報サンプル



[1] Microsoft Windows の JPEG 処理にバッファオーバーフローの脆弱性
（中略）
Microsoft Windows および Office には、JPEG 形式の画像の処理にバッファオーバーフローの脆弱性があります。結果として、遠隔から第三者が JPEG 形式の画像ファイルを経由して、ユーザの権限を取得する可能性があります。対象となる製品の詳細については、以下の関連文書を参照してください。

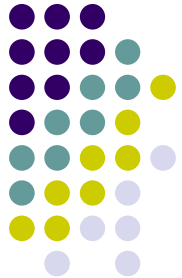
この問題は、Microsoft が提供する修正プログラムを適用することで解決します。

関連文書 (日本語)
JP Vendor Status Note JVNTA04-260A
Microsoft Windows JPEG コンポーネントにバッファオーバーフロー
http://jvn.jp/cert/JVNTA04-260A.html

マイクロソフト セキュリティ情報
JPEG 処理 (GDI+) のバッファ オーバーランにより、コードが実行される (833987) (MS04-028)
http://www.microsoft.com/japan/technet/security/bulletin/ms04-028.asp

「JPCERT/CC REPORT 2004-09-22」より

ぜい弱性情報の特徴

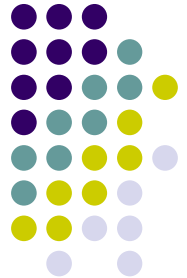


- よくあるパターン
  - 対象
  - 概要説明
  - 想定される被害
  - 解決策
  - リファレンス
- 説明なしに専門用語が使われる
  - 知らないうちに当たり前のように使われるようになっている

ぜい弱性情報の特徴

- 説明なしに専門用語を使うのは・・・
  - 読者は知っているはずなので
  - 知的好奇心を刺激する意図で
  - 用語に通じている人の自尊心や虚栄心をくすぐる効果をねらって

ぜい弱性情報の特徴

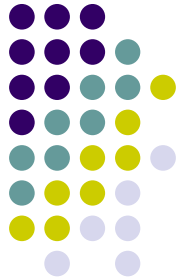


- 読む側に知識が求められている？
  - 「セキュアプログラミング」のための情報ならば結構ある
    - 書籍やWebなど
    - 日本語情報も多い
  - 原理の解説は探せば見つかるのだが・・・
    - 日本語情報は乏しい
    - 英語情報はやんちゃで不親切
    - 理解するにはかなりの基礎知識が要求される

本講義のねらい

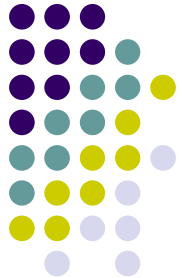
- Cプログラムのぜい弱性キーワードのメカニズムを示す
  - 技術屋として、知らないままにしておかないようにする
    - 知らないまま使われたり使ったりするのは気持ちが悪い
  - 「風が吹けば桶屋が儲かる」からの脱却
  - 「三味線」と「猫」の関係も分かるようになるかも

《基礎知識》



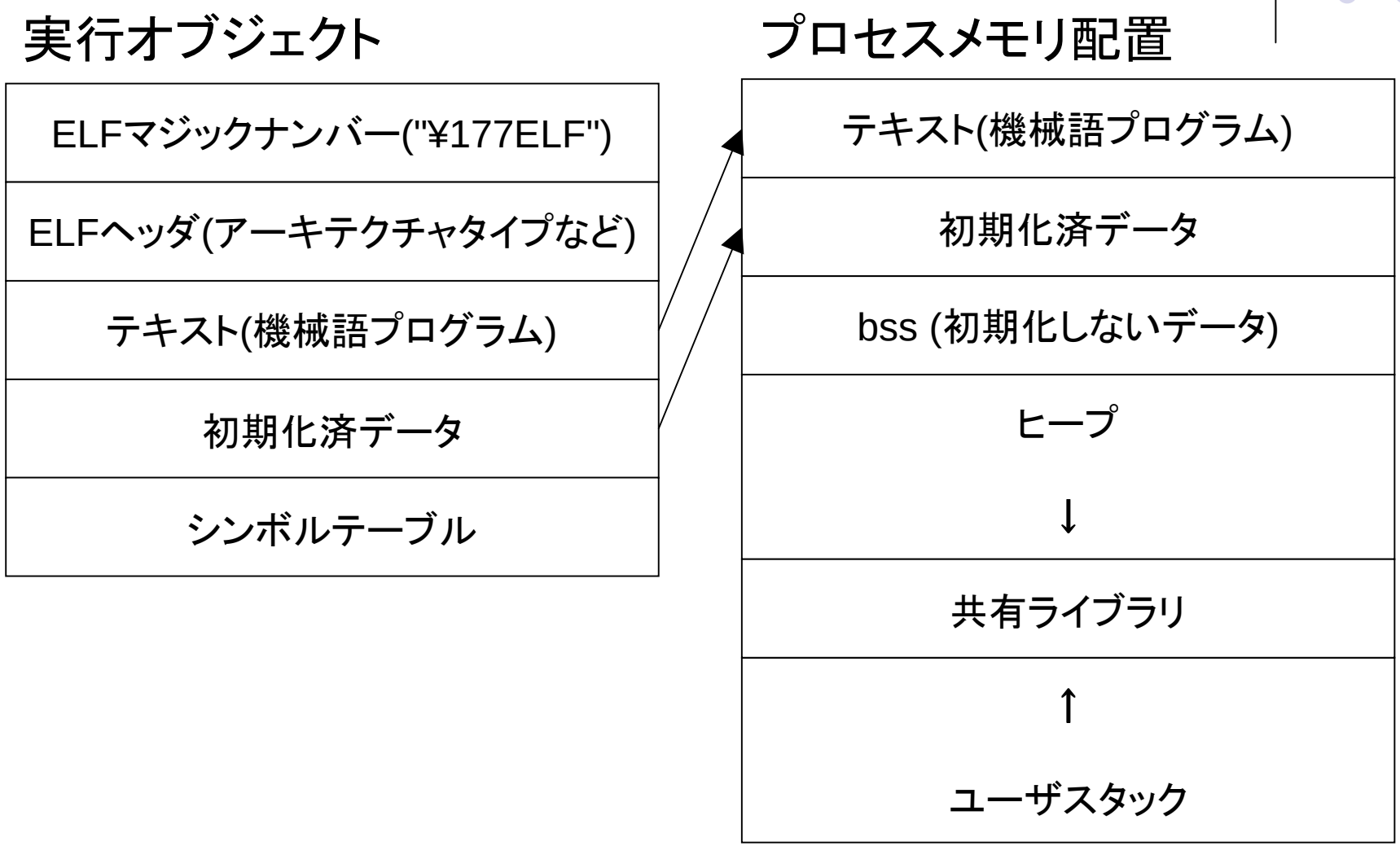
- プログラムが**動く**しくみ
- 攻撃者のねらい

プログラムが動くしくみ

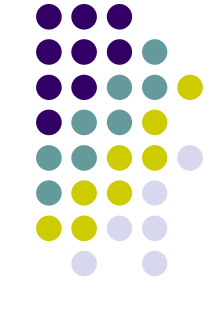


- 実行オブジェクトをメモリにロード
  - CPUが機械語プログラムを読み込み、命令実行
- 実行オブジェクト
  - 機械語プログラム + データ で構成
  - （参考）代表的なオブジェクト形式
    - 歴史的な UNIX a.out
    - System V 初期の COFF(Common Object File Format)
    - UNIX ELF(Executable and Linking Format)
    - Windows PE(Portable Executable)
    - MS-DOS .COM, .EXE, IBM 360, Intel/Microsoft OMF など

実行オブジェクトとメモリ配置（ELF）



プログラムの実行



プロセスメモリ配置

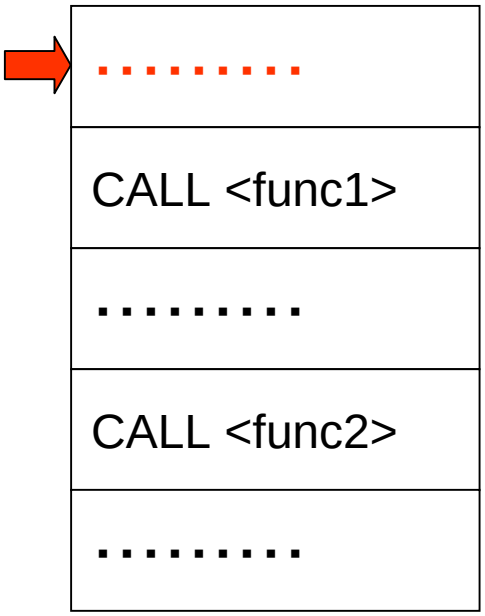
テキスト(機械語プログラム)
初期化済データ
bss (初期化しないデータ)
ヒープ ↓
共有ライブラリ
↑ ユーザスタック

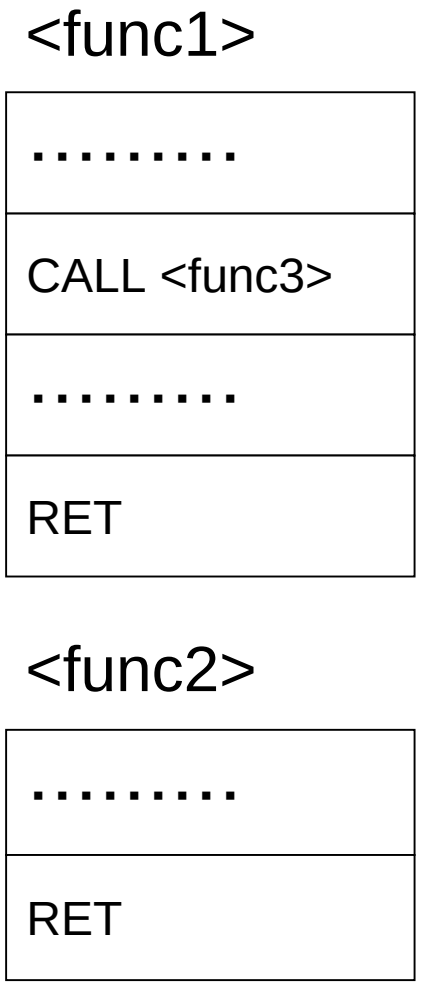
テキスト領域の機械語プログラムをメモリから読み込み、順次命令を実行する。

変数などのデータやプログラム内部の管理情報はスタック領域やヒープ領域に格納され、適宜参照、変更される。

- スタック
  - 一時的に確保されるメモリ領域
  - 自動変数(ローカル変数)、関数の引数や、戻りアドレス(呼び出し元アドレス)などが格納される
- ヒープ
  - 動的に確保されるメモリ領域
  - malloc 関数やシステムコール brk、mmap などで割り当てられる

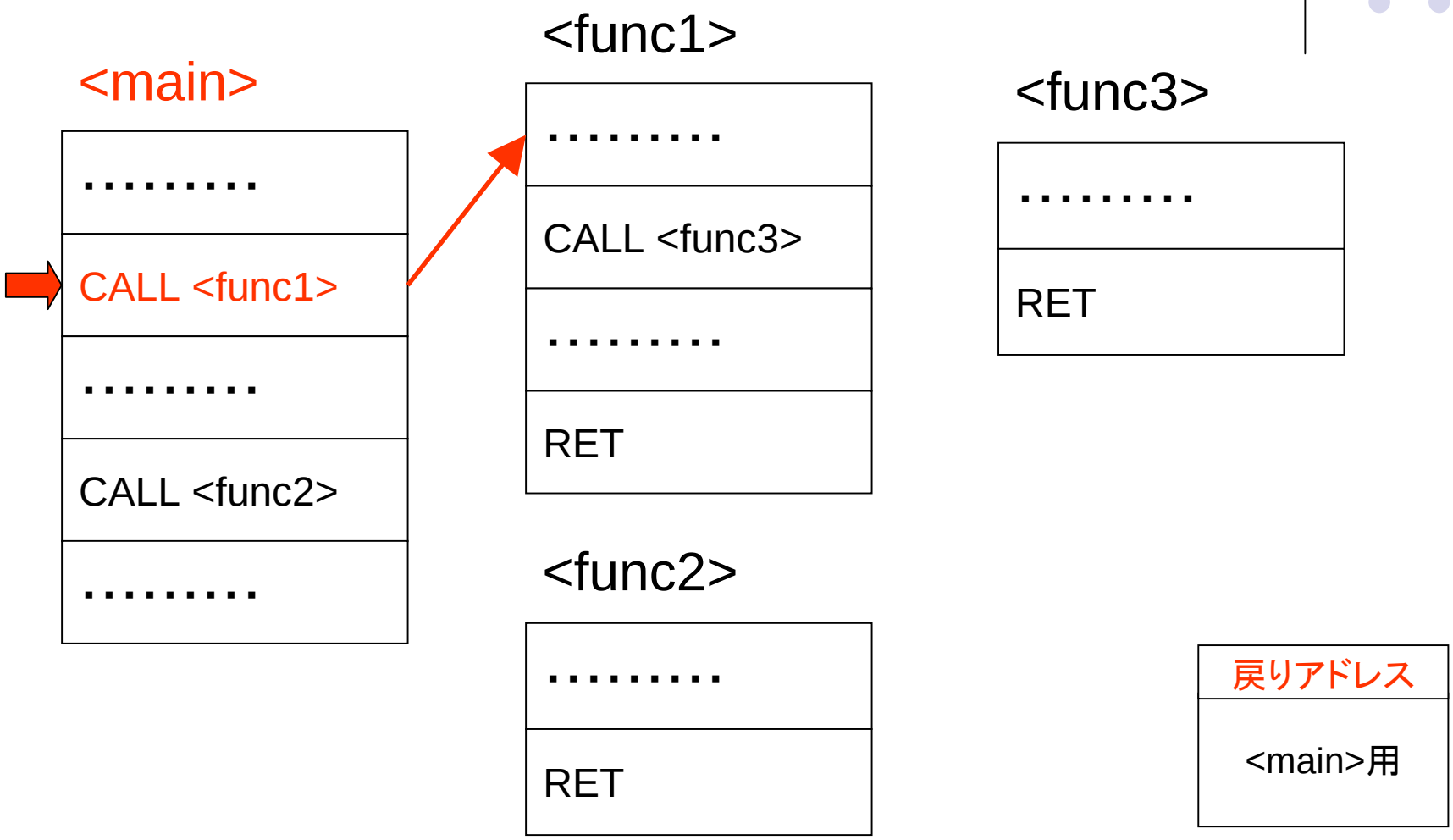
プログラムの実行



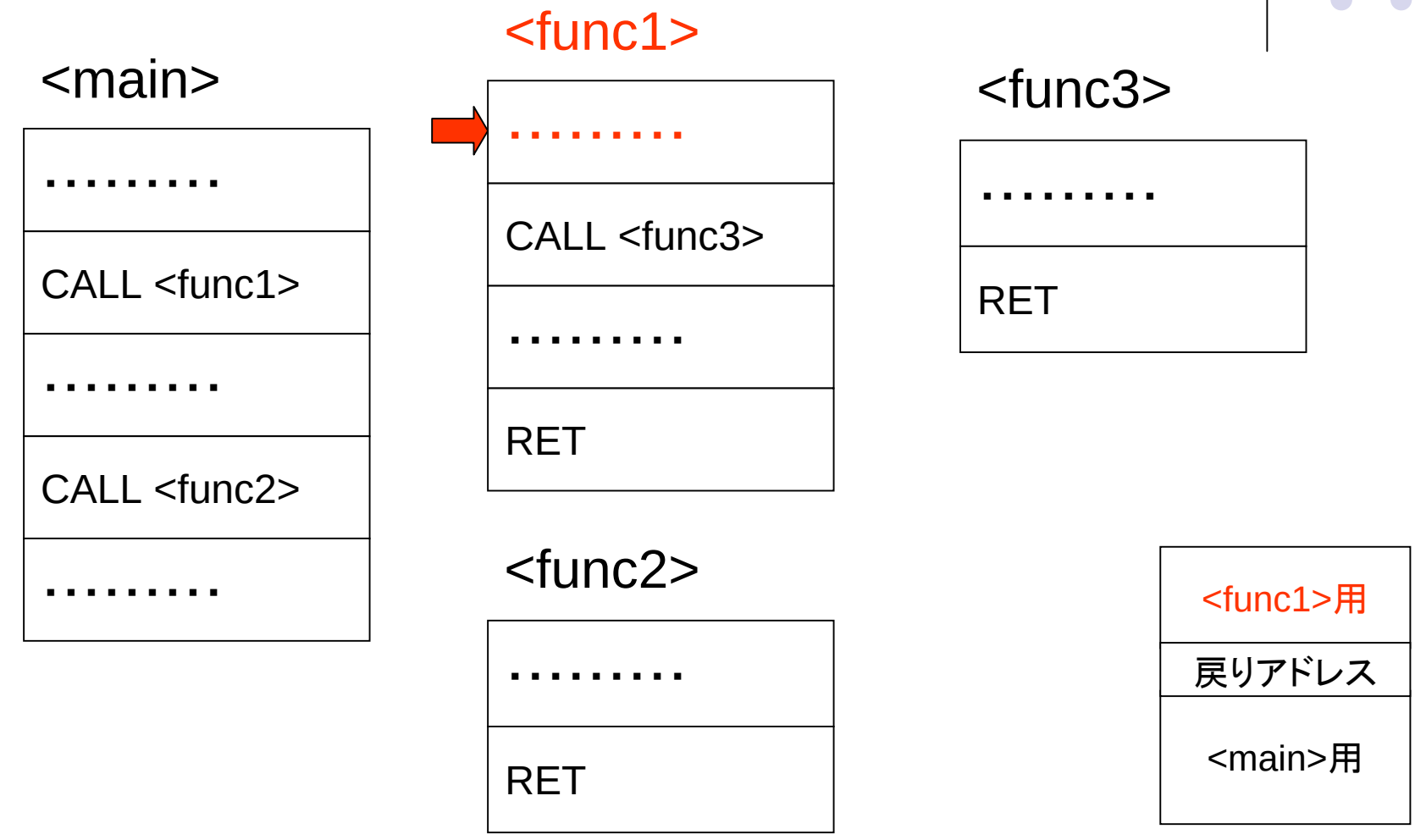


ユーザスタック

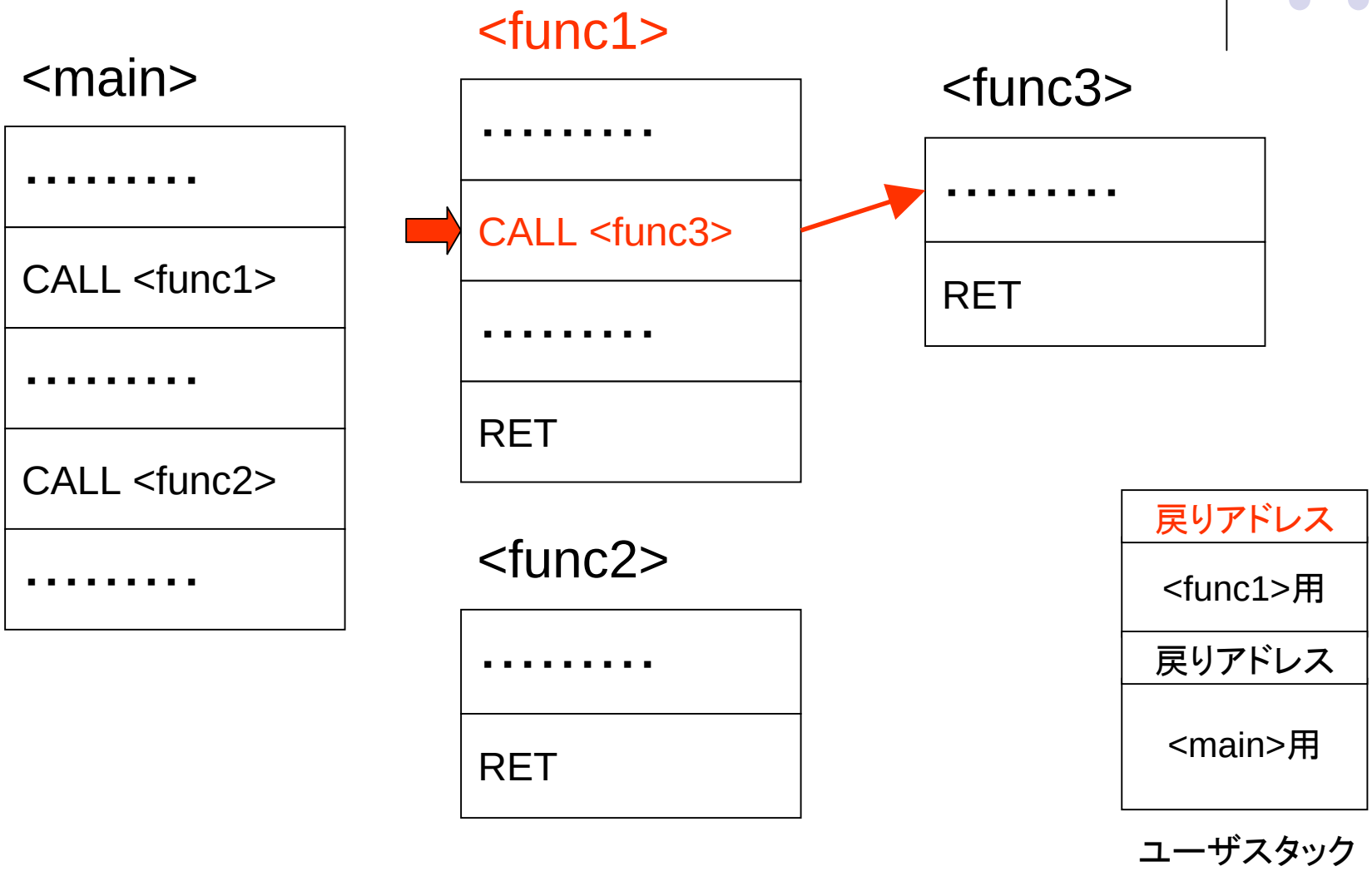
プログラムの実行


<main>
.....
CALL <func1>
.....
CALL <func2>
.....
<func1>
.....
CALL <func3>
.....
RET
<func2>
.....
RET
<func3>
.....
RET
戻りアドレス
<main>用
ユーザスタック

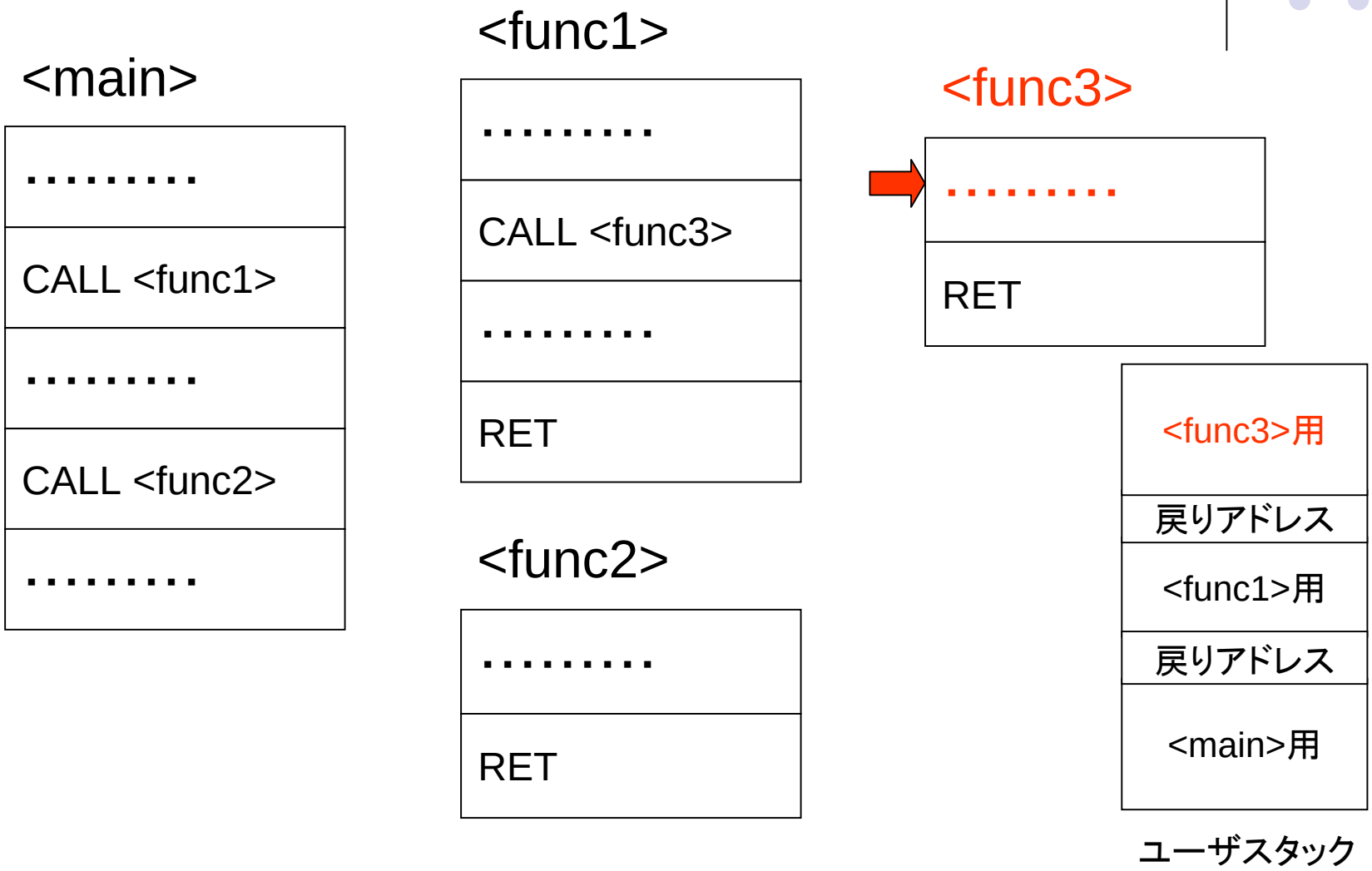
プログラムの実行


<main>
.....
CALL <func1>
.....
CALL <func2>
.....
<func1>
.....
CALL <func3>
.....
RET
<func2>
.....
RET
<func3>
.....
RET
<func1>用
戻りアドレス
<main>用
ユーザスタック

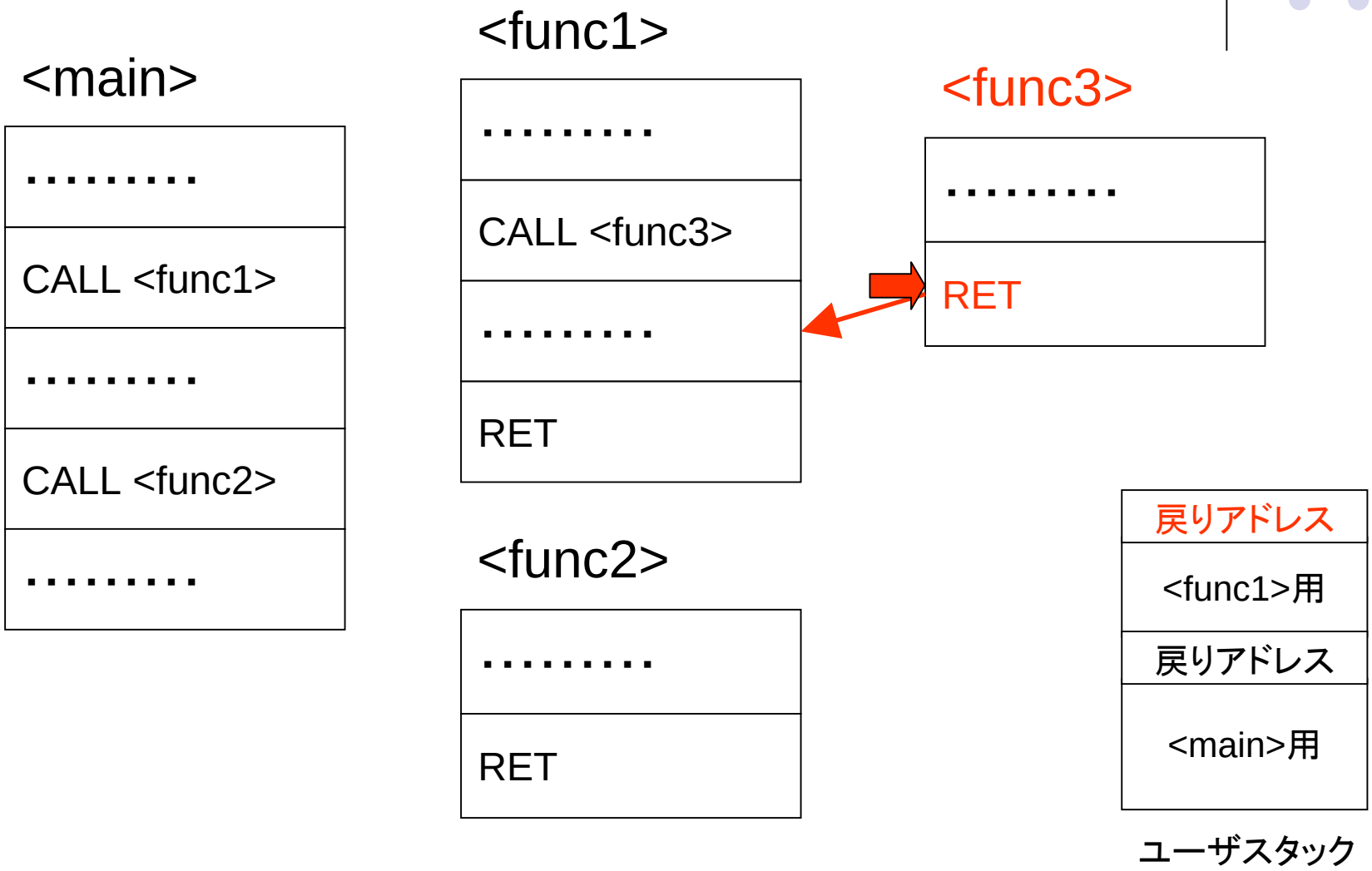
プログラムの実行


<main>
.....
CALL <func1>
.....
CALL <func2>
.....
<func1>
.....
CALL <func3>
.....
RET
<func3>
.....
RET
<func2>
.....
RET
戻りアドレス
<func1>用
戻りアドレス
<main>用
ユーザスタック

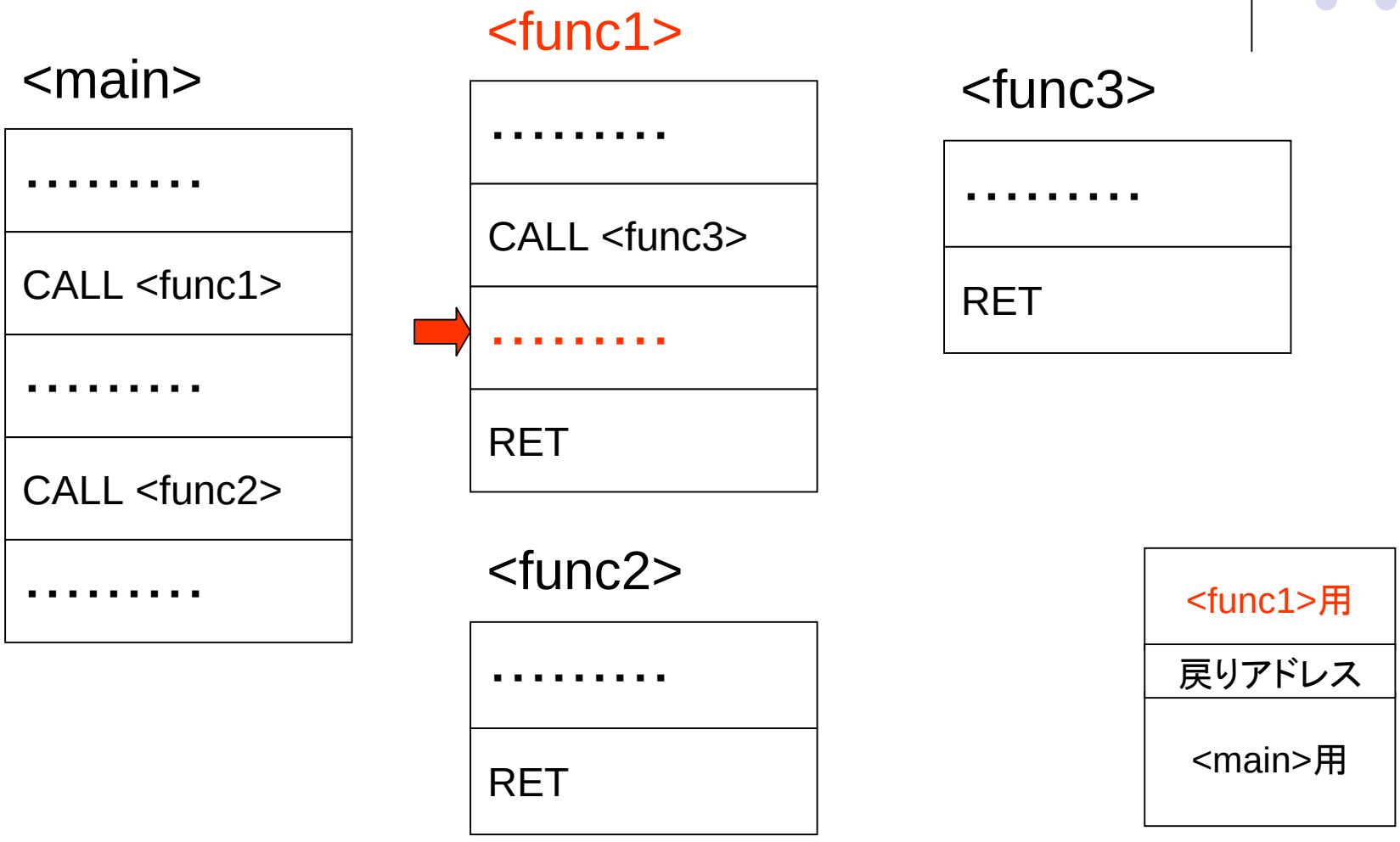
プログラムの実行


<main>
.....
CALL <func1>
.....
CALL <func2>
.....
<func1>
.....
CALL <func3>
.....
RET
<func2>
.....
RET
<func3>
.....
RET
<func3>用
戻りアドレス
<func1>用
戻りアドレス
<main>用
ユーザスタック

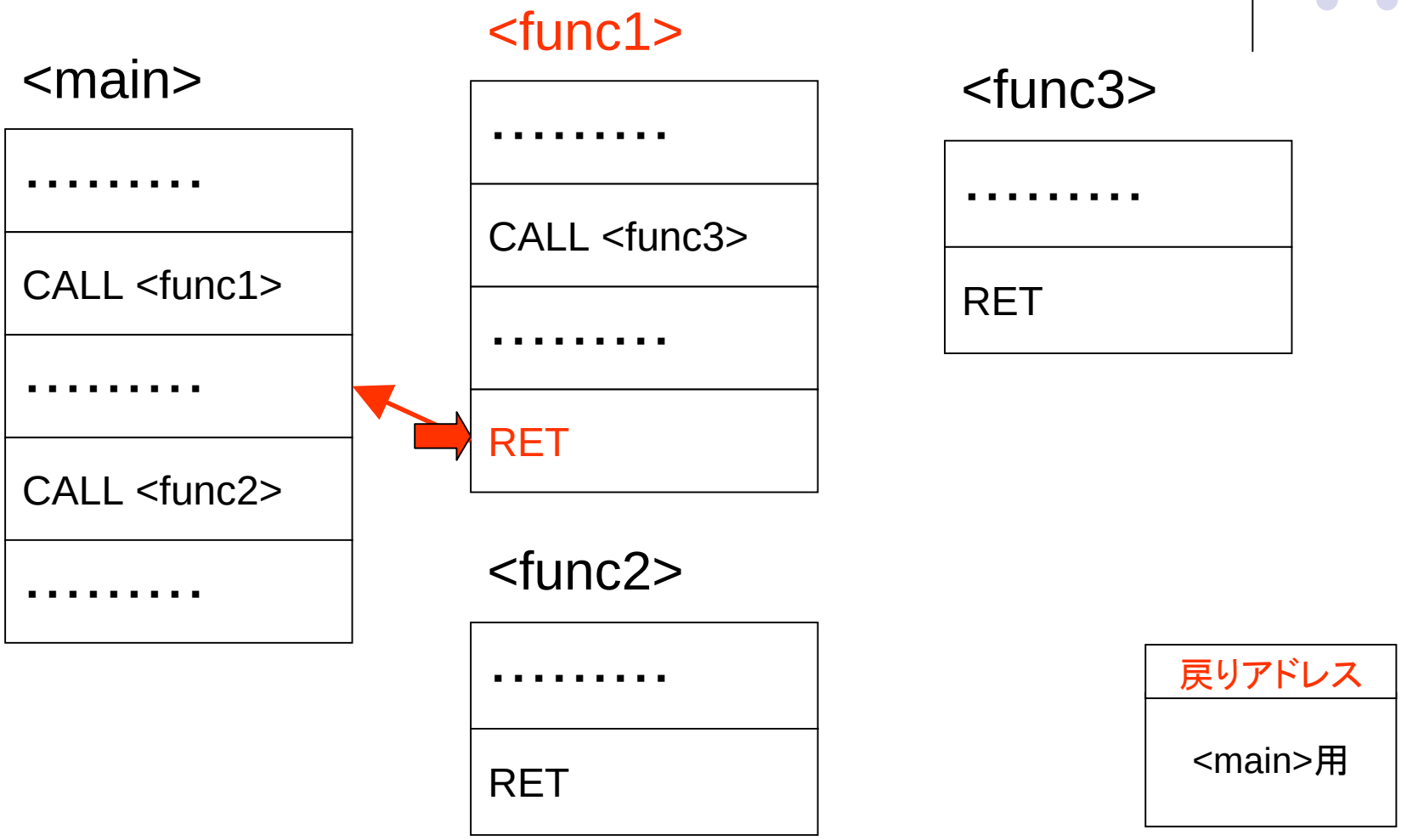
プログラムの実行


<main>
.....
CALL <func1>
.....
CALL <func2>
.....
<func1>
.....
CALL <func3>
.....
RET
<func2>
.....
RET
<func3>
.....
RET
戻りアドレス
<func1>用
戻りアドレス
<main>用
ユーザスタック

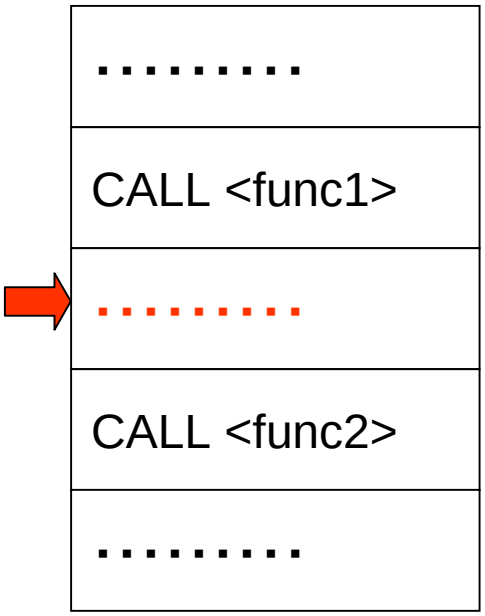
プログラムの実行

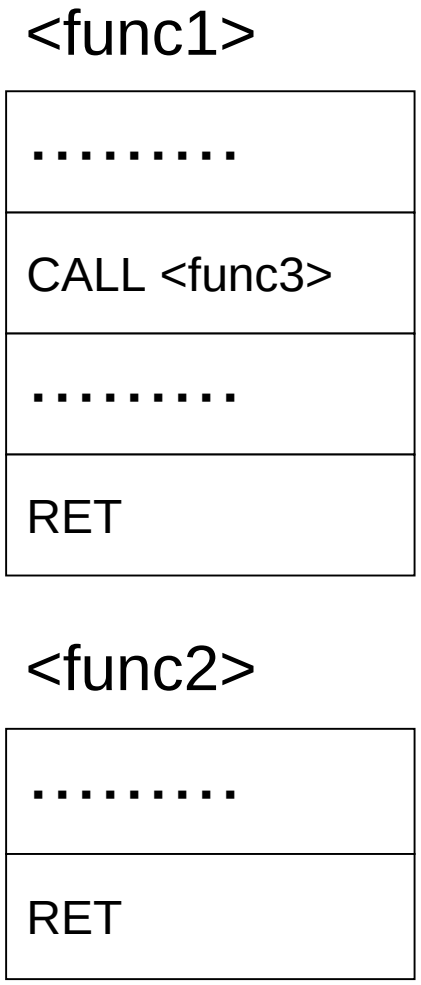

<main>
.....
CALL <func1>
.....
CALL <func2>
.....
<func1>
.....
CALL <func3>
.....
RET
<func2>
.....
RET
<func3>
.....
RET
<func1>用
戻りアドレス
<main>用
ユーザスタック

プログラムの実行


<main>
.....
CALL <func1>
.....
CALL <func2>
.....
<func1>
.....
CALL <func3>
.....
RET
<func2>
.....
RET
<func3>
.....
RET
戻りアドレス
<main>用
ユーザスタック

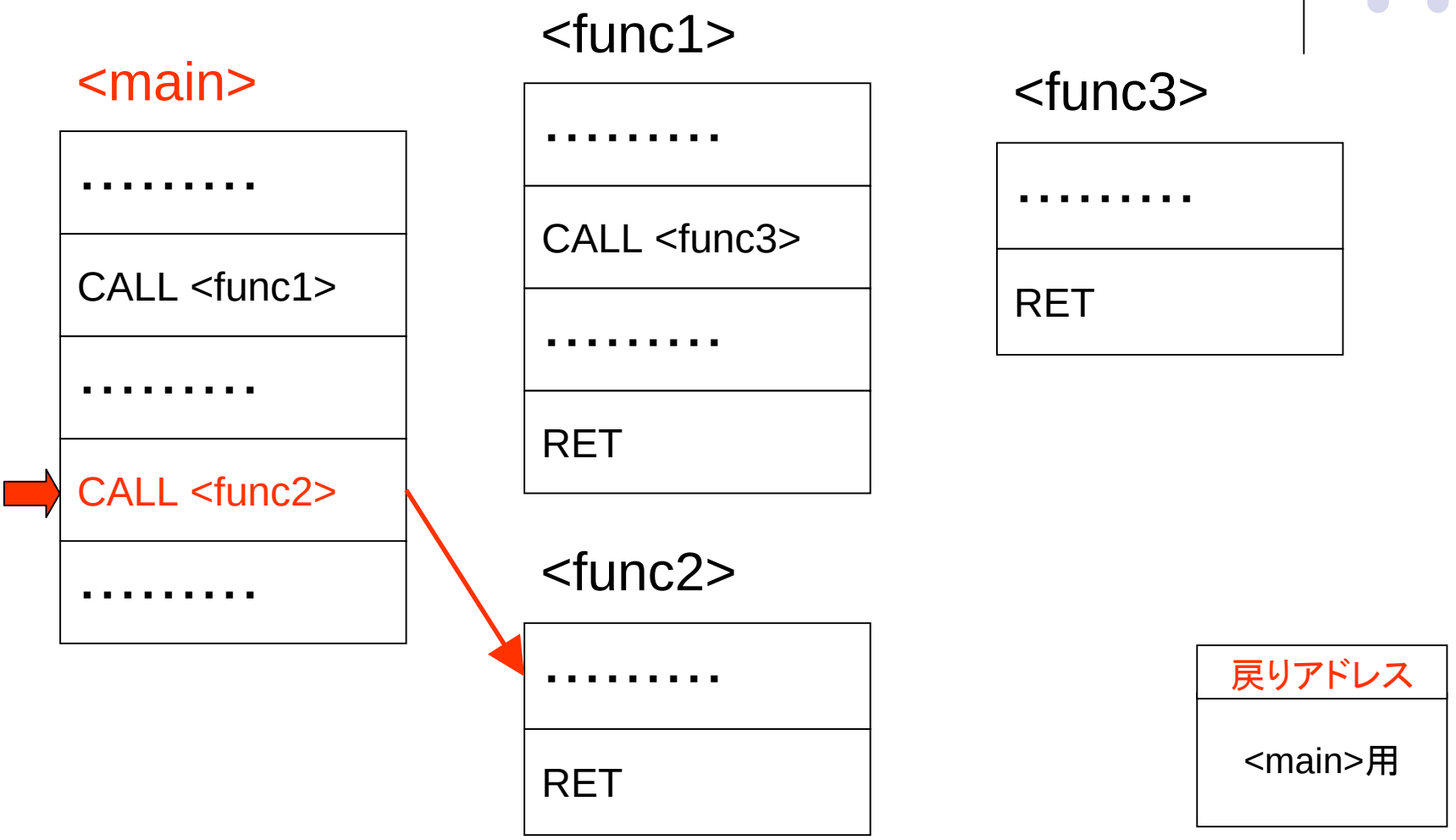
プログラムの実行



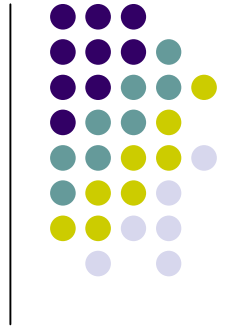


ユーザスタック

プログラムの実行


<main>
.....
CALL <func1>
.....
CALL <func2>
.....
<func1>
.....
CALL <func3>
.....
RET
<func2>
.....
RET
<func3>
.....
RET
戻りアドレス
<main>用
ユーザスタック

プログラムの実行



<main>

・・・・・・・・・
CALL <func1>
・・・・・・・・・
CALL <func2>
・・・・・・・・・

<func1>

・・・・・・・・・
CALL <func3>
・・・・・・・・・
RET

<func3>

・・・・・・・・・
RET

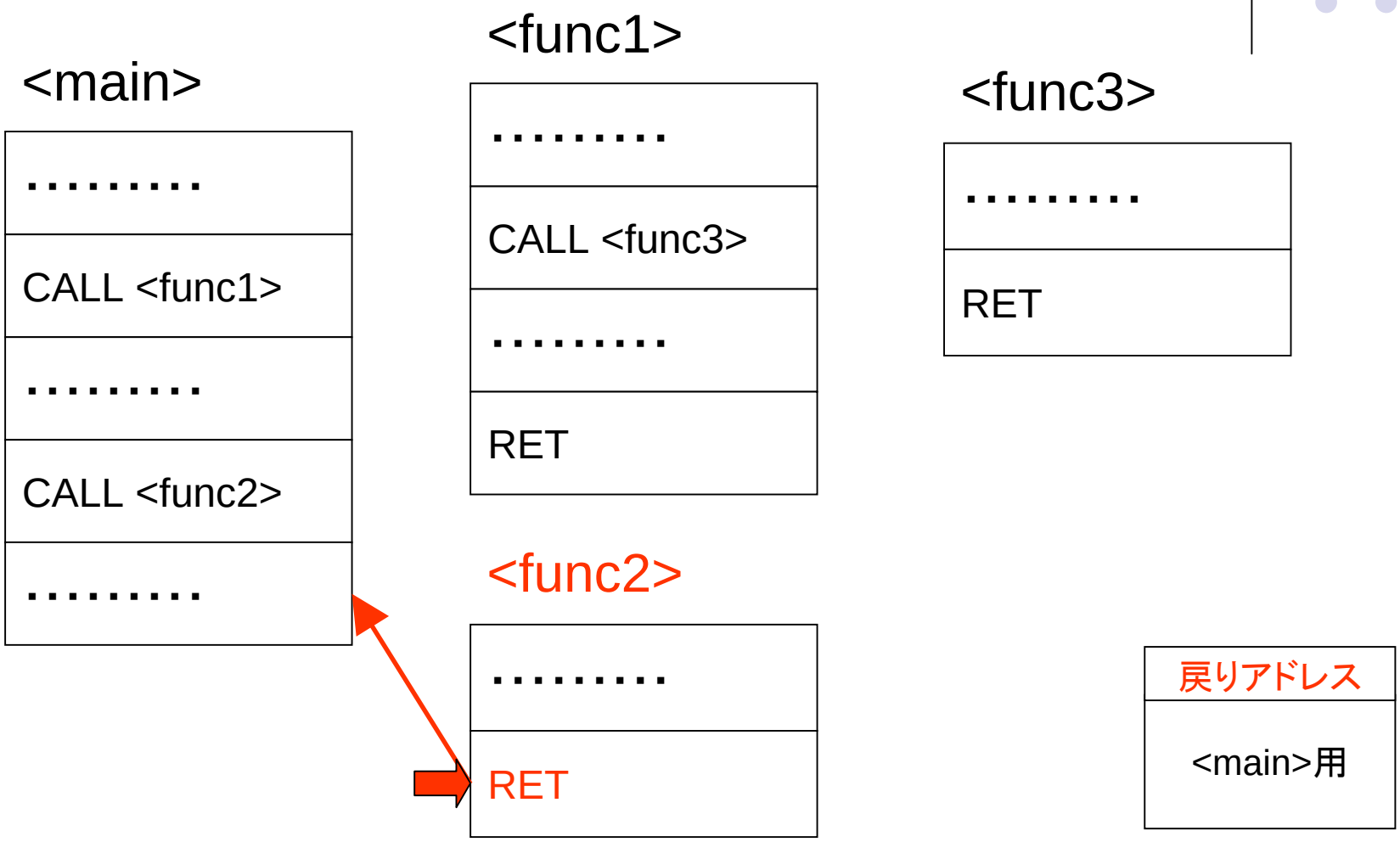
<func2>

・・・・・・・・・
RET

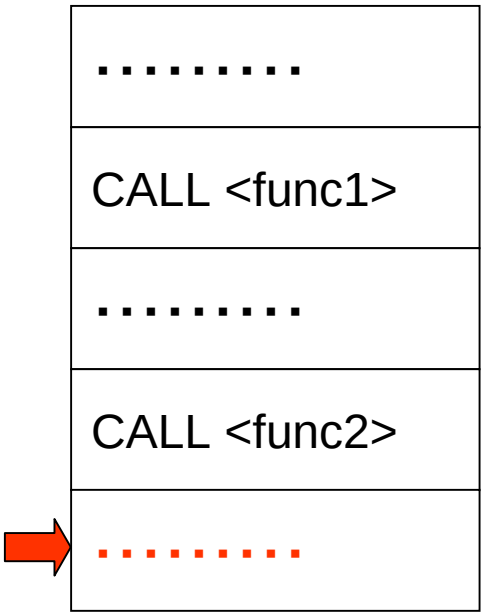
<func2>用
戻りアドレス
<main>用

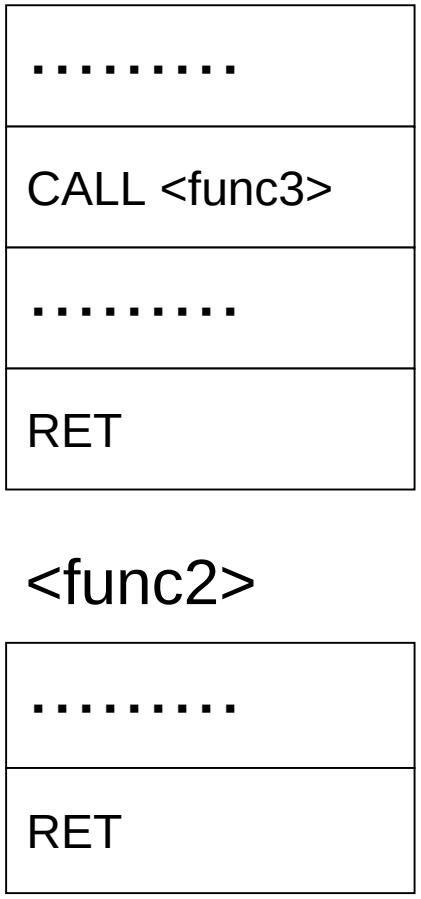
ユーザスタック

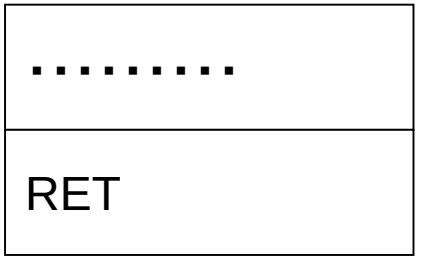
プログラムの実行


<main>
.....
CALL <func1>
.....
CALL <func2>
.....
<func1>
.....
CALL <func3>
.....
RET
<func2>
.....
RET
<func3>
.....
RET
戻りアドレス
<main>用
ユーザスタック

プログラムの実行

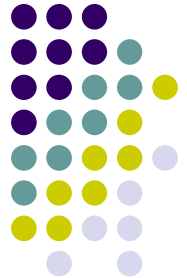






ユーザスタック

攻撃者のねらい



- メモリ中の重要データの読み出し、書き換え
  - 条件判定用データを書き換え、アクセス制限を迂回
  - パスワードや暗号鍵など、秘密情報の読み出し
- 攻撃用コードの実行

攻撃用コードの実行（1）

fp1.c:

```
#include <stdio.h>

int main()
{
    unsigned long *fp;

    fp = (unsigned long *) 0x08049694;
    (*fp) = 0x804853e;

    printf("Hello, ");
    fprintf(stdout, "World¥n");

    return 0;
}
```

実行結果　(FreeBSD-4.10)

```
% cc -o fp1 fp1.c
% ./fp1
Hello, Hello, Hello, Hello,
Hello, Hello, Hello, Hello,
Hello, Hello, Hello, Hello,
Hello, Hello, Hello, Hello,
Hello, Hello, Hello, Hello,
Hello, Hello, Hello, Hello,
Hello, Hello, Hello, Hello,
Hello, Hello, Hello, Hello,
Hello, Hello, Hello, Hello,
Hello, Hello, Hello, Hello,
Hello, Hello, Hello, Hello,
Hello, Hello, Hello, Hello,
Hello, Hello, Hello, ^C
```

攻撃用コードの実行(2)

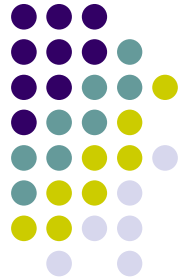
fp2.c:

```
#include <stdio.h>
char shellcode[] =
    "¥xeb¥x0e¥x5e¥x31¥xc0¥x88¥x46¥x07¥x50¥x50¥x56¥xb0¥x3b¥x50¥xcd"
    "¥x80¥xe8¥xed¥xff¥xff¥xff¥x2f¥x62¥x69¥x6e¥x2f¥x73¥x68";

int main()
{
    unsigned long *fp;

    fp = (unsigned long *) 0x080496b4;
    (*fp) = (unsigned long) shellcode;

    printf("Hello, ¥n");
    fprintf(stdout, "World¥n");

    return 0;
}
```

実行結果 (FreeBSD-4.10)

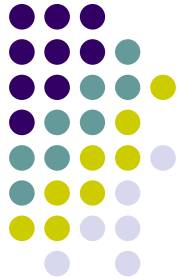
```
% cc -o fp2 fp2.c
% ./fp2
Hello,
$  (← /bin/sh のプロンプト)
$ ps $$
  PID  TT  STAT      TIME COMMAND
34489  p2  S      0:00.01  (sh)
$
```

攻撃者のねらい



- 攻撃用コードの実行
  - ねらったアドレスのデータを書き換えて
  - 攻撃用コードに処理を飛ばす
- 攻撃用コードの実行につながるデータ
  - サブルーチン（関数）の戻りアドレス
  - 関数のポインタ
  - 書き込み可能領域上の、将来呼ばれるコード

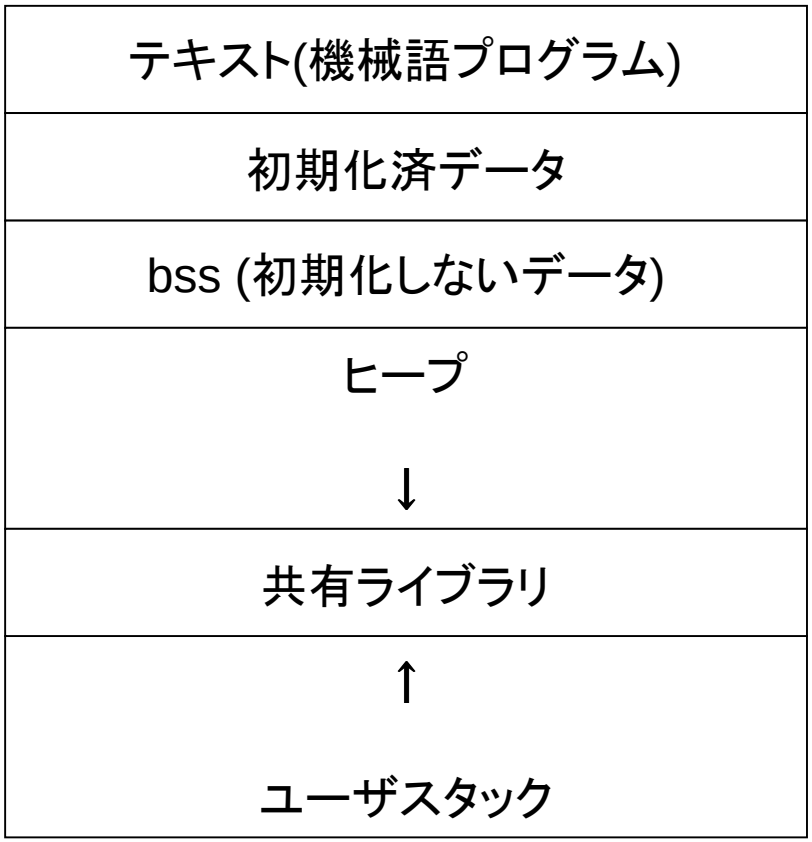
《Cプログラムのぜい弱性原理》



- Buffer Overflow (Stack-based / Heap-based / off-by-one)
- Double free Bug
- Format String Bug

バッファとは？

- 連続に確保された、サイズ固定のメモリ領域
- スタック領域またはヒープ領域に確保される



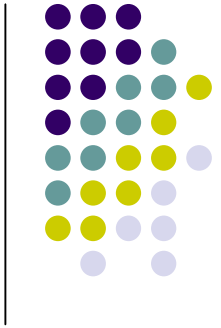
- スタック
  - 一時的に確保されるメモリ領域
  - 自動変数(ローカル変数)、関数の引数や、戻りアドレス(呼び出し元アドレス)などが格納される
- ヒープ
  - 動的に確保されるグローバル変数用のメモリ領域
  - malloc 関数やシステムコール brk、mmap などで割り当てられる

Buffer Overflowとは

- Buffer Overrun とも呼ばれる
- 確保したバッファのサイズを越えてデータが書き込まれてしまう状態

```
char c[10];

c[0] = 'H';
c[1] = 'e';
c[2] = 'l';
c[3] = 'l';
c[4] = 'o';
c[5] = ',';
c[6] = 'W';
c[7] = 'o';
c[8] = 'r';
c[9] = 'l';          /* ここまで */
c[10] = 'd';         /* overflow! */
c[11] = '¥0';        /* overflow! */
```

Buffer Overflow

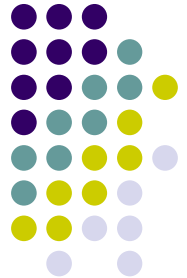


- ぜい弱性情報の20%程度とポピュラー

Vulnerability Type	2004	2003	2002	2001
Input Validation Error	296 (52%)	530 (53%)	662 (51%)	744 (49%)
(Boundary Condition Error)	46 (8%)	81 (8%)	22 (2%)	51 (3%)
(Buffer Overflow)	**105 (18%)**	**237 (24%)**	**287 (22%)**	**316 (21%)**
Access Validation Error	43 (8%)	92 (9%)	123 (9%)	126 (8%)
Exceptional Condition Error	77 (13%)	150 (15%)	117 (9%)	146 (10%)
Environment Error	6 (1%)	3 (0%)	10 (1%)	36 (2%)
Configuration Error	17 (3%)	49 (5%)	68 (5%)	74 (5%)
Race Condition	6 (1%)	17 (2%)	23 (2%)	50 (3%)
Design Error	135 (24%)	269 (27%)	408 (31%)	399 (26%)
Other	41 (7%)	20 (2%)	1 (0%)	8 (1%)

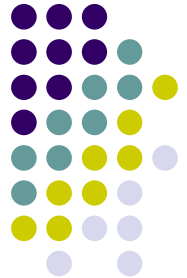
NIST（米国商務省標準技術局）ICAT Metabase (2004年8月18日版) より
http://icat.nist.gov/icat.cfm?function=statistics

Buffer Overflowの防ぎ方



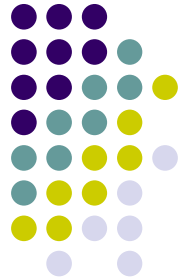
- 境界チェックをしっかり行なう、数え間違いをしない
  - 1バイトでも、あふれると危険なこともある
- 危険な関数をできるだけ使わない
  - gets → fgets, (v)scanf → fgets+(v)sscanf
  - getenv, realpath, getwd は後の処理でサイズに注意

Buffer Overflowの防ぎ方



- 境界チェックを要求する関数を使う
  - strncat, strncpy, snprintf など
    - トランケートされてもいいのかどうか要注意
  - ナル文字 '¥0' での終端を徹底する
    - strncpy はナル文字が付加されないことがある
    - strncat は常にナル文字を付加する
    - snprintf, vsnprintf は ISO C99 で標準化された関数で、常にナル文字を付加するが、ISO C99 より前の C の処理系で付加しないものがある

Stack-based Buffer Overflow



- スタック領域に確保されたバッファについて起こるBuffer Overflow のこと
- スタック領域に確保された変数データや配列データが上書きされる
- 攻撃者のターゲットは・・・
  - 条件判定用データを書き換え、アクセス制限を迂回
  - サブルーチンの戻りアドレスを書き換え、攻撃用コードにとばす

Stack-based Buffer Overflow

bo.c:

```
#include <stdio.h>

int main()
{
    int i;
    int a[5];

    for (i = 0; i <= 5; i++) {
        printf("Before: i = %d, ", i);
        a[i] = 0;
        printf("After: i = %d¥n", i);
    }

    return 0;
}
```

実行結果 (FreeBSD-4.10)

```
% cc -o bo bo.c
% ./bo
Before: i = 0, After: i = 0
Before: i = 1, After: i = 1
Before: i = 2, After: i = 2
Before: i = 3, After: i = 3
Before: i = 4, After: i = 4
Before: i = 5, After: i = 0
Before: i = 1, After: i = 1
Before: i = 2, After: i = 2
Before: i = 3, After: i = 3
Before: i = 4, After: i = 4
Before: i = 5, After: i = 0
Before: i = 1, After: i = 1
Before: i = 2, After: i = 2
^C
```

Stack-based Buffer Overflow

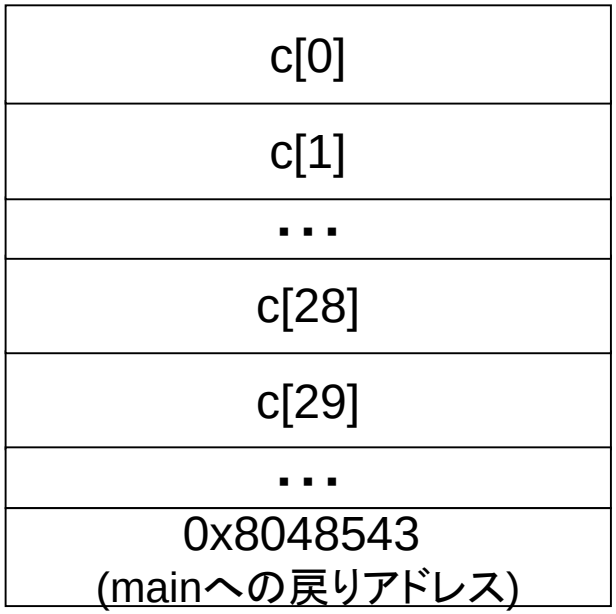
sbo.c:

```
#include <stdio.h>

void func()
{
    char c[30];
    gets(c);
    printf("%s¥n", c);
    return;
}

int main()
{
    func();
    return 0;
}
```

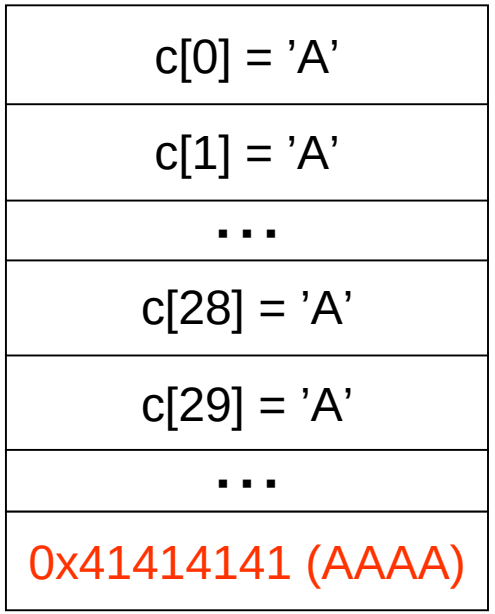
実行結果 (FreeBSD-4.10)

```
% cc -o sbo sbo.c
% perl -e 'print "A"x40;' | ./sbo
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
%
```

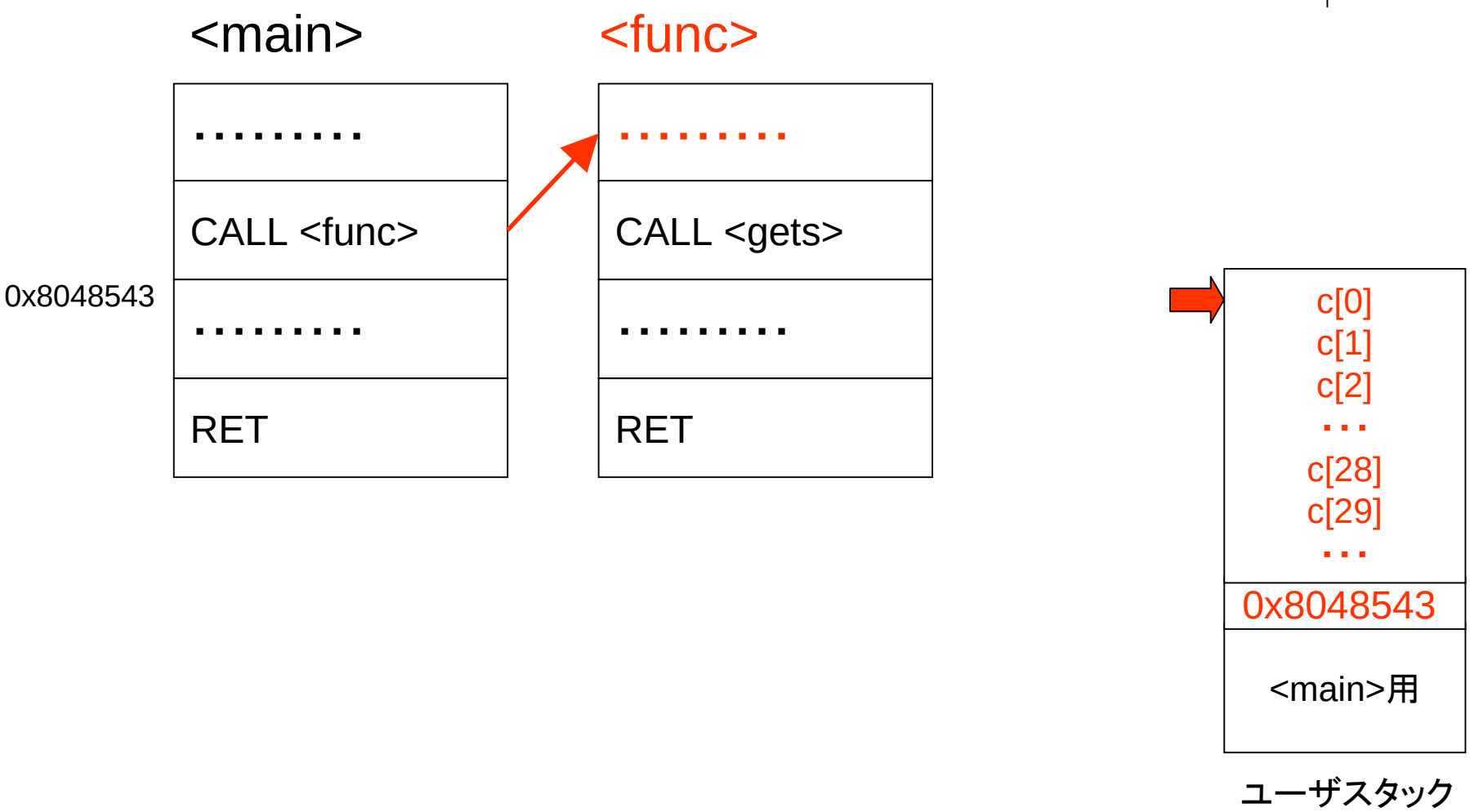
func呼び出し時のスタック



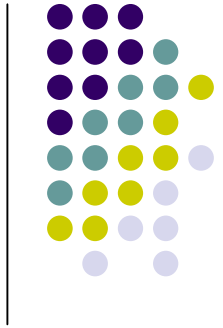
gets後のスタック

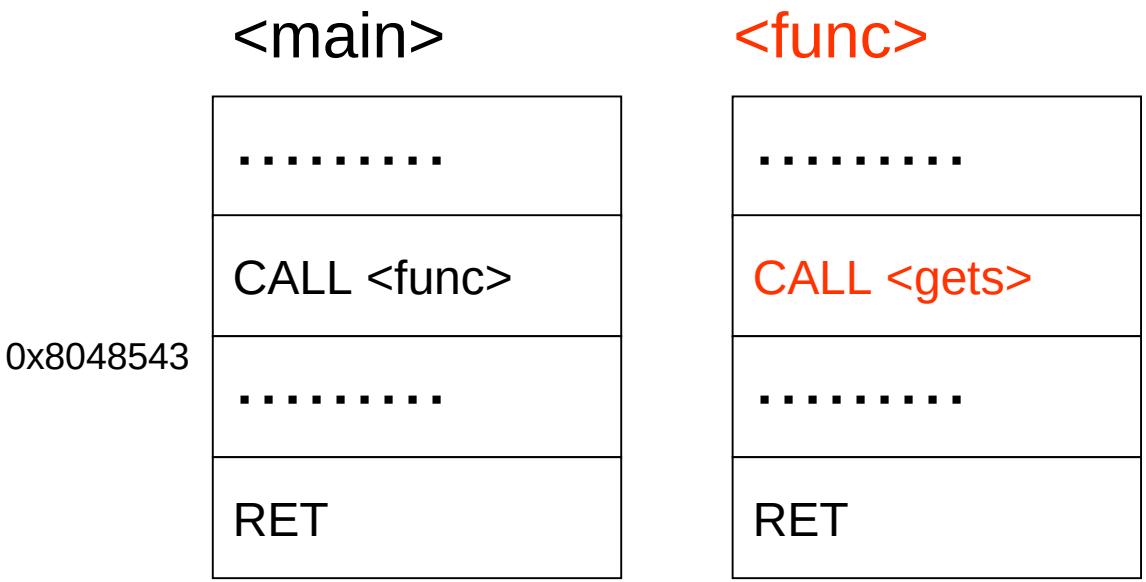


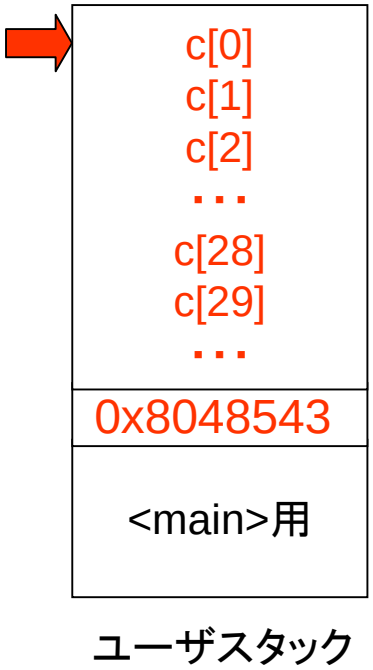
Stack-based Buffer Overflow



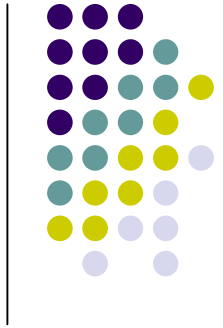
Stack-based Buffer Overflow

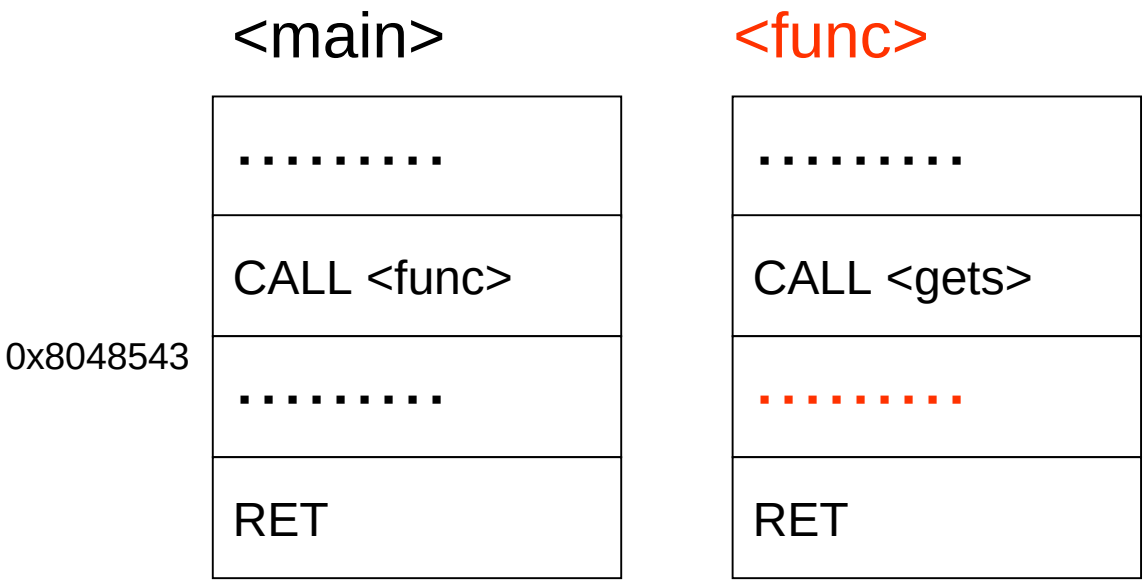


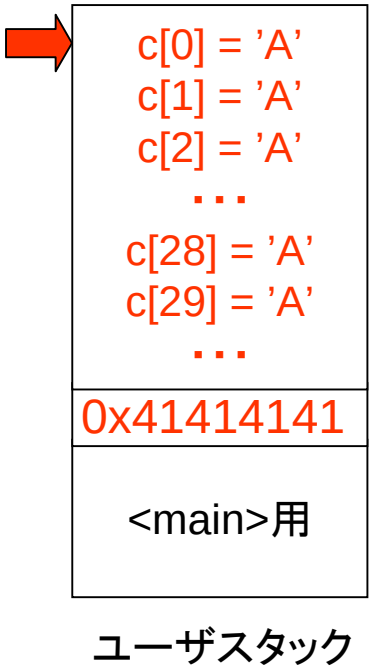




Stack-based Buffer Overflow

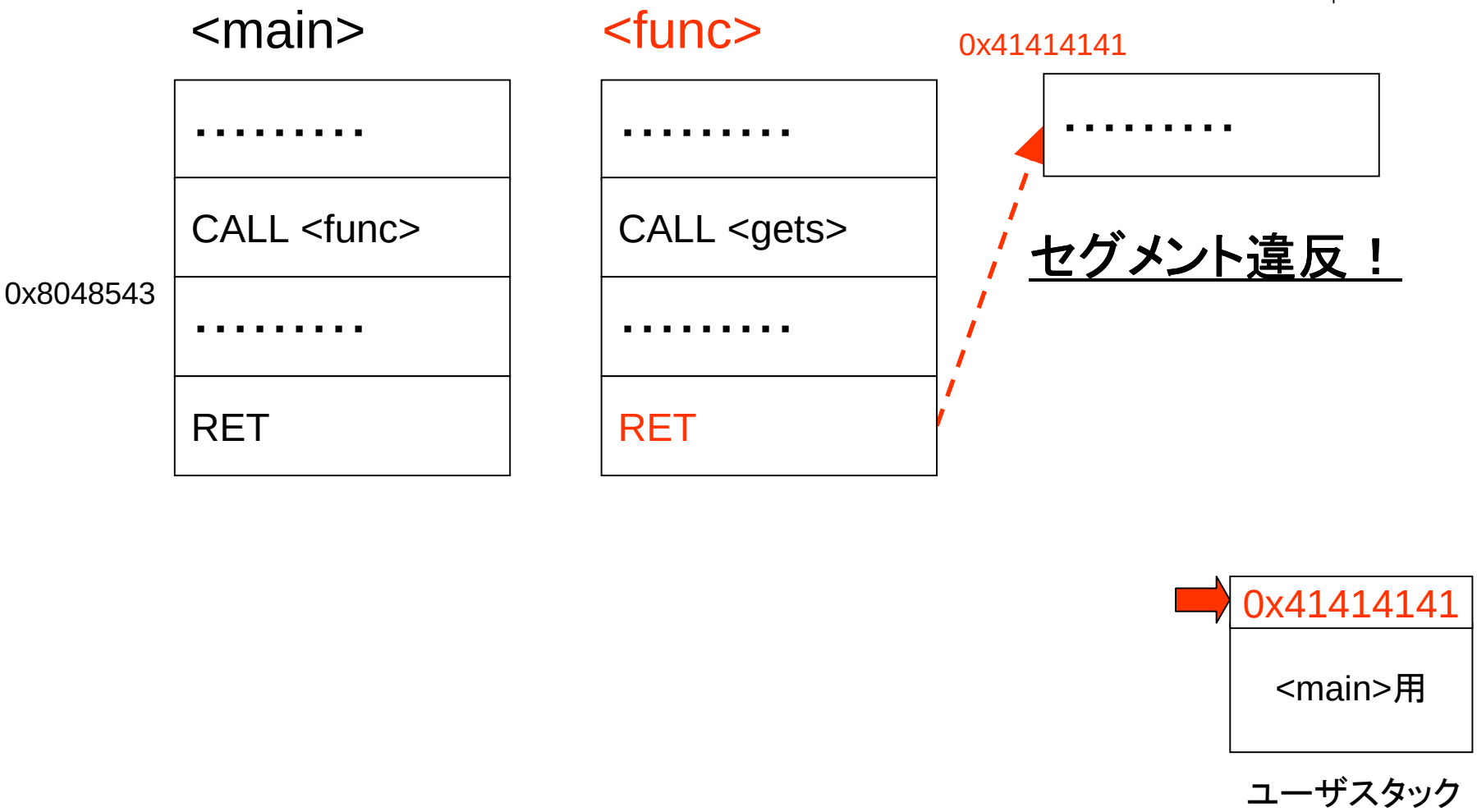




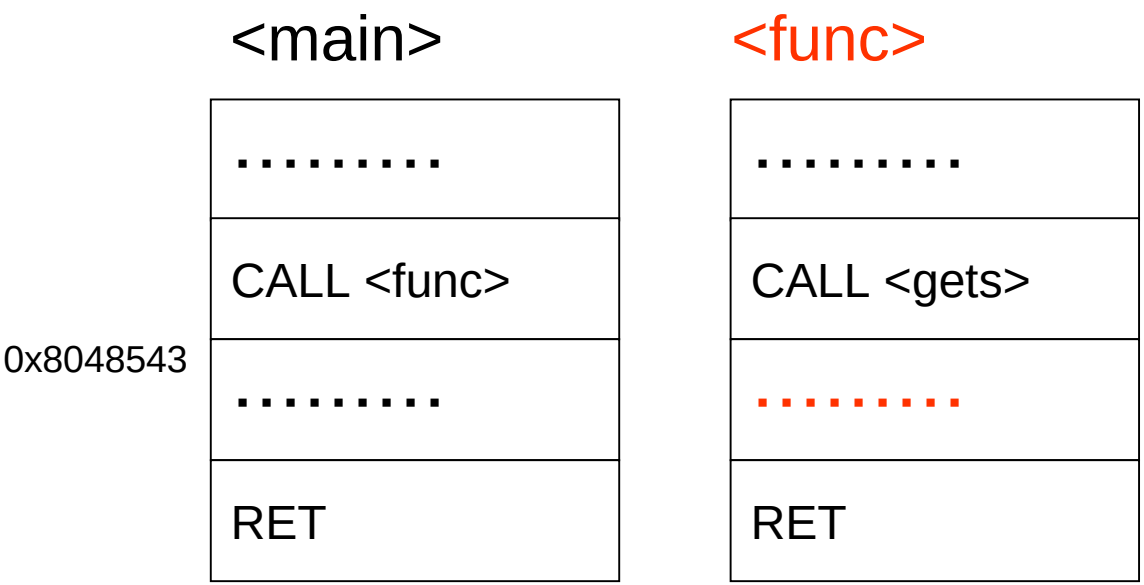


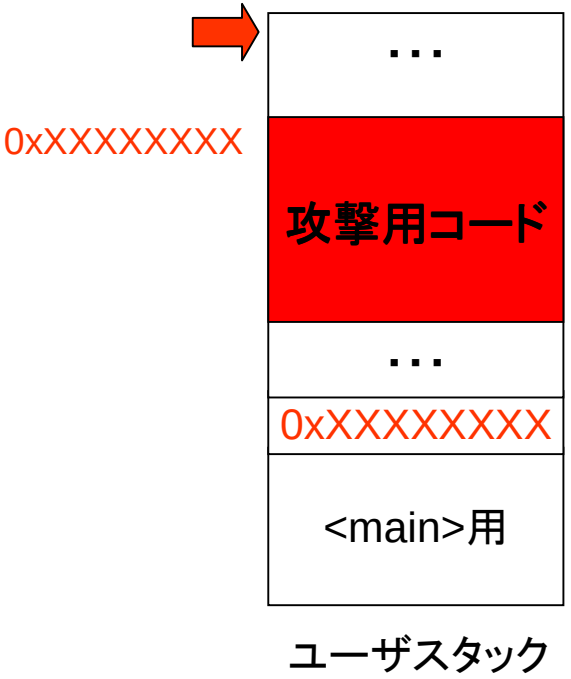
ユーザスタック

Stack-based Buffer Overflow

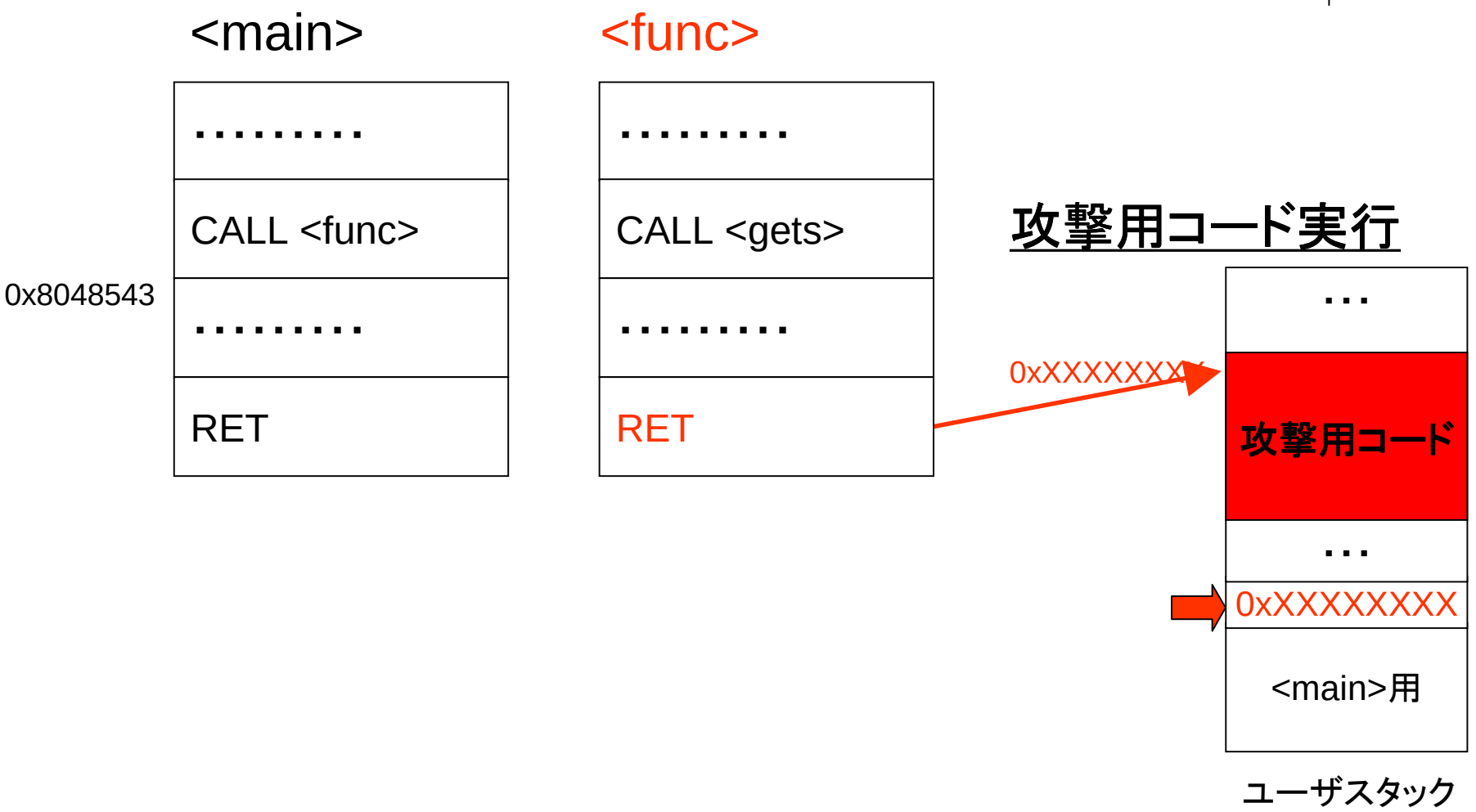


Stack-based Buffer Overflow

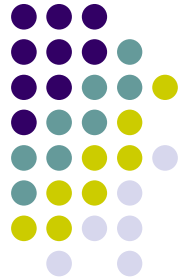




Stack-based Buffer Overflow

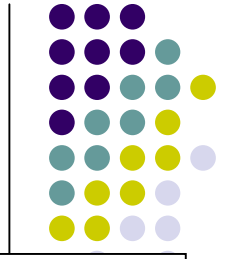


Heap-based Buffer Overflow



- ヒープ領域に確保されたバッファについて起こるBuffer Overflow のこと
- ヒープ領域に確保されたデータが上書きされる
- 各バッファが持つ管理情報が破壊、改ざんされる
- 攻撃者のターゲットは・・・
  - 攻撃用コードの追加
  - 関数ポインタの書き換え

Heap-based Buffer Overflow



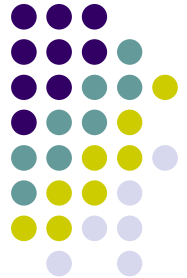
hbo.c:

```
#include <stdio.h>
#include <string.h>
#define BUFSIZE 8
int main()
{
    unsigned long diff;
    char *buf1, *buf2;

    buf1 = (char *) malloc(BUFSIZE);
    buf2 = (char *) malloc(BUFSIZE);
    diff = (unsigned long) buf2 - (unsigned long) buf1;
    printf("buf1 = %p, buf2 = %p, diff = 0x%x bytes¥n", buf1, buf2, diff);
    memset(buf2, 'A', BUFSIZE - 1);
    buf2[BUFSIZE - 1] = '¥0';
    printf("before overflow: buf2 = %s¥n", buf2);
    memset(buf1, 'B', (unsigned int) (diff + 3));
    printf("after overflow: buf2 = %s¥n", buf2);
    printf("after overflow: buf1 = %s¥n", buf1);
    return 0;
}
```

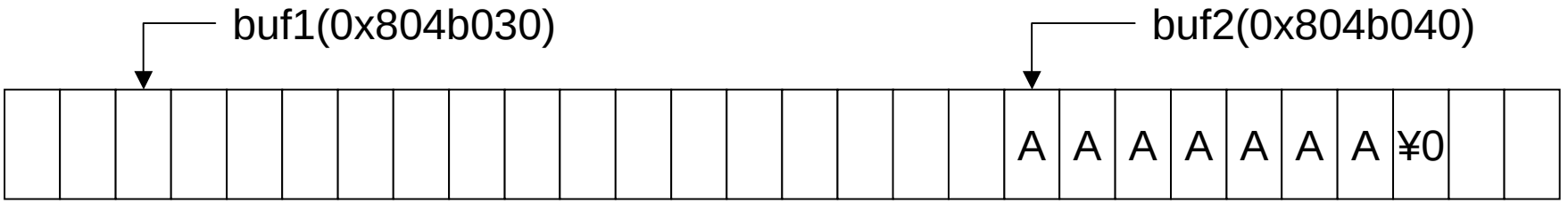
実行結果 (FreeBSD-4.10)

```
% cc -o hbo hbo.c
% ./hbo
buf1 = 0x804b030, buf2 = 0x804b040, diff = 0x10 bytes
before overflow: buf2 = AAAAAAA
after overflow: buf2 = BBBAAAA
after overflow: buf1 = BBBBBBBBBBBBBBBBBBBAAAA
```

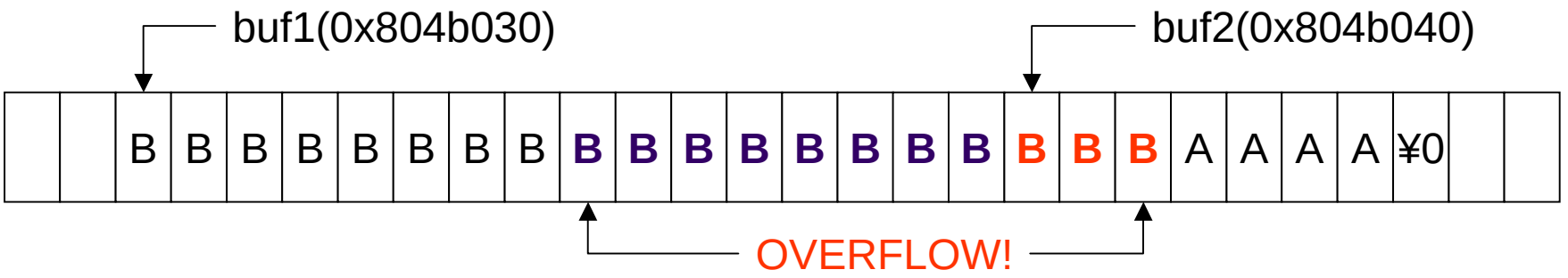
Heap-based Buffer Overflow



Overflow 前



Overflow 後



Heap-based Buffer Overflow

7.20.3 記憶域管理関数 calloc 関数, malloc 関数, 及び realloc 関数の連続する呼出しによって割り付けられる記憶域の順序及び隣接性は, 未規定とする。(以下略)

7.20.3.2 free 関数

形式

```
#include <stdlib.h>
void free(void *ptr);
```

機能 free 関数は, ptr が指す領域を解放し、その後の割付けに使用できるようにする。ptr が空ポインタの場合, 何もしない。それ以外の場合, 実引数が calloc 関数, malloc 関数若しくは realloc 関数によって以前に返されたポインタと一致しないとき, 又はその領域が free 若しくは realloc 関数の呼出しによって解放されているとき, その動作は未定義とする。

返却値 free 関数は, 値を返さない。

7.20.3.3 malloc 関数

形式

```
#include <stdlib.h>
void malloc(size_t size);
```

機能 malloc 関数は, 大きさが size であるオブジェクトの領域を割り付ける。割り付けられたオブジェクトの値は, 不定とする。

返却値 malloc 関数は, 空ポインタ又は割り付けた領域へのポインタを返す。

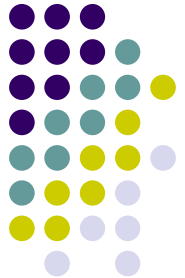
「JIS X 3010:2003 プログラム言語C」より

Heap-based Buffer Overflow

- malloc関数とfree関数の実装
  - 広く使われているものは数える程度
  - ヒープの未使用領域から必要十分なサイズのブロックを割り当てる
  - 管理情報（メタデータ）をデータ領域の前後に持つ

アルゴリズム	採用OS
BSD kingsley (Chris Kingsley) BSD phk (Poul-Henning Kamp) dlmalloc (Doug Lea) System V (AT&T) RTL Heap (Microsoft)	4.4BSD, AIX, Ultrix FreeBSD, OpenBSD libg++, GNU Hurd, Linux Solaris, IRIX Windows

Heap-based Buffer Overflow



- 処理系の実装ごとに攻撃手法は異なる
  - メタデータを書き換えることで攻撃につなげる
    - このような状態を memory corruption と呼ぶ

> **corruption –n.**
> 1 堕落; 弊風, 違法[腐敗]行為, 買収, 汚職.
> 2 腐敗;《古》腐敗させるもの, 悪影響;《方》膿(うみ).
> 3 ((原文の))改悪, 変造; ((言語の))なまり, 転訛.

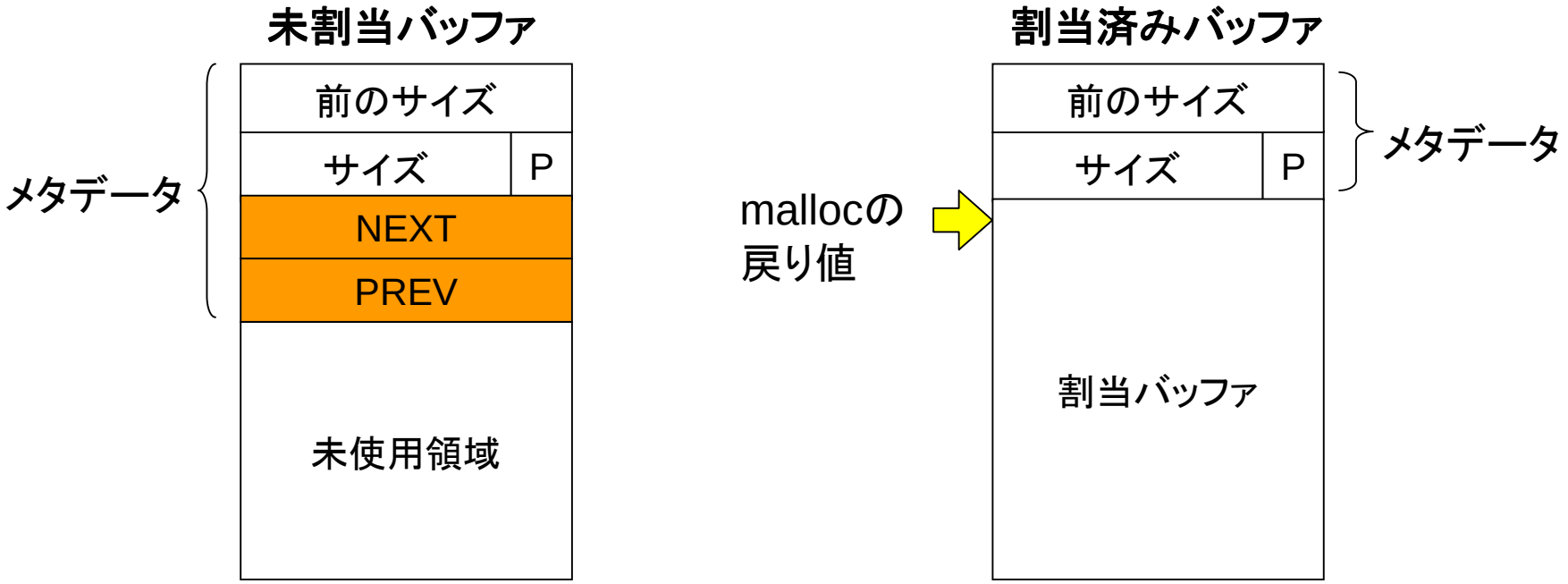
「リーダーズ英和辞典」より

  - たいていの実装で攻撃用コード実行を引き起こすことができる

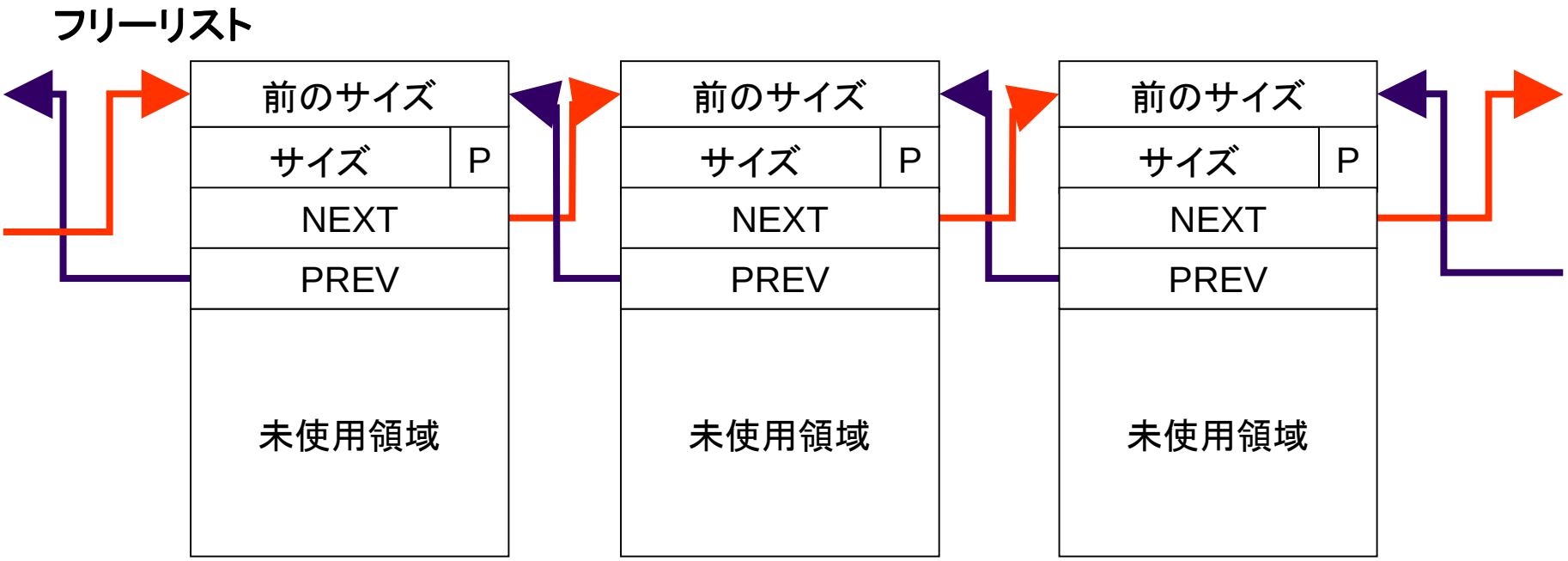
Heap-based Buffer Overflow

- dlmalloc の malloc と free
  - 未割当領域を、フリーリスト（サイズ毎に用意）と呼ぶ双方向リストで管理している
  - malloc は、指定されたサイズに必要十分な領域をフリーリストから払い出し、当該領域をフリーリストから外す
  - free は、不要になったバッファをフリーリストに戻す
  - malloc で払い出された領域には、利用できるデータ領域の前に、管理情報（メタデータ）が格納されている
  - malloc / free により、フリーリストにつながれた前後の要素中のメタデータが書き換えられる

Heap-based Buffer Overflow

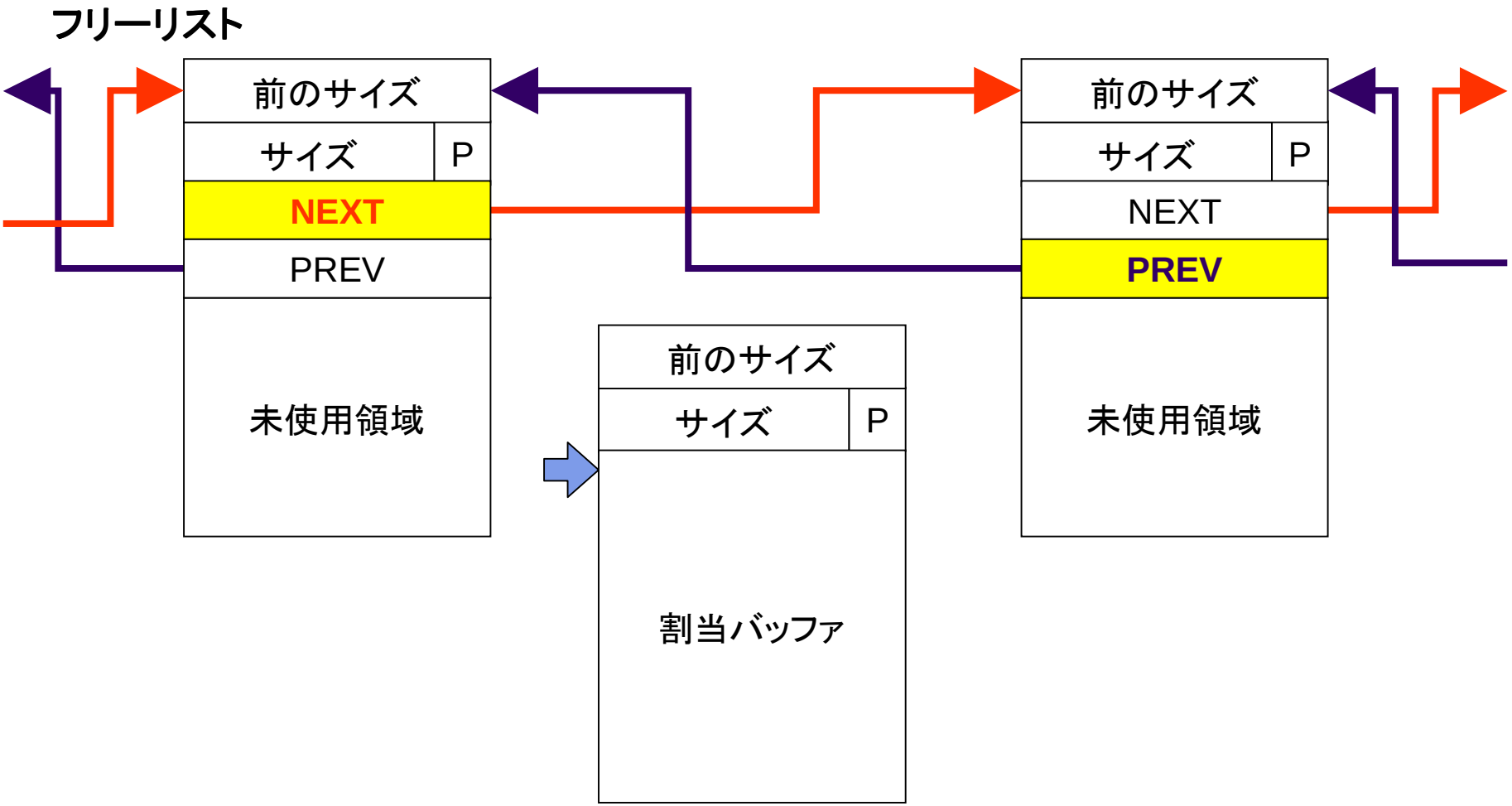


Heap-based Buffer Overflow

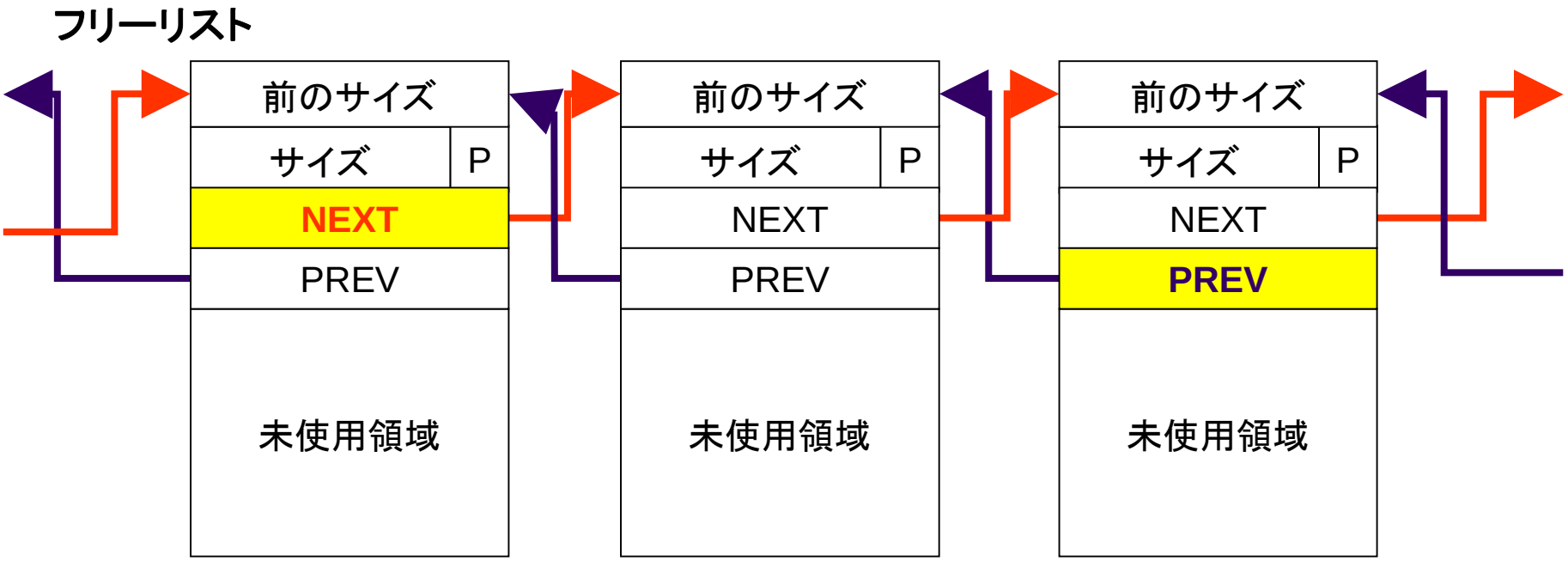


malloc 関数や free 関数では、フリーリストからのブロックの切り離しやブロックのつなぎ換えが発生する。

Heap-based Buffer Overflow

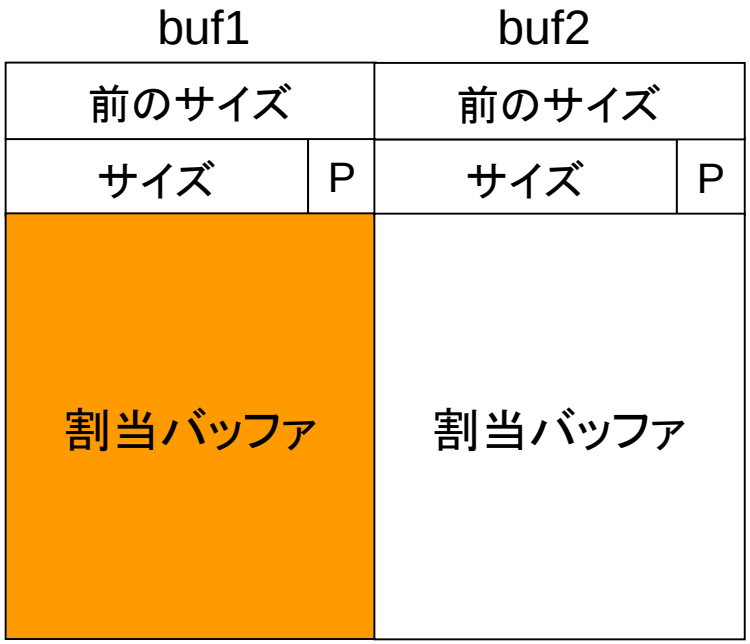


Heap-based Buffer Overflow

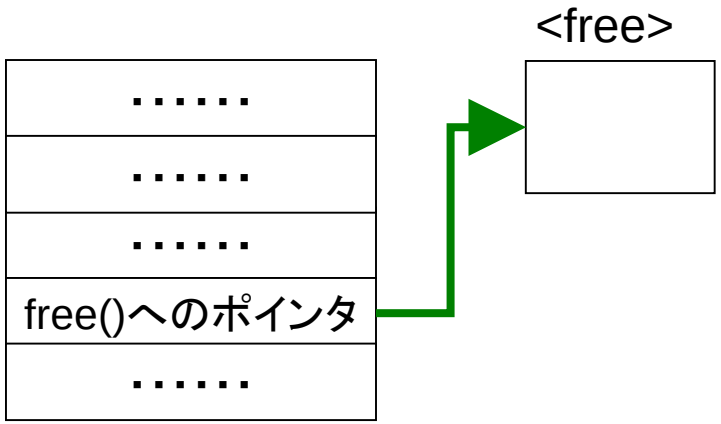


Heap-based Buffer Overflow

- ヒープ領域のバッファがあふれると、隣接する次のバッファ領域またはフリーリスト上の次の領域のメタデータを上書きできる

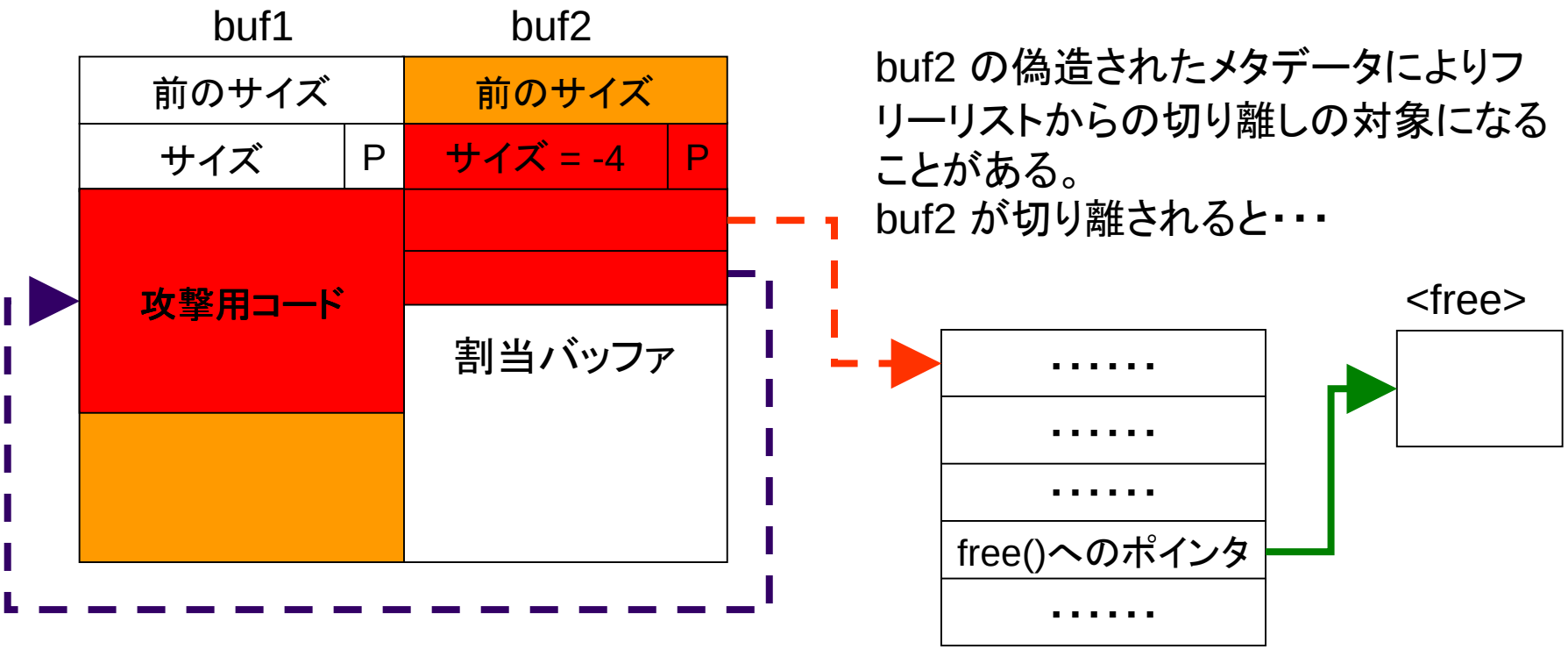


buf1, buf2 のブロックが隣接して割り当てられている場合、buf1 が overflow 可能だと、buf2 のメタデータ領域が書き換えられる。



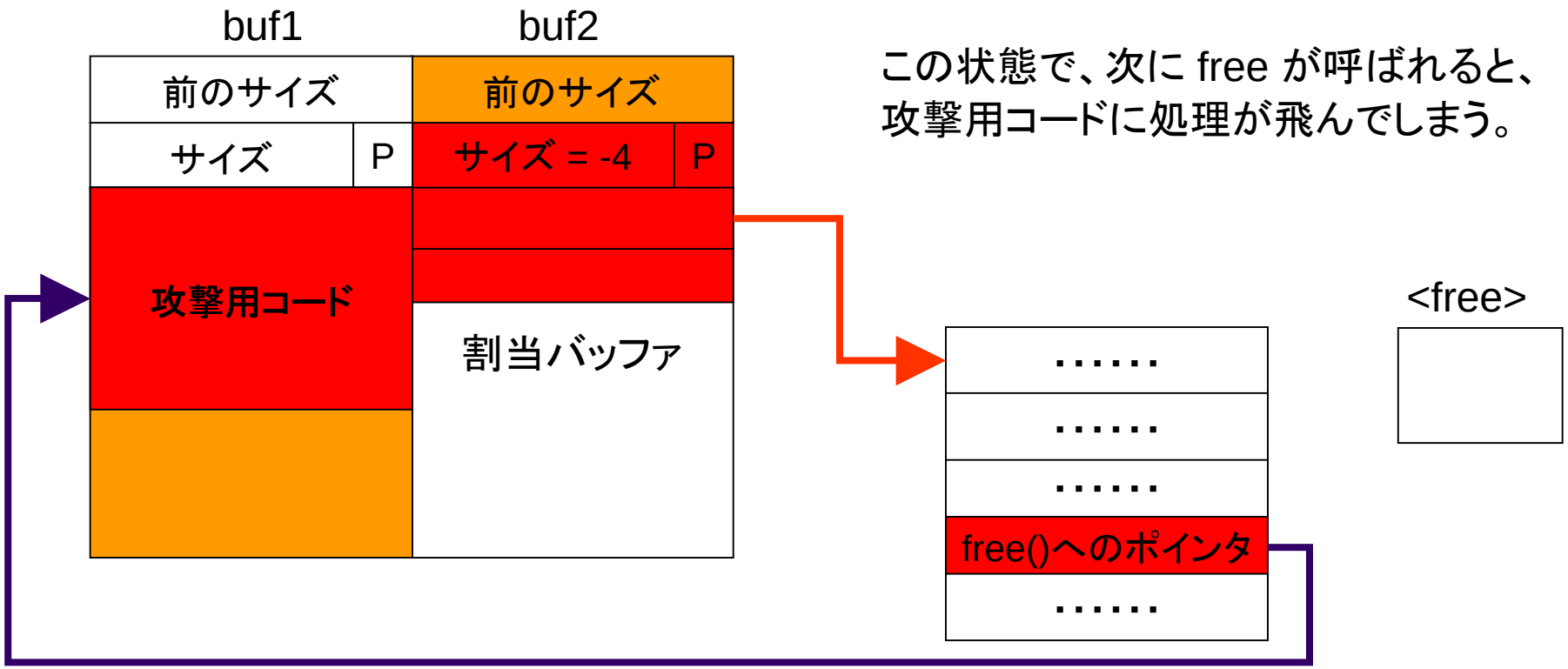
Heap-based Buffer Overflow

- ヒープ領域のバッファがあふれると、隣接する次のバッファ領域またはフリーリスト上の次の領域のメタデータを上書きできる



Heap-based Buffer Overflow

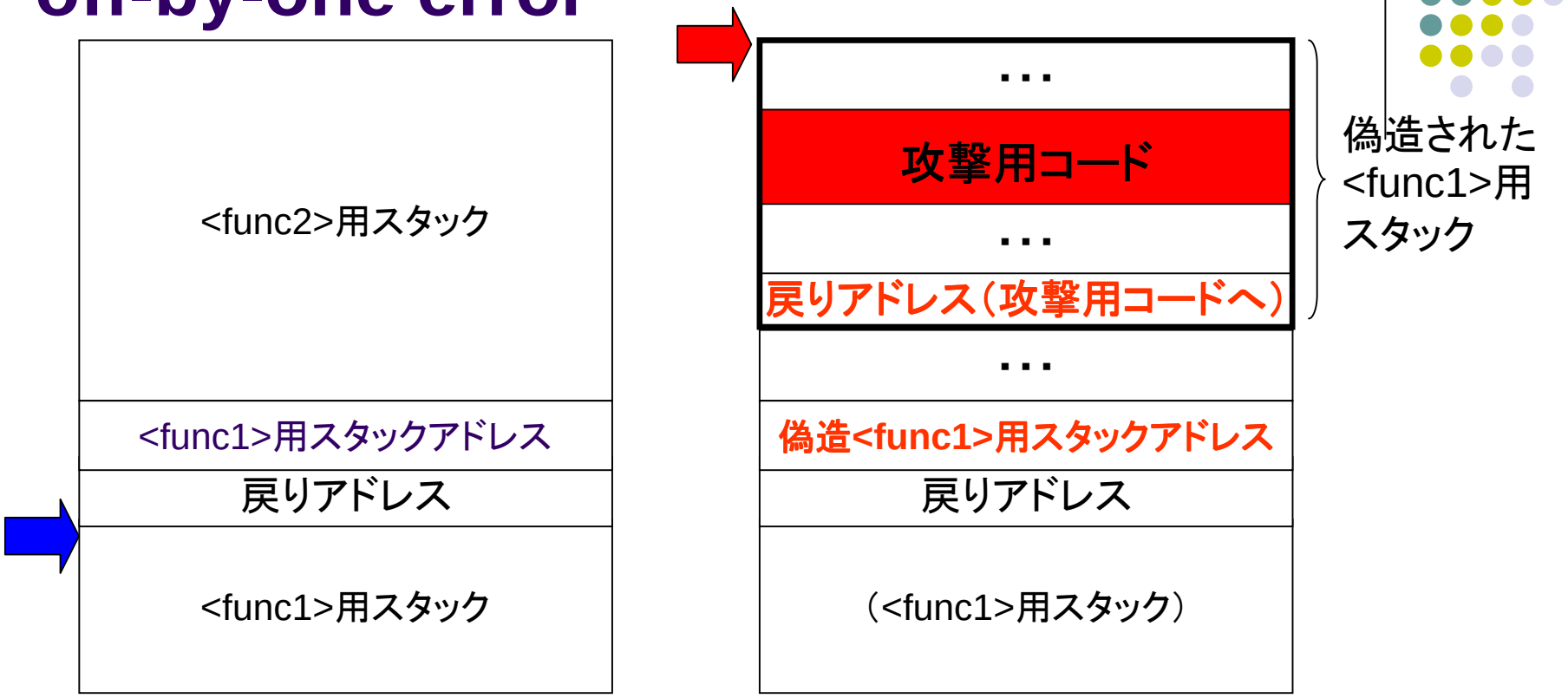
- ヒープ領域のバッファがあふれると、隣接する次のバッファ領域またはフリーリスト上の次の領域のメタデータを上書きできる



off-by-one error

- 1バイト / 数バイト程度の小さなサイズの Buffer Overflow
  - 「杭の間のフェンスの数は？」問題
  - 単純な数え間違いが原因
- あふれる量はわずかでも、攻撃につながることがある
  - 重要データの改変
  - 呼び出し元スタック領域の偽造 ➔ 攻撃用コード実行へ

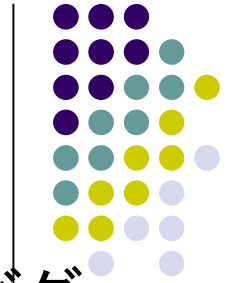
off-by-one error



- func1 から func2 が呼び出され、
- func2 用の配列がわずかにあふれる場合

- func2 のバッファに攻撃用コードを書き込む
- あふれる分で func1 用スタックアドレスを書き換える
- func2 が終了し、func1 に処理が戻ると、偽造されたスタックを利用するようになる
- func1 の処理が終わると、攻撃用コード実行

Double free Bug



- mallocで割り当てられたバッファを二重解放してしまうバグ
- 二重 free の動作は規格上未定義

7.20.3.2 free 関数
形式

```
#include <stdlib.h>
void free(void *ptr);
```

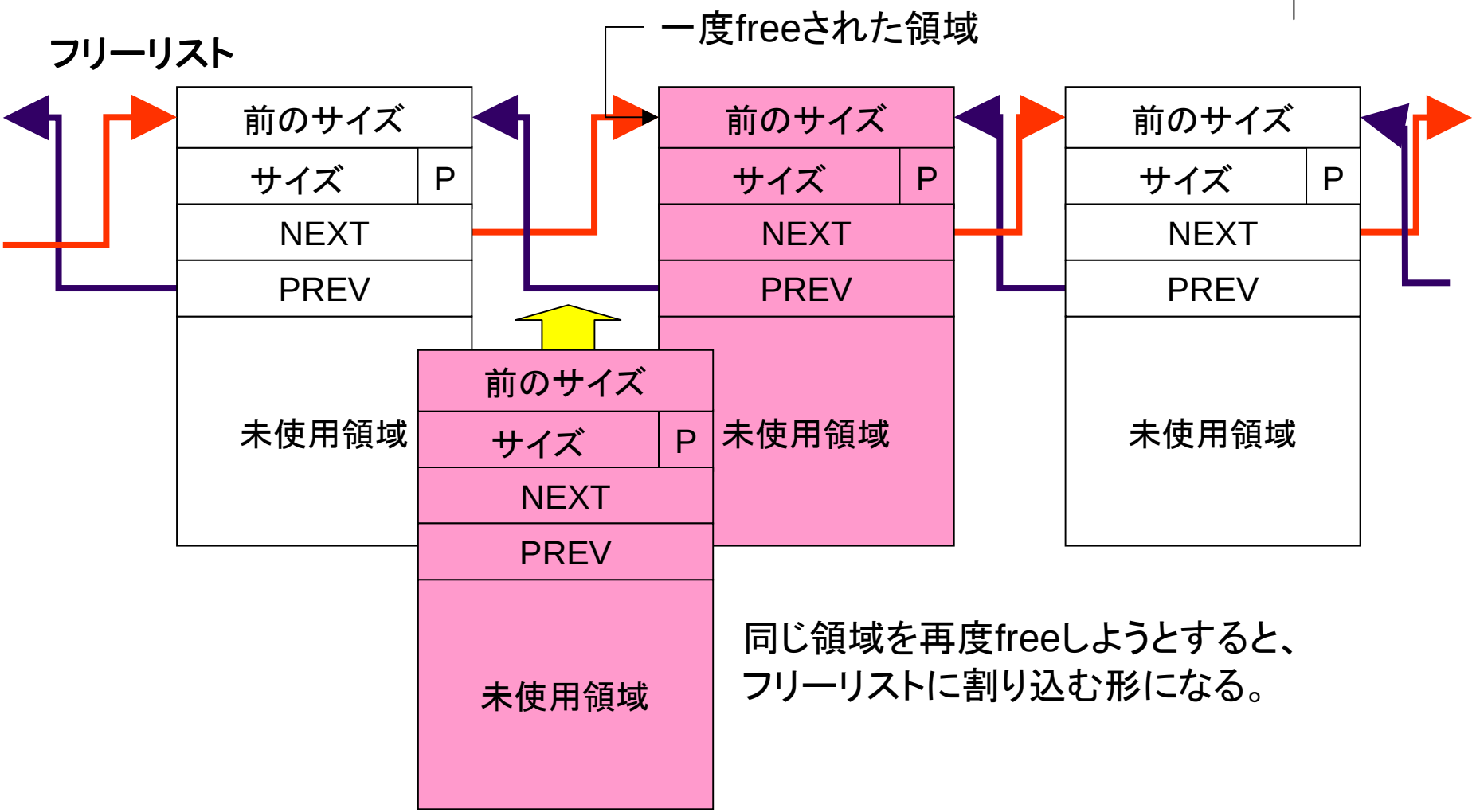
機能 free 関数は, ptr が指す領域を解放し、その後の割付けに使用できるようにする。ptr が空ポインタの場合, 何もしない。それ以外の場合, 実引数が calloc 関数, malloc 関数若しくは realloc 関数によって以前に返されたポインタと一致しないとき, 又はその領域が free 若しくは realloc 関数の呼出しによって解放されているとき, その動作は未定義とする。

返却値 free 関数は, 値を返さない。

「JIS X 3010:2003 プログラム言語C」より

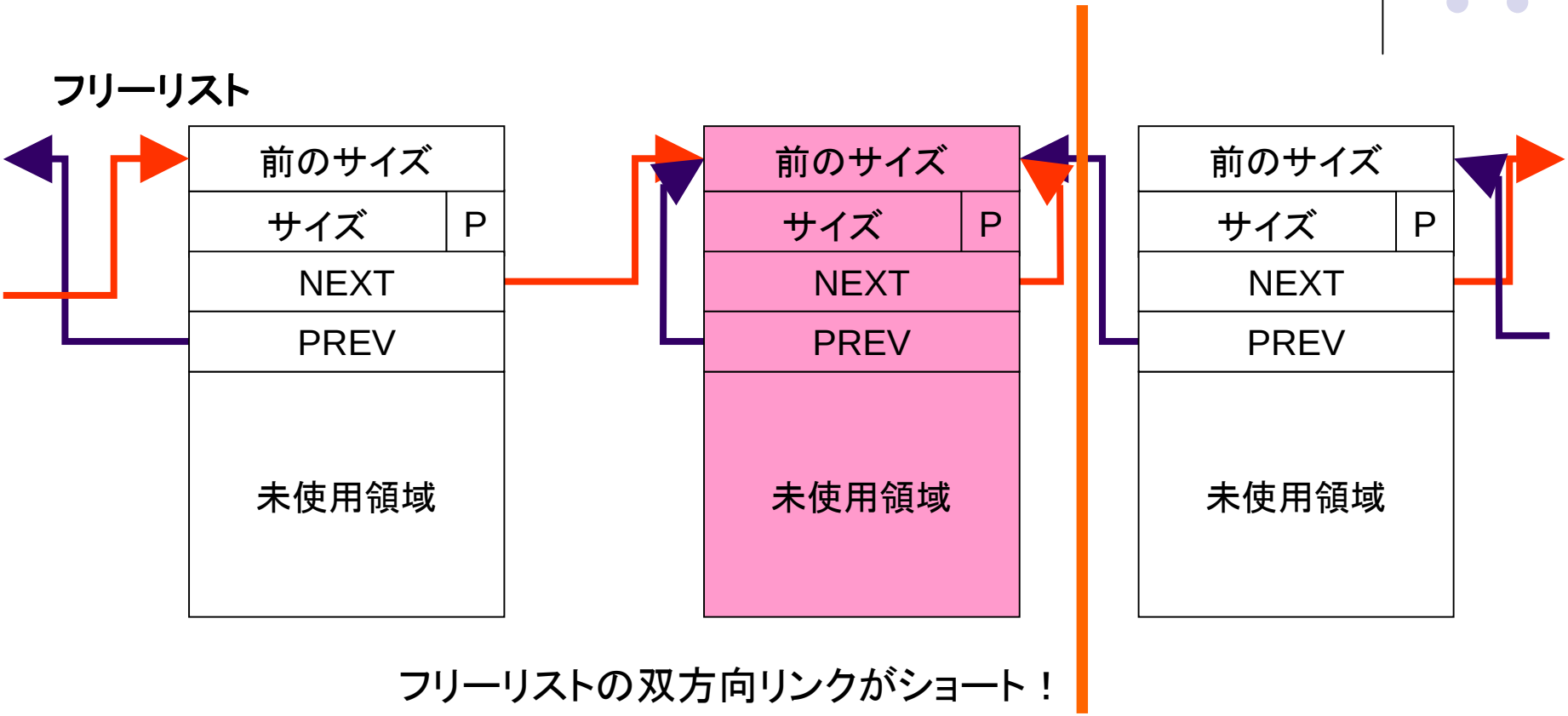
- 実装によってはバッファのメタデータが破壊される
  - memory corruption の発生
  - dlmallocの場合、フリーリストの双方向リンクがショートしてしまう

Double free Bug

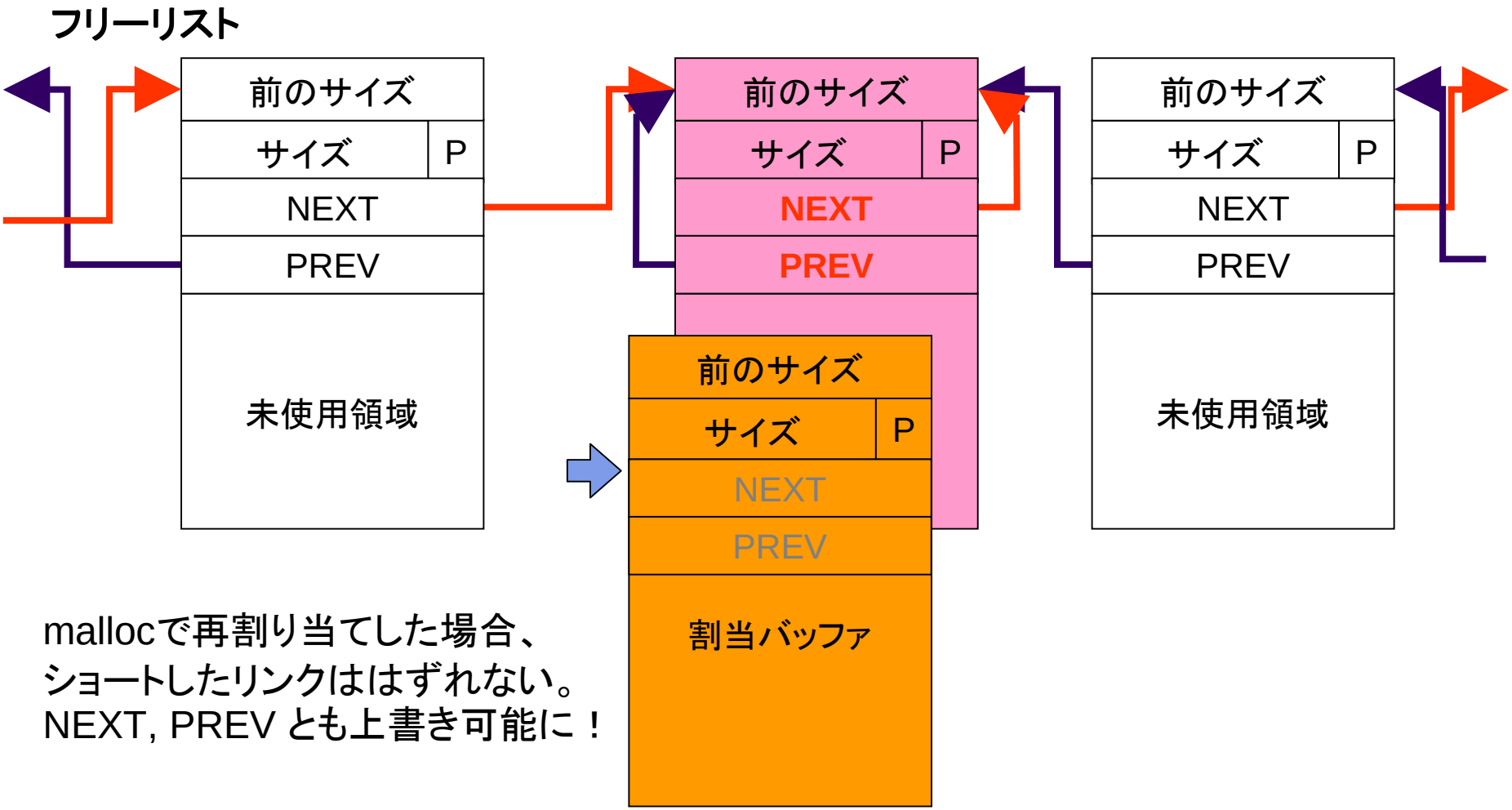


同じ領域を再度freeしようとすると、
フリーリストに割り込む形になる。

Double free Bug

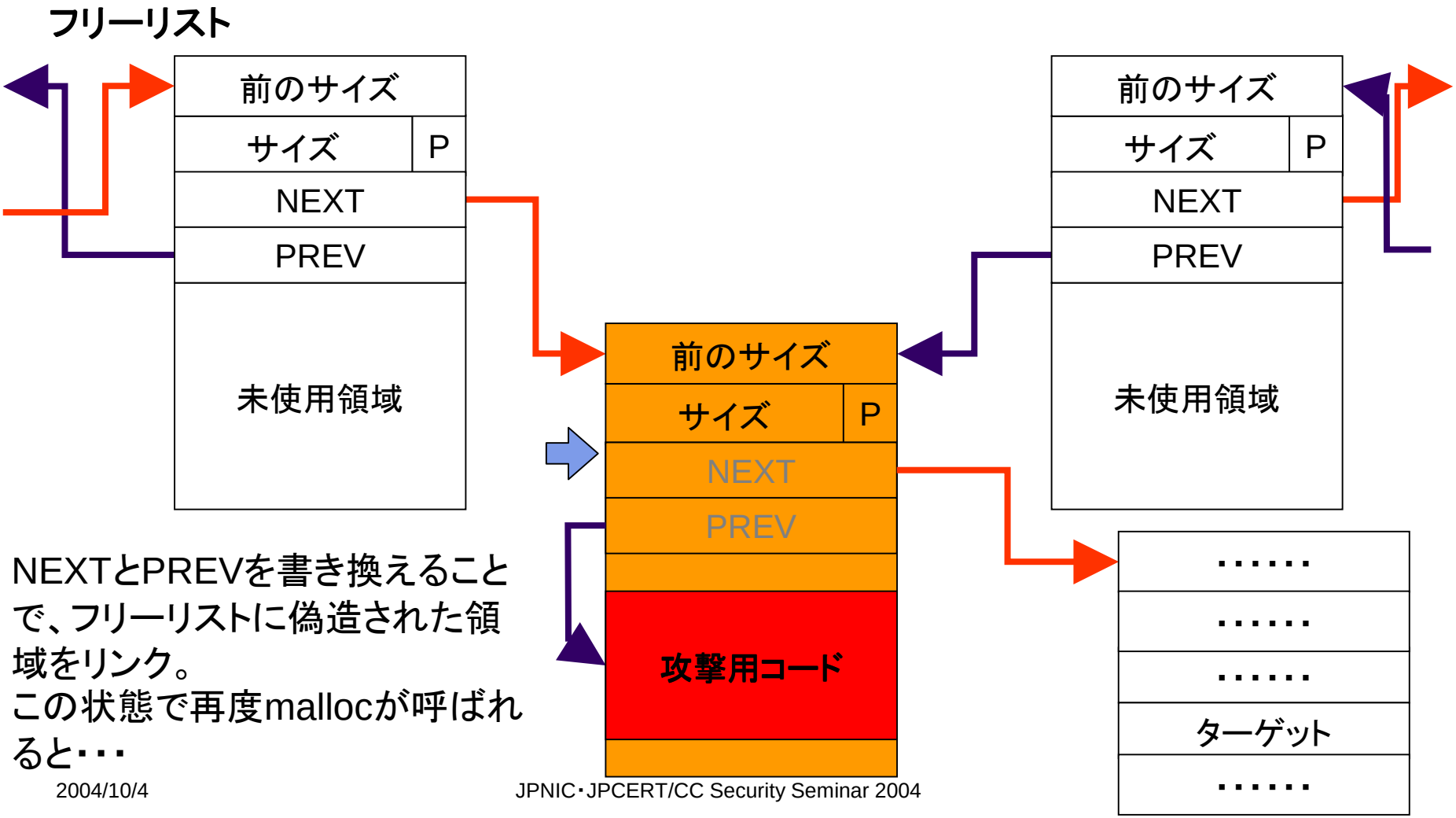


Double free Bug



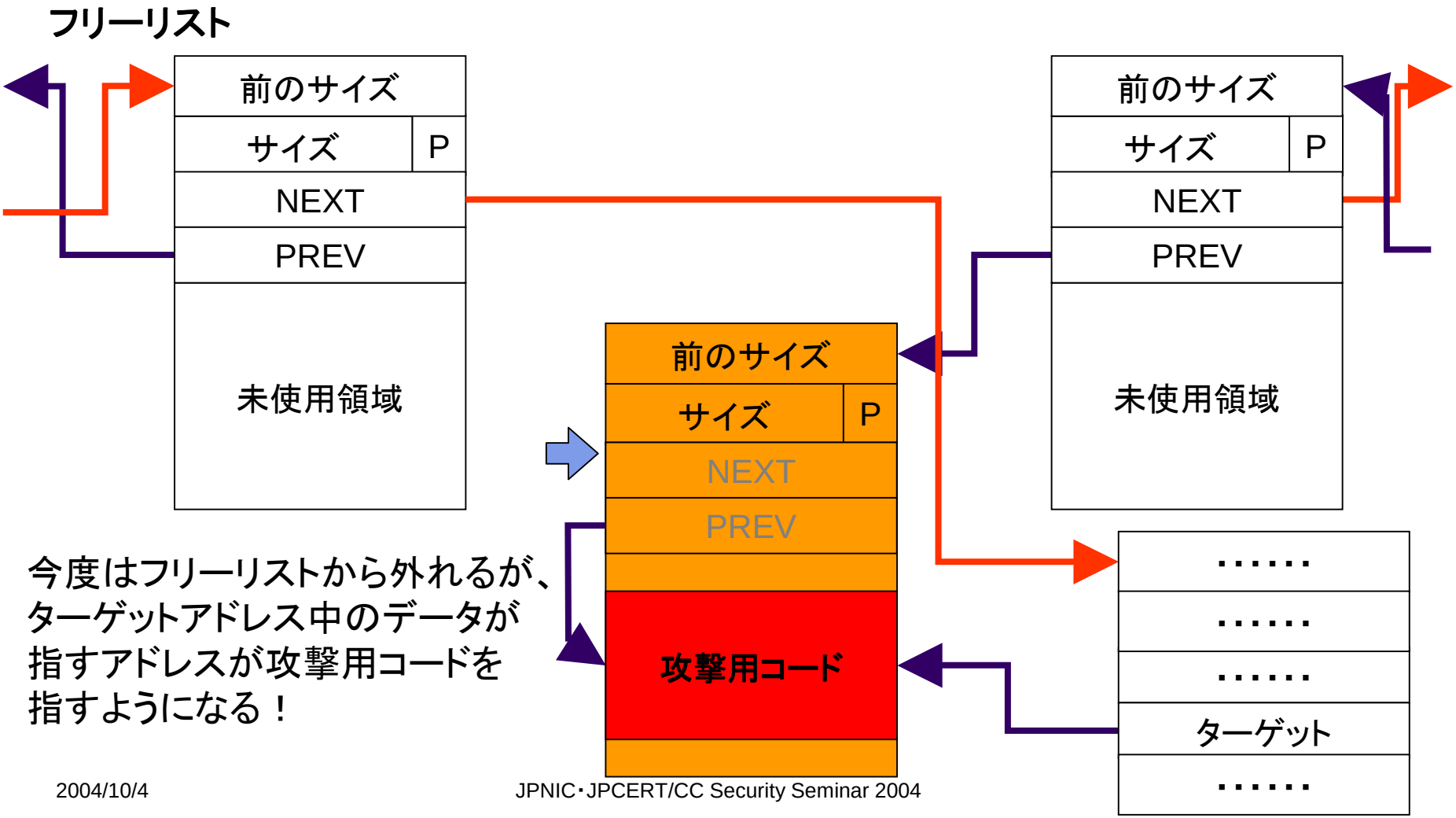
mallocで再割り当てした場合、
ショートしたリンクははずれない。
NEXT, PREV とも上書き可能に！

Double free Bug

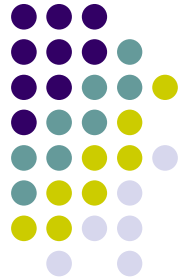


NEXTとPREVを書き換えることで、フリーリストに偽造された領域をリンク。
この状態で再度mallocが呼ばれると・・・

Double free Bug

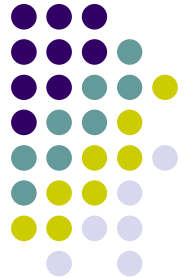


Format String Bug



- Format String とは「書式」を示す文字列のこと
- 書式付き入出力関数で書式を指定するのに使用
  - printf ファミリ (fprintf, printf, sprintf, snprintf, vfprintf, vprintf, vsprintf, vsnprintf)
  - setproctitle, syslog など
- 書式付き関数は、引数の数が不定
  - %で始まる指令の数だけ増える
  - 不足している場合の動作は未定義

Format String Bug



7.19.6.1 fprintf 関数

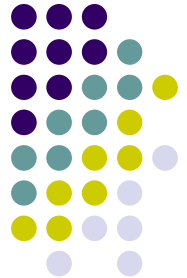
形式

```
#include <stdio.h>
int fprintf(FILE * restrict stream,
        const char * restrict format, ...);
```

機能 fprintf 関数は, format が指す文字列 (書式) の制御に従って, stream が指すストリームへ書き込む。format は, それに続く実引数の, 出力の際の変換方法を指定する。書式に対して実引数が不足しているときの動作は, 未定義とする。実引数が残っているにもかかわらず書式が尽きてしまう場合, 余分の実引数は, 評価するだけで無視する。fprintf 関数は, 書式文字列の終わりに達したときに呼び出し元に復帰する。
(以下略)

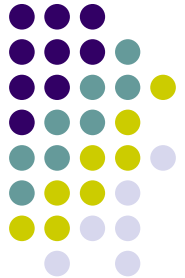
「JIS X 3010:2003 プログラム言語C」より

Format String Bug



- 例: fprintf(F, str);
  - 文字列 str に外部から入力した文字列が入るとアウト
  - ➔fprintf(F, **”%s”**, str);
- バグの発見が容易なのが特徴
- 任意のアドレス情報を参照可能になる
- 任意のアドレスにデータを書き込める

Format String Bug



fs1.c:

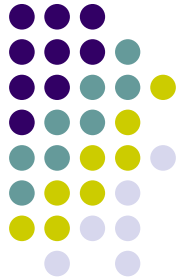
```
#include <stdio.h>

int main()
{
    fprintf(stdout, "%08x¥n");

    return 0;
}
```

実行結果 (FreeBSD-4.10)

```
% cc -o fs1 fs1.c
% ./fs1
bfbff708   ← どこかのアドレスの格納データ
%
```

- スタック上に引数があるものとして、引数があるはずのアドレスを参照している

Format String Bug



fs2.c:

```
#include <stdio.h>
int main()
{
    char fmt[1024];
    if (fgets(fmt, sizeof(fmt), stdin) == NULL) exit(1);
    fprintf(stdout, fmt);
    return 0;
}
```

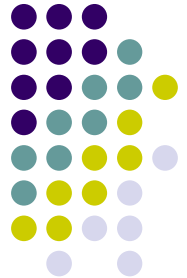
実行結果　(FreeBSD-4.10)

```
% cc -o fs2 fs2.c
% echo "AAA0AAAB_%08x.%08x.%08x.%08x.%08x.%08x.%08x" | ./fs2
AAA0AAAB_280ec580.00000090.00000000.0000058b.30414141.42414141.3830255f
%
```

スタック上のデータが参照された

('A':0x41 'B':0x42 '0':0x30 '8':0x38 '_':0x5f '%':0x25)

Format String Bug



7.19.6.1 fprintf 関数
(中略)
o,u,x,X unsigned int 型の実引数を dddd 形式の符号無し 8進表記(o), 符号無し 10進表記(u), 又は符号無し 16進表記(x 若しくは X) に 変換する。(以下略)
n 実引数は, 符号付き整数型へのポインタでなければならない。この fprintf の呼び出しで その時点までに出力ストリームに書き込まれた文字数を, この整数に格納する。(以下略)

「JIS X 3010:2003 プログラム言語C」より

- %n で、当該引数に相当するアドレスに、整数データが書き込まれる
- 多くの処理系で、書式文字列で任意のアドレスを指定して、アドレス内容参照や、%n を用いたデータ書き込みが可能
  ➔攻撃用コードの実行につながる

《まとめ》

- ぜい弱性は、規格上未定義の部分の処理動作に潜む
  - 処理系では、起こらないものとして実装されていることが多い
  - 未定義の状態に陥らないようにするのはプログラマの責任
    - そのような状態に陥るのは単なるバグ（ただし致命的）
- バグがなければ触れないアドレスのデータ書き換えが鍵
  - 管理情報の書き換えが、任意のアドレスの書き換えにつながる
  - 関数のポインタやジャンプ先が変わると、攻撃用コードが実行される

参考文献

- Cの標準規格
  - ISO/IEC 9899:1999 Programming Language C
  - JIS X 3010:2003 プログラム言語C
- 「Cプログラミングの落とし穴」Andrew Koenig（新紀元社）
- 「エキスパートCプログラミング -- 知られざる C の深層」Peter van der Linden（アスキー出版局）
- 「はじめて読むPentium マシン語入門編」蒲地輝尚, 水越康博（アスキー出版局）
- 「Linkers & Loaders」John R. Levine（オーム社）
- 「The Design and Implementation of the FreeBSD Operating System」Marshall Kirk McKusick, George V. Neville-Neil（Addison Wesley）
- 「The Shellcoder's Handbook Discovering and Exploiting Security Holes」Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel, Mehta, Riley Hassell（WILEY）
- 「EXPLOITING SOFTWARE HOW TO BREAK CODE」Greg Hoglund, Gary McGraw（Addison Wesley）
- Phrack Magazine（http://www.phrack.org/）
- 「Exploiting Format String Vulnerability」scut / team teso （formatstring-1.2.pdf）
- 「A Memory Allocator」Doug Lea（http://gee.cs.oswego.edu/dl/html/malloc.html）