CS11 BINARY EXPLOITATION – LECTURE 3

Heap Overflows, More Return Address Fun

Review

- All about memory corruption!
  - Out of bounds indexing
  - Buffer overflows
  - Different methods of accomplishing the same thing
    - Overwriting the instruction pointer (via a return address).

This Week

- Start out by going over another type of memory corruption
  - Heap Overflows
- Talk about what we can do when there *isn't* a perfect function already in memory.

Heap Overflows

- Last week, we talked about buffer overflows on the stack.
  - Exploitation path is clear – just overwrite return address higher up on the stack!
- Exploiting an overflow in memory that’s not stack – for example, on the heap, seems less obvious
  - …but we can do it anyway ☺

Heap Review

- Heap is some (usually physically contiguous) chunk of memory.
  - malloc() gets a chunk of memory, free() releases it.
- It would be expensive for the kernel to have to manage heap chunks…so this is done in user mode!
- Memory chunk typically has the following layout:
  - Header
    - Size
    - Next chunk
    - Previous Chunk
  - Actual Memory
- Chunks will be laid out sequentially (usually, anyway).

Heap – Exploitation Techniques

- Suppose we have an overflow in an allocated bit of heap memory.
- [our chunk's metadata] [our chunk] [chunk X's metadata]
- If we overflow past the end of our chunk, we can control some of chunk X's metadata!
    - How can we abuse that?

Heap – Exploitation Techniques

- Chunk X will be in some linked list (of free blocks, or allocated blocks) – which list doesn’t matter too much.
- Its next and previous pointers are interesting.
- Consider the following code, which usually runs when a chunk will get “unlinked” from its current list (when a free block is allocated, or an allocated block is freed):

```
void *prev = chunk->prev;
void *next = chunk->next;
prev->next = next;
next->prev = prev;
/* { more stuff happens here } */
```

Heap – Exploitation Techniques

- Chunk X will be in some linked list (of free blocks, or allocated blocks) – which list doesn’t matter too much.
- Its next and previous pointers are interesting.
- Consider the following code, which usually runs when a chunk will get “unlinked” from its current list (when a free block is allocated, or an allocated block is freed):

```
void *prev = chunk->prev; // Overflow: we control prev/next
void *next = chunk->next;
prev->next = next; // We control this!
next->prev = prev;
/* { more stuff happens here } */
```

Heap – Exploitation Techniques

- Limitations:
  - With doubly-linked list, both prev/next must be addresses.
  - We want to overwrite stuff on stack though…so this shouldn't really be a problem ☺
  - Much simpler with singly-linked list metadata, but this isn't standard.
- Apart from that…it's a pretty much arbitrary write!
- Use it **exactly** the same way you'd use out-of-bounds write from last week.
  - With clever tricks, you can even sometimes get an arbitrary read.
    - Cause free list to point into memory you want to read…"allocate" the memory you're interested in.

Back to Return Addresses

- With that out of the way, we can get back to the really interesting problem at hand:
- How can we get arbitrary code execution if there isn't already a perfect function for us to jump to in memory?
  - Lots of ways! ☺
- Today we'll be going over two in particular: "shellcode", and "return-to-libc".

Shellcode

- Hacking terminology for **code an attacker has gotten into memory as program input.**
- The idea is pretty obvious.
    - “If there isn’t code to do what I want in memory already, I’ll just put my own in and jump to it!”

Shellcode - Idea

- You write an assembly payload to do what you want beforehand, and compile it.
  - Usually… system('/bin/sh');
- When program is prompting for input, you provide the assembly payload
  - Payload ends up on the stack, or in the heap
- When you overwrite a return address, you tell the program to jump to the payload you wrote.
- ???
- Profit!

Shellcode - Caveats

- There are some pretty serious problems with shellcode:
  - Modern systems protect against executing data, especially data on the stack.
  - You need to know the address on the stack where your code will run from.
- We’ll be ignoring both of these, for now.
  - HW problems will be compiled with “-z execstack”.
  - To deal with knowing where code is…run programs with “setarch $ (uname –m) –R ./hw_program”.
    - This will disable all address layout randomization, stack will be at static address.

Shellcode – Caveats

- Shellcode *is* usable on a lot of platforms…but not on a modern OS.
  - Even if we ignore problems for educational reasons, we want to do better!
- Let’s go back to the big idea from last week: “Maybe we can find something usable already in memory?”

What’s in Memory?

- Program might not have some perfect function we can jump to.
  - In fact, it almost certainly doesn’t. People are bad at writing C, but they aren’t *that* bad.
- What’s in memory?
  - Program’s code
  - Program’s stack
  - Program’s heap

What’s in Memory?

- Program might not have some perfect function we can jump to.
  - In fact, it almost certainly doesn’t. People are bad at writing C, but they aren’t *that* bad.
- What’s in memory?
  - Program’s code
  - Program’s stack
  - Program’s heap
  - **Libraries the program links against.**

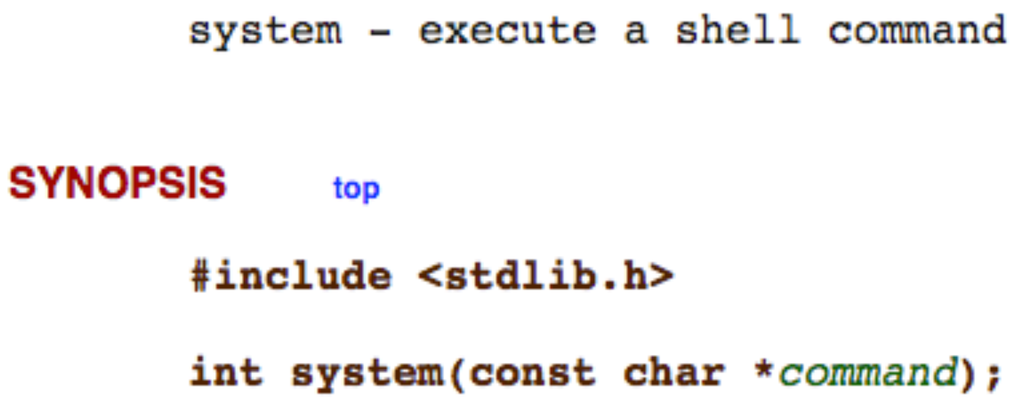
External Libraries

- Developers like to share code, and bundle up the results for other to re-use.
  - .dlls on windows, .sos on linux – same concept.
- These are a big attack surface!
  - If a library has something we can use, then we don’t have to care about
- Most programs don’t use too many .dlls/.sos…but almost all of them use one in particular!

The Standard Library

- The C standard library (libc) is an amazing attack vector for us.
- Many, many, many programs will #include <stdlib.h>…and end up getting linked with a copy of libc.
- For most programs, all that libc really does is load main().
  - …but the whole library will be in memory, anyway!
- What else gets put in memory when a developer types #include <stdlib.h>?

#include <stdlib.h>



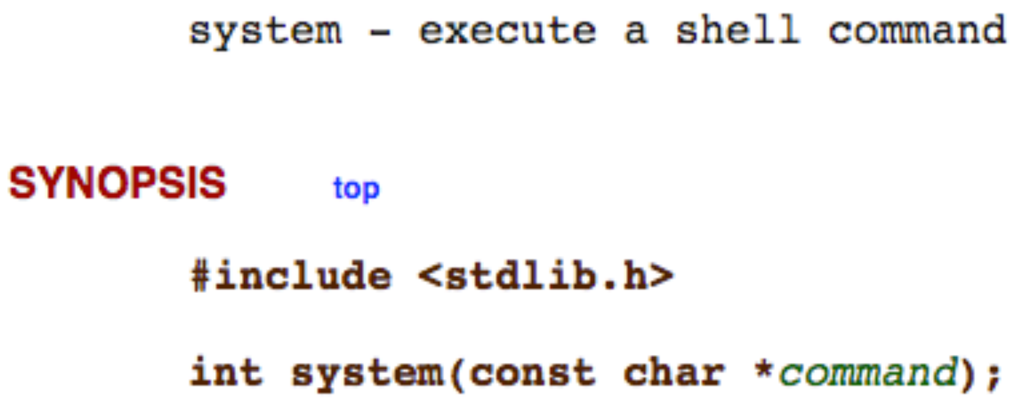
```
system - execute a shell command
```

SYNOPSIS top

```
#include <stdlib.h>

int system(const char *command);
```

#include <stdlib.h>



```
       system - execute a shell command
```

SYNOPSIS top

```
       #include <stdlib.h>

       int system(const char *command);
```

- Nice.

Return-to-libc

- system() is *exactly* the function we want to call (with "/bin/sh" as an argument)
- Turns out, every C program using libc (the standard library) has a copy of system() in memory.
- Even better: Many builds of libc have system("/bin/sh") somewhere in memory
  - Not a problem even if they don't, since we can usually control the arguments to a function

Return-to-libc

- Our attack is simple.
- libc is somewhere in memory.
  - If we know where it is (randomization is disabled), great!
  - If not…libc is what calls main(). The return address main() jumps to points to libc!
    - We can use this to figure out where it is, with some kind of arbitrary read.
    - Kind of complicated, though, you guys don't worry about this for now.
- If we know where libc is, we can jump to system("bin/sh") in libc!
- "Problem solved!" ☺

Return-to-libc

- Of course, this isn’t perfect.
  - This is a super obvious attack. And C developers won’t like it.
  - Interestingly, it’s not really directly mitigable!
    - Choices are basically “don’t use the standard library” or “deal with this”.
  - It gets mitigated in other (not super effective) ways.
    - Stack guards, for example, which I’ll be talking about next week.
- Ret2libc is as complicated as we’ll make things, for a few weeks.
- Still…we can do better!
  - There are still ways to do what we want even when it’s not an option!
  - The last set in particular will focus on the follow-up to ret2libc.

This Week's Set

- One problem: shellcode!
  - It's basically last week's buffer overflow problem…without the perfect spawn_shell() function.
- There was originally a heap overflow problem in this set, too.
  - I got told it was "too complicated" when these sets were tested, heh…
  - I'm working on simplifying it. Plan was to include it Set 4 and have Set 3 be 1-week set, but there have been requests to extend Set 3.
  - May make the heap overflow problem "optional".
    - Don't want to make Set 4 have too much content, even if two weeks are allowed. Currently, Set 4 is a function pointer overwrite problem, and a ret-to-libc problem with stack guards turned on.
- I am very open to suggestions on how to make everyone happy.
  - Waiting to push out Set 3 until after I give lecture, so we can figure out the plan before people start working on stuff.
- AWS IP should go live (for this set and set 2) on Wednesday or Thursday.

Next Week

- More advanced techniques!
  - We’ll still be using return-to-libc, but more ways of getting control of the instruction pointer.
  - C++ objects, vtables, and function pointers!
  - Printf (and string formatting, generally) vulns.
- We’ll also talk about stack guards, one of the current standard defenses against stack overflows.
  - Spoilers: they’re not perfect.
- Questions? Feedback?