COMPSCI 390R

Exploit Mitigation

UMASS CYBERSEC CLUB

Topics to Cover:

1. Project 3 hotfix
    - Challenge 2 had a ‘it works on my machine’ bug, should be all set

2. Thursday’s Lecture is a lab

UMASS
CYBERSEC
CLUB

Recap of what we’ve learned last class:

Format Strings/GOT & PLT

- Format String bugs can lead to memory leaks or arbitrary memory writes
- Learned how dynamic compilation works with the GOT and PLT
- Can exploit the GOT through write and reads

GOT/PLT Review

UMASS CYBERSEC CLUB

Reminder

Global Offset Table

- Has dynamic linker info and function offsets loaded into memory during runtime

Procedure Linkage Table

- When you call a library function you really call the PLT entry for it.
  - ex. calling fgets redirects you to fgets@plt
- Is a table of GOT offsets that redirect execution to the dynamic linker
- Dynamic linker will populate the GOT if it’s the first call, else it just redirects to the GOT offset

Exploiting the GOT/PLT

Leaking LIBC addresses

- If we can leak a function address from the GOT, we can calculate where the libc is loaded based off the function offset

Writing to the GOT

- If we can overwrite a GOT offset we can:
  - Write the address of another function in LIBC
  - Write the address to shellcode or another function in the program that we want to execute (think a decryption function)

UMASS CYBERSEC CLUB

Attacks & Mitigations

UMASS CYBERSEC CLUB

Why we have/need mitigations:

Getting hacked kind of sucks:

- If we have the tools to stop vulnerabilities we should take them
- Even if we can't force people to write perfect code, we can have mitigations that make exploits harder
- We can't have a perfect system though, there's always a trade off of security vs performance

Buffer Overflows:

Easiest attack to exploit:

- Can change local variables in a function
- Can redirect execution to other locations in memory
- Can write shellcode and redirect back to it later

What can we do to stop this?

Buffer Overflows (fix):

Data Execution Prevention/No-Execute (DEP/NX)

- Assign memory regions Read/Write/eXecute permissions
  - Can use gdb command ‘vmmap’ for the Virtual Memory Map and its permissions
- Sections that are data, stack/heap/globals should not have the eXecute flag!

This stops a large amount of possible shellcode attacks!

UMASS CYBERSEC CLUB

Buffer Overflows (fix 2):

There are still problems

- We can still overwrite local variables
- We can still redirect execution! (This one is huge!)

Possible solutions?

UMASS CYBERSEC CLUB

Buffer Overflows (fix 2):

Reordering the Stack

- Reorder values on the stack such that buffers are above all variables

This should stop us from overwriting local variables

What if there are multiple buffers though?

Sometime the compiler gets this wrong

UMASS CYBERSEC CLUB

Buffer Overflows (fix 2):

Stack Canaries/Stack Cookie

- Add a dummy ‘canary/cookie’ variable right before the base pointer
- Check if the variable has changed at the end of the function, if so throw an error!
- Typically the stack canary starts with a NULL-byte since this stops some string-based overwrites
- Always long enough to avoid brute forcing

UMASS CYBERSEC CLUB

How can we bypass these?

Bypassing DEP/NX and Stack Canaries

Bypassing DEP/NX

- We can no longer execute shellcode on the stack, but ROP chains make use of existing gadgets from executable code sections so they are not affected
- Find alternative sections that are writable and executable to put shellcode in (or make your own if you can call mprotect)

Bypassing Stack Canaries

- Find a way (printf or other alternatives) to leak the value from the stack and then add it to your buffer overflow/rop-chain
- Some vulnerabilities like out of bounds array accesses may be able to just leave the canary intact with their overflow

UMASS CYBERSEC CLUB

Our best attack thus far

Ret2Libc

1. Find a buffer overflow that we can abuse in the program
2. If stack canaries are enabled, look for a leak such that we can find out its value and modify our ROP chain so it doesn't modify the canary
3. Use gadgets to setup parameters to call a function such as system("/bin/sh")
4. Use a gadget to call system in LIBC (Should be loaded in the same location every time with the current mitigations)

UMASS CYBERSEC CLUB

Randomization Mitigations

ROP:

Little harder than buffer overflow, still relatively easy:

- Can call any code in the program just like buffer overflows
- Can use gadgets to execute shellcode without executing stack memory as long as we know the address the opcodes are at

What can we do to stop this?

UMASS CYBERSEC CLUB

ROP (fix 1):

Address Space Layout Randomization:

- Randomizes the location of the stack, heap and any loaded libraries
- Usually always on and built into the kernel now on most operating systems (its been on for all challenges thus far)

This in tangent with PIE makes finding gadgets relatively hard

Bypassing ASLR:

Bypassing ASLR with Ret2PLT

1. Find a buffer overflow that you can abuse with a ROP chain
2. Call a function such as "puts" by calling its address in the PLT.
3. Pass the GOT entry for "puts" as the parameter so when we call the PLT, we end up printing the address of "puts"
4. If we know the libc version we can calculate where the base is by subtracting the puts offset from the leaked address
5. Recall the function that gave you the buffer overflow, ROP again to perform a Ret2Libc

UMASS CYBERSEC CLUB

ROP (fix 2):

Position-Independent Code (PIE):

- Instead of hard coding locations of data and opcodes, compile with all return values and data access set to offsets related to a base address
- Our code can now be loaded anywhere in memory if we want to, but we still need a system to do that for us.
- If ASLR is also enabled, it will randomize the location of the code, data, and GOT/PLT

This does nothing on its own, with ASLR though it’s very annoying

Bypassing ASLR/PIE

Leaking the base of the program

- Use printf leaks to leak values such as return addresses or addresses of globals can help us find out where in memory the program is located
- Take that address, calculate the offset by seeing where it is in gdb, then add the difference in your payload’s offsets (Ret2PLT or Ret2Libc)
- Running exploits locally we can double check if our offset is

One more defense mechanism:

Partial Relocation Read-Only:

- Usually always on, puts the GOT before any global variables so if you can cause a buffer overflow on a global variable, you can't change the values of the GOT

Full RELRO:

- Loads all GOT entry values at the startup of the program and not dynamically, then sets the GOT to read only
- This usually isn't on by default as this can cause the program to take a while to startup if you're loading thousands of symbols during

UMASS CYBERSEC CLUB

What our new exploits need to look like:

NX + Canaries:

- Find a memory leak to retrieve the canary, else you can only control local variables (if the buffer location isn't changed), then ROP

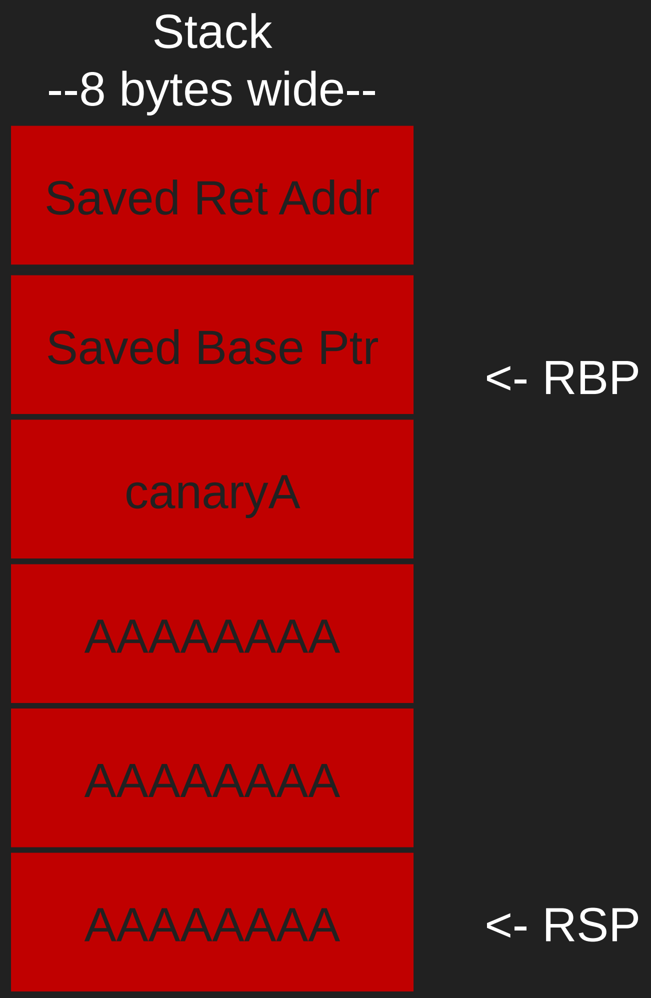
ASLR + PIE:

- Leak address of program or libc
    - If program, use program gadgets or now calculate GOT location

Extra Memory Leak Techniques:

Overwriting Null Bytes:

- If a buffer is read as a string and we can overwrite null bytes on variables higher up and they will also be printed as a string



UMASS CYBERSEC CLUB