Week 07
PWN II: (Somewhat More) Modern Binary Exploitation

Kevin

Meeting Flag

sigpwny{AAAAAAA\x06\x75\x72\x24\x73\x7f}

Announcements

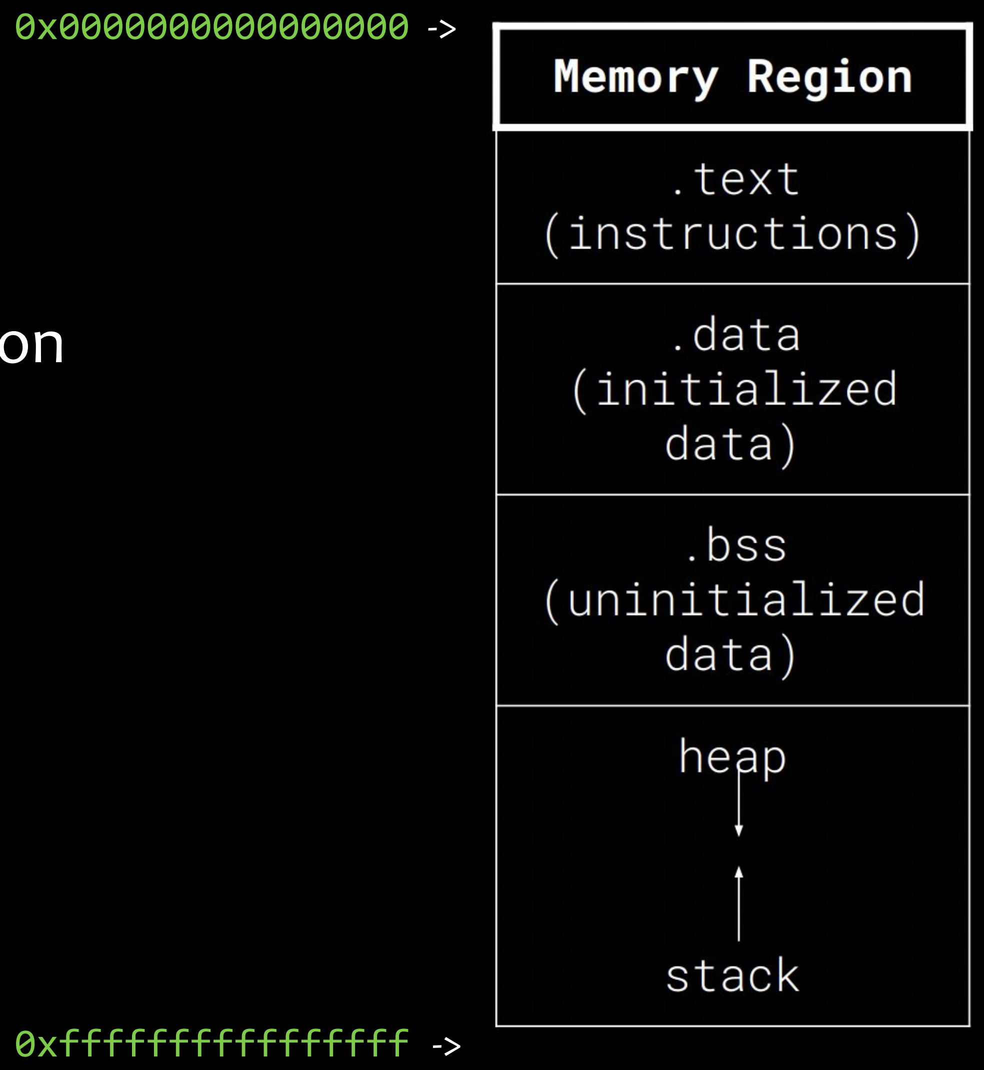
- Hello From Purdue!

- Recruiting CTF OCT 23, tell your friends!
    - Big advertisements going out tomorrow :)

- Halloween Get Together!
    - Your families are invited to meet! Costumes encouraged :)

Modern Binary Exploitation

- Key differences from Thursday:
  - 64-bit binary
  - Mitigations against common attacks
    - ASLR
    - NX
    - Stack canary
    - RELRO



ASLR + PIE

- Address space layout randomization
  - Randomized stack, heap, and shared library addresses
- Position independent executable
  - Randomized program addresses
- Bypassed with leaks
  - Many, many ways to obtain these
  - Usually program-specific
  - For the purpose of this presentation, programs will provide leaks

```
void vulnerable()
{
    char buf[32];
    gets(buf);
}

void main()
{
    setvbuf(stdin,NULL,_IONBF,0);
    setvbuf(stdout,NULL,_IONBF,0);
    printf("This is SIGPwny stack4,
    go\n");
    printf("You don't get the address of
    give_flag this time :(\n");
    vulnerable();
}
```

NX - Non-executable stack

- Every memory segment has 3 permission bits
  - Read - code is able to read from the memory
  - Write - code is able to write to the memory
  - Execute - data in memory can be executed as code
- NX removes the permission bit from the stack
- Why is this useful?

NX - Non-executable stack

- Every memory segment has 3 permission bits
  - Read - code is able to read from the memory
  - Write - code is able to write to the memory
  - Execute - data in memory can be executed as code
- NX removes the permission bit from the stack
- Why is this useful?
  - User data goes in the stack
  - User data could be interpreted as code
  - The executable stack could be used as part of a malicious user’s exploit

Bypassing NX

- Non-NX exploit: jump to shellcode on stack
- With NX, can’t execute shellcode on the stack

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
void vulnerable()
{
    char buf[32];
    printf("&buf = %p\n", &buf);
    printf("&printf = %p\n", &printf);
    gets(buf);
}

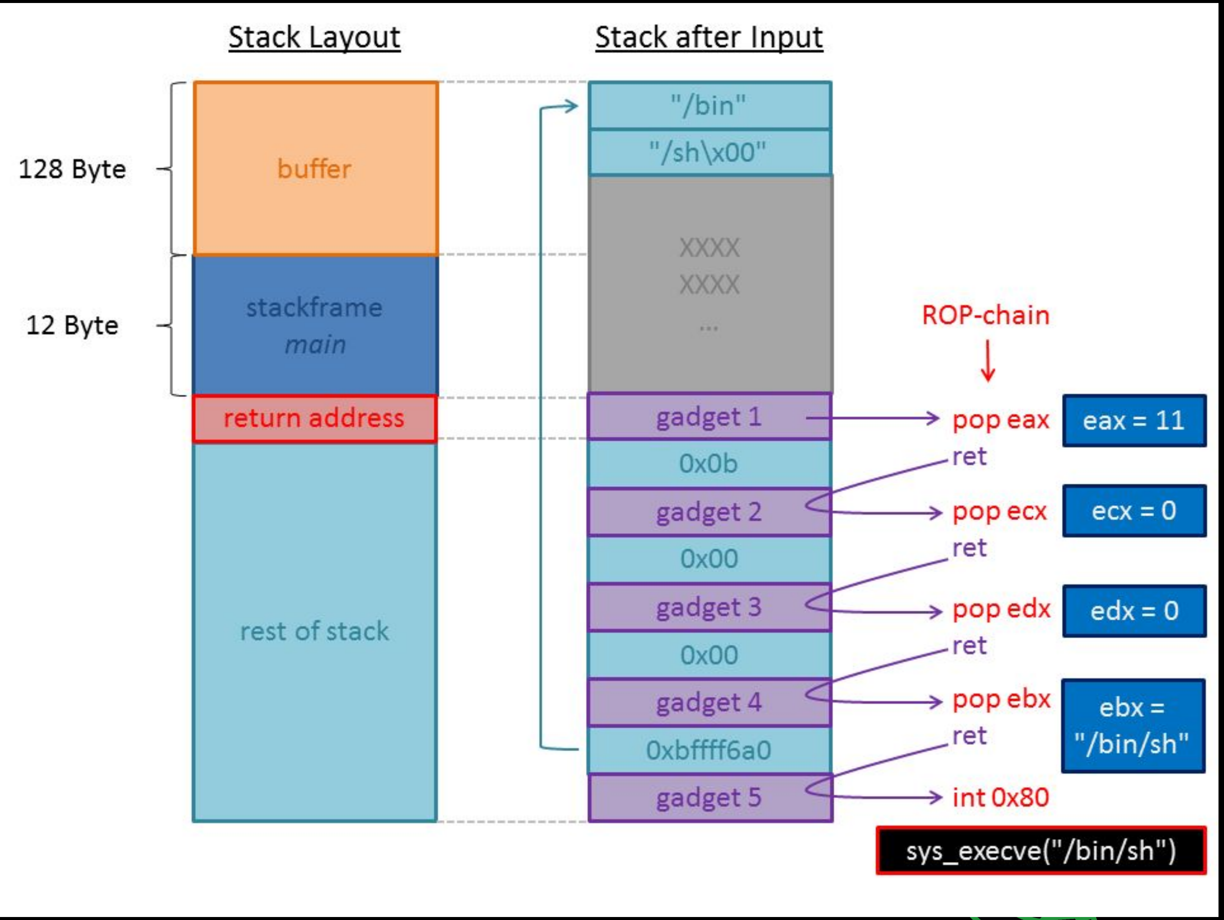
void main()
{
    setvbuf(stdin,NULL,_IONBF,0);
    setvbuf(stdout,NULL,_IONBF,0);
    printf("This is SIGPwny stack5, go\n");
    printf("We don't have a function to print
the flag anymore :(. But ASLR and NX are both
off. Use shellcode!\n");
    vulnerable();
}.
```

Bypassing NX: Shared libraries

- We need to find useful code to execute
  - But the program is tiny
- Where does the printf function come from?
  - Another program!
- Given a libc address (in this case, printf), one can calculate the address of any code in libc, then we can return to it
- Many useful pieces of code in libc, such as `system`
- “ret2libc”

Bypassing NX: ROP chains

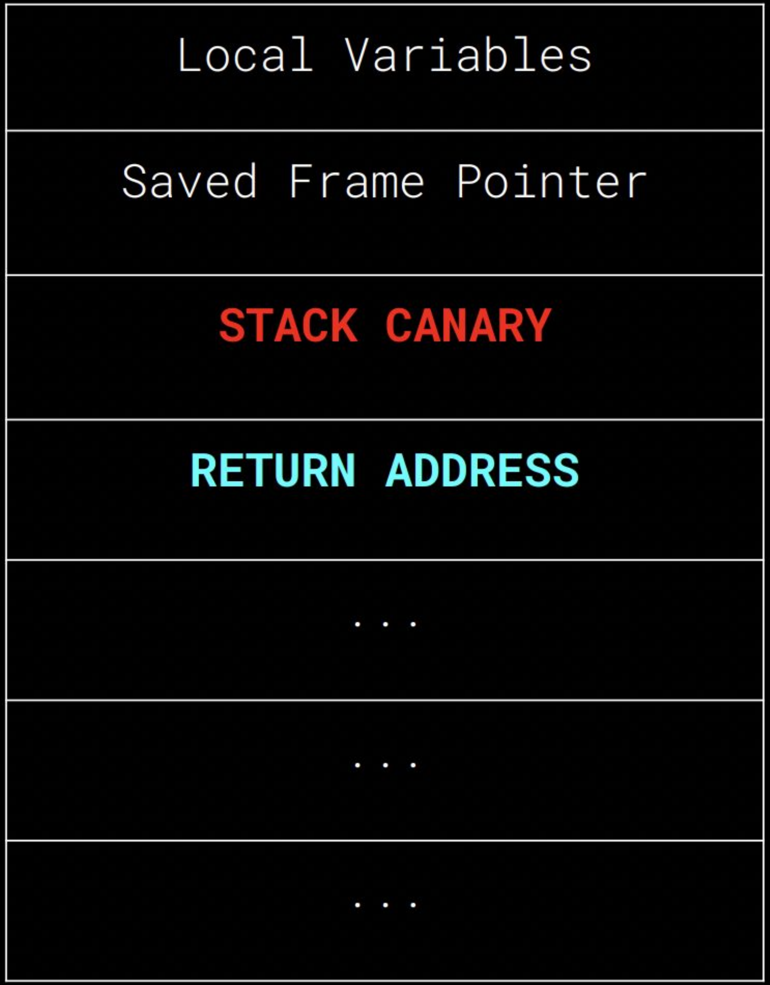
- "Return-oriented programming"
- Chain together pieces of code of the form
  - <instruction 1>
  - <instruction 2>
  - …
  - ret
- In large programs, you can perform arbitrary operations with a ROP chain



Stack Canary

Stack Canary

- Randomized value placed between frame pointer and return address on stack
- Overwriting a vulnerable buffer in a local variable requires also overwriting the CANARY before you can change the RETURN ADDRESS
- Randomized value is checked before the function returns to make sure it hasn't been changed
- Program immediately crashes if value has been changed



Stack Canary

- Almost entirely prevents stack buffer overflow exploitation
  - A couple cases where this isn't true
- Hackers now use other vulnerabilities
  - Heap-based vulnerabilities (use-after-free, heap buffer overflows)
    - Not covered in this meeting
  - Program-specific forms of memory corruption

Bypassing stack canaries

1. Arbitrary memory read and stack address leak
    a. Read canary from memory
    b. Include it in buffer overflow input
2. Forking program that has observable and recoverable crashes (i.e. nginx)
    a. Overflow the buffer through the first byte of the canary
    b. If the program crashes, the canary byte was wrong, so try again with a new guess
    c. Brute force the canary byte-by-byte
    d. Write-up: https://activities.tjhsst.edu/csc/writeups/justctf-2020-pinata

RELRO

- "REIocation Read-Only"
- Relocations:
  - Shared library addresses are resolved using the Procedure Linkage Table (PLT)
  - A call to printf actually calls printf in the Procedure Linkage Table (PLT):
    - ```
      call    8048410 <printf@plt>
      ```
  - PLT: table of functions that retrieve the addresses of shared library (e.g. libc) functions and store them in the Global Offset Table (GOT)
  - https://sigpwny.com/presentation-content/SP2021/global_offset_table.pdf for more information on GOT and PLT - Thomas
      ```

Overwriting the GOT

- PLT functions jump to addresses stored in the GOT
- If we overwrite a GOT address with our own address, we can change what shared library function calls do

```
08048410 <puts@plt>:
 8048410:       ff 25 94 99 04 08       jmp    DWORD PTR ds:0x8049994
 8048416:       68 10 00 00 00          push   0x10
 804841b:       e9 c0 ff ff ff          jmp    80483e0 <.plt>
```

Partial RELRO

- Changes memory order to make it harder to overwrite GOT addresses
- Without partial RELRO, global variables come before the GOT
  - A buffer overflow on a global variable would allow a GOT overwrite
- With partial RELRO, the GOT comes before global variables (.bss)

Full RELRO

- All shared library function addresses are resolved at program start-up, and the GOT has its write permission removed
- Downsides:
  - Program start-up can become slow for large programs
- Upsides:
  - No more GOT overwrites!
    - But probably plenty of other stuff you can overwrite, especially in real programs

Next Meetings

Next Meetings

Next Thursday: Physical Security

- How to secure your house
- Lockpicking and Safe Cracking!!!

Sunday Seminar: The Big Rick

- IOT Botnet Hacking
- An awesome story in Ethical Hacking By Minh!