Lab 1: Supplementary Information

CS 6264-OCY

Overview

- Basics of Binary Exploitation
  - Fuzzing
  - Finding overflow location
  - Leveraging overflow
- Binary Exploitation Defenses and Bypasses
- Further Reading

Basics of Binary Exploitation

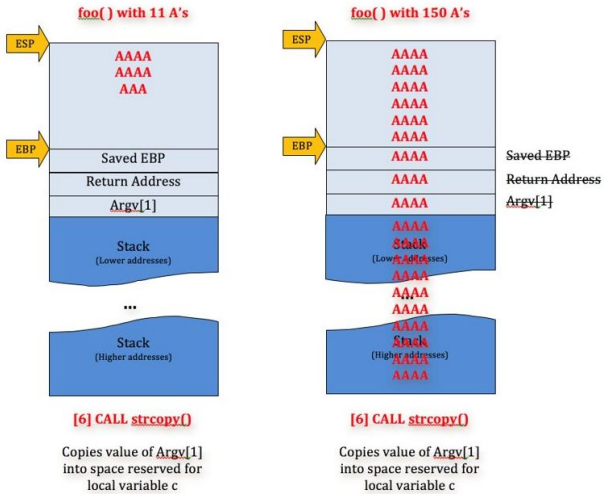
Fuzzing

- Sending unexpected or invalid inputs to program to see if it crashes
- If it does, an overflow in the program may have occurred
- For example:

```
> ./program1 aaaaaaaaaaaaaaaa (no crash)
> ./program1 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa (no crash)
> ./program1 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
(no crash)
> ./program1
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaa (crash! EIP value invalid at 0x61616161,
Overflow may exist)
```

Finding overflow location

- After an overflow is found, you will want to gain execution by controlling the value of the instruction pointer register (EIP)
- If an overflow exists in the stack, you are able to control the value of the EIP as the return address is popped off the stack upon a function return
- Location (offset) of return address can be found by having a unique string (i.e. Aa0Aa1Aa2Aa3…), then checking the value of EIP after the crash to find the location of that substring in the fuzzing string



Fuzzing (cont.)

- Both processes can be automated with Angr:
  - Initialize an Angr project that will execute the binary
  - Initialize a simulation manager based on the project
  - Create a stash in the manager to put dumps of runs in which the memory is corrupt (i.e. `sm.stashes['mem_corrupt'] = []`)
  - Let the manager step through the program until it reaches an unconstrained state (i.e. a crash)
  - Check if the program can let the PC register point to a specific value while in the unconstrained state
  - If so, you have a memory corruption! If you print the input that allowed the simulation manager to reach this state and count the number of bytes until your specific value, you now know the offset at which you overwrite the return address!

Example: Fuzzing with Angr

```
<SimulationManager with 1 active, 19 deadended, 13 deferred>
<SimulationManager with 1 active, 20 deadended, 12 deferred>
<SimulationManager with 1 active, 20 deadended, 13 deferred>
<SimulationManager with 1 active, 20 deadended, 14 deferred>
<SimulationManager with 1 active, 20 deadended, 15 deferred>
<SimulationManager with 1 active, 20 deadended, 16 deferred>
<SimulationManager with 1 active, 20 deadended, 17 deferred>
<SimulationManager with 1 active, 20 deadended, 18 deferred>
<SimulationManager with 1 active, 20 deadended, 18 deferred>
<SimulationManager with 1 active, 20 deadended, 18 deferred>
<SimulationManager with 1 active, 20 deadended, 18 deferred>
<SimulationManager with 1 active, 20 deadended, 18 deferred>
WARNING | 2021-01-11 18:48:09,544 | angr.engines.successors | Exit state has over 256 possible solutio
ns. Likely unconstrained; skipping. <BV32 packet_0_stdin_8_1024[743:736] .. packet_0_stdin_8_1024[751:
744] .. packet_0_stdin_8_1024[759:752] .. packet_0_stdin_8_1024[767:760]>
b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff'
```

Simulation manager has found an exploitable state and dumped the crashing input!

```
if path.satisfiable(extra_constraints=[path.regs.pc == b"CCCC"]):
    path.add_constraints(path.regs.pc == b"CCCC")
```

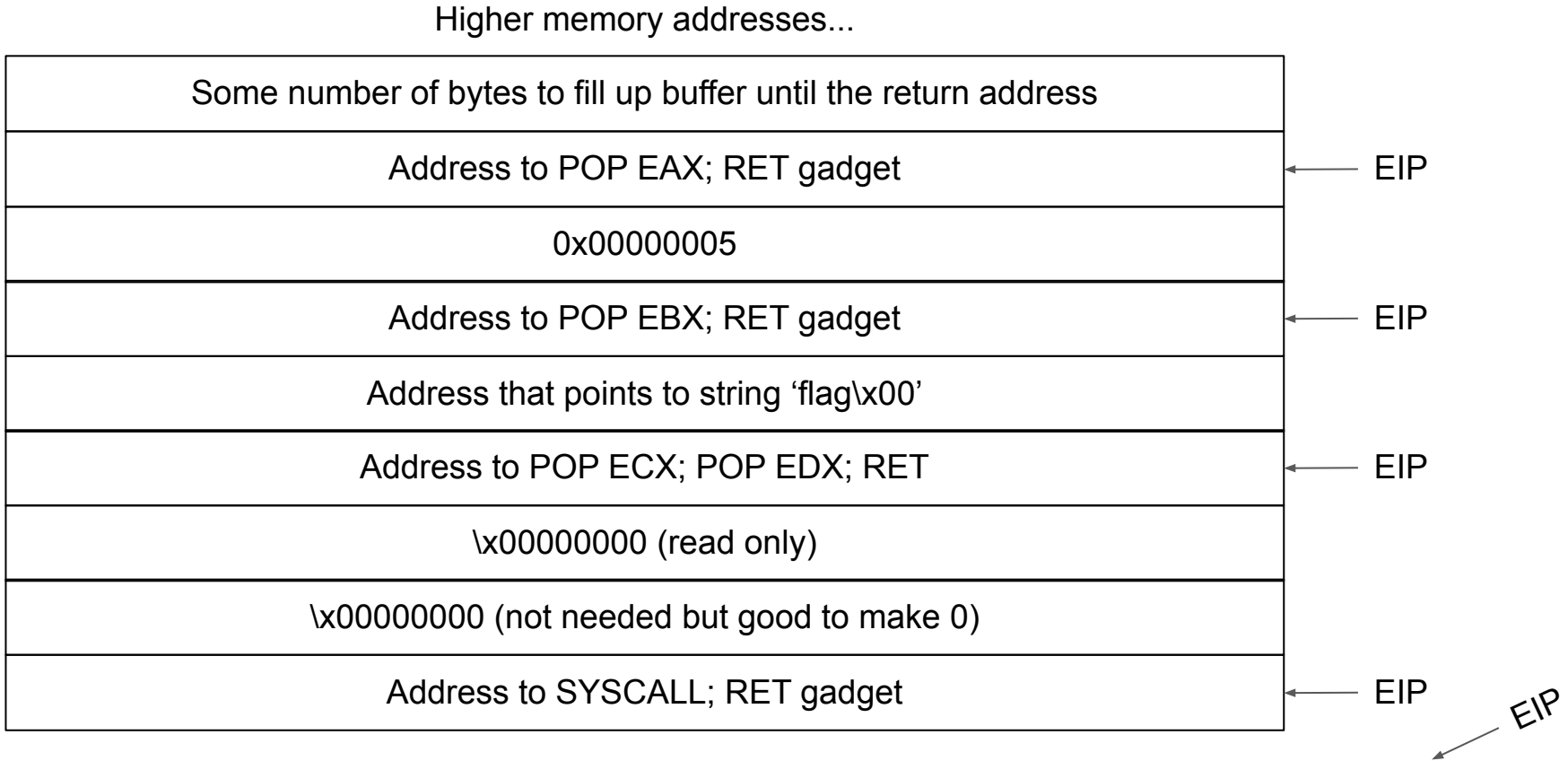
Leveraging Overflow

- 2 main methods for program execution
- Shellcode
  - Directly execute shellcode from the stack after injecting a POP-RET gadget into the return address
  - After program POP-RETs, the EIP will be pointing to the top of the stack where the rest of the shellcode resides and will execute the shellcode
  - Can be generated with shellcraft module in pwntool

Leveraging Overflow (cont.)

- Return-Oriented-Programming (ROP)
  - Usually used when you can't execute code (i.e. instructions) on the stack (i.e. DEP is enabled in the binary)
  - Instead of actual instructions being on the stack, addresses to those instructions are inserted on the stack instead (called gadgets, must end with a RET instruction to get back to stack)
  - For example, you can execute some syscalls
    - Normally, an OS doesn't allow the user/program to perform certain actions (i.e. opening a file, allocating memory, cleaning up a program after it exits, etc.)
    - If a user/program wishes to perform these actions, you will need to use a system call
    - Basically acts as an API to the underlying OS to execute those restricted actions

Example: Setting up an OPEN syscall to open ‘flag’



Binary Defenses and Bypasses

Binary Defenses and Bypasses

- Traditional Defenses - discussed in lectures
  - DEP - bypassed with ROP
  - ASLR - bypassed if you can leak an address like libc base or stack address
  - Canary - bypassed if canary is static and found by reversing binary
- Custom Defenses
  - Seccomp: limit the types of syscalls that you can make
    - For example, the program may only be allowed to make open(), read(), write(), and exit() system calls
    - Thus, if you wanted to make an exec() syscall, the program will exit with a SIGKILL or SIGSYS
  - Anti-debugging: prevent someone from attaching a debugger to the binary for reversing

Binary Defenses and Bypasses (cont.)

- How to bypass custom defenses?
  - Seccomp
    - Can’t really bypass, so you will have to work within the limitations set by the binary
    - However, you can check what syscalls are allowed with Seccomp-tools
  - Anti-debugging
    - Find where the check for anti-debugging software happens in disassembler
    - Patch out the check in the binary so that you can attach a debugger
    - For example, maybe it checks a specific string value that will always be true when a program is run in with a debugger (hint, hint)
    - Then, you can either edit the bytecode to have a different test (i.e. changing JNZ to JZ) or edit the string it compares to to a different string

Further Reading

Automated Fuzzing Resources

Uncover buffer overflow example (check #piecing-it-together for a full walkthrough)
https://breaking-bits.gitbook.io/breaking-bits/vulnerability-discovery/automated-exploit-development/buffer-overflows

angr solver example (while loop at line 67 is another example of finding the offset to overwrite the PC)
https://github.com/angr/angr-doc/blob/master/examples/insomnihack_aeg/solve.py

Seccomp Resources

(secure computing mode) linux kernel security facility (first paragraph is a good overview)
https://en.wikipedia.org/wiki/Seccomp

Seccomp-bpf (third paragraph of the Wikipedia article is a helpful overview)
an extension to seccomp that allows filtering of system calls

seccomp-tools: tools for seccomp analysis AKA check what syscalls are allowed (check #command-line-interface for instructions for how to run)
https://github.com/david942j/seccomp-tools

Binary Patching Resources

Patching with Ghidra at 3:35

https://www.youtube.com/watch?v=8U6JOQnOOkg

with vim, Binary Ninja, Ghidra and radare2

https://www.youtube.com/watch?v=LyNyf3UM9Yc