IERG4130

More than Buffer Overflow

By Ke ZHANG
27/9/2022

Outline

- Overview – software (application) security
- Walk through some common vulnerabilities/attacks
  - Learned in lecture
  - Extensions

Security

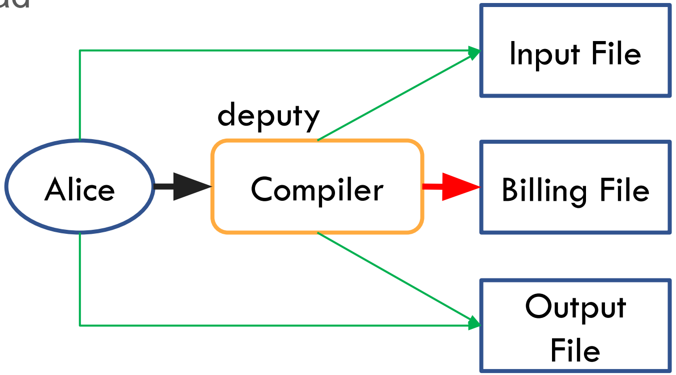
- CIA – Confidentiality, Integrity, Availability
- Threat Model (Adversary Model)
- Attack & Defense
- Software (application) security
  - Software - Input, code, environment
  - Remote attacks (e.g., network service, web browser) and local attacks (e.g., SUID application, OS)
  - About SUID program

SUID Program

- What is SUID program?
  - Process – real user ID(who runs it), effective user ID(used to check permission)
  - Usually, EID = RID
  - For SUID program, EID = owner of the program.
- Why SUID program?
  - High privilege to do something – e.g., change user passwd
  - The SUID program may authenticate you, and only do what is programmed
- Attack SUID program
  - Gain privilege to do “other things”

Software Vulnerability

- Design Vulnerability – logic/high-level flaws/problems
  - E.g., lack of authentication, wrong assumption of thread model, new attack interface…
  - How to analysis and defense? – recall security design principles
  - Confused Deputy Problem – “privilege escalation”
    - Alice doesn’t have permission to access Billing File, but Compiler does.
    - Another example: CSRF (later in Web Security)



Software Vulnerability

- Implementation Vulnerability
  - E.g., buffer overflow, format string vulnerability …
  - Unexpected event, e.g., malicious input…
  - System defense solution. E.g., ASLR…
  - Write secure software. E.g., Don’t trust user input – validate, sanitize; filtered output …
- Deployment Vulnerability
  - E.g., weak password, incorrect configuration, over-privileged user…

COMMON ATTACKS

File Access Attacks

- Path Attacks
  - `path = strncat("/var/log/app", user_input, free_size); file = open(path, O_RDWR);`
  - What if user_input = "../../../home/userA"; traversal directory attack `/var/log/app/../../../home/userA = /home/userA`
  - Defense: sanitize user input
- Another example: $PATH environment variable
  - used to search for commands
  - system("ls"), we could add "/home/attacker/" to $PATH and write a malicious "ls"
  - Environment variable based attacks (hidden input from untrusted users)
- TOCTTOU
  - Time-of-Check < Time-of-Attack < Time-of-Use
  - Race Condition (later slides)

Command Injection

- Invoke external commands
  - system(“xxx”) – internally, /bin/sh -c “xxx”
  - secure “version”: execve(1,2,3), directly call the command 1: command, 2: arguments of 1, 3: environment variables
  - Important: isolation of code and data
  - Another example: popen()
  - Exist in other languages (e.g., SQL)

```
cmd = “cat /var/log/” + user_input;
system(cmd);
```

```
user_input = “app; /bin/sh”
```

Then Get the root shell !
(if a SUID-root program use the above “code”)

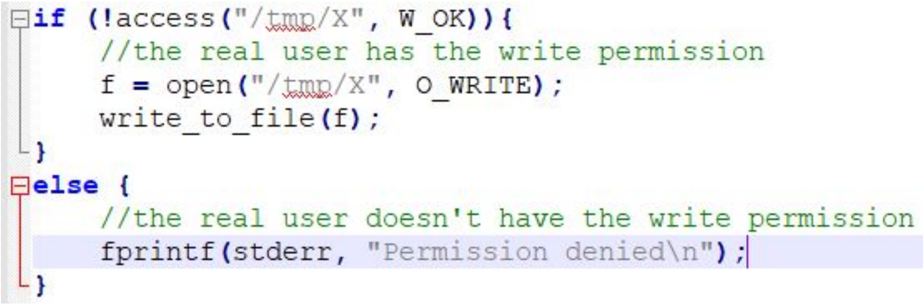
Command Injection

- Shellshock
    - Bash shell: function export to another shell instance
    - While the other shell parse the function, it execute the “tail” of the function

```
foo = ‘ () { echo “hello world”; }; echo “extra”; ’
export foo
bash_shellshock
```

Race Condition

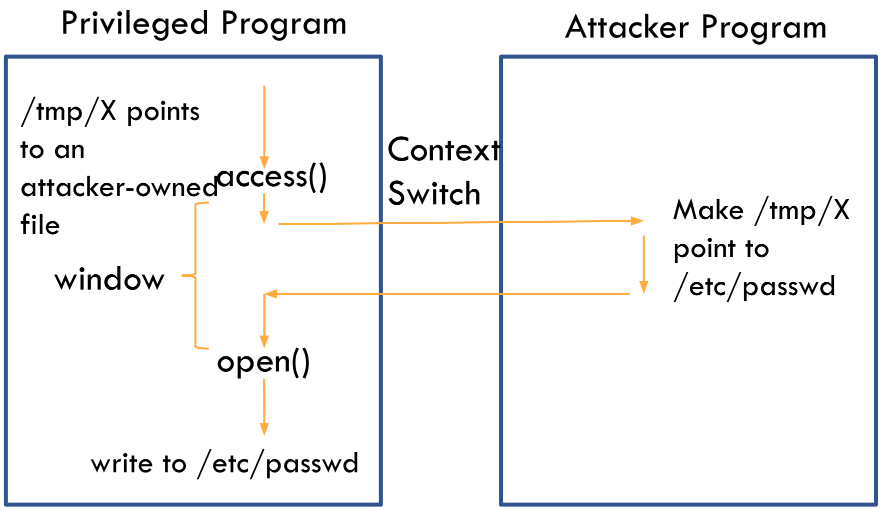
- Two concurrent threads access a shared resource, make the result dependent on the sequence or timing of the threads
    - Example: check money balance -> withdrawn money -> update balance
- TOCTTOU race condition example
    - SUID program to write a file
    - access() checks the real user permission
    - open() will check effective user and open
    - After access() checking, “quickly” point the /tmp/X to /etc/passwd, utilize the SUID privilege, write to file

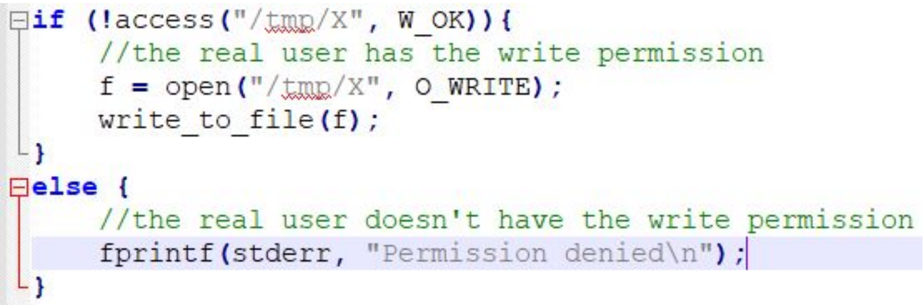


```
if (!access("/tmp/X", W_OK)){
    //the real user has the write permission
    f = open("/tmp/X", O_WRITE);
    write_to_file(f);
}
else {
    //the real user doesn't have the write permission
    fprintf(stderr, "Permission denied\n");
}
```

Race Condition

- How to point /tmp/X to /etc/passwd?
  - Impossible to modify the privileged program code, internal memory
  - Solution: symbolic link: ln –s
- How to win the condition?
  - Try enough times
  - Hit the window





```
if (!access("/tmp/X", W_OK)){
    //the real user has the write permission
    f = open("/tmp/X", O_WRITE);
    write_to_file(f);
}
else {
    //the real user doesn't have the write permission
    fprintf(stderr, "Permission denied\n");
}
```

Memory Corruption & Overflow Attacks

- Attack class – memory corruption
    - Attack Mean – Corrupt the memory of a process
    - Attack Goal – Take Control of a process: run with the privilege, execute attacker’s code
    - E.g., Stack, heap, format string…
- Overflow Attacks – why it can happen?
    - Mix the data and control code; allow users to overwrite the code/data
- Stack Overflow, Integer Overflow, Heap Overflow…
- …

Overflow Attacks

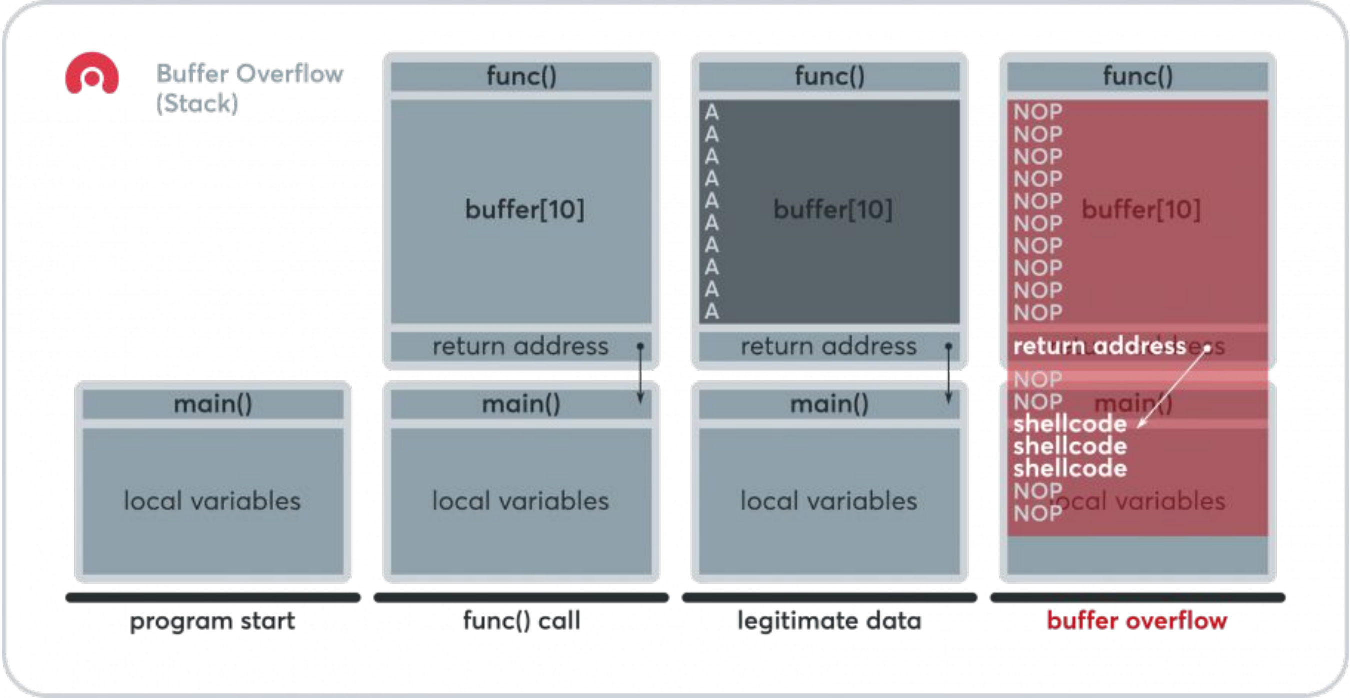
- What can be overwritten? (Data, RA, EBP, Function Pointer…)
- What causes the overwrite? (copying, array index, integer overflow, loop overflow…)
- Where is overwritten? (Stack, Heap, .data/.bss…)

Exploit Stack Overflow

- Goal: construct input payload resulting in control flow hijacking
  - overwrite return address with our code (shell code) address
    - need find offset(return address, buffer) -> GDB (or guess?) padding1
    - need know shellcode address
  - write shellcode into stack
  - Add padding – attention to strcpy(); “\x90” NOP padding2
  - Final payload: buffer[] = padding1 + addr of shellcode + padding2 + shellcode
  - in lecture note, shellcode is at beginning of buffer – both are OK

Exploit Stack Overflow

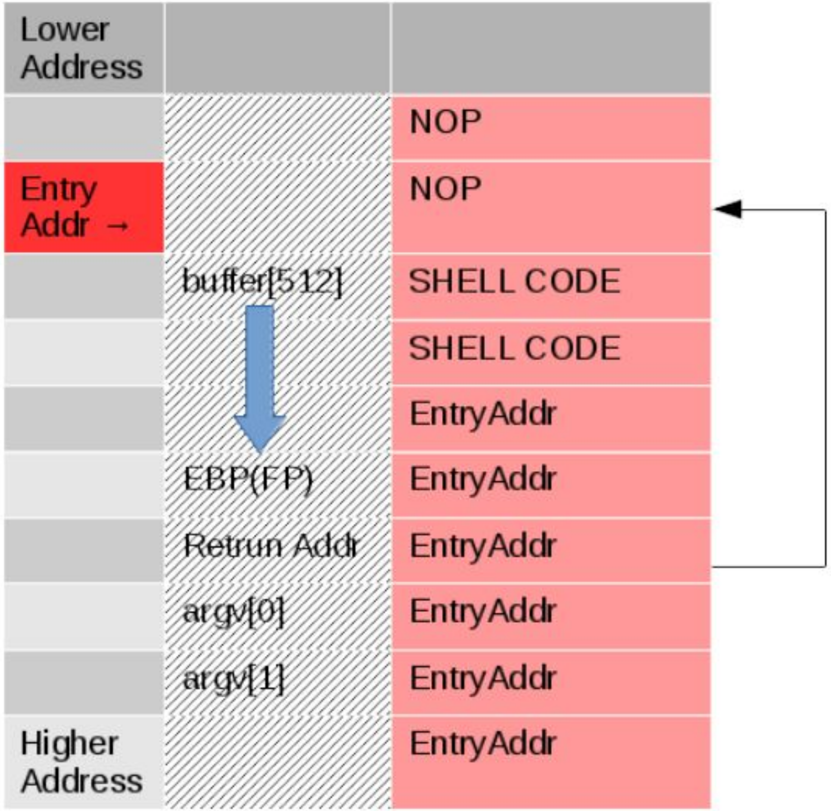
Final payload: buffer[] = padding1 + addr of shellcode + padding2 + shellcode



Exploit Stack Overflow

in lecture note, shellcode is at beginning of buffer – both are OK

Final payload: buffer[] = shellcode + + (padding2) + addr of buffer



Lower Address		
		NOP
Entry Addr →		NOP
	buffer[512]	SHELL CODE
		SHELL CODE
		EntryAddr
	EBP(FP)	EntryAddr
	Retrun Addr	EntryAddr
	argv[0]	EntryAddr
	argv[1]	EntryAddr
Higher Address		EntryAddr

Exploit Stack Overflow

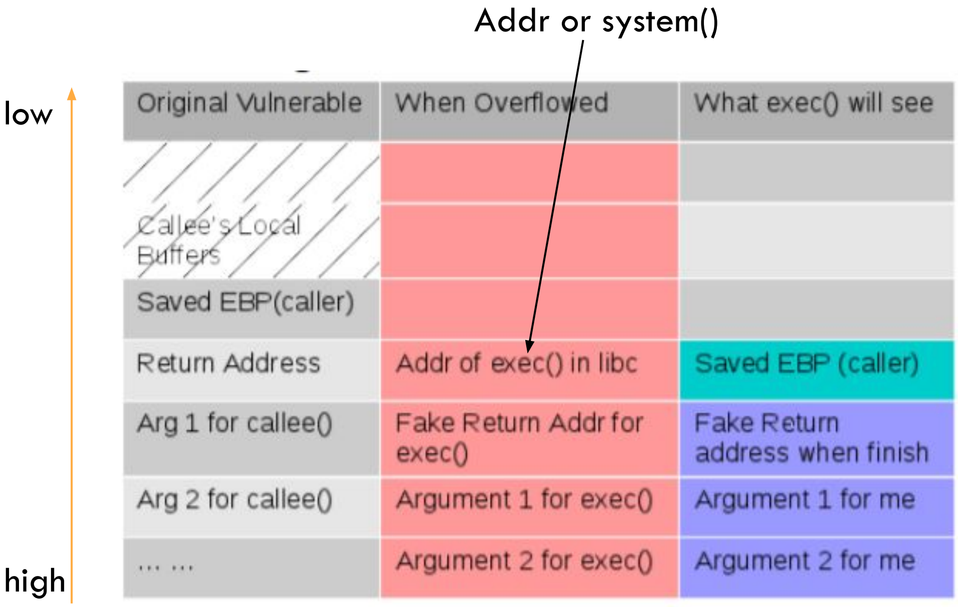
- python framework – pwntools
    - https://github.com/Gallopsled/pwntools
    - After you familiar with the attacks, try it
    - Rapid prototype, simplify exploit writing

Defense Solution - Detection or Prevention

- write secure, bug-free program. use memory-safe language.
- StackGuard – “canary”
- ASLR
- NX-bit
    - You can’t execute code in the stack any more. Code should be in .TEXT
    - But many “executable code” in the memory, can we utilize them?
    - How?

Return-to-libc attack

- Work flow
  - use figure from lecture note
  - libc - shared library in memory
  - Try to jump to system() in the libc
  - Then run /bin/sh
- find system() and “/bin/sh” address
- How to set the argument?

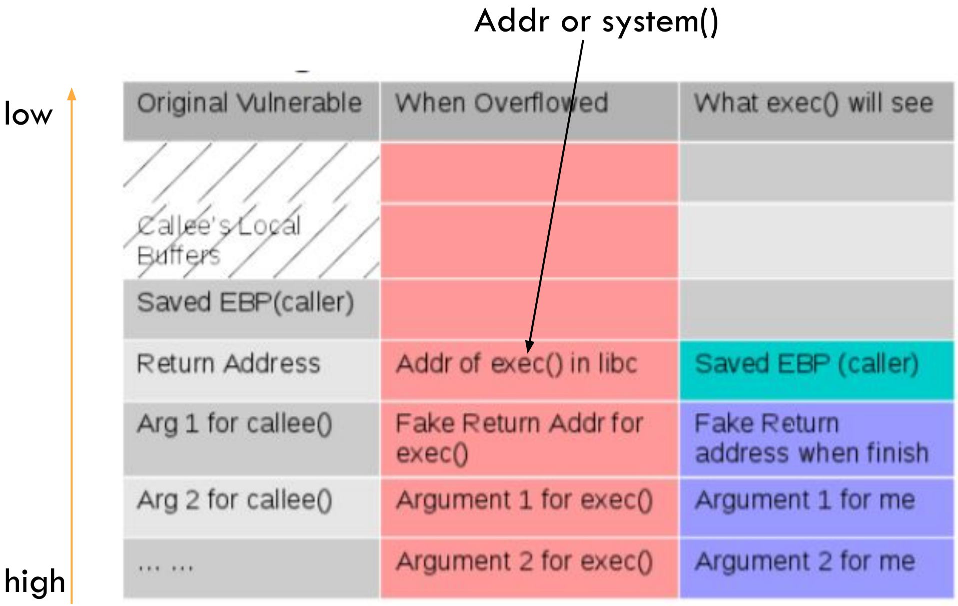

Addr or system()
low
high

Original Vulnerable	When Overflowed	What exec() will see
Callee's Local Buffers		
Saved EBP(caller)		
Return Address	Addr of exec() in libc	Saved EBP (caller)
Arg 1 for callee()	Fake Return Addr for exec()	Fake Return address when finish
Arg 2 for callee()	Argument 1 for exec()	Argument 1 for me
… …	Argument 2 for exec()	Argument 2 for me

Return-to-libc attack

- find system() and “/bin/sh” address
  - gdb – print addr of system()
  - write “/bin/sh” to stack; use environment variable; find “/bin/sh” directly in libc
- where to write the argument? -> %ebp + 8

  Final Payload = **padding1+addr of system() +padding2+addr of “/bin/sh”**
- what if no target function (or hard to find) or we want more flexible and “stronger” mean ? - ROP


Addr or system()
low
high

Original Vulnerable	When Overflowed	What exec() will see
Callee's Local Buffers		
Saved EBP(caller)		
Return Address	Addr of exec() in libc	Saved EBP (caller)
Arg 1 for callee()	Fake Return Addr for exec()	Fake Return address when finish
Arg 2 for callee()	Argument 1 for exec()	Argument 1 for me
... ...	Argument 2 for exec()	Argument 2 for me

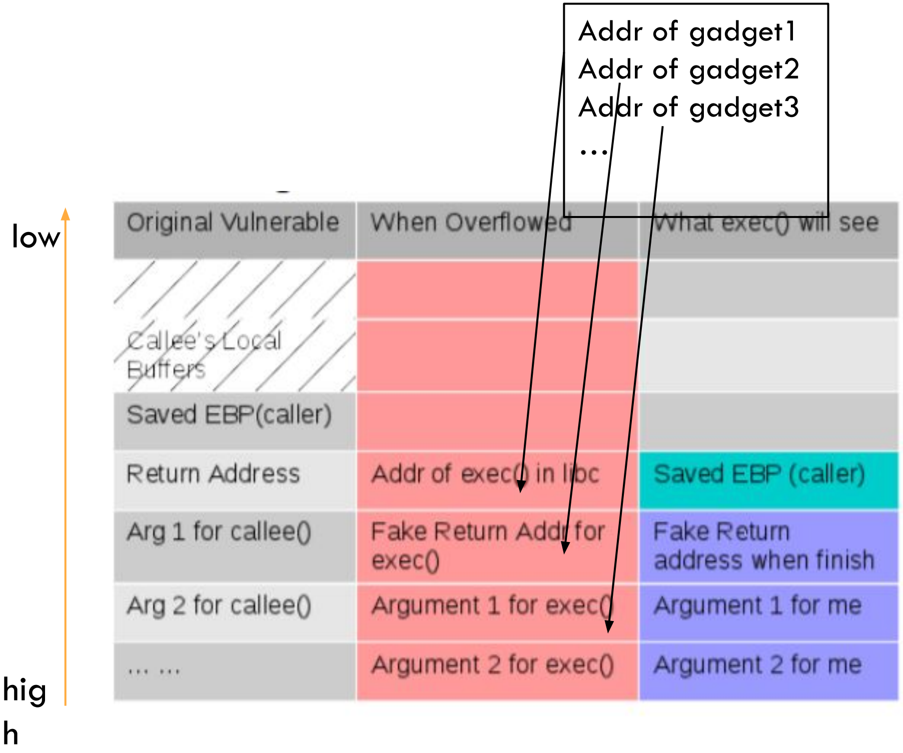
Think: can we chain more than one functions?

Return-Oriented Programming (ROP)

- Generalize return-to-libc
  - Don’t use function call
  - Execute a series of code snippet
- How
  - modify return address to point to an existing code snippet (“gadget”)
  - gadget end with “ret” to next gadget
- Example – invoking a system call

gadget

```
pop eax;
ret;
```

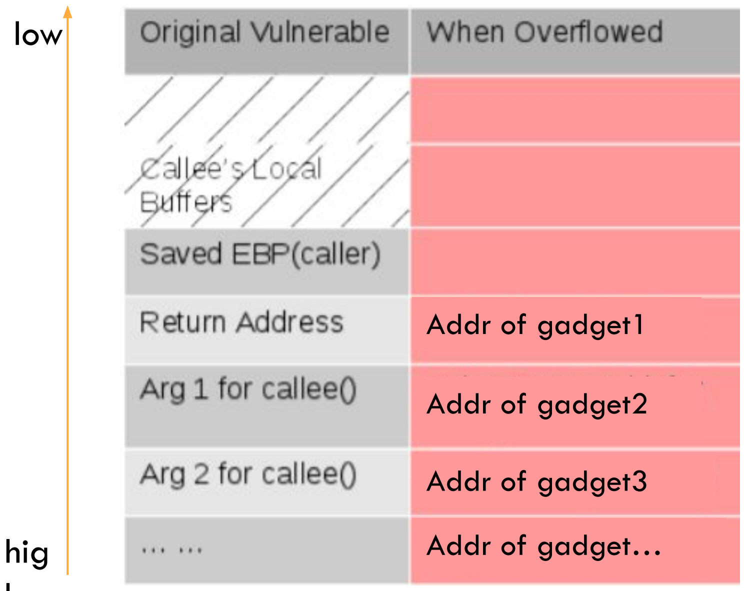


Addr of gadget1
Addr of gadget2
Addr of gadget3
…

Original Vulnerable	When Overflowed	What exec() will see
Callee's Local Buffers		
Saved EBP(caller)		
Return Address	Addr of exec() in libc	Saved EBP (caller)
Arg 1 for callee()	Fake Return Addr for exec()	Fake Return address when finish
Arg 2 for callee()	Argument 1 for exec()	Argument 1 for me
… …	Argument 2 for exec()	Argument 2 for me

Return-Oriented Programming (ROP)

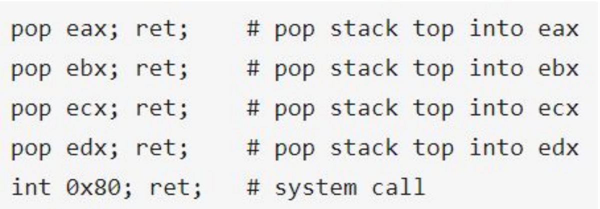
- Example – invoking a system call
- payload = **padding+addr of gadget1+addr of gadget2+…addr of gadget n**
- mprotect (void *addr, size_t len, int prot) used to modify stack to executable (No.125)
- recall system call procedure – eax, ebx, ecx, edx should be 125, addr, 0x10000, 7 (RWX)
- how to find gadget. E.g., tools like ROPgadget, rp++, even use grep to search “ret”
- how to set param, write to stack and use “pop gadget”
- Final payload = **padding+addr1+param1+addr2+param2…**



Original Vulnerable	When Overflowed
Callee's Local Buffers	
Saved EBP(caller)	
Return Address	Addr of gadget1
Arg 1 for callee()	Addr of gadget2
Arg 2 for callee()	Addr of gadget3
… …	Addr of gadget…

gadget

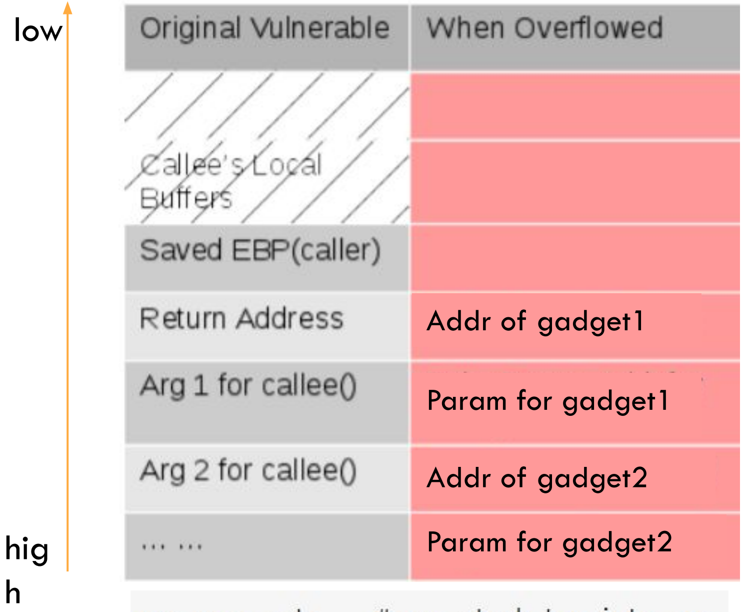
pop eax;
ret;



```
pop eax; ret;     # pop stack top into eax
pop ebx; ret;     # pop stack top into ebx
pop ecx; ret;     # pop stack top into ecx
pop edx; ret;     # pop stack top into edx
int 0x80; ret;    # system call
```

Return-Oriented Programming (ROP)

- Example – invoking a system call
- payload = **padding+addr of gadget1+addr of gadget2+…addr of gadget n**
- mprotect (void *addr, size_t len, int prot) used to modify stack to executable (No.125)
- recall system call procedure – eax, ebx, ecx, edx should be 125, addr, 0x10000, 7 (RWX)
- how to find gadget. E.g., tools like ROPgadget, rp++, even use grep to search "ret"
- how to set param, write to stack and use "pop gadget"
- Final payload = **padding+addr1+param1+addr2+param2…**



Original Vulnerable	When Overflowed
Callee's Local Buffers	
Saved EBP(caller)	
Return Address	Addr of gadget1
Arg 1 for callee()	Param for gadget1
Arg 2 for callee()	Addr of gadget2
… …	Param for gadget2

gadget

pop eax;
ret;

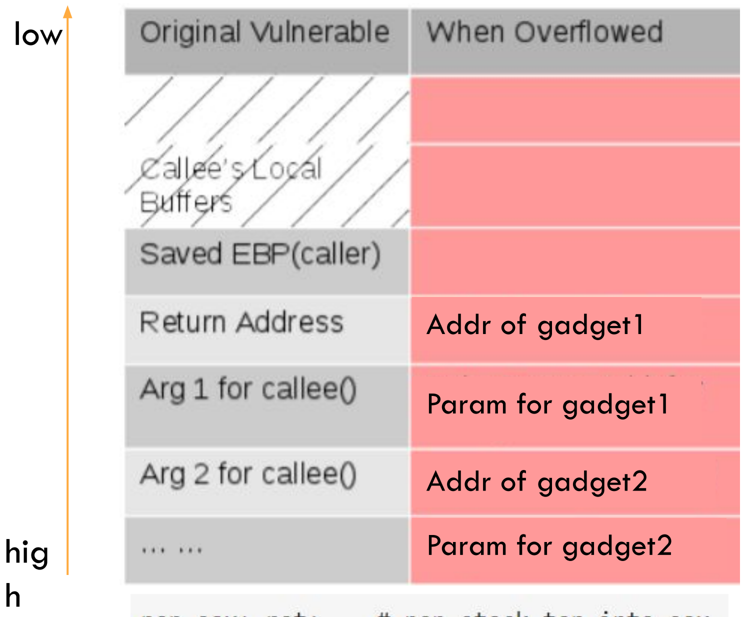
```
pop eax; ret;      # pop stack top into eax
pop ebx; ret;      # pop stack top into ebx
pop ecx; ret;      # pop stack top into ecx
pop edx; ret;      # pop stack top into edx
int 0x80; ret;     # system call
```

Return-Oriented Programming (ROP)

- Example – invoking a system call
- Final payload = **padding+addr1+param1+addr2+param2…**
- Attention:
- 125 = \x7d\x00\x00\x00 can be calculated

```
pop eax; ret;         # pop stack top 0x1111118e into eax
pop ebx; ret;         # pop stack top 0x11111111 into ebx
sub eax, ebx; ret;    # eax -= ebx
```

- `int` means interrupt, and the number 0x80 is the interrupt number. An interrupt transfers the program flow to whomever is handling that interrupt, which is interrupt `0x80` in this case. In Linux, `0x80` interrupt handler is the kernel, and is used to make system calls to the kernel by other programs.



Original Vulnerable	When Overflowed
Callee's Local Buffers	
Saved EBP(caller)	
Return Address	Addr of gadget1
Arg 1 for callee()	Param for gadget1
Arg 2 for callee()	Addr of gadget2
… …	Param for gadget2

gadget

pop eax;
ret;

```
pop eax; ret;      # pop stack top into eax
pop ebx; ret;      # pop stack top into ebx
pop ecx; ret;      # pop stack top into ecx
pop edx; ret;      # pop stack top into edx
int 0x80; ret;     # system call
```

Format String

- Read & Write memory data
- Try it in Lab 1

Heap Exploitation

- What is Heap?
- Beyond heap overflow
- Use after free
- Unlink
- House of XXX series
- …

IF YOU FEEL INTERESTED…

- More Labs on SEED project
- play CTF

Thank you!

Q&A