50.020 Security
Lecture 8: Operating System Security I

Introduction

Operating System Security

We focus on Malware (i.e., Malicious software) threats to Operating System.

Focus of This Lecture:

- General things about attacks: attack stages
- Operating system attack example: buffer overflow attacks
  - How buffer overflow attacks work

Focus of Next Lecture:

- Operating system attack example: buffer overflow attacks (continue)
  - Countermeasures of buffer overflow attacks
  - Variants of buffer overflow attacks
- More about malware

Attack Stages

Attack Stages

For any attack, the attacker needs to prepare

- Target victim identification
- Goal to achieve
- Payload that achieves the goal
- Attack vector for payload

Possible Attack Goals

A compromised machine can be used for:

- Malicious web server, distribute illegal content, phishing
- Email spam source
- Steal virtual and real money from accounts
- Launch social engineering attacks
- Bot in a botnet
- Blackmail (data encryption, publishing of private data)

Attack Strategies

The method the attacker uses to get payload to victim

- Most common:
  - Phishing, Spearphishing
  - Social engineering
  - Watering Hole Attack
- Less common (why? NAT'ing)
  - Direct/ active attacks (on hosts)

Email Security

- In general, emails are not secure
  - Content can be eavesdropped during transmission
  - Sender can claim any identity
  - Emails using HTML can track reads by user, contain malicious code
- Everyone knows spam, but emails can also be used for *phishing*

Phishing Attacks

- Emails that try to trick the user to visit a website
  - Pretends to be from legitimate source (e.g. Google)
  - Asks user to change the password
  - Link provided will lead to attacker website
  - Website will look like legitimate site, but steal credentials
- Often sent to random victims, easy to identify
  - User does not have account with target service
  - Typos or other unprofessional content
  - Link directs to something suspicious
- *Spearphishing* is a variant in which the attacker crafts a specific email for target, to maximise likelyhood of success
  - Attacker could pretend to be a person victim knows
  - Account information of user could be included
  - Email could be sent at time where user is busy

Spearphishing Example

Your Mailbox all@sutd.edu.sg, May Shutdown in 48-72 hours.

Your Mailbox all@sutd.edu.sg, May Shutdown in 48-72 hours.
Sat 17-Feb-18 7:31 AM

Account Require Verification

Dear all,

This is **sutd.edu.sg Office Administrator** managing your account server, We're writing to inform you that your Account will be permanently locked and shutdown! on our server due to your failure to verify and re-confirm your account **ownership**.

This a secure step to keep all account updated, secured and remove all malicious treat on our server. Kindly click on www.sutd.edu.sg/re-verify to securely get verified to continue using our server.

We are going to permanently remove your account (**all@sutd.edu.sg**),
if you fail to adheld to our instruction.

NOTE: ***This is a one time user verification carried out in purpose to provide a more secured platform and shut down robot or malicious users created in purpose of spamming and other fraudulent activities.***

Best Regards,
Administrator.

(C) 2018 sutd.edu.sg. All rights reserved. NMLSR ID 399801

Social Engineering

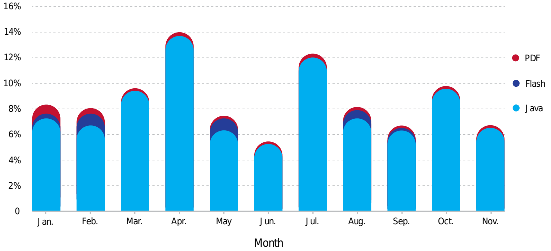
- Spearphishing is an example of a *social engineering* attack
- Any attack in which the attacker tries to trick the user in performing an action
  - Spoofed emails that tell customers to send payment to different account
  - Phone calls
  - Mail
- Usually pretending to be person of authority, or in need of help
- Inherent lack of authentication in real-world is a problem

Watering Hole Attacks

- General attack strategy: trick the user to download malware
  - Share malware in filesharing tools
  - Embed malware in websites under attacker control
- Watering Hole Attacks
  - Identify specific websites visited by victims, compromise them to inject malware

Execution of downloaded content

Malicious Attacks generated through PDF, Flash, Java, in 2013
Source: Cisco Cloud Web Security reports



- When browsing the web, lots of content is downloaded
- It is then evaluated or executed locally
  - Javascript, images, PDF, Flash, videos, ...
- Any of these is a potential attack vector!
- Modern browsers try to isolate the rendering engine from the OS

Malware

- Lets assume attacker identified target and strategy
- A payload exists that achieves the goal if executed on victim's machine
- How to deliver the payload?
  - We need an *attack vector* to deploy payload
- Attacker vector + payload = Malware

Operating System Attack Example: Buffer Overflow Attacks

Overview Buffer Overflow

- Is a major **attack vector** to inject code to highjack the target machine's execution path. The injected code here is a form of **payload**.
- Originally exploited data/code mix in von Neumann memory
  - Data provided by attacker could be executed in place
  - Control flow data (return address) next to user data
  - Some functions allow user to write more data than intended

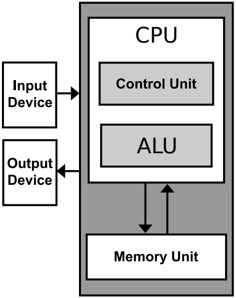
Overview Buffer Overflow: Continue

- Buffer overflow attacks exploit well-known vulnerabilities
  - In particular, related to C language
  - In particular, related to important insecure[1] functions in LibC
- The overflow overwrites data on the stack, which will influence control flow
  - Most importantly, the stored *return address*
  - Other variables can also be overwritten
  - Stack frame could also be manipulated

[1] https://wiki.sei.cmu.edu/confluence/display/c/MSC24-C.+Do+not+use+deprecated+or+obsolescent+functions

Refresher: Architectures

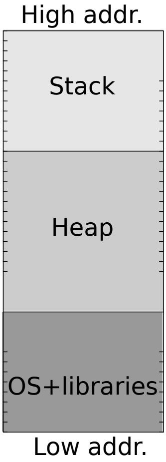
- Computers commonly use *von Neumann* architecture
  - Important for us: memory holds both *data* and *instructions*
- This fundamentally enables a series of attacks
  - Attackers are able to write data over legitimate instructions
  - Attacker's data is then executed as instructions
  - Attacker could also overwrite datastructures of running code
- Most prominent attack: *stack-based buffer overflow attacks* (our focus)
  - But also: heap overflow, . . .



Von Neumann Architecture

Memory Layout

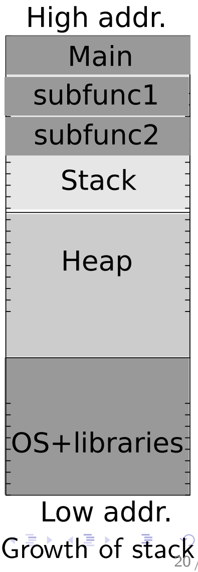
- Computers have different memory layers: registers, cache, RAM, discs
- Each process is presented with an abstract linear memory space to use
- OS takes care of translating memory access to the caches, RAM, disc
- Linear memory space is divided in sections
  - Stack is used for static allocation
  - Heap is used for dynamic allocation



Linear memory space

Memory use by processes

- Processes use stack to store data structures known at *compile-time*
  - Compiler knows size, can reserve appropriate memory area in advance
- The heap stores dynamic datastructures
  - Size or number of datastructures not known in advance
  - Memory space for that data has to be dynamically allocated (malloc)



Call Stack

- A **call stack** is a stack data structure that stores information about the active subroutines (i.e., **callees**) of a computer program (i.e., **caller**).
- A call stack can be used to keep track of the point to which each active subroutine should return control when it finishes executing.
- Watch the following videos for call stack introduction:
  - `https://www.youtube.com/watch?v=Q2sFmqvpBe0`
  - `https://www.youtube.com/watch?v=XbZQ-EonR_I` (More detailed)
- Each subfunction call will add new memory space on stack for that subfunction
  - As result, stack use is growing with deeper function call nesting
- If a subfunction returns, its memory space is *freed* (not erased)
  - Next subfunction call will overwrite

Calling conventions

- Stack space for each function is called *stack frame*[2]
- Stack frame header is prepared by calling function
- *Calling convention* defines the way that each stack frame header is organized
- This way, caller and callee can be sure that stack is in expected format
- Here, we will only discuss the System V AMD64 ABI calling convention for Linux/Mac

[2] `https://www.cs.rutgers.edu/~pxk/419/notes/frames.html`

64 bit stack layout (System V AMD64)

Registers:

- RBP: base pointer, which points to the base of the stack frame
- RSP: stack pointer, which points to the top of the stack frame
- RDI, RSI, RDX, RCX, R8, R9 used to provide arguments

To call a function:

- up to 6 arguments are passed in registers
- two more arguments/pointers can be passed on the stack
- Then, the return address is stored on stack
- Then, the calling RBP is stored on stack
- RBP is set to address of old RBP
- RSP is set to RBP-(space for variables)

Example Assembly (You will know more during Lab5)


```
[-----------------------------------registers-----------------------------------]
RAX: 0x7fffffffdaf0 --> 0x1
RBX: 0x0
RCX: 0x0
RDX: 0x7fffffffdc30 --> 0x7fffffffe076 ("LC_PAPER=en_SG.UTF-8")
RSI: 0x7fffffffdc18 --> 0x7fffffffe038 ("[redacted]/lab5/vulnapp")
RDI: 0x7fffffffdaf0 --> 0x1
RBP: 0x7fffffffdb30 --> 0x400690 (<__libc_csu_init>:    push   r15)
RSP: 0x7fffffffdaf0 --> 0x1
RIP: 0x40064d (<main+15>:       call   0x4005b6 <getlines>)
[-------------------------------------code-------------------------------------]
   0x400642 <main+4>:   sub    rsp,0x40
   0x400646 <main+8>:   lea    rax,[rbp-0x40]
   0x40064a <main+12>:  mov    rdi,rax
=> 0x40064d <main+15>:  call   0x4005b6 <getlines>
```

Example minimal setup of stack for subfunction `getlines()`, and call. Call will push RBP and RIP onto stack automatically.

```
Dump of assembler code for function getlines:
   0x00000000004005b6 <+0>:     push   rbp
   0x00000000004005b7 <+1>:     mov    rbp,rsp
   0x00000000004005ba <+4>:     sub    rsp,0x20
                                ...
   0x00000000004005eb <+53>:    leave
   0x00000000004005ec <+54>:    ret
End of assembler dump.
```

Subfunction `getlines()` prepares its stack frame, and `leave` + `ret` at end.

Example C code (8 arguments)

Example (Function call (C/AMD64/Linux))

```
long myFunc(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
    char myBuffer[24];
    gets(myBuffer);
    return a+b+c+d+e+f+g+h;
}
```

- Important points about this function
  - 24*8(=3*64) bit variables
  - 8*64 bit arguments

Example stack layout

AMD64 calling convention

High addr.	calling frame
RBP +24	h
RBP +16	g
RBP +8	stored RIP
RBP	saved RBP
RBP -8	myBuffer[16]
RBP -16	myBuffer[8]
RSP	myBuffer[0]
Low addr.	

Registers

RDI	a
RSI	b
RDX	c
RCX	d
R8	e
R9	f

Note: Stack "grows downwards"

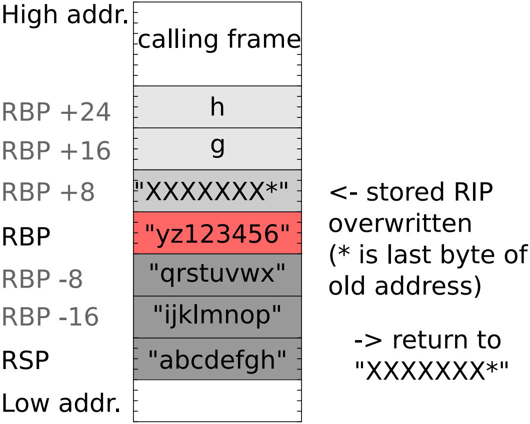
Simple Buffer Overflow

- Assume `myBuffer` contains `abcdefghijklmnopqrstuvwxyz\0`
- `gets()` continues to read in until \0 char

High addr.	calling frame	
RBP +24	h	
RBP +16	g	
RBP +8	stored RIP	
RBP	"yz\0"	<- stored RBP overwritten
RBP -8	"qrstuvwx"	
RBP -16	"ijklmnop"	
RSP	"abcdefgh"	-> continuation of calling function may fail....
Low addr.		

Malicious Buffer Overflow

- Assume `myBuffer` contains `abcdefghijklmnopqrstuvwxyz123456XXXXXXX\0`
- return addr. is overwritten with `XXXXXXX`



Code injection attacks

- The return address would point to the address of some malicious code designed by Attacker
- If stack is executable:
  - Easy way to execute arbitrary code ("shellcode")
  - For example, shellcode can spawn reverse shell to attacker

Next Lecture

- Continue with buffer overflow attacks:
  - Countermeasures of buffer overflow attacks
  - Variants of buffer overflow attacks
- More about Malware