Chapter 4. Buffer Overflow

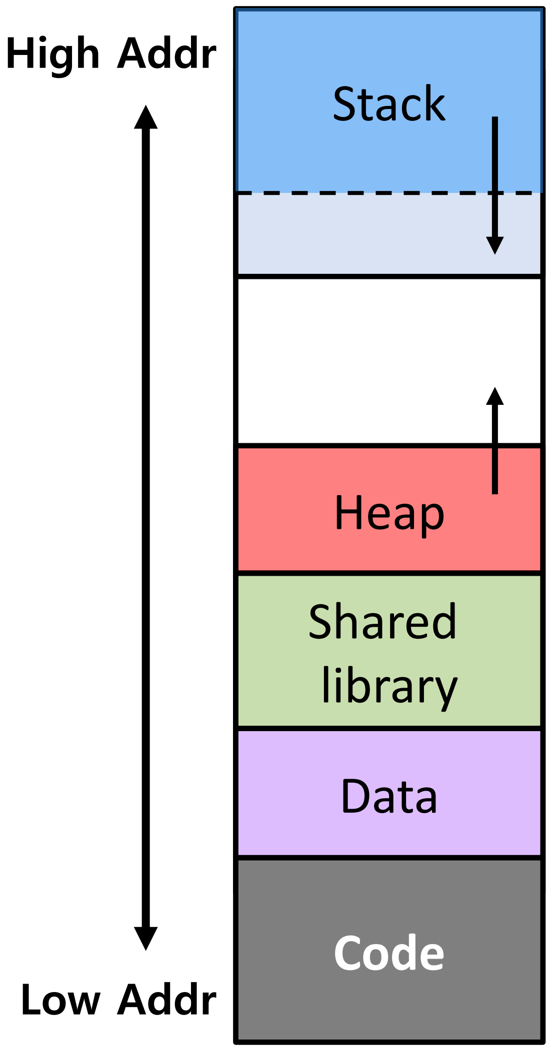
Prof. Jaeseung Choi
Dept. of Computer Science and Engineering
Sogang University

Topics

- **Memory layout of a program**
- **Basic concept of buffer overflow**
  - Stack memory corruption and control hijack
  - Exploitation with shellcode
- **The first round of war between attacker vs. defender**
  - (Mitigation) Stack canary, NX
  - (Bypassing) Memory disclosure

Memory Layout

- **Stack**
  - Stack frames of executing functions
- **Heap**
  - Memory blocks dynamically allocated by using `malloc()` or `new()`
- **Shared library**
  - Functions that you didn't write directly
- **Data**
  - Global variables of your program
- **Code (a.k.a. Text)**
  - Instructions of the functions that you wrote

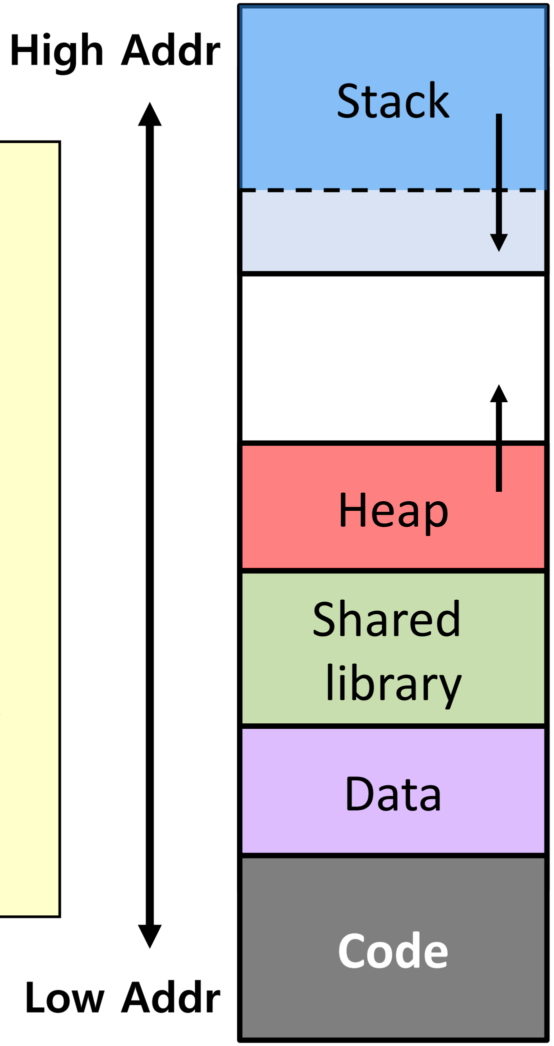


Memory Layout Example

```
int i_arr [65536];
char *str = "Hello world";
int count = 0;

int f() { return 0; }

int main() {
  void *p;
  int i = 0;
  p = malloc(256); // p points to a memory block
  printf(str);
  return 0;
}
```



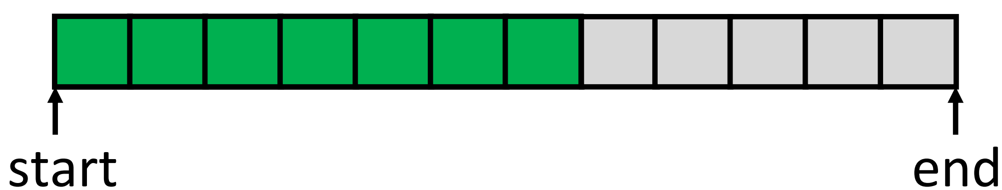
Topics

- Memory layout of a program
- Basic concept of buffer overflow
  - Stack memory corruption and control hijack
  - Exploitation with shellcode
- The first round of war between attacker vs. defender
  - (Mitigation) Stack canary, NX
  - (Bypassing) Memory disclosure

Buffer Overflow (BOF)

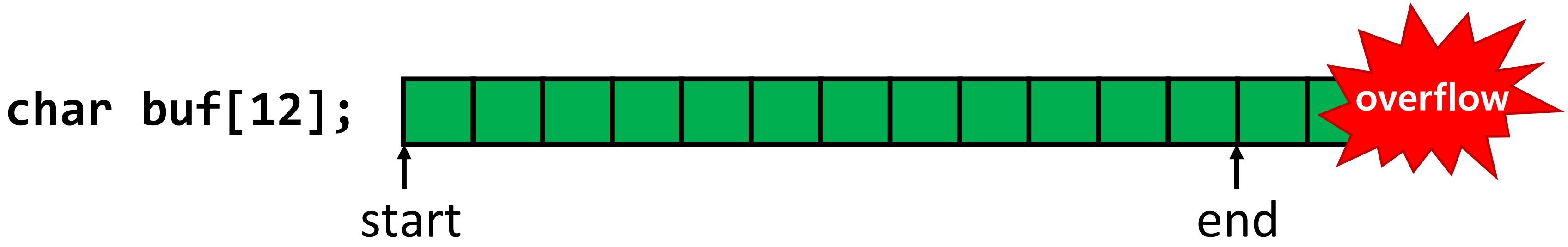
- **C has no automatic check on array index and boundary**
  - Also, some functions (like **`gets`**) don’t check the input length
  - This allows to write **past the end of an array** (buffer): overflow!
  - Such write can **corrupt** other data in the memory

`char buf[12];`



Buffer Overflow (BOF)

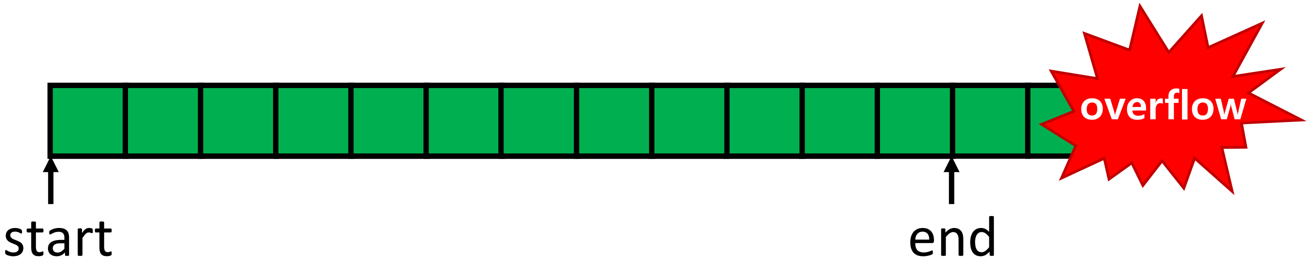
- **C has no automatic check on array index and boundary**
  - Also, some functions (like **`gets`**) don't check the input length
  - This allows to write **past the end of an array** (buffer): overflow!
  - Such write can **corrupt** other data in the memory



Buffer Overflow (BOF)

- **C has no automatic check on array index and boundary**
  - Also, some functions (like **`gets`**) don’t check the input length
  - This allows to write **past the end of an array** (buffer): overflow!
  - Such write can **corrupt** other data in the memory
- **What kind of critical data can be corrupted?**
  - ***Return address*** saved in the stack frame is a good example
  - Corruption of saved return address allows an attacker to manipulate the program counter (a.k.a. ***control hijack***)

`char buf[12];`



Classic Buffer Overflow

- **Overflow of a buffer in the stack memory**
  - Called *stack-based buffer overflow,* sometimes *stack smashing**
  - Not to be confused with "stack overflow"
- **Often caused by using unsafe string-handling functions**
  - **`gets()`, `scanf("%s", ...)`, `strcpy()`, `strcat()`**...
- **Became famous because hackers could easily exploit them and infect machines**
  - Ex) *Morris Worm* (the first internet worm) in 1988 also exploited stack-based buffer overflow vulnerability in **`fingerd`** server

* "Smashing the stack for fun and profit", http://phrack.org/archives/issues/49/14.txt

General Discussion on BOF

- **Buffer overflow does not always occur in stack**
  - For example, it can also occur in heap-allocated memory
- **Unsafe function is not the only cause of buffer overflow**
  - Arrays can be misused in many other ways
  - The following code is also a popular pattern of buffer overflow
    - Some prefer to call this ***out-of-bound access*** (more general)

```c
int main(void) {
    int arr[32];
    int idx;
    scanf("%d", &idx);
    arr[idx] = 1; // Error
    return 0;
}
```

Example Program with BOF

```
void echo(void) {
    char buf[8];
    gets(buf);
    puts(buf);
}

int main(void) {
    echo();
    return 0;
}
```

```
jschoi@ubuntu:~$ ./bof
Hello
Hello
```

```
jschoi@ubuntu:~$ ./bof
0123456789ABCDE
0123456789ABCDE
```

```
jschoi@ubuntu:~$ ./bof
0123456789ABCDEF
0123456789ABCDEF
Segmentation fault
```

Starts to crash from this point (we will see why)

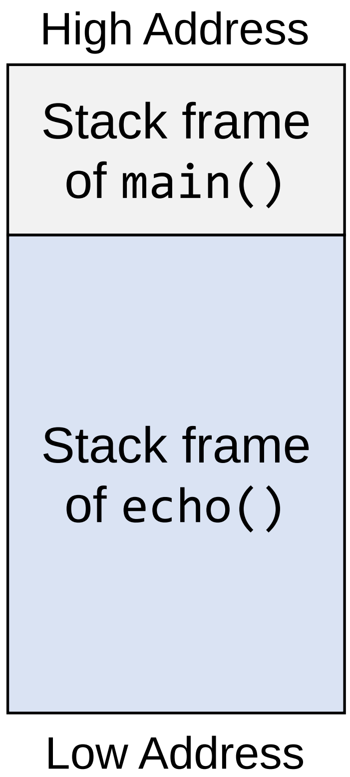
Assembly Code of the Example

```
void echo(void) {
    char buf[8];
    gets(buf);
    puts(buf);
}


int main(void) {
    echo();
    return 0;
}
```

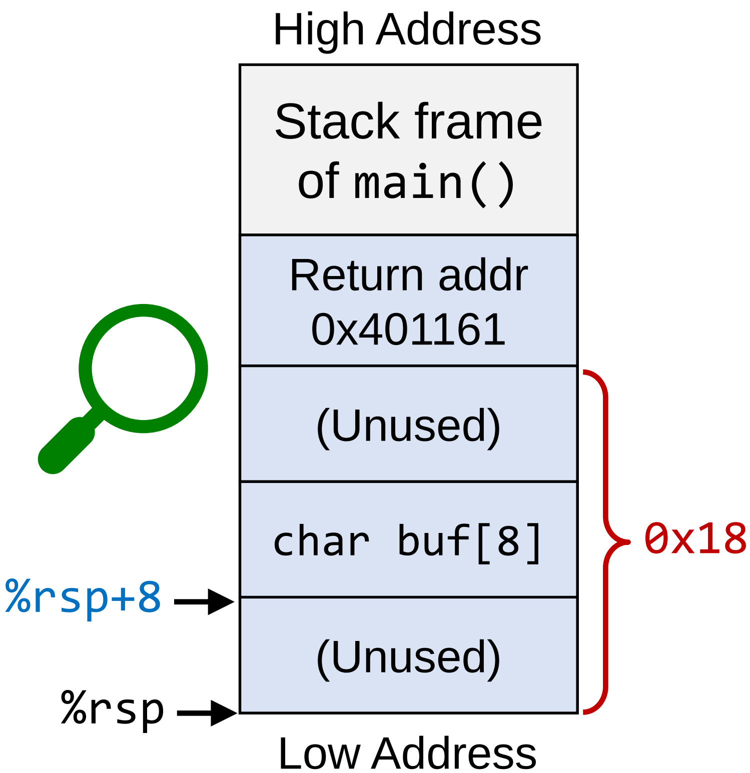
```
(gdb) disassemble echo
0x401136:  sub    $0x18,%rsp
0x40113a:  lea    0x8(%rsp),%rdi
0x40113f:  mov    $0x0,%eax
0x401144:  call   0x401040 <gets@plt>
0x401149:  lea    0x8(%rsp),%rdi
0x40114e:  call   0x401030 <puts@plt>
0x401153:  add    $0x18,%rsp
0x401157:  ret
(gdb) disassemble main
0x401158:  sub    $0x8,%rsp
0x40115c:  call   0x401136 <echo>
0x401161:  mov    $0x0,%eax
0x401166:  add    $0x8,%rsp
0x40116a:  ret
```

Stack Frame Layout




```
(gdb) disassemble echo
0x401136:  sub    $0x18,%rsp
0x40113a:  lea    0x8(%rsp),%rdi
0x40113f:  mov    $0x0,%eax
0x401144:  call   0x401040 <gets@plt>
0x401149:  lea    0x8(%rsp),%rdi
0x40114e:  call   0x401030 <puts@plt>
0x401153:  add    $0x18,%rsp
0x401157:  ret
(gdb) disassemble main
0x401158:  sub    $0x8,%rsp
0x40115c:  call   0x401136 <echo>
0x401161:  mov    $0x0,%eax
0x401166:  add    $0x8,%rsp
0x40116a:  ret
```

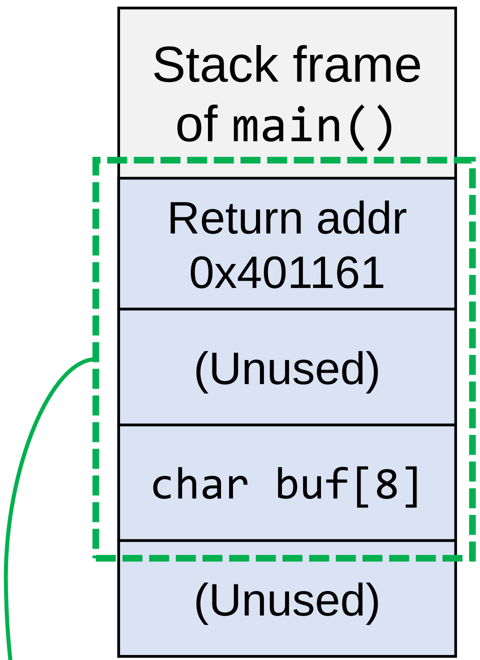
Stack Frame Layout



Let's take a closer look on the stack frame

```
(gdb) disassemble echo
0x401136:   sub    $0x18,%rsp
0x40113a:   lea    0x8(%rsp),%rdi
0x40113f:   mov    $0x0,%eax
0x401144:   call   0x401040 <gets@plt>
0x401149:   lea    0x8(%rsp),%rdi
0x40114e:   call   0x401030 <puts@plt>
0x401153:   add    $0x18,%rsp
0x401157:   ret
(gdb) disassemble main
0x401158:   sub    $0x8,%rsp
0x40115c:   call   0x401136 <echo>
0x401161:   mov    $0x0,%eax
0x401166:   add    $0x8,%rsp
0x40116a:   ret
```

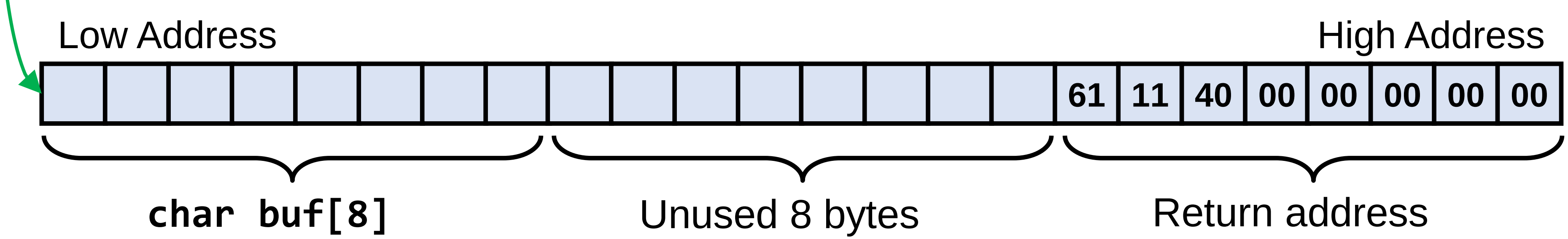
What happens in the stack frame?



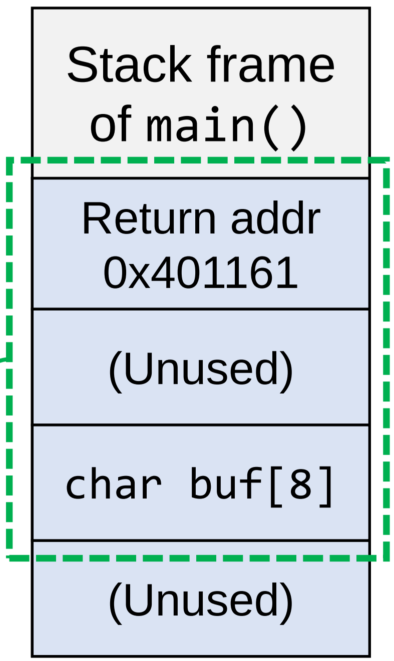

```
void echo(void) {
    char buf[8];
    gets(buf);
    ...
```

```
sub     $0x18,%rsp
lea     0x8(%rsp),%rdi
...
call    0x401040 <gets@plt>
```

- **When `gets(buf)` is called...**
  - Each character in the input string will be stored into char buf[8], ***starting from lower address***



Example Input #1




```
void echo(void) {
     char buf[8];
     gets(buf);
     ...
```

```
sub     $0x18,%rsp
lea     0x8(%rsp),%rdi
...
call    0x401040 <gets@plt>
```

```
jschoi@ubuntu:~$ ./bof
Hello
Hello
```

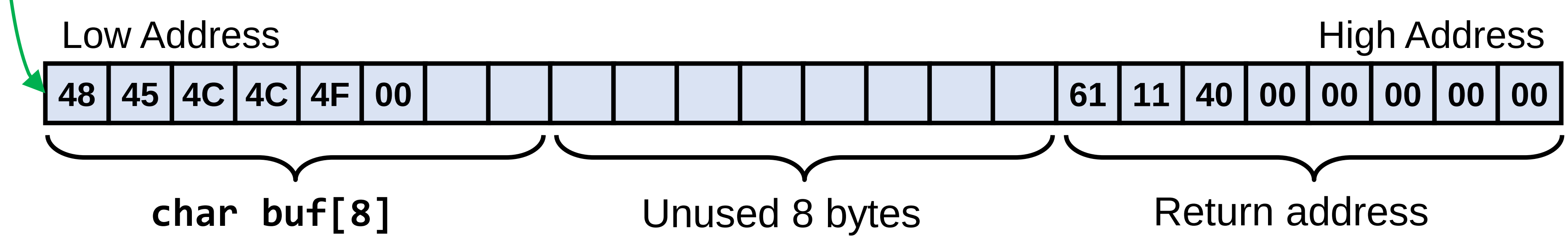
(ASCII Encoding)

'H': 0x48

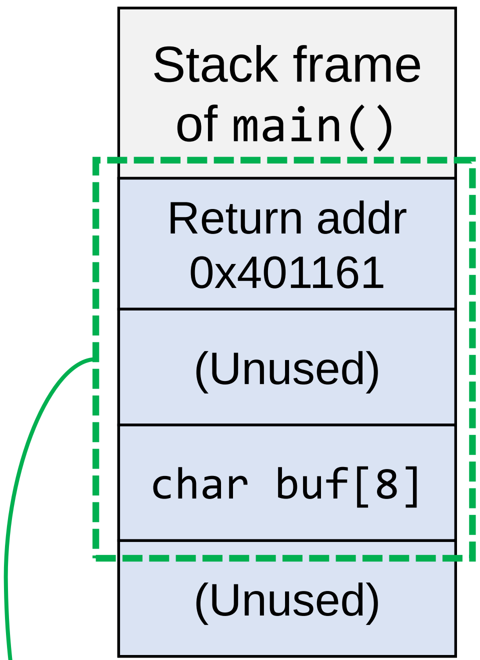
'E': 0x45

'L': 0x4C

'O': 0x4F



Example Input #2




```
void echo(void) {
     char buf[8];
     gets(buf);
     ...
```

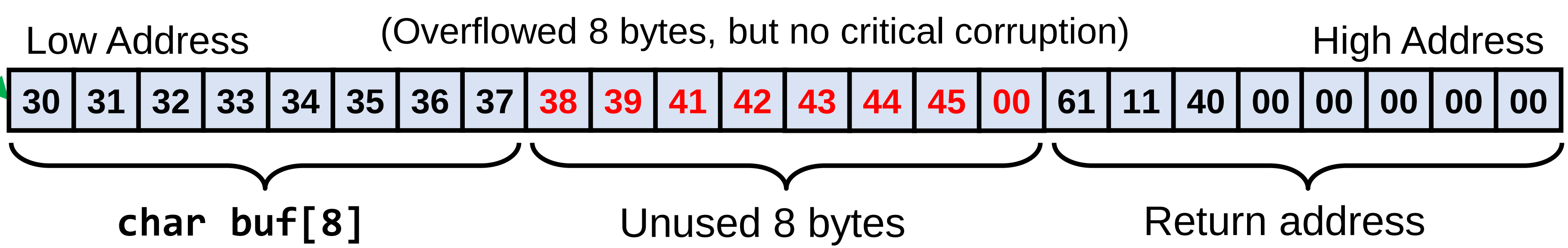
```
sub     $0x18,%rsp
lea     0x8(%rsp),%rdi
...
call    0x401040 <gets@plt>
```

```
jschoi@ubuntu:~$ ./bof
0123456789ABCDE
0123456789ABCDE
```

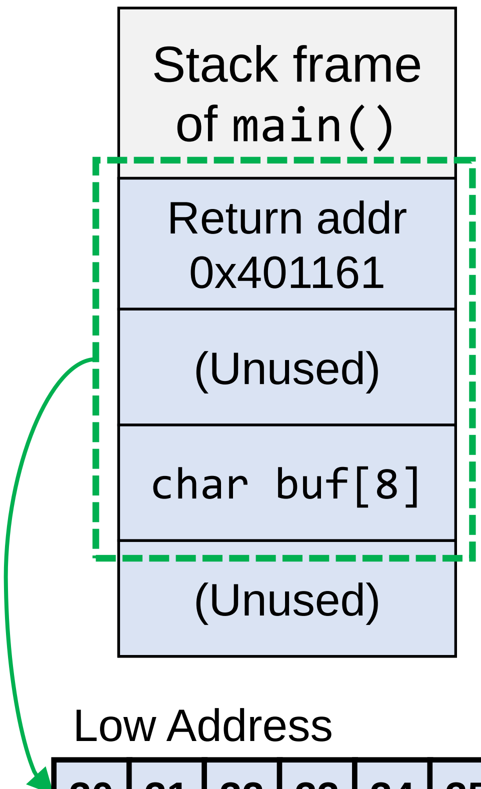
(ASCII Encoding)

`'0': 0x30`

`'A': 0x41`



Example Input #3




```
void echo(void) {
     char buf[8];
     gets(buf);
     ...
```

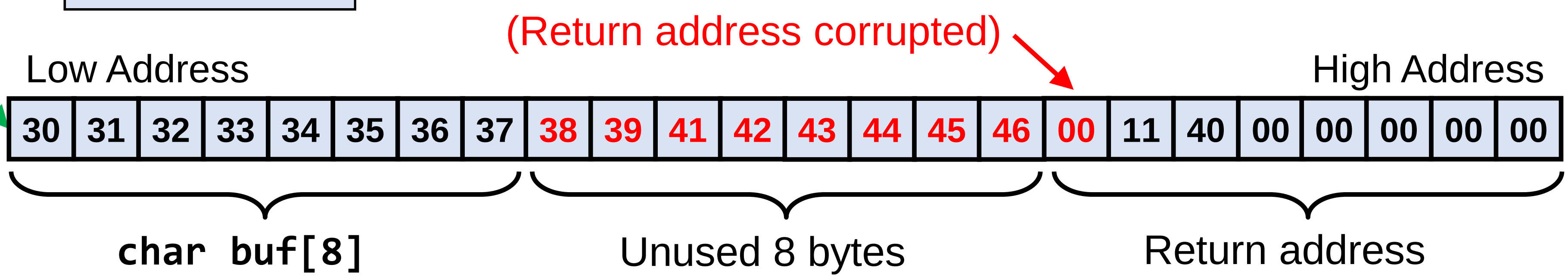
```
sub     $0x18,%rsp
lea     0x8(%rsp),%rdi
...
call    0x401040 <gets@plt>
```

```
jschoi@ubuntu:~$ ./bof
0123456789ABCDEF
0123456789ABCDEF
Segmentation fault
```

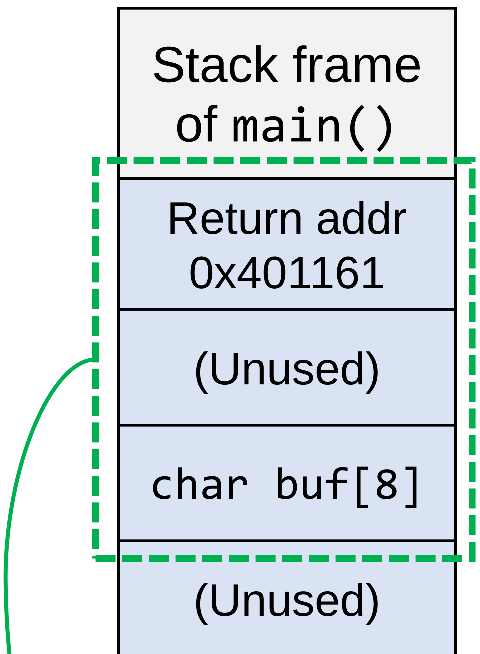
(ASCII Encoding)

'0': 0x30

'A': 0x41



Example Input #3: Crash

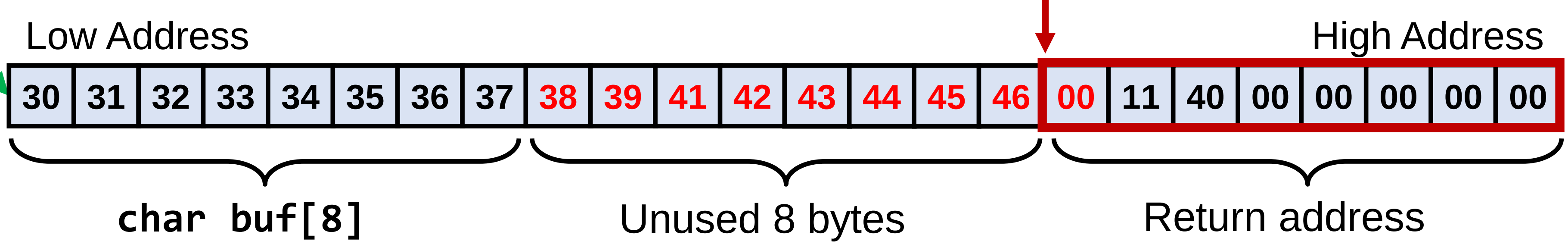



```
void echo(void) {
     char buf[8];
     gets(buf);
     puts(buf);
}
```

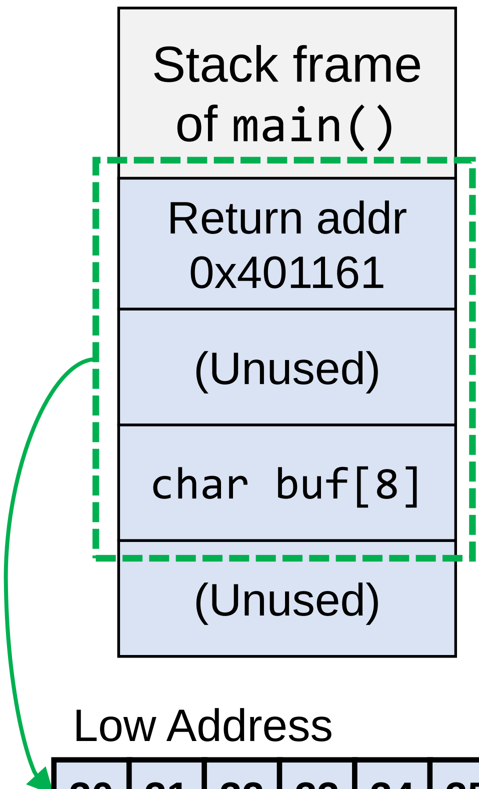
```
sub    $0x18,%rsp
lea    0x8(%rsp),%rdi
...
add    $0x18,%rsp
ret
```

```
jschoi@ubuntu:~$ ./bof
0123456789ABCDEF
0123456789ABCDEF
Segmentation fault
```

ret will jump to 0x401100 (malfunction and crash)



Example Input #4: Control Hijack




```
void echo(void) {
     char buf[8];
     gets(buf);
     puts(buf);
}
```

```
sub     $0x18,%rsp
lea     0x8(%rsp),%rdi
...
add     $0x18,%rsp
ret
```

```
jschoi@ubuntu:~$ ./bof
0123456789ABCDEFabc
0123456789ABCDEFabc
Segmentation fault
```

ret will jump to 0x636261, which is attacker's input

Control Hijack!

Low Address — High Address

30	31	32	33	34	35	36	37	38	39	41	42	43	44	45	46	61	62	63	00	00	00	00	00
`char buf[8]`								Unused 8 bytes								Return address							

We’ve seen that hackers can manipulate program counter (control hijack) by corrupting the stack

… but how does it lead to arbitrary code execution?

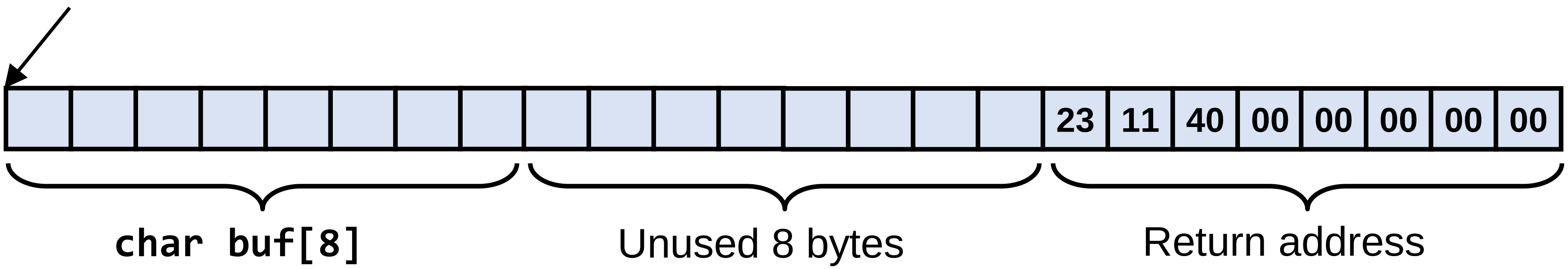
Code Execution

- **Inject malicious code in the memory (e.g., in `buf[]`)**
  - And overwrite the return address with the address of **`buf`**

```c
void echo(void) {
    char buf[8];
    gets(buf);
    puts(buf);
}
```

```
jschoi@ubuntu:~$ ./bof
j0YX45P... (omitted)
j0YX45P... (omitted)
Segmentation fault
```

Let's assume this address is **`0x606060`**



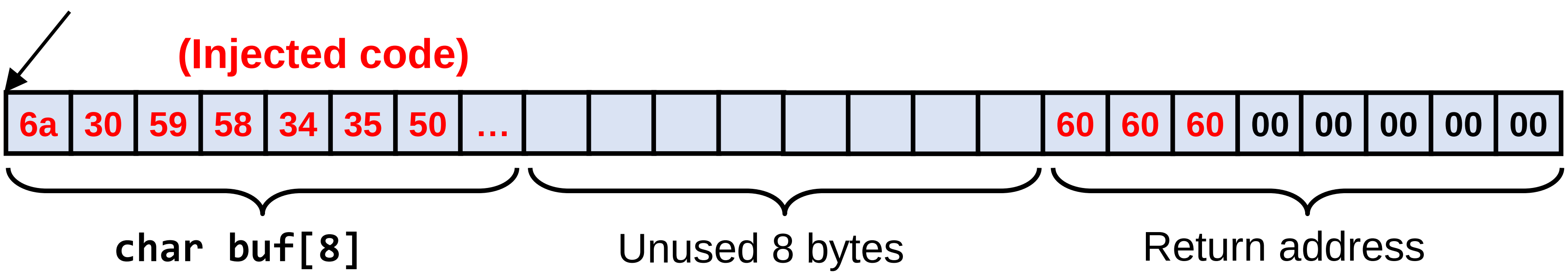
Code Execution

- **Inject malicious code in the memory (e.g., in `buf[]`)**
  - And overwrite the return address with the address of **`buf`**

```
void echo(void) {
    char buf[8];
    gets(buf);
    puts(buf);
}
```

```
jschoi@ubuntu:~$ ./bof
j0YX45P... (omitted)
j0YX45P... (omitted)
Segmentation fault
```

Let’s assume this address is **`0x606060`**

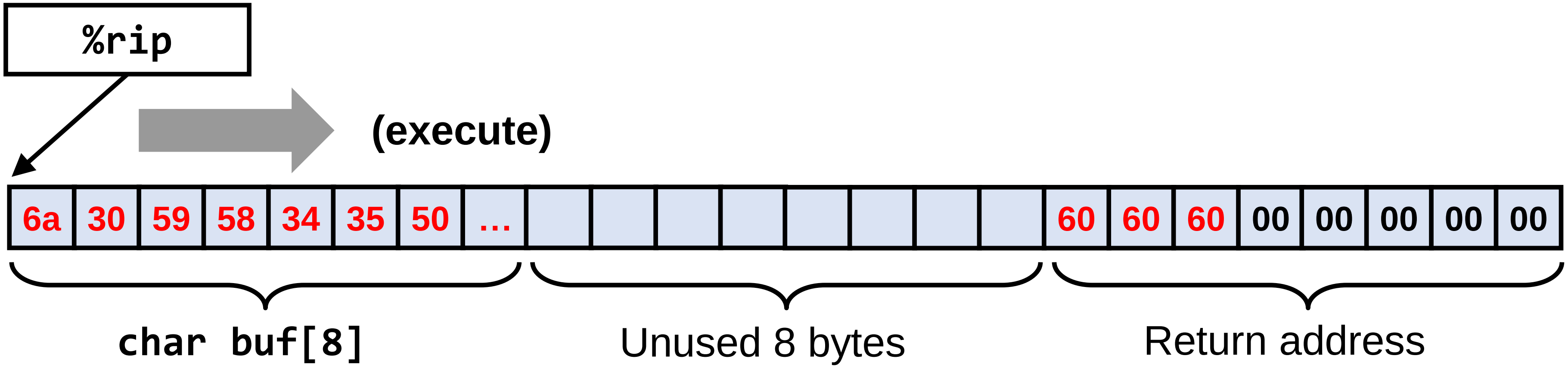


Code Execution

- **Inject malicious code in the memory (e.g., in `buf[]`)**
  - And overwrite the return address with the address of **`buf`**
  - Upon return, the content of **`buf`** will be executed as instructions

```
void echo(void) {
    char buf[8];
    gets(buf);
    puts(buf);
}
```

```
jschoi@ubuntu:~$ ./bof
j0YX45P... (omitted)
j0YX45P... (omitted)
Segmentation fault
```



How can hacker inject "code"?

- **Program reads in string (data) as input, how can a hacker inject "code" into the program memory?**
  - In fact, there is nothing special that the hacker has to do
- **Recall that machine code is just a sequence of bytes**
  - Just like any other data (e.g., integers, strings)
- **In the previous page, "`j0YX45P...`" was used as input**
  - ASCII code of this string is: `6A 30 59 58 34 35 50`
  - These bytes are also interpretable as x86-64 instructions below

```
   0:   6a 30                   push   $0x30
   2:   59                      pop    %rcx
   3:   58                      pop    %rax
   4:   34 35                   xor    $0x35,%al
   6:   50                      push   %rax
```

Shellcode

- **In the previous page, I said *"inject malicious code"***
  - But what kind of malicious code?
- **Once executed, this code will spawn a shell**
  - If a shell is given, hacker can run any command from now on!
  - Such kind of malicious code is called ***shellcode***
  - Roughly speaking, it is **`execve("/bin/sh")`** written in assembly instructions

```
# Shellcode Example
xor     %rdx, %rdx
mov     $0x6873..., %rbx
...
mov     $0x3b, %al
syscall
```

Run

```
jschoi@ubuntu:~$ ./bof
... (omitted)
... (omitted)
$ ls; rm -rf *
```

Shell is spawned

Topics

- Memory layout of a program
- Basic concept of buffer overflow
  - Stack memory corruption and control hijack
  - Exploitation with shellcode
- **The first round of war between attacker vs. defender**
  - **(Mitigation) Stack canary, NX**
  - **(Bypassing) Memory disclosure**

Defense against BOF

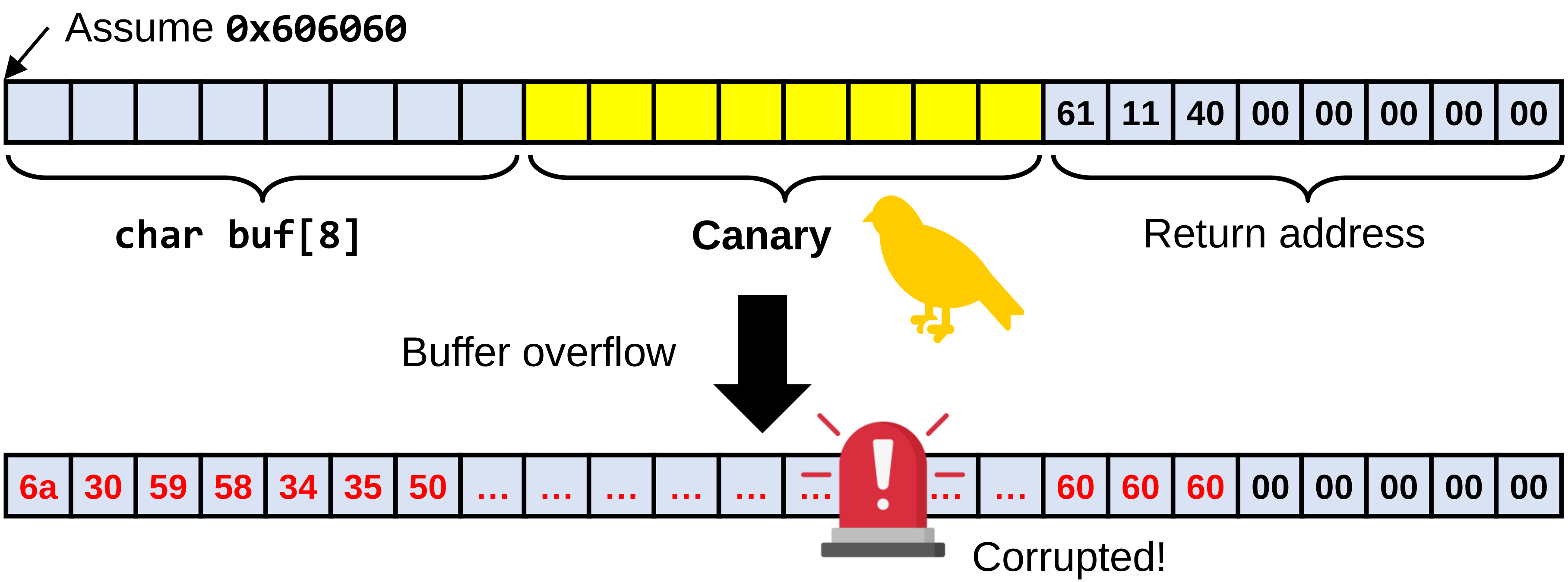
- **How can we protect a program from BOF, then?**
- **Solution 1: Removing the buffer overflow itself**
  - Ex) Replace with **`fgets()`**, **`scanf("%8s",...)`**, etc.

```
void safe_echo(void) {
    char buf[8];
    fgets(buf, 8, stdin);
    puts(buf);
}
```

  - However, complete elimination of program bugs is hard
- **Solution 2: Exploit mitigation**
  - Even if a bug exists, we can **make it hard to exploit** that bug
  - Needs coordination of CPU, OS, compiler, etc.

Mitigation: Stack Canary

- **Place randomized bytes called canary* between the buffer and the return address**
  - Canary is prepared right after entering a function
  - Before the function returns, check if it was changed (corrupted)

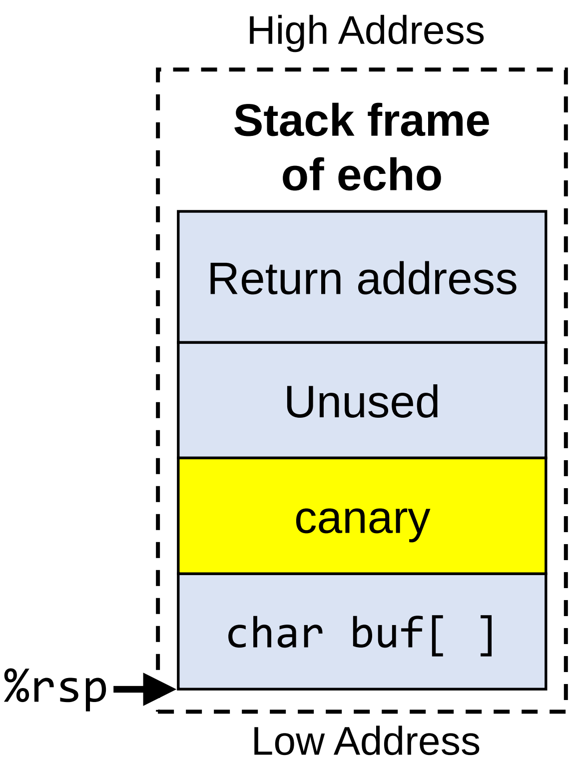


***Canary**: Name of a bird that miners brought to a cave for gas leak detection

Credit: Icons are from Flaticon (by Freepik)

Assembly Code for Stack Canary

- **Nowadays, compilers will emit the following code**




```
(gdb) disassemble echo
0x401146: sub    $0x18,%rsp
0x40114a: mov    %fs:0x28,%rax           Canary
0x401153: mov    %rax,0x8(%rsp)          setup
0x401158: xor    %eax,%eax
0x40115a: mov    %rsp,%rdi
0x40115d: call   0x401050 <gets@plt>
0x401162: mov    %rsp,%rdi
0x401165: call   0x401030 <puts@plt>
0x40116a: mov    0x8(%rsp),%rax          Canary
0x40116f: sub    %fs:0x28,%rax           check
0x401178: jne    0x40117f <echo+57>
0x40117a: add    $0x18,%rsp
0x40117e: ret
0x40117f: call   0x401040 <__stack_chk_fail@plt>
```

`%fs:0x28` stores random bytes that hackers can't know

Bypassing Stack Canary

- **Does stack canary prevent all security issues? NO**
- **First, overwriting other local variables in the same stack frame is already a serious problem**
  - Even if the hacker cannot overwrite the return address
- **Next, hackers can still exploit BOF in heap memory**
  - Ex) Overwrite function pointer field in a heap-allocated `struct`
- **Also, certain type of BOF can't be detected with canary**
  - Recall the general discussion in page 10
- **Last but not least, hackers can disclose the memory content and learn the stack canary value!**

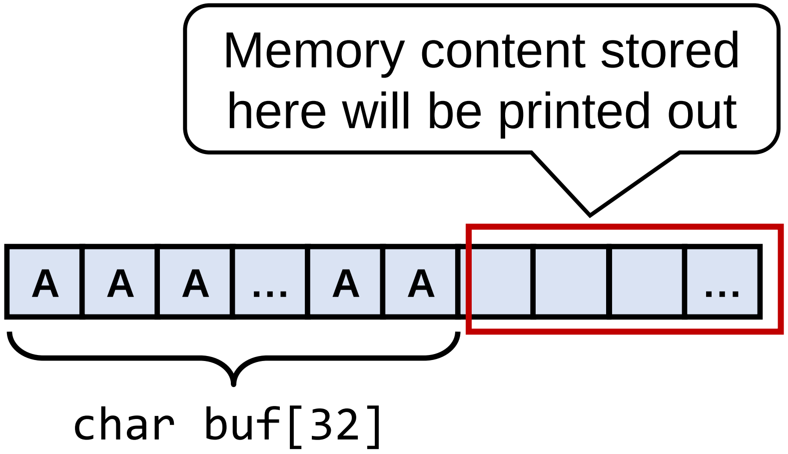
Memory Disclosure

- **Exploiting a vulnerability to disclose some information in the memory**
- **Again, misuse of array is the most common source of vulnerability that allows memory disclosure**
  - Buffer overflow that ***reads*** the data past the end of an array
  - Of course, BOF is not the **only** source of memory disclosure
- **Various kind of information can be disclosed**
  - Private user data, secret key in cryptography, etc.
  - In this slide, let’s focus on disclosing the **stack canary value**
    - If stack canary value is known, hacker can overwrite return address and **pretend as if nothing has happened**

Memory Disclosure Example

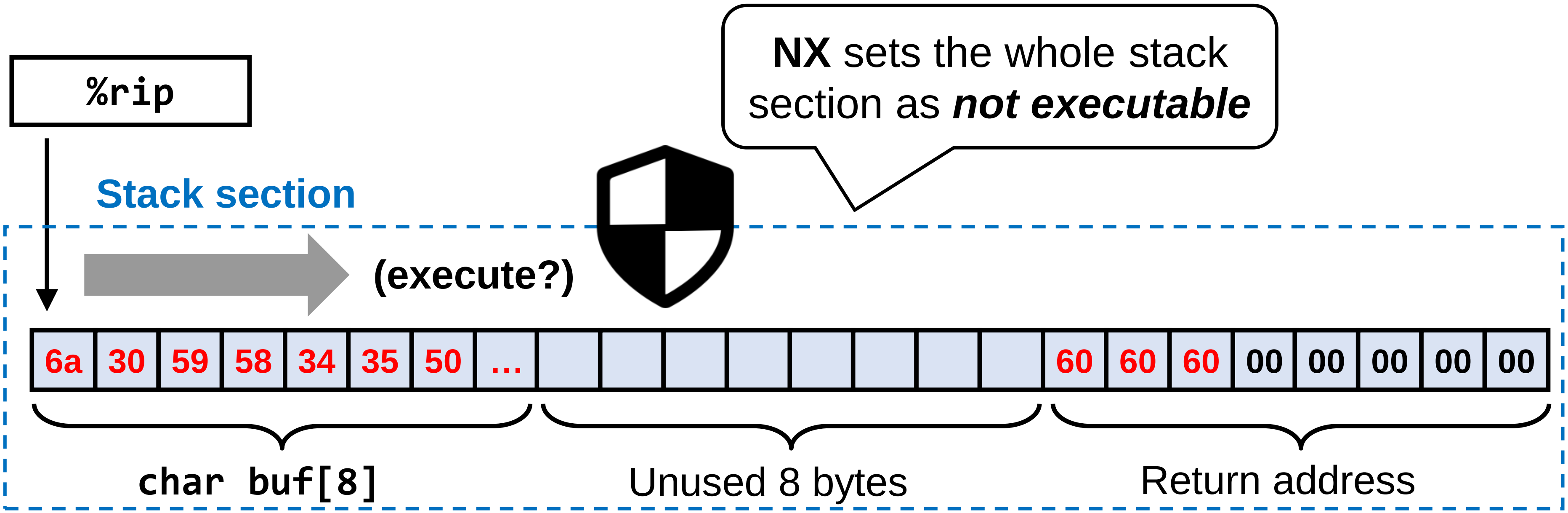
- **In the code below, `write(1, buf, len)` prints out `len` bytes of data stored in `buf`**
  - Unlike `printf("%s", buf)`, it does not stop at NULL character
- **The famous *Heartbleed* vulnerability was also caused by a similar mistake of trusting user input**
  - Review **Chapter 1. Overview**

```c
void vuln(void) {
  char buf[32] = {'A', ...};
  int len;
  scanf("%d", &len);
  write(1, buf, len);
}
```



Mitigation: NX

- **NX*: <u>N</u>on-e<u>x</u>ecutable memory**
- **In old systems, all memory sections were executable**
- **NX introduced *execute* permission for each section**
  - Mark Stack, Heap, Data section as non-executable (cf. page 3)



*Called DEP (data execution protection) in Windows

Bypassing NX?

- NX can effectively prevent the execution of shellcode injected in the memory
- So does NX completely prevent code execution through buffer overflow exploitation?

Of course, the answer is NO

In the next chapter, we will cover the second round of attacker vs. defender

Side-Note:
Access Control & SUID

Access Control

- **Intuitively, access control is about *what kind of permission should be given* to each user of a system**
  - There are formal models about this, but let's keep it simple here
  - Linux file system is a good example:
    - Any user can execute **`cat`**, but cannot modify its content
    - Only **`jason`** user can access the **`secret.txt`** file

```
/home/jason $ ls -l /usr/bin/cat
-rwxr-xr-x 1 root root 35280  /usr/bin/cat
/home/jason $ ls -l secret.txt
-rw------- 1 jason jason 16  secret.txt
```

Setuid Bit (SUID)*

- **Have you ever wondered how `passwd` command works?**
  - This command must update **`/etc/shadow`** file
  - **`/etc/shadow`** file is writable only by root, of course
  - Then how can you update your password (as a non-root user)?
- ***Setuid bit* is a mechanism that enables this**
  - When you execute **`/usr/bin/passwd`**, you temporarily run it with the privilege of the file owner (**`root`** in this case)

```
/home/jason $ ls -l /etc/shadow
-rw-r----- 1 root shadow 828  /etc/shadow
/home/jason $ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 59976  /usr/bin/passwd
```

*Not to be confused with `setuid()` function

What if SUID program has BOF?

- **The expected behavior of `/usr/bin/passwd` is fixed**
  - It must read in your new password twice, compare if they are same, and then update `/etc/shadow` file
- **But if `/usr/bin/passwd` has BOF, hacker can exploit it and make the program do other things**
  - Run the code that the hacker (not the developer) wants
  - Ex) Hacker can even make it run `execve("/bin/bash"...)`
  - … what happens then?