

**Assignment - 4**  
**(Double Free Vulnerability)**

Name - Vasu A. Bhatt

Roll No - CS23M072

Secure Systems Engineering  
(CS6570)

## Table of Contents

<b>Table of Contents.....</b>	<b>1</b>
<b>Challenge : 1.....</b>	<b>2</b>
❖ Tools Used:.....	2
❖ Technique Used In Exploit:.....	3
❖ Flag:.....	5
❖ Output:.....	6
<b>Challenge : 2.....</b>	<b>7</b>
❖ Tools Used:.....	7
❖ Technique Used In Exploit:.....	8
❖ Flag:.....	11
❖ Output:.....	12

# **Challenge : 1**

## ❖ Tools Used:

1. Pwndbg - For debugging the given binary.
2. Pwntools - For implementing the format for exploit script

### ❖ Technique Used In Exploit:

- Given C code has the double free vulnerability which allows us to change the content of any memory location (Write permitted) of a given process with our desired content.
- The idea is to use this vulnerability to subvert the flow of execution to execute the 'binsh()' function, which is effectively calling the shell.
- Here, we have used the '\_\_\_free\_hook' as our target location where we are going to write the address of 'binsh()' function using double free vulnerability, which will effectively spawn the shell.
- '\_\_\_malloc\_hook' and '\_\_\_free\_hook' are function pointers provided by the GNU C Library (glibc) that allow programmers to customise the behaviour of memory allocation (malloc) and deallocation (free).
- Programmers can set these hooks to point to their custom function, allowing them to add additional behaviour (such as logging or memory tracking) when calling 'free()' and 'malloc()' functions.
- This will allow us to change the program's behaviour when calling the 'free()' after we have successfully replaced the address pointed by the '\_\_\_free\_hook' using double free vulnerability.
- For exploiting double free vulnerability first we will 'malloc()' using command 'g' 9 times, which will allocate 9 chunks in the heap.
- Now we will 'free()' using command 'd', 7 times which will store 7 chunks in the 'tcache' bins.

- Now we will 'free()' 8th chunk and 9th chunk which are going to get stored in 'fast' bins, and freeing one more time will create an alias(Not physically present but 'ptmalloc' treating it as a different chunk) of 8th chunk and on the next ptr address of 9th chunk, address of 8th chunk will get stored in 'fast' bin.
- Now we will again 'malloc()' 7 times which will remove the 7 chunks stored inside 'tcache' bin.
- When we 'malloc()' again, the 8th chunk will be remove from 'fast' bin, and where we will store the address of '\_\_\_free\_hook' in place of the forward pointer,hence in the alias also this value will be stored.
- Now we will 'malloc()' for two more times which will remove the 9th chunk and the 8th chunk's alias.
- Now in the fast bin the address of '\_\_\_free\_hook' is stored as the valid free chunk, and on 'malloc()' it will give it as the chunk to use, where we will store the address of 'binsh()' function.
- Now '\_\_\_free\_hook' is pointing to the 'binsh()' function which will be called when we do 'free()' in the end, spawning a cell, and by 'cat' command on the server we can read the flag stored in that.

❖ Flag:

SSE24{fr33\_cyc1ing\_0n\_th3\_h34p}

## ❖ Output:

```
ubuntu@ubuntu-2204: ~/Downloads/exploits_sse-20240405T061403Z-001/cs6570_assignment_4_pas...
- [p]rint all your tokens
- e[x]it the program
Action: Enter the name for the token:
=====
What can I do for you?
- [g]enerate a new token
- [d]iscard an existing token
- [p]rint all your tokens
- e[x]it the program
Action: Enter the name for the token:
=====
What can I do for you?
- [d]iscard a new token
- [d]iscard an existing token
- [p]rint all your tokens
- e[x]it the program
Action: Enter the name for the token:
=====
What can I do for you?
- [g]enerate a new token
- [d]iscard an existing token
- [p]rint all your tokens
- e[x]it the program
Action: Enter the index of the token: : 1: x: not found
$ ls
flag
run
$ cat flag
SSE24{fr33_cycling_0n_th3_h34p}
[*] Got EOF while reading in interactive
$
```

Above image shows the working of exploit and extracting the flag from the server

## **Challenge : 2**

### ❖ Tools Used:

1. Pwntools - For implementing the format for exploit script
2. One\_gadget - For finding the gadget of `execve("/bin/sh")` for spawning shell



## ❖ Technique Used In Exploit:

- Given C code has the double free vulnerability which allows us to change the content of any memory location (Write permitted) of a given process with our desired content.
- The idea is to use this vulnerability to subvert the flow of execution to execute the gadget of 'execve("/bin/sh")' function, which is effectively calling the shell.
- The gadget 'execve("/bin/sh")' can be found by using 'one\_gadget' which finds it in the 'glibc' library and gives us the address of it inside it.

```

ubuntu@ubuntu-2204: ~/Downloads/Assignment_4
ubuntu@ubuntu-2204:~/Downloads/Assignment_4$ one_gadget libc.so.6
0x4f29e execve("/bin/sh", rsp+0x40, environ)
constraints:
  address rsp+0x50 is writable
  rsp & 0xf == 0
  rcx == NULL || {rcx, "-c", r12, NULL} is a valid argv

0x4f2a5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  address rsp+0x50 is writable
  rsp & 0xf == 0
  rcx == NULL || {rcx, rax, r12, NULL} is a valid argv

0x4f302 execve("/bin/sh", rsp+0x40, environ)
constraints:
  [rsp+0x40] == NULL || {[rsp+0x40], [rsp+0x48], [rsp+0x50], [rsp+0x58], ...} is a valid argv

0x10a2fc execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL || {[rsp+0x70], [rsp+0x78], [rsp+0x80], [rsp+0x88], ...} is a valid argv
ubuntu@ubuntu-2204:~/Downloads/Assignment_4$

```

Above image shows the use of 'one\_gadget' which is giving us the address of our gadget

- Here, we have used the '\_\_\_free\_hook' as our target location where we are going to write the address of 'execve("/bin/sh")' gadget using double free vulnerability, which will effectively spawn the shell.
- '\_\_\_malloc\_hook' and '\_\_\_free\_hook' are function pointers provided by the GNU C Library (glibc) that

allow programmers to customise the behaviour of memory allocation (malloc) and deallocation (free).

- Programmers can set these hooks to point to their custom function, allowing them to add additional behaviour (such as logging or memory tracking) when calling 'free()' and 'malloc()' functions.
- This will allow us to change the program's behaviour when calling the 'free()' after we have successfully replaced the address pointed by the '\_\_free\_hook' using double free vulnerability.
- First we are storing the base address of 'glibc' provided in the beginning by the challenge.
- We will then find the actual addresses of '\_\_free\_hook' and 'execve("/bin/sh")' gadget by adding that base value to their relative addresses in glibc.
- For exploiting double free vulnerability first we will 'malloc()' using command 'g' 9 times, which will allocate 9 chunks in the heap.
- Now we will 'free()' using command 'd', 7 times which will store 7 chunks in the 'tcache' bins.
- Now we will 'free()' 8th chunk and 9th chunk which are going to get stored in 'fast' bins, and freeing one more time will create an alias (Not physically present but 'ptmalloc' treating it as a different chunk) of 8th chunk and on the next ptr address of 9th chunk, address of 8th chunk will get stored in 'fast' bin.
- Now we will again 'malloc()' 7 times which will remove the 7 chunks stored inside the 'tcache' bin.
- When we 'malloc()' again, the 8th chunk will be removed from 'fast' bin, and where we will store the address of '\_\_free\_hook' in place of the forward

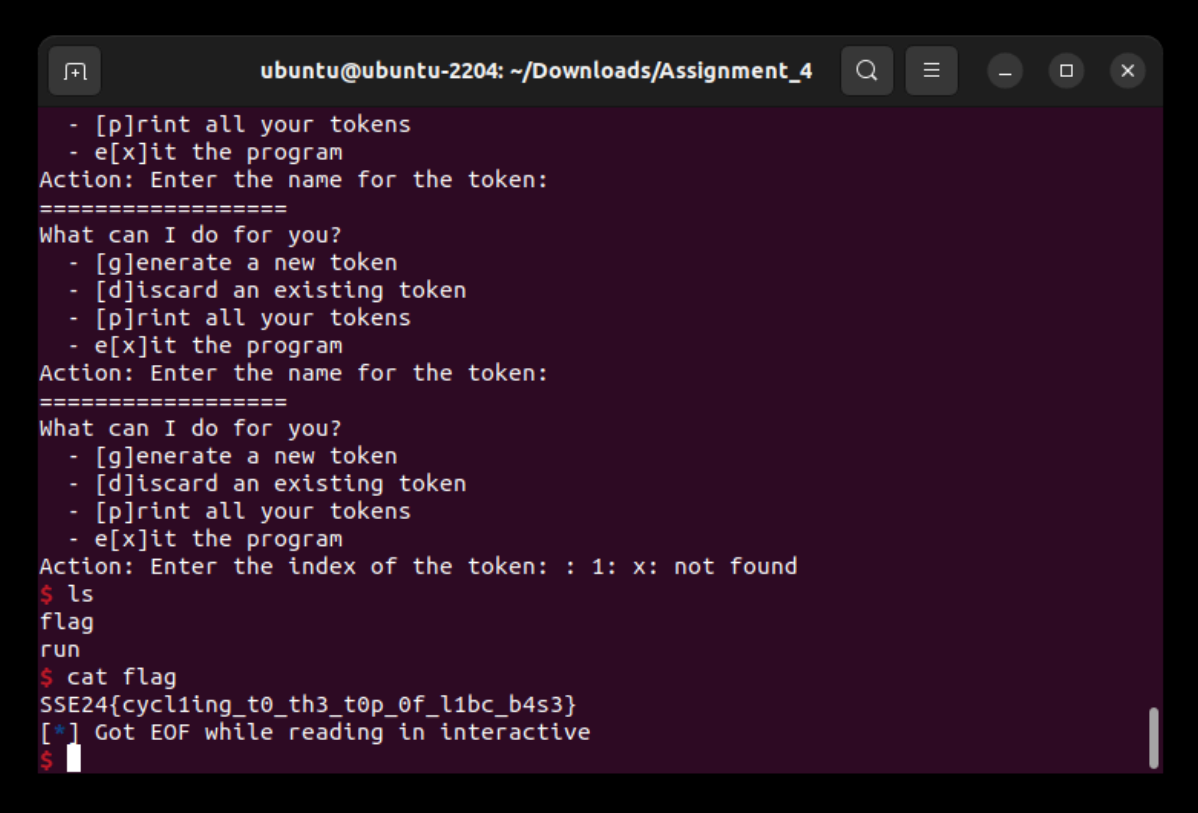
pointer, hence in the alias also this value will be stored.

- Now we will 'malloc()' for two more times which will remove the 9th chunk and the 8th chunk's alias.
- Now in the fast bin the address of '\_\_\_free\_hook' is stored as the valid free chunk, and on 'malloc()' it will give it as the chunk to use, where we will store the address of 'execve("/bin/sh")' gadget.
- Now '\_\_\_free\_hook' is pointing to the 'execve("/bin/sh")' gadget which will be called when we do 'free()', spawning a shell, and by 'cat' command on the server we can read the flag stored in that.

❖ Flag:

SSE24{cycl1ing\_t0\_th3\_t0p\_0f\_l1bc\_b4s3}

## ❖ Output:

A terminal window titled 'ubuntu@ubuntu-2204: ~/Downloads/Assignment\_4' with standard Ubuntu window controls. The terminal shows the output of an exploit session. It starts with a menu: '- [p]rint all your tokens', '- e[x]it the program'. The user enters 'Action: Enter the name for the token:', followed by a separator '=====' and the prompt 'What can I do for you?'. The menu is repeated. The user enters 'Action: Enter the name for the token:', followed by another separator and the prompt. The user then enters 'Action: Enter the index of the token: : 1: x: not found'. The user runs '\$ ls' and the output is 'flag' and 'run'. The user runs '\$ cat flag' and the output is 'SSE24{cyclling\_t0\_th3\_t0p\_0f\_l1bc\_b4s3}'. The session ends with '[\*] Got EOF while reading in interactive' and a final '\$' prompt.

```
ubuntu@ubuntu-2204: ~/Downloads/Assignment_4
- [p]rint all your tokens
- e[x]it the program
Action: Enter the name for the token:
=====
What can I do for you?
- [g]enerate a new token
- [d]iscard an existing token
- [p]rint all your tokens
- e[x]it the program
Action: Enter the name for the token:
=====
What can I do for you?
- [g]enerate a new token
- [d]iscard an existing token
- [p]rint all your tokens
- e[x]it the program
Action: Enter the index of the token: : 1: x: not found
$ ls
flag
run
$ cat flag
SSE24{cyclling_t0_th3_t0p_0f_l1bc_b4s3}
[*] Got EOF while reading in interactive
$
```

Above image shows the working of exploit and extracting the flag from the server