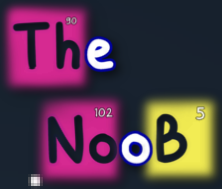
WargamesMY 2021 CTF Writeup

11 December 2021

By TheNooB



easyrsa

In this challenge, we are given a python script `chal.py`:

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

# Generate public key
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}")
```

We're also given the output of the code:

```
# Output:
# n =
18304313499627278872497347106781088765844971752924494936581137294399251598122054491970352624997804891597368572151975199012875361892862378834285683652903007044947055619342684830043887388520198889986897901082447405903406407594879218477486010303949150584062013743002102223027695933055784742794025435157653364122080068590543935345708412031440210341402077436597564563551761860586694310508179443543695439431800710044789886211675787943097380881754374101281232331925135583396076104230927588170771167823704058142635052157494826748528892520175896965304225473931239786248838890697275575852108793053198833013718162055280904344071
# c =
32659517071722427097274727393868734947032499122855052653711463931960303724137818039301646631493722284393337338650823300116427049130912496534972003780944335044675673331902347399251034625522516076080289558400382270145968250408400105717421279883105890875233026757400073857141322987269830394697983222045539746252193303031007436789219663978696572790141244587721946385401837555648059230249319084653095611900234769213541458191562314660299246403058564728320063378773616082325606774278596866388424931701492973081462101902738956895920426343367096059611851871333644229929157413629018144692136808944126707871537758673177856001071# hint =
18304313499627278872497347106781088765844971752924494936581137294399251598122054491970352624997804891597368572151975199012875361892862378834285683652903007044947055619342684830043887388520198889986897901082447409085241920609043746758562546428726601779051836945819727406203022234361932871768234345227095844165965086543100805475233734788611736563726752110713668306358343947215223729066594029304583431226741472301578327786047842152745501205045067602882077634449040311631828852598470552682455871506975125259548912060
```

“This challenge should be quite straightforward for those who know how RSA works.”, but this person is not me. Funnily enough, I actually learnt about RSA before from other CTF challenge, but just doesn’t have the incentive to remember it, I guess examination is still important? Anyway, I ended up relearning the algorithm from Wikipedia of course. Summary below taken from wiki:

1. Choose two distinct prime numbers p and q.
2. Compute $n = pq$.
3. Compute $\lambda(n)$, where λ is Carmichael's totient function. Since $n = pq$, $\lambda(n) = \text{lcm}(\lambda(p),\lambda(q))$, and since p and q are prime, $\lambda(p) = \varphi(p) = p - 1$ and likewise $\lambda(q) = q - 1$. Hence $\lambda(n) = \text{lcm}(p - 1, q - 1)$.
4. Choose an integer e such that $1 < e < \lambda(n)$ and $\gcd(e, \lambda(n)) = 1$; that is, e and $\lambda(n)$ are coprime.
5. Determine d as $d \equiv e^{-1} \pmod{\lambda(n)}$; that is, d is the modular multiplicative inverse of e modulo $\lambda(n)$.

To be honest, I only partially understand what’s going on, since I don’t have enough math background to do this (Like I didn’t even learn about modular arithmetic, let alone those weird lambda phi thingy). But that’s fine, let’s keep going.

On the same Wikipedia page, I saw this: “In the original RSA paper, the Euler totient function $\varphi(n) = (p - 1)(q - 1)$ is used instead of $\lambda(n)$ for calculating the private exponent d…”

Hmm interesting, we might need this later. (Yes we need this)

Let’s analyse what we have. Of course we have the cipher text *c*, and also *n*=*p***q*. We are also given a hint which is computed with *p***q*-*p*-*q*+1. From the code, we also get *e*=0x1001 (which translate to 65537, but we can just keep it in hexadecimal form.)

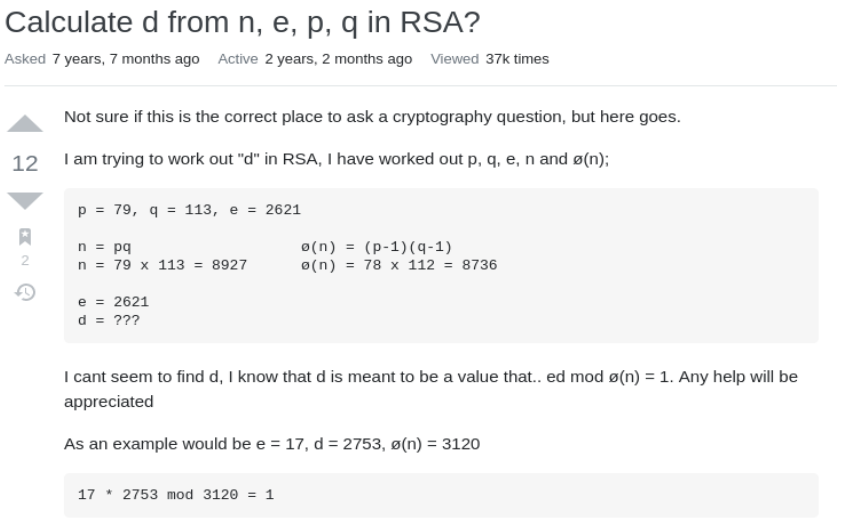
To decrypt the ciphertext, we need to compute $d \equiv e^{-1} \pmod{\lambda(n)}$. We do have *e* but we don’t have $\lambda(n)$. Wait just now I saw something: “…**$\varphi(n) = (p - 1)(q - 1)$** is used instead of ***λ(n)***…” Ok but what is $(p - 1)(q - 1)$? Expanding it yield:

$$(p-1)(q-1) = pq - p - q + 1$$

which looks a bit familiar, it is the hint. So we can compute *d* with the hint instead and solve for the plain text. Simple right? Just plug them into the formula? Wait what does the formula even mean?

$$d \equiv e^{-1} \mod \varphi(n)$$

The “-1” at *e* is not $\frac{1}{e}$, but is something called “modular multiplicative inverse” of *e*. Good, I don’t know how to do that, time for research:



Calculate d from n, e, p, q in RSA?

Asked 7 years, 7 months ago Active 2 years, 2 months ago Viewed 37k times

12

Not sure if this is the correct place to ask a cryptography question, but here goes.

I am trying to work out "d" in RSA, I have worked out p, q, e, n and ø(n);

```
p = 79, q = 113, e = 2621

n = pq                  ø(n) = (p-1)(q-1)
n = 79 x 113 = 8927     ø(n) = 78 x 112 = 8736

e = 2621
d = ???
```

I cant seem to find d, I know that d is meant to be a value that.. ed mod ø(n) = 1. Any help will be appreciated

As an example would be e = 17, d = 2753, ø(n) = 3120

```
17 * 2753 mod 3120 = 1
```

And the answer:

12

You are looking for the modular inverse of *e* (mod *n*), which can be computed using the extended Euclidean algorithm:

```
function inverse(x, m)
    a, b, u := 0, m, 1
    while x > 0
        q := b // x # integer division
        x, a, b, u := b % x, u, x, a - q * u
    if b == 1 return a % m
    error "must be coprime"
```

Thus, in your examples, `inverse(17, 3120)` = 2753 and `inverse(2621, 8736)` = 4373. If you don't want to implement the algorithm, you can ask Wolfram|Alpha for the answer.

Now I can just copy the function. Oh wait it is not in Python, I have to translate it myself:

```python
def inverse(x, m): #x is e, m is phin
    a, b, u = 0, m, 1
    while x > 0:
        q = b // x # integer division
        x, a, b, u = b % x, u, x, a - q * u
    if b == 1:
        return a % m
```

After getting *d*, we can find the plain text with this:

$$m(c) = c^d \mod n$$

And finally my solution:

```python
from Crypto.Util.number import *

n = [REDACTED TO SAVE SPACE]
c = [REDACTED TO SAVE SPACE]
e = 0x10001
hint = [REDACTED TO SAVE SPACE]

def inverse(x, m): #x is e, m is phin
    a, b, u = 0, m, 1
    while x > 0:
        q = b // x # integer division
        x, a, b, u = b % x, u, x, a - q * u
    if b == 1:
        return a % m

d = inverse(e,hint)

msg = pow(c, d, n)
msg = long_to_bytes(msg)
msg = msg.decode("utf-8")

print(msg)
```

Output: `wgmy{227d1562df0d940d94d75b0512f4bc6c}`

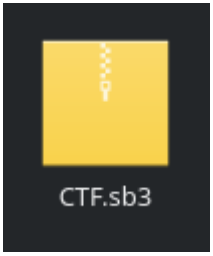
Note: After reading writeup by H0j3n, I realised there is built in implementation of modular inverse from Python 3.8 onward, so you might be interested in that as well:

> Python 3.8 now comes with the functionality of computing modular inverses. In this case, the `exp` argument may be negative, on the condition that base is *relatively prime* to mod, i.e, the only common integer divisor of base and mod is 1.
>
> So, when using the `pow()` function with negative `exp`, the function will perform as follows:
>
> ```
> pow(inv_base, -exp, mod)
> ```

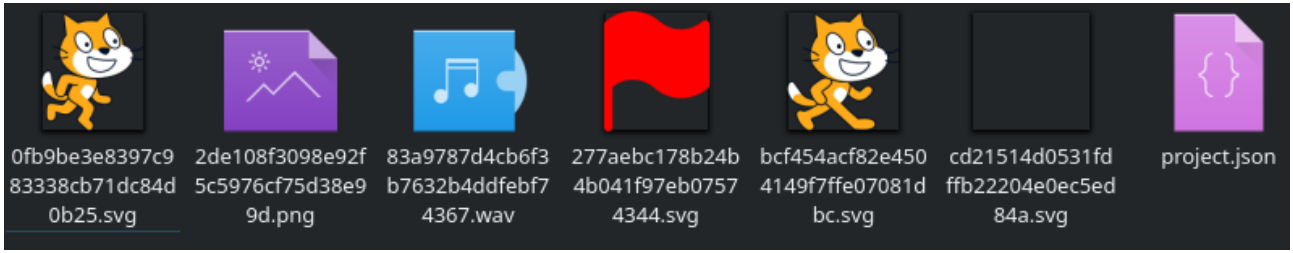
Capture The Flag

Challenge description: A file from an orange cat, it say better start from scratch.

We are given a file `CTF.sb3`. Upon downloading it, it looks like an archive file in `Dolphin`:



So of course, I extracted it:

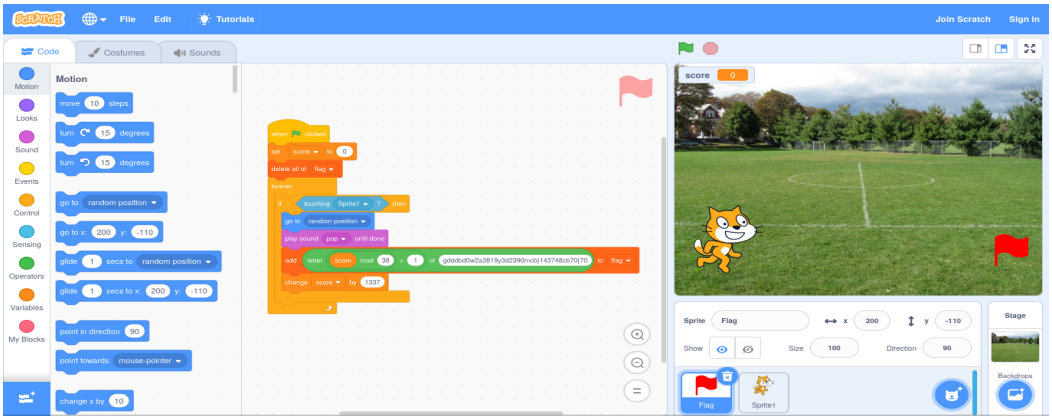


Ermm the content make no sense, wait that cat… ain't that Scratch logo? Oh yeah the challenge description: "A file from an orange cat, it say better start from scratch."

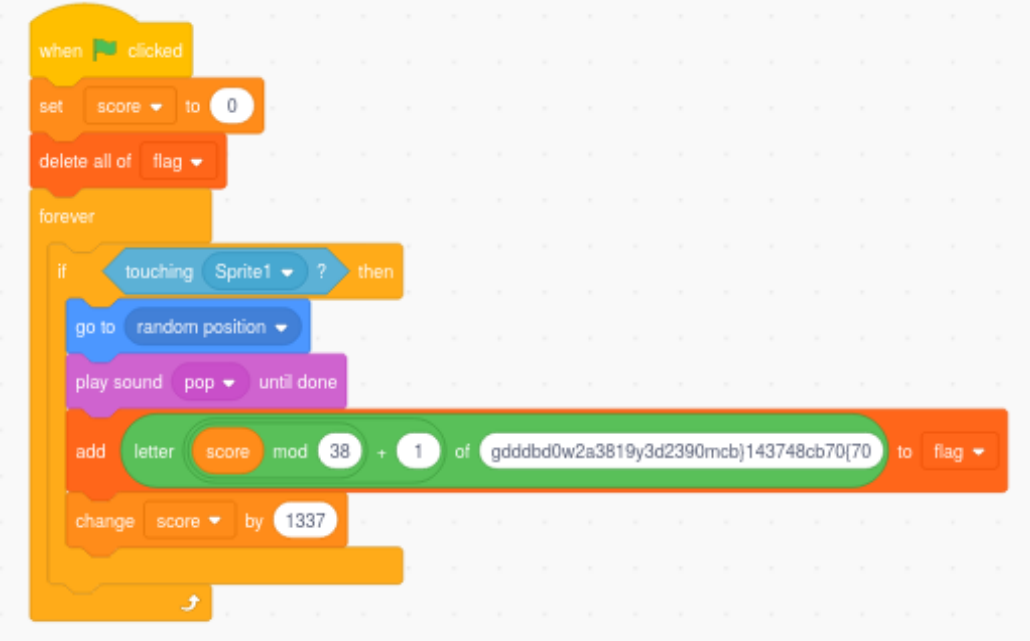
What is an SB3 file?

An SB3 file is a project created with Scratch 3.0, a program development platform created at the Massachusetts Institute of Technology (MIT). It contains a program written in Scratch programming language, which is used for creating stories, small games, and animations. SB3 files replaced Scratch .SB2 files.

Yeah it makes sense now. So let's load it into Scratch at https://scratch.mit.edu/projects/editor/:



See, I never used Scratch before, so I'm not really familiar with how things work. But I can read pseudocode tho:



It sounds like the flag is generated from the string `gdddbd0w2a3819y3d2390mcb}143748cb70{70` by rearranging it in some way shown in the code. Though we need to keep in mind that `1` in Scratch actually represent the first letter, unlike most language where `0` is the first element. Translating it into Python:

```
shuffled = "gdddbd0w2a3819y3d2390mcb}143748cb70{70"

score = 0
last = shuffled[(score % 38)]
flag = last

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

print(flag)
```

Output: `gwym7{b831bd23c47d1947dadb8009030d32}c`

Uh, seems like the output need some processing. From there you can probably easily guessed that first character need to be swapped with second, third with fourth and so on, just based on the fact that flag start with `wgmy{`.

I’m not sure how Scratch works, but upon running the program and playing it, it showed some new code which confirmed this guess:

So let’s continue writing code:

```
def swap(s, i, j): #Some code copied from stackexchange ofc
    i -= 1
    j -= 1
    return ''.join((s[:i], s[j], s[i+1:j], s[i], s[j+1:]))

for i in range(1,int(len(flag)/2)+1):
    n = 2*i-1
    flag = swap(flag,n,n+1)

print(flag)
```

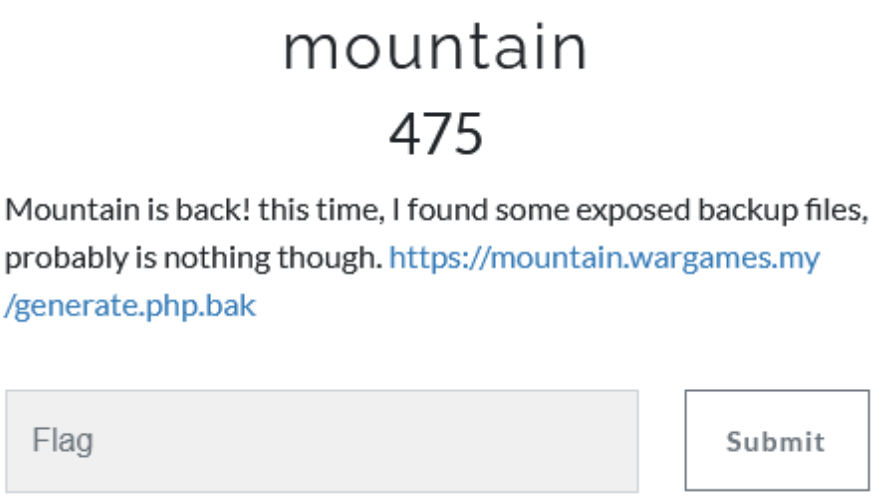
Output: `wgmy{78b13db324cd79174adbd089030d023c}`

Full code:

```
shuffled = "gdddbd0w2a3819y3d2390mcb}143748cb70{70"

def swap(s, i, j):
    i -= 1
    j -= 1
    return ''.join((s[:i], s[j], s[i+1:j], s[i], s[j+1:]))

score = 0
last = shuffled[(score % 38)]
flag = last

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

for i in range(1,int(len(flag)/2)+1):
    n = 2*i-1
    flag = swap(flag,n,n+1)

print(flag)
```

mountain

I have screenshot for this (only the challenge description tho :( ):



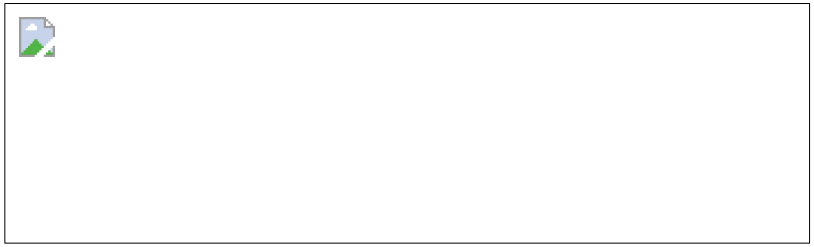
Content of `generate.php.bak`:

```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;
}
?>
```

From the content we can see that something called `verify_code` is generated with `mt_rand`, and `mt_srand` is called with `verify_code`. Next, `acc_passwd` is generated with `mt_rand` again. Ducky search reveals

`mt_rand` is a random number generator based on Mersenne Twister, while the function `mt_srand` is used to set the seed of the RNG.

To put it simple, a random 10-digits number is generated and used as `verify_code` which is then used to seed to RNG before generating the `acc_passwd`. It might be unclear what those variables are, so let's take a look at the website at mountain.wargames.my:



Ok never mind I didn't screenshot it but basically the website is like this:

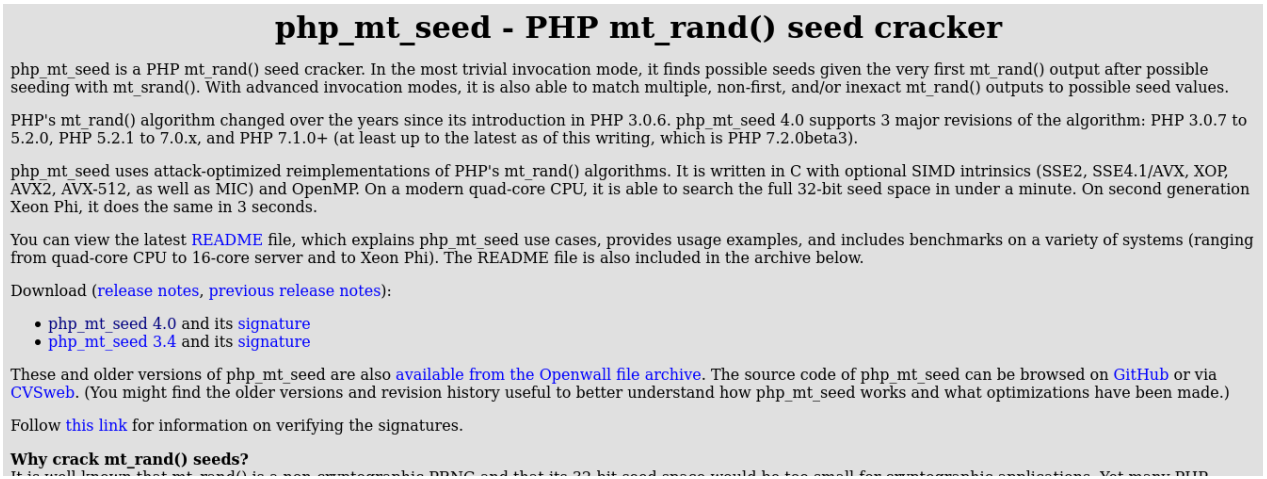
1. The homepage consist of registration section and login section (and msg saying that they're going bankrupt, but unrelated xd).
2. To register, you have to enter username, email and solve captcha. Here's a catch, the email must have domain of `@wargames.gov.my`, which needless to say doesn't exist. Oh yeah I tried using guerillamail and apparently there's an easter egg with that?
3. After registering with proper email and username, the following message is shown:

```
Success! Hello user3, your password is: 1900165458
You need to activate your account first. Check your email for
activation/verification link.

The link might look something like:
https://mountain.wargames.my/verify.php?
username=henson&verify=1929258756
```

Ok so things make sense now. `verify_code` probably refer to the verification code sent to the email, while the `acc_passwd` is shown to us. Since the email obviously didn't exist, there are no way to get the code with the "legit" way (the backend likely didn't even send any email). The only way is to somehow guess the verification code.

4. If you try to login without verifying, a error will be shown saying account is not verified.

Initially I thought maybe `mt_rand` is flawed in some ways that allow us to know the seed given the first output. But I couldn't find anything applicable. I also found something to bruteforce `mt_rand` seed but ignored it since I thought this is a computer security competition, what's the point of bruteforcing?



php_mt_seed - PHP mt_rand() seed cracker

php_mt_seed is a PHP mt_rand() seed cracker. In the most trivial invocation mode, it finds possible seeds given the very first mt_rand() output after possible seeding with mt_srand(). With advanced invocation modes, it is also able to match multiple, non-first, and/or inexact mt_rand() outputs to possible seed values.

PHP's mt_rand() algorithm changed over the years since its introduction in PHP 3.0.6. php_mt_seed 4.0 supports 3 major revisions of the algorithm: PHP 3.0.7 to 5.2.0, PHP 5.2.1 to 7.0.x, and PHP 7.1.0+ (at least up to the latest as of this writing, which is PHP 7.2.0beta3).

php_mt_seed uses attack-optimized reimplementations of PHP's mt_rand() algorithms. It is written in C with optional SIMD intrinsics (SSE2, SSE4.1/AVX, XOP, AVX2, AVX-512, as well as MIC) and OpenMP. On a modern quad-core CPU, it is able to search the full 32-bit seed space in under a minute. On second generation Xeon Phi, it does the same in 3 seconds.

You can view the latest README file, which explains php_mt_seed use cases, provides usage examples, and includes benchmarks on a variety of systems (ranging from quad-core CPU to 16-core server and to Xeon Phi). The README file is also included in the archive below.

Download (release notes, previous release notes):

- php_mt_seed 4.0 and its signature
- php_mt_seed 3.4 and its signature

These and older versions of php_mt_seed are also available from the Openwall file archive. The source code of php_mt_seed can be browsed on GitHub or via CVSweb. (You might find the older versions and revision history useful to better understand how php_mt_seed works and what optimizations have been made.)

Follow this link for information on verifying the signatures.

Why crack mt_rand() seeds?

However out of desperation I decided to try it anyway. I downloaded the archive and compiled it. After reading the README.txt, I found it is pretty

easy to use from the example, just run the binary with the `acc_passwd` given as argument:

```
...
     How to use php_mt_seed.

php_mt_seed should be run from the command line, with command-line
arguments given to it according to the syntax described below.

Usage of php_mt_seed can be trivial or complex, depending on use
case
details.  Here's a trivial usage example:

First generate a "random" number using PHP, e.g. with:

    $ php5 -r 'mt_srand(1234567890); echo mt_rand(), "\n";'
    1328851649

Then run the cracker (in this example, on the same system as we
used for
the build above):

    $ time ./php_mt_seed 1328851649
...
```

So I did it, it took a bit of time:

```
$ time ./php_mt_seed 1900165458
Pattern: EXACT
Version: 3.0.7 to 5.2.0
Found 0, trying 0xfc000000 - 0xffffffff, speed 2042.4 Mseeds/s
Version: 5.2.1+
Found 0, trying 0x20000000 - 0x21ffffff, speed 16.2 Mseeds/s
seed = 0x20b9770f = 549025551 (PHP 7.1.0+)
Found 1, trying 0x78000000 - 0x79ffffff, speed 16.6 Mseeds/s
seed = 0x793568d5 = 2033543381 (PHP 5.2.1 to 7.0.x; HHVM)
seed = 0x793568d5 = 2033543381 (PHP 7.1.0+)
Found 3, trying 0x7c000000 - 0x7dffffff, speed 16.6 Mseeds/s
seed = 0x7d283640 = 2099787328 (PHP 5.2.1 to 7.0.x; HHVM)
seed = 0x7d283640 = 2099787328 (PHP 7.1.0+)
Found 5, trying 0xaa000000 - 0xabffffff, speed 16.6 Mseeds/s
seed = 0xabc1f35b = 2881614683 (PHP 5.2.1 to 7.0.x; HHVM)
Found 6, trying 0xfe000000 - 0xffffffff, speed 16.5 Mseeds/s
Found 6

real    4m21.747s
user    8m27.329s
sys     0m0.260s
```

As you can see, there's only 4 result, and the first one wasn't even 10 digits. So it is not too far fetched to try them one by one by inserting the value into the verification link and loading it in browser:

```
https://mountain.wargames.my/verify.php?username=user3&
verify=[the brute forced result]
```

And get the right link:

Success! Account activated.

Then login with the `acc_passwd` given:

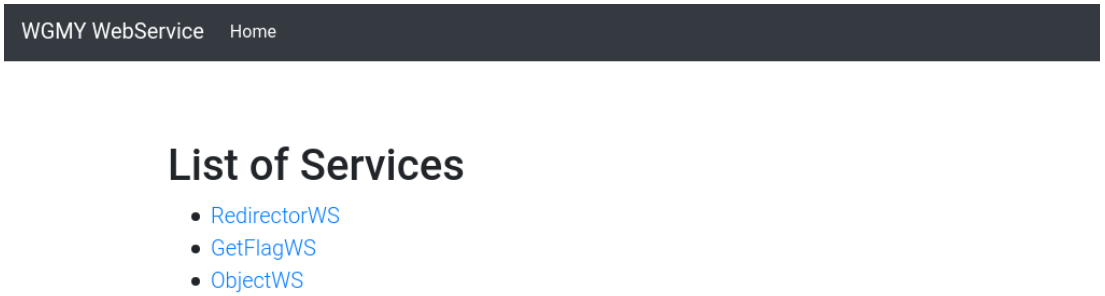
```
Success! Logged in. Keep this somewhere safe.
wgmy{d22772b35b8e80088f41e8662cc3fc81}
```

After reading other participant's writeup, they all seems to just brute force it, so maybe this is the intended solution? I guess the moral of the story is to be careful when using random number generator, sometimes bad things gonna happen if used wrongly.

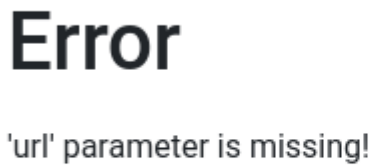
Unsolved challenges

Other challenges that I looked at and had some thoughts but didn't solve.

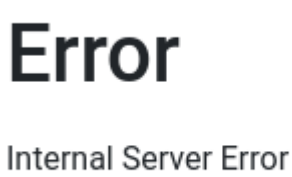
- **Webservice 1**
  - Entering the website looks like this



  - `GetFlagWS` and `ObjectWS` require login. `RedirectorWS` return this:



- The moment I saw the word “redirector” I knew that would be used to redirect us to the flag, so I tried simple thing: `https://[hostname]/ws/unprotected/redirector?url=[hostname]/ws/getflag`



- Ok gave up. (maybe a bit too early, I agree)

- **TryYourLuck**
  - Given the source code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(int argc, char const *argv[])
{
     // Generate number using random seed
     unsigned char seed[4];
     int fd = open("/dev/urandom", O_RDONLY);
     read(fd, seed, 4);
     close(fd);
     srand(*(unsigned int *)seed);

     float money = 1000.00f;
     float bet = 0;
     int number = 0;
     int guess = 0;
```

```
    printf("Try your luck with this simple guessing game!\
n");
    printf("You only have 10 chances to bet and guess\n");
    printf("Get the flag if you win one billion\n");
    printf("Good Luck!!\n\n");
    fflush(stdout);

    for (int i = 0; i < 10; ++i)
    {
    printf("Enter bet: ");
    fflush(stdout);
    scanf("%f",&bet);
    // Check if bet is too less or too high
    if (bet > 499999999 ||  bet < 1)
    {
        printf("Hacker Alert!!\n");
        return 0;
    }
    number = rand() % 0x1337;
    printf("Guess number: ");
    fflush(stdout);
    scanf("%i",&guess);
    if(number == guess){
        money += bet;
        printf("Correct guess!\n");
    }else{
        money -= bet;
        printf("Wrong guess.. money left %.2f\n",money);
    }
    }

    // If money more than 1 billion will print the flag
    if (money < 1000000000){
    printf("See you next time..\n");
    }else{
    char flag[40];
    int fd = open("./flag.txt", O_RDONLY);
    read(fd, flag, 40);
    close(fd);
    printf("Congrats!! Flag is %s\n",flag);
    }
    return 0;
}
```

- I thought maybe something wrong with the RNG again, but things seems ok.
- Maybe buffer overflow or something similar? But I have no idea about how C works as well (sad there's so many things I don't know)
- I can input negative number at `bet` so minus minus equal to plus? Nah didn't pass `bet < 1`.
- Overflowing variable? Does it even work with `float`? Let's try some very big number. Nope.
- Out of idea.

- **hohoho**
  - Given source code again, the way token generated is interesting:

```
...
SECRET = os.urandom(8)

class User:
     def __init__(self, name, token):
     self.name = name
     self.mac = token

     def verifyToken(self):
     realmac = hashlib.md5(SECRET +
self.name.encode(errors="surrogateescape")).hexdigest()
     return self.mac == realmac

def generateToken(name):
     mac = hashlib.md5(SECRET +
name.encode(errors="surrogateescape")).hexdigest()
     return mac
```

- MD5 collision (it is not)? I heard of it before, but never tried it. But even if so, the output likely contain unprintable byte, how am I supposed to feed that into the server?
- Too tired, gave up. The solution is something called length extension. Never heard of it before, so any unlikely gonna be success even if I didn’t give up.

Conclusion

10	PwnStars	1561
11	c3wbis	1550
12	TheNooB	1465
13	d3nsploit	1465
14	Storm	1232

We got 12th place this year. Total of 3 challenges solved excluding forensics (Let's not talk about that). It's been a fun experience and I did learn some new things from reading others' writeup. See you next year?