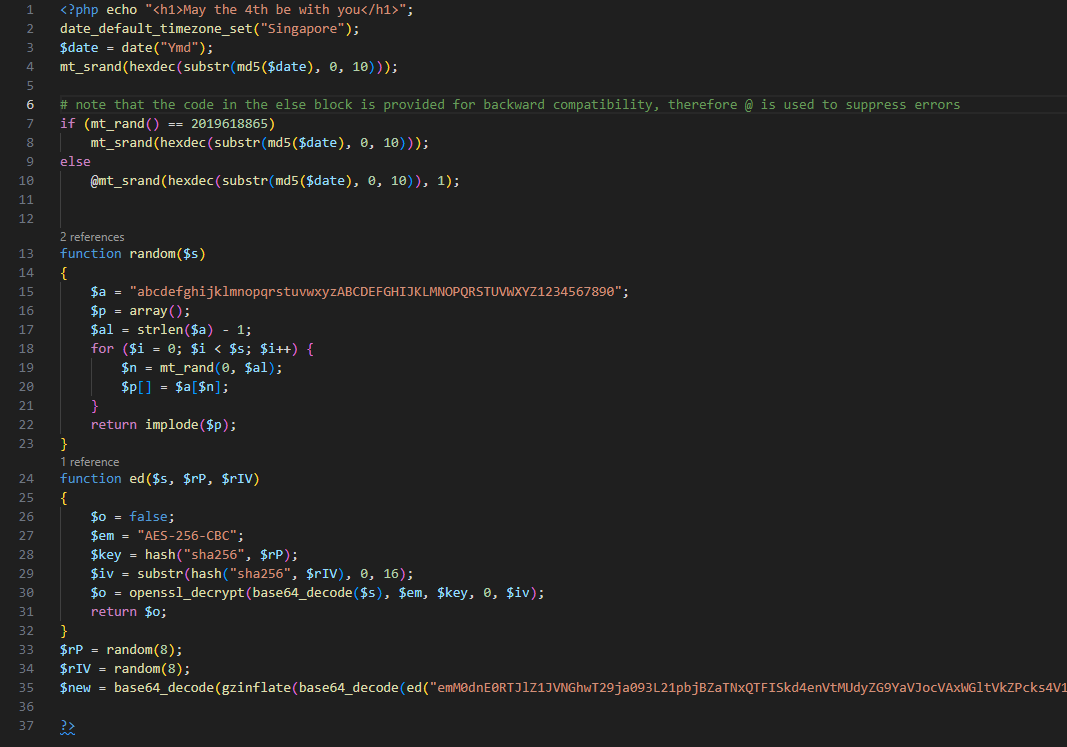
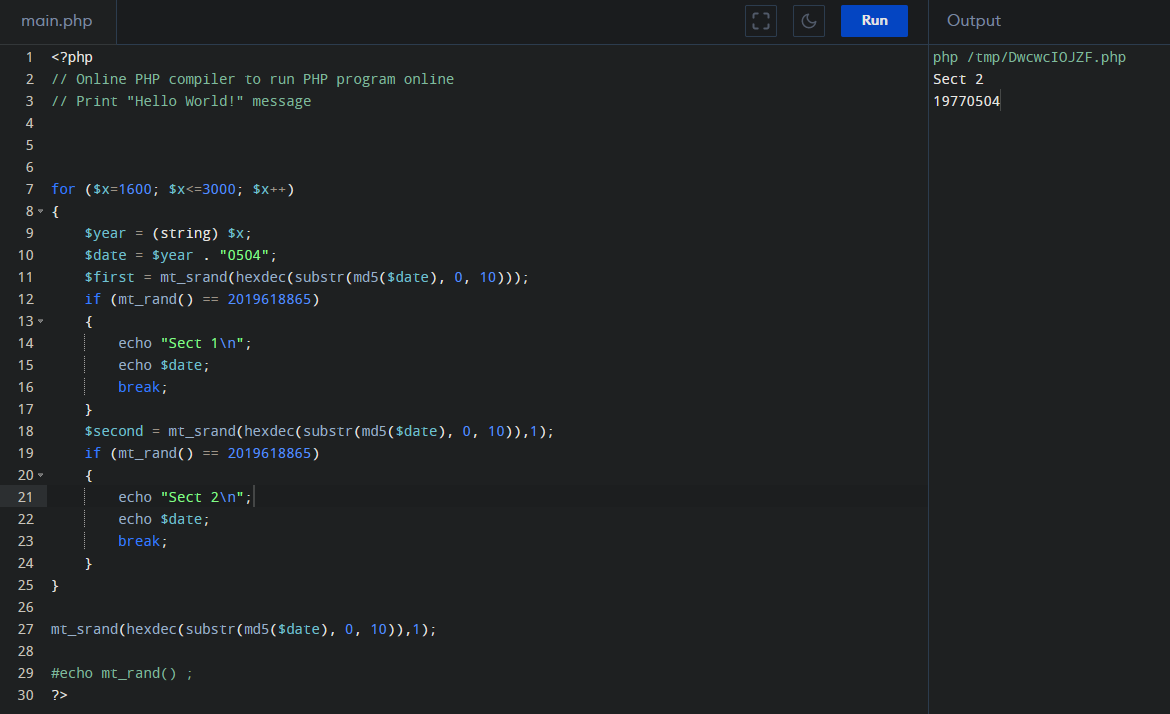
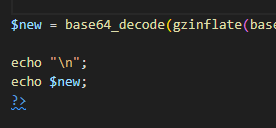
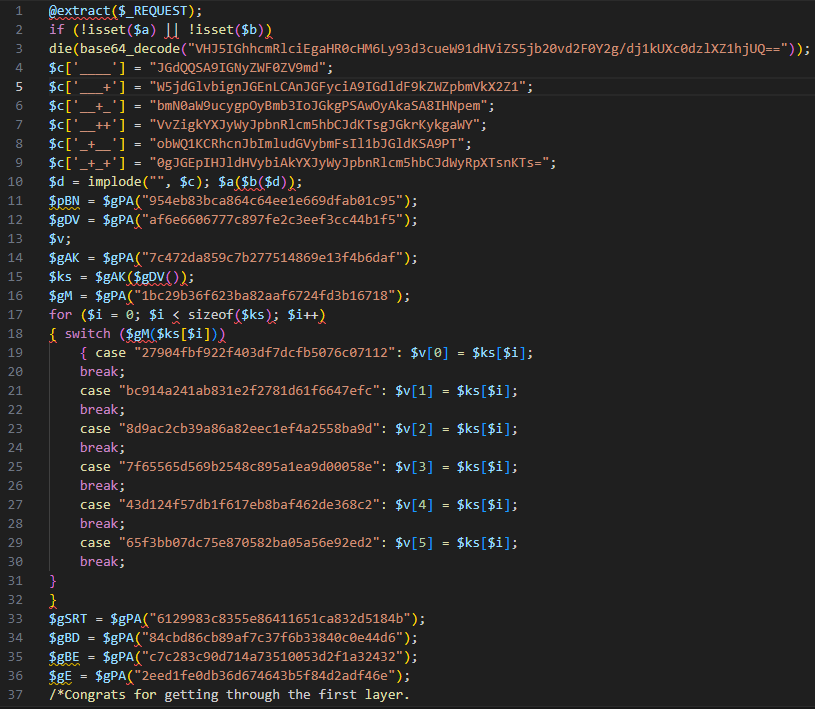
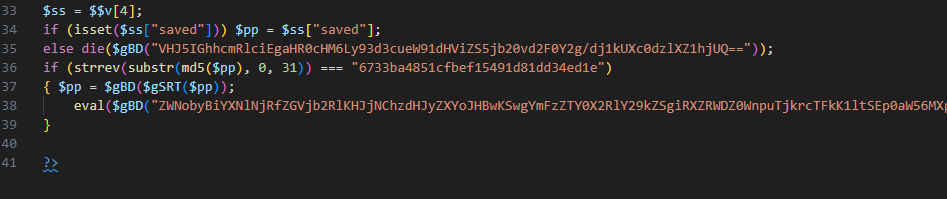
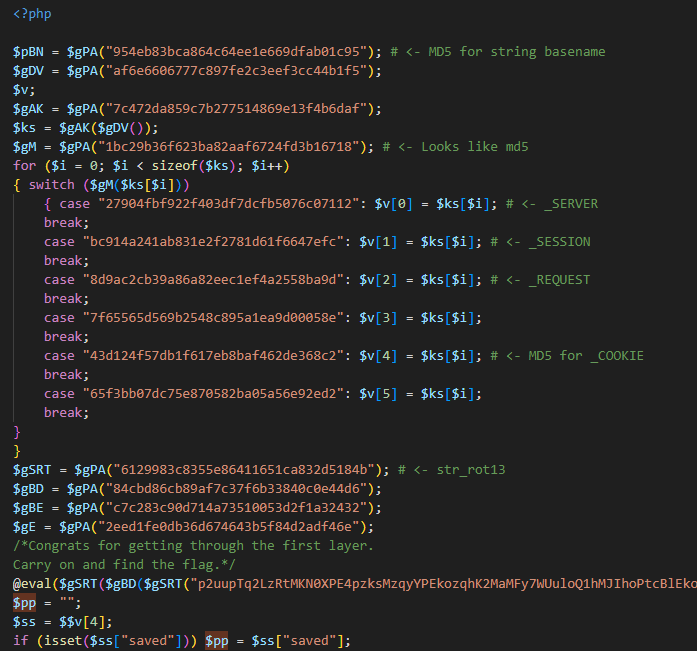
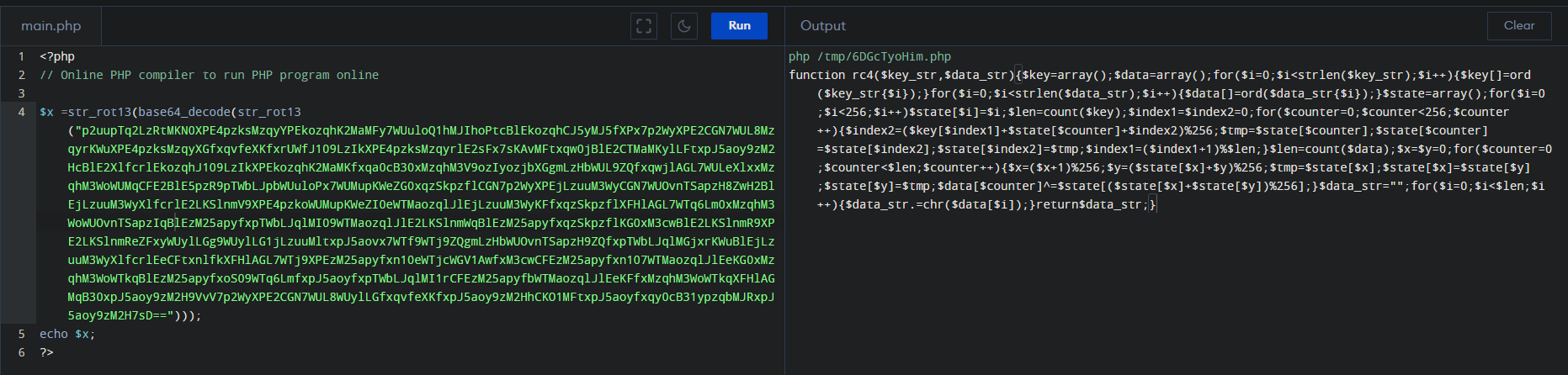
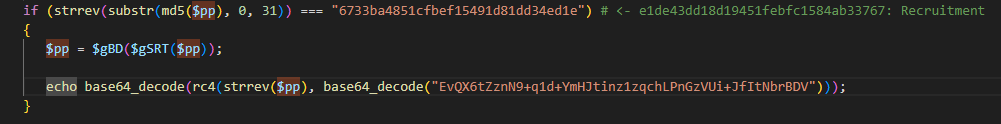
# Challenge 1

B64 and OpenSSL in PHP

1. The PHP contains multiple base64\_decode commands, so the first order would be to simplify the code as much as possible to make it readable.



1. Next, we note the use of mt\_srand, which is a random number generator (rng). Since the seed is provided, the value that appears with each execution is deterministic (fixed). Therefore, we’ll work with that in mind.
2. The use of if-else block is an indication to check if the value obtained from the rng is correct. Looking at the documentation, we note that this is required due to a change in the implementation of the rng algorithm in newer versions.
3. Now, given a clue from the string "<h1>May the 4th be with you</h1>", it is suffice to say that the month and day are already fixed, all that’s left is to guess the year. To do so, we can use a for loop.   
     
   We use the condition mentioned in the original code as a sanity check when performing the loop, and we obtain the year to be “1977”  
     
   
4. Additionally, there is an eval statement on the last line of the file, since we don’t have any idea of what it does, eval is changed to a variable instead. This is so we can analyze the output instead of having it evaluate immediately.  
     
   
5. The result is another PHP script.  
     
   
6. We look at the first if statement, decoding it will give us a link to a Youtube video to Rick Astley’s Never Gonna Give You Up video, very common in cybersecurity 😊.  
     
   
7. Next, we remove lines 2 and 3 (if statement) as it is a hindrance. Looking at line 10, we also note another presence of $a and $b. Looking throughout the code, we have no idea what this does, neither is it assigned to a variable nor is it printed. Additionally, there are no further calls to this variable so it is removed. Lines 4 to 10 are also similar so those are removed.
8. Now we have a problem where we have $gPA but it is not defined anywhere in the code. Looking around, we note on line 35 that it was similar to line 3 in bullet point 6. Therefore, we can assume with reasonable confidence that $gBD is base64\_decode.  
   
9. Line 37 is also particularly noteworthy, as it needs $gSRT to be executed. Now, if we look at all the variables, we note that the assignment always requires $gPA(“<32 characters>”). If we look carefully at the <32 characters> portion, we note that it is in hex, indicative of an MD5 hash. Trying our luck with some of the MD5 hashes with an online MD5 lookup service (https://www.md5online.org/md5-decrypt.htm), we get the names of the functions that we require. In this case, str\_rot13.  
     
   
10. Next we evaluate the @eval statement, and this gives us the rc4 function (required in step 13)  
      
    
11. We turn our attention to line 36, where we have a 31 character MD5 string. Since MD5 is thirty-two characters long, then there are only 16 possibilities to test. Doing so we get the MD5 hash and we get the corresponding plaintext “e1de43dd18d19451febfc1584ab33767: Recruitment”
12. We also fix up the last statement by converting all the base64 into something more readable  
      
    
13. Essentially the code is just the following after cleaning up. (Note that PHP8 will output errors so PHP7 needs to be used instead)  
      
    <?php

function rc4($key\_str, $data\_str)

{

    $key = array();

    $data = array();

    for ($i = 0; $i < strlen($key\_str); $i++) {

        $key[] = ord($key\_str { $i});

    }

    for ($i = 0; $i < strlen($data\_str); $i++) {

        $data[] = ord($data\_str { $i});

    }

    $state = array();

    for ($i = 0; $i < 256; $i++)

        $state[$i] = $i;

    $len = count($key);

    $index1 = $index2 = 0;

    for ($counter = 0; $counter < 256; $counter++) {

        $index2 = ($key[$index1] + $state[$counter] + $index2) % 256;

        $tmp = $state[$counter];

        $state[$counter] = $state[$index2];

        $state[$index2] = $tmp;

        $index1 = ($index1 + 1) % $len;

    }

    $len = count($data);

    $x = $y = 0;

    for ($counter = 0; $counter < $len; $counter++) {

        $x = ($x + 1) % 256;

        $y = ($state[$x] + $y) % 256;

        $tmp = $state[$x];

        $state[$x] = $state[$y];

        $state[$y] = $tmp;

        $data[$counter] ^= $state[($state[$x] + $state[$y]) % 256];

    }

    $data\_str = "";

    for ($i = 0; $i < $len; $i++) {

        $data\_str .= chr($data[$i]);

    }

    return $data\_str;

}

$pp = "Recruitment";

if (strrev(substr(md5($pp), 0, 31)) === "6733ba4851cfbef15491d81dd34ed1e") # <-e1de43dd18d19451febfc1584ab33767: Recruitment

{

   $pp = base64\_decode(str\_rot13($pp));

   echo base64\_decode(rc4(strrev($pp), base64\_decode("EvQX6tZznN9+q1d+YmHJtinz1zqchLPnGzVUi+JfItNbrBDV")));

}

?>

1. This gives us the following when executed.  
   FLAG{My\_F15st\_b@By\_St3ps}

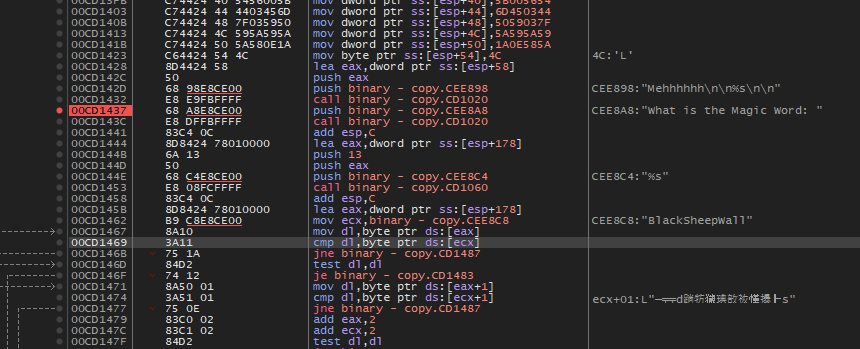
# Challenge 2

Debugging of Terminal application

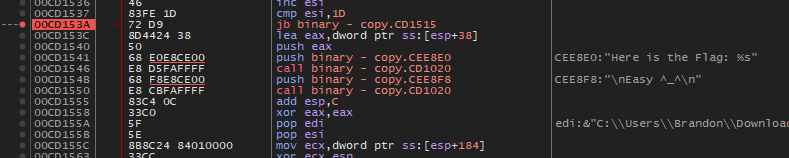
1. Opening up the program, we see the following image



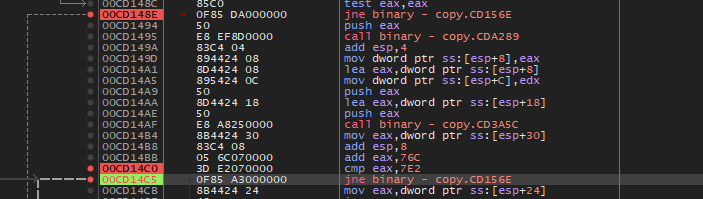
1. Typing any word terminates the application so we use a debugger to analyse the program, in this case x32dbg.
2. Using the string as a keyword, we locate the corresponding string in asm.  
   We also further notice that there is a cmp and ecx instruction there, therefore the magic word should be “BlackSheepWall”.



1. We find a line downstream that outputs the flag in the code



1. Above that, there are several cmp and jne instructions along the way, so the easiest method would either be to change the data contained in the registers used for comparison, or change the ZF flag to 1 to bypass the jne checks, that way the pointer will not jump to the specified address, but proceed down to the next address.

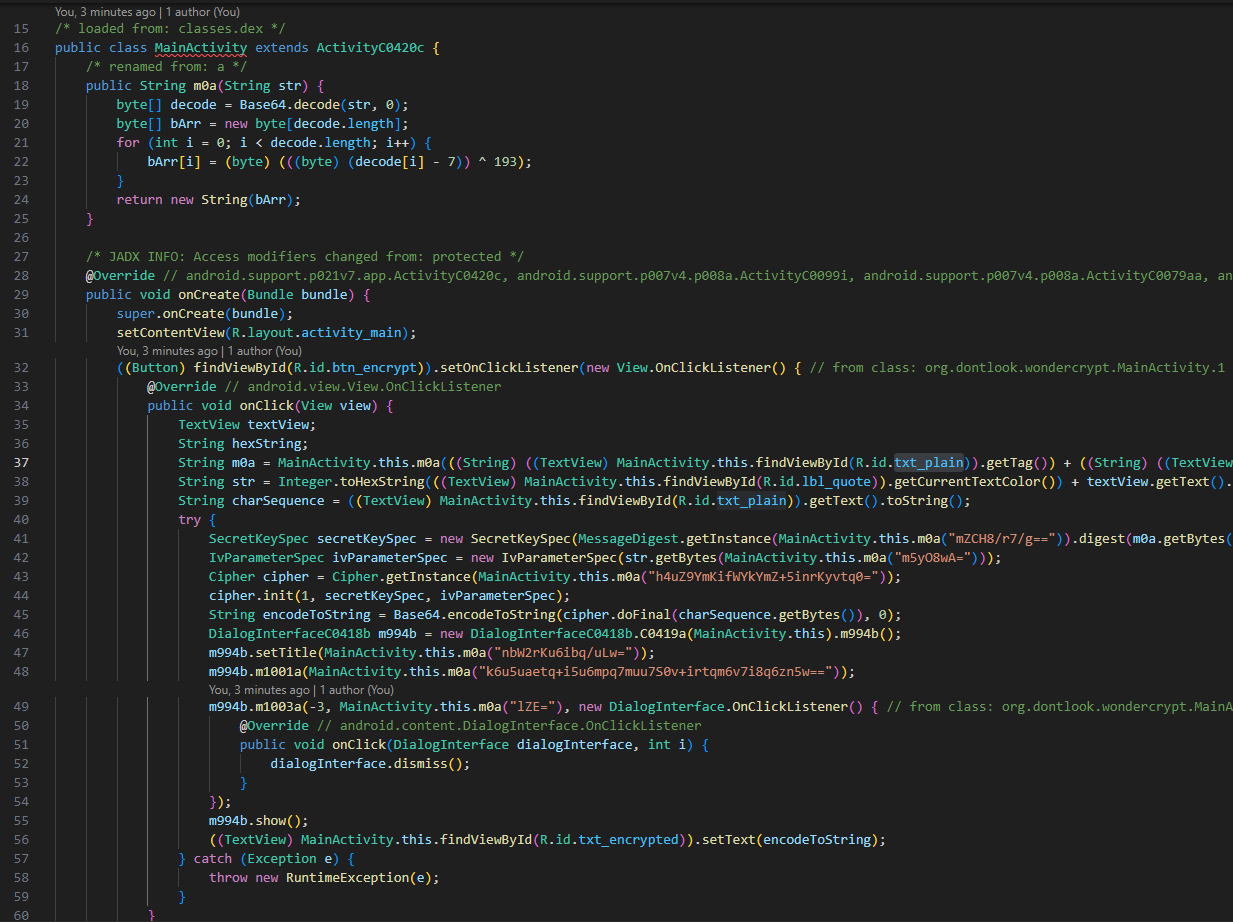
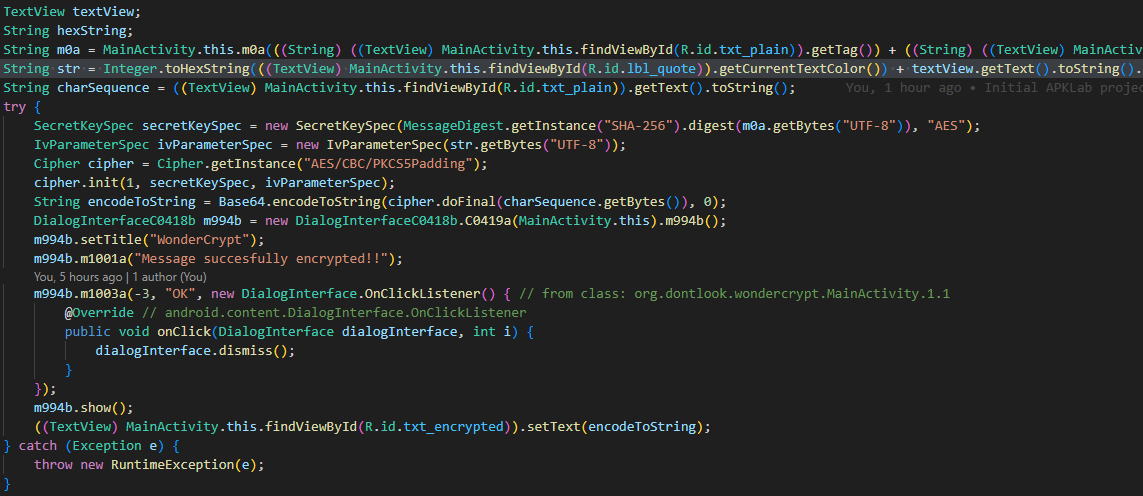


1. Bypassing those cmp and jne checks, we get the flag printed onto the screen. (FLBKx!@1dd4iu6w]M7kalhihnj?/~)



# Challenge 3

Wondercrypt APK

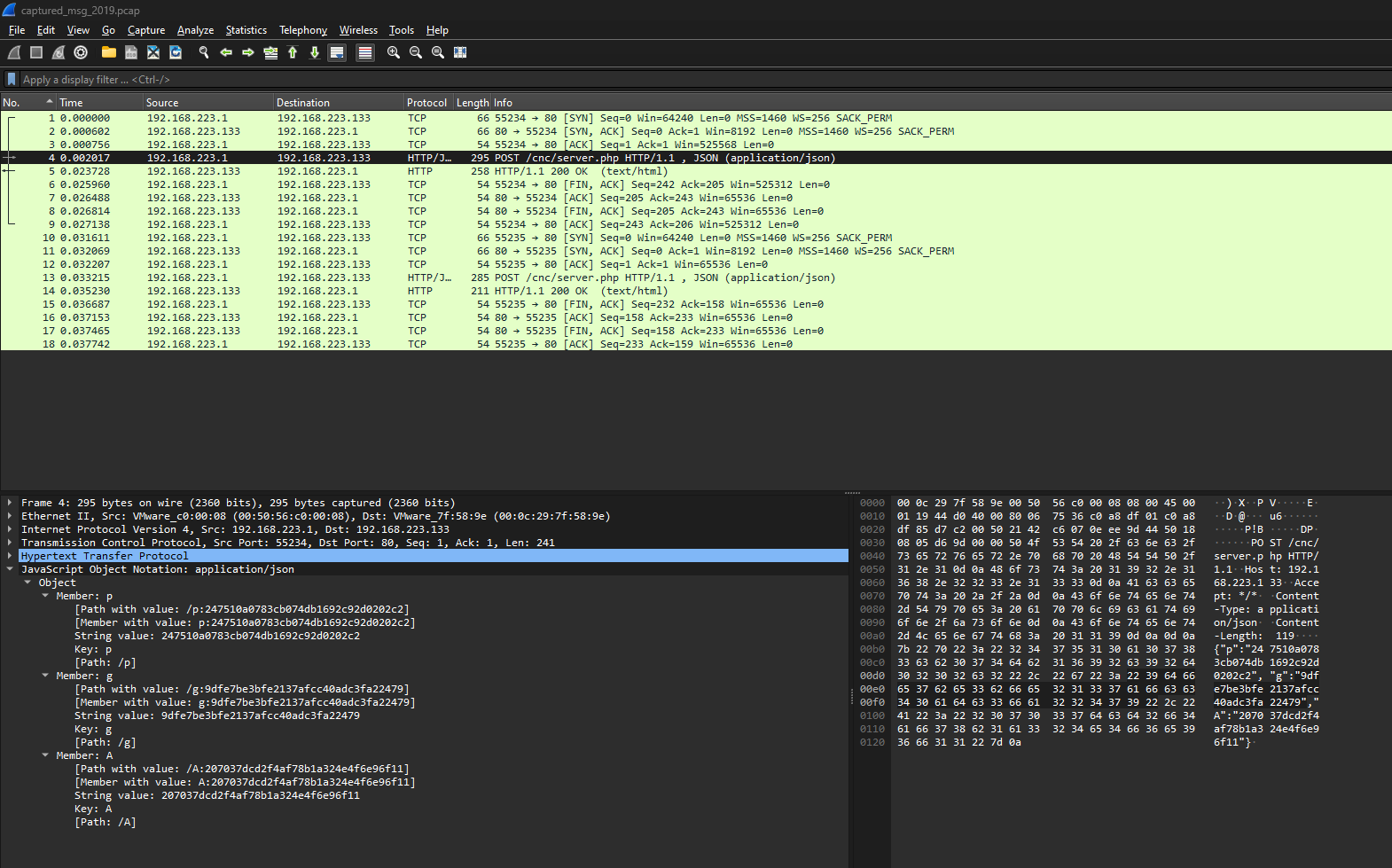
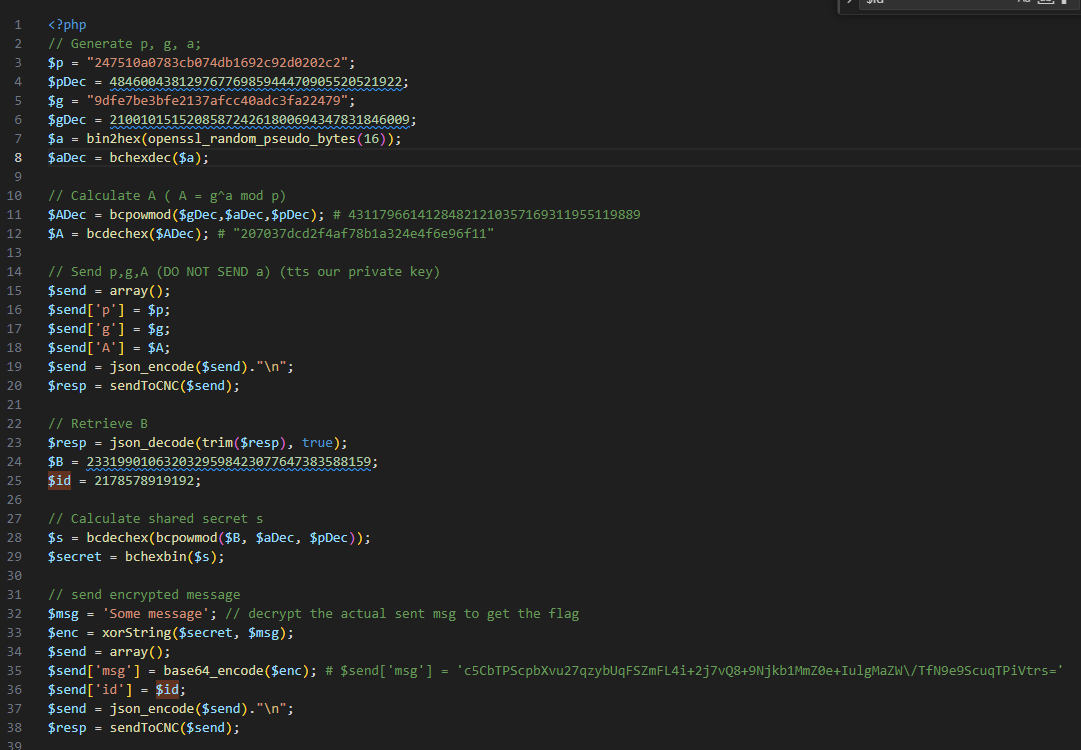
1. We first need to find a way to decompile the APK, and we can do so using the APKLab extension in VSCode. This gives us some form of readable source code that we can work with.  
     
   
2. There are a few obfuscated string, but thankfully the helper function is also included in the file so after decoding the strings we find that the secret key uses AES/SHA-256 digest, and the cipher that we’re using is AES/CBC/PKCS5Padding.
3. However, the current issue is that the app will not run under Android Studio (even with the target SDK and emulator matching the level mentioned in the AndroidManifest.xml file). Specifically, the app will crash when open, so there are a few things that we would be unable to get from the code and if the app doesn’t run. That is, the colour of the text and text from textview.  
     
   

4. Otherwise, the string can be decrypted by using the following steps

1. Perform a base64 decode on the string
2. Use the same cipher but change the parameter “1” in cipher.init to 2 (decrypt)

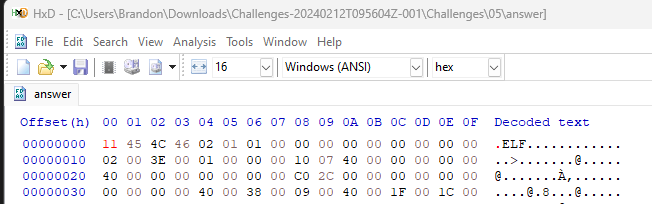
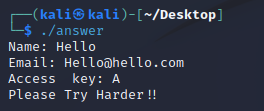
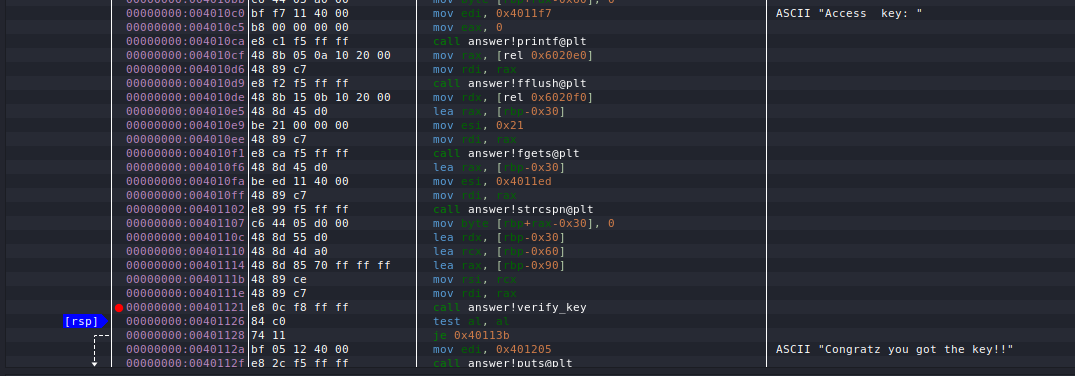
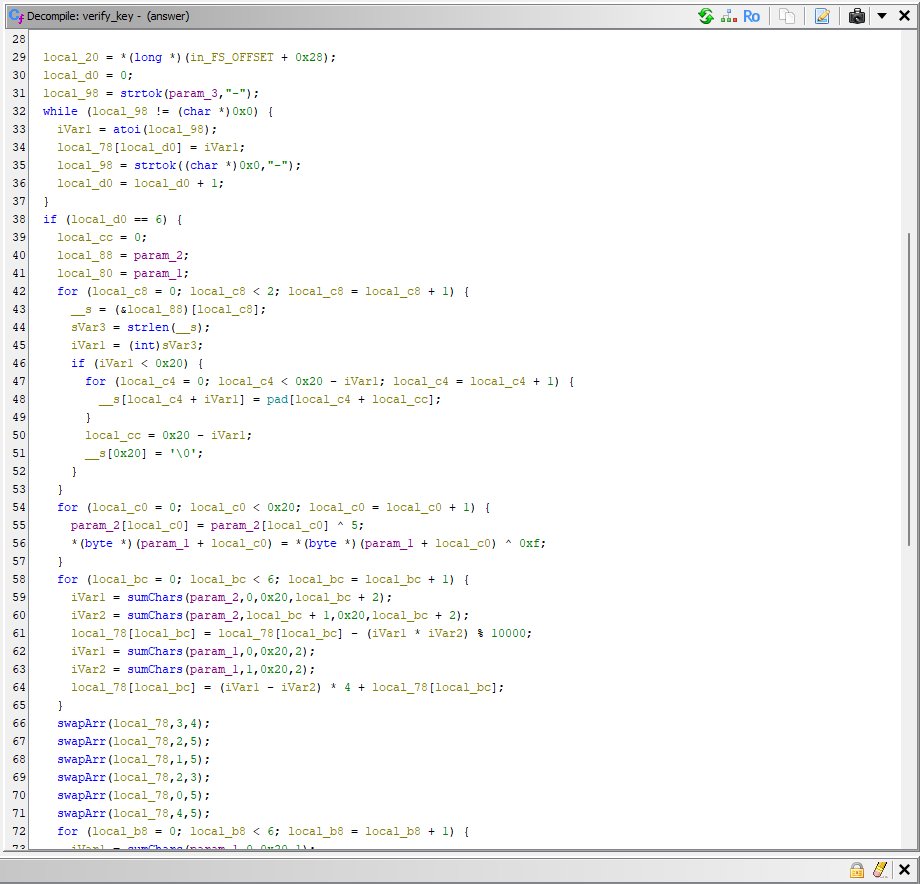
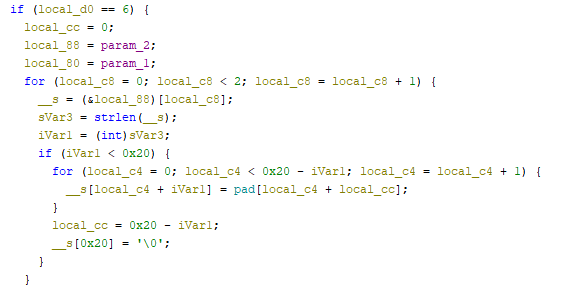
# Challenge 4

OpenSSL Keys

1. Given a php file and a pcap capture file, we open the php file first and we find that we are dealing with openssl and encryption techniques. We also find that variables $p, $g and $A are sent, and other variables such as $msg, $B are received.
2. We look at the pcap file, and we find some of the information contained within POST and subsequent GET requests.  
     
   
3. We fill up some of the information with what is obtained to make it easier to read.  
     
   
4. We also note that the type of encryption used is asymmetric key encryption. User generates using 3 random numbers that are 16 bytes long, with the formula
5. In addition to point 4, the shared secret is calculated using , where B is obtained from the server. This is because of the following equations  
   , A is sent to the server  
   , B is sent to the user  
     
   The shared secret calculated from the user would be   
   The shared secret calculated from the server would be
6. Due to the fact that $a is not known, and we are dealing with large numbers, obtaining the inverse would be difficult as multiplication operations are performed over a finite field. (Essentially, multiplying g to $A multiple times such that , where is the inverse of $a.) Given that we don’t have any additional information in the pcap file to go by, brute force would be the only option, but due to the fact that there are possible combinations, it will take a considerable amount of time to obtain the key.
7. However, if a key is obtained, then the process of getting the message is relatively easy. All that is required would be to decode the message that was sent, and XORing with the secret to obtain the flag (Since 2 applications of XOR would give you the original message).
8. Alternatively, since brute forcing will be computationally heavy and can potentially take really long, an alternative method is to send a known message. This does not require $a to be known. The reason for this is as follows  
     
   a. We know that a message was sent back to us with the secret being XOR’ed to it (i.e. )  
     
   b. Using a known message , we know that the message would eventually be XOR’ed with (i.e. )  
     
   c. Using these two messages, we can obtain without knowing the secret. (We can also apply the same technique to obtain the secret, and this helps us to bypass knowing what is $a)  
     
   We take both encrypted messages and apply the XOR operator  
    sent using this shared secret without having to repeat all these steps again.

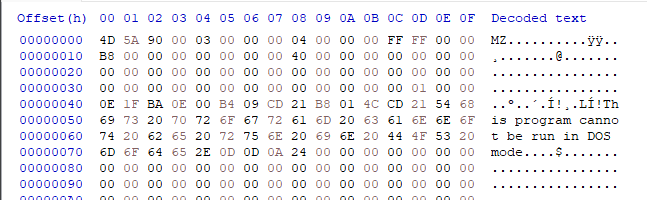
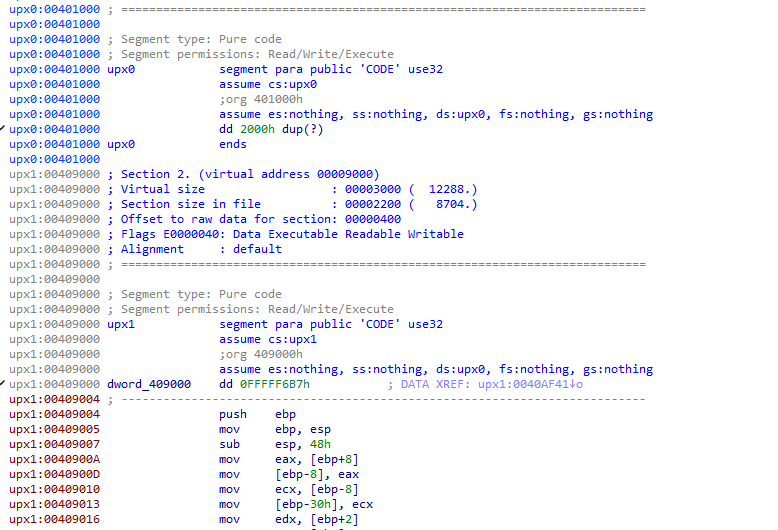
# Challenge 5

ELF File

1. Looking at the provided file, it does not have an extension. We first determine what type of file by using a hex editor and attempt to look for the magic header. In this case, we see ELF, indicative of a Linux executable.  
     
   
2. We transfer this executable onto Linux and make it executable using “chmod +x ./answer”
3. Next, we attempt to see what the application does by running it once. We see it requires 3 inputs, a name, an email and an access key.  
     
   
4. Putting the executable through edb debugger, we note that there is a function called verify\_key before a success/failure message is printed on the screen, indicating that there is a function to check the key.  
     
   
5. As everything is in assembly and hard to read, we use Ghidra to decompile the program. As we know the name of the function, it will be easier to locate the region to look at and to view its corresponding pseudocode. As the pseudocode does not reflect the original source code, it can be difficult to figure out what the code does. Additionally, we also see several other functions being called, such as strlen, strtok, atoi from the stdlib library as well as other functions such as sumChars and swapArr, and these are called multiple times in the subroutine.  
     
   
6. The difficulty of this is figuring out which variable is referring to what data, and will take a significant amount of time to clean, to get better code readability. Additionally, because the pseudocode is primarily shown in C, understanding whether a pointer is being referenced will also take additional time to analyse.  
   1. Some things worth noting are that the required key may be of length 6, as there are some portions of the code where a function is called 6 times in sequence or 6 is referenced as a check.  
      
7. Due to time constraints, the algorithm to obtain the key was not fixed in time. However, understanding the 3 functions, specifically verify\_key, sumChars and swapArr would help in deriving a key for any combination of name and email address.

# Challenge 7

UPX Executable

1. Looking at the file to be analysed, it does not have a file extension, so we analyse the first few bytes of the executable and look for a magic header. In this case, this is a Windows exe file. Further analysis shows that this is a 32-bit console based application.  
     
   
2. Executing the program, we get nothing than a blank screen. Opening it up in IDA, we immediately notice upx0, upx1 header names. This indicates the use of the UPX packer for this application, to obfuscate code.  
     
   
3. To perform analyses, we would have to obtain the unpacked executable. There are several guides on how to do so using debuggers
   1. [How to unpack UPX packed malware with a SINGLE breakpoint | by Saket Upadhyay | InfoSec Write-ups (infosecwriteups.com)](https://infosecwriteups.com/how-to-unpack-upx-packed-malware-with-a-single-breakpoint-4d3a23e21332)
   2. [Manual Unpacking of UPX Packed Binary File - www.SecurityXploded.com](https://securityxploded.com/unpackingupx.php)

However, these were unsuccessful when attempting to obtain the unpacked executable.

1. The general flow would be to decompile/debug the unpacked executable, and perform analysis similar to challenge 2 and 5 to obtain the email. Alternatively, one could also grab all readable strings from the binary to determine if the email is hardcoded into the binary.

# Challenge 8

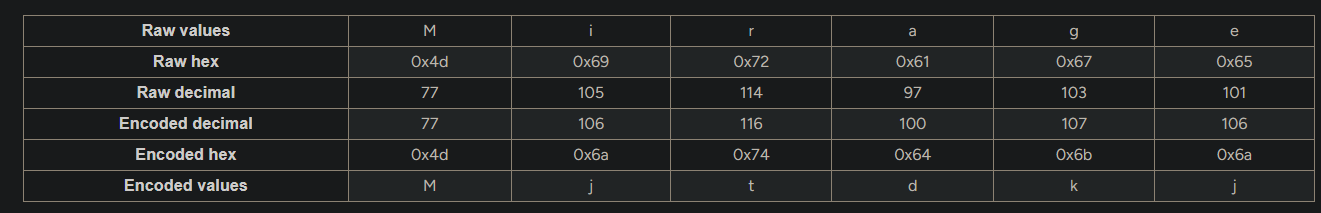
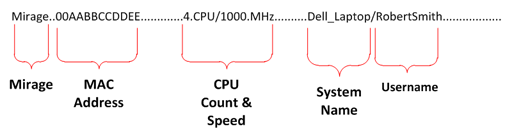
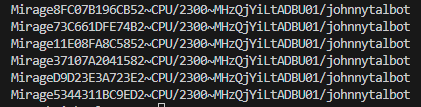
# Challenge 9

Mirage pcap analysis

Reading through the Secureworks blog, we get a rough understanding of how the payload looks like.

Looking through the PCAP file, we note 6 different payloads across 500 HTTP streams. (\*)

Answering the questions in order

1. The victim’s computer name is QjYiLtADBU01 and the username is johnnytalbot. The data is encoded in the payload itself, however the encoding adds a value based on the offset position.  
     
     
   Therefore, to obtain the raw data we would do the opposite (subtraction).  
     
     
   Looking at this image from the blog, we can effectively deduce the components of the payload  
     
     
     
   This gives us the computer name and username of the computer that was seen beaconing  
     
     
     
   Note that invalid values such as “\x00” is omitted by Python when the values are printed out but strings have been checked when it was still in a list form.
2. Using the same technique as bullet point 1, we see 6 unique MAC addresses. These are
   1. 8F:C0:7B:19:6C:B5
   2. 73:C6:61:DF:E7:4B
   3. 11:E0:8F:A8:C5:85
   4. 37:10:7A:20:41:58
   5. D9:D2:3E:3A:72:3E
   6. 53:44:31:1B:C9:ED
3. We note that in the PCAP file, the POST requests are sent to newwork[.]dyndns[.]org. DynDNS is a service that allows users to map a domain to a dynamic IP address. It **used to** be available for free, so there could be a possibility that the user hosting the Mirage C2 server may not be traceable.   
     
   Secondly, we do not see any GET requests from the PCAP file, therefore this is not the second variant that was identified by SecureWorks.
4. It is probable that only 1 host has been infected. The reasons for this are
   1. Assuming that the computer is not part of the domain, then during the installation of Windows, the default name of the computer is randomly generated. Therefore, it is highly unlikely that 2 or more computers on the same network would have identical names.
   2. Assuming that the computer is part of the domain, then it will also not be possible as Active Directory would not allow it. This implies that the <computer name>/<username> combination must be unique in the domain.
   3. There is only one source MAC address from which the packets are sent from (the infected computer). Again, it is not probable that the network interfaces would have the same MAC address on the same network. Even if a software such as VMWare is used, whereby the MAC address can be manually altered, this may result in loss of packets and as such is not ideal.
5. The most probable event that had happened from the analysis of the PCAP data is that Mirage is enumerating the network interfaces on the infected machine. This could be an attempt at uncovering networks that are internal facing and have no access to the wider internet, and may be an avenue for pivoting across the internal network.