# Project 2 - Block It Up!

## CSCE 713 Software Security

Peiwen Wang (128001139)

peiwen.wang@tamu.edu

Jingdi Zhang (825007281)

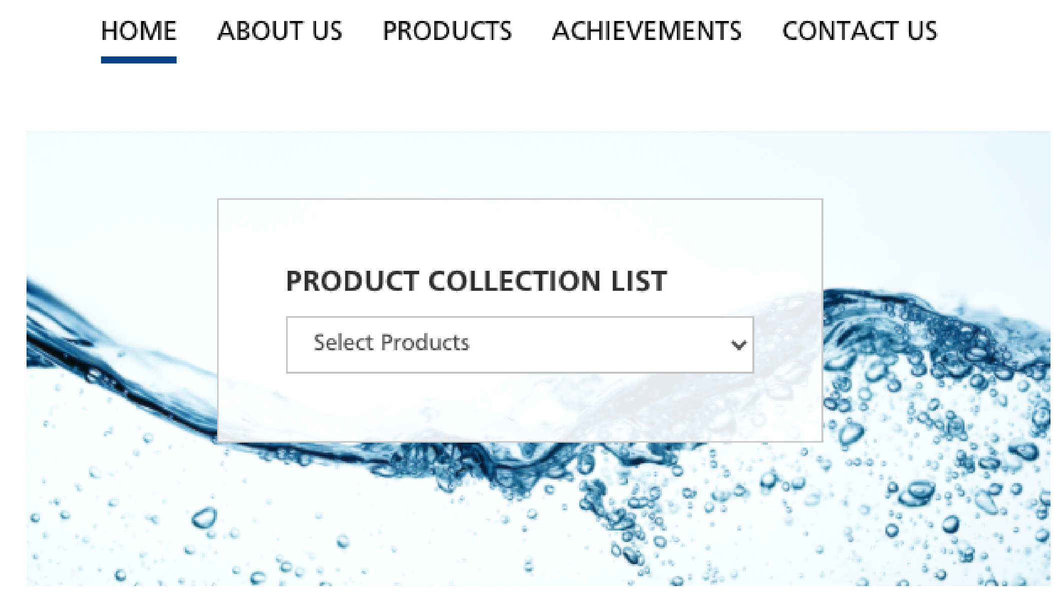
cindy1024i@tamu.edu

# Program 1  Sparrow (C)

https://github.com/codercheng/sparrow

## Introduction

Sparrow is a HTTP web server, so we fed a website in to demonstrate the vulnerability reproduction and mitigation. The running server which hosts the websites shown as below.




## Vulnerability 1.1

=> Vulnerability description: **Buffer overflow**. The developer used the **sprintf** to store the directory information into the buffer with 512 B. But the developer did not check if the length of the directory entries exceeds 512 B or not. The attacker can make use of this to corrupt the next buffer or program code to run other evil programs.
=> Vulnerability categories: **Spatial Memory Attacks**
=> Vulnerability source code snippet

The vulnerability is located in Line 153 or Line 155 in file.c

```
Line 106          char newpath[512];
Line 147          while ((temp_path = readdir(dir)) != NULL) {

                  if (!strcmp(temp_path->d_name, ".") || !strcmp(temp_path->d_name, "..") ||
!strcmp(temp_path->d_name, ".res"))
```

```
                         continue;

                 if (path[strlen(path) - 1] == '/')
Line 153              sprintf(newpath, "%s%s", path, temp_path->d_name);
                 else
Line 155              sprintf(newpath, "%s/%s", path, temp_path->d_name);

                 lstat(newpath, &s);
```

To run through the vulnerability in file.c, we have to run through the else if branch in line 617 of sparrow.c file, as below. More specifically, it’s in the process_dir_html function.

sparrow.c file

```
602             ev_stop(loop, sockfd, EV_WRITE);
603             if (fd_records[sockfd].http_code != DIR_CODE) {
604                 int ret = ev_register(loop, sockfd, EV_WRITE, write_http_body);
605                 int r = process_dir_html(fd_records[sockfd].path, sockfd);
606                 if (ret == -1) {
607                     if (conf.log_enable) {
608                         log_error("ev register err\n");
609                     }
610                     else {
611                         fprintf(stderr, "ev register err in write_http_header1()\n");
612                     }
613                     delete_timer(loop, sockfd);
614                     return NULL;
615                 }
616             }
617             else if (fd_records[sockfd].http_code == DIR_CODE) {
618
619                 int r = process_dir_html(fd_records[sockfd].path, sockfd);
```

=> Vulnerability Reproduce:

a. However, we discovered that this branch never got called, so we manually moved the function into the above branch. The picture below shows the position of actual vulnerability

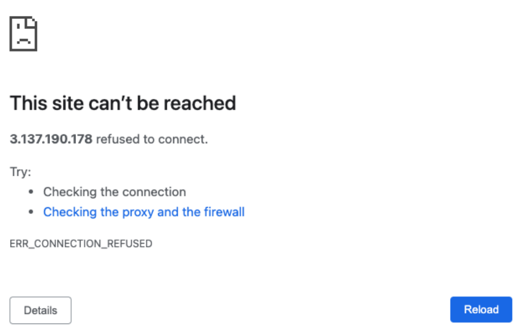
```
/www/CSS/Font/Frutiger.woff./www/CSS/Font/Frutiger.woff========dir_html_maker======hhhhhhh===============
/www/CSS/Font/Frutiger.wofftestsssssstemp_pathtemp_pathtemp_pathtemp_path not iftemp_pathtemp_path not iftemp_pathtemp_p
ath not iftemp_pathtemp_path not iftemp_pathtemp_path not iftemp_pathtemp_path not if./www/404.html==============ppppp==
=============
/www/404.html./www/404.html========dir_html_maker======hhhhhhh===============
```

b. Inside the else branch, we assign the path to a string with larger size. Then the program crashes, and the web pages cannot be loaded.

```c
        if (path[strlen(path) - 1] == '/'){
            // printf ("temp_path if");
            // printf("%s",path);
            sprintf(newpath, "%s%s", path, temp_path->d_name);
        }
        else{
            printf ("temp_path not if");
            printf("%s",path);
            path="sdfsdfefwasd-wastewater-treatment-products-wastewater-trea
            sprintf(newpath, "%s/%s", path, temp_path->d_name);
        }

        lstat(newpath, &s);
```

The website crashes, and the memory block after the “newpath” buffer has been overwritten and the program is not able to run on the next time.

```
ssssddddddddddddddddddddddddddddddddddddddsaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaddddddddxxxxxxxxxxxxxxxqqqqqqqqqqqqqq
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
qqqqqqqqqqqqqqqqqqqqqqqqjnkjnjtemp_pathtemp_path not ifsdfsdfefwasd-wastewater-treatment-products-wastewater-treatment-p
roducts-wastewater-treatment-products-wastewater-treatment-products-wastewater-treatment-products-wastewater-treatment-p
roducts-wastewater-treatmasdasadasdxasxasdssssssssssssssssssssssdddddddd*** stack smashing detected ***: <unknown> termi
nated
Aborted (core dumped)
ccc@DESKTOP-VC8V910:/mnt/c/Users/zhang/Documents/sparrow$
```




=> Mitigation tool: AddressSantizer

=> Mitigation process:

Add the **-fsanitize=address** to the compiler flags and linker flags in the makefile.

```makefile
BIN := sparrow
OBJS := sparrow.o thread_manage.o file.o ev_loop.o config.o
async_log.o url.o min_heap.o cJSON.o picohttpparser.o
CC := gcc
DEBUG := -g -Wall
CFLAGS := -fsanitize=address -Wall -c $(DEBUG)
LFLAGS :=  -fsanitize=address -pthread -lrt -lm
```

=> Demonstration of Mitigation:: Use addressSantizer to find the error and gcc enables canary. These two checks disable the buffer overflow attack, and our program has been protected.

```
==279==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffb0406fba0 at pc 0x7ffb0ccde8f9 bp 0x7ffb0406f700
p 0x7ffb0406ee90
WRITE of size 526 at 0x7ffb0406fba0 thread T3
    #0 0x7ffb0ccde8f8 in __interceptor_vsprintf (/usr/lib/x86_64-linux-gnu/libasan.so.4+0x9e8f8)
    #1 0x7ffb0ccdec86 in __interceptor_sprintf (/usr/lib/x86_64-linux-gnu/libasan.so.4+0x9ec86)
    #2 0x7ffb0e00d037 in dir_html_maker /mnt/c/Users/zhang/Documents/sparrow/file.c:169
    #3 0x7ffb0e00b543 in process_dir_html /mnt/c/Users/zhang/Documents/sparrow/sparrow.c:700
    #4 0x7ffb0e00ac65 in write_http_header /mnt/c/Users/zhang/Documents/sparrow/sparrow.c:608
    #5 0x7ffb0e00f3bf in ev_run_loop /mnt/c/Users/zhang/Documents/sparrow/ev_loop.c:222
    #6 0x7ffb0e00bf16 in worker_threads_entrance /mnt/c/Users/zhang/Documents/sparrow/thread_manage.c:27
    #7 0x7ffb0c6876da in start_thread (/lib/x86_64-linux-gnu/libpthread.so.0+0x76da)
    #8 0x7ffb0c3a171e in __clone (/lib/x86_64-linux-gnu/libc.so.6+0x12171e)
```

# Vulnerability 1.2

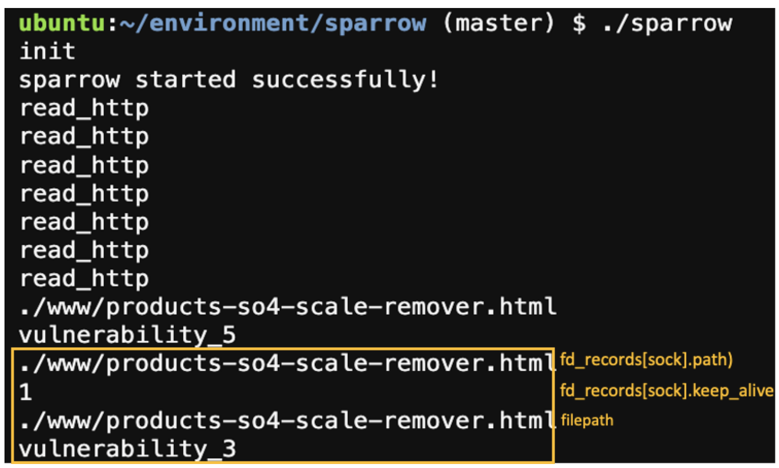
=> Vulnerability description: **Buffer overflow**. The developer used **strcpy** to copy the filename to the member of the struct fd_record_t, but didn't make sure the size of the filename is short enough to be stored to the destination to prevent overflow.
=> Vulnerability categories: **Spatial Memory Attacks**
=> Vulnerability source code snippet
The vulnerability is located in Line 463 in sparrow.c file.

```
Line 355    char filename[1024 + 1 + strlen(work_dir)];//full path
...
Line 454    if (fd_records[sock].http_code != 304) {
                        ...
Line 463     strcpy(fd_records[sock].path, filename);

        }
        // fd_record_t is defined as below in the ev_loop.h
        typedef struct {
                int active;

                EV_TYPE events;
                cb_func_t cb_read;
                cb_func_t cb_write;

                int ffd;
                unsigned int write_pos;
                unsigned int read_pos;
                unsigned int total_len;
                char buf[MAXBUFSIZE];
                int http_code;
                char path[128];
                int keep_alive;
                void* timer_ptr;
        } fd_record_t;
```

=> Vulnerability reproduce:

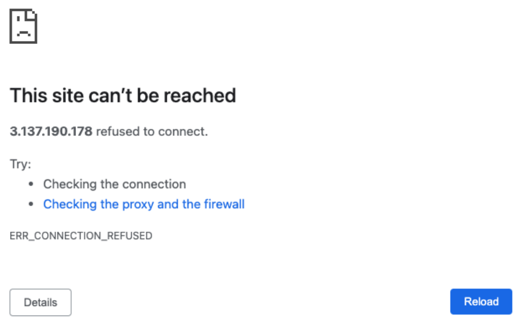
a. Firstly, we inserted the code to fprintf to print the fd_records[sock].path). And the below screenshot shows that the code actually executed the vulnerability part.

```
/*** CODE USED TO CHECK THE PATH AND fd_records ***/
     fprintf(stdout, "%s\n", fd_records[sock].path);
    char text[10];
    sprintf(text, "%d",fd_records[sock].keep_alive);
    fprintf(stdout, text);
    fprintf(stdout, "\n");
    fprintf(stdout, "%s\n", filename);
    fprintf(stdout, "vulnerability_3\n");
 /*** CODE END ***/
```




b. The program failed to check the length of filename before passing it in the strcpy function. So we create a filename with a longer name consisting 256 characters. ./www/products-wastewater-treatment-products-wastewater-treatment-products-wastewater-treatment-products-wastewater-treatment-products-wastewater-treatment-products-wastewater-treatment-products-wastewater-treatment-products-wastewater-treatment-products-wastewater-treatment-product.html.

c. The website crashes, and the memory block after the “fd_records[sock].path” buffer has been overwritten and the program is not able to run on the next time.




The directory is also included so it's possible to exceed the length limit 256. Before using strcpy, the fd_records[sock].path is "./www", after use strcpy, even the max filename is smaller than the 255 characters, but if we append together, the length exceeds 256 characters. The results turn out that the web server crashed.

=> Mitigation: AddressSantizer
=> Demonstration of Mitigation: AddressSantizer then discovered this error and the program aborted, and our program has been protected.

```
./www/products-wastewater-treatment-products-wastewater-treatment-products-wastewater-treatment-products-wastewater-treatment-products-wastewater-t
ment-products-wastewater-treatment-products-wastewater-treatment-product-product.html
ASAN:DEADLYSIGNAL
=================================================================
==5735==ERROR: AddressSanitizer: SEGV on unknown address 0x6c6d74682e94 (pc 0x55918b226a6e bp 0x7f57bf39dcb0 sp 0x7f57bf39dbe0 T2)
==5735==The signal is caused by a WRITE memory access.
    #0 0x55918b226a6d in write_http_body /home/ubuntu/environment/sparrow/sparrow.c:740
    #1 0x55918b22a10e in ev_run_loop /home/ubuntu/environment/sparrow/ev_loop.c:221
    #2 0x55918b226d11 in worker_threads_entrance /home/ubuntu/environment/sparrow/thread_manage.c:27
    #3 0x7f57c71036da in start_thread (/lib/x86_64-linux-gnu/libpthread.so.0+0x76da)
    #4 0x7f57c6e2c71e in __clone (/lib/x86_64-linux-gnu/libc.so.6+0x12171e)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV /home/ubuntu/environment/sparrow/sparrow.c:740 in write_http_body
==5735==ABORTING
```

# Vulnerability 1.3

=> Vulnerability description: **time-to-check-to-time-to-use**. Before opening the file, an attacker can use a symbolic link to track the software to open another "secret" file which will lead to the information leak.
=> Categories: **Concurrency Attacks**
=> Vulnerability source code snippet
The vulnerability is located in Line 431 in sparrow.c file.

```
            if (fd_records[sock].http_code != 304) {
Line 431            fd = open(filename, O_RDONLY);
        ...

                    if (fd == -1) {
                            if (conf.log_enable) {
```

```
                        log_error("can not open file:%s\n", filename);

                    }
                    else {
                        fprintf(stderr, "can not open file:%s, because:%s\n",
filename, strerror(errno));
                    }
                    safe_close(loop, sock);
                    return NULL;
                }
                fd_records[sock].ffd = fd;
            }
```

=> Vulnerability Reproduce:

a. We first enable the log option in the configuration part and run the program to create a log file. We created a symbolic link called **contact-us.html** (inside the “/www” directory), and let it direct to the 2021-11-14.log file in the “./log” folder.
   ubuntu:~/environment/sparrow/www (master) $
   **ln -s ~/environment/sparrow/log/2021-11-14.log contact-us.html**

b. Use the ls -l command to display all files in “./www” folder. We can see the symbolic link has been created successfully.

```
ubuntu:~/environment/sparrow/www (master) $ ls -l
total 184
-rw-rw-r-- 1 ubuntu ubuntu   195 Nov 13 20:09  404.html
lrwxrwxrwx 1 ubuntu ubuntu     8 Nov 15 02:04  505.html -> filename
drwxrwxr-x 3 ubuntu ubuntu  4096 Nov  6 20:24  CSS
-rw-rw-r-- 1 ubuntu ubuntu 10369 Nov  6 20:40  about-us.html
-rw-rw-r-- 1 ubuntu ubuntu  7777 Nov  6 20:40  achievements.html
lrwxrwxrwx 1 ubuntu ubuntu    51 Nov 15 18:39  contact-us.html -> /home/ubuntu/environment/sparrow/log/2021-11-14.log
-rw-rw-r-- 1 ubuntu ubuntu 15525 Nov  6 20:24  home-eng.html
```

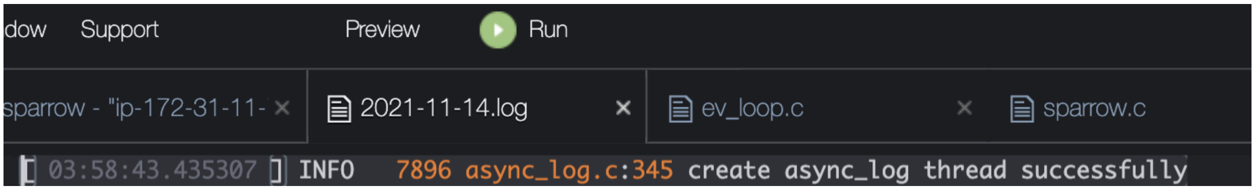
c. Then, when we click the button to contact-us.html, the webpage is like below. Compared to the content in the log file, we can tell that the information in the log has been leaked out.

Webpage:

Not Secure | 18.118.119.72:8080/contact-us.html

[ 03:58:43.435307 ] INFO 7896 async_log.c:345 cre

Log file:




=> Mitigation:

- (No code modification) Container: Use deploy this program in the container, which could reduce the risk of symbolic attack
  We chose **Docker as the container,** and to compile and run our program inside the container.
- (Code modification) Switch the open() function in Line 431 to **openat()** function**:** since using openat() could prevent the race condition by checking if the file being open is inside the correct working directory or not. As the code below shows, we firstly get the directory descriptor dirfd, and pass it into the openat. Then we are able to check the directory of the file we are opening.

```
        if (fd_records[sock].http_code != 304) {

            // Mitagation of the vulnerability: TOCTOU
            DIR *dir;
           dir = opendir("./www");
           int curr_fd =  dirfd(dir);

Line 431   fd = openat(curr_fd, filename, O_RDONLY);
           // fd = open(filename, O_RDONLY);
               ...
}
```

=> Demonstration of Mitigation:

(No code modification) Docker

a. Commands in the shell to build the Docker image called sparrow
   `docker build -t sparrow -f ./Dockerfile ./sparrow`
   `cd sparrow`
b. Compile the program inside the container
   `docker run --rm -v "$PWD":/app -w /app gcc:4.9 make`

As we can see, the sparrow program has been compiled successfully inside the container.

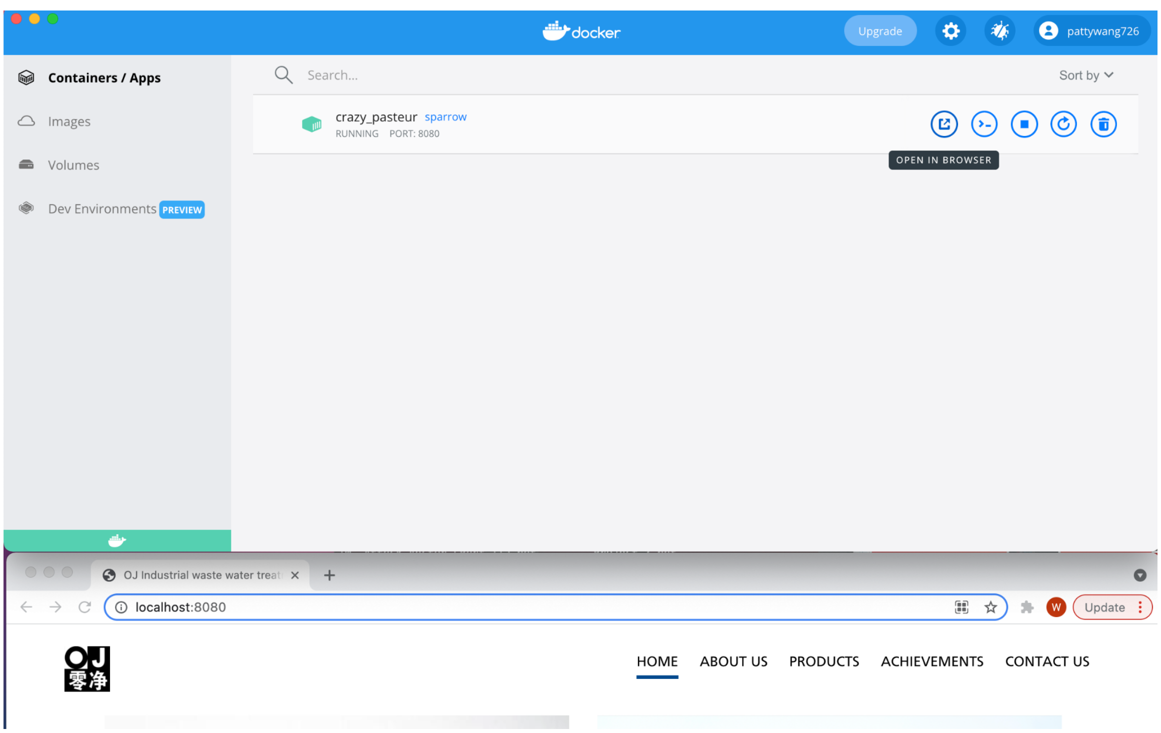
```
pattywang@Pattys-MacBook-Pro sparrow % docker run --rm -v "$PWD":/app -w /app gcc:4.9 make
gcc -Wall -c -g -Wall  sparrow.c
sparrow.c: In function 'write_http_body':
sparrow.c:745:49: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
      add_timer(loop, 40, process_timeout, 0, 0, (void*)sockfd);
                                                 ^
sparrow.c:749:49: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
      add_timer(loop, 40, process_timeout, 0, 0, (void*)sockfd);
                                                 ^
gcc -Wall -c -g -Wall thread_manage.c
gcc -Wall -c -g -Wall file.c
gcc -Wall -c -g -Wall ev_loop.c
gcc -Wall -c -g -Wall config.c
gcc -Wall -c -g -Wall async_log.c
gcc -Wall -c -g -Wall url.c
gcc -Wall -c -g -Wall min_heap.c
min_heap.c: In function 'add_timer':
min_heap.c:168:11: warning: cast from pointer to integer of different size [-Wpointer-to-int-cast]
  int fd = (int)ptr;
           ^
gcc -Wall -c -g -Wall cJSON.c
gcc -Wall -c -g -Wall picohttpparser.c
gcc sparrow.o thread_manage.o file.o ev_loop.o config.o async_log.o url.o min_heap.o cJSON.o picohttpparser.o -pthread -lrt -lm  -o sparrow
```

c. Run the container, and let the port of the container (8080) connected to the host machine (8080)

```
docker run --rm -v "$PWD":/app -w /app -p 8080:8080 sparrow
```

**Result:** The sparrow web server is running well, and our website can be accessed through localhost:8080. Since if the host path to the container is a symbolic link, the container will not be able to access them. Thus this TOCTOU vulnerability could be mitigated using container.

```
pattywang@Pattys-MacBook-Pro sparrow % docker run --rm -v "$PWD":/app -w /app -p
 8080:8080 sparrow
```



# Vulnerability 1.4

=> Vulnerability description: **time-to-check-to-time-to-use**. Before opening the file, an attacker can use a symbolic link to trick the software to read another "secret" file line by line, which will lead to the information leak.
=> Vulnerability Categories: **Concurrency Attacks**
=> Vulnerability source code snippet
The vulnerability in the source file is located in Line 99 in **config.c file**

```
            int read_config(config_t *conf) {
                char config_line[512];
Line 99           FILE *fp = fopen(CONFIG_FILE_PATH, "r");
                if (fp == NULL) {
                 fprintf(stderr, "Can not open config file:%s\n",
strerror(errno));
                 return -1;
      }
           ...
}
```

=> Vulnerability Reproduce:

a. Firstly, we print the CONFIG_FILE_PATH out, and find out it points to the directory: config/sparrow.conf. So we created a symbolic link called **sparrow.conf** (inside the "/config" directory), and let it direct to the 2021-11-17.log file in the "./log" folder.
   ubuntu:~/environment/sparrow/config (master) $
   **ln -s ~/environment/sparrow/log/2021-11-17.log sparrow.conf**

b. Use the ls -l command to display all files in "./config" folder. We can see the symbolic link has been created successfully.

```
ubuntu:~/environment/sparrow/config (master) $ ls -l
total 4
lrwxrwxrwx 1 ubuntu ubuntu  51 Nov 17 04:40 sparrow.conf -> /home/ubuntu/environment/sparrow/log/2021-11-17.log
-rw-rw-r-- 1 ubuntu ubuntu 248 Nov 13 20:09 test.conf
```

c. Then, when we just started the program, the config.c file will be run through. The shell will print the content as below. Compared to the content in the 2021-11-17.log file, we can tell that the content in the log file has been leaked out.

```
ubuntu:~/environment/sparrow (master) $ ./sparrow
config/sparrow.conf
BAD FORMAT:[ 03:35:30.001405 ] INFO   9841 async_log.c:345 create async_log thread successfully

BAD FORMAT:[ 03:35:30.048462 ] INFO   9841 sparrow.c:106 sparrow started successfully!

BAD FORMAT:[ 03:36:38.442170 ] INFO   9841 async_log.c:351 Log is going to be closed...

BAD FORMAT:[ 03:36:38.442170 ] INFO   9841 async_log.c:351 Log is going to be closed...

BAD FORMAT:[ 03:36:41.196072 ] INFO   9842 async_log.c:351 Log is going to be closed...

init
```

```
bash - "ip-172-31-11-74"   2021-11-17.log   sparrow.c   sparrow.c

[ 03:35:30.001405 ] INFO   9841 async_log.c:345 create async_log thread successfully
[ 03:35:30.048462 ] INFO   9841 sparrow.c:106 sparrow started successfully!
[ 03:36:38.442170 ] INFO   9841 async_log.c:351 Log is going to be closed...
[ 03:36:38.442170 ] INFO   9841 async_log.c:351 Log is going to be closed...
[ 03:36:41.196072 ] INFO   9842 async_log.c:351 Log is going to be closed...
[ 03:41:02.490360 ] INFO   10663 async_log.c:345 create async_log thread successfully
[ 03:41:02.556438 ] INFO   10663 sparrow.c:106 sparrow started successfully!
[ 03:41:19.790938 ] INFO   10663 async_log.c:351 Log is going to be closed...
```

=> Mitigation:

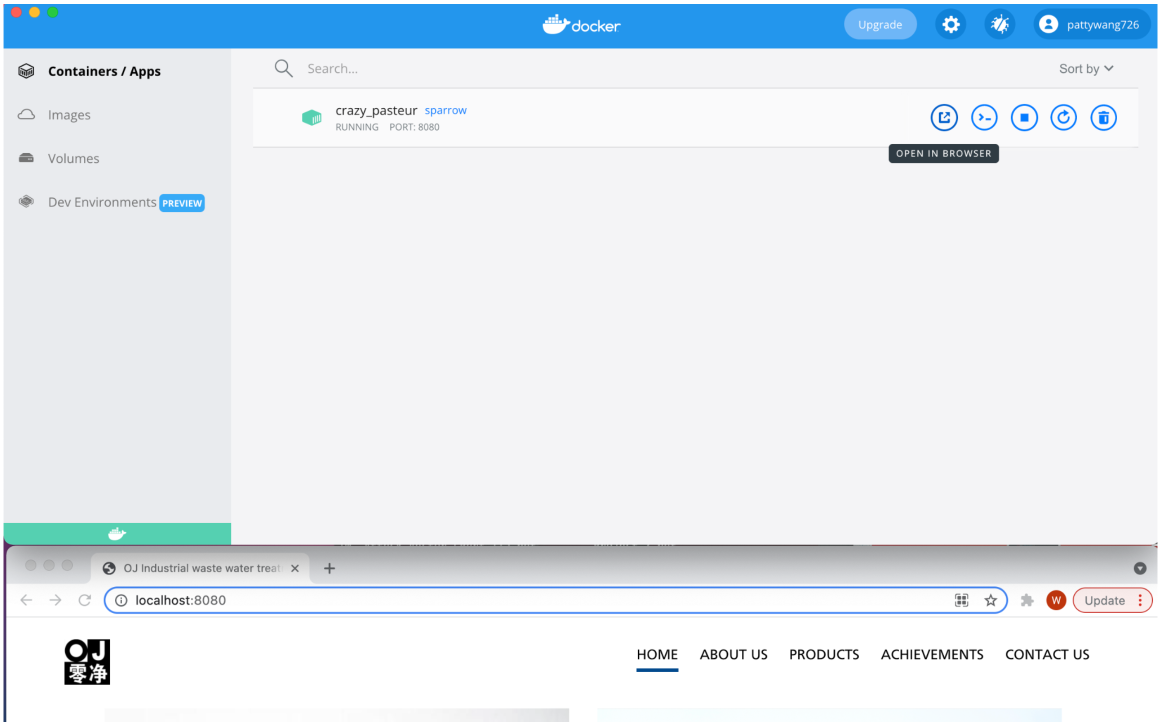
- (No code modification) Container: Use deploy this program in the container, which could reduce the risk of symbolic attack.

- (Code modification) Switch the open() function in Line 99 to **openat()** function**: since using openat**() could prevent the race condition by checking if the file being open is inside the correct working directory or not.

=> Demonstration of Mitigation:

**Result:** The sparrow web server is running well inside the container, and our website can be accessed through localhost:8080. Since if the host path to the container is a symbolic link, the container will not be able to access them. Thus this TOCTOU vulnerability could be mitigated using containers.

```
pattywang@Pattys-MacBook-Pro sparrow % docker run --rm -v "$PWD":/app -w /app -p
 8080:8080 sparrow
```




# Vulnerability 1.5

=> Vulnerability description: **Use of Uninitialized Variable.** The vulnerability is due to the member, epfd, of the loop being not initialized before it is used in the epoll_ctl. And the epfd is epoll file descriptor. If the epfd is not initialized, and possibly contains junk data, and thus an attacker can take control of these contents, to manipulate the events in the programm. Thus, this vulnerability may modify the control flow and enable code execution as the attacker wants.

=> Categories: **Control Flow Attacks**

=> Vulnerability Source code snippet:

The vulnerability in the source file is located in Line 179 in **ev_loop.c file**

```
        ...
Line 179if (-1 == epoll_ctl(loop->epfd, EPOLL_CTL_MOD, fd, &ev)) {
                        fprintf(stderr, "epoll_ctl mod in ev_stop\n");
                        ev_clear(fd);
                        return -1;
                }
        ...
                return 0;
        }
        return 0;
}

// ev_loop_t is defined as below in the ev_loop.h
```

```
typedef struct ev_loop_t{
                    int epfd;
                    int maxevent;
                    int etmodel;
                    //fd_record_t *fd_records;
                    struct epoll_event *events;

                    //timer
                    //struct ev_timer_t **heap;
                    void **heap;
                    int heap_size;
                    int heap_capacity;
                    int timer_fd;
          }ev_loop_t;
```

As we can see from the code, the epfd is defined in the struct ev_loop_t, but no initialization of epfd happens in the whole source code. Thus, before the epoll_ctl is invoked, the epfd is not initialized. And the epfd is epoll file descriptor, and epoll_ctl is used to add file descriptors it wants to be monitored under the specific epoll list, **thus if the epfd is not clarified, possibly the file descriptors are not monitored successfully,** and an attacker can make use of this to manipulate or crash the software.

=> Vulnerability Reproduce:

Insert code to print the uninitialized value out.

```
        char text_epfd[10];
        sprintf(text_epfd, "%d", loop->epfd);
        fprintf(stdout, "%s\n", text_epfd);
```

As we can see in the below pictures, the loop->epfd is randomly assigned.

```
vulnerability_6
3
vulnerability_6
3
read_http
vulnerability_6
3
read_http
vulnerability_6
8
read_http
vulnerability_6
3
read_http
vulnerability_6
5
```

```
ubuntu:~/environment/sparrow (master) $ ./sparrow
init
sparrow started successfully!
read_http
vulnerability_6
3
read_http
vulnerability_6
5
read_http
vulnerability_6
8
read_http
vulnerability_6
3
read_http
vulnerability_6
8
read_http
vulnerability_6
5
read_http
vulnerability_6
8
read_http
vulnerability_6
5
read_http
vulnerability_6
3
read_http
vulnerability_6
5
read_http
vulnerability_6
3
```

=> Mitigation:

- (No code modification)  Use Valgrind to detect the unintilization.

- (Code modification)  Just initialize the loop->epfd to the correct epoll file descriptor .

=> Demonstration of Mitigation:

- (No code modification)  Use Valgrind, and the utilization has been detected.

```
==6915== Syscall param epoll_ctl(event) points to uninitialised byte(s)
==6915==    at 0x57250DA: epoll_ctl (syscall-template.S:78)
==6915==    by 0x10E008: ev_stop (ev_loop.c:179)
==6915==    by 0x10C203: write_http_header (sparrow.c:631)
==6915==    by 0x10E24F: ev_run_loop (ev_loop.c:221)
==6915==    by 0x10CAF5: worker_threads_entrance (thread_manage.c:27)
==6915==    by 0x53EB6DA: start_thread (pthread_create.c:463)
==6915==    by 0x572471E: clone (clone.S:95)
==6915==  Address 0xa632d04 is on thread 2's stack
==6915==  in frame #1, created by ev_stop (ev_loop.c:162)
==6915==
==6915== Syscall param epoll_ctl(event) points to uninitialised byte(s)
==6915==    at 0x57250DA: epoll_ctl (syscall-template.S:78)
==6915==    by 0x10DCA2: ev_register (ev_loop.c:120)
==6915==    by 0x10C245: write_http_header (sparrow.c:633)
==6915==    by 0x10E24F: ev_run_loop (ev_loop.c:221)
==6915==    by 0x10CAF5: worker_threads_entrance (thread_manage.c:27)
==6915==    by 0x53EB6DA: start_thread (pthread_create.c:463)
==6915==    by 0x572471E: clone (clone.S:95)
==6915==  Address 0xa632cf4 is on thread 2's stack
==6915==  in frame #1, created by ev_register (ev_loop.c:82)
```

## Vulnerability 1.6 & 1.7

Notice: The reason we put two vulnerabilities here is that even two vulnerabilities appear in the same line, they belong to the different variables. They should be treated as two vulnerabilities. In the source code starting from line 334, it used str_equal function twice.

=> Vulnerability description: **Buffer Overflow**. Str_equal function measures whether the first len bytes of two strings are equal or not. Inside the function, it used **memcmp** to compare the two strings. Out-Of-Bound access will happen if the length of the second string t is larger than the length of the first string str.
=> Vulnerability categories: **Spatial Memory Attacks**
=> Vulnerability Source code snippet:
The vulnerability in the source file is located in Line 335 in **sparrow.c file**

```
Line 79 char *str_2_lower(char *str, int len) {
      int i;
      for (i = 0; i < len; i++) {
            str[i] = tolower(str[i]);
      }
      return str;
}
Line 48 static
        int str_equal(char* str, size_t len, const char* t)
        {
           return memcmp(str_2_lower(str, len), t, len) == 0;
        }

Line 334    //Keep-alive and modified time
for (i = 0; i != (int)num_headers; ++i) {

Line 335      if (str_equal((char *)headers[i].name, headers[i].name_len,
"connection") &&
           str_equal((char*) headers[i].value, headers[i].value_len, "keep-alive"))
        {
          ...
        }
```

=> Reproducing process:

a. Insert the code below inside the for loop but before the if statement to print the headers[i].name and  headers[i].name_len:

```
printf("strlenhdN:%d\n",strlen((char *)headers[i].name));
fprintf(stdout, "%d\n", headers[i].name_len);
printf("strlenhdV:%d\n",strlen((char *)headers[i].value));
```

```
fprintf(stdout, "%d\n", headers[i].value_len);
printf("\n\n\n\n");
```

b. Vulnerability 6
   Like the third chunk of the example shown below, the headers[i].name_len is 13, but the length of "connection" is 10. **memcmp** function will not check the boundary which may cause the vulnerability.

```
gcc sparrow.o thread_manage.o file.o ev_loop.o config.o async_log.o url.o min_heap.o cJSON.o picohttpparser.o -pthread
 -lrt -lm  -o sparrow
ccc@DESKTOP-VC8V910:/mnt/c/Users/zhang/Documents/sparrow$ ./sparrow
init
sparrow started successfully!
strlenhdN:1118
4
strlenhdV:1112
14




strlenhdN:1096
10
strlenhdV:1084
10




strlenhdN:1072
13
strlenhdV:1057
9
```

c. Vulnerability 7
   Like the below picture, headers[i].value_len is 64, but the length of "keep-alive" is 10. **memcmp** function will not check the boundary which may cause the vulnerability.

```
strlenhdN:1046
9
strlenhdV:1035
64
```

=> Mitigation: **AddressSantizer**
   Add the **-fsanitize=address** to the compiler flags and linker flags in the makefile.

```
BIN := sparrow
OBJS := sparrow.o thread_manage.o file.o ev_loop.o config.o
```

```
async_log.o url.o min_heap.o cJSON.o picohttpparser.o
CC := gcc
DEBUG := -g -Wall
CFLAGS := -fsanitize=address -Wall -c $(DEBUG)
LFLAGS :=  -fsanitize=address -pthread -lrt -lm
```

=> Mitigation process: AddressSantizer then discovered this error and the program aborted, which can prevent those vulnerabilities.

```
=================================================================
==15988==ERROR: AddressSanitizer: global-buffer-overflow on address 0x555dd625040b at pc 0x7f87e6068bb5 bp 0x7f87de49da90 sp 0x7f87de49d
READ of size 13 at 0x555dd625040b thread T1
    #0 0x7f87e6068bb4  (/usr/lib/x86_64-linux-gnu/libasan.so.4+0xafbb4)
    #1 0x555dd6236c44 in str_equal /home/ubuntu/environment/sparrow_test/sparrow.c:48
    #2 0x555dd6238d35 in read_http /home/ubuntu/environment/sparrow_test/sparrow.c:335
    #3 0x555dd623efc3 in ev_run_loop /home/ubuntu/environment/sparrow_test/ev_loop.c:216
    #4 0x555dd623bd31 in worker_threads_entrance /home/ubuntu/environment/sparrow_test/thread_manage.c:27
    #5 0x7f87e5a036da in start_thread (/lib/x86_64-linux-gnu/libpthread.so.0+0x76da)
    #6 0x7f87e572c71e in __clone (/lib/x86_64-linux-gnu/libc.so.6+0x12171e)

0x555dd625040b is located 0 bytes to the right of global variable '*.LC32' defined in 'sparrow.c' (0x555dd6250400) of size 11
  '*.LC32' is ascii string 'connection'
0x555dd625040b is located 53 bytes to the left of global variable '*.LC33' defined in 'sparrow.c' (0x555dd6250440) of size 11
  '*.LC33' is ascii string 'keep-alive'
SUMMARY: AddressSanitizer: global-buffer-overflow (/usr/lib/x86_64-linux-gnu/libasan.so.4+0xafbb4)
Shadow bytes around the buggy address:
  0x0aac3ac42030: f9 f9 f9 f9 00 00 f9 f9 f9 f9 f9 f9 00 06 f9 f9
  0x0aac3ac42040: f9 f9 f9 f9 00 00 06 f9 f9 f9 f9 f9 00 02 f9 f9
  0x0aac3ac42050: f9 f9 f9 f9 00 04 f9 f9 f9 f9 f9 f9 00 00 00 00
  0x0aac3ac42060: 00 00 00 05 f9 f9 f9 f9 04 f9 f9 f9 f9 f9 f9 f9
  0x0aac3ac42070: 04 f9 f9 f9 f9 f9 f9 f9 04 f9 f9 f9 f9 f9 f9 f9
=>0x0aac3ac42080: 00[03]f9 f9 f9 f9 f9 f9 00 03 f9 f9 f9 f9 f9 f9
  0x0aac3ac42090: 00 00 07 f9 f9 f9 f9 f9 00 00 02 f9 f9 f9 f9 f9
  0x0aac3ac420a0: 00 00 00 01 f9 f9 f9 f9 02 f9 f9 f9 f9 f9 f9 f9
  0x0aac3ac420b0: 06 f9 f9 f9 f9 f9 f9 f9 07 f9 f9 f9 f9 f9 f9 f9
  0x0aac3ac420c0: 00 01 f9 f9 f9 f9 f9 f9 00 00 02 f9 f9 f9 f9 f9
  0x0aac3ac420d0: 00 00 06 f9 f9 f9 f9 f9 00 00 00 00 02 f9 f9 f9
```

## Vulnerability 1.8

=> Vulnerability description: **Data Race.** This software is running with 3 Threads. As we can see, the fd_records[fd] is read in Line 213 of the **"ev_loop.c" file** , while  fd_records[fd] may be written when the read system call is executed by another Thread.
=> Vulnerability categories: **Concurrency Attacks**

=> Vulnerability Source code snippet:
The vulnerability in the source file is located in Line 213 in **ev_loop.c file,** and  Line 232 in **sparrow.file.**
<u>ev_loop.c file</u>

```
Line 213  fd_record_t record = fd_records[fd];
```

<u>sparrow.file</u>

```
Line 232  nread = read(sock, buf + fd_records[sock].read_pos, MAXBUFSIZE - fd_records[sock].read_pos);
```

=> Vulnerability Reproduce:   valgrind --tool=helgrind ./sparrow

```
==5962== Possible data race during read of size 2 at 0x5ECB202 by thread #2
==5962== Locks held: none
==5962==    at 0x4C3DBD0: memmove (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==5962==    by 0x10E163: ev_run_loop (ev_loop.c:213)
==5962==    by 0x10CAF5: worker_threads_entrance (thread_manage.c:27)
==5962==    by 0x4C38C26: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==5962==    by 0x53F16DA: start_thread (pthread_create.c:463)
==5962==    by 0x572A71E: clone (clone.S:95)
==5962==
==5962== This conflicts with a previous write of size 8 by thread #3
==5962== Locks held: none
==5962==    at 0x53FB474: read (read.c:27)
==5962==    by 0x10ACAE: read_http (sparrow.c:232)
==5962==    by 0x10E1BA: ev_run_loop (ev_loop.c:216)
==5962==    by 0x10CAF5: worker_threads_entrance (thread_manage.c:27)
==5962==    by 0x4C38C26: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==5962==    by 0x53F16DA: start_thread (pthread_create.c:463)
==5962==    by 0x572A71E: clone (clone.S:95)
==5962==  Address 0x5ecb202 is 856,514 bytes inside a block of size 67,428,352 alloc'd
==5962==    at 0x4C32F2F: malloc (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==5962==    by 0x10D881: ev_create_loop (ev_loop.c:53)
==5962==    by 0x10CB75: worker_threads_init (thread_manage.c:42)
==5962==    by 0x10A625: main (sparrow.c:75)
==5962==  Block was alloc'd by thread #1
```

=> Mitigation: Use the **ThreadSanitizer** to Detect the malicious use cases.

Add the **-fsanitize=address** to the compiler flags and linker flags in the makefile.

```
BIN := sparrow
OBJS := sparrow.o thread_manage.o file.o ev_loop.o config.o
async_log.o url.o min_heap.o cJSON.o picohttpparser.o
CC := gcc
DEBUG := -g -Wall
CFLAGS := -fsanitize=thread -Wall -c $(DEBUG)
LFLAGS := -fsanitize=thread -pthread -lrt -lm
```

=> Demonstration of Mitigation:  As we can see from the screenshot below, the Data race has been detected.

```
SUMMARY: ThreadSanitizer: data race /home/ubuntu/environment/sparrow/ev_loop.c:213 in ev_run_loop
==================
==================
WARNING: ThreadSanitizer: data race (pid=7307)
  Read of size 8 at 0x7f71814b3408 by thread T3:
    #0 ev_run_loop /home/ubuntu/environment/sparrow/ev_loop.c:213 (sparrow+0x8307)
    #1 worker_threads_entrance /home/ubuntu/environment/sparrow/thread_manage.c:27 (sparrow+0x62a2)
    #2 <null> <null> (libtsan.so.0+0x2e3bf)

  Previous write of size 1 at 0x7f71814b340f by thread T2:
    #0 str_2_lower /home/ubuntu/environment/sparrow/util.h:82 (sparrow+0x2ca3)
    #1 str_equal /home/ubuntu/environment/sparrow/sparrow.c:50 (sparrow+0x2cfc)
    #2 read_http /home/ubuntu/environment/sparrow/sparrow.c:336 (sparrow+0x40c8)
    #3 ev_run_loop /home/ubuntu/environment/sparrow/ev_loop.c:216 (sparrow+0x839c)
    #4 worker_threads_entrance /home/ubuntu/environment/sparrow/thread_manage.c:27 (sparrow+0x62a2)
    #5 <null> <null> (libtsan.so.0+0x2e3bf)

  Location is heap block of size 67428352 at 0x7f71813b2000 allocated by main thread:
    #0 malloc <null> (libtsan.so.0+0x319a3)
    #1 ev_create_loop /home/ubuntu/environment/sparrow/ev_loop.c:53 (sparrow+0x748b)
    #2 worker_threads_init /home/ubuntu/environment/sparrow/thread_manage.c:42 (sparrow+0x6385)
    #3 main /home/ubuntu/environment/sparrow/sparrow.c:75 (sparrow+0x2e1d)

  Thread T3 (tid=7325, running) created by main thread at:
    #0 pthread_create <null> (libtsan.so.0+0x5fe84)
    #1 worker_threads_init /home/ubuntu/environment/sparrow/thread_manage.c:45 (sparrow+0x6440)
    #2 main /home/ubuntu/environment/sparrow/sparrow.c:75 (sparrow+0x2e1d)

  Thread T2 (tid=7324, running) created by main thread at:
    #0 pthread_create <null> (libtsan.so.0+0x5fe84)
    #1 worker_threads_init /home/ubuntu/environment/sparrow/thread_manage.c:45 (sparrow+0x6440)
    #2 main /home/ubuntu/environment/sparrow/sparrow.c:75 (sparrow+0x2e1d)
```

# Vulnerability 1.9

=> Vulnerability description: **Buffer Overflow**. In this vulnerability, it used **memcmp** directly to compare the two strings. Out-Of-Bound access will happen if the length used to compare is longer than the shortest string, which is “/” here.
=> Vulnerability categories: **Spatial Memory Attacks**
=> Vulnerability Source code snippet:
The vulnerability in the source file is located in Line 335 in **sparrow.c file**

```
Line 363 if (memcmp(action, "/", action_len) == 0) {
                    snprintf(filename, 512, "%s/%s", prefix,
conf.def_home_page);
                }
```

=> Vulnerability Reproduce:

a. When we insert code to print out the “action” string, we set the output as below:

```
ubuntu:~/environment/sparrow (master) $ ./sparrow
init
sparrow started successfully!
http://localhost:8080/
vulnerability_9
/ HTTP/1.1
host: 3.16.135.122:8080
connection: keep-alive
upgrade-insecure-requests: 1
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/95.0.4638.54 Safari/537.36
accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0
accept-encoding: gzip, deflate
accept-language: en-US,en;q=0.9
if-modified-since: Sat Nov  6 20:24:43 2021


vulnerability_9
/CSS/Jesde2.css HTTP/1.1
host: 3.16.135.122:8080
connection: keep-alive
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/95.0.4638.54 Safari/537.36
accept: text/css,*/*;q=0.1
referer: http://3.16.135.122:8080/
accept-encoding: gzip, deflate
accept-language: en-US,en;q=0.9
if-modified-since: Sat Nov  6 20:24:26 2021
```

b. As we can see, the string pointer by “action” is definitely longer than “/”, but the length used to compare the strings is action_len, which is larger than 1, the length of the “/”.

=> Mitigation: **AddressSantizer**

Add the **-fsanitize=address** to the compiler flags and linker flags in the makefile.

```
BIN := sparrow
OBJS := sparrow.o thread_manage.o file.o ev_loop.o config.o
async_log.o url.o min_heap.o cJSON.o picohttpparser.o
CC := gcc
DEBUG := -g -Wall
CFLAGS := -fsanitize=address -Wall -c $(DEBUG)
LFLAGS := -fsanitize=address -pthread -lrt -lm
```

=> Demonstration of Mitigation: AddressSantizer then discovered this error, and the program will be aborted, if this problem is not fixed.

```
ubuntu:~/environment/sparrow_test (master) $ ./sparrow
init
sparrow started successfully!
=================================================================
==20302==ERROR: AddressSanitizer: global-buffer-overflow on address 0x55de252be2a2 at pc 0x7f4d57457bb5 bp 0x7f4d4f89d6a0 sp 0x7f4d4f89
READ of size 15 at 0x55de252be2a2 thread T1
    #0 0x7f4d57457bb4  (/usr/lib/x86_64-linux-gnu/libasan.so.4+0xafbb4)
    #1 0x55de252a7021 in read_http /home/ubuntu/environment/sparrow_test/sparrow.c:368
    #2 0x55de252acda6 in ev_run_loop /home/ubuntu/environment/sparrow_test/ev_loop.c:216
    #3 0x55de252a9b14 in worker_threads_entrance /home/ubuntu/environment/sparrow_test/thread_manage.c:27
    #4 0x7f4d56df26da in start_thread (/lib/x86_64-linux-gnu/libpthread.so.0+0x76da)
    #5 0x7f4d56b1b71e in __clone (/lib/x86_64-linux-gnu/libc.so.6+0x12171e)

0x55de252be2a2 is located 0 bytes to the right of global variable '*.LC35' defined in 'sparrow.c' (0x55de252be2a0) of size 2
  '*.LC35' is ascii string '/'
0x55de252be2a2 is located 62 bytes to the left of global variable '*.LC36' defined in 'sparrow.c' (0x55de252be2e0) of size 6
  '*.LC36' is ascii string '%s/%s'
SUMMARY: AddressSanitizer: global-buffer-overflow (/usr/lib/x86_64-linux-gnu/libasan.so.4+0xafbb4)
Shadow bytes around the buggy address:
  0x0abc44a4fc00: 00 00 06 f9 f9 f9 f9 f9 00 02 f9 f9 f9 f9 f9 f9
  0x0abc44a4fc10: 00 04 f9 f9 f9 f9 f9 f9 00 00 00 00 00 00 00 05
  0x0abc44a4fc20: f9 f9 f9 f9 04 f9 f9 f9 f9 f9 f9 f9 00 03 f9 f9
  0x0abc44a4fc30: f9 f9 f9 f9 00 03 f9 f9 f9 f9 f9 f9 00 00 07 f9
  0x0abc44a4fc40: f9 f9 f9 f9 00 00 02 f9 f9 f9 f9 f9 00 00 00 01
=>0x0abc44a4fc50: f9 f9 f9 f9[02]f9 f9 f9 f9 f9 f9 f9 06 f9 f9 f9
  0x0abc44a4fc60: f9 f9 f9 f9 07 f9 f9 f9 f9 f9 f9 f9 00 01 f9 f9
  0x0abc44a4fc70: f9 f9 f9 f9 00 00 02 f9 f9 f9 f9 f9 00 00 06 f9
  0x0abc44a4fc80: f9 f9 f9 f9 00 00 00 00 02 f9 f9 f9 f9 f9 f9 f9
  0x0abc44a4fc90: 00 02 f9 f9 f9 f9 f9 f9 00 03 f9 f9 f9 f9 f9 f9
  0x0abc44a4fca0: 00 00 00 00 00 00 00 00 00 00 00 00 00 04 f9 f9
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
```

## Vulnerability 1.10

=> Vulnerability description: Buffer overflow.
=> Categories: **Spatial Memory Attacks**
=> Source code snippet
The vulnerability in the source file is located in Line 511 in **sparrow.c file.** Before using the **strcpy** function, the code failed to check whether the destination has enough space to do the copy operation. **Strcpy** doesn't do bound checking. Content_type only can hold 32 bytes, but it forgot to check the size of mime_type[index].l_type.

```
#ifndef _MIME_H
#define _MIME_H

#include <string.h>

typedef struct {
      char l_type[64];
      char s_type[16];
} mime_type_t;


char content_type[32];
           if (suffix == NULL) {

           else {

      // printf("strlenhdN:%d\n",strlen((char *)headers[i].name));
           fprintf(stdout, "sdfsd%d\n", index);
           if (index == -1) {
                strcpy(content_type, "text/plain");
           }
           else {

                char name[64] =
"application/pdfasssssssssssssssssdddddddddddddddddddddddddddd";
                 strncpy(mime_type[index].l_type, name, 64);
                  printf("strlenhdN:%s\n",mime_type[index].l_type);
                 strcpy(content_type, mime_type[index].l_type);
                          }
```

```
                       }
                 }
```

=> Reproducing process:

a. Replace the content of mime_type[index].l_type with a long char array
b. Print the mime_type[index].l_type to see the contents, which prove that vulnerability got executed

```
gcc sparrow.o thread_manage.o file.o ev_loop.o config.o async_log.o url.o min_heap.o cJSON.o picohttpparser.o -pthread -lrt -lm  -o sparrow
ccc@DESKTOP-VC8V910:/mnt/c/Users/zhang/Documents/sparrow$ ./sparrow
init
sparrow started successfully!
sdfsd108
strlenhdN:application/pdfasssssssssssssssssssddddddddddddddddddddddddddddddpdf
./www/files/lingpaifang.pdf=============ppppp==============
./www/files/lingpaifang.pdf./www/files/lingpaifang.pdf========dir_html_maker======hhhhhhh==============
block read err:Not a directory
^C
ccc@DESKTOP-VC8V910:/mnt/c/Users/zhang/Documents/sparrow$ ASAN_OPTIONS=halt_on_error=0 ./sparrow
init
```

=> Mitigation: **AddressSantizer**

Add the **-fsanitize=address** to the compiler flags and linker flags in the makefile.

```
BIN := sparrow
OBJS := sparrow.o thread_manage.o file.o ev_loop.o config.o
async_log.o url.o min_heap.o cJSON.o picohttpparser.o
CC := gcc
DEBUG := -g -Wall
CFLAGS := -fsanitize=address -Wall -c $(DEBUG)
LFLAGS := -fsanitize=address -pthread -lrt -lm
```

=> Demonstration of Mitigation: AddressSantizer then discovered this error, and the program will be aborted, if this problem is not fixed.

```
==424==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7f83f706fbf0 at pc 0x7f83ffca63a6 bp 0x7f83f706e670 sp 0x7f83f706de18
WRITE of size 68 at 0x7f83f706fbf0 thread T3
    #0 0x7f83ffca63a5  (/usr/lib/x86_64-linux-gnu/libasan.so.4+0x663a5)
    #1 0x7f8401209f30 in read_http /mnt/c/Users/zhang/Documents/sparrow/sparrow.c:511
    #2 0x7f840120f367 in ev_run_loop /mnt/c/Users/zhang/Documents/sparrow/ev_loop.c:217
    #3 0x7f840120c051 in worker_threads_entrance /mnt/c/Users/zhang/Documents/sparrow/thread_manage.c:27
    #4 0x7f83ff6876da in start_thread (/lib/x86_64-linux-gnu/libpthread.so.0+0x76da)
    #5 0x7f83ff3a171e in __clone (/lib/x86_64-linux-gnu/libc.so.6+0x12171e)

Address 0x7f83f706fbf0 is located in stack of thread T3 at offset 4256 in frame
    #0 0x7f8401207eac in read_http /mnt/c/Users/zhang/Documents/sparrow/sparrow.c:211

  This frame has 16 object(s):
    [32, 36) 'minor_version'
    [96, 100) 'action_len'
    [160, 164) 'kvs_num'
    [224, 232) 'method'
    [288, 296) 'path'
    [352, 360) 'method_len'
    [416, 424) 'path_len'
    [480, 488) 'num_headers'
    [544, 552) 'action'
    [608, 616) 'last_modified_time'
    [672, 688) 'curtv'
    [736, 737) 'kvs'
    [800, 944) 'filestat'
    [992, 4192) 'headers'
    [4224, 4256) 'content_type'
    [4288, 4352) 'name' <== Memory access at offset 4256 partially underflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism or swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow (/usr/lib/x86_64-linux-gnu/libasan.so.4+0x663a5)
Shadow bytes around the buggy address:
  0x0ff0fee05f20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0ff0fee05f30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0ff0fee05f40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0ff0fee05f50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0ff0fee05f60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```