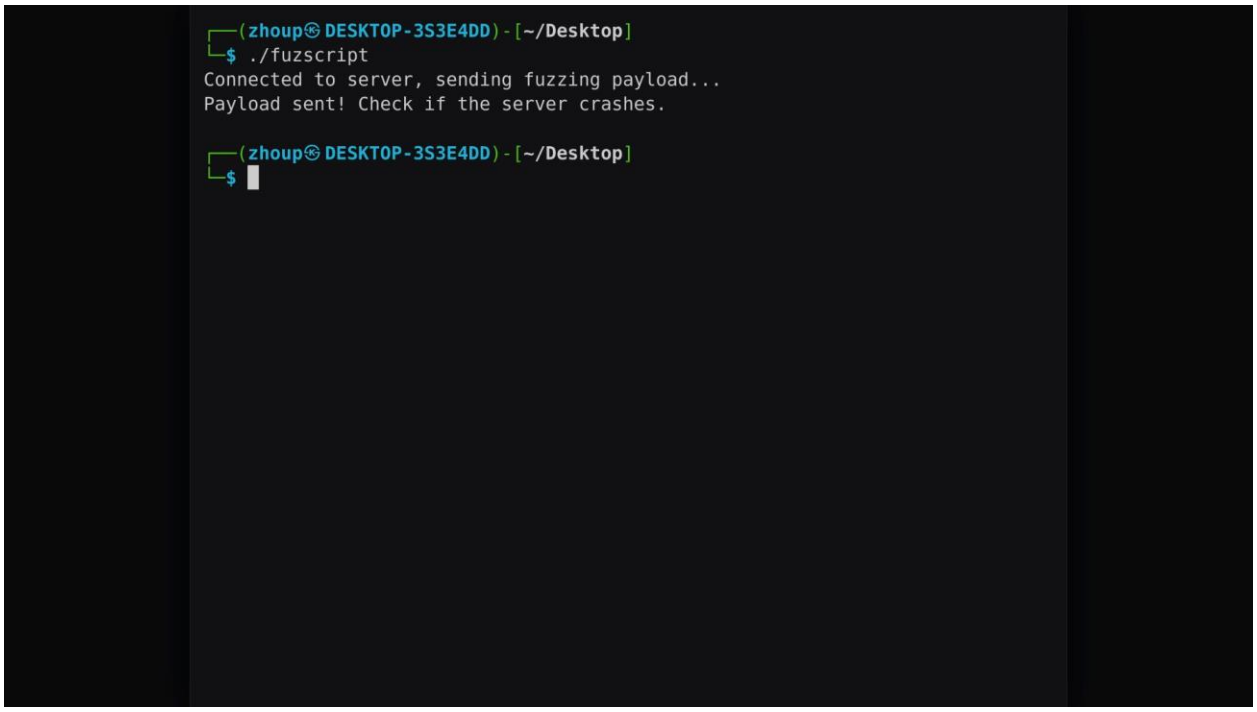
Server Exploitation Exploration

Paris Zhou

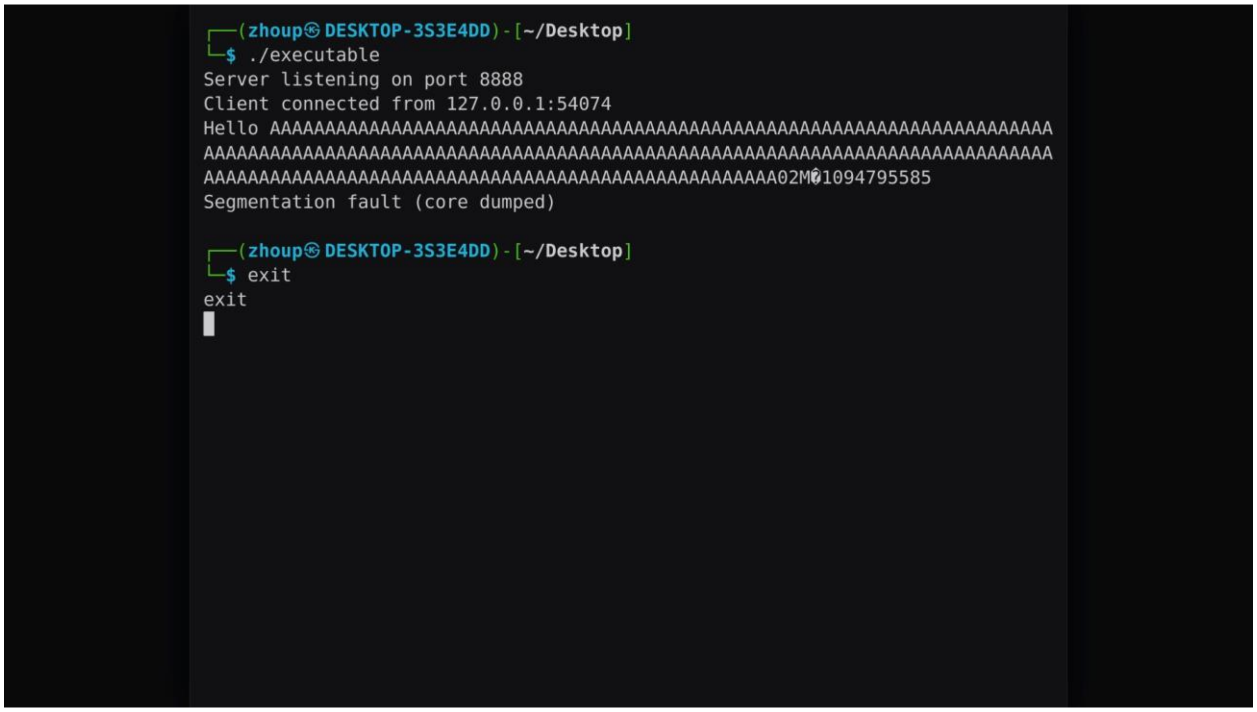
2/26/2025

Screenshot of Fuzzing Result



```
┌──(zhoup㉿DESKTOP-3S3E4DD)-[~/Desktop]
└─$ ./fuzscript
Connected to server, sending fuzzing payload...
Payload sent! Check if the server crashes.

┌──(zhoup㉿DESKTOP-3S3E4DD)-[~/Desktop]
└─$ 
```



```
┌──(zhoup㉿DESKTOP-3S3E4DD)-[~/Desktop]
└─$ ./executable
Server listening on port 8888
Client connected from 127.0.0.1:54074
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA02M�1094795585
Segmentation fault (core dumped)

┌──(zhoup㉿DESKTOP-3S3E4DD)-[~/Desktop]
└─$ exit
exit
```

What Was Done to Trigger the Issue

To identify vulnerabilities in the test server, I developed a custom C-based fuzzing script.

The script:

1. Establishes a TCP connection to the server on 127.0.0.1:8888.
2. Sends a specially crafted payload consisting of 200 ‘A’ characters, exceeding the `64-byte` buffer limit in the server.
3. Monitors for a crash, such as the server becoming unresponsive or terminating unexpectedly.

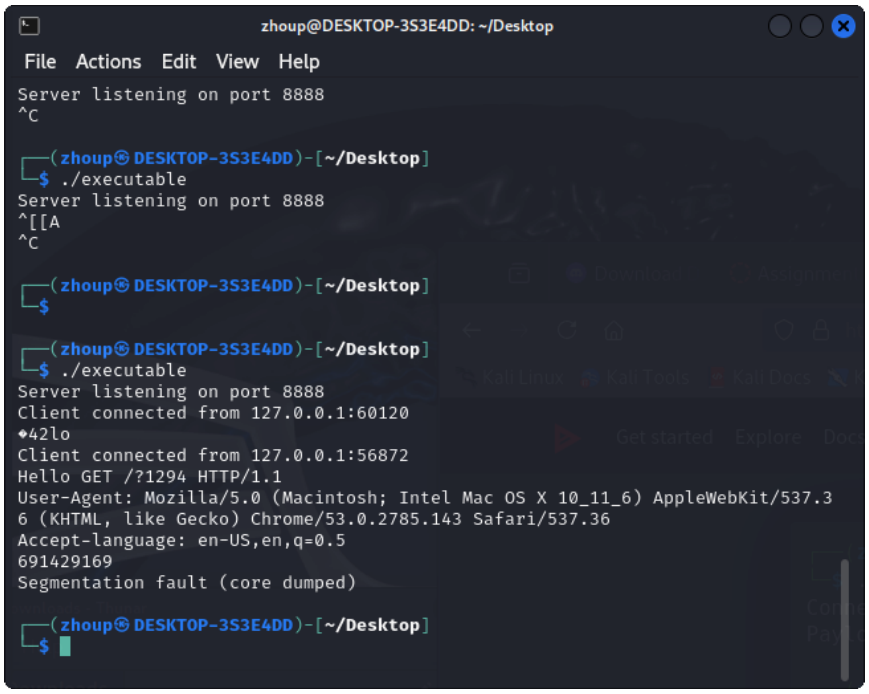
How I Know the Fuzzing Worked

The fuzzing test successfully caused the server to crash. Here’s how I confirmed the vulnerability:

- **Server Output:** The server stopped responding after receiving the fuzzing payload.
- **Segmentation Fault:** Running `dmesg | tail` displayed a segmentation fault message, confirming a buffer overflow.

This confirms that the server is vulnerable to buffer overflow attacks, and the issue needs to be patched to prevent exploitation.

Screenshot of DoS Result



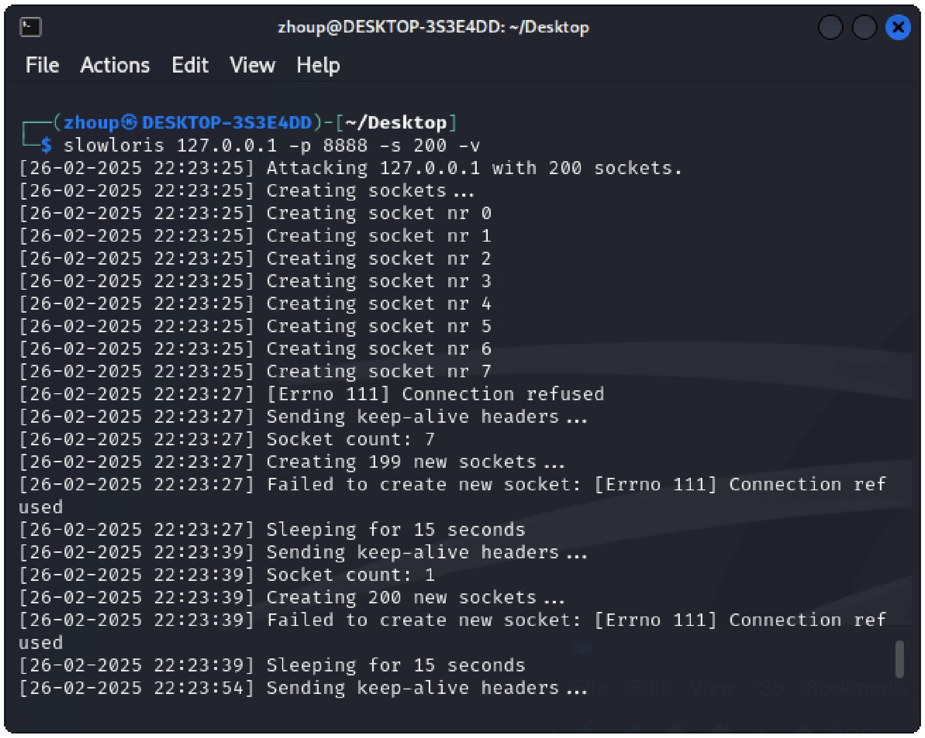
```
Server listening on port 8888
^C

┌──(zhoup㉿DESKTOP-3S3E4DD)-[~/Desktop]
└─$ ./executable
Server listening on port 8888
^[[A
^C

┌──(zhoup㉿DESKTOP-3S3E4DD)-[~/Desktop]
└─$

┌──(zhoup㉿DESKTOP-3S3E4DD)-[~/Desktop]
└─$ ./executable
Server listening on port 8888
Client connected from 127.0.0.1:60120
♦42lo
Client connected from 127.0.0.1:56872
Hello GET /?1294 HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.3
6 (KHTML, like Gecko) Chrome/53.0.2785.143 Safari/537.36
Accept-language: en-US,en,q=0.5
691429169
Segmentation fault (core dumped)

┌──(zhoup㉿DESKTOP-3S3E4DD)-[~/Desktop]
└─$
```



```
┌──(zhoup㉿DESKTOP-3S3E4DD)-[~/Desktop]
└─$ slowloris 127.0.0.1 -p 8888 -s 200 -v
[26-02-2025 22:23:25] Attacking 127.0.0.1 with 200 sockets.
[26-02-2025 22:23:25] Creating sockets ...
[26-02-2025 22:23:25] Creating socket nr 0
[26-02-2025 22:23:25] Creating socket nr 1
[26-02-2025 22:23:25] Creating socket nr 2
[26-02-2025 22:23:25] Creating socket nr 3
[26-02-2025 22:23:25] Creating socket nr 4
[26-02-2025 22:23:25] Creating socket nr 5
[26-02-2025 22:23:25] Creating socket nr 6
[26-02-2025 22:23:25] Creating socket nr 7
[26-02-2025 22:23:27] [Errno 111] Connection refused
[26-02-2025 22:23:27] Sending keep-alive headers ...
[26-02-2025 22:23:27] Socket count: 7
[26-02-2025 22:23:27] Creating 199 new sockets ...
[26-02-2025 22:23:27] Failed to create new socket: [Errno 111] Connection ref
used
[26-02-2025 22:23:27] Sleeping for 15 seconds
[26-02-2025 22:23:39] Sending keep-alive headers ...
[26-02-2025 22:23:39] Socket count: 1
[26-02-2025 22:23:39] Creating 200 new sockets ...
[26-02-2025 22:23:39] Failed to create new socket: [Errno 111] Connection ref
used
[26-02-2025 22:23:39] Sleeping for 15 seconds
[26-02-2025 22:23:54] Sending keep-alive headers ...
```

What Was Done to Trigger the Issue

To demonstrate that the test server is vulnerable to a **Denial of Service (DoS) attack**, I used **Slowloris** to exhaust the server's available connections. The goal was to keep connections open for an extended period, preventing the server from accepting new clients.

The attack was executed using the following command:

```
slowloris 127.0.0.1 -p 8888 -s 200 -v
```

This test simulates a Slowloris attack, which holds multiple connections open for extended periods, preventing new legitimate connections from being processed. Unlike high-bandwidth floods, this attack is stealthy and resource-efficient but devastating to unprotected servers.

How I Know the DoS Worked

1. Server Crashed with a Segmentation Fault: Instead of just becoming unresponsive, the server crashed entirely, resulting in a segmentation fault and core dump.
2. Limited Socket Creation Before Crash: Despite attempting to open 200 connections, the server only handled about 8 connections before crashing, indicating a severe memory handling issue.
3. No Response from Server: Attempts to connect using `nc 127.0.0.1 8888` failed immediately after the attack, confirming the server had completely stopped.
4. Observed Crash Details: Checking logs with `dmesg | tail` revealed a segmentation fault, meaning the server was unable to handle the malformed or excessive requests from Slowloris.

This confirms that the server is highly vulnerable to DoS attacks through connection exhaustion and also due to poor memory management leading to a crash.

Identified Issue in the Original Server Code

The primary issue in the original server code was a buffer overflow vulnerability in the `vulnerable_fuction()` function:

```
void vulnerable_fuction(char *buffer){
        char name[64];
        strcpy(name, buffer);  // Unsafe copying leads to buffer overflow
}
```

The function copies the received buffer into a fixed-size array (`name[64]`) without bounds checking.
If an input longer than 64 bytes is received, it overwrites adjacent memory, leading to a crash or potential remote code execution.
Our fuzzing test successfully triggered this vulnerability, confirming that the server was exploitable.

How the Issue Was Fixed

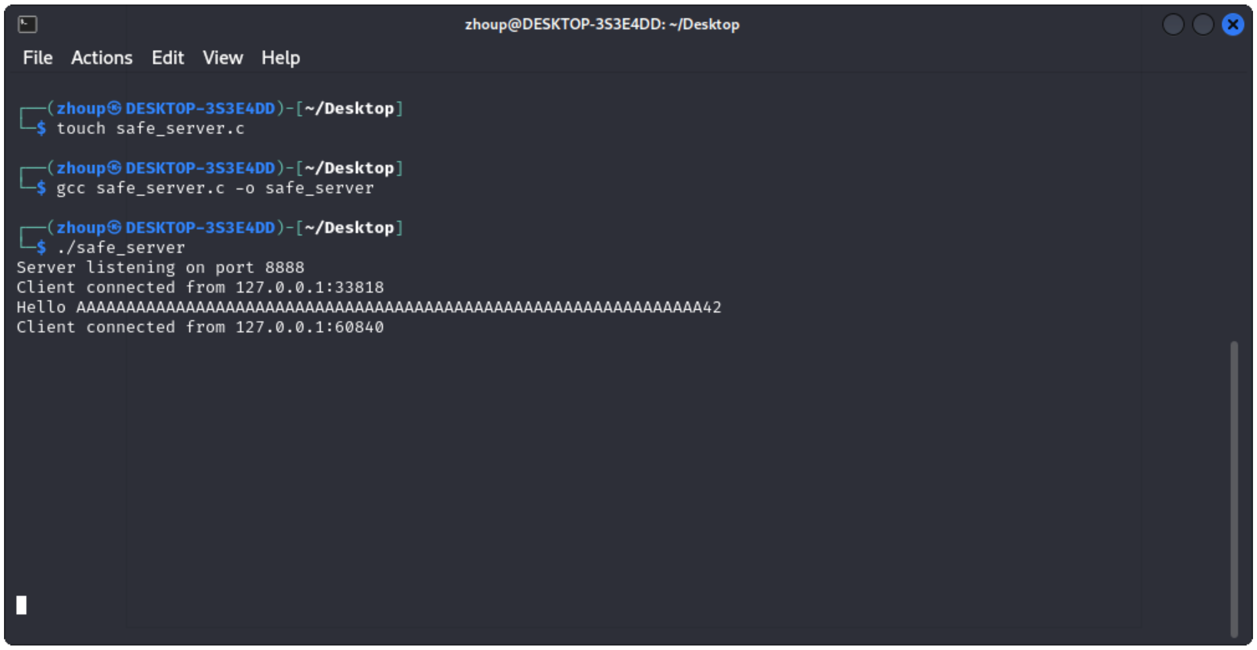
To eliminate the buffer overflow, I modified the code to use **`strncpy()`** instead of `strcpy()`, ensuring that only the allowed number of bytes is copied.

```
void secure_function(char *buffer) {
        char name[64];
int num = 42;
        strncpy(name, buffer, sizeof(name) - 1);  // Copying with bounds checking
name[sizeof(name) - 1] = '\0';  // Null-terminate to prevent overflows
printf("Hello %s%d\n", name, num);
}
```

Additional improvements:

- Replaced `vulnerable_function()` with `secure_function()` to prevent overflows.
- Added null-termination to ensure safe string handling.
- Updated main server logic to use `secure_function()` instead of the vulnerable function.

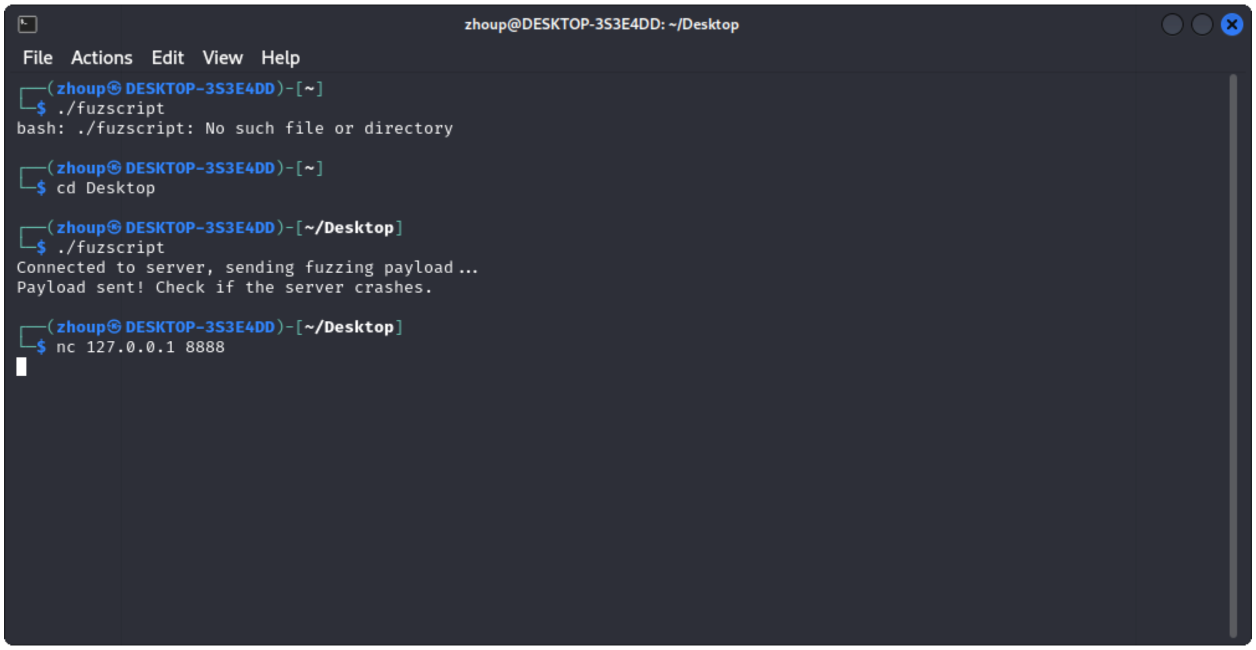
Demonstrating That Fuzzing No Longer Works



```
┌──(zhoup㉿DESKTOP-3S3E4DD)-[~/Desktop]
└─$ touch safe_server.c

┌──(zhoup㉿DESKTOP-3S3E4DD)-[~/Desktop]
└─$ gcc safe_server.c -o safe_server

┌──(zhoup㉿DESKTOP-3S3E4DD)-[~/Desktop]
└─$ ./safe_server
Server listening on port 8888
Client connected from 127.0.0.1:33818
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA42
Client connected from 127.0.0.1:60840
```



```
┌──(zhoup㉿DESKTOP-3S3E4DD)-[~]
└─$ ./fuzscript
bash: ./fuzscript: No such file or directory

┌──(zhoup㉿DESKTOP-3S3E4DD)-[~]
└─$ cd Desktop

┌──(zhoup㉿DESKTOP-3S3E4DD)-[~/Desktop]
└─$ ./fuzscript
Connected to server, sending fuzzing payload ...
Payload sent! Check if the server crashes.

┌──(zhoup㉿DESKTOP-3S3E4DD)-[~/Desktop]
└─$ nc 127.0.0.1 8888
```

After implementing the fix:

1. **Re-ran the fuzzing script** with the same payload (200+ "A" characters).
2. **Server remained stable** – no crash or segmentation fault.

Why This Solution Works

- The **buffer overflow vulnerability is eliminated** because `strncpy()` enforces a size limit.
- Null termination prevents **data corruption** from overflowing into adjacent memory.
- Even if an attacker **sends an excessively long input**, the server will safely handle it without crashing.