Extending Nyx-Net: A Standalone API for Grammar-Based Fuzzing in Stateful Systems

Songyuan Bo, Miftahul Huq

Grad Sem in Computing Security
(CSEC 759.03)

Yinxi Liu

10/26/2024

Table of Contents:

1. Abstract

This paper investigates the current state of network fuzzing tools, focusing on Nyx-Net and its limitations in handling complex, grammar-driven input structures. To address these challenges, we present a Grammar-Based Packet Generator written in Python 3, designed to enhance fuzzing processes by offering improved efficiency, accuracy, and automation. Our project integrates seamlessly with fuzzers such as Nyx-Net via APIs, enabling real-time feedback-driven packet generation for targeted testing. The paper provides a detailed overview of the system’s design, implementation, and potential for future improvements, along with the GitHub link for access and collaboration. By merging the randomness of traditional fuzzing with grammar-based precision, this tool represents a significant step forward in advancing network fuzzing methodologies.

2. Introduction

Fuzz testing is a form of dynamic analysis of a software and one of the main approaches for vulnerability discovery, especially in recent years and for complex domains containing many states, such as network services and Inter-Process Communication (IPC) systems. That’s where Nyx-Net comes in, a snapshot-based fuzzing tool, capable of performing efficient fuzzing for stateful and complicated systems. By combining hypervisor-based snapshots, selective network emulation, and incremental state resets, Nyx-Net has shown remarkable improvements in performance, yielding up to 300x faster testing throughput and 70% better coverage compared to state-of-the-art fuzzers.

Despite these advances, Nyx-Net is no different from most other fuzzing tools in that it relies mostly on generative approaches for input generation. While such techniques are powerful in certain contexts, they fail when the input structures are complex, such as those defined by context-free grammars or protocol specifications. Nyx-Net requires users to save captured network packets as “.pcap” files in advance to serve as seeds. Using these seed files, Nyx-Net generates various mutations to simulate different network conditions and packet behaviors, enabling comprehensive testing. Like other fuzzers, the randomness of the mutations can sometimes lead to suboptimal testing results.

To further enhance the efficiency and accuracy of Nyx-Net, we developed a Fuzzer helper program - Grammar-Based Packet Generator with API using Python to assist in dynamically generating packets. This stand-alone program will be able to interact with fuzzers such as Nyx-Net through APIs, receiving real-time feedback on the results of each testing iteration. Based on this feedback, it will determine the focus for the next set of test packets, ensuring a more targeted and effective fuzzing process.

When fuzzers are combined with our program, it effectively merges the benefits due to the randomness property with targeted directionality. Theoretically, this combination increases the likelihood of the fuzzer uncovering potential issues, making it easier and faster to identify problems. Additionally, it enhances the automation aspect, making the testing process more efficient and effective.

3. Related Work

3.1 Snapshot-Based Fuzzing Approaches

Snapshot-based fuzzing, as exemplified by Nyx-Net, is a big leap in fuzzing complex and stateful systems. With the leveraging of hypervisor-based snapshots and incremental resets, Nyx-Net achieves a very good throughput and coverage on targets such as network servers, IPC mechanisms, and even complex applications like Firefox. However, Nyx-Net primarily employs generative fuzzing techniques that limit its capability to effectively handle grammar-driven input formats.

Other snapshot-based fuzzers, like Agamotto, focus on efficiency via incremental snapshot mechanisms and target kernel drivers or low-level system components. While these tools achieve impressive results, they lack support for structured, grammar-based input generation that is crucial when testing systems with complex protocols or input dependencies.

3.2 Grammar-Based Fuzzing

Grammar-based fuzzing puts much emphasis on generating inputs that conform to certain grammar or protocol rules. Tools like AFLsmart and Nautilus are designed to handle structured inputs by using pre-defined grammars or context-free grammars to generate valid test cases. For example, AFLsmart embeds grammar specifications through Peach Pit files, while Nautilus uses a purely generative approach by means of context-free grammars.

However, all of these tools are invariably restricted by the strong binding of the grammar capabilities to specific fuzzing frameworks. For instance, AFLsmart requires very detailed specifications of grammars, which cannot be generalized easily for different fuzzing scenarios. Furthermore, the tools are not modularized, and thus it's very hard to integrate grammar-based fuzzing into Nyx-Net or other active state-of-the-art fuzzers.

3.3 Challenges in Network Fuzzing

Fuzzing network services adds more complexity due to the statefulness of protocols and the dependency between sequential inputs. Some new tools, such as AFLnet, have been proposed to extend the coverage-guided fuzzing of AFL into network protocols. However, AFLnet has poor scalability and efficiency when handling complex message exchanges or stateful interactions. Moreover, AFLnet does not support grammar-based input generation intrinsically, further limiting its applicability in complex protocol fuzzing.

3.4 Application Programming Interface

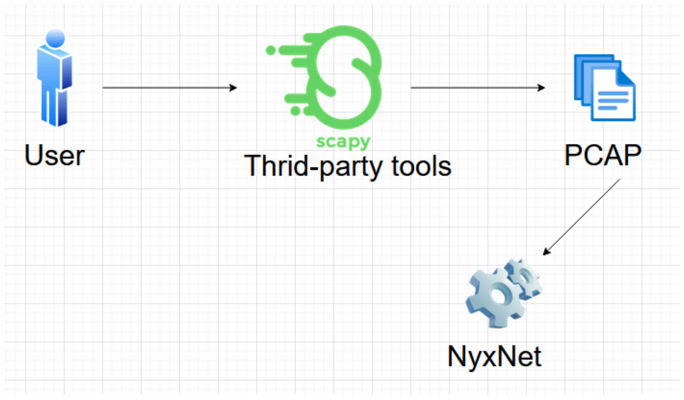
An Application Programming Interface (API) is a set of rules and protocols that allows different software applications to communicate with each other. It acts as a bridge, enabling one application to request and exchange data or services from another. APIs simplify integration, promote reusability, and allow developers to build complex systems by leveraging existing functionalities without needing to understand their internal workings.

4. System design

Considering factors such as compatibility, readability, and the ease of modifying the source code, we have decided to use Python 3 to develop a Grammar-Based Packet Generator with API. Python 3's widespread adoption and extensive libraries make it an ideal choice for ensuring compatibility with modern systems, while its clear syntax enhances accessibility for others to understand and contribute to the codebase or modifying it according to one’s specific needs.

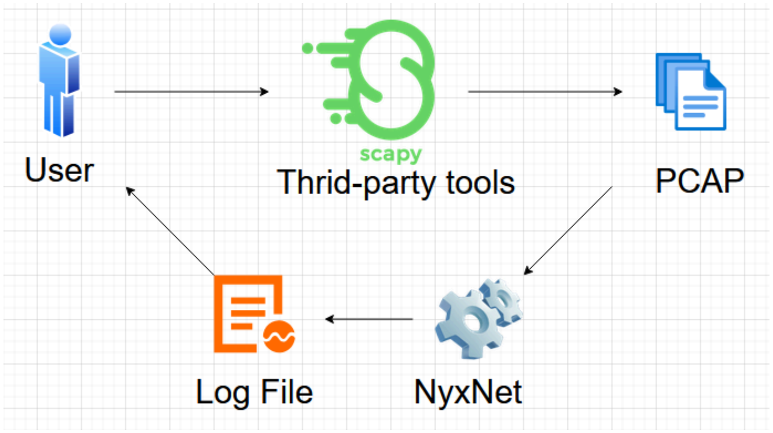
4.1 Functionality

The primary function of our packet generator is grammar-based packet generation, enabling the random creation of packets that adhere to specified grammatical rules. Such packets can be TCP, UDP, ICMP, or it can be as simple as some HTTP Requests. Additionally, our generator can interact in real-time with the calling fuzzer. Whenever a packet fails the fuzzer's tests in an iteration, the generator can receive the test results to decide whether to generate more similar packets for focused testing.



<Figure 1: Traditional Fuzzing process>

As shown in Figure 1, the traditional fuzzing process requires users to use third-party tools such as scapy or Wireshark to generate/capture some packets and store them as a PCAP file as seed before starting the testing of the Program Under Test (PUT).



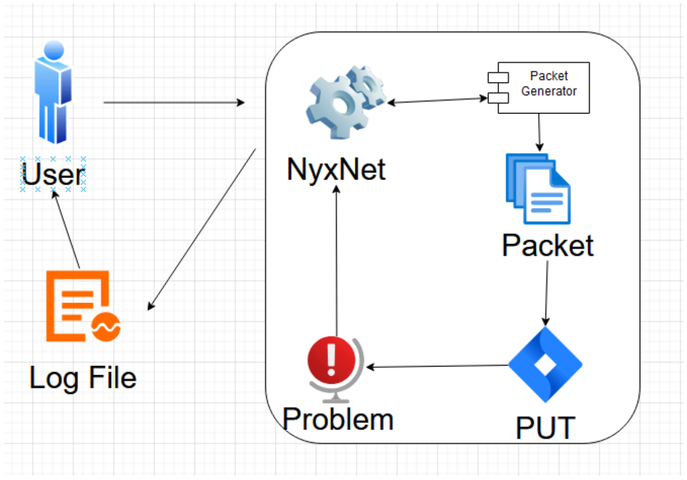
<Figure 2: Traditional Fuzzing process 2>

As shown in Figure 2, the fuzzer uses the seed to generate multiple rounds of random mutations to test the Program Under Test (PUT). Only after all mutations and iterations are completed does the fuzzer generate a report, which is saved as a log file.

If a user wants to test a web server (e.g., Apache) to check for potential issues (such as bugs or crashes) when processing HTTP requests, they first need to obtain some HTTP request packets as seeds. However, during this process, the user might fail to generate a sufficient variety of seeds due to incomplete considerations—for example, not generating seeds that include a specific HTTP request method.

Another issue is that traditional fuzzers cannot guarantee that the mutations they produce are syntactically valid HTTP requests. In other words, the fuzzer might generate requests that are inherently malformed, leading to the Program Under Test (PUT) failing several tests which cause false positives.

Now, suppose the web server has a vulnerability when handling the POST method. During the fuzzing process, the user would remain unaware of this and would have no way to focus the tests specifically on this method. The user would only realize the issue after all the iterations of the fuzzer are complete and a report in the log file highlights it. At that point, the user would need to generate new seeds for the POST method and initiate another round of targeted testing, further delaying the process.



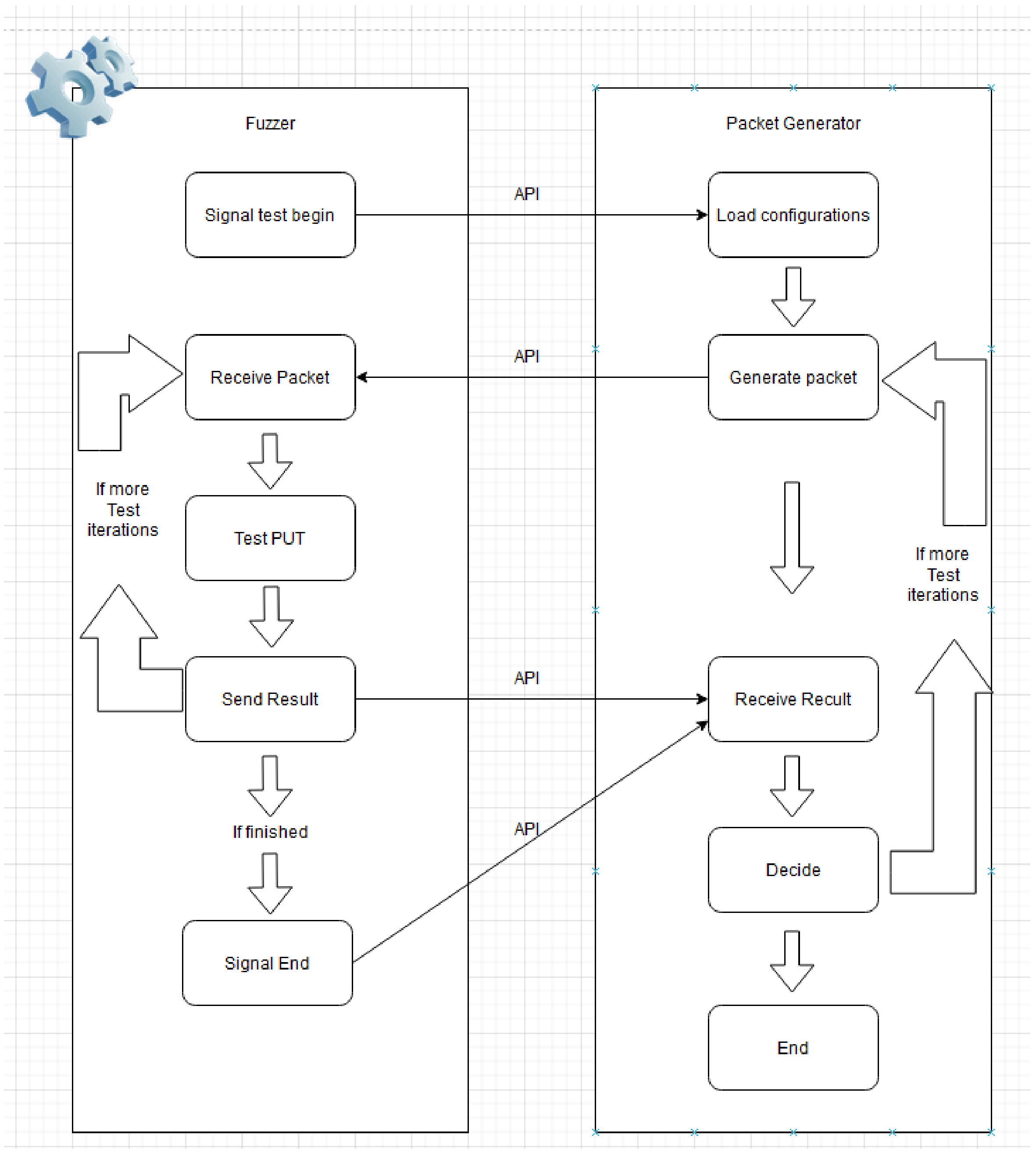
<Figure 3: Improved Fuzzing process>

As illustrated in Figure 3, when the fuzzer integrates with our program via APIs for testing, the user no longer needs to provide seeds (e.g., pcap files). Our program ensures that all scenarios (such as the uncommon HTTP request methods mentioned previously) are accounted for. Furthermore, it guarantees that all randomly generated packets are syntactically valid, significantly reducing the likelihood of false positives.

Additionally, whenever the PUT fails due to a specific HTTP request method—such as in our earlier example where the web server failed when handling the POST method—the fuzzer can directly notify our generator. In response, our generator will immediately create additional similar packets to conduct focused testing, improving both efficiency and accuracy in uncovering potential issues.

The best part is that this entire process requires no manual intervention from the user. Everything is fully automated—users only need to start the test and wait for the results. They can then see which packets caused the web server to fail and review the outcomes of the focused testing, all without any additional effort on their part.

4.2 Execution Flow



<Figure 4: Execution flow>

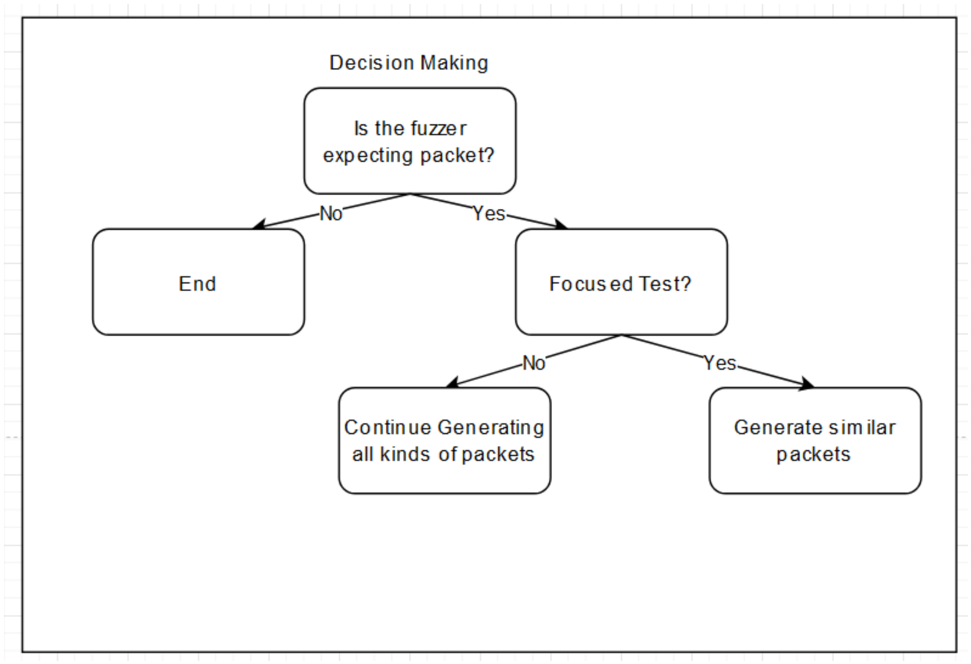
Figure 4 illustrates the workflow between the fuzzer and our packet generator. When the testing begins, the fuzzer sends a start signal via the API, and our program loads the initial configurations. These configurations specify details such as the type of packets to generate (e.g., HTTP requests), the number of packets to generate per kind, and the depth of focused testing (the number of additional similar packets to generate) in case a test fails.

Assume the following information as the configuration for the example:

- Type of packet = HTTP Request
- Number of packets per kind = 3
- Focused Test depth = 5

Once these configurations are fully loaded, the program begins packet generation. Using the example of testing a web server, our program will generate packets for every HTTP request method based on the configurations. For instance, if the configuration specifies generating three packets per kind, the program will create three packets for each HTTP method. Each packet is then sent to the fuzzer via the API, and the fuzzer uses it to test the Program Under Test (PUT). The fuzzer returns the test results to our program through the API.

Upon receiving the results, our program makes decisions (will be discussed more in detail later) about whether to continue generating packets and, if so, what type of packets to generate. This iterative process continues until all iterations are completed, at which point the program exits.


Decision Making
Is the fuzzer expecting packet?
No
Yes
End
Focused Test?
No
Yes
Continue Generating all kinds of packets
Generate similar packets

<Figure 4: Decision Making>

As shown in Figure 5, during each test iteration, our program decides whether to generate packets and, if so, which type of packets and whether to perform focused testing.

Returning to our example of a web server, our program generates HTTP request packets. This class of packets includes various types (or kinds) of packets, corresponding to HTTP request methods. These methods include, but are not limited to, \`GET\`, \`POST\`, \`PUT\`, \`DELETE\`, \`HEAD\`, and \`OPTIONS\`.

Our program sequentially iterates through each method, randomly generating 3 (as specified in the configuration) valid packets per method. If any packet fails the test, it triggers focused testing, where an additional 5 (as specified in the configuration) packets for that specific method are generated on top of the initial 3. The program then proceeds to test the next method, ensuring all configurations are systematically applied.

Since our project implements the Application Interface (API), it can easily be combined with tools such as Nyx-Net. Unlike other tools, which closely integrate grammar-based capabilities into a specific framework, our API is modular and reusable. It not only enhances Nyx-Net with more sophisticated handling of grammar-based inputs but also generalizes to a solution that can be widely adopted by other fuzzers. This flexibility provides a complementary approach to existing work, and bridges the gap between generative and grammar-based fuzzing for stateful and complex systems.

5. Implementation

Our project consists of three main components:

1. Prototype (Proof of Concept):
This is the first version which serves as a proof of concept, primarily demonstrating the functionality of our program. In this version, there is no actual API interaction. Instead, we manually input `1` and `0` to simulate the fuzzer's test results for each iteration.

2. HTTP Packet Generator:
In this version, we introduced API integration into our generator. Due to time constraints, we focused only on generating HTTP request packets, specifically implementing 6 types (or methods) within this class. Theoretically, our generator should be capable of generating a wide variety of packets, including but not limited to common ones such as TCP, UDP, ICMP, DHCP, HTTP Request/Reply, DORA packets, and DNS. In the future, we plan to continuously improve and expand our project (Please refer to the "Future Works" section of this paper for more details).

3. Fake Fuzzer:
We developed a fake fuzzer that does not possess actual fuzzing capabilities but instead randomly simulates test results and interacts with our generator. This tool is designed to test the interaction and functionality of our generator.

5.1 Prerequisites

As mentioned earlier in this document, our project is developed using Python 3, specifically Python 3.10.11. Therefore, please ensure that your runtime environment has Python 3.10.11 or a later version installed.

Required Python libraries:

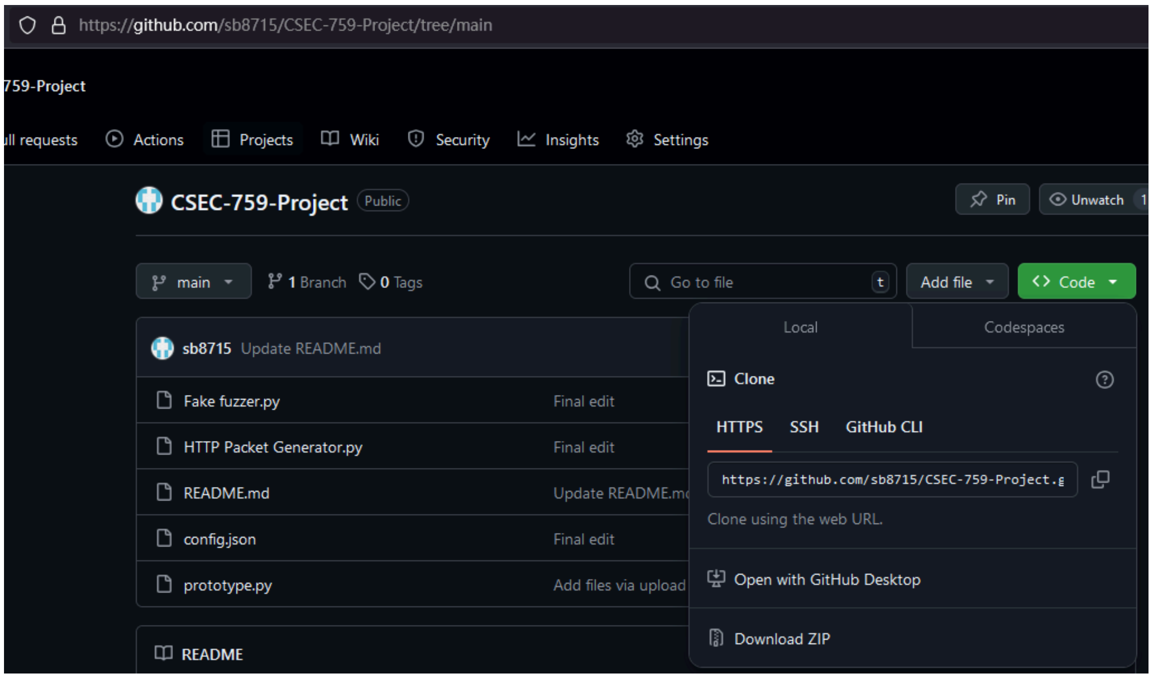
- requests
- Flask

Make sure to install the required libraries by running the following command:

```
pip install requests Flask
```

Make sure nothing is running using the port 5000 and it is available for the project python code to use (or modify the script to use a different port)

Go to our Github page and download/clone this project using this following link: https://github.com/sb8715/CSEC-759-Project/tree/main



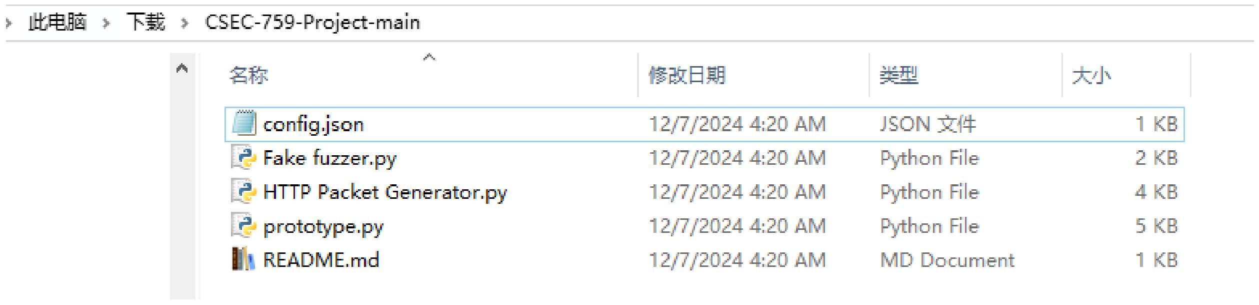
sb8715 Update README.md

Fake fuzzer.py	Final edit
HTTP Packet Generator.py	Final edit
README.md	Update README.md
config.json	Final edit
prototype.py	Add files via upload

```
https://github.com/sb8715/CSEC-759-Project.g
```

<Figure 5: Github Project Page>

Unzip/extract the project files:

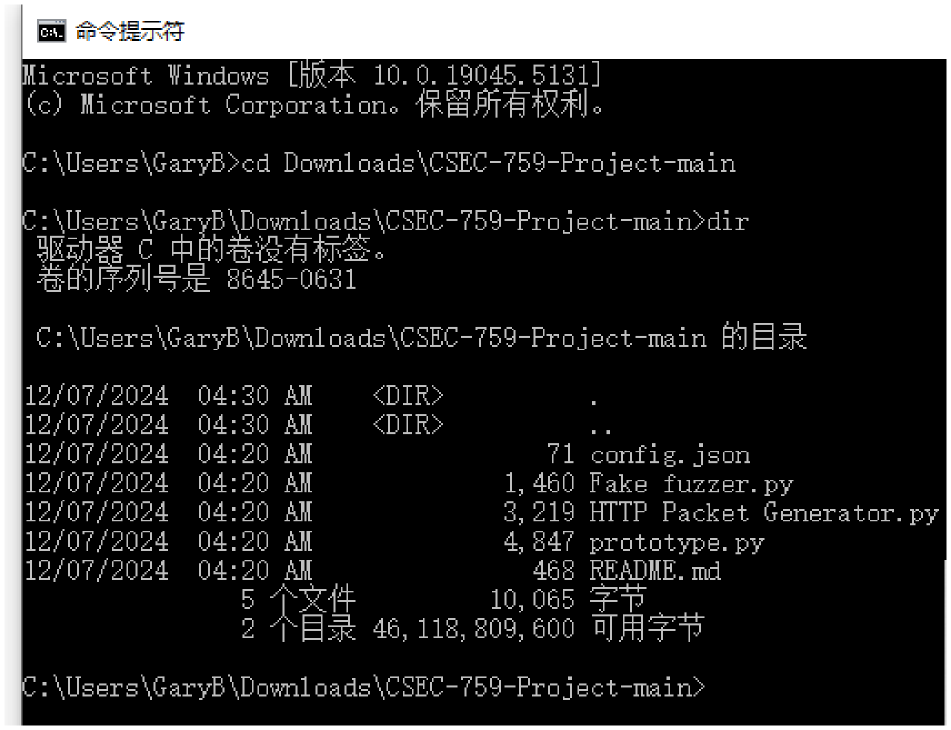


名称	修改日期	类型	大小
config.json	12/7/2024 4:20 AM	JSON 文件	1 KB
Fake fuzzer.py	12/7/2024 4:20 AM	Python File	2 KB
HTTP Packet Generator.py	12/7/2024 4:20 AM	Python File	4 KB
prototype.py	12/7/2024 4:20 AM	Python File	5 KB
README.md	12/7/2024 4:20 AM	MD Document	1 KB

<Figure 6: Project files>

5.2 Prototype

Step 1. Open a terminal/console and change your current working directory to our project directory:

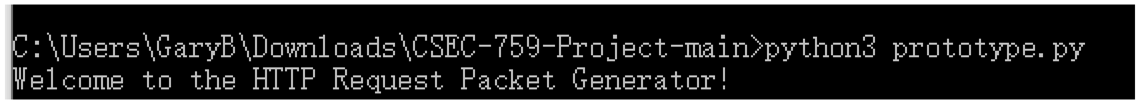


```
命令提示符
Microsoft Windows [版本 10.0.19045.5131]
(c) Microsoft Corporation。保留所有权利。

C:\Users\GaryB>cd Downloads\CSEC-759-Project-main

C:\Users\GaryB\Downloads\CSEC-759-Project-main>dir
 驱动器 C 中的卷没有标签。
 卷的序列号是 8645-0631

 C:\Users\GaryB\Downloads\CSEC-759-Project-main 的目录

12/07/2024  04:30 AM    <DIR>          .
12/07/2024  04:30 AM    <DIR>          ..
12/07/2024  04:20 AM                71 config.json
12/07/2024  04:20 AM             1,460 Fake fuzzer.py
12/07/2024  04:20 AM             3,219 HTTP Packet Generator.py
12/07/2024  04:20 AM             4,847 prototype.py
12/07/2024  04:20 AM               468 README.md
               5 个文件         10,065 字节
               2 个目录 46,118,809,600 可用字节

C:\Users\GaryB\Downloads\CSEC-759-Project-main>
```

<Figure 7: Step 1>

Step 2. Run the prototype by using the following command:

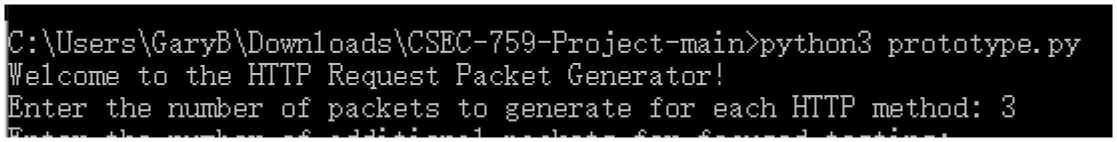
python3 prototype.py



```
C:\Users\GaryB\Downloads\CSEC-759-Project-main>python3 prototype.py
Welcome to the HTTP Request Packet Generator!
```

<Figure 8: Step 2>

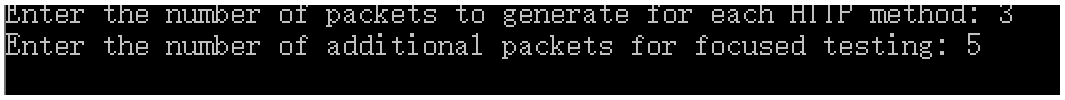
Step 3. Set the configuration for the number of packets to generate for each HTTP method, in this case we will enter 3.



```
C:\Users\GaryB\Downloads\CSEC-759-Project-main>python3 prototype.py
Welcome to the HTTP Request Packet Generator!
Enter the number of packets to generate for each HTTP method: 3
```

<Figure 9: Step 3>

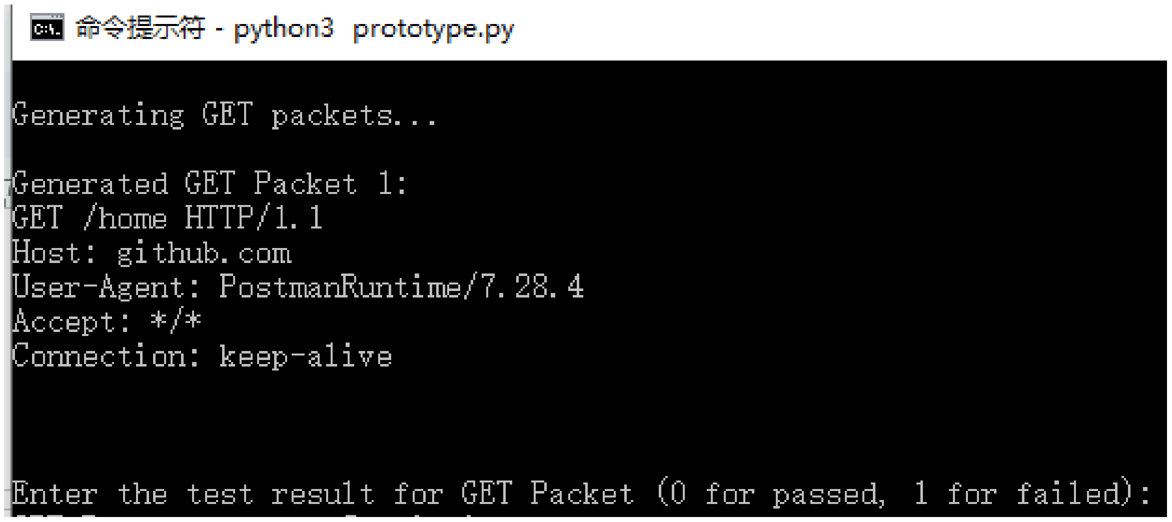
Step 4. Set the depth of focus test, in this case we will enter 5.



```
Enter the number of packets to generate for each HTTP method: 3
Enter the number of additional packets for focused testing: 5
```

<Figure 10: Step 4>

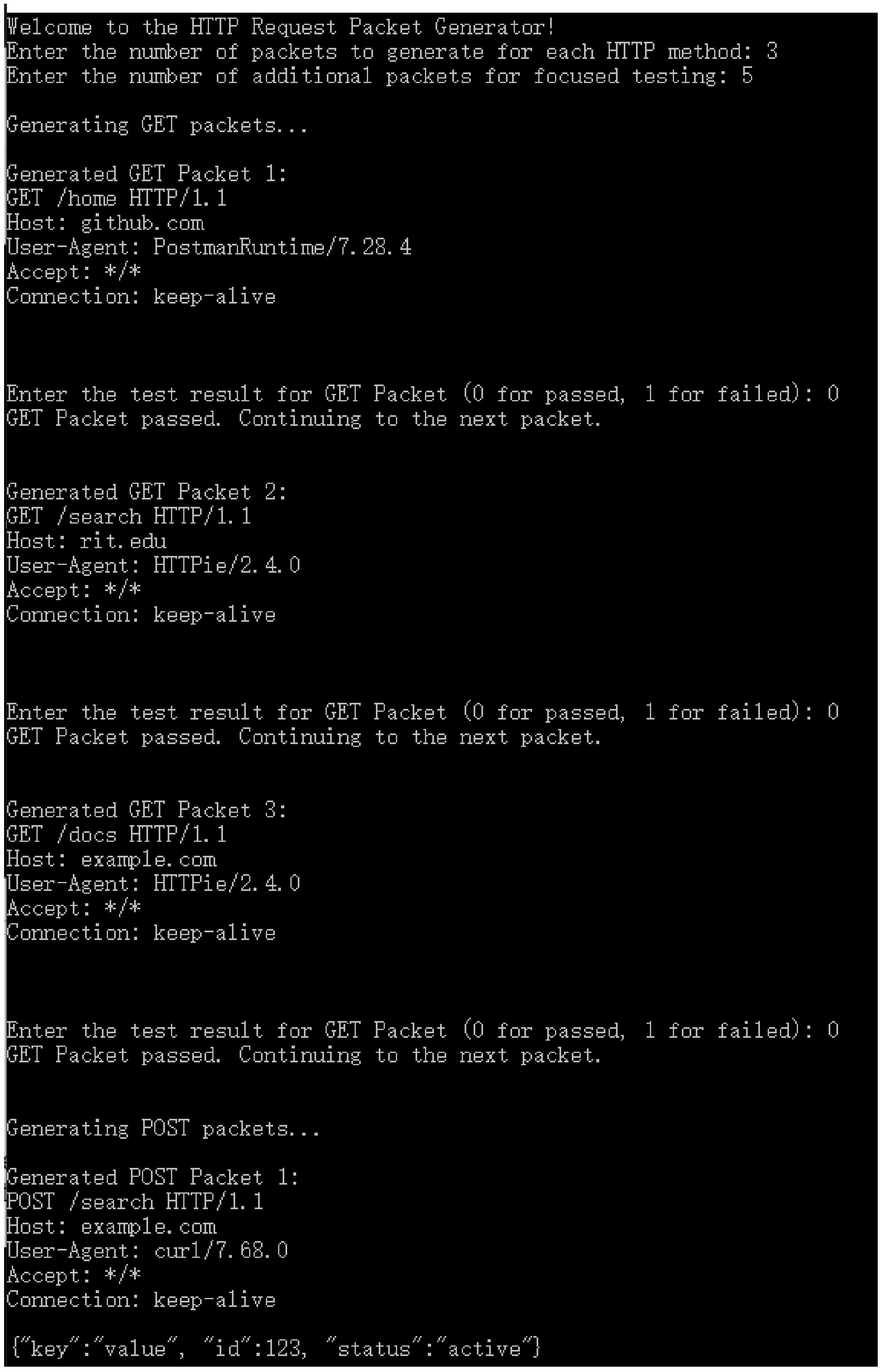
Step 5. Now the program will generate packets and display them, after generation, the program will pause and wait for us to give the test result.



```
Generating GET packets...

Generated GET Packet 1:
GET /home HTTP/1.1
Host: github.com
User-Agent: PostmanRuntime/7.28.4
Accept: */*
Connection: keep-alive



Enter the test result for GET Packet (0 for passed, 1 for failed):
```

<Figure 11: Step 5>

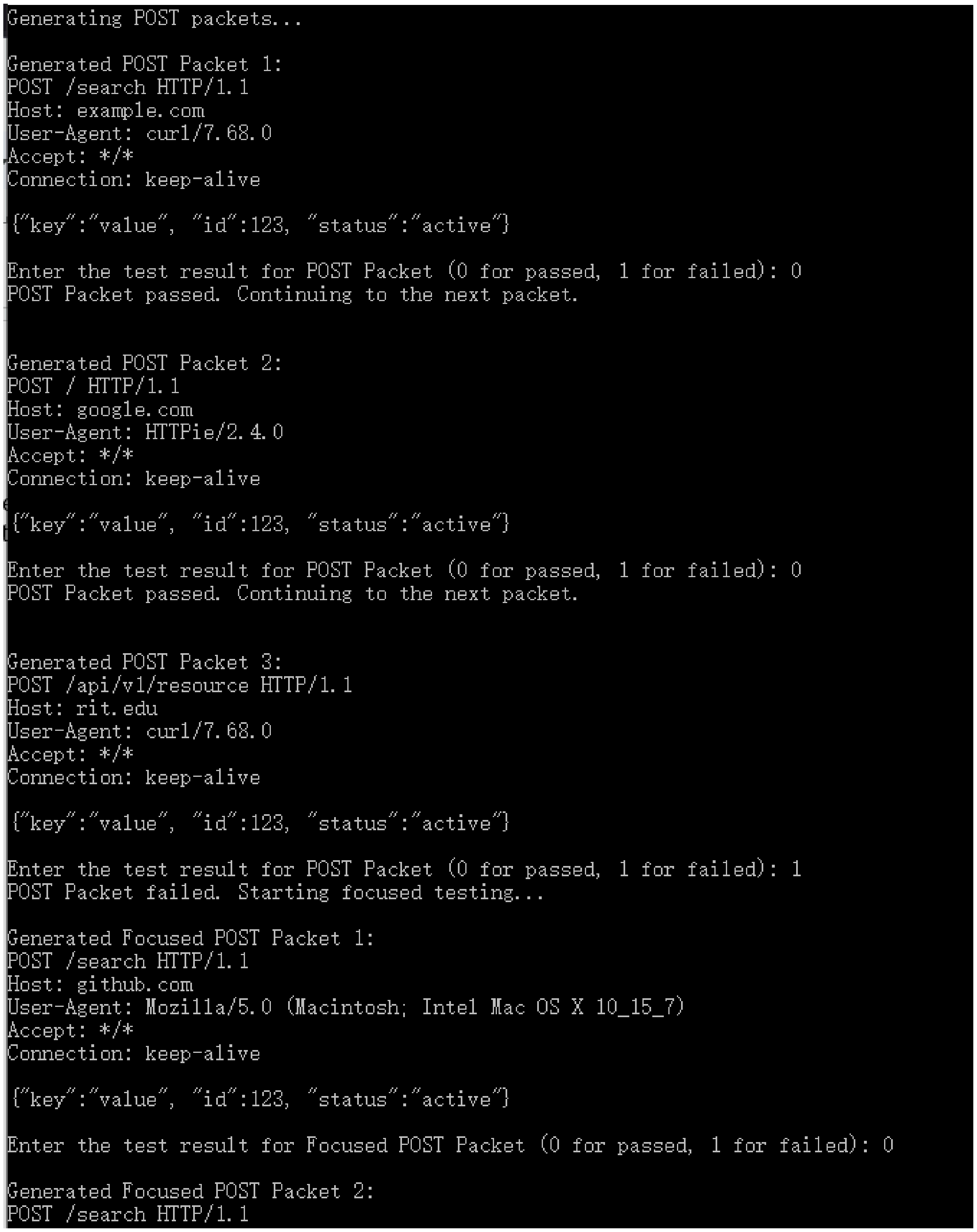
Step 6. Assuming the Web Server only have problems handling the POST method, we will let everything else pass. Pay attention to the detail of the packets generated, their payloads are randomly selected such as the required resources, Host, User-Agent, etc. As figure 12 shown below, since all GET Packet passed the test, no focus test for GET is triggered and we can see that the program only generated 3 packets for the GET method as it was configured to do so. Then it moves on to the next method which is the POST method.



```
Welcome to the HTTP Request Packet Generator!
Enter the number of packets to generate for each HTTP method: 3
Enter the number of additional packets for focused testing: 5

Generating GET packets...

Generated GET Packet 1:
GET /home HTTP/1.1
Host: github.com
User-Agent: PostmanRuntime/7.28.4
Accept: */*
Connection: keep-alive



Enter the test result for GET Packet (0 for passed, 1 for failed): 0
GET Packet passed. Continuing to the next packet.


Generated GET Packet 2:
GET /search HTTP/1.1
Host: rit.edu
User-Agent: HTTPie/2.4.0
Accept: */*
Connection: keep-alive



Enter the test result for GET Packet (0 for passed, 1 for failed): 0
GET Packet passed. Continuing to the next packet.


Generated GET Packet 3:
GET /docs HTTP/1.1
Host: example.com
User-Agent: HTTPie/2.4.0
Accept: */*
Connection: keep-alive



Enter the test result for GET Packet (0 for passed, 1 for failed): 0
GET Packet passed. Continuing to the next packet.


Generating POST packets...

Generated POST Packet 1:
POST /search HTTP/1.1
Host: example.com
User-Agent: curl/7.68.0
Accept: */*
Connection: keep-alive

{"key":"value", "id":123, "status":"active"}
```

<Figure 12: Step 6>

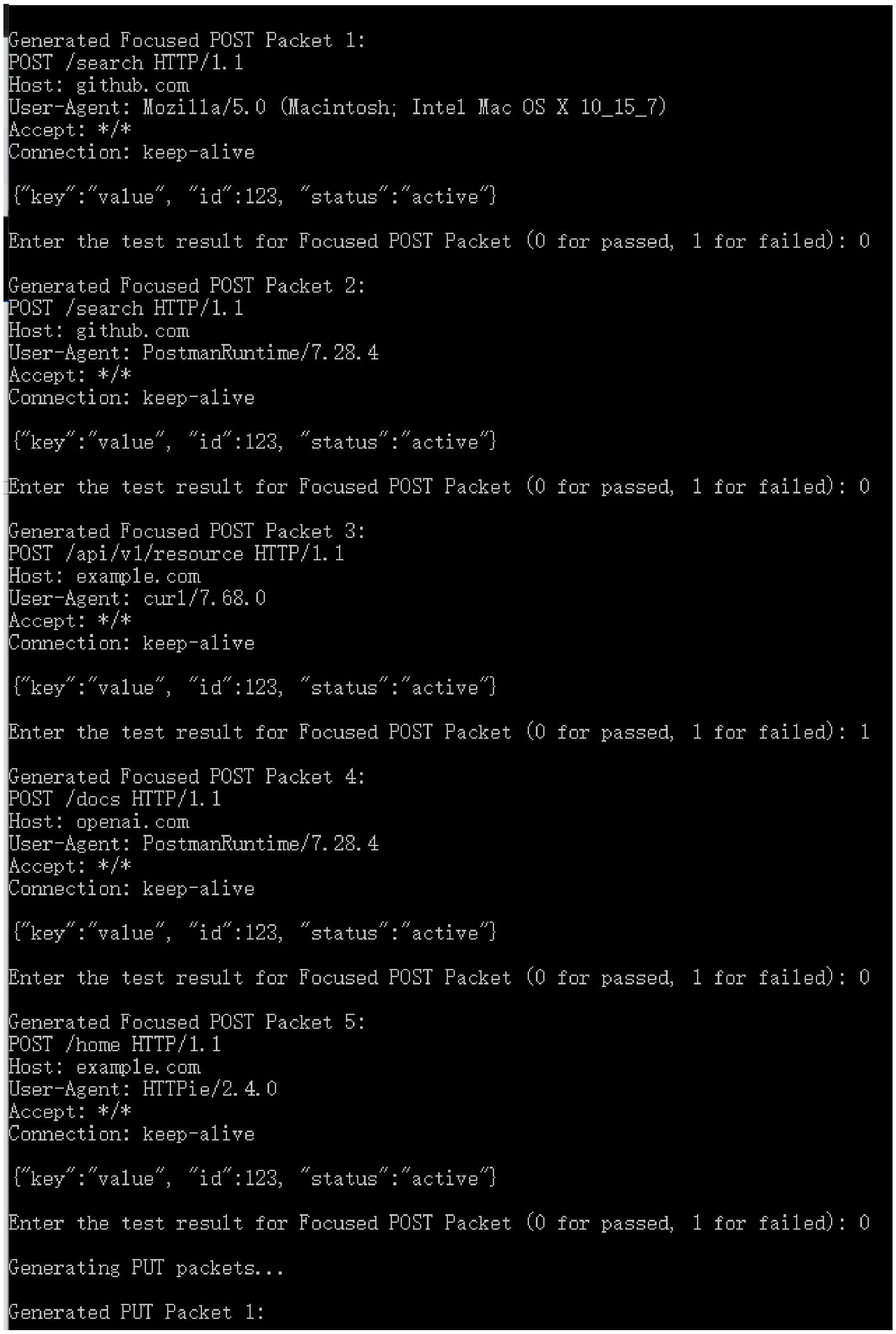
Step 7. As figure 13 shown below, we can see that the first two tests for POST method passed but the last one failed, and it only takes one fail for the program to trigger the focus test.



```
Generating POST packets...

Generated POST Packet 1:
POST /search HTTP/1.1
Host: example.com
User-Agent: curl/7.68.0
Accept: */*
Connection: keep-alive

{"key":"value", "id":123, "status":"active"}

Enter the test result for POST Packet (0 for passed, 1 for failed): 0
POST Packet passed. Continuing to the next packet.


Generated POST Packet 2:
POST / HTTP/1.1
Host: google.com
User-Agent: HTTPie/2.4.0
Accept: */*
Connection: keep-alive

{"key":"value", "id":123, "status":"active"}

Enter the test result for POST Packet (0 for passed, 1 for failed): 0
POST Packet passed. Continuing to the next packet.


Generated POST Packet 3:
POST /api/v1/resource HTTP/1.1
Host: rit.edu
User-Agent: curl/7.68.0
Accept: */*
Connection: keep-alive

{"key":"value", "id":123, "status":"active"}

Enter the test result for POST Packet (0 for passed, 1 for failed): 1
POST Packet failed. Starting focused testing...

Generated Focused POST Packet 1:
POST /search HTTP/1.1
Host: github.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
Accept: */*
Connection: keep-alive

{"key":"value", "id":123, "status":"active"}

Enter the test result for Focused POST Packet (0 for passed, 1 for failed): 0

Generated Focused POST Packet 2:
POST /search HTTP/1.1
```

<Figure 13: Step 7>

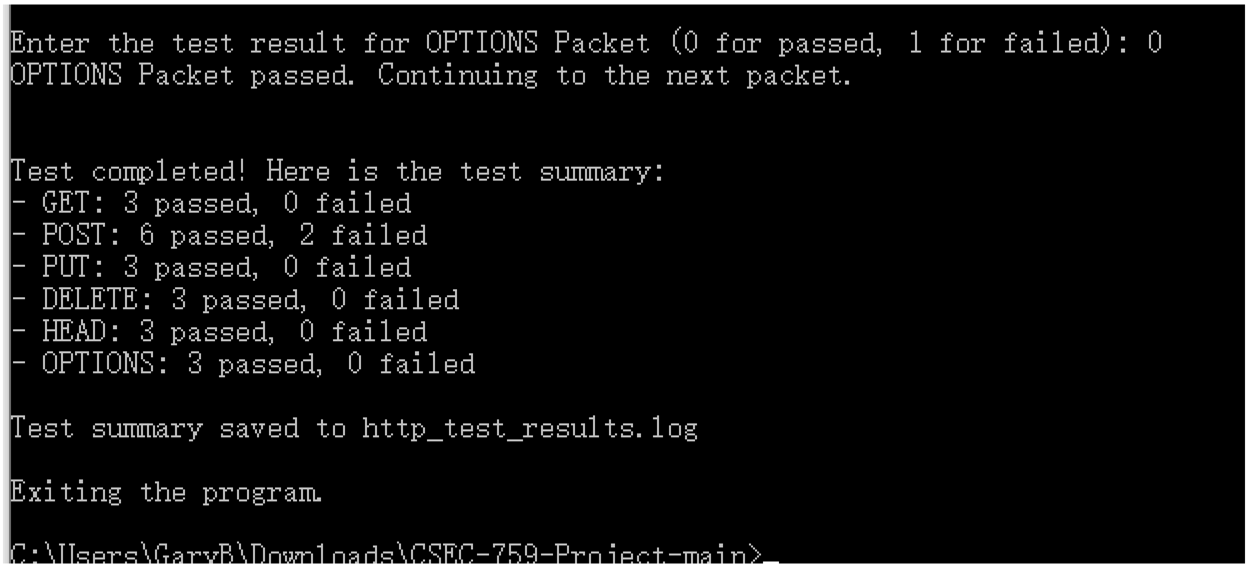
Step 8. As figure 14 shown below, once the focused test is triggered, it will generate 5 extra packets for that particular method failed for additional test iterations before moving on to the next method.



```
Generated Focused POST Packet 1:
POST /search HTTP/1.1
Host: github.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
Accept: */*
Connection: keep-alive

{"key":"value", "id":123, "status":"active"}

Enter the test result for Focused POST Packet (0 for passed, 1 for failed): 0

Generated Focused POST Packet 2:
POST /search HTTP/1.1
Host: github.com
User-Agent: PostmanRuntime/7.28.4
Accept: */*
Connection: keep-alive

{"key":"value", "id":123, "status":"active"}

Enter the test result for Focused POST Packet (0 for passed, 1 for failed): 0

Generated Focused POST Packet 3:
POST /api/v1/resource HTTP/1.1
Host: example.com
User-Agent: curl/7.68.0
Accept: */*
Connection: keep-alive

{"key":"value", "id":123, "status":"active"}

Enter the test result for Focused POST Packet (0 for passed, 1 for failed): 1

Generated Focused POST Packet 4:
POST /docs HTTP/1.1
Host: openai.com
User-Agent: PostmanRuntime/7.28.4
Accept: */*
Connection: keep-alive

{"key":"value", "id":123, "status":"active"}

Enter the test result for Focused POST Packet (0 for passed, 1 for failed): 0

Generated Focused POST Packet 5:
POST /home HTTP/1.1
Host: example.com
User-Agent: HTTPie/2.4.0
Accept: */*
Connection: keep-alive

{"key":"value", "id":123, "status":"active"}

Enter the test result for Focused POST Packet (0 for passed, 1 for failed): 0

Generating PUT packets...

Generated PUT Packet 1:
```

<Figure 14: Step 8>

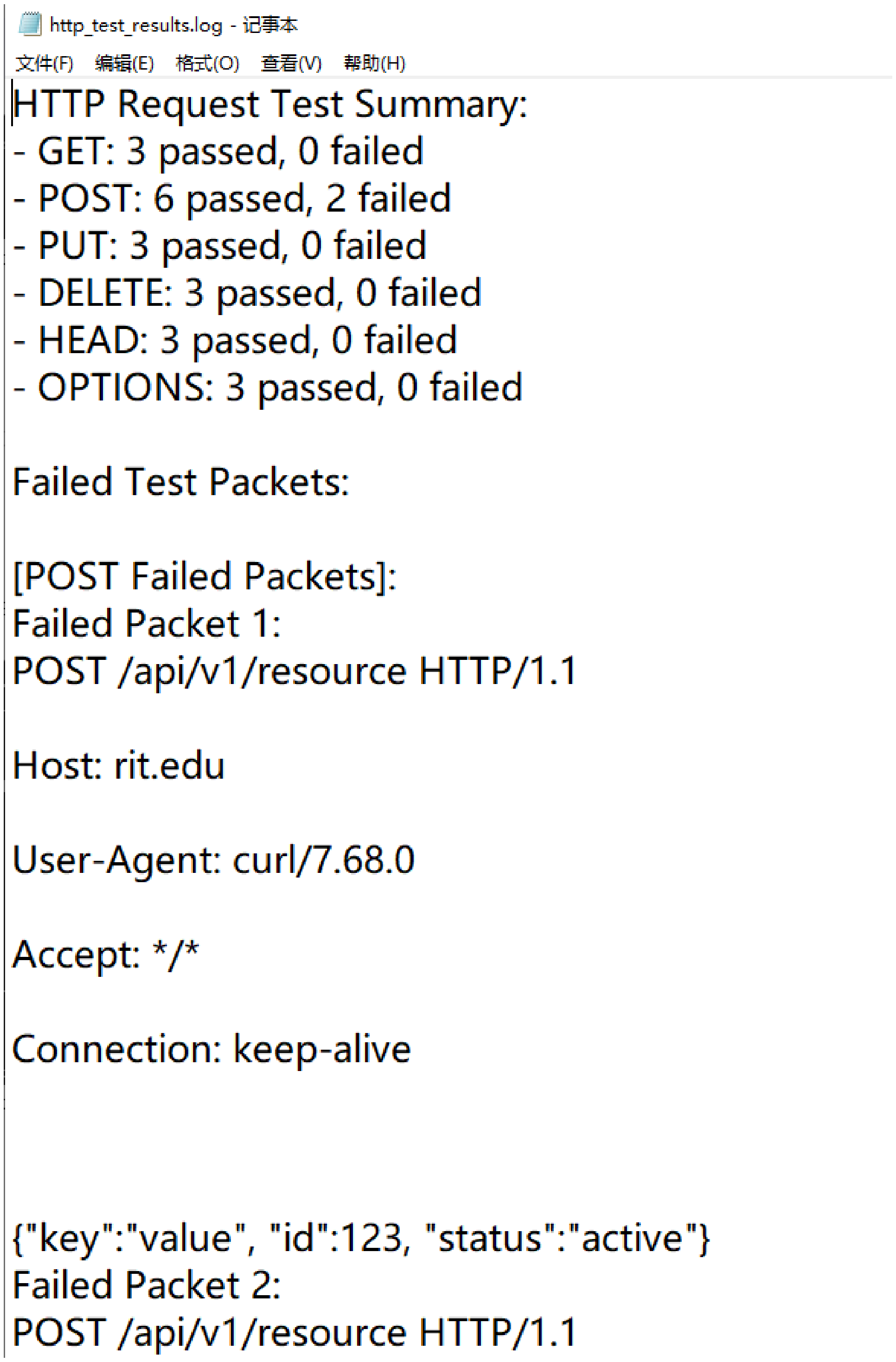
Step 9. Once the entire test is finished, it will display a summary along with a log file recording the packets that caused the test to fail.



```
Enter the test result for OPTIONS Packet (0 for passed, 1 for failed): 0
OPTIONS Packet passed. Continuing to the next packet.


Test completed! Here is the test summary:
- GET: 3 passed, 0 failed
- POST: 6 passed, 2 failed
- PUT: 3 passed, 0 failed
- DELETE: 3 passed, 0 failed
- HEAD: 3 passed, 0 failed
- OPTIONS: 3 passed, 0 failed

Test summary saved to http_test_results.log

Exiting the program.

C:\Users\GaryB\Downloads\CSEC-759-Project-main>_
```

<Figure 15: Step 9>

Step 10. The log file will contain a summary of the entire test along with the failed packets as shown in Figure 16 below.

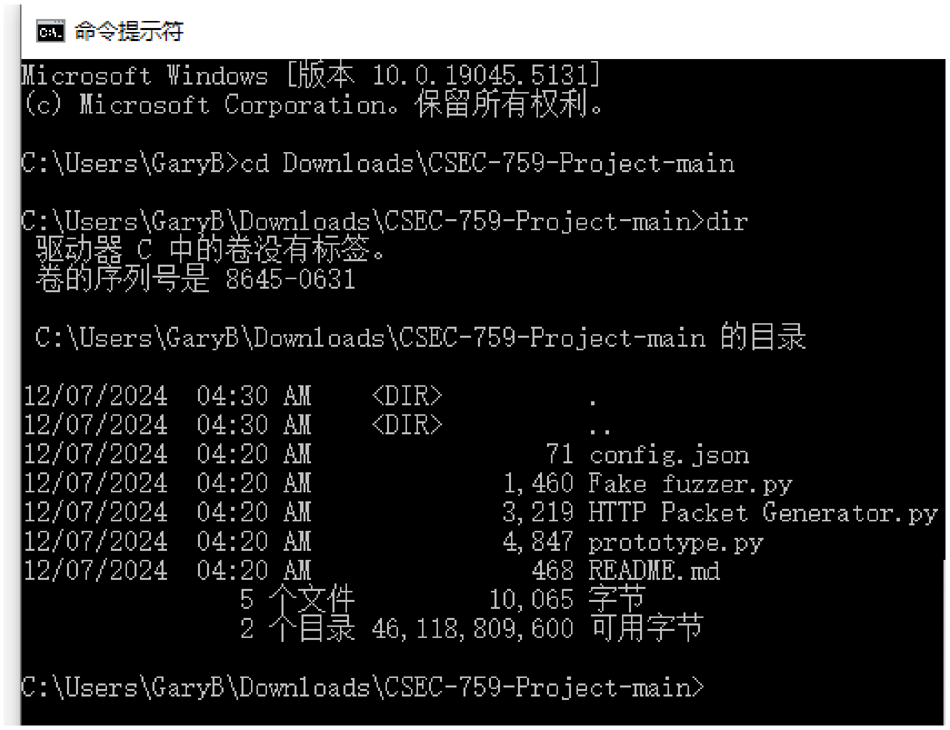

http_test_results.log - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
HTTP Request Test Summary:
- GET: 3 passed, 0 failed
- POST: 6 passed, 2 failed
- PUT: 3 passed, 0 failed
- DELETE: 3 passed, 0 failed
- HEAD: 3 passed, 0 failed
- OPTIONS: 3 passed, 0 failed

Failed Test Packets:

[POST Failed Packets]:
Failed Packet 1:
POST /api/v1/resource HTTP/1.1

Host: rit.edu

User-Agent: curl/7.68.0

Accept: */*

Connection: keep-alive



{"key":"value", "id":123, "status":"active"}
Failed Packet 2:
POST /api/v1/resource HTTP/1.1
```

<Figure 16: log file>

5.3 HTTP Packet Generator with API

Step 1. Open a terminal/console and change your current working directory to our project directory:

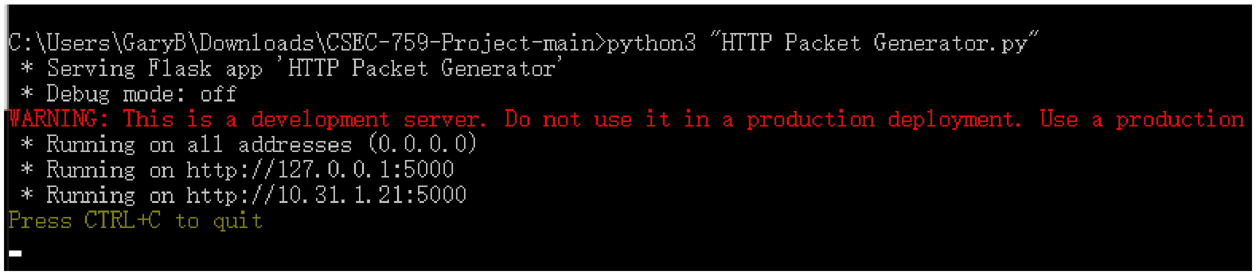


```
命令提示符
Microsoft Windows [版本 10.0.19045.5131]
(c) Microsoft Corporation。保留所有权利。

C:\Users\GaryB>cd Downloads\CSEC-759-Project-main

C:\Users\GaryB\Downloads\CSEC-759-Project-main>dir
 驱动器 C 中的卷没有标签。
 卷的序列号是 8645-0631

 C:\Users\GaryB\Downloads\CSEC-759-Project-main 的目录

12/07/2024  04:30 AM    <DIR>          .
12/07/2024  04:30 AM    <DIR>          ..
12/07/2024  04:20 AM                71 config.json
12/07/2024  04:20 AM             1,460 Fake fuzzer.py
12/07/2024  04:20 AM             3,219 HTTP Packet Generator.py
12/07/2024  04:20 AM             4,847 prototype.py
12/07/2024  04:20 AM               468 README.md
               5 个文件         10,065 字节
               2 个目录 46,118,809,600 可用字节

C:\Users\GaryB\Downloads\CSEC-759-Project-main>
```

<Figure 17: Step 1>

Step 2. Run the generator by using the following command:

python3 HTTP Packet Generator.py

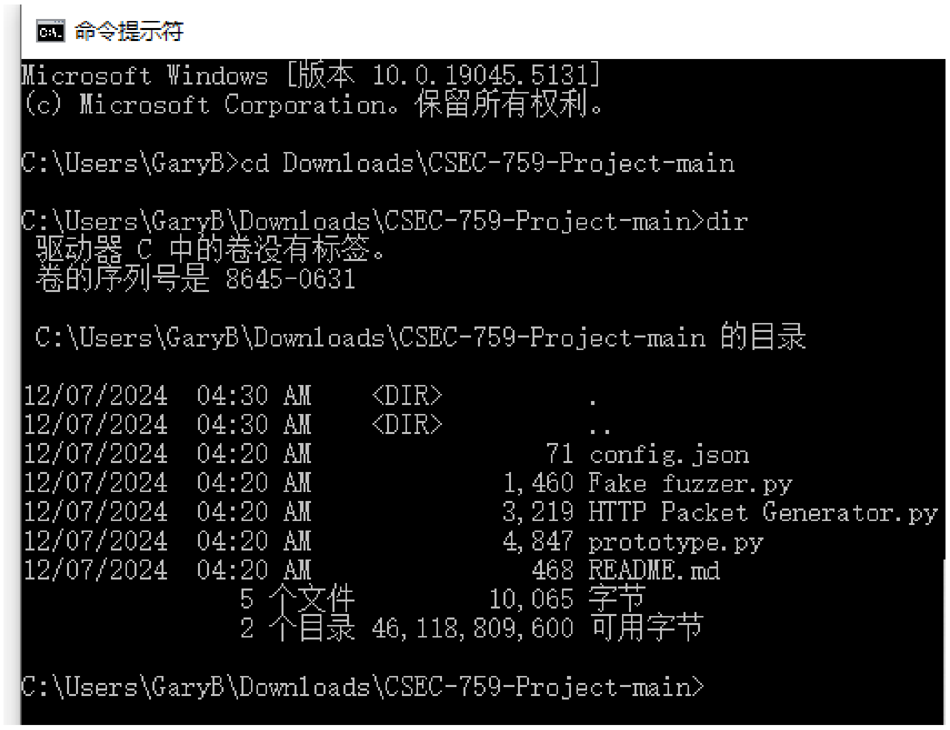


```
C:\Users\GaryB\Downloads\CSEC-759-Project-main>python3 "HTTP Packet Generator.py"
 * Serving Flask app 'HTTP Packet Generator'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://10.31.1.21:5000
Press CTRL+C to quit
```

<Figure 18: Step 2>

As Figure 18 is shown above, there will be a red warning about the server, it is completely normal, no need to worry about it for now. Right now our generator is operational and running on port 5000, awaiting for a fuzzer to call it via API.

Step 3. Open another terminal and change to our project directory.

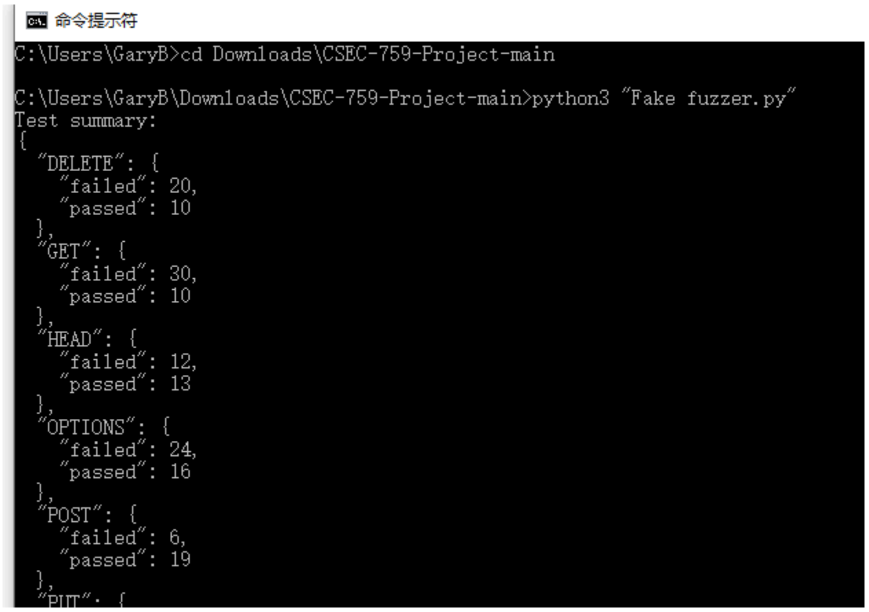


```
命令提示符
Microsoft Windows [版本 10.0.19045.5131]
(c) Microsoft Corporation。保留所有权利。

C:\Users\GaryB>cd Downloads\CSEC-759-Project-main

C:\Users\GaryB\Downloads\CSEC-759-Project-main>dir
 驱动器 C 中的卷没有标签。
 卷的序列号是 8645-0631

 C:\Users\GaryB\Downloads\CSEC-759-Project-main 的目录

12/07/2024  04:30 AM    <DIR>          .
12/07/2024  04:30 AM    <DIR>          ..
12/07/2024  04:20 AM                71 config.json
12/07/2024  04:20 AM             1,460 Fake fuzzer.py
12/07/2024  04:20 AM             3,219 HTTP Packet Generator.py
12/07/2024  04:20 AM             4,847 prototype.py
12/07/2024  04:20 AM               468 README.md
               5 个文件         10,065 字节
               2 个目录 46,118,809,600 可用字节

C:\Users\GaryB\Downloads\CSEC-759-Project-main>
```

<Figure 19: Step 3>

Step 4. Use the following command to run the fake fuzzer in order to test our generator:

python3 fake fuzzer.py



```
命令提示符
C:\Users\GaryB>cd Downloads\CSEC-759-Project-main

C:\Users\GaryB\Downloads\CSEC-759-Project-main>python3 "Fake fuzzer.py"
Test summary:
{
  "DELETE": {
    "failed": 20,
    "passed": 10
  },
  "GET": {
    "failed": 30,
    "passed": 10
  },
  "HEAD": {
    "failed": 12,
    "passed": 13
  },
  "OPTIONS": {
    "failed": 24,
    "passed": 16
  },
  "POST": {
    "failed": 6,
    "passed": 19
  },
  "PUT": {
```

<Figure 20: Step 4>

6. Future Works

6.1 Add more packet class and sub types support

As the above section, the current implementation highlights some limitations of our program. At this stage, it can only generate the HTTP request class and its six subtypes which are the methods. In the future, we plan to enhance our project by not only expanding the range of HTTP request methods but also supporting additional packet types, such as ICMP packets, DHCP DORA packets, DNS packets and so much more. Furthermore, we aim to enable the ability to encapsulate these packets into TCP and UDP layers.

We also plan to improve our program architecture by leveraging Python's class features. This will allow users to define their own custom packet classes and seamlessly integrate them into our generator, providing greater flexibility and extensibility for diverse testing needs.

The final version of our project will be able to receive the parameters/configurations of the packets required by the fuzzer through the API. This feature will be discussed in greater detail in Section 6.3 of this document.

6.2 Add more random choice variety to consider all scenarios

At this current time our project generates a limited variety of the values/payloads of the HTTP requests. For example as Figure 14 shown above, the “Host” generated are: example.com, google.com, github.com, openai.com. All of those “Host” values are strings/English characters. We will add more types of value such as numbers, and special characters other than just letters to the string. For example: http://www.hao123.com/.

We will also improve the capabilities of our generator to generate contents that are more complex while following RFC protocol documentations. For example, right now the Host value generated by our generator only has 2 parts: example.com, something.something. In future it will be able to generate more complex values such as: sh12345.gov.cn

6.3 Add more API interaction supports

Currently, our project supports four types of interactions with the fuzzer through the API: receiving start and end test signals, sending packets, and receiving test results.

In the future, we plan to add more API interaction capabilities, such as allowing the fuzzer to directly transmit the specific configuration of the packets it requires to our generator through the API. We will also take into account the workflows and methods of different fuzzers to make our project more flexible and user-friendly. For example, some fuzzers may not have the ability to provide feedback on test results between each iteration. In such cases, our generator might end up idly waiting for a result before generating and sending the next packet, causing the entire testing process to stall. To address this, we plan to use the configuration communicated via API and adapt our project to support different working modes and processes based on the specific capabilities of the fuzzer being used.

6.4 Technical documentation of our project

Although we have a fake fuzzer as an example of how users can utilize the API to interact with our generator, some users that aren't that familiar with programming and APIs might still have zero idea of how to use our generator or utilize the API. After fully developing our project, we plan to create a thorough, detailed technical documentation of our project. Unlike this project report, our technical documentation will be focusing on explaining to users how to use our generator, how to form interactions between the fuzzer and generator using API.

7. Conclusion

In this paper, we have explored the limitations of current network fuzzing tools, such as Nyx-Net, and introduced our Grammar-Based Packet Generator with API as a complementary solution to enhance fuzz testing processes. By addressing the shortcomings of generative approaches in handling complex, grammar-driven input structures, our project offers a modular and extensible tool that can dynamically generate structured packets, interact seamlessly with fuzzers, and adapt based on real-time feedback.

Our packet generator bridges the gap between generative and grammar-based fuzzing, bringing significant improvements in efficiency, accuracy, and automation. This capability ensures that fuzzers can uncover vulnerabilities more effectively in stateful and complex systems. Furthermore, the modular design of our API makes it a versatile addition to existing fuzzing frameworks, enabling its integration with Nyx-Net and other tools while remaining flexible for broader applications.

The implementation details, including our prototype, HTTP Packet Generator, and Fake Fuzzer, demonstrate the feasibility and functionality of our solution. Our focus on Python 3 and extensive use of APIs ensures compatibility with modern systems and fosters collaboration and contributions from the community.

Looking ahead, our future work aims to expand the generator's capabilities by supporting more packet types, enhancing randomness to cover diverse scenarios, and enriching API interactions for greater adaptability to different fuzzing workflows. Additionally, we plan to develop comprehensive technical documentation to make our tool accessible and user-friendly for a wider audience.

Our Grammar-Based Packet Generator with API not only improves upon traditional fuzzing methods but also opens new possibilities for advancing network fuzzing research. By enabling more targeted, efficient, and automated testing processes, it represents a meaningful step forward in the quest to enhance software security.

8. References

[1] Schumilo, S., Aschermann, C., Jemmett, A., Abbasi, A., and Holz, T. 2022. Nyx-Net: Network Fuzzing with Incremental Snapshots. In Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, Rennes, France. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3492321.3519591

[2] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In USENIX Security Symposium, 2020.

[3] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNET: A Greybox Fuzzer for Network Protocols. In IEEE International Conference on Software Testing, 2020.

[4] Pham, V.-T. "AFLSmart++: Smarter Greybox Fuzzing." Proceedings of the SBFT'23 Workshop. School of Computing and Information Systems, University of Melbourne, 2023.

[5] Cornelius Aschermann, Patrick Jauernig, Tommaso Frassetto, Ahmad-Reza Sadeghi, Thorsten Holz, and Daniel Teuchert, "NAUTILUS: Fishing for Deep Bugs with Grammars," Proceedings of the Network and Distributed System Security Symposium (NDSS), February 24–27, 2019, San Diego, CA, USA. DOI: https://doi.org/10.14722/ndss.2019.23412.

[6] Scapy Development Team. (n.d.). *Scapy Documentation*. Retrieved November 27, 2024, from https://scapy.readthedocs.io/en/latest/