Fuzzing for Software Security Testing

Objective:

This lab aims to introduce students to fuzzing, a dynamic software testing technique used to discover vulnerabilities in software applications. By the end of this lab, students will be able to:

1. Understand the basics of fuzzing and its importance in security testing.
2. Set up a fuzzing environment.
3. Use a fuzzing tool to test a sample application.
4. Analyze the results of a fuzzing test.

Prerequisites:

- Basic understanding of programming (Python/C/C++).
- Familiarity with command-line interfaces.
- Basic knowledge of software vulnerabilities (e.g., buffer overflows, crashes).

Lab Setup

Tools Required:
1. Fuzzing Tool: AFL (American Fuzzy Lop) or libFuzzer
2. Target Application: A simple C program with potential vulnerabilities (provided below).
3. Operating System: Linux (Ubuntu recommended).

Setup Instructions:

1. Install AFL:

```
sudo apt update
sudo apt install afl
```

2. Create a Target Application: Save the following C code as vulnerable.c:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void vulnerable_function(char *input) {
    char buffer[100];
    strcpy(buffer, input); // Potential buffer overflow
}

int main(int argc, char *argv[]) {
    if (argc > 1) {
        FILE *file = fopen(argv[1], "r");
        if (!file) {
            perror("Error opening file");
            return 1;
        }

        char input[1024];
        size_t bytesRead = fread(input, 1, sizeof(input) - 1,
file);
        fclose(file);

        input[bytesRead] = '\0';

        vulnerable_function(input);
    } else {
        printf("Usage: %s <filename>\n", argv[0]);
    }
    return 0;
}
```

3. Compile the Target Application with AFL

```
afl-gcc -o vulnerable vulnerable.c
```

Lab Steps

Step 1: Understand the Target Application

- ➢ Review the vulnerable.c code and identify the potential vulnerability (buffer overflow).
- ➢ Discuss why this vulnerability is dangerous and how it can be exploited.

Step 2: Run the Fuzzer

1. Create an input directory for AFL:

```
mkdir input
echo "seed" > input/seed.txt
```

2. Run AFL on the target application:

```
afl-fuzz -i input -o output ./vulnerable @@

  - -i input: Specifies the input directory with seed files.
  - -o output: Specifies the output directory for fuzzing results.
  - ./vulnerable @@: The target application, where @@ is a placeholder for
the input file.
```

Step 3: Monitor the Fuzzing Process

- ➢ Observe the AFL interface, which provides real-time statistics such as:
- ➢ Total Executions: Number of test cases executed.
- ➢ Unique Crashes: Number of unique crashes found.
- ➢ Hangs: Number of timeouts or hangs.
- ➢ Let the fuzzer run for 5-10 minutes.

Step 4: Analyze the Results

1. After stopping the fuzzer, check the output directory:

```
ls output/crashes
```

- ➢ Look for files that caused crashes (e.g., `id:000000,sig:11`).

2. Replay the crash:

```
./vulnerable output/crashes/id:000000,sig:11
```

- ➢ Observe the crash and discuss why it occurred.

Step 5: Fix the Vulnerability

- ➢ Modify the vulnerable.c code to fix the buffer overflow (e.g., use strncpy instead of `strcpy`).
- ➢ Recompile and rerun the fuzzer to verify the fix.

Questions

1. What is the purpose of fuzzing in software security testing?
2. How does AFL generate test cases to find vulnerabilities?
3. What other types of vulnerabilities can fuzzing detect besides buffer overflows?
4. How can you improve the efficiency of a fuzzing campaign?

Bonus Challenge

1. Use a different fuzzing tool (e.g., libFuzzer or Honggfuzz) and compare its effectiveness with AFL.
2. Write a custom fuzzer in Python using the random or hypothesis library to test the same application.

Deliverables

1. A lab report summarizing your findings, including:
    - ➢ Screenshots of the AFL interface.
    - ➢ Analysis of the crashes found.

- ➢ Explanation of how you fixed the vulnerability.

2. Answers to the lab questions.

This lab provides hands-on experience with fuzzing, a critical technique in modern software security testing.
By the end, students will have a solid foundation in using fuzzing tools to identify and mitigate vulnerabilities in software applications.

Optional Task

Create a directory structure for the codebase:

```
mkdir -p codebase/src codebase/include codebase/tests
```

1. Save the following files in the codebase directory:

File 1: `src/main.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "network.h"
#include "file_handling.h"
#include "crypto.h"
#include "data_processing.h"
#include "authentication.h"

int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Usage: %s <module_id> <input>\n", argv[0]);
        printf("Module IDs:\n");
        printf("1 - Network Module\n");
        printf("2 - File Handling Module\n");
        printf("3 - Crypto Module\n");
        printf("4 - Data Processing Module\n");
        printf("5 - Authentication Module\n");
        return 1;
```

```c
    }

    int module_id = atoi(argv[1]);
    char *input = argv[2];

    switch (module_id) {
        case 1:
            network_handler(input);
            break;
        case 2:
            file_handler(input);
            break;
        case 3:
            crypto_handler(input);
            break;
        case 4:
            data_processor(input);
            break;
        case 5:
            authenticate(input);
            break;
        default:
            printf("Invalid module ID\n");
            break;
    }

    return 0;
}
```

File 2: `src/network.c`

```c
#include <stdio.h>
#include <string.h>
#include "network.h"

void network_handler(char *input) {
```

```c
        char buffer[100];
        strcpy(buffer, input); // Potential buffer overflow
        printf("Network Handler: %s\n", buffer);
    }
```

File 3: `src/file_handling.c`

```c
    #include <stdio.h>
    #include <stdlib.h>
    #include "file_handling.h"

    void file_handler(char *input) {
        FILE *file = fopen(input, "r");
        if (file == NULL) {
            printf("File not found: %s\n", input);
            return;
        }
        char buffer[100];
        fgets(buffer, sizeof(buffer), file); // Potential file
handling issue
        printf("File Content: %s\n", buffer);
        fclose(file);
    }
```

File 4: `src/crypto.c`

```c
    #include <stdio.h>
    #include <string.h>
    #include <openssl/md5.h>
    #include "crypto.h"

    void crypto_handler(char *input) {
        unsigned char digest[MD5_DIGEST_LENGTH];
        MD5((unsigned char*)input, strlen(input), digest); //
Potential crypto misuse
        printf("MD5 Hash: ");
        for (int i = 0; i < MD5_DIGEST_LENGTH; i++) {
```

```
        printf("%02x", digest[i]);
    }
    printf("\n");
}
```

File 5: `src/data_processing.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "data_processing.h"

void data_processor(char *input) {
    int data[10];
    int index = atoi(input);
    data[index] = 42; // Potential out-of-bounds access
    printf("Data at index %d: %d\n", index, data[index]);
}
```

File 6: `src/authentication.c`

```
#include <stdio.h>
#include <string.h>
#include "authentication.h"

void authenticate(char *input) {
    char password[10] = "secret";
    if (strcmp(input, password) == 0) {
        printf("Authentication successful!\n");
    } else {
        printf("Authentication failed!\n");
    }
}
```

File 7: `include/network.h`

```
#ifndef NETWORK_H
#define NETWORK_H
void network_handler(char *input);
#endif
```

File 8: `include/file_handling.h`

```
#ifndef FILE_HANDLING_H
#define FILE_HANDLING_H
void file_handler(char *input);
#endif
```

File 9: `include/crypto.h`

```
#ifndef CRYPTO_H
#define CRYPTO_H
void crypto_handler(char *input);
#endif
```

File 10: `include/data_processing.h`

```
#ifndef DATA_PROCESSING_H
#define DATA_PROCESSING_H
void data_processor(char *input);
#endif
```

File 11: `include/authentication.h`

```
#ifndef AUTHENTICATION_H
```

```
#define AUTHENTICATION_H
void authenticate(char *input);
#endif
```

2. Compile the Codebase with AFL:

```
cd codebase

afl-gcc -o codebase src/main.c src/network.c
src/file_handling.c src/crypto.c src/data_processing.c
src/authentication.c -lssl -lcrypto
```

Step 1: Understand the Codebase

➢ Review the codebase and identify the five modules:

1. Network Module: Handles network-related operations (buffer overflow vulnerability).
2. File Handling Module: Handles file operations (file handling vulnerability).
3. Crypto Module: Handles cryptographic operations (potential misuse of crypto functions).
4. Data Processing Module: Processes data (out-of-bounds access vulnerability).
5. Authentication Module: Handles user authentication (potential authentication bypass).

➢ Discuss the potential vulnerabilities in each module.

Step 2: Run the Fuzzer on Each Module

1. Create an input directory for AFL:

```
mkdir input
echo "seed" > input/seed.txt
```

2. Run AFL on each module individually by modifying the main function to call only the target module. For example, to test the Network Module:

```
int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <input>\n", argv[0]);
        return 1;
    }
    network_handler(argv[1]);
    return 0;
}
```

Recompile and run AFL:

```
afl-gcc -o network_test src/main.c src/network.c -lssl -l
crypto

afl-fuzz -i input -o output_network ./network_test @@
```

3. Repeat this process for the other modules:

- ➢ File Handling Module: Modify main to call file_handler.
- ➢ Crypto Module: Modify main to call crypto_handler.
- ➢ Data Processing Module: Modify main to call data_processor.
- ➢ Authentication Module: Modify main to call authenticate.

Step 3: Monitor and Analyze Results

- ➢ Monitor the AFL interface for each module's crashes, hangs, and unique paths.
- ➢ Analyze the crashes and determine the root cause of each vulnerability.

Step 4: Fix the Vulnerabilities

- ➢ Fix each vulnerability in the code:

1. Network Module: Replace *strcpy* with *strncpy*.
2. File Handling Module: Add proper file path validation and error handling.
3. Crypto Module: Use secure cryptographic practices (e.g., avoid MD5 for sensitive data).
4. Data Processing Module: Add bounds checking for array access.
5. Authentication Module: Implement secure password handling (e.g., hashing).

- ➢ Recompile and rerun the fuzzer to verify the fixes.