Coursework 05
Detection of Software Vulnerabilities: Dynamic Analysis

Introduction

This coursework introduces students to the basic concepts to test security properties in C programs. In particular, this coursework provides four exercises: *(i)* compute the statement, decision, branch, and condition coverage; *(ii)* compute the MC/DC; *(iii)* apply coverage test generation for security; and (iv) apply existing fuzzing tools to find security vulnerabilities in open-source C applications.

Learning Objectives

By the end of this lab, you will be able to:

- Understand dynamic detection techniques to identify security vulnerabilities.
- Generate executions of the program along paths that will lead to the discovery of new vulnerabilities.
- Be able to work on black-box fuzzing: grammar-based and mutation-based fuzzing.
- Be able to work on white-box fuzzing: dynamic symbolic execution.

1) **(Code Coverage)** Compute the statement, decision, branch, and condition coverage for the following fragments of C code.

i.

```c
#include <stdio.h>
int main() {
  int number1, number2;
  printf("Enter two integers: ");
  scanf("%d %d", &number1, &number2);

  if (number1 >= number2) {
    if (number1 == number2) {
      printf("Result: %d = %d",number1,number2);
    }
    else {
      printf("Result: %d > %d", number1, number2);
    }
  }
  else {
    printf("Result: %d < %d",number1, number2);
  }
  return 0;
}
```

ii.

```c
#include <stdio.h>
int main() {
  int var1, var2;
  printf("Input the value of var1:");
  scanf("%d", &var1);
  printf("Input the value of var2:");
  scanf("%d",&var2);
  if (var1 != var2) {
    printf("var1 is not equal to var2\n");
    if (var1 > var2) {
      printf("var1 is greater than var2\n");
    }
    else {
      printf("var2 is greater than var1\n");
    }
  }
  else {
    printf("var1 is equal to var2\n");
  }
  return 0;
}
```

2) **(Modified condition/decision coverage)** Compute the MC/DC for the following fragment of C code.

```c
#include <stdio.h>
void findElement(int arr[], int size, int key) {
  for (int i = 0; i < size; i++) {
    if (arr[i] == key) {
      printf("Element found at position: %d", (i + 1));
      break;
    }
  }
}
int main() {
  int arr[] = { 1, 2, 3, 4, 5, 6 };
  int n = 6;
  int key = 3;
  findElement(arr, n, key);
  return 0;
}
```

3) (Coverage Test Generation for Security) Use ESBMC to produce test cases that can expose two types of bugs: *buffer overflow* and *typecast overflow*.

```
#define BUFFER_MAX 10
static char buffer[BUFFER_MAX];
int first, next, buffer_size;
void initLog(int max) {
  buffer_size = max;
  first = next = 0;
}
int removeLogElem(void) {
  first++;
  return buffer[first-1];
}
void insertLogElem(int b) {
  if (next < buffer_size) {
    buffer[next] = b;
    next = (next+1)%buffer_size;
  }
}
```

4) **(Fuzzing)** Choose an open-source C application that can take input from the command line in some complex file format [1]. You should check whether this open-source application is mentioned on https://www.cvedetails.com/vulnerability-list/; if so, you may want to test the old version (e.g., the well-known security vulnerabilities of the gzip application can be found online at https://www.cvedetails.com/vulnerability-list/vendor_id-72/product_id-1670/GNU-Gzip.html). Try out the fuzzing tools (Radamsa [2], zuff [3], and afl [4]) with/without instrumentation with additional checks for memory safety code (valgrind [5], AddressSanitizer [6]).

References:

[1] https://people.csail.mit.edu/smcc/projects/single-file-programs/.

[2] Radamsa, https://gitlab.com/akihe/radamsa, accessed on 08-05-2020.

[3] Zuff, http://caca.zoy.org/wiki/zzuf, accessed on 08-05-2020.

[4] AFL, https://lcamtuf.coredump.cx/afl/, accessed on 08-05-2020.

[5] Valgrind, https://valgrind.org/, accessed on 08-05-2020.

[6] AddressSanitizer, https://github.com/google/sanitizers/wiki/AddressSanitizer, accessed on 08-05-2020.