In the name of God

Sharif University of Technology
Department of Computer Engineering

CE 815: Secure Software Systems

Mehdi Kharrazi *Azar 6^{th}, 1400*

Homework 1*

1 BAT: Binary Analysis Tool

Part II: Disassembling elves with Capstone

1.1 Instructions

In the previous assignment, we made a parser for executable in Linux, now we are going to make use of that to disassemble the code in an executable. Covering an ISA like x86 (or almost any ISA) is a daunting task which fortunately we do not have to handle thanks to Capstone. You can use the tutorial on S4Lab blog to help you get started with this tool.
What Capstone does is fairly simple: you give Capstone the bytes and get back the instructions in a human-friendly format. Notice that besides C/C++ there are other bindings for Capstone in most programming languages.

The task expected of you in this assignment, is to use your parser to find code sections and output the disassembled instructions in a file. Mistaking data for code is one of the challenges you might face in this assignment. Along with your program, you must hand in a short document explaining your work, and the output of your dis-assembler for the given example programs.

1.2 Delivery

You should submit a report, explaining your program. You should also submit your code and explain how to run it.

*This homework was developed by Iman Hosseini and Solmaz Salimi, edited by Razieh Eskandari and Parisa Ezzatpanah

2 PIN Tool

Pin is a dynamic binary instrumentation framework for the IA-32, x86-64 and MIC instruction-set architectures that enables the creation of dynamic program analysis tools. The tools created using Pin, called Pintools, can be used to perform program analysis on user space applications on Linux*, Windows* and macOS*.

As a dynamic binary instrumentation tool, instrumentation is performed at run time on the compiled binary files. Thus, it requires no recompiling of source code and can support instrumenting programs that dynamically generate code.

2.1 Control Flow Integrity

Suppose you have an arbitrary vulnerable program that has unreliable input from stdin or file and then copies the input to the desired buffer without checking the length, which leads to BoF vulnerability, such as the following program.

a) For the given code, you should first compile it and then use **PIN** API to dynamically instrument the compiled program to report the instruments. You can compile the code yourself, therefore you can have a binary that perfectly suits the PIN version and any other tools you prefer. We strongly recommend reading Shell storm reference.

b) Is it possible to write a PIN-based tool to detect/prevent the BoF attack at run-time? If yes, then write a PIN-based tool and show its results for the below vulnerable program in presence/absence of exploitation. If not, then discuss why it is impossible to do otherwise.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//#include <unistd>
//using namespace std;
int smashme ( char * blah, char * smash) {
        strcpy (smash , blah);
        return 0;
}

void function (char * arg){
        char buf1 [500] = "Hi, Welcome to the software security course :)\n";
        char buf2 [300];
        smashme(arg,buf2);
        printf("%s",buf1);
}
int main ( int argc , char** argv) {
 if (argc == 2) {
        function ( argv [ 1 ] ) ;
        }
 else{
        printf("please provide two arguments");
        exit(EXIT_FAILURE);
    }
  return 0;
}
```

2.2 Delivery

You should submit code, its documentation and output in txt format.

3 SAT: Source Code Analysis Tool

In these series of assignment, you will implement some simple static analyses tools for Java Program enabling vulnerability detection.

First, you should generate an AST (abstract syntax tree) for a Java program in order to perform these analyses.

Part I: Integer Analysis

3.1 Instructions

The goal of this part is to implement a simple **Constant Integer** analyzer to detect:

a) division by zero,

b) negative array index and

c) bad shift *-i.e., shift by a constant that is greater than 31 or less than 0 -*

In this phase, you should only track variables of type int which are **constant**. Also, you could assume that other variables are unknown. But pay attention that we are extending these analyses in the next HW. Hence, your implementation should be as high-level and extendable as possible. Some sample input and output are provided in *handouts*.

3.2 Desired Output:

Your analysis should report warnings when it detects an array access that may involve a negative array index, a division that may result in a division by zero, a shift which might be logically incorrect.

3.3 Delivery

You should submit code, its documentation and output in txt format.