In the name of God

Sharif University of Technology
Department of Computer Engineering

CE 874: Secure Software Systems

Mehdi Kharrazi *Aban 4^{th}, 1398*

Homework 1*

1 BAT: Binary Analysis Tool

Part II: Disassembling elves with Capstone

1.1 Instructions

For simplicity, you only need to handle ELF executables for the rest of this assignment.

In the previous assignment, we made a parser for executables in Windows and 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 disassembler 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.

2 CFI with Binary Patching

2.1 Instructions

In this part of assignment, you get familiar with **Static Instrumentation** and its application in CFI*.

Imagine, that you are give the following vulnerable code:

*Acknowledgement: This homework was developed by Iman Hosseini and Solmaz Salimi

*The idea of this part of the assignment is partially based on "A Ghetto Implementation of CFI on x86", published on PoC||GTFO, issue11.

```
#include <string.h>
void smashme (char* blah) {
char smash [16] ;
strcpy (smash , blah) ;
}
int main (int argc , char** argv) {
if (argc>1) {
smashme (argv [1]) ;
 }
```

We know that we can simply use BoF to overwrite the *return address* in `smashme`. If we knew the exact *return address* in this function we can patch the binary code to guarantee the expected run. To do so, you should add a check before `ret` instruction in `smashme` function.

You are expected to this binary patching, for the given vulnerable BoF program, you exploited in HW0. You can use any tool for ELF binary modification, but your final Binary should be a valid executable with proper check for both `bar` and `foo` functions.

2.2 Delivery

You should submit your new **Valid ELF Binary** and explain your modified code. Also answer the following question in your report:
Explain is it possible to have an **Automatic and General** tool to **fully statically** find function parent's return address and patch the child function to enforce CFI.

3 Dynamic Taint Analysis

3.1 Instructions

For the given code, you should first compile it and then use **PIN** API to dynamically instrument the compiled program to taint the *read* data.

```
int ce874(char a, char b, char c)
{
  a = 8;
  b = 7;
  c = 4;
  return 0;
}

int ce442(char* buf)
{
  char c, b, a;

  a = buf[0];
  b =  buf[4];
  a = buf[8];
  ce874(a, b, c);
  return true;
}

int main(int argc, char** argv)
{
  int fd;
  char * buf;
  if (!(buf = malloc(32)))
    return -1;
  fd = open("./ce874.txt", O_RDONLY);
  read(fd, buf, 32), close(fd);
  ce442(buf);
}
```

First create an input file for `read` function with proper data. 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, but **you are not allowed to copy any part of its solution**.

You can find the sample output in handouts repo. Your code should find the appropriate taint range and taint both memory and registers. Pay attention that your code should work for other programs with small changes.

3.2 Delivery

You should submit both code and its output in txt format. The report for this section should explain the steps for taint analysis and different parts of your code.