

David Warren

warredl3

david.l.warren@vanderbilt.edu

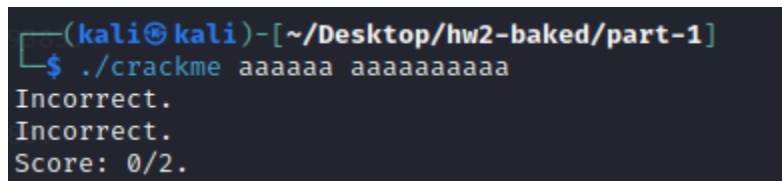
CS8395 — Homework 2

Summary

This assignment aims to familiarize students with binary analysis using Ghidra, a tool created by the NSA, and the use of LLVM to analyze and transform programs. These are both tools used in security for a variety of analyses. To complete this assignment, I will be using VirtualBox 7.0.6. I am using the prebuilt virtual machine for VirtualBox off of the Kali Linux website. The kernel version is 6.0.0. Ghidra is version 10.2.2 and I am using llvm-13.

Task 1

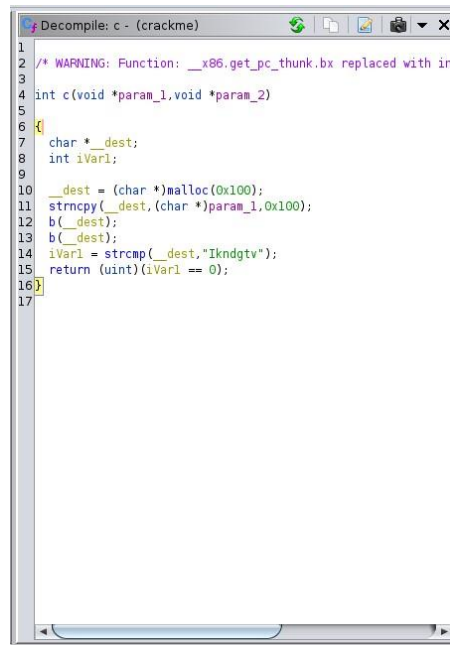
This task is to use Ghidra to solve a crackme with two passwords. The execution of the crack me is as follows: the program will indicate if a password is “Correct” or “Incorrect” and provide a score out of 2 for the total number of solved passwords. See *Figure 1*. The goal is to find the correct values for passwords 1 and 2.



```
(kali㉿kali)-[~/Desktop/hw2-baked/part-1]
$ ./crackme aaaaaa aaaaaaaaaa
Incorrect.
Incorrect.
Score: 0/2.
```

Fig.1 Sample execution

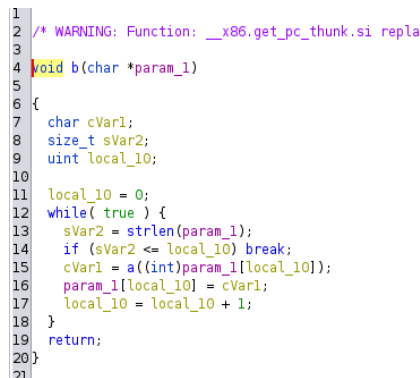
To start, I created a new project in Ghidra and imported the crackme to analyze. The first thing I did was trace through main() to get an idea of the control flow of the code. Tracing through the code and then confirming with the defined strings tool, I observed four interesting strings: “Correct,” “Incorrect,” “Violet,” and “Ikndgtv.” The first two were in functions that were called to indicate results given by the current score. The score was being determined by two other functions which I investigated. The simpler of the next two functions called in main (which we will call d()) does a simple strcmp() of its parameter to a string I saw before (“Violet”) and incremented score to indicate a correct password. This is one of the two passwords. The previous function (which we will call c()) is a bit more complex. See *Figure 2*.



```
1
2 /* WARNING: Function: __x86.get_pc_thunk.bx replaced with in
3
4 int c(void *param_1,void *param_2)
5
6 {
7     char *__dest;
8     int iVar1;
9
10    __dest = (char *)malloc(0x100);
11    strncpy(__dest,(char *)param_1,0x100);
12    b(__dest);
13    b(__dest);
14    iVar1 = strcmp(__dest,"Ikndgtv");
15    return (uint)(iVar1 == 0);
16 }
17
```

Fig 2. Function c()

This function calls b() twice with the __dest char* variable, and param_1. It appears param_1 might be undergoing some transformation in b(). After seeing, this the function b() must be explored. b() is a simple function that loops through param_1 and calls a function a() on each character. See *Figure 3*.



```
1
2 /* WARNING: Function: __x86.get_pc_thunk.si repla
3
4 void b(char *param_1)
5
6 {
7     char cVar1;
8     size_t sVar2;
9     uint local_10;
10
11    local_10 = 0;
12    while( true ) {
13        sVar2 = strlen(param_1);
14        if (sVar2 <= local_10) break;
15        cVar1 = a((int)param_1[local_10]);
16        param_1[local_10] = cVar1;
17        local_10 = local_10 + 1;
18    }
19    return;
20 }
21
```

Fig. 3. Looping through characters of param_1 in b()

With this knowledge, it appears a transformation of each individual character is occurring. This happens two times in function c(). Now to analyze a(). Function a() can be seen below in *Figure 4*.

```

char a(char param_1)
{
    char local_5;

    if (param_1 < '\0') {
        local_5 = 'z';
    }
    else if (param_1 < 'z') {
        local_5 = param_1 + '\x01';
    }
    else {
        local_5 = '!';
    }
    return local_5;
}

```

Fig. 4. Character transformation

Function a() takes a char and modifies it based on its ASCII value, transforming it to z if it is less than the ‘\0’ character or if it is greater than ‘z’ it is replaced with ‘!’ and as the comparison string has no ‘!’ we know this branch does not occur for our password. To solve password 2, reverse the transformation given ASCII values of the comparison string. I wrote the pseudocode and reverse transformations below seen in Figure 5, along with displaying results from the crackme.

```

3 //loop through the given input twice
4
5 //call a() on each character in string
6
7 /**
8
9 if char < '034'
10 char = 122
11
12 else if char < '122'
13 char = char + 1
14
15 else
16 char = 033
17
18 */
19
20 // final result must be "Ikndgtv"
21
22 // reverse the cipher of Ikndgtv
23
24 // 073 107 110 100 103 116 118
25
26 // 072 106 109 099 102 115 117
27
28 // 071 105 108 098 101 114 116

```

```

kali@kali: ~/Desktop/hw2-baked/part-1
File Actions Edit View Help
Incorrect.
Score: 0/2.

(kali@kali)-[~/Desktop/hw2-baked/part-1]
$ ./crackme Violet asdfasf
Correct.
Incorrect.
Score: 1/2.

(kali@kali)-[~/Desktop/hw2-baked/part-1]
$ ./crackme Violet Ikndgtv
Correct.
Incorrect.
Score: 1/2.

(kali@kali)-[~/Desktop/hw2-baked/part-1]
$ ./crackme Violet Gimbert
Correct.
Incorrect.
Score: 1/2.

(kali@kali)-[~/Desktop/hw2-baked/part-1]
$ ./crackme Violet Gilbert
Correct.
Correct.
Score: 2/2.

(kali@kali)-[~/Desktop/hw2-baked/part-1]
$

```

Fig 5. Pseudocode and results.

To reach the comparison text, I modify the ASCII values in reverse order from the simple cipher two times, one for each call of b(). All character values are within 2 of 122 so the first and last code blocks will never be executed for this password. Shift all ASCII values down twice and password 2 has been solved. A patch could have also been implemented to the binary file through Ghidra, but I was interested in how the passwords were being confirmed and opted to find the values of the passwords directly.

Task 2

The next task involves using LLVM to analyze and instrument a program to implement a rudimentary stack canary. LLVM is a compiler framework that can build programs that analyze and transform programs for us. While commonly used for compiler optimizations, LLVM can be used for security purposes. Static analysis could be used with LLVM to identify and patch common security vulnerabilities such as buffer overflows, SQL Injections, or cross site scripting vulnerabilities. This also alleviates some work from the programmer as this static analysis can be done over an entire code base rather than manual code reviews.

To begin Task 2, I created the makefile that will be used to compile the LLVM pass that will be applied to the vulnerable file subject.c. I modified InstrumentFunctions.cpp to augment instructions in the IR I will create to compare the generated rudimentary stack canary with the value of the stack canary after a vulnerable strcpy() call. I used clang to create the LLVM intermediate representation of this program containing the vulnerability. I compiled the IR with the workaround provided on Piazza to directly lower the .ll file to a .o file. See *Figure 6* below for results of the program execution with good input and bad input.

```
(kali@kali)-[~/Desktop/hw2-baked/part-2]
$ cd pass/build

(kali@kali)-[~/Desktop/hw2-baked/part-2/pass/build]
$ make
Consolidate compiler generated dependencies of target InstrumentFunctions
[100%] Built target InstrumentFunctions

(kali@kali)-[~/Desktop/hw2-baked/part-2/pass/build]
$ cd ..

(kali@kali)-[~/Desktop/hw2-baked/part-2/pass]
$ cd ..

(kali@kali)-[~/Desktop/hw2-baked/part-2]
$ clang -m32 -emit-llvm -S -fno-stack-protector -static subject.c -o subject.ll

(kali@kali)-[~/Desktop/hw2-baked/part-2]
$ $LLVM_DIR/bin/opt -load-pass-plugin ./pass/build/lib/libInstrumentFunctions.so --passes="cs8395-hw2" subject.ll -S -o instrumented.ll

(kali@kali)-[~/Desktop/hw2-baked/part-2]
$ clang -m32 instrumented.ll -o instrumented.o

(kali@kali)-[~/Desktop/hw2-baked/part-2]
$ ./invoke.sh ./instrumented.o goodinput
Loading input file ...
Stack canary was 1804289383 and is now 1804289383.
Stack canary in vulnerable is the same!
Completed.
Stack canary was 846930886 and is now 846930886.
Stack canary in main is the same!

(kali@kali)-[~/Desktop/hw2-baked/part-2]
$ ./invoke.sh ./instrumented.o attack
Loading input file ...
Stack canary was 1804289383 and is now -1869574000.
Stack canary in vulnerable is NOT the same!
zsh: segmentation fault ./invoke.sh ./instrumented.o attack

(kali@kali)-[~/Desktop/hw2-baked/part-2]
$
```

Figure 6. instrumented.o good and bad input executions

As you can see in the figure, the instrumentation compares the initial stack canary value, and the stack canary value after the vulnerable function. After this, it gives an update on the stack canary in “f” function and its status.

How was this implemented in InstrumentFunctions.cpp? The following lines were added to augment the IR file in *Figure 7*. See below.

```

136
137
138 Value *FormatStrPtr = Builder.CreatePointerCast(PrintfFormatStrVar, PrintfArgTy, "formatStr"); // good result
139 Value *FormatStrBadPtr = Builder.CreatePointerCast(PrintfFormatBadStrVar, PrintfArgTy, "formatStr"); // bad result
140 Value *CanaryValPtr = Builder.CreatePointerCast(CanaryValVar, PrintfArgTy, "formatStr"); // canary val
141 auto FuncName = Builder.CreateGlobalStringPtr(F.getName()); // get the name of function we are in
142
143
144 llvm::Value* oldCanary = ConstantInt::get(Ctx, APInt(32, canaryValue)); // create a value with the original canaryValue
145 inst_iterator I;
146 for (I=inst_begin(F), E=inst_end(F); I != E; ++I) // iterate through until you find return and break to modify
147     if (isa<llvm::ReturnInst> (&*I))
148         break;
149 Builder.SetInsertPoint(&*I);
150 llvm::Value* newCanary = Builder.CreateLoad(IntegerType::getInt32Ty(Ctx), &*newStackCanary); // grab the program canary as value
151 Builder.CreateCall(
152     Printf, {CanaryValPtr, oldCanary, newCanary}
153 );
154 llvm::Value* condition = Builder.CreateICmp(CmpInst::ICMP_EQ, newCanary, oldCanary); // create a EQ compare and store the result between them
155 llvm::Instruction *ThenTerm, *ElseTerm;
156 llvm::Instruction* splitBefore = &*I;
157 llvm::SplitBlockAndInsertIfThenElse(condition, splitBefore, &ThenTerm, &ElseTerm); // split block before return on condition we built
158 Builder.SetInsertPoint(ThenTerm); // ifthen print the good result
159 Builder.CreateCall(
160     Printf, {FormatStrPtr, FuncName}
161 );
162 Builder.SetInsertPoint(ElseTerm); // else print the bad result
163 Builder.CreateCall(
164     Printf, {FormatStrBadPtr, FuncName}
165 );

```

Figure 7. Changes to be made to the IR file

The stack canary is generated above at the beginning of each function already. After preparing our print statements on lines 138-141, I create an `llvm::Value*` with the canary value generated in the IR. Next, I iterate through the functions instructions until I find the return statement, saving its pointer and breaking out of the instruction iterator's loop. Crashes may occur if you do not exit the iterator loop while attempting to insert or modify instructions. I want to insert instructions where the current return statement is, subsequently pushing the return statement down. The first instruction I want to insert is a load. I use the `CreateLoad()` function on line 150 to obtain the current value of `newStackCanary` that is located on the stack (as this would be overwritten in a buffer overflow). Next, I print a statement for results. Then I create a value from a compare statement between the original and new stack canaries on line 154. Now that we have the result from the compare, we can create some control flow logic. We want to create two blocks, one for if the canaries are the same or different. These block insertion points are created at lines 158 and 162 respectively. In the control blocks, we print the corresponding success or failure status of the canary comparison from line 154. Thus, the rudimentary stack canary is implemented in the IR. The documentation I used was from LLVM-17 and references from the LLVM programmers manual and its sample exercise Kaleidoscope.

There are problems with this stack canary however. As the stack canary is implemented in each of the functions and is just a rand canary, analysis from a malicious actor would reveal the stores of this value. As such, with this IR the instructions for comparisons are also in each of the functions. The address of this variable or comparison could be modified to subvert the canary check and render it useless. As it is a simple `rand()` canary, it may even be brute forced with enough resources. An attacker may use `gdb` to alter their buffer and check byte by byte when the comparison fails. Doing so, they may find where the buffer is located and procedurally recreate the canary. We can assume sufficient knowledge of `gdb`, `llvm` and assembly, access to the `.s` or `.ll` file, or enough resources would be sufficient to compromise this technique.