DCC888 – Introduction

Name: _____ ID: _____

1. This question refers to the program below, which has been written in C. This program computes the value stored in the variable `sum` in a rather naïve way:

```
#include <stdio.h>
#define CUBE(x) (x)*(x)*(x)
int main() {
  int i = 0;
  int x = 2;
  int sum = 0;
  while (i++ < 100) {
    sum += CUBE(x);
  }
  printf("%d\n", sum);
}
```

   (a) If we compile this program with, say, `gcc -O1`, then we obtain the assembly code below:

```
_main:
  pushl %ebp
  movl  %esp, %ebp
  pushl %ebx
  subl  $20, %esp
  call  L3
L3:
  popl  %ebx
  movl  $800, 4(%esp)
  movl  %eax, (%esp)
  call  _printf
  addl  $20, %esp
  popl  %ebx
  leave
  ret
```

   The assembly program does not contain the loop in the original program. What happened to this loop?

(b) How do you think `gcc` has been able to transform the original program, written in C, in the optimized assembly code? Try to think about a few optimizations which, combined, could lead to the assembly program.

2. This question refers to the program below, which was written in PHP:

```
$id = $_GET["user"];

if ($id == '') {
  echo "Invalid user: $id"
} else {
  $getuser = $DB->query
      ("SELECT * FROM `table` WHERE id='$id'");
  echo $getuser;
}
```

(a) This program contains a vulnerability to form of software attack called *SQL Injection*. Explain what is an SQL Injection attack, and how the program above can be exploited in this way.

(b) Compilers can be – and have been – used to discover this kind of code vulnerability [4, 5, 7]. How do you think a compiler could help the developer to identify an SQL vulnerability in a program?

3. This question refers to the program below, which has a bug:

```
1 void read_matrix(int* data, char w, char h) {
2   char buf_size = w * h;
3   if (buf_size < BUF_SIZE) {
4     int c0, c1;
5     int buf[BUF_SIZE];
6     for (c0 = 0; c0 < h; c0++) {
7       for (c1 = 0; c1 < w; c1++) {
8         int index = c0 * w + c1;
9         buf[index] = data[index];
10      }
11    }
12    process(buf);
13  }
14 }
```

BUF_SIZE	120_{char}
strlen(data)	132_{char}
buf_size	-124_{char}

w	=	0	0	0	0	0	1	1	0	$= 6_{\text{char}}$
h	=	0	0	0	1	0	1	1	0	$= 22_{\text{char}}$
h * w	=	1	0	0	0	0	1	0	0	$= -124_{\text{char}}$

(a) What is the bug that the program above contains? As a hint, the values of a few key variables are shown in boxes.

(b) Compilers can – and, again, have been able to – detect, and in some cases remove, this kind of bug automatically [1, 2, 6]. How do you think a compiler could find such a bug, and how do you think a compiler could remove it?

4. This question refers to the two programs seen in Figure 1. These programs implement the same operation. They match a password (`pw`) against an input (`in`). If a match is observed, then functions `diff1` and `diff2` return 1, otherwise they return 0.

```
1 int diff1(char *pw, char *in) {
2   int i;
3   for (i=0; i<8; i++) {
4     if (pw[i]!=in[i]) {
5       return 0;
6     }
7   }
8   return 1;
9 }
```

```
1 #define F(i) diff |= pw[i] ^ in[i]
2
3 int diff2(char *pw, char *in) {
4   int diff = 0;
5   F(0);
6   F(1);
7   F(2);
8   F(3);
9   F(4);
10  F(5);
11  F(6);
12  F(7);
13
14  return !diff;
15 }
```

Figure 1: These programs are part of Question 4.

1. One of the functions seen in Figure 1, e.g., `diff1` or `diff2`, always take the same time to run, independent on the value of `pw` or `in`. Which function is this one?

2. If a function always takes the same time to run, then we say that it is *isochronous*. Cryptographic algorithms should always be isochronous. Because one of the functions in Figure 1 is not isochronous, it is easy to guess its password. Explain how an adversary can discover the password of the non-isochronous implementation of `diff` by measuring the time that implementation takes to run.

3. Again, compilers can be used to prove that the implementation of an algorithm is - or is not - isochronous [3]. Provide a general idea on how this can be accomplished.

References

[1] David Brumley, Dawn Xiaodong Song, Tzi cker Chiueh, Rob Johnson, and Huijia Lin. RICH: Automatically protecting against integer-based vulnerabil ities. In *NDSS*. USENIX, 2007.

[2] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in c/c++. In *ICSE*, pages 760–770. IEEE, 2012.

[3] Alexander Lux and Artem Starostin. A tool for static detection of timing channels in java. *Journal of Cryptographic Engineering*, 1(4):303–313, 2011.

[4] Ørbæk P. and J. Palsberg. Trust in the λ-calculus. *Static Analysis*, 3(2):75–85, 1995.

[5] Andrei Alves Rimsa, Marcelo D'Amorim, and Fernando M. Q. Pereira. Tainted flow analysis on e-SSA-form programs. In *CC*, pages 124–143. Springer, 2011.

[6] Raphael Ernani Rodrigues, Victor Hugo Sperle Campos, and Fernando Magno Quintao Pereira. A fast and low overhead technique to secure programs against integer overflows. In *CGO*, pages 1–11. ACM, 2013.

[7] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *S&P*, pages 1–15. IEEE, 2010.