Nicole Wijkman Student ID: 20938875 nwijkman@connect.ust.hk

COMP3632 – Homework 2

Question 1 –

Reverse Engineering

(a) (3pt) What is the typical threat model for adversarial reverse engineering.

The typical threat model consists of:

The attackers – an adversary.

The attacker's capability – Find and expose source code which can in turn be used to find vulnerabilities or personal data.

The information/data the attacker is trying to steal – Passwords, important personal data, secret data (anything valuable that they can expose through the attack).

(b) (3pt) Explain what is linear disassembling.

Linear disassembling is one of the two ways of taking a binary executable and disassembling it into its assembly instructions. It decodes each byte in a linear fashion, starting from the first byte in the executable file and going all the way until the end.

(c) (3pt) Explain what is recursive disassembling, and what is the advantage of recursive disassembling comparing to linear disassembling.

Recursive disassembling is the second of the two ways of taking a binary executable and disassembling it into its assembly instructions. Unlike linear disassembling, recursive disassembling works through following the program control transfers. An advantage recursive disassembling has over linear is that linear disassembling will read any embedded data segments as an instruction, which is avoided by recursive disassembling.

(d) (3pt) Briefly clarify the procedure of control-flow recovery in modern C/C++ decompilers.

The decompiler first disassembles the code and follows this up with type- and variable recovery, to be able to do control-flow recovery. It can then generate a control-flow graph by using the recovered variables and can from this begin to recover the control-flow (for example by finding loops).

Nicole Wijkman Student ID: 20938875 nwijkman@connect.ust.hk

Question 2 –

The C function **strcat** appends a copy of the **source** string to the **destination** string. The terminating null character in **destination** is overwritten by the first character of **source**, and a null-character is included at the end of the new string formed by the concatenation of both in **destination**. **destination** is the return value. The definition of the interface of **strcat** is:

```
char * strcat ( char * destination, const char * source );
```

(a) (3pt) Is strcat safe? Why?

It is unsafe. This is because of how the unchecked size of the source string may result in a buffer overflow when it is appended to the target string. This can in turn lead to security vulnerabilities through buffer overflow attacks.

(b) (10pt) strncat is a function with similar functionality of strcat. strncat appends the first num characters of source to destination, plus a terminating null-character. destination is the return value. Please give an implementation of strncat. Its interface is char * strncat (char * destination, const char * source, size t num);

```
char * strncatImplement(char* dest, const char*
source, size_t size){
    int destLen = 0;
    int i;

    while(dest[destLen] != '\0'){
        destLen++;
    }
    for(i = 0; i < size; i++, destLen++){
        dest[destLen] = source[i];
    }
    dest[destLen+1] = '\0';
    return dest;
}
```

(c) (3pt) What problem is solved by strncat?

*The buffer overflow problem that could occur through **strcat** is solved by **strncat**, if used correctly. The function will not try to append more than the specified number of bytes to the destination string, leading to a program being safer from a buffer-attack, as this will now require more work.*

(d) (5pt) Is strncat safe? Please explain your answer.

*The function **strncat** is safe if used correctly, as improper use might still result in buffer overflow, leading to security vulnerabilities through buffer overflow attacks. For it to be safe, the function cannot append more bytes than what fits in the destination string, as a too small destination string will result in vulnerabilities for a buffer overflow attack.*

Nicole Wijkman Student ID: 20938875 nwijkman@connect.ust.hk

Question 3 –

(20pt) In addition to stack-based buffer overflow attacks, heap overflows can also be exploited. Consider the following C code, which illustrates a heap overflow.

(a) (2pt) Compile and execute this program. What is printed?

```
BEFORE: buf2 = 22222222
AFTER: buf1 = 11111111111111111111111111111122222222
```

(b) (8pt) memset and printf are invoked twice, respectively. Please list all possible security issues of them?

The memset() function does not check the allocated size of the char array, meaning it can enable buffer overflow. It also does not add a terminating null character at the end. The printf() function will print data from the stack until it finds a null character, meaning it can end up printing more than it was meant to. In the example code it ends up printing the overflowed buf1 as well as buf2, even if the instruction was to simply print buf1.

(c) (10pt) In terms of C/C++ memory management, what is the difference between stack and heap? In particular, which one is allocated/deallocated automatically, and which one needs programmers to take care of (you can search materials online but shouldn't directly copy)?

The main difference between the memory management of the stack and the heap is how the stack memory is only used by a single thread of execution while the heap memory is used by the whole application. Stack only stores local variables which can all be re-sized, while the heap stores global variables which cannot be re-sized. The allocation and de-allocations of the stack happens automatically, while it must be managed by the programmer for the heap.

Nicole Wijkman Student ID: 20938875 nwijkman@connect.ust.hk

Question 4 –

(10pt) Integer overflows can also be exploited. Consider the following C code, which illustrates an integer overflow.

```
int get item (int id x )
{
int array [1000] ;
// initialize array
. . .
// end initialization
if( id x >= 1000 ) return -1;
returnarray[id x] ;
}
```

(a) (6pt) What is the potential problem with this code? Besides integer overflow, which security issue is triggered, stack overflow or heap overflow?

A potential problem is integer overflow, as a too large number cannot be handled correctly and will be read as something different than what it is, making the code unreliable. Besides the integer overflow, stack overflow is triggered due to the int id x being stored in the stack.

(b) (4pt) How to solve this problem?

The problem of integer overflow can be solved by changing the parameter for id x to a long instead of an int. The program will then read the variable correctly and the comparison will no longer be unreliable.

Nicole Wijkman Student ID: 20938875 nwijkman@connect.ust.hk

Question 5 –

(16pt) Recall that an opaque predicate is a "conditional" that is actually not a conditional. That is, the conditional always evaluates to the same result, although it is not obvious.

(a) (6pt) Please provide two conditions of opaque predicate as example, and explain how to use them. (Please do not use too complicated conditions.)

```
// Example 1

int a = 10;
int b = a;

if(a == b){
    // some code
}
```

```
// Example 2

int c = 10;

if((pow(c, 2)) >= 0){
    // some code
}
```

Both examples are used for obfuscation, leaving more code for a potential attacker to analyze. They will always return true, meaning that they can be used to always run some code of the authors choice.

(b) (5pt) A side effect of inserting opaque predicates is that they can slow down the execution speed. Please explain the reason of the side effect, and how to alleviate it?

The reason that inserting opaque predicates results in a slower execution speed is similar to the answer of (a); they result in a higher quantity of code, and with that an additional number of computations. A way to alleviate this is to not use conditions which are too complicated as these would require a longer run-time as the program is executed.

(c) (5pt) A side effect of inserting opaque predicates is that they can increase the size of the executable. Please explain the reason of the side effect, and how to alleviate it?

The reason that the size of the executable increases is because there has to be additional code to create the opaque predicates. This side effect can be slightly alleviated by optimizing the code, writing in the simplest way without doing any unnecessary steps.

Nicole Wijkman Student ID: 20938875 nwijkman@connect.ust.hk

Question 6 –

(21pt) Considering the following C++ code. Function number ratio calculates the ratio of number characters in the input string s.

```
#include <string>

float numberratio(string s)
{
    int n = 0;
    for (int i = 0; i < s.size(); ++i)
        if (s[i] >= '0' && s[i] <= '9')
            ++n;
    return n / s.size();
}
```

(a) (3pt) Describe how to launch fuzz testing towards this function.

To launch fuzz testing towards this function, one would have to generate random inputs in a list (with some of them being invalid inputs) and use these to test any exploitable, abnormal cases. One would then have to write a test which takes this list of data and analyzes how the program behaves – if there occurs any crashing etc.

(b) (9pt) What bugs would you expect a fuzzer to identify in this function? Why? And how to fix this bug?

The fuzz test would recognize bugs which resulted in a program crash, for example if the size of the string is 0, the function will return n divided by 0, leading to a crash which the fuzz testing can be expected to identify. To fix this, one can simply check the size of the string and return 0 if the size is equal to 0.

Another example is if the input is NULL; leading to the function being unable to make the division. To fix this, one would have to double check that the input is not equal to NULL.

(c) (9pt) What bugs would be more difficult for a fuzzer to find in this function? Why? And how to fix this bug?

A bug that the fuzzer would have a difficult time finding would be how the returned value always will be equal to 0 or 1, since the function returns an int divided by an int. An int always rounds down, meaning that unless the entire string is made up out of numbers, the function returns 0. As a fuzzer typically distinguishes between crashing and non-crashing inputs, the function returning an int will most likely go unnoticed.

To fix this bug the values have to be made into float values and then be returned.