Ghidra PlugIn: "Detecting unsecure functions"

Lab Report

by

Luca Weist

3084294

submitted to

Rheinische Friedrich-Wilhelms-Universität Bonn

Institut für Informatik IV

Arbeitsgruppe für IT-Sicherheit

in degree course

Computerscience (B.Sc.)

First Supervisor: Prof. Dr. Michael Meier
University of Bonn

Second Supervisor: Dr. Matthias Frank
University of Bonn

Sponsor: Daniel Baier
University of Bonn

Bonn, April 22, 2021

ABSTRACT

There are various C coding practices which are considered to be unsafe. Yet these practices are still very commonly used and some are even included in C standard library functions. Because of this, there are many programs that contain vulnerabilities caused by the use of these unsafe practices. To detect programs that fall in this category, a given program is statically analysed and examined for the usage of a subset of these unsafe coding practices, whether used in a C standard library function or a custom function.

Contents

1 INTRODUCTION

The European Union Agency For Cybersecurity defines a vulnerability as follows: "The existence of a weakness, design, or implementation error that can lead to an unexpected, undesirable event compromising the security of the computer system, network, application, or protocol involved" [24].

Software is continuously becoming more important in both people's everyday life [22] as well as various government and industry sectors of critical importance, such as healthcare [48] and the car industry [8]. It follows that making sure that software is secure, meaning sufficiently free of vulnerabilities, is more important than ever.

An example of the harmful effects an exploited vulnerability can have is an incident that occurred in 2020, in which a patient in a hospital died as a consequence of said hospital being hacked using a weak spot in the hospital's software. It was also not the only occurrence of patients having to be relocated due to a hospital's software vulnerability. In 2019, 764 healthcare providers were attacked by similar means. [54]

As of 2020, C is the most popular programming language in the world, and has been so for most of its existence[67]. It is a highly portable language [58] that is used for a wide range of sectors of software development. For example, C is the most commonly used language for embedded systems[23] and the kernel of the three most popular operating systems are written mostly in it[39, 35].

However, according to a study that examined open-source projects written in the 7 most popular programming languages, projects written in C contain almost 50% of all vulnerabilities. They also have the highest proportion of vulnerabilities of critical severity. [62]

As such, it is clear that there exists a very large amount of vulnerable C programs. A number of those vulnerabilities are caused by the usage of unsafe functions built into the C standard library, libc [14], others by the misuse of other built-in functions [15]. In any custom function, an enormous amount of different weaknesses could potentially be contained [13].

Within this report, an implementation of a module for the reverse-engineering tool Ghidra [42] is proposed. The purpose of said module is to statically detect a subset of such weaknesses, specifically ones related to memory handling, within a function of a given C program. Additionally, it also detects the use of a number of different libc functions. For both of these functions, it is assumed that said program is not protected by further anti-reverse-engineering mechanisms.

Firstly, chapter 2 introduces the required background knowledge. Then, chapter 3 specifies the subset of detected weaknesses as well as the detected libc functions. Finally, the implementation of the module is presented in chapter 4.

2 BACKGROUND

This chapter introduces the fundamentals required to understand the implementation of the module in chapter 4. First, assembly and binary code are presented. Then, reverse-engineering as a general concept and in the context of software is established. Lastly, the reverse-engineering tool Ghidra is quickly introduced before delving into the, for this report, relevant aspects of Ghidra's API.

2.1 ASSEMBLY AND BINARY CODE

To get an executable program file from source code written in a compiled, high-level language, said source code will have to be compiled. A rough overview of the compilation process for C source code compiled with GCC [18] is depicted in figure 1. During the compilation process the original C source code is first converted into assembly code [69]. The resulting assembly code in turn gets converted to binary code, also called machine code, which forms the executable program. In reality, there is often a large amount of binary files that will have to be linked afterwards. However, this process, as well as the pre-processor operations, are largely irrelevant to the contents of this report, so they will not be discussed further.

ASSEMBLY

Although it serves as an intermediate step between high level language and binary code, assembly is not just an intermediate representation of the code. It is a full low-level programming language [31], meaning code written in assembly achieves, in comparison to higher level languages such as C++ or Java [3], very little abstraction from the details of CPU operations [63]. In fact, assembly instructions have direct equivalents in machine code [36]. Consequently, writing assembly code is more similar to writing machine code than programming in a high-level language and acts as an interface to the CPU. The programmer has direct access to registers, memory built directly into the CPU [66], and can optimize for specific hardware, amongst other advantages [56].
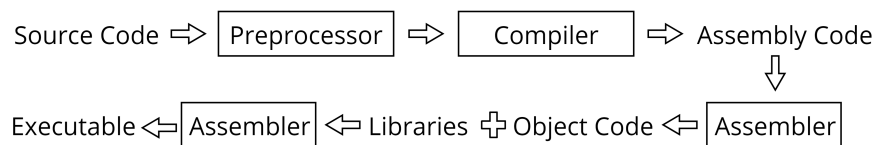
FIGURE 1: *Compilation model for GCC [69]*

On the other hand, assembly's lack of abstraction also comes with a set of disadvantages. For one, since higher level languages are closer to everyday English than assembly, assembly is less human-

readable [56]. Writing assembly code is also a lengthier process, not only for the aforementioned reason, but also because each instruction has to be manually declared. For comparison, one line of C code is often equivalent to multiple instructions in assembly, which becomes evident in the comparison between C code and equivalent assembly code in listings 2.1 and 2.2. The C source code was compiled with GCC and optimizations turned off, to achieve the most candid comparison possible. An additional disadvantage is that because of assembly codes proximity to binary code, it has to reflect the given CPU's instruction set, which results in assembly code not being portable to different CPU architectures [56].

The following paragraphs describe assembly syntax. For this, the x64 instruction set is considered, so some details may not apply to other instruction sets.

In x64, an instruction statement always consist of exactly one instruction, also called mnemonic or opcode [50]. Depending on the instruction, the statement also contains zero to three operands, each divided by a comma [50]. Operands effectively serve as parameters to the instruction. Optionally, a statement can also have a label attached to it, which will act as a reference to that statements location in the code [51]. This label can then be used as an operand [51], for instance to jump to that specific location during execution and continue the execution from there. An instruction statement can also contain a comment [50]. These serve the same purpose as comments in higher level languages, namely to provide metadata information.

```c
int compare(int a, int b) {
    if (a > b)
        return 1;
    if (a < b)
        return -1;

    return 0;
}
```

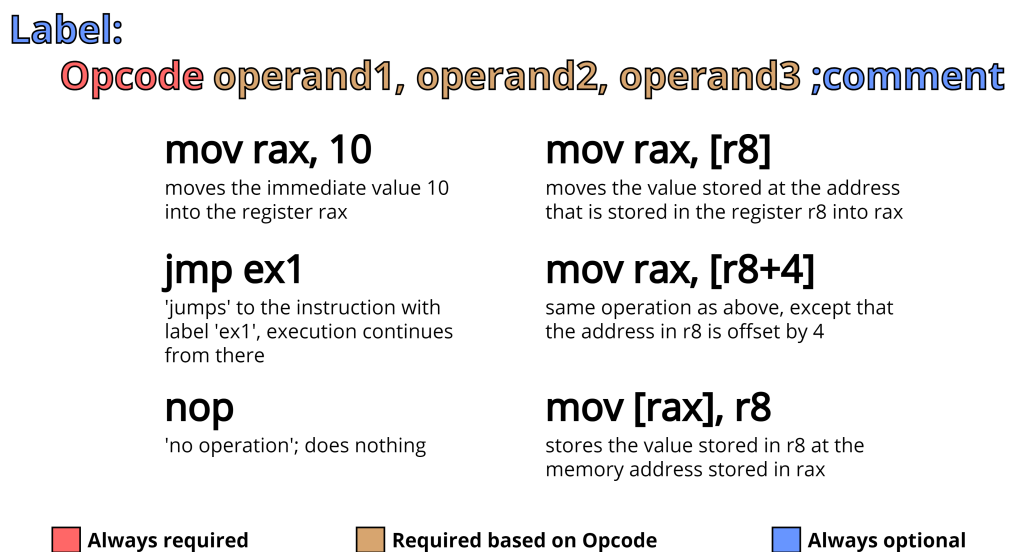**Listing 2.1:** *An example of a simple C function that compares two integers*

```
0000000000001149 <compare>:
    1149: endbr64
    114d: push rbp
    114e: mov  rbp,rsp
    1151: mov  DWORD PTR [rbp-0x4],edi
    1154: mov  DWORD PTR [rbp-0x8],esi
    1157: mov  eax,DWORD PTR [rbp-0x4]
    115a: cmp  eax,DWORD PTR [rbp-0x8]
    115d: jle  1166 <power+0x1d>
    115f: mov  eax,0x1
    1164: jmp  117a <power+0x31>
    1166: mov  eax,DWORD PTR [rbp-0x4]
    1169: cmp  eax,DWORD PTR [rbp-0x8]
    116c: jge  1175 <power+0x2c>
    116e: mov  eax,0xffffffff
    1173: jmp  117a <power+0x31>
    1175: mov  eax,0x0
    117a: pop  rbp
    117b: ret
}
```

Listing 2.2: *The function in 2.1, assembled*

There is a number of different types of operands. Immediate operands are constant values, such as numerical values or characters. Register operands directly refer to the contents of a specific

register by its name. The last type of operand that will be introduced are called memory operands, which can be addressed in multiple different ways. One way to address memory operands is to place a memory address immediate within square brackets. Square brackets in assembly behave like asterisks do in C; they dereference memory addresses, so an address literal in square brackets is equivalent to the value stored at said address. This method is called direct memory addressing. Indirect memory addressing is similar, but instead of an address literal a register operand is dereferenced. Here, the value stored in the register is treated like a memory address. In this manner, the expression is evaluated as the value stored at the address stored in said register. This addressment can be supplemented by offsets and can be scaled to access relative memory addresses. [55]

In figure 2, annotated examples of both the instruction statement structure as well as the different modes of addressment are shown.

Label:
Opcode operand1, operand2, operand3 ;comment

mov rax, 10
moves the immediate value 10 into the register rax

jmp ex1
'jumps' to the instruction with label 'ex1', execution continues from there

nop
'no operation'; does nothing

mov rax, [r8]
moves the value stored at the address that is stored in the register r8 into rax

mov rax, [r8+4]
same operation as above, except that the address in r8 is offset by 4

mov [rax], r8
stores the value stored in r8 at the memory address stored in rax

■ Always required   ■ Required based on Opcode   ■ Always optional

Figure 2: *Examples for assembly statements, in Intel Syntax*

It is worthy of mention that there are two common assembly syntaxes, AT&T and Intel [71]. In table 1, general comparisons between Intel syntax and AT&T syntax instructions can be seen. Unless otherwise specified, all assembly code in this report uses the Intel syntax.

Table 1: *Incomplete overview of differences between Intel and AT&T assembly syntax [9]*

	Intel	AT&T
Instructions	Untagged add	Tagged with operand sizes: addq (q = quadword = 32 bit)
Registers	eax, eax, etc.	%eax, %ebx, etc.
Immediate	0x100	$0x100
Indirect	[eax]	(%eax)
Operands order	<Mnemonic> <destination> <source>	<Mnemonic> <source> <destination>

Binary code

Binary code, or machine code, is the language of computers and is usually represented as binary numbers [64], hence the term binary code also being used to refer to machine code. On execution of a binary file, the CPU reads the binary stream, interpreting the incoming information as specific operations. Since it is hardly human-readable, it is extremely uncommon for programmers to actually write in binary code [64]. Because of this machine code is almost purely a generated language.

The obscurity of machine code is visualised in listing 2.3, which shows the compiled machine code of the functions in listings 2.2 and 2.1.

```
1  Binary representation:
2  11110011 00001111 00011110 11111010 01010101 01001000 10001001 11100101 10001001
3  01111101 11111100 10001001 01110101 11111000 10001011 01000101 11111100 00111011
4  01000101 11111000 01111110 00000111 10111000 00000001 00000000 00000000 00000000
5  11101011 00010100 10001011 01000101 11111100 00111011 01000101 11111000 01111101
6  00000111 10111000 11111111 11111111 11111111 11111111 11101011 00000101 10111000
7  00000000 00000000 00000000 00000000 01011101 11000011
8
9  Hexadecimal representation:
10 F3 0F 1E FA 55 48 89 E5 89 7D FC 89 75 F8 8B 45 FC 3B 45 F8 7E 07 B8 01 00 00 00
11 EB 14 8B 45 FC 3B 45 F8 7D 07 B8 FF FF FF FF EB 05 B8 00 00 00 00 5D C3
```

Listing 2.3: *The function in 2.1, 2.2 compiled to binary code*

2.2 Reverse-engineering

This section introduces the concept of reverse-engineering in general, whereas the following one introduces the reverse-engineering tool Ghidra specifically.

Generally speaking, reverse-engineering describes the process of deconstructing or analysing a finished product in order to gather information about its design [38]. For instance, if one wanted to know more about how a specific CPU worked, one could disassemble it and observe what kinds of parts were used in its production and how they are organized. In the realm of software, reverse-engineering commonly refers to the act of analysing a given program to gain information about concepts it uses as well as its implementation [53], often with the goal of reconstructing the original source code as exact as possible [60]. Use cases for reverse-engineering are, amongst other things, malware analysis [37] or as a tool for learning about programming concepts and designs [57].

There are two typical ways to analyse a compiled program. The first one, static analysis, refers to the examination of the actual code. Through static analysis, possible values for the program variables can be deduced and every possible execution path considered. However, often it is not possible to work out any concrete values and thus execution path, especially if the program takes input from a source that can not be controlled by the reverse-engineer. The other kind of analysis,

dynamic analysis, enables the recording of specific values by executing the program and monitoring memory and instructions for the duration of its execution. Dynamic analysis conversely only covers one execution path at a time. The combination of those two approaches is called hybrid analysis. [55]

Some prevalent static analysis techniques are presented below.

Basic blocks

Basic blocks are blocks containing straight sequences of code that do not contain any outflowing branches, aside from at their exit point. This implies that the only conditional in the block, if any, must be at the very end of the block. There also must not be any inflowing branches to any where in the basic block, with the exception of the entry. [27]

Single static assignment form

During program execution, there is a multitude of different execution paths that might be traversed. Because of this, there often are multiple possible values for any variable at any given point of execution. However for static analysis knowing all possible values of a variable, especially at a point of potential vulnerability, is important. Single static assignment (SSA) form aids in simplifying the process of gathering all possible values of a variable. [55]

The core principle behind SSA is that each variable may only be assigned exactly once. SSA form is achieved by replacing every reassignment of a variable with the assignment of a new, generated variable. Further, the generated variables must still be able to be traced back to their respective original variable. [55]

Often, the original variable will be reassigned differently in two or more execution paths which end up being merged. For all such cases, the at that point different possible values must be merged as well. For this, the phi-function is introduced to SSA. The phi-function takes all possible values as input and assigns it to yet another generated variable. An example of simple SSA syntax can be seen in 3. [55]
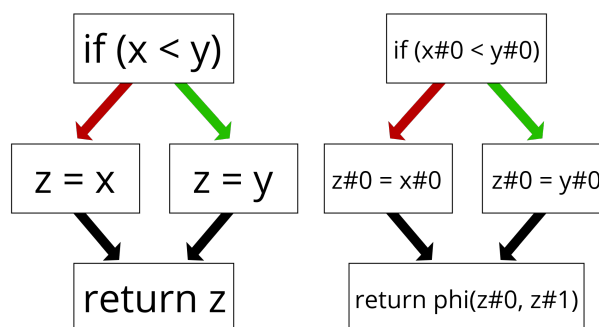
Figure 3: *A simple slice of an execution path. On the left in normal syntax; on the right, the syntax is transformed to SSA [55]*

7

Forward- and backward slicing

A variable is considered a descendant of another value after said value, or another descendant of said value, serves as input to it. Forward slicing serves to get all the descendants of a given value or variable. This is done by checking all relevant assignments. Analogously, backward slicing allows for the identification of the ancestors of a value. This in turn also allows to get the original source of a value, for example to check whether said value stems from a trusted source. [55]

There are various software tools that aid the user in reverse-engineering [4], one of which, Ghidra, will be individually discussed in the following section. Many of those tools share mutual functionalities [40, 61, 33]. A major one is presented below.

Disassembler and Decompiler

An assembler is a simple compiler, analogous to compilers for higher level languages, that translates assembly code to machine code [65]. Inversely, a disassembler converts machine code back into assembly [32]. A decompiler is to a compiler what a disassembler is to an assembler; a decompiler translates machine code back into a high level language [59], for example C.

Disassembler and decompiler are useful since machine code is not appropriately accessible to most programmers [2]. Being able to view the code in a more human-readable way potentially speeds up the static analysis, e.g. working out the actual functionality of a program or function. This is all the more the case with decompilers over disassemblers, since the large majority of programmers is more familiar with higher level languages [67].

However, both decompiler and disassembler can only approximate the original source code in most cases. This is in part a result of compilers performing optimizations on the source code during compilation, to improve execution time and memory usage [29].

Because of this multiple different lines of code can result in the same compiled machine code, making it impossible for the decompiler or disassembler to differentiate between those lines of code. Those originally different lines will be identical in the new, decompiled code. This becomes evident in listings 2.4 to 2.7, where an example of two different functions and the disassembled code returned by objdump [19] for each of those functions is shown. For both functions, the objdump output, which has been stripped of its binary code, is identical. The binary was compiled with GCC an the -O3 flag set. The -O3 flag maximizes the amount of optimizations the compiler carries out [18].

```
1  int function1() {
2      int a = 1;
3      int b = 2;
4      int numbers[10];
5
6      for (int i = 0; i < 10; i++)
7          numbers[i] = a + b;
8
9      return a + b;
10
11     int d = 3;
12     return a + d;
13 }
```

Listing 2.4: *An example of a function*

```
1  int function2() {
2      int a = 1;
3      int b = 2;
4
5      return a + b;
6  }
```

Listing 2.5: *An example of a different but similar function*

```
1  0000000000001140 <function1>:
2      1140: endbr64
3      1144: mov eax,0x3
4      1149: ret
5      114a: nop WORD PTR [rax+rax*1+0
       x0]
```

Listing 2.6: *The disassembled code of the function in 2.4*

```
1  0000000000001150 <function2>:
2      1150: endbr64
3      1154: mov eax,0x3
4      1159: ret
5      115a: nop WORD PTR [rax+rax*1+0
       x0]
```

Listing 2.7: *The disassembled code of the function in 2.5*

Yet another reason as to why decompilers are unable to restore the original source code is the loss of semantic information. Variable names, for instance, are lost after compilation [70].

2.3 Ghidra

Released to the public as open source project in 2019, Ghidra is a reverse-engineering tool developed by the National Security Agency [42].

In figure 4 Ghidra's standard view can be seen, which is shown once a project is opened. As mentioned in the previous section, Ghidra contains an integrated disassembler as well as a decompiler, both of which execute upon importing a binary file. The decompiler window can be seen on the far right, with the disassembled code window, here called Listing, next to it. Ghidra offers many additional convenience features that aid in exploring both versions of the code. For example, there is the Program Trees window, which displays all segments of the disassembled file and allows the user to skip to any of them with the click of a button. Similarly, the Symbol Tree window holds sorted lists of all imports, exports, functions, labels, classes and namespaces. This window also contains a search function. Both of these windows can be seen in figure 4 on the left hand side.
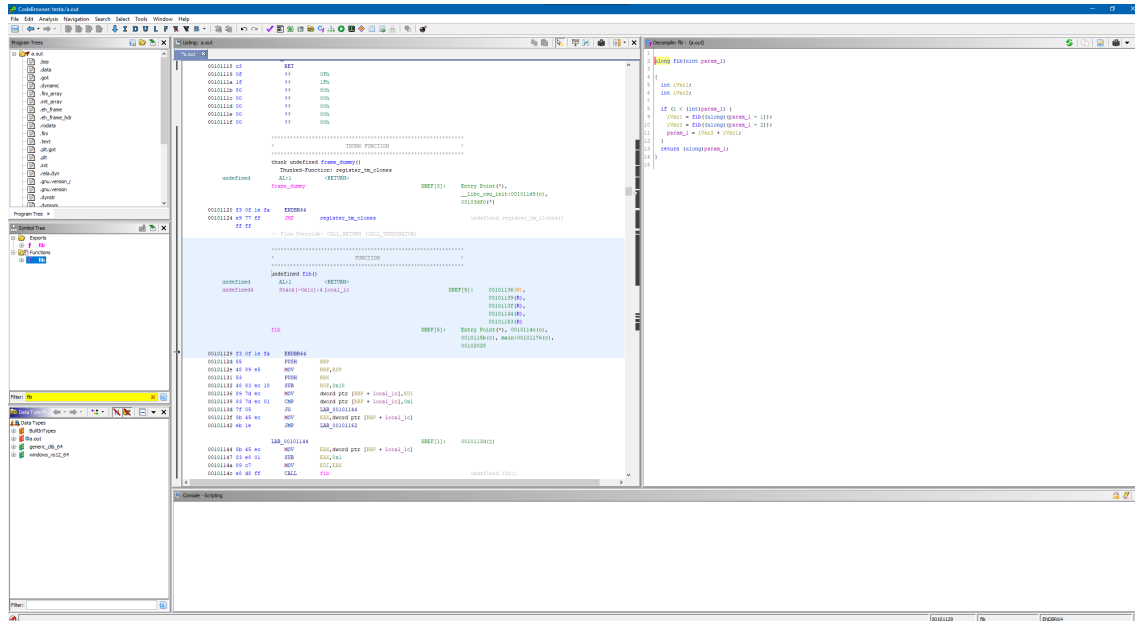
FIGURE 4: *Ghidra's standard project view*

Another major feature of Ghidra is the ability to extend its functionality through custom Java plugins [49, 43]. Some major aspects of its API are introduced below.

P-CODE

P-code is Ghidra's proprietary register transfer language, which aids as a strong tool to simplify further static analysis. Two ways how p-code achieves this are described below. [45]

For one, p-code allows for static analysis to be largely independent of the respective instruction set. Each instruction of every CPU instruction set supported by Ghidra is mapped to a sequence of p-code operations. These operations' purpose is to represent every operation that occurs during the execution of the respective instruction. For example, the ADD instruction in the x86 instruction set is not only mapped to a mathematically equivalent p-code operation, but also to other operations that each set one of the flags that are set upon execution of ADD. For all p-code operations, "parts of the processor state [serve] as input and output variables". These variables are introduced in the next section. [45]

Additionally, when using the API, the p-code is automatically analysed and then optimized. The directly mapped operations, before any optimizations are performed, are called raw p-code. [45] In its optimized form, the p-code is transformed into static single assignment form [7]. Beyond that, operations that have no effect are excluded, and some operations are replaced by fewer, more complex ones.

The most relevant example for such operations is MULTIEQUAL, which serves as a phi operation and merges different execution paths [46].

Varnode

In and of itself, a varnode purely represents a memory or register location, and consists of three components: the base address space, the offset and the varnode's size. By being used as input or output to a p-code operation, as mentioned above, they might be interpreted to be either an integer, a floating point number or a Boolean value. [45]

Through the API, further information about a varnode can be recalled. There are, for example, methods to get all descendants of a varnode, or to get the corresponding high-level variable, if it exists. [47]

Other important features

Ghidra's API supports, as mentioned in section P-code, static single assignment form. Beyond that however, functions to directly support back- and forward-slicing [41] as well as the automatic detection and retrieval of basic blocks are provided [44] as well.

3 SPECIFICATION

First, general weaknesses that the module detects are introduced. Afterwards, the included libc [28] functions are defined.

3.1 GENERAL WEAKNESSES

In this section, the general weaknesses that the module detects are defined. When choosing what weaknesses are included in this subset, two factors are primarily considered. First, the score assigned to each weakness by the CWE [10], which attempts to describe its prevalence and severity [11]. Second, because of the limited scope of this report, the ease of implementation of an algorithm detecting the respective vulnerability is also considered.

Further, a differentiation is made between potentially and inherently unsafe functions. For all weaknesses considered in this report, the deciding factor of whether a function is potentially or inherently unsafe stems from where the function input is derived from. If the relevant function input is passed as a parameter the function is considered only potentially unsafe, since the calling function can handle all necessary parameter checks. If the function input is derived from user input within the function, e.g. through the command line or from a file belonging to the user, there is no way for the calling function to sanitize it. Therefore, the function is considered inherently unsafe. To illustrate this idea, an example of two equivalent functions, one potentially, the other inherently unsafe, can be seen in listing 3.1.

```
1   void potentialOutOfBoundsRead(char* buffer, int n) {
2       printf("%c", buffer[n]);
3   }
4
5   void inherentOutOfBoundsRead(char* buffer) {
6       int index;
7       scanf("%i", &index);
8
9       printf("%c", buffer[index]);
10  }
```

LISTING 3.1: *Two functions with an Out-of-bounds Read vulnerability, one potential, the other inherent*

Weaknesses stemming from insufficient restrictions of operations on a buffer take up the largest proportion of the subset. Such buffer errors are the most common type of weakness in C programs [62]. Additionally, the class of weaknesses containing such errors were also assigned the highest

score by the CWE in 2019 [12] and remained in the top five in 2020 [11]. This is, in part, because the scores assigned by the CWE consider the prevalence of the respective weakness [12, 11]. However, buffer errors are also of critical severity, as they violate all three components of the CIA security triad [5]. For example, they often cause memory corruption, allow the unauthorized reading of and writing to memory and, in some cases, can even lead to the execution of malicious code [21].

Another class of weaknesses included is related to the otherwise erroneous handling of pointers. One of the weaknesses in this class is the dereference of potential NULL pointers. Also a common type of vulnerability [12], dereferencing a NULL pointer will cause undefined behaviour, often prematurely ending execution [52] or, in rare cases where NULL is interpreted as memory address 0, cause the execution of code [16]. Use After Free and Double Free are considered part of this class as well.

In table 2, each specific type of vulnerability the module detects, both inherently and potentially unsafe implementations, is listed.

Table 2: *List of general weaknesses that the module detects, information taken from CWE [10]*

Weakness name	CIA Violation	Description
Out-of-bounds Read	C	Memory outside of the buffer is read
Out-of-bound Write	IA	Memory outside of the intended buffer is written to
Classic Buffer Overflow	CIA	A larger amount of data is written into a buffer than its size
Use of Out-of-range Pointer Offset	CIA	Performing arithmetic on valid pointer, pointing to invalid position
NULL Pointer Dereference	CIA	Dereferencing memory that does not point anywhere
Use After Free	CIA	Dereferencing memory after it has been deallocated
Double Free	CIA	Attempting to deallocate already deallocated memory

An example for Out-of-bounds Read can be seen in the previous listing, 3.1. Out-of-bounds write is perfectly analogous, of course with the difference that the memory is written to instead of read. The same goes for Use of Out-of-range Pointer Offset, except that instead of a straightforward memory access as in Out-of-bounds Read, pointer arithmetic is performed first.

A basic example of a function with a Null Pointer Dereference weakness is depicted in listing 3.2 The function pointerDereference simply dereferences the pointer it received as a parameter without any further checks. Therefore, if the variable ptr is NULL, this function will cause undefined behaviour. This can be avoided by a simple NULL check beforehand, where the printf statement is only executed if ptr is not NULL.

```
1  void pointerDereference(int* ptr) {
2      printf("%i This is potentially unsafe", *ptr);
3  }
```

Listing 3.2: *Function with potential Null Pointer Dereference vulnerability*

Since there is no way in C to validate that memory that a pointer points to has been deallocated using the free function [17, 68], the module only detects the Use After Free and Double Free weaknesses if all relevant free instructions, as well as the dereference in the case of Use After free, occur within the function. These weaknesses can arise from mistakes made in the control flow of a function. A simple exemplary case of such a mistake can be seen in listing 3.3.

```
1  void freeVulnerabilities(int a, int b, int c, int* ptr) {
2      if (a < 10)
3          free(ptr);
4      if (b > a)
5          free(ptr);
6      if (c == 100)
7          printf("%i", *ptr);
8  }
```

Listing 3.3: *Function with control flow error, resulting in both potential Use After Free and potential Double Free vulnerability*

In listing 3.4, an instance of a function with an inherent Buffer Overflow vulnerability is shown. This example portrays a system 32 bit, 4 byte addresses. It is however completely applicable to 64 bit systems as well, with the exception of course the base pointer and addresses being twice the size. A buffer with a fixed size of 8 byte is declared, but user input is directly copied into it without any size checking whatsoever, so a potential attacker could input a tailored string to overwrite not only the previously declared integer variables, but also the return address saved in the stack frame. By overwriting the return address with the address of another function, an attacker can gain control of the control flow of the program or use other functions that are known to contain other vulnerabilities. This process is illustrated in figure 5. Since scanf [20] is used in listing 3.4, an attacker has to first fill the entire buffer, 8 bytes, followed by the two integer variables, 4 bytes each, and finally the saved base pointer, 4 bytes in this example, before he can overwrite the return address. As input string, a sequence of 20 random ASCII characters followed by the address of the function the attacker wishes to execute can be used, as each ASCII character is exactly one byte in size [1].

```
1  void bufferOverflow() {
2      int a = 0, b = 1;
3      char buf[8];
4
5      scanf("%s", buf);
6
7      ...
```

8 }

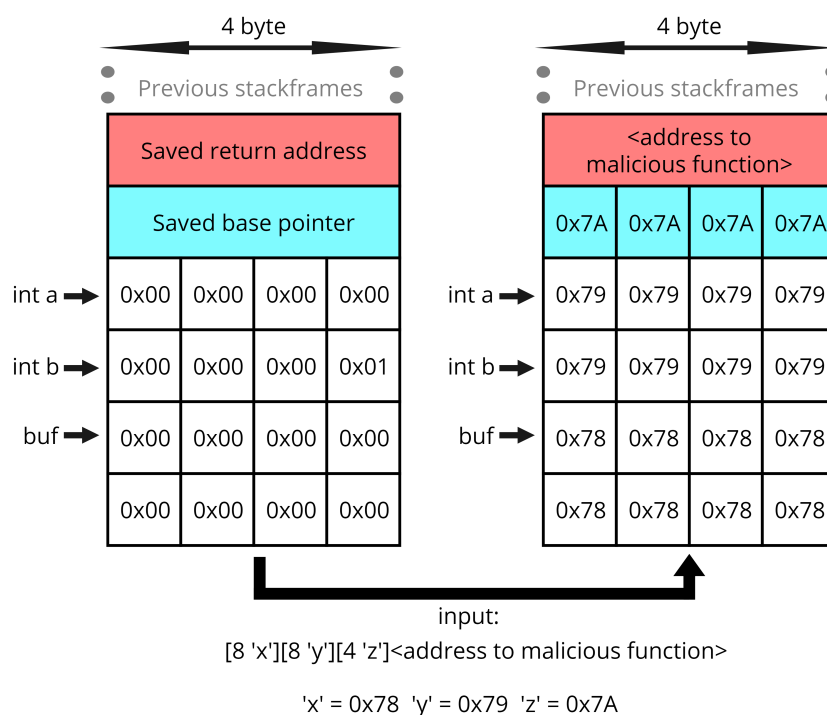Listing 3.4: *Function with Buffer Overflow vulnerability*

Figure 5: *A representation of the stack frame of the function in 3.4, before and after execution of line 5*

This example serves to visualize an attack using a common buffer error. In most real world scenarios, there are multiple mechanisms preventing this specific type of attack, such as the stack canary [30] and Address Space Layout Randomization (ASLR) [34].

3.2 Libc functions

The libc functions that the module detects are a subset of the top 30 most used functions that contain vulnerabilities according to the paper 'Detection of security vulnerabilities in C language applications' [6]. Additionally, the *gets* function is also detected, as it is considered to be inherently unsafe by the CWE [14]. All considered libc functions can be seen in table 3, along with their respective vulnerability.

Table 3: *List of libc functions that the module detects*

Vulnerability	Functions
Format string vulnerability	fprintf, printf, sprintf, snprintf
Buffer and memory error	gets, strlen, memset, memcpy, malloc, strcpy, memmove, atoi, strdup, calloc, strchr

4 METHODOLOGY

4.1 IMPORTANT HELPER FUNCTIONS

Prior to presenting the analysis structures, some of the utility function are quickly introduced. All of these can be found in the GitLab repository of this lab [25].

4.1.1 ISPOINTER

The function *isPointer* receives a variable as input and returns an integer that indicates whether said variable is a pointer. It is possible for *isPointer* to return a value indicating that nothing about the variable could be inferred.

A noteworthy source of false negatives in *isPointer* is that if the variable is multiplied anywhere within the function it is contained in, a value indicating that said variable is not a pointer is returned. The decision to include this check was made for two reasons. For one, there are only very limited cases where multiplying an address could be desired. The other reason is that, due to a specific check, the rate of correctly determined variables rises, even for variable that are not multiplied. This multiplication check is only relevant in cases where none of the more definitive checks imply anything about whether the given variable is a pointer.

4.1.2 GETDEREFERENCINGINSTRUCTIONS

getDereferencingInstructions takes a variable as parameter, returning a set of all p-code operations that dereference a varnode containing the given variable. For this report, a variable is considered to be contained in a varnode if the varnode is returned by a forward-slice performed on the variable.

4.1.3 REACHABLEIF

The purpose of *reachableIf* is to determine whether a specific p-code operation is reachable from the function entry point, given various conditions. It is a recursive function that receives a basic block, a p-code operation and a variable as input, as well as two functions. On its initial call, the passed in basic block has to be the first basic block in the function that is being analysed. If this is not the case, its behaviour is undefined. There are various other parameters for *reachableIf*, all of which however are purely relevant to its implementation details.

The first basic block of a function is always reachable. Because of this, it is marked as vulnerable. Upon entering *reachableIf*, the variable is checked for relevant reassignments within the current block. This first check is implemented by means of the function parameter *reassignmentCondition*.

reassignmentCondition

Implementations for this function receive a basic block as well as a variable and a p-code operation as input. It then iterates over all the p-code operations contained within the passed in basic block, returning the integer value *1* if the value of the variable is changed in a way that would make passing through the basic block on execution unsafe. It returns *-1* if the variable is instead changed so that the basic block is inherently safe. *0* is returned if the value of the variable is not, or at least not in any relevant way. The considered modifications of the variable here are subtraction, addition, multiplication, division and direct reassignment. In case of the phi operation MULTIEQUAL [45], see P-code and Single static assignment form, *reassignmentCondition* applies the same checks to each argument to MULTIEQUAL. If the current block also contains the p-code instruction tested for reachability, the order of the reassignment and the p-code is considered.

The definitions of safe, unsafe and irrelevant changes depend on the implementation of the function passed to *reachableIf* as *reassignmentCondition*. An example for such an implementation is the one for the Null Pointer Dereference analyser. Here, in case of a Null assignment to the variable, *1* would be returned. If however a constant value unequal to *0* would be assigned to it, it would be guaranteed to be not Null for all following basic blocks, at least up until the next reassignment. Because of this, *-1* would be returned, indicating that this basic block is inherently safe.

For the sake of the scope of this lab, value tracing is only performed in case of direct reassignment. Otherwise, only constant checks are applied, returning 1 for non-constant modifications.

After the reassignment check, the blocks that the current one can flow into are iterated over. On each of them, *reachableIf* is called recursively. The information whether the previous block was vulnerable is passed in. If so, the now current block is considered vulnerable as well. This can, as was just explained, change after the *reassignmentCondition* is checked for the new block.

However, guard statements at the end of the previous block can cause the new one not to be vulnerable, even if the previous one was. This is accounted for by previously executing the *edgeCondition*.

edgeCondition

The second of the two aforementioned functions is *edgeCondition*, which in turn takes two basic blocks as parameter, in addition to a variable. Essentially, this function serves to check if the first basic block's exit point can possibly flow into the other's entry point under certain conditions. The considered conditions, in analogue to *reassignmentCondition* depend purely on the implementation of the function passed to *reachableIf*. In order to achieve this, *edgeCondition* analyses the outflow condition of the first block. This is done by getting the flag or Boolean varnode that is used as an argument to CBRANCH [45], then analysing its definition. If the guard statement prevents the first block from flowing into the other, false is returned. Otherwise, *edgeCondition* returns true. For example, the function *leadsToIfNull* is passed to *reachableIf* as *edgeCondition* for the Null Pointer Dereference analysis. *leadsToIfNull* returns false only if the guard statement guarantees the the passed in variable can not be Null for the first block to flow into the other.

Similarly to *reassignmentCondition*, only very basic value tracing is performed, once again due to scope. If a parameter variable is compared to a varnode that is itself entirely made out of

parameters, the analyser assumes this to be a valid guard statement. The behaviour is analogous for locally declared variables.

Using these two helper functions, *reachableIf* recursively iterates over all possible paths, returning a Boolean value indicating whether the passed in basic block is reachable from the function entry point with the variable in an unsafe state, where either no guard statements prevent the function from flowing into the relevant block or where it was reassigned in such a way that the guard statements become ineffective. A class variable defines the calculation depth, defining how many times a block may be visited.

The various implementations of *edgeCondition* and *reassignmentCondition* will not be presented. All of them follow the respective structure explained above however, only the conditions being checked changing based on the implementation. As mentioned before, the implemented versions can be found in [25].

4.2 Analysers

Prior to reading this section, it is recommended to read section Important helper functions, especially the subsection reachableIf and its sections edgeCondition and reassignmentCondition. These helper function's functionality will not be explained within the descriptors of the analysers.

In this section, the analysis structure used for each vulnerability is presented. Each function is individually analysed. This is the case for all analysers described below.

4.2.1 Null Pointer Dereference

The Null pointer dereference analysis attempts to detect any dereference of what is, at runtime, potentially a Null pointer.

At the start of the analysis, each of the local variables of the current function is assigned a value. This value is the corresponding return value of *isPointer* and thus describes whether the variable is a pointer or not. After the assignment has finished, the analyser iterates over every dereference contained in the given function, retrieved by *getDereferencingInstructions*.

For each dereference, the dereferenced varnode is tested for contained variables by performing a backward slice on it. All of the variables returned by the backward slice are then added to a collection. If it is empty, the current iteration is concluded. Otherwise, the collection is then filtered further by removing the variables irrelevant for the rest of the analysis. If the collection contains one or more variables that have been inferred to be pointers, all non-pointers are removed from the collection. Additionally, if the user chooses the option to prefer false negatives, each of the variables for which *isPointer* could not infer whether it is a pointer are removed as well. The possible states that the collection can result in by following these rule are as follows: The collection includes only non-pointers and 'unknowns', only pointers and 'unknowns', only pointers or only 'unknowns'.

Once the collection has been filtered, each included variable is passed into *reachableIf*, along with the corresponding dereference operation and basic block.

4.2.2 Out-of-bounds Access

This analyser's purpose is to detect any access, whether read or write, to a memory location on the heap that makes use of an offset that might be negative or larger than the space allocated for the given location.

Similarly to the beginning of the Null Pointer Dereference analyser, every variable is assigned a value which indicates whether it is a pointer. This is, once again, achieved using *isPointer*. The analyser then iterates over all dereferences.

For Out-of-bounds Access's analyser, there is an early special case which does not require using *reachableIf*. If all of the variables contained in the current dereference are guaranteed to not be pointers, the dereference is considered a vulnerability by default and the analysis for this dereference is finished. It also does not matter in this special case whether the variables stem from within the function or were passed in as parameters, the dereference is considered as inherently vulnerable either way. The reasoning for this choice is that numeric variables, for example an integer, should not be used to store addresses, instead using a pointer.

If this special case does not apply, all pointers and 'unknowns' are both inserted into a new collection. This collection is then iterated over, treating the current variable of each iteration as the pointer, the base of the current dereference. All other variables contained in the collection are treated as the offset. If the user chooses the option to prefer false negatives, each offset variable is tested by means of *reachableIf* individually. As long as any of them satisfies all conditions, the whole current set of offset variables is considered safe. If however the user chooses to prefer false positives, all offset variables are passed into *reachableIf* at once, using yet another collection. If this method is chosen, the varnodes that contain exactly these variables will be tested. A varnode, that, for example, contains only two out of three offset variables will be considered irrelevant for the sake of simplifying the analysis, at the cost of not detecting various edge-cases.

Different from the Null Pointer Dereference analyser, two runs of *reachableIf* are required for each set of offset variables (or respectively per variable in each set, depending on what option the user chooses). The first run checks whether the offset can potentially be negative, thus trying to access memory space before the location that is being pointed to. The second one analogously tests whether the offset is likely to be a value that is larger than the space allocated to the pointer. Because of these two separate runs, two implementations for each of the two helper functions are necessary.

4.2.3 Buffer Overflow

The Buffer Overflow analyser detects cases where an attempt is made to store more data into a buffer than its size allows. It is firmly based on Out-of-bounds Access's analyser.

On initiating the analysis, this analyser functions in the same manner as the two previous ones by gathering all dereferences in the current function and determining which of the local variables are pointers. After those first two steps are completed, all LOAD [45] instructions are removed from the collection storing the dereferences. This way, only the STORE [45] instructions remain for the further analysis. The collection of relevant dereferences is filtered once more however. All

of the remaining instructions that are not contained within a loop or whose offset is not modified during iteration are removed.

After the collection of dereferences has been filtered, the rest of the analysis is fully equivalent to that of the Out-of-bounds Access analyser, with the exception that the offset is only tested for possibly being too large. A negativity test is not performed, since Buffer Underflow is technically a different class of vulnerability from Buffer Overflow.

4.2.4 Use-after-free and Double Free

The Use-after-free analyser detects any dereference of a pointer that might occur after the location it points to has been deallocated. Analogously, the Double Free analysis detects additional deallocations. Both of these are rudimentarily implemented, as logically mutually exclusive statements are not considered in the analysis. If, for example, there are two if statements in a function that can not possibly both be true, the paths that visit both of them are still considered as valid paths. An example for such a function can be seen in listing 4.1.

```
1  void doubleFree(int * a, int b) {
2
3      if (b > 100) {
4          printf("b was large %i", *a);
5          free(a);
6      }
7
8      if (b < 10) {
9          printf("b was small %i", *a);
10         free(a);
11     }
12 }
```

Listing 4.1: *An example for a function with two logically mutually exclusive statements*

Both analysers are methodically very similar to each other. At the beginning of the analysis, all p-code operations contained in the current function are inserted into a list, which is then filtered by operators. All operations that do not use the CALL [45] operator are removed, only function calls remaining in the list. The remaining items are then iterated over, each being checked for the function they call. Only if the called function is a library function and called "free" does the current item remain in the list. Otherwise, it is removed.

Once the list has been filtered, each variable of the function is iterated over. In each iteration, the aforementioned list is filtered once again to only include the function calls that take the current variable as its first parameter. An additional collection is created, containing all operations that reassign the current variable. In the case of the Use-after-free analyser, yet another collection is created, this one containing all dereferences of the variable.

Both of these analysers are different from the other analysers presented in this report in that they do not make use of *reachableIf*. Instead, they use a conceptually similar function with a slightly

differing implementation. The recursive traversal of the paths is implemented in the same way. The function however does not take an *edgeCondition* or a *reassignmentCondition* as parameter. Instead it simply calls itself on all outflowing blocks. Upon entry in this function, which, on its initial call, receives all created collections as well as the first basic block in the function as input, all operations in the basic block are iterated over. If the current operation deallocates the variable, the variable is marked as such. If it is reassigned, this mark is removed again.

In the case of Double Free, if the variable is deallocated within the current operation while already being marked as deallocated, the function returns true. This indicates that the function contains a Double Free vulnerability for the current variable. For Use-after-free's analyser, the function instead checks whether the operation dereferences the variable while being marked as deallocated. If so, true is returned.

5 SUMMARY

This report presents a static approach for the detection of various vulnerabilities identified by the CWE, all of which have been implemented in the course of the lab. The implementation is realized as a plug-in for the reverse-engineering tool Ghidra, thus also making strong use of its API. Namely, the vulnerabilities detected are Null Pointer Dereference, Out-of-bounds Access, Buffer Overflow as well as Use-after-free and Double Free. For each of these vulnerabilities, the methodology used in their respective detection algorithms is presented. In addition to detecting the aforementioned vulnerabilities, the module presented in this report also detects the usage of various libc functions. These libc functions are a subset of the most used libc functions containing vulnerabilities, according to the paper 'Detection of security vulnerabilities in C language applications' [6]. In addition to these, the function *gets* is detected as well.

Each case of a detected vulnerability (with the exception of Use-after-free and Double Free), as well as each libc function, is assigned into a group, either being *potentially* or *inherently* unsafe. The definitions of these groups has been simplified for the sake of the scope of this lab. A vulnerability is *inherently* unsafe if the vulnerable variable is locally declared. Vice versa, the vulnerability is *potentially* unsafe if the variable stems from outside the function, being passed in as a parameter.

A short evaluation of the analysis is given below. Afterwards, some of the major limitations of the implementations are explained.

5.1 EVALUATION

To give a rough intuition about the plugins rate of successful detections, a small C program is analysed. The results can be seen in table 4 below. In case different results are returned if the option to prefer false negatives is chosen, the results with the better rate of correct detections are reflected in the table.

TABLE 4: *Results of analysing eval.c [26]*

Vulnerability	Amount of correctly detected functions	Amount of falsely detected functions
Null Pointer Dereference	10 / 12	0
Out-of-bounds Access	17 / 20	1
Buffer Overflow	7 / 10	1
Double Free and Use-after-free	4 / 5	0

The functions contained in the program were not written with the plugin's implementation in mind - meaning, they do not actively play into the plugin's strength or weaknesses. However, it should be mentioned that the majority of the functions are kept relatively short and simple.

5.2 Limitations / Future Work

Due to the limited scope and available time for this lab, a number of important aspects have only been implemented rudimentarily, if at all.

The most relevant one is value tracing which, in the large majority of cases, has only been implemented in a basic fashion. As explained in Methodology, to check what variables a Varnode "contains", a backward slice is performed. However, the returned set does not contain any information about how these variables come together to form the varnode. Therefore, any checks that make use of at least one varnode are at best approximated. Improving the value tracing where it is implemented and implementing it where it is not would inherently lead to far stronger detection results.

In its current implementation, all functions contained within the program are iterated over, each being individually analysed. Another valuable improvement to the analysis would be to instead recursively walk over all functions, analysing each function before its calling functions. In addition to this, analysis of the possible return values and modifications of parameters would be performed as well. This would then allow for the better detection of vulnerabilities in calling functions, vastly improving *reassignmentCondition*. Take, for example, a function that contains a reassignment to a variable using the return value of another function as input. Only if the called function has been inferred to possibly return *o* will the reassigned variable now be cause for a potential Null pointer dereference, at least until the next reassignment.

Of course, as is virtually inherently the case for this type of analysis, there is a vast amount of smaller fixes and improvements that can be applied to the analysis concepts used.

References

[1] ABERDEEN, Godred Fairhurst - University of: *ASCII*. URL: https://erg.abdn.ac.uk/users/gorry/eg2069/ascii.html (visited on 01/04/2021).

[2] BBC: *Instructions*. URL: https://www.bbc.co.uk/bitesize/guides/z2342hv/revision/2#:~:text=Languages%20can%20be%20defined%20as,mechanical%20workings%20of%20the%20CPU. (visited on 12/31/2020).

[3] BBC: *Types of programming languages*. URL: https://www.bbc.co.uk/bitesize/guides/z4cck2p/revision/1 (visited on 01/02/2021).

[4] BHAT, Omkar ; YEPREM, Zoya ; LINGESH, Vijay: *Comparison of 3 Reverse Engineering Tools*. 2019.

[5] BMCBLOGS: *What is the CIA Security Triad?* URL: https://www.bmc.com/blogs/cia-security-triad/ (visited on 01/04/2021).

[6] BOUDJEMA, El Habib et al.: "Detection of security vulnerabilities in C language applications". In: *Security and Privacy* 1.1 (2018), e8. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spy2.8. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spy2.8.

[7] BULAZEL, Alexei: *Working With Ghidra's P-Code To Identify Vulnerable Function Calls*. URL: https://www.riverloopsecurity.com/blog/2019/05/pcode/ (visited on 04/04/2021).

[8] CAMACHO, Claudio M.: *How car manufacturers are becoming software companies*. URL: https://www.tuxera.com/blog/how-car-manufacturers-are-becoming-software-companies/ (visited on 12/29/2020).

[9] COLLEGE, Andrew Clifton - Fullerton: *Intel vs AT&T syntax*. URL: http://staffwww.fullcoll.edu/aclifton/courses/cs241/syntax.html (visited on 01/03/2021).

[10] CWE. URL: https://cwe.mitre.org/index.html (visited on 12/29/2020).

[11] CWE: *2020 CWE Top 25 Most Dangerous Software Weaknesses*. 2020. URL: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html (visited on 11/21/2020).

[12] CWE: *CWE - 2019 CWE Top 25 Most Dangerous Software Errors*. 2019. URL: https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html (visited on 11/21/2020).

[13] CWE: *CWE VIEW: Software Development*. 2020. URL: https://cwe.mitre.org/data/definitions/699.html (visited on 12/04/2020).

[14] CWE: *CWE-242: Use of Inherently Dangerous Function*. 2020. URL: https://cwe.mitre.org/data/definitions/242.html (visited on 12/04/2020).

[15] CWE: *CWE-676: Use of Potentially Dangerous Function*. 2020. URL: https://cwe.mitre.org/data/definitions/676.html (visited on 12/04/2020).

[16] CWE: *NULL Pointer Dereference*. URL: https://cwe.mitre.org/data/definitions/476.html (visited on 01/04/2021).

[17] DIE.NET: *free(3) . Linux man page*. URL: https://linux.die.net/man/3/free (visited on 01/04/2021).

[18] DIE.NET: *gcc(1) - Linux man page*. URL: https://linux.die.net/man/1/gcc (visited on 12/31/2020).

[19] DIE.NET: *objdump(1) - Linux man page*. URL: https://linux.die.net/man/1/objdump (visited on 12/31/2020).

[20] DIE.NET: *scanf(3) - Linux man page*. URL: https://linux.die.net/man/3/scanf (visited on 01/04/2021).

[21] DU, Dr. Wenilang: *Computer Security: A Hands-on Approach*. CreateSpace, 2017.

[22] EDWARDS, Greg: *Technology in Everyday Life*. URL: https://www.jfg-nc.com/technology-in-everyday-life/ (visited on 12/29/2020).

[23] EETIMES, embedded: *2019 Embedded Markets Study*. 2019. URL: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf (visited on 12/04/2020).

[24] ENISA: *Glossary - Enisa*. URL: https://www.enisa.europa.eu/topics/threat-risk-management/risk-management/current-risk/risk-management-inventory/glossary (visited on 12/29/2020).

[25] *Git Repository of the plugin*. URL: https://git.cs.uni-bonn.de/baierd/pg_ghidra_plugin (visited on 04/20/2021).

[26] *Git Repository of the plugin*. URL: https://git.cs.uni-bonn.de/baierd/pg_ghidra_plugin/-/tree/master/Praktischer%20Teil (visited on 04/20/2021).

[27] GNU: *Basic blocks*. URL: https://gcc.gnu.org/onlinedocs/gccint/Basic-Blocks.html (visited on 04/05/2021).

[28] GNU: *The GNU C Library Reference Manual*. URL: https://www.gnu.org/software/libc/manual/html_mono/libc.html (visited on 04/03/2021).

[29] GURU99: *Phases of Compiler*. URL: https://www.guru99.com/compiler-design-phases-of-compiler.html (visited on 12/31/2020).

[30] HAT, Huzaifa Sidhurwala - Red: *Security Technologies: Stack Smashing Protection*. 2018. URL: https://access.redhat.com/blogs/766093/posts/3548631 (visited on 01/04/2021).

[31] HOPE, Computer: *Assembly Language*. URL: https://www.computerhope.com/jargon/a/al.htm (visited on 01/02/2021).

[32] HOPE, Computer: *Disassembler*. URL: https://www.computerhope.com/jargon/d/dissembl.htm (visited on 12/30/2020).

[33] INFOSEC: *The Basics of Ida Pro*. URL: https://resources.infosecinstitute.com/topic/basics-of-ida-pro-2/ (visited on 12/30/2020).

[34] KAPERSKY: *Address Space Layout Randomization (ASLR)*. URL: https://encyclopedia.kaspersky.com/glossary/address-space-layout-randomization-aslr/ (visited on 01/04/2021).

[35] LEXTRAIT, Vincent: *The Programming Languages Beacon*. 2016. URL: https://www.mentofacturing.com/Vincent/implementations.html (visited on 12/04/2020).

[36] MazeGen: *X86 Opcode and Instruction Reference*. 2017. url: http://ref.x86asm.net/coder64.html (visited on 01/03/2021).

[37] Megira, S ; Pangesti, A R ; Wibowo, F W: "Malware Analysis and Detection Using Reverse Engineering Technique". In: *Journal of Physics: Conference Series* 1140 (2018), p. 012042. url: https://doi.org/10.1088/1742-6596/1140/1/012042.

[38] Merriam-Webster: *Definition of reverse engineer*. url: https://www.merriam-webster.com/dictionary/reverse%20engineer (visited on 12/30/2020).

[39] NetMarketShare: *Operating System Market Share*. 2020. url: https://www.netmarketshare.com/operating-system-market-share.aspx?options=%7B%22filter%22%3A%7B%22%24and%22%3A%5B%7B%22deviceType%22%3A%7B%22%24in%22%3A%5B%22Desktop%2Flaptop%22%5D%7D%7D%5D%7D%2C%22dateLabel%22%3A%22Custom%22%2C%22attributes%22%3A%22share%22%2C%22group%22%3A%22platform%22%2C%22sort%22%3A%7B%22share%22%3A-1%7D%2C%22id%22%3A%22platformsDesktop%22%2C%22dateInterval%22%3A%22Monthly%22%2C%22dateStart%22%3A%222020-04%22%2C%22dateEnd%22%3A%222020-04%22%2C%22hiddenSeries%22%3A%7B%7D%2C%22segments%22%3A%22-1000%22%7D (visited on 12/04/2020).

[40] Ninja, Binary: *Binary Ninja Features*. url: https://binary.ninja/features/ (visited on 12/30/2020).

[41] NSA: *DecompilerUtils*. url: https://ghidra.re/ghidra_docs/api/ghidra/app/decompiler/component/DecompilerUtils.html (visited on 04/04/2021).

[42] NSA: *Ghidra*. url: https://www.nsa.gov/resources/everyone/ghidra/ (visited on 01/03/2021).

[43] NSA: *Ghidra Official Examples*. url: https://github.com/NationalSecurityAgency/ghidra/tree/master/Ghidra/Extensions/sample/src/main/java/ghidra/examples (visited on 01/10/2021).

[44] NSA: *HighFunc*. url: https://ghidra.re/ghidra_docs/api/ghidra/program/model/pcode/HighFunction.html (visited on 04/04/2021).

[45] NSA: *P-code Reference Manual*. url: https://ghidra.re/courses/languages/html/pcoderef.html (visited on 04/04/2021).

[46] NSA: *P-code Reference Manual*. url: https://ghidra.re/courses/languages/html/additionalpcode.html (visited on 04/04/2021).

[47] NSA: *Varnode*. url: https://ghidra.re/ghidra_docs/api/ghidra/program/model/pcode/Varnode.html (visited on 04/04/2021).

[48] O'Connor, Stephen: *The Importance of Medical Software in Today's Healthcare Practices*. url: https://www.adsc.com/blog/the-importance-of-medical-software-in-todays-healthcare-practices (visited on 12/29/2020).

[49] O'Reilly: *Overview of Ghidra*. url: https://www.oreilly.com/library/view/getting-started-with/9781098115265/ch01.html (visited on 01/05/2021).

[50] Oracle: *Instructions*. url: https://docs.oracle.com/cd/E19120-01/open.solaris/817-5477/ennby/index.html (visited on 01/03/2021).

[51] Oracle: *Statements*. url: https://docs.oracle.com/cd/E19120-01/open.solaris/817-5477/eoqjt/index.html (visited on 01/03/2021).

[52] OWasp: *Null Dereference*. URL: https://owasp.org/www-community/vulnerabilities/Null_Dereference (visited on 01/04/2021).

[53] Mark Stamp PETER STAVROULAKIS, ed.: *Handbook of Information and Communication Security*. Springer, 2010.

[54] PRESS, AP - Associated: *German hospital hacked, patient taken to another city dies*. URL: https://apnews.com/article/technology-hacking-europe-cf8f8eee1adcec69bcc864f2c4308c94 (visited on 12/29/2020).

[55] PROF. DR. PETER MARTINI Prof. Dr. Elmar Padilla, Martin Clauß: *Lecture: Program Analysis and Binary Exploitation*. URL: https://net.cs.uni-bonn.de/wg/cs/teaching/wt-202021/pabe/ (visited on 01/11/2021).

[56] PROJECT, The Linux Documentation: *Pros and Cons*. URL: https://tldp.org/HOWTO/Assembly-HOWTO/x133.html#:~:text=The%20advantages%20of%20Assembly,software%20threads%20or%20hardware%20devices (visited on 01/02/2021).

[57] RAD, Hamid: *Reverse Engineering as a Learning Tool in Design Process*. 2012.

[58] RAYMOND, Eric: *The Art of UNIX Programming*. Addison-Wesley, 2003.

[59] ROUSE, Margaret: *decompile*. URL: https://whatis.techtarget.com/definition/decompile (visited on 12/31/2020).

[60] ROUSE, Margaret: *reverse engineering*. URL: https://searchsoftwarequality.techtarget.com/definition/reverse-engineering (visited on 12/29/2020).

[61] SHORTJUMP!: *Ghidra: A quick overview for the curious*. URL: https://0xeb.net/2019/03/ghidra-a-quick-overview/ (visited on 12/30/2020).

[62] SOFTWARE, WhiteSource: *Most secure programming languages - WhiteSource*. 2020. URL: https://www.whitesourcesoftware.com/most-secure-programming-languages/ (visited on 12/04/2020).

[63] TECHOPEDIA: *Low-Level Language*. URL: https://www.techopedia.com/definition/3933/low-level-language (visited on 01/02/2021).

[64] TECHTARGET: *Machine code*. URL: https://whatis.techtarget.com/definition/machine-code-machine-language (visited on 01/05/2021).

[65] TECHTERMS: *Assembler*. URL: https://techterms.com/definition/assembler (visited on 12/30/2020).

[66] TECHTERMS: *Register*. URL: https://techterms.com/definition/register (visited on 01/03/2021).

[67] TIOBE: *TIOBE Index for December 2020*. 2020. URL: https://www.tiobe.com/tiobe-index/ (visited on 12/04/2020).

[68] UMMIT, Steve: *Pointer Allocation Strategies*. URL: https://www.eskimo.com/~scs/cclass/int/sx7.html (visited on 01/04/2021).

[69] UNIVERSITY, John Burkardt - Florida State: *C Program Compilation*. URL: https://people.sc.fsu.edu/~jburkardt/classes/isc_2012/c_program_compilation.pdf (visited on 01/02/2021).

[70] UNIVERSITY, Weber State: *Variables and Memory Addresses*. URL: http://icarus.cs.weber.edu/~dab/cs1410/textbook/4.Pointers/vars_address.html (visited on 12/31/2020).

[71] WIKI, OSDev: *Assembly*. URL: https://wiki.osdev.org/Assembly (visited on 01/03/2021).

List of Figures

List of Tables

LISTINGS

Statement of Authorship

I hereby confirm that the work presented in this lab report has been performed and interpreted solely by myself except where explicitly identified to the contrary. I declare that I have used no other sources and aids other than those indicated. This work has not been submitted elsewhere in any other form for the fulfilment of any other degree or qualification.

Bonn, April 22, 2021

_____
Luca Weist