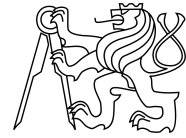


CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title: Application Security Analysis
Student: Bc. Tomáš Kvasnička
Supervisor: Ing. Tomáš Zahradnický, Ph.D.
Study Programme: Informatics
Study Branch: Computer Security
Department: Department of Computer Systems
Validity: Until the end of winter semester 2017/18

Instructions

Get acquainted with methods and software used in reverse engineering of computer software. Use studied methods and software to perform vulnerability assessment of an application provided by the supervisor. Focus primarily on connections of the assessed application to the Internet. Document and assess all found vulnerabilities and give measures leading to their mitigation.

References

Will be provided by the supervisor.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague September 8, 2016

CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS



Master's thesis

Application Security Analysis

Bc. Tomáš Kvasnička

Supervisor: Ing. Tomáš Zahradnický, Ph.D

8th January 2018

Acknowledgements

I would like to thank everyone helping me with this thesis. You people know who you are.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 8th January 2018

.....

Czech Technical University in Prague

Faculty of Information Technology

© 2018 Tomáš Kvasnička. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Kvasnička, Tomáš. *Application Security Analysis*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Hlavním cílem této práce je výzkum metod a software používaných v oblasti počítačové bezpečnosti, následovaný nezávislým posouzením zadané aplikace. Tato analýza se soustředí na potenciální bezpečnostní nedostatky. Opravdovým přínosem této práce je analyzovaná a ohodnocená reálně používaná aplikace, spolu s doporučeními snižující efekt nalezených problémů.

Klíčová slova počítačová bezpečnost, zranitelnost

Abstract

The aim of this work is research of methods & software used in computer security followed by an independent security review of a given application. This study focuses on potential security weaknesses. The real benefit of this thesis is analyzed and assessed real-world application along with recommendations leading to mitigation of found defects.

Keywords computer security, vulnerability

Contents

Introduction	1
1 About this thesis	3
1.1 Exact problem formulation	3
1.2 Expected results	8
1.3 Thesis structure	8
2 Research	11
2.1 Possible attack vectors	11
2.2 Known methods and attacks	19
2.3 Specialized software	27
3 Analysis	31
3.1 Application structure	31
3.2 Executables and libraries	36
3.3 In-depth inspection of selected parts	39
4 Assessment	45
4.1 Defects	45
4.2 Practical assessment	47
5 Mitigation & Recommendations	55
5.1 Unsecured network communication	55
5.2 Buffer overflow	56
Conclusion	61
Bibliography	63
A Acronyms	69

List of Figures

2.1	Stack frame structure in memory	13
3.1	<i>procmon</i> captured creation of <i>uninstall.exe</i>	33
3.2	Network communication after filtering out <i>Microsoft</i> servers	34
3.3	Network communication captured using <i>procmon</i>	35
3.4	<i>flawfinder</i> partial example output	41
3.5	Out-of-bound read found by <i>Coverity</i>	41
3.6	<i>strcpy</i> implementation by <i>jennifer.dll</i>	43
4.1	Downloading installer from SRV under normal circumstances	47
4.2	Intercepted connection	48
4.3	Tampered binary downloaded after poisoning ARP cache	49
4.4	Tampered binary stored locally	49

List of Tables

3.1	Security attributes of binaries	37
-----	---	----

Introduction

First of all, thank you all for being interested in this topic and for choosing my master's thesis as a source of information. I will try to provide you with the most relevant, accurate and up-to-date data.

This master thesis focuses on a security analysis of a selected, real-world application used worldwide by hundreds of millions people every day. The aim of the thesis is to get acquainted with methods and software used by security professionals, use this gained knowledge to perform a vulnerability assessment of this application and document all found issues while giving measures leading to their mitigation. This particularly means extensively researching multiple areas of computer security, performing various security tests against the given application and documenting the results.

What we want to achieve here is a correctly performed detailed security review of a foreign software project. This means we want to focus on download of the application, its installation procedure, its network communication and finally its inputs. We will examine each of these parts trying to detect weaknesses. Application's inappropriate behaviour leading to information disclosure, network breach or other similar issues is our objective. The thesis will be considered as a standalone security review performed by a 3rd party.

The main reason for performing this analysis is researching and summarizing existing information while gaining experience in security analysis field of computer science. Also, if real security threats get found during the process of creating this thesis, we will inform appropriate institutions & developers, make sure they understand the risks and try to cooperate with them on fixing all found bugs as soon as possible. Therefore every person in this world might benefit from this thesis by having a more secure application installed on their computers.

About this thesis

General phrases used in this thesis are inspired by [1]. The first chapter of this thesis introduces the thesis itself. We are going to explain the nature of the problem, remind what should a proper security analysis consist of, describe used environment along with possible attack scenarios and also introduce the structure of the thesis. This is an extension to the introduction to make sure we all understand the situation and what will be happening in the following chapters.

1.1 Exact problem formulation

In this thesis, we will focus on analyzing, testing and documenting given application from a security point of view. The reason for being interested in such a topic is that computers became something very essential in our everyday lives and a major part of the human population cannot imagine living without them anymore. In addition to that, we are nowadays using computers for tasks related to, e.g. national security, financing, emergency situations and most importantly our health. Therefore it is only reasonable to make sure computer applications get designed, developed and used in the most secure way they can be.

The application selected for analysis is a world-wide popular piece of software that has repeatedly been included in various online and paper magazines as a “must have” app for every single computer ¹. Due to this fact, we will unfortunately not be able to reveal the application’s actual name as in case any real deficiencies get found our thesis could put a significant amount of people in danger. On the other hand, it would not look appropriate to have a

¹ *We are not citing mentioned magazines on purpose, to minimize the risk of identifying inspected application.*

major part of the thesis blackened out, so we decided to give our application a new name - *Mary*, statistically the most used female name in the USA.

Mary aims to provide a complete solution for a complicated problem, which can be divided into two separate sub-problems. Each sub-problem gets solved by *Mary's* separate part, and each of these parts accepts many different input/output file types while offering various algorithms to work with. This should provide us with a right feeling that *Mary* is a truly large software project.

Security analysis is rather a complex process, so a more detailed description of what to expect follows in next subsections. We will first focus on a description of the analysis itself, then we are going to take a look at the environment we will be working in and last, but not least, are going to be brief comments on expected space for some problems.

1.1.1 Security analysis

A proper security review consists of researching every possible aspect in which targeted system can have weaknesses.[2][3][4] This process can be separated into multiple smaller parts to make it more easy to understand. We will now take a closer look at some of these parts.

- **Acquire** Although frequently overlooked, this is the part every sophisticated attacker will want to abuse the most. As long as the attacker can exploit the process of acquiring the application, he is also very likely able to inject his code into every computer that installs such an application. In the worst case scenario, this can mean unlimited and undetected control of the victim's station. Especially nowadays when software is usually downloaded from the Internet instead of being distributed over some physical medium (DVD, USB stick,) this process becomes one of the essentials that must be well protected. Since we will try to behave like a sophisticated attacker during our analysis, we will get familiar with obtaining *Mary* very well.
- **Install** Installation is highly dependent on given environment, but in any case, it represents the first place where foreign code may get executed on a user's machine. Because installation usually needs write access to system folders, changes important settings and generally modifies the way a computer is going to behave it requires administrator/root privileges - and anything running with these privileges must get checked with extreme caution. Therefore we will look at this process in great detail.

- **Network inputs** As long as targeted application communicates over the network, it has to accept at least some network input. Moreover, as long as an application accepts any input, it can be attacked. In case of network input, the situation is far more critical given the fact that the attack can be handled remotely - the victim does not even have to execute a single program. Also, under certain conditions, the application might be running with administrator privileges. These facts make the possibility of network input itself sound promising. Therefore, we will focus on *Mary* from this point of view as well.
- **Local inputs** Local inputs present a typically attacked part of an application. Although they require a little bit more interaction with the victim than network inputs (e.g. opening a malicious file) they are a more frequent way an application talks to its user. *Mary* is a great example for this part of the analysis, as it accepts a wide range of different file types - extensive research on this topic will be done in chapter 3.
- **Access rights** The last example of what must be included in every security analysis are access rights. They include for example filesystem access rights or user privileges. The default installation of *Mary* creates numerous objects, runs multiple processes and accesses various system entities. This is also going to be examined in a comprehensive way.

1.1.2 Chosen environment

First things that influenced our work environment were operating systems supported by *Mary*. *Mary* consists of multiple parts and libraries, each of them coming with their license, terms and conditions. Several parts of *Mary* can be compiled on Windows, Linux, macOS, FreeBSD and even Android builds exist. However, we wanted to test the application in its full version - to do so, this forces several conditions for the environment. We can now take a look at the overall setup scheme.

- **Operating system** Due to several facts, the chosen OS will be Windows 10. Windows is the only system where all parts of *Mary* can run, and its version 10 is very current at the time of writing this text. It is also well supported by Microsoft, offers various security mechanisms and is among the most popular operating systems used by the general public.[5][6] We will run this operating system using the Virtual Box virtualization software.
- **System settings** We will keep all the settings to their defaults, update the system periodically and let Windows handle all the maintenance as in the case of a typical user. The only difference, when compared to a real installation, is going to be the Virtual Box Guest Additions which

is a software package of Windows drivers especially suitable for a virtualized OS. Without these drivers, the system does not provide a good reaction time nor is it able to use all available hardware in an optimal way. The network will be configured using DHCP, and the addresses are going to be provided by the Virtual Box software.

- **Other installed software** As we try to keep the system as default as it can be, we want to keep 3rd party software at its minimum. On the other hand, we can not perform a security analysis without installing appropriate specialized tools that we will get to in section 2.3. Apart from these necessary utilities and *Mary*, we install only a few handy applications - Total Commander, Mozilla Firefox, Microsoft Visual Studio 2015, Google Drive, git, Python and Sublime Text. We especially want to avoid anti-virus programs, firewalls and other kinds of security software as those might interfere with the analysis.

1.1.3 Space for weaknesses

At the end of this section, we will briefly take a look at another important topic - *Mary's* weaknesses. Software development is a non-trivial process involving numerous activities - designing, implementation, documentation, testing, updates releasing and many others. All these activities must be done in a secure way to produce an all-around secured product.[7] As we will see later, this is unfortunately not always the case. Now, we can take a quick look at where we are going to expect security flaws in the case of *Mary*.

1.1.3.1 Download

As mentioned before in subsection 1.1.1, the first contact with the software vendor and the software itself might provide us with an opportunity of finding an unseen problem. A usual way for *Mary* to get to its users is to be downloaded from the Internet. This means that *Mary's* vendor should try to guarantee that the server we are downloading from is actually owned by the software vendor and is not just an identical copy of it created by some attackers. Also, we should have a way to check the integrity of the downloaded file to minimize the chance that the file is just a forged binary pretending to be *Mary*. If these two aspects are not safely covered, there might be an opportunity for an attacker.

1.1.3.2 Install

The next thing to focus on when looking for potential problems is the installation. In the middle between the download and the installation is the digital signature - the installer theoretically downloaded in the previous section should also be digitally signed. On recent Windows systems, this makes sure

that the User Account Control (UAC) is able to verify the signature and thus the software vendor. This guarantees that the file has not been tampered with. Next fact is that the installation may need administrative rights. If other binaries get executed from the installation process, and their content can get changed, the attacker could be able to force running custom code with the highest possible access rights. Also, nowadays installation procedures usually offer us various software products in addition to the one we are actually installing. When left with their default settings on they install these additional software products, so the whole principle of securing the application now applies to them as well. These and many others present a set of scenarios which can get abused.

1.1.3.3 Inputs

After the application's download and installation phase are considered to be reviewed, its inputs are to be examined. Generally, we can say that every application that accepts user input has to allocate memory to copy that input into. Attackers may try to test whether the size of the buffer is constant and try to provide more data than fits into such a buffer, trying to subvert the application into an unexpected state. When unexpected behavior is not just well expected but even controlled by the attacker, he/she might be able to take control of the execution flow by injecting his/her instructions.[8] For this reason, we must pay great attention to what *Mary* accepts as input file formats.

- **Local input** We will understand local input as any data that may be submitted to the program and do not originate from a network. It may include files, command line arguments, settings from registry keys, GUI interaction with the user and others. We will be most interested in *Mary's* file-based input.
- **Network input** On the other hand, all data processed by the application directly from the network will be considered a network input. For example communication with the update server, or targeted commercials. We will examine *Mary's* network inputs thoroughly as well.

1.1.3.4 Access rights

Access rights are one of the main security mechanisms present in modern operating systems. On our target platform access rights are not associated just with files, but many other system objects have them as well.[7] We will be most interested in access rights of registry keys and files themselves. These get created & set during the installation so they will be closely connected to the installation process. Another thing to look at can be an execution of

privileged actions by unprivileged users, manifests and UAC's role in these scenarios.

1.2 Expected results

The big picture end of this thesis is represented by analyzed application from a security professional's point of view. This means that we are in the first place interested in an overall research of this complex topic and if this research brings up something positive we also want to try to dive deeper into the practical side of things.

In greater detail, this means that we want to gain as much knowledge as we can about several topics - where to look for possible attack vectors, what are generally known attack scenarios and how to apply these scenarios to our situation. Also, we want to focus on methods that security professionals use when they are trying to test a given piece of software - what methods are suitable for what attack vectors, what can we do when we have target's source code and what on the other hand we can not or what extra methods can we use if the application communicates over the network. At the end of the research, we will want to take a look at software utilities that use discovered attack methods for suitable attack vectors. These and many others represent required knowledge to at least know where to start with our analysis.

After getting familiar with the necessary background, we want to focus on the review itself. We want to use the acquired knowledge, find out application weaknesses and see where they can lead to. This will be represented by utilization of appropriate software tools and proper application of discovered methods.

In the very end of the thesis we want to do a summary of everything we have found out, give recommendations to software vendors and draw an appropriate conclusion.

1.3 Thesis structure

Here we will briefly describe the structure of this thesis - it is divided into four main parts: Research, Analysis, Assessment and Recommendations.

1.3.1 Research

This will be the first part of this thesis coming right after this introduction. At the beginning of this chapter, we are going to do a short reminder of known common attack vectors applicable to applications. Next, we will focus on

methods that use these attack vectors to abuse a given weakness of a particular software project. Following these theoretical schemes, we have to take a closer look at specialized software utilities that use given methods to perform actual computer attacks. After finishing this chapter, we are going to be able to understand what attack vectors can security specialists use when analyzing an application, we will have a detailed idea of how methods attacking some kind of an application work, and we will know which software tools to use.

1.3.2 Analysis

At the beginning of this section, we will briefly introduce the application by taking a look at the process of acquiring & installing while also analyzing possible inputs. Next, we will focus on network communication performed by the application and on its installed executables & libraries, based on the results of the previous part. In the end, we will take an in-depth look at those pieces of the application that will seem to be most likely susceptible to any kind of an attack. After finishing this chapter, we will have a detailed knowledge of the internals of the application, understand its installation process, common usage scenarios and possible security deficiencies.

1.3.3 Assessment

Generally, we will want to assess all discovered problems (if any), document them and describe them in greater detail. Then comes the practical part of our thesis where we will try to assess whether it is possible to abuse each discovered problem and possibly create a proof of concept code. Here we will take a closer look at how to achieve this and elaborate on used methods, required skills and appropriate software.

1.3.4 Recommendations

In the last chapter of this thesis, we are going to examine measures leading to the mitigation of found weaknesses. Here we will specifically focus on each found problem, discuss possible ways to minimize the risk and give recommendations regarding next steps required for securing the application.

Research

In the second chapter of this thesis, we are going to focus on the knowledge required to perform a security review.

First, we will examine known attack vectors. Next, we will take a look at known methods and attacks that use these vectors and may result in possible security breaches. Finally, we will discuss specialized software used by security professionals.

2.1 Possible attack vectors

Attack vectors are in our context defined as opportunities for an attacker to cause harm to our system. A certain level of risk caused by some of these vectors is always present in any computer system and can not be avoided. In other cases, the level can be reduced to a reasonable minimum or the risk can be even eliminated completely.[2][9] *Please keep in mind that this should not be considered a comprehensive list of all possible attack vectors that currently exist in the computer world, as no such thing is possible due to several facts.*

2.1.1 Local attack vectors

Here we will focus on attack vectors that do not necessarily require network access. This means we will try to elaborate some of the vectors that may also work over the network, but will be applicable locally with physical access to the system/application as well.

2.1.1.1 Buffer overflow

- **Description** Buffer overflow (BO) is a common bug that gives an attacker a way to force a running process to either crash or to perform an unintended operation. Such unintended operation may be for example

running code provided by the attacker. Abusing a buffer overflow bug is usually heavily dependent on the architecture of the CPU, OS and the program design. As it will be a major part of the analysis in chapter 3, we will take a closer look at it now.[10][11][8]

- **Principle** Abuse of multiple computer design essentials, leading to unintended behaviour. It is a product of several design patterns combination that we will look at below.[10][11][8]

- **Execution flow** First, there is a hardware principle of the execution flow. CPU reads instructions from an address given by the value of the PC (Program Counter) register. This register is known as EIP/RIP on Intel-based processor architectures. If a program bug allows an attacker to set this value, the CPU will try to execute instructions from an address supplied by the attacker. And if this address is within a page/segment with appropriate access rights, attacker's code will be executed.[12][13]

- **Bounds checking** Next, programming languages like C or C++ are not, under certain conditions, performing bounds checking. When a programmer allocates 50 bytes of memory and then copies 1500 bytes of data into the allocated memory, the compiler may not detect such a situation.

Though avoiding bounds checking improves the speed of the programs, it gives an attacker a possibility to write bytes into the program's memory. When combined with the execution flow from the previous point, the contents of the buffer can be executed by the CPU as instructions.[14]

- **Stacks and heaps** Lastly, operating system data structures defining the layout of a running program in memory. A buffer overflow may be stack or heap-based. Each running program has typically one stack per thread and may have multiple heaps.[15][16][17]

- * **Stack overflow** A stack frame is a structure in memory created by the compiler. One of its purposes is to define the address of next instruction after a function returns. The stack overflow represents a situation, where an attacker overwrites a buffer allocated on the stack. Consider the following C and assembly code:

```
#include <stdio.h>

int main(void) {
    char buffer[8];
    gets(buffer);
    return 0;
}

0x00001f70 <+0>:      push    ebp
0x00001f71 <+1>:      mov     ebp, esp
```

```

0x00001f73 <+3>:      sub     esp,0x18
0x00001f76 <+6>:      lea    eax,[ebp-0xc]
0x00001f79 <+9>:      mov    DWORD PTR [ebp-0x4],0x0
0x00001f80 <+16>:     mov    DWORD PTR [esp],eax
0x00001f83 <+19>:     call   0x1f94 ; gets
0x00001f88 <+24>:     xor    ecx,ecx
0x00001f8a <+26>:     mov    DWORD PTR [ebp-0x10],eax
0x00001f8d <+29>:     mov    eax,ecx
0x00001f8f <+31>:     add    esp,0x18
0x00001f92 <+34>:     pop    ebp
0x00001f93 <+35>:     ret

```

Listing 2.1: Stack overflow vulnerable code

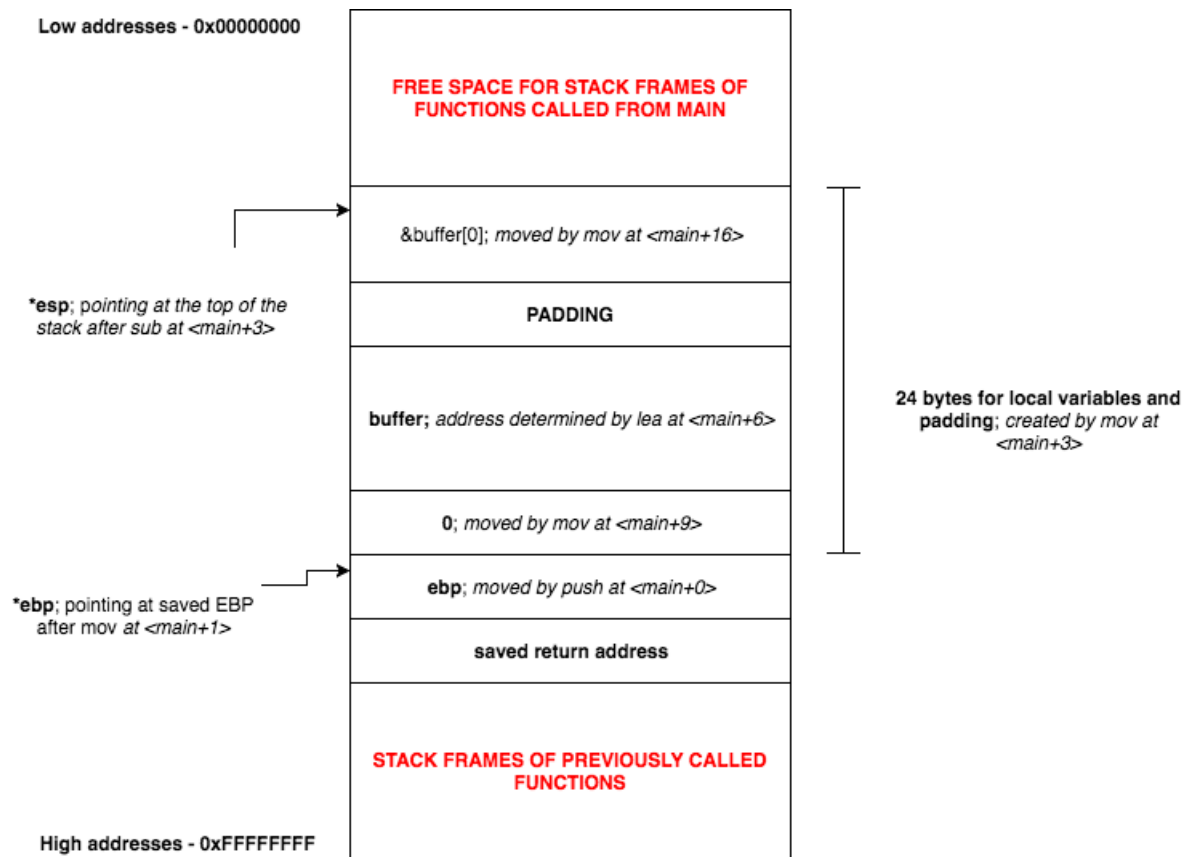


Figure 2.1: Stack frame structure in memory

Before the `call` instruction gets executed, the stack looks like in figure 2.1. Data written to the `buffer` now depend entirely on user input, and its length does not get checked. An attacker may insert enough bytes to overwrite saved return address and change the execution flow.[11][8]

- * **Heap overflow** Heaps are data structures created by the memory manager, and therefore, their structure is highly dependent on used OS and the memory manager implementation.

In general, attackers attempt to corrupt memory manager's internal structures by writing past end of the buffer in such a way that custom code gets inadvertently executed. Such an action is only possible if the memory manager stores at least some management data together with the allocated memory. Due to its complexity and specific requirements, heap overflows are way beyond the scope of this thesis. Also, after discussion with the supervisor, we will not be developing a proof of concept code if such a problem gets found.[11][18]

- **Examples** Buffer overflows got found in numerous applications. When functions that do not check the number of bytes written/read (for example `strcat()`, `strcpy()`, or `gets()`) process user input directly, a possibility of buffer overflow exists. Also, buffer overflows are typical for native-language code, such as C, C++ or assembly.[11][19]
- **Impacts** Injecting custom instructions into a running process and executing these instructions or crashing the process.
- **Protection** As buffer overflows are a real problem from the security point of view, there are numerous advanced techniques which try to minimize their impact. Address Space Layout Randomization (ASLR) or Data Execution Prevention (DEP) are mechanisms provided by the OS while stack canaries or SafeSEH get handled by the compiler. Note that SafeSEH is available for Windows applications only and DEP might have different names on a different platform. Of course, all these will not prevent the programmer from bad programming habits and usage of known vulnerable library functions.[7][10]

2.1.1.2 Security misconfiguration

- **Description** The operating system has many security features. Since security features do not mean secure features, security misconfiguration often allows access to or tampering with data. Therefore, these have to be configured appropriately otherwise they can be useless or even harmful.[2][9]
- **Principle** Insufficient administrator knowledge and/or human mistake.
- **Examples** Too low/strict access rights, usage of `sudo` with no password, root login over password protected `ssh`, unsecured Intelligent Platform Management Interface (IPMI), too many open files/connections for application, too high/low stack/heap size.

- **Impacts** Here we will list the impacts based on a specific security system implemented by the OS.²
 - **Access rights** Read/write access to unwanted files/devices, executing commands as their owner using SUID/SGID bits, replacing content in privileged directories
 - **Firewall** Open ports for privileged or private services, communication of unwanted software, information leakage
 - **Authentication** Privilege escalation, password cracking
 - **System limits** Exhausting system resources, application crash, password cracking
- **Protection** General protection against any misconfiguration is a never-ending process of learning. Therefore except for professional training, courses and self-studying we can recommend tools like *salt* or *puppet* to automatize configuration across multiple servers and minimize the possibility of human mistake. However, nothing will prevent an uneducated administrator from deploying bad but valid configuration.

2.1.1.3 Physical access

- **Description** Physical access to the computer itself usually means we can bypass almost every security feature in the operating system. We might not be able to access the data on the targeted system, but unless we are in an extremely secured environment, we will be able to use the computer in any way we want.[2][9]
- **Principle** Physical access allows the attacker to restart the computer, boot from another medium than the predefined device and start a different operating system. The boot process can be secured from BIOS/UEFI, but with the physical access, we are always able to reset BIOS/UEFI to its default settings which in most scenarios effectively removes these passwords.
- **Examples** N/A
- **Impacts** Possible access to all data stored on the system, installation of a possibly different operating system, rootkits and backdoors.²
- **Protection** Full drive encryption offers protection to the data stored on the system even when an attacker has physical access to the disk itself. Also, placing the system into an environment where no physical access is possible (e.g. data centres with certification) provide enough safety for most real-life scenarios.²[2][9]

²Examples and tips only, many others usually exist

2.1.2 Remote attack vectors

We understand remote attack vectors as opportunities coming from the network. This means we will not focus here on possibly vulnerable applications but rather on network design patterns that allow abusing by its very definition.

2.1.2.1 Information disclosure

- **Description** Revealing too much information is a common threat found in software applications. Sensitive information can be anything that will give the attacker an insight about what hardware are we using, what software are we running in which version, what are the physical locations of our equipment and so forth. The attacker can use this information to better aim at the target system, e.g. by looking for known vulnerabilities of identified components.[2][20][21]
- **Principle** Generally, it abuses the principle of misconfiguration and default unsecured settings. It uses for example default unsecured configurations left out by administrators, wrong settings, forgotten settings or exposed debug information.
- **Examples** Service banners of all kinds (SMTP, POP, SSH, etc.), HTTP header `Server`, SMTP commands `VERFY/EXPR`, DNS zone transfers, MS-RPC dumps, NetBIOS names and relations including empty relations, Google look-ups based on documentation/default web pages (`inurl: "/phpinfo.php"`, `inurl: "/moodle/login/index.php"`, etc.).[2][9]
- **Impacts** Impact can range from negligible to critical - what devices and OS are we running, which usernames are valid, what software in which version are we using, what is the logical map of our network, what are the domain names we are using and where do they resolve to, what IDS/IPS appliances are present.²
- **Protection** Detailed examination of current settings and defaults, changing the configuration to expose minimum information, updating the source code, ideally performing a 3rd party code review.²

2.1.2.2 Network sniffing

- **Description** Network sniffing can be separated into passive and Man-in-the-Middle (MitM) techniques. Passive attacks do not require active modification of network traffic, while MitM sniffing attacks consist of redirecting targeted traffic through attacker's computer and then examining passing traffic. Information transmitted over the network may contain technical information, passwords, personal information, credit card numbers and much more.[2][9][22]

- **Principle** All unencrypted information transmitted over the network is readable not only by the recipient of the information but also by everyone else with access to it. Access to this information can be, especially in local networks, easily achieved as we will see later in 2.2.3.
- **Examples** Rogue WiFi AP, ARP/DNS poisoning attacks, rogue DHCP server, ICMP/BGP traffic re-routing.²[23]
- **Impacts** Revealing user names and passwords for all commonly unencrypted network protocols (FTP, SMB, HTTP, SMTP, IMAP, POP3 and many others), reading other people's emails, acquiring private documents and conversations.²
- **Protection** Using encrypted versions of network protocols (if they exist), tunnelling specific network traffic, using VPN even at the local network.² Generally, most of network sniffing can be prevented using encrypted traffic as much as possible.

2.1.2.3 Bad network protocol design

- **Description** Network protocols define the way processes talk to each other and form the core of network communication overall. Their design is typically very mature and therefore does not meet current security requirements. To retain backward compatibility, however, these protocols are still used.[24][25]
- **Principle** Protocol design did not consider security as important enough or did not consider it as a factor at all. Such protocols often lack any kind of security mechanisms, whether it is by its very nature or just by mistake.[24][9]
- **Examples** Cache poisoning in ARP/DNS/ICMP protocols, TCP connection hijacking, ICMP live hosts identification, unsecured FTP/Telnet/HTTP/... [26]
- **Impacts** Highly dependent on a type of an attack - it can lead to information disclosure due to network sniffing, transmitted data being tampered with, network application crash, denial of network access and so on.²
- **Protection** Very difficult because the protocols are hard to update and these flaws are built into them. Recent network devices from enterprise-class manufacturers like Arista or Cisco try to detect some common attacks and prevent them. Also, one can minimize the usage of these protocols or completely disable them, although that will inevitably lead to loss of functionality. Another option is to replace some of them with

a most recent protocol with similar functionality - HTTP with HTTPS, or FTP with SFTP, for example.[26][9]

2.1.2.4 Denial of Service

- **Description** An Internet service that is not responding, is slow or has high latency is the same as a non-existent service. In case a service is the primary source of generated funds, it should be kept alive and working at all costs. Unfortunately, the possibility of exhausting resources of the system providing such a service can never be avoided entirely.
- **Principle** Exhausting selected resources on a targeted system. DoS is often represented by using all available system memory, opening a maximum number of file descriptors or crashing the application by a specially crafted input.[2][27]
- **Examples** TCP SYN flood, UDP flood, ping of death, reflection attacks, slowloris, fork/XML bomb.[2][27]
- **Impacts** The response time of attacked service is indefinite, leading to timeouts on the client side. The impact may range from crucial (for example financial or trust loss) to negligible, based on the type of the service.
- **Protection** Custom NIC drivers passing packets directly to user-space application, DDoS protection services like Cloudflare.²

2.1.2.5 Wireless networks

- **Description** Wireless networks remove the need to be physically connected to the adjacent network. This allows the attacker to for example sit in his car outside a company while sniffing their network traffic.
- **Principle** Current Wi-Fi attacks benefit from bad design principles in encryption schemes used. It is not the network protocol as a whole but just its encryption part. Two main encryption schemes exist today - WEP and WPA/WPA2 and they both have their security flaws. Also, another defect can be found in several WPS implementations which effectively breaks the level of security provided by WPA/WPA2.[23][28]
- **Examples** WEP IV re-initialization & ARP packet injection, WPA2 KRACK, WPS pixie-dust, WPA/WPA2 handshake bruteforcing.[29][30][23]
- **Impacts** Same as gaining unauthorized access to the physical network - network sniffing, data tampering, connection hijacking, service denials and much more.²

- **Protection** Separate guest SSIDs, separate VLAN and network range for wireless traffic, correct firewall rules, WPA2-CCMP only usage, client devices updates after the KRACK attack.²[23][30][28]

2.2 Known methods and attacks

Now when we know about common attack vectors, we can take a closer look at known methods and particular attacks that researchers use when examining applications.

First, we want to take a look at methods that are used for getting more information about a targeted system. This is going to include up-to-date and sophisticated network methods that help us find out what is required to continue with a review any further. Then we will focus on methods used for finding buffer overflows. At the end of this section, other advanced techniques mostly regarding network protocols will get examined.

2.2.1 Reconnaissance

Detailed information about the targeted system is the first step of every security analysis. These information help security professionals to for example search for known exploits or reverse engineer detected software.

- **Publicly available information sources** Publicly available information sources include many different databases - specific web services collecting leaked passwords, Google indexes or WHOIS data containing administrative info for almost every online entity.[21][2]
- **Network map** Creating a network map usually involves identifying live hosts, mapping DNS names to IP addresses and determining used network paths leading to those IP addresses. Identifying live hosts is mostly done by sending ICMP ECHO REQUEST messages to targeted IP address and waiting for an answer. The `ping` utility present in modern operating systems does this. Mapping DNS names to IP addresses is a complicated task due to several reasons. At first, there is no way to obtain all sub-domains of a domain from a public point of view. And second, DNS mappings change in time and become invalid based on the TTL of every DNS record. A technique called DNS zone transfer might be in certain situations used to obtain them. Determining network paths is done by sending out a packet with IP TTL equal to 1 and waiting for the first ICMP TIME EXCEEDED message. Then the TTL gets increased, and this procedure gets repeated. A tool called `traceroute` handles this process.[2][9]

- **Software detection** Software detection distinguishes operating systems and applications running on analyzed machines, preferably with their versions. OS fingerprinting is a method that relies on several patterns produced differently by every OS. For example default IP TTL, default TCP window size, filled fields in DHCP requests, TCP options set for a connection and others. Detecting running network applications is done by port scanning - sending an opening packet to a predefined set of ports on an analyzed host and waiting for a response. Version detection usually occurs in application layer protocol by examining HTTP `Server` header, SSH login banner, or SMTP server EHLO response, for example.[31]

2.2.2 Buffer overflow

Next, we will take a look at methods and procedures that help find buffer overflows.

2.2.2.1 Reverse engineering

Phrases and definitions in this subsection are inspired by lectures from [32]. Software reverse engineering (RE) is a process of analyzing an application to create a representation of the system at a higher level of abstraction. This, in our case, means we will have a compiled executable binary and we will want to know how exactly does it work, what algorithms and data structures does it contain and where it might have possible weaknesses.[33]

- **General info** RE tries to recover the source code of an application that is available only as an executable binary. It is impossible to obtain the source code fully as information gets lost during compilation and linking. Also, RE has ethical and legal aspects - we are permitted to use RE to understand how the software works. We should not use it to bypass copy protections or create copies of some particular software.[33]
- **Types of RE** We can make a difference between *dead code analysis* and *live code analysis*. *Dead code analysis* is performed against a non-running executable typically by a tool called disassembler. *Live code analysis* is on the other hand performed by a tool called debugger on a running application.

In *dead code analysis*, the application has no idea that it is being analyzed and therefore can not defend itself in any way. However, when analyzing a complex binary (e.g. a packed one), it might be complicated to imagine what the application does without actually running it. *Live code analysis*, on the other hand, executes application's code so we

can clearly see what is happening and when. But as an opposite, the application has now the chance to detect it is being debugged and react appropriately.

- **Required knowledge** To RE an application a security researcher must possess a vast amount of knowledge. One of it is detailed knowledge of the target platform ABI. This includes data alignment in the memory, how are parameters passed to functions, how are CPU registers used, symbol names mangling and others. Also, the researcher must be familiar with the API of system calls and also with the API of standard libraries for given OS. This information allows separating parts of the binary that are actually kernel/library code. Advanced knowledge of the assembly language and basic programming paradigms in it is so essential we do not think it has to be explained any deeper.
- **Commonly examined parts of a binary** Here we want to take a look at the parts of a binary executable containing information helpful for determining what algorithms and data structures in the binary do. This does not represent examining the algorithms themselves but control structures created by the language runtime to execute the application correctly. We will also focus on Windows binaries only, as that is the case of our analyzed application.
 - **PE header** This is the header that every executable for MS Windows has. It contains the first level of elementary information that a security research wants to take a look at. Except for information required for running the code various hints and flags can be seen here - if the application supports ASLR/DEP/SafeSEH, what DLLs does it import from, when was it compiled, by which vendor is it supplied, its version and many others. Also, application's resources can be found here.[34]
 - **IAT** Import Address Table is directly connected with the PE header as it is part of it. It contains names and ordinary numbers of functions from imported DLLs which are used by the loader when executing the application. Based on the content of this table the loader fills in the addresses of required functions. These addresses can be in several scenarios replaced in foreign processes, actively changing imported functions.[34][35]
 - **RTTI** This information gets typically used for operators like `typeid` or `dynamic_cast` of the C++ language. In RE we use private structures like `RTTITypeDescriptor`, `RTTIBaseClassDescriptor` or the main one `RTTIClassHierarchyDescriptor` to find out used class names and class hierarchy.[36]

- **Debugging and disassembling** These are the two main techniques used in RE. Each of them uses specialized software (disassemblers and debuggers) to achieve its goals. Here we will take a look at basic principles and in section 2.3 we will focus on particular tools.

- **Disassembling** Translation of particular binary code into code in a human-readable assembly language of the targeted CPU. Two traditional approaches on how to achieve this exist - *linear sweep* and *recursive traversal*.

Linear sweep is fast and straightforward as it performs disassembly in a linear fashion byte by byte from the start of the `.text` section. This approach gets easily confused by mixing data with code and ambiguous code flows (i.e. obfuscation). *Recursive traversal*, on the other hand, starts at the entry point of the program and disassembles instruction after instruction following every possible jump/call, therefore examining all the parts of the reachable code. Unreachable parts of the binary are considered to be data, and thus this technique does not get confused when data gets mixed with code. Both these approaches are far more sophisticated in real usage and nowadays tools used in RE combine them to provide the best results.[37][38]

- **Debugging** It is a process of finding bugs in an application while it is running. In RE, this is typically used to watch the execution flow, to bypass obfuscation and generally to understand internal algorithms and API calls.

Debugger works by either creating a process we want to debug or by attaching to an already running one. The debugger then obtains all debugging events for the process and reacts appropriately to them. Debugging events represent situations like starting a new thread, loading a library or handling an exception. Besides these events the debugger has full control over the targeted process - it can change its data in registers, alter execution code flow or instructions directly, dump contents of memory and many more. Most importantly, the debugger can set breakpoints (software and hardware ones) and trace executed instructions. These features allow it to stop the execution flow on a predefined place and also to record the progress of a running application.[10][39]

- **Application protection** Many commercial applications are reverse engineered to bypass the application's license. Also, many applications are reverse engineered to find undiscovered vulnerabilities. For this reason

the CPU architecture, OS and programmers use various techniques that help them to protect applications. Generally, we can split these techniques based on their purpose.

- **Security protection** These are mechanisms that are supposed to protect the application from attacks like buffer overflows. Of course, no protection mechanism will prevent the programmer from using insecure programming functions and paradigms. Here we will only state a few examples, and since these will get examined as buffer overflow protections, we will take a closer look at them in 5.2.

ASLR - a function offered by the OS which must be supported by the application causes the binary file to be always loaded at random address in the memory, thus making exploits based on buffer overflow much harder to develop. Next is DEP, realized by NX/XD bit in CPU. This bit causes marked memory pages to be not executable. Others are canaries, unique values at the end of the stack frame that detect stack-based buffer overflows and SafeSEH, a mechanism which prevents custom Structured Exception Handlers (SEH) to be installed by an attacker.[10][40]

- **Know-how protection** Know-how protection, on the other hand, secures the algorithms inside the application. Such a protection gets typically used in commercial software and games as their developers want to prevent the existence of illegal copies. Also, many malware authors use these methods to hide what their software is actually doing from an anti-virus. Most commonly used techniques are obfuscation and encryption.
- **Obfuscation** Obfuscation generally stands for making a code of a program less understandable. Its primary purpose is to fatigue analysts/attackers by obscuring program's code. Though obfuscation does not increase the security of the program, it is often used as a supplemental protection mechanism. It is a must for applications running on byte-code oriented platforms such as Dalvik VM, JVM or .NET. Otherwise, their code could be quickly recovered by means of RE. This particularly means, for example, renaming variables and functions, opaque predicates, or string encryption. Code parallelization, splitting into multiple processes, table interpretation, dead and irrelevant code insertion or removal of library calls are techniques which make the life of a security professional genuinely challenging.[41]

2.2.2.2 Fuzzing

Fuzzing is a process of finding bugs in software by supplying pseudo-random data as application input. The main purpose of fuzzing is usually crashing the application being fuzzed. A correctly written application shall always withstand this process. If a crash or any other exceptional condition occurs, then there is a bug. We are going to focus on black-box fuzzing, where the fuzzer does not know the internal structure of the program being fuzzed. Another type is white-box fuzzing, where the fuzzer is able to check whether all code paths got hit. Such fuzzing requires access to the source code of the product. Grey-box fuzzing uses AI techniques and is currently considered state-of-the-art in vulnerability detection.[42][43][44]

- **General info** Testing programs by random inputs is an old technique used even with punched cards.[45] However, fuzzing these days differs a lot, and state-of-the-art fuzzing software uses advanced AI techniques to cover as much code as possible. It has several advantages - it is able to test a given application without any prior knowledge about its structure/inputs, once configured it can run for months without human interaction, and it usually reveals bugs that were missed by other sorts of testing techniques. On the other hand, fuzzing from its very nature carries a few drawbacks - indefinitely long runtime (based on scheduled tests), configuring for a complex application input can be very time consuming and code covered by fuzzing differs a lot based on the quality of fuzzer and its settings.[43][46]
- **Types of fuzzers** Fuzzers can be categorized as *dumb* and *smart*. *Dumb* fuzzers do not know anything about the input structure and thus supply random data to the application. This has an advantage of a very easy configuration and setup but requires a significant amount of time to pass integrity or file format checks typically found in programs. On the opposite, *smart* fuzzers are aware of the file/protocol format and can keep certain values fixed or calculated correctly to provide valid inputs. The latter approach provides greater code coverage but requires much more time to prepare and configure.[43][42]

Another way to look at fuzzer categorization is based on the way used to generate new inputs. Three different classes can be distinguished: *mutation*, *generation* and *evolutionary*. *Mutation-based* fuzzers take an already existing input and mutate it - shift bits, swap bytes, repeat parts of data and so forth. *Generation-based* fuzzers have a specification of the data they are generating and create new data from scratch - this requires more intelligence from the fuzzer but provides a more robust way of testing. Advanced fuzzers combining these two methods exist as

well. *Evolutionary-based* fuzzers are state-of-the-art pieces of software in fuzzing as of now. These use techniques like genetic programming to improve submitted inputs over the time and cover as much code as possible.[43][46][44][47]

- **Fuzzing process** The fuzzing process greatly differs based on the used fuzzer, environment, debugging capabilities and tested application. For dumb fuzzers on Linux platform, testing programs with simple and easy to use interface can be as simple as piping the output of `/dev/urandom` into the program while checking its behaviour with tools like `gdb` or `valgrind`. For smart, mutation and generation-based fuzzers with complex configuration, this process complicates slightly.[43][48]

To keep the information level at a reasonable amount, we can say that the process of fuzzing consists of several parts. First is specifying the format of the input we want to get fuzzed - this can be a file format, network protocol structure or anything else that targeted application accepts. After being done with the format, we have to define an interface for passing generated inputs to the tested application. Typically this can be done using a network socket or command line arguments. Next, we must establish a connection between debugger/memory-checker-tool and our application to detect anomalies. To achieve this, the fuzzer usually starts a debugger/memory-checker-tool and instructs it to run our application within itself. Finally, the fuzzer repeats generating inputs and supplying them to the tested application until a predefined condition is reached (a bug is found, a timeout is reached and so on).[43][48]

2.2.2.3 Static analysis

Static code analysis is a method used for discovering bugs by examining the source code of the application. It is a process similar to the compilation itself but with focus on buffer overflows, separate branches handling and many other security-related issues. Current compilers (for example `clang`) contain such a static analyzer and provide additional warnings during the compilation. We are going to focus on C/C++ static analysis, although it is principally not limited to these programming languages.[49][50]

- **General info** “Static” represents the fact that the analysis is non-runtime and should be a part of the build process. It finds all syntactic and lexical issues, and it also tries to understand the semantics of the program. In its most simple form, it could be done by a very strictly configured compiler. A more advanced version of a static analyzer searches for known dangerous functions (for example `gets()`, `sprintf()`, or `strcpy()`[11]), static arrays allocated at the stack, easily abused system

calls and much more.[51] However, current professional tools for static analysis go far beyond these abilities and try to simulate the execution using many possible scenarios - taking particular jumps, supplying certain inputs, looping for a specified number of times and so on. This has the target of finding particular execution flow and its matching input that in combination can lead to some anomaly (memory overflow, use-after-free, ...). The output from the analysis points to lines in a code where a given anomaly might happen.[52]

Although the static analysis is a very powerful technique when looking for bugs, it has several drawbacks. Most notably it requires the source code of the product. Next is a large number of false positives - static analysis usually gives a significant amount of output which does not really present a security vulnerability. This leads to missing real threats once they actually appear. Also, the analysis can be very time-consuming which makes it hard to integrate into the build process.[53]

- **Examples** Here we want to present few examples of dangerous statements found usually in C/C++ programs.
 - **Non-bounds-checking functions** `strcpy()`, `sprintf()`, ...
 - **System calls executing programs** `execve()`, `system()`, ...

2.2.3 Network protocol cache poison

As an example of methods that attack the principles and design patterns of network protocols, we will take a look at cache poisoning. This technique allows us to fake data in protocol caches thus forcing the protocol logic to communicate with rogue machines instead of the correct ones. This can have various effects based on the used network protocol - for example traffic sniffing, denial of service or traffic redirection.

- **DNS** An attacker tries to insert malformed records into the cache of selected recursive caching DNS server. Such a server is usually operated by the ISP as they want to provide its users with fast DNS answers. The attacker first issues a DNS query for a particular domain forcing the recursive DNS server to forward such a query to configured authoritative DNS server. Before the authoritative server manages to respond the attacker sends a forged answer back to the caching DNS server. This way the DNS recursive server stores the response from the attacker and ignores the real one from the authoritative server. Such an attack can lead to a situation where all future queries for a whole domain get forwarded to attackers computer who therefore gains full control over what gets returned in answers to those queries.[54][55]

- **ARP** Where DNS cache poison can work regardless of network topology and attacker's location, ARP cache poison only works on local networks. Its usage is on the other hand much more straightforward, and it attacks victims computer directly. ARP gets typically used for translation between MAC and IP addresses with the protocol being a part of every common OS. The OS keeps a cache of MAC - IP mappings. Records can get easily inserted into this cache from any computer on the local network. After inserting custom records, the attacker can force the victim to send all its traffic to attacker's computer by supplying his MAC address.[26][56]

2.3 Specialized software

In this last section, we will try to give some practical examples of software tools used by professionals when doing security reviews. We will categorize these utilities based on their purpose and provide a short comment with a link to visit for more details. We will try to minimize used space and keep this section as a straightforward list. Also, these descriptions will be directly citing the official documentation of specified software.

- **Information gathering** `dnsenum`, `dnswalk`, `fierce`, `fping`, `nmap`, `maltego`, `dmitry`, `p0f`, `recon-ng`, `ettercap`, `dsniff`, `nfspy`, `sslstrip`
 - **dnsenum** Perl script to enumerate DNS information of a domain. <https://github.com/fwaeytens/dnsenum>
 - **dnswalk** DNS debugger performing zone transfers and database checks. <https://github.com/davebarr/dnswalk>
 - **fierce** DNS reconnaissance tool for locating non-contiguous IP space. <https://github.com/mschwager/fierce>
 - **fping** Program to send ICMP echo probes to network hosts, similar to ping, but much better performing when pinging multiple hosts. <https://fping.org/>
 - **nmap** Utility for network discovery and auditing. <https://nmap.org/>
 - **maltego** Tool for analyzing real-world relationships between information that is publically accessible on the Internet. <https://www.paterva.com/>
 - **dmitry** Program with the ability to gather as much information as possible about a host. <https://github.com/jaygreig86/dmitry/>
 - **p0f** Tool that utilizes an array of sophisticated, purely passive traffic fingerprinting mechanisms to identify the players behind any

- incidental TCP/IP communications (often as little as a single normal SYN) without interfering in any way.
<http://lcamtuf.coredump.cx/p0f3/>
- **recon-ng** Full-featured Web Reconnaissance framework.
<https://bitbucket.org/LaNMaStEr53/recon-ng>
 - **ettercap** Comprehensive suite for man in the middle attacks.
<http://ettercap.github.io/ettercap/>
 - **dsniff** Collection of tools for network auditing and penetration testing. <https://www.monkey.org/~dugsong/dsniff/>
 - **nfspsy** Library for automating the falsification of NFS credentials when mounting an NFS share.
<https://github.com/bonsaiviking/NfSpy>
 - **sslstrip** Tool to provide a demonstration of the HTTPS stripping attacks. <https://moxie.org/software/sslstrip/>
- **Vulnerability detection** golismero, sparta, lynis, burpsuite, zap, sqlmap, bbqsql
 - **golismero** Easily expandable framework for web security testing.
<https://github.com/golismero/golismero>
 - **sparta** Application which simplifies network infrastructure penetration testing by aiding the penetration tester in the scanning and enumeration phase. <https://github.com/SECFORCE/sparta>
 - **lynis** Security auditing tool for UNIX derivatives like Linux, macOS, BSD, and others. <https://github.com/CISOfy/lynis>
 - **burpsuite** Web vulnerability scanner.
<https://portswigger.net/burp>
 - **zap** Automatically finds security vulnerabilities in web applications. <http://www.zaproxy.org/>
 - **sqlmap** Penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers. <http://sqlmap.org/>
 - **bbqsql** Blind SQL injection framework, semi-automatic and fast.
<https://github.com/Neohapsis/bbqsql/>
 - **Exploitation and attacks** pyrit, john the ripper, hydra, aircrack-ng, reaver, metasploit, beef
 - **pyrit** Pyrit allows to create massive databases, pre-computing part of the IEEE 802.11 WPA/WPA2-PSK authentication phase in a space-time-tradeoff. <https://code.google.com/archive/p/pyrit/>

- **john the ripper** Password cracker.
<http://www.openwall.com/john/>
- **hydra** Multi-protocol, parallel, network password cracker.
<https://github.com/vanhauser-thc/thc-hydra>
- **aircrack-ng** Complete suite of tools to assess WiFi network security. <https://www.aircrack-ng.org/>
- **reaver** Reaver implements a brute force attack against Wifi Protected Setup (WPS) registrar PINs in order to recover WPA/WPA2 passphrases. <https://code.google.com/archive/p/reaver-wps/>
- **metasploit** Penetration testing framework.
<https://www.metasploit.com/>
- **beef** Penetration testing tool that focuses on the web browser.
<http://beefproject.com/>
- **Reversing and fuzzing** IDA, ollydbg, nasm, apktool, peach, afl, choronzon, vuzzer
 - **IDA** IDA is a multi-processor disassembler and debugger.
<https://www.hex-rays.com/products/ida/>
 - **ollydbg** Assembler level analysing debugger for Windows.
<http://www.ollydbg.de/>
 - **nasm** Assembler designed for portability and modularity.
<http://www.nasm.us/>
 - **apktool** Tool for reverse engineering Android APK files.
<https://ibotpeaches.github.io/Apktool/>
 - **peach** Automated, scalable and seamless fuzzer.
<https://www.peach.tech/>
 - **afl** Security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. <http://lcamtuf.coredump.cx/afl/>
 - **choronzon** Evolutionary knowledge-based fuzzer.
<https://github.com/CENSUS/choronzon>
 - **vuzzer** Application-aware evolutionary fuzzer.
<https://github.com/vusec/vuzzer>
- **OS tools and others** windbg, gdb, wireshark, CFF explorer, sysinternals, hxd
 - **windbg** Kernel/User mode code debugger for Windows.
<https://developer.microsoft.com/en-us/windows/hardware/download-windbg>

2. RESEARCH

- **gdb** GNU Project debugger.
<https://www.gnu.org/software/gdb/>
- **wireshark** Widely-used network protocol analyzer.
<https://www.wireshark.org/>
- **CFF explorer** PE header inspection.
<http://ntcore.com/exsuite.php>
- **sysinternals** Utilities to help manage, troubleshoot and diagnose Windows systems and applications.
<https://docs.microsoft.com/en-us/sysinternals/>
- **hxd** Hex Editor and Disk Editor.
<https://mh-nexus.de/en/hxd/>

Analysis

This chapter brings us closer to our application, *Mary*. This is where the more practical part of our thesis begins as we will analyze the application here.

First, we will take a look at the overall structure of our application - how to acquire it, installation procedure, application inputs and others. Next, we will examine binaries and libraries and try to determine their purpose, developer and security features. Finally, based on the facts found in the first two sections we will take a detailed look at the theoretically problematic parts.

3.1 Application structure

The first section will introduce us to the application structure with a focus on any theoretical security deficiencies. Let us first take a look at the possible ways of obtaining the application and its installation procedure. We will be interested especially in the distribution source for our application, created files or registry keys and the default access rights of these objects. Also, network communication will get in our interest as well as local inputs. In this section, we will try to briefly introduce all possible places where a security issue might be hidden, and in the upcoming sections, we will dive deeper into these parts.

3.1.1 Acquire and install

The only way to obtain *Mary* is to download it. *Mary* exists in numerous language versions and has been mirrored to many popular software-download servers, but we will silently ignore all of these. The only download source we will focus on is the official web page of *Mary*'s vendor which provides versions localized in English. Also, we will focus on the most up-to-date version, 5.40, as it is of now.

3. ANALYSIS

The installation will be done under an account that was created during the installation of our OS. This account is by default in the *Administrators* group, and it is also by default protected by the UAC. Also, we will leave all installation options to their defaults and use the most user-friendly NEXT-NEXT-NEXT way to install *Mary*.

3.1.1.1 Download

Two official websites from which *Mary* can be downloaded exist, each of them leading to a different location, but the same file. As it was confirmed by the vendor, one of the pages serves as a developer version, and the other one is for business purposes. They use Apache as their web server and support only older HTTP/1.1. Both these pages do not use the secured version of the HTTP protocol and therefore transfer all the data unencrypted. Also, both these pages do not offer checksums of available files. The downloaded document is a self-extracting binary file with `exe` extension, digitally signed by the vendor.

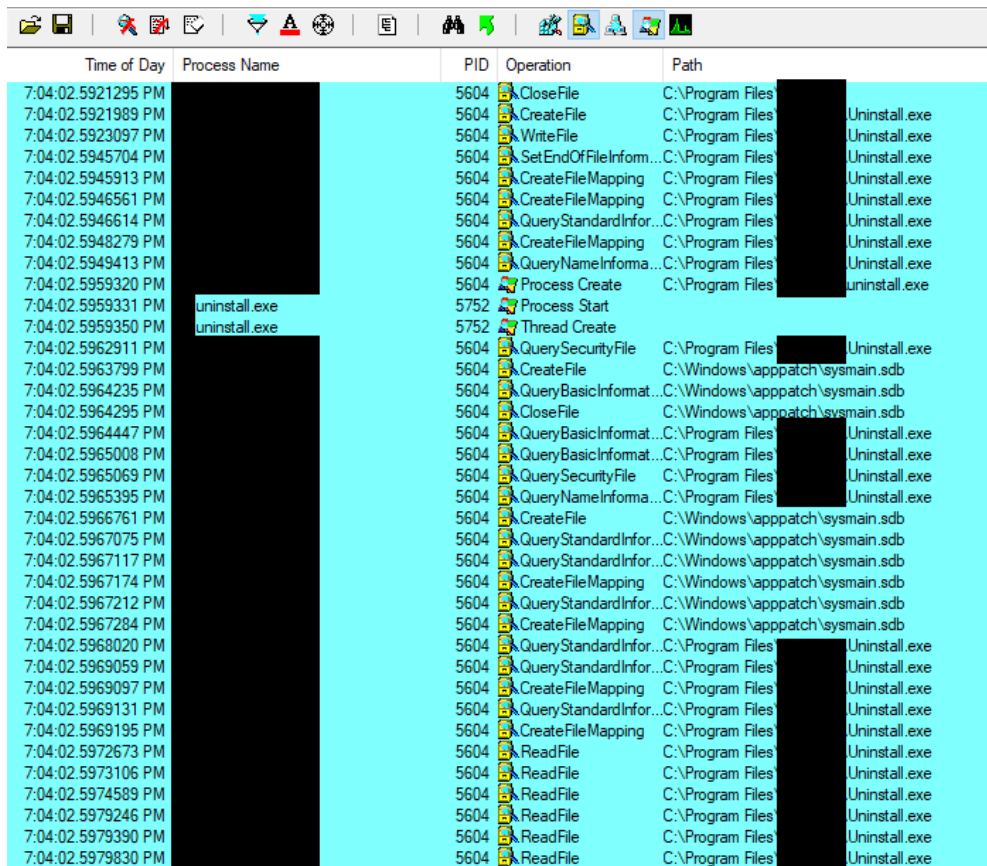
As *Mary* has more than 500 million users worldwide, there is a significant probability that in a vast number of local area networks happens the download happens within a reasonable period. Using the method of ARP poisoning mentioned in 2.2.3, an attacker might be able to redirect these download requests to a computer under his/her control. Also, data exchanged using the HTTP protocol without encryption can be easily manipulated using a MitM attack at the TCP level.[57][58] This is definitively an interesting opportunity requiring more in-depth research. We will focus on it in a greater detail in subsection 3.3.1.

3.1.1.2 Install

The installation starts with UAC privilege elevation where the dialogue shows that the application is digitally signed by the vendor. Only one window is shown during the installation, offering nothing but the path where to install *Mary*. We keep everything at its defaults which leads to the creation of files under the `C:/Program Files/Mary` directory. The installer uses the `ReadFile` call to read itself from the disk and then the `WriteFile` call to create files in given path. Based on the offset it has read from itself, it knows which file to create. Apart from creating most of the necessary files it also reads and writes numerous registry keys. In its very end, it uses the `ProcessCreate` call to launch `uninstall.exe` which is a file it has previously created. This process finishes the installation - most importantly, it sets appropriate file types to be opened by *Mary*, icon paths, `shellex` registry keys and so on.

This whole process has been monitored by three useful tools - `procmon`, `regshot`

3.1. Application structure



Time of Day	Process Name	PID	Operation	Path
7:04:02.5921295 PM		5604	CloseFile	C:\Program Files\...
7:04:02.5921989 PM		5604	CreateFile	C:\Program Files\...Uninstall.exe
7:04:02.5923097 PM		5604	WriteFile	C:\Program Files\...Uninstall.exe
7:04:02.5945704 PM		5604	SetEndOfFileInform...	C:\Program Files\...Uninstall.exe
7:04:02.5945913 PM		5604	CreateFileMapping	C:\Program Files\...Uninstall.exe
7:04:02.5946561 PM		5604	CreateFileMapping	C:\Program Files\...Uninstall.exe
7:04:02.5946614 PM		5604	QueryStandardInform...	C:\Program Files\...Uninstall.exe
7:04:02.5948279 PM		5604	CreateFileMapping	C:\Program Files\...Uninstall.exe
7:04:02.5949413 PM		5604	QueryNameInforma...	C:\Program Files\...Uninstall.exe
7:04:02.5959320 PM		5604	Process Create	C:\Program Files\...uninstall.exe
7:04:02.5959331 PM	uninstall.exe	5752	Process Start	
7:04:02.5959350 PM	uninstall.exe	5752	Thread Create	
7:04:02.5962911 PM		5604	QuerySecurityFile	C:\Program Files\...Uninstall.exe
7:04:02.5963799 PM		5604	CreateFile	C:\Windows\appatch\sysmain.sdb
7:04:02.5964235 PM		5604	QueryBasicInformat...	C:\Windows\appatch\sysmain.sdb
7:04:02.5964295 PM		5604	CloseFile	C:\Windows\appatch\sysmain.sdb
7:04:02.5964447 PM		5604	QueryBasicInformat...	C:\Program Files\...Uninstall.exe
7:04:02.5965008 PM		5604	QueryBasicInformat...	C:\Program Files\...Uninstall.exe
7:04:02.5965069 PM		5604	QuerySecurityFile	C:\Program Files\...Uninstall.exe
7:04:02.5965395 PM		5604	QueryNameInforma...	C:\Program Files\...Uninstall.exe
7:04:02.5966761 PM		5604	CreateFile	C:\Windows\appatch\sysmain.sdb
7:04:02.5967075 PM		5604	QueryStandardInform...	C:\Windows\appatch\sysmain.sdb
7:04:02.5967117 PM		5604	QueryStandardInform...	C:\Windows\appatch\sysmain.sdb
7:04:02.5967174 PM		5604	CreateFileMapping	C:\Windows\appatch\sysmain.sdb
7:04:02.5967212 PM		5604	QueryStandardInform...	C:\Windows\appatch\sysmain.sdb
7:04:02.5967284 PM		5604	CreateFileMapping	C:\Windows\appatch\sysmain.sdb
7:04:02.5968020 PM		5604	QueryStandardInform...	C:\Program Files\...Uninstall.exe
7:04:02.5969059 PM		5604	QueryStandardInform...	C:\Program Files\...Uninstall.exe
7:04:02.5969097 PM		5604	CreateFileMapping	C:\Program Files\...Uninstall.exe
7:04:02.5969131 PM		5604	QueryStandardInform...	C:\Program Files\...Uninstall.exe
7:04:02.5969195 PM		5604	CreateFileMapping	C:\Program Files\...Uninstall.exe
7:04:02.5972673 PM		5604	ReadFile	C:\Program Files\...Uninstall.exe
7:04:02.5973106 PM		5604	ReadFile	C:\Program Files\...Uninstall.exe
7:04:02.5974589 PM		5604	ReadFile	C:\Program Files\...Uninstall.exe
7:04:02.5979246 PM		5604	ReadFile	C:\Program Files\...Uninstall.exe
7:04:02.5979390 PM		5604	ReadFile	C:\Program Files\...Uninstall.exe
7:04:02.5979830 PM		5604	ReadFile	C:\Program Files\...Uninstall.exe

Figure 3.1: *procmon* captured creation of *uninstall.exe*

and *wireshark*. Thanks to that we can be sure that, for example, no network communication happens during the install as no packets related to the installation were captured by *wireshark*. This is also confirmed by the output from *procmon*, which recorded exactly zero network related system calls. In addition to using *wireshark* we have obtained a full list of registry keys created and changed by the installation - the tool *regshot* helped us with that by doing a snapshot of the keys before and immediately after the installation. Information acquired using *procmon* assures we know what system calls did the installer use. Therefore we have a good big-picture idea about what happened.

Overall the installer behaves properly, does not offer any third party bloatware, nor does it connect to any servers to either send some information or to download unnecessary plugins and so on. The installation process does not, therefore, seem like a valid place to investigate.

3. ANALYSIS

No.	Time	Source	Destination	Proto	Length	Info
280	27.3...	fe80::ffff:ffff:ffff	ff02::2	ICMPv6	103	Router Solicitation
281	27.3...	fe80::8000:f227:a10a:8602	fe80::ffff:ffff:ffff	ICMPv6	151	Router Advertisement

Figure 3.2: Network communication after filtering out *Microsoft* servers

3.1.1.3 Access rights

We were checking the access rights of two object types - registry keys and files. Both these object types got created by the installation. Using `regshot`, we acquired a list of all of the newly created registry keys. The list of files was acquired using `procmon` by following file-system related calls. Finally, using `accesschk`, we acquired the access rights.

Registry keys are generally well protected. The *Users* group always has just read rights, the user installing the application can only rewrite several less interesting keys (e.g. file type associations). More critical keys (e.g. `shelllex`) are writable only by the *Administrators* group and *SYSTEM* user account. The same goes for files - no typical mistakes like files writable by the *Everyone* account appear here, and all important files (e.g. `*.exe`, `*.dll`) are also writable only by the *Administrators* and *SYSTEM*.

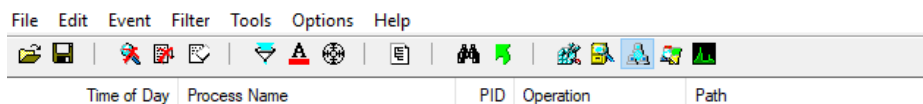
Overall the application works with minimal possible access rights, does not leave any unprotected files in the system and tries to follow best security principles.[7] This forces us to do the decision of excluding access rights as a possible attack vector and focus on other parts of the application.

3.1.2 Network communication

We have already checked network communication during the installation process - there is none. Also, *Mary* is not a network application. Therefore it does not open a listening port nor does it accept incoming connections. Other most common examples of network communication include updates, banners with commercials, database updates, online help and so on. Unfortunately, none of these happen in *Mary* during typical usage.

After trying every single button that is provided by the GUI and even after supplying many different types of input files not a single packet is transmitted, nor a single network-related system function is called. This has been extensively tested using both `wireshark` and `procmon`. The only two scenarios under which something related to network happens are the following:

- **Unlicensed usage** *Mary* has a trial version for 40 days, after which the user must uninstall it or buy a full version. However, it continues to function with a commercial pop-up window coming up every fifth launch. This pop-up connects to several domains in a secured manner (using HTTPS with certificate validation) and also to several other domains using HTTP only. The data transmitted using the HTTP protocol can contain JavaScript which gets executed by *Mary*'s JS engine. Although this may sound interesting, we will not focus on this network communication. It has the same prerequisites as the unsecured download of the installer (ARP/DNS poison, connection hijack and so forth) while providing fewer possibilities when compared to supplying a customized binary (running custom JS code inside of *Mary*'s sandbox vs running completely custom code with theoretical privilege escalation). Mitigation and recommendations will therefore be the same as stated in 5.1. Nevertheless, the vendor has of course been informed about this situation along with a few recommendations. Also, it should be stressed out that such weakness is only present when using the product in contradiction with its license terms.
- **Home page redirect** This is even less interesting than the previous option. When *Mary* is asked to open its homepage, it passes the link to a default browser as an argument. All the network communication itself is then done by the browser, which therefore means, none of the code from *Mary* actually works with it.

Figure 3.3: Network communication captured using *procmon*

3.1.3 Other program inputs

Based on the results from previous subsections, we now know we have to focus on some other than network inputs. We choose to focus on files, as they are the main type of accepted data and therefore their processing should cover a large

part of the code base. *Mary* can open 15 different file types when executing algorithm A (ALG-A) and any file type when running algorithm B (ALG-B). From the analysis point-of-view, we will focus on ALG-A. Although accepting all file types as an input might create more attack opportunities than in case of ALG-A, it is in practice more complicated to force a victim to execute ALG-B than ALG-A. As some libraries share names with compatible file types, it is highly likely that code processing these file types will be in these DLLs. Some of them are directly from authors of *Mary*, and some are third-party libraries, developed and compiled elsewhere. Mixing code of multiple authors always increases space for a mistake.

Because of the fact that no interesting network inputs exist, we will focus on file inputs in the rest of the review. These inputs are processed by various libraries, and therefore we will take a closer look at them in the next section.

3.2 Executables and libraries

All *Mary*'s executables and libraries will be introduced in this section. We will first focus on a brief overview of them and therefore create a list with a short description for each. Here, we are also going to examine PE headers and respective security flags of these binary files. Based on these headers, we will then take a closer look at two interesting libraries which happen to lack all basic security features and therefore provide theoretical space for issues. To maintain application's anonymity, all its files will be renamed in the same way the application is.

3.2.1 Overview

This is a list of all libraries bundled with *Mary*. Each of these libraries provides algorithms to work with one particular file type that *Mary* can handle. The original names of these libraries are well known enough to reveal the file types they work with. However, this was also experimentally proven merely by opening a given file type and making sure *Mary* loads the appropriate library using the `LoadLibrary` call.

- **elizabeth.dll** Opens and works with `.eli` files. This library will get thoroughly examined in 3.2.2 and 3.3.2.
- **mary.dll** and **mary64.dll** Both handle the native format, `.mar`, with `mary64.dll` being the 64-bit version of the library.
- **jennifer.dll** Opens and works with `.jen` files. This library will get thoroughly examined in 3.2.3 and 3.3.3.

Other binary files installed follow. This includes binary files that are actually used when running *Mary* as well as files used during the install/uninstall.

- **algA.exe** Run ALG-A from the command line
- **algB.exe** Run ALG-B from the command line
- **uninstall.exe** Finish installation or start uninstallation
- **mary.exe** Run GUI application offering the possibility to execute both ALG-A and ALG-B as well as other minor features. Also, handle the rest of possible file types.

Therefore we have four binaries and four dynamic libraries. Given the information we have introduced in 2.1.1.1 and 2.2.2, the fact that no network communication exists and the fact that *Mary* is quite a broad application consisting of libraries developed by different vendors, we have decided to focus on buffer overflows. We will be interested only in stack-based buffer overflows.

Security properties of an application when considering buffer overflow attacks are mostly given by the used programming techniques and compilation options. Using compilation options appropriately presents a powerful way to protect the application. Therefore we will not consider such binary files as problematic. The determination of the compilation options before the used programming techniques narrows down the amount of code to be analyzed by some other means. Security attributes will be examined using the **PESecurity** - for their detailed description, please see sections 5.2 and 5.1.

File name	ASLR	DEP	Authenticode	SafeSEH
elizabeth.dll	False	False	False	False
algA.exe	True	True	True	True
mary.dll	True	True	True	True
mary64.dll	True	True	True	True
jennifer.dll	False	False	False	False
uninstall.exe	True	True	True	True
algB.exe	True	True	True	True
mary.exe	True	True	True	True

Table 3.1: Security attributes of binaries

The output of **PESecurity** tells us one important thing. Two libraries have ASLR & DEP & Authenticode & SafeSEH turned off - this puts the whole application in danger.[59] As long as we have at least one library with ASLR turned off, then this library is always going to be loaded at the same address (unless relocated because the address is already occupied). Moreover, as long as we know the address of one library, we can use the ROP technique with

instructions from this library to take control of the execution flow.[10][11] On the other side, when *elizabeth.dll* nor *jennifer.dll* is loaded, the application uses all compilation security features it can. This is a sign that the authors of *Mary* are aware of security threats and these problematic libraries might be coming from a third party. Given the results of the PE headers analysis, we will now focus on the two libraries in greater detail.

3.2.2 *elizabeth.dll*

At first, we will take a look at *elizabeth.dll*. We will use *CFFexplorer* to get as much public information about the library as we can. *elizabeth.dll* is a part of a large, open-source software project with a known author, it is developed for all standard operating systems, and in our case, it is used to provide the functionality of ALG-A for one particular file type, *.eli*. We can, therefore, find its source code available online and we can also find an already compiled version of *elizabeth.dll* bundled with the source code. This allows us to check hashes of the two versions and make sure that these files are the same. Now, we can see that this library is being developed & compiled by a third party.

From the source code documentation, it is apparent why there is no ASLR switched on - it is being compiled in an ancient MSVC6, where simply no ASLR option exists. Also, this MSVC6 uses quite an old format of its project files. Therefore we are forced to convert it to present type of MSVC solution. Thanks to such a conversion, we can now study the source code of this library with the possibility of compiling, debugging and everything else like a regular developer. This offers much more information than reverse engineering the compiled version of the library.

Mary uses a library compiled with no security features, publicly available source code and an easy way of loading. These facts make it reasonable to select *elizabeth.dll* as a candidate for further investigation. We are able to put the code through static analysis followed by smart fuzzing based on the file format specification. This will be examined in 3.3.2 and next chapter.

3.2.3 *jennifer.dll*

Here, we follow the procedure already used with the previous library. We first determine the author of the library and use this information to find out more details. This is a closed source software project of a German company which does not exist anymore. Moreover, we reveal that *.jen* is an old file format which has been used before *Mary* developed its own and better format, *.mar*. Same as with *elizabeth.dll* this library is also used to provide the functionality of ALG-A only.

ASLR is missing here because of the simple fact that the library was compiled in 2005 which was the first year when ASLR was introduced on Linux, two years before it was available on Microsoft systems. The “age” of the library also suggests that nobody fixed any security deficiencies that were discovered in the library in the last decade.

As we do not have the source code here, we will not be able to do the static analysis, but we can, on the other hand, try to find existing exploits, or reverse engineer the library to find the out file format structure and run smart fuzzing with this file format. This is again definitively an opportunity which deserves more time. Same as with the previous library, we will take a look at this in subsection 3.3.3 and next chapter.

3.3 In-depth inspection of selected parts

In the previous section, we have chosen three parts of the application that deserve more time and detailed analysis - HTTP installer download, *elizabeth.dll* and *jennifer.dll*. Here, we will focus on these parts.

First, we are going to take a look at the download by making a short summary of requirements for our fake web and tampered binary. Next, *elizabeth.dll* is about to be put through a static analysis which is going to be thoroughly examined, leading to partial conclusions. Finally, we are going to reverse engineer *jennifer.dll* to find out more information about it, and we are also going to try to use the existing exploits against the library.

3.3.1 Unsecured download

Now, we know that there are two domains which can be used for *Mary*'s download, both using HTTP/1.1 without encryption. We choose one of these domains and focus solely on it, but the vulnerability is so general, it is not a problem to use it for the second domain as well. There are many ways to abuse unencrypted network communication - downloading the full original web page, creating a customized binary replacing the original installer and redirecting requests to our server instead of the vendor's one is the chosen method for us.

We have decided to use **HTTrack** to confirm that we can download the whole web locally, store it and that the downloaded version looks just the same as the original. Next, **nginx** is the selected web server which gets configured to serve these files under the same host-names as the vendor's web server. This way we have a locally available web looking precisely the same as the vendor's web with the possibility of supplying our files whenever we decide to do so.

Now we have to create the file to be supplied and redirect network traffic to our `nginx` web server. Such an application should behave like the original installer while providing functionality for anything an attacker wants. Unless privilege escalation is required for the additional functionality, the user should not be notified by the UAC that anything unexpected is happening and the digital signature of the original installer should be correctly verified and valid. Through this method with the combination of non-existent control checksums, the user will have no easy way to determine that the installer has been manipulated with.

The description of the application itself along with the chosen way of traffic redirection will be the primary topic of section 4.2.1.

3.3.2 `elizabeth.dll`

Static analysis has been done using multiple tools - `rats`, `flawfinder` and `Coverity`. Their output is not part of the thesis to maintain application's anonymity, but it is, of course available, to eligible entities per request.

The situation is straightforward in case of `rats` and `flawfinder`. These applications accept source code files directly as an input, process it line by line, do not compile the program and work by matching well-known expressions. This approach has the advantage of a quick setup and easy usage but also brings some drawbacks - it might be unable to recognize specific types, logically unreachable parts of the code or parse pre-processor directives, therefore leading to a lot of false positives. `Coverity`, on the other hand, works in a more sophisticated way with a more complicated setup. We found out that it is costly for commercial usage and after having a short talk with the company representatives, it has been cleared out that for a trial, educational and all other purposes they provide a one-hour web session with a dedicated specialist. This has not been accepted as an optimal solution because much more time needs to be spent on a proper static analysis. Fortunately, `Coverity` works free as an online service for open source projects. Doing some minor changes to `elizabeth.dll` along with uploading it to `github` under proper license, was therefore sufficient to get it connected with the online version of `Coverity`. This way we have finally managed to get the static analysis done also using a more advanced tool used by real professionals.

`rats` and `flawfinder` have pointed out many potentially dangerous lines in the code. Most noticeably the usage of functions such as `MultiByteToWideChar` or `memcpy`, usage of stack-allocated fixed size buffers or possible lines where an exception might be thrown. None of these places has been evaluated as actually vulnerable, mostly because of following reasons.

- Correct bounds checking
- Memory allocation just for the size that has to be written
- Working with memory allocated on heap
- Static limits for parts of code working with static sized buffers

Final Results

- ./src/██████████.c:86: [3] (misc) *InitializeCriticalSection: Exceptions can be thrown in low-memory situations. Use InitializeCriticalSectionEx.*
- ./src/██████████.c:163: [2] (buffer) *memcpy: Does not check for buffer overflows when copying to destination (CWE-120).*
- ./src/██████████.c:347: [2] (buffer) *memcpy: Does not check for buffer overflows when copying to destination (CWE-120).*
- ./src/██████████.c:870: [2] (buffer) *memcpy: Does not check for buffer overflows when copying to destination (CWE-120).*
- ./src/██████████.c:967: [2] (buffer) *memcpy: Does not check for buffer overflows when copying to destination (CWE-120).*

Figure 3.4: *flawfinder* partial example output

Coverity provides a more complex algorithm for threat detection and therefore eliminates most of these false positives. Therefore, it was able to find only one defect from the *high-impact* category - out-of-bounds read. However, this defect is currently also understood as not interesting for the following reasons.

- **Hardware limits** It might, even in theory, only work at systems where access to addresses in the form of `[base-address + offset]` with `offset` being from the range 0 - 4294967295 does not cause an exception. On 32-bit systems, this will cause an integer overflow followed by access to address 0 which will be followed by an exception. Such an exception will be handled by Mary and a warning dialogue window will be shown to the user.
- **Unpredictable values** Values at addresses mentioned above would have to be dereferenceable as valid addresses without causing an exception as well. Without the possibility to affect these values in advance.
- **Uncertain writes** The possibility to write custom values into the memory is still uncertain even if we somehow manage to bypass previously mentioned conditions.

```

8. decr: Decrementing numPs . The value of numPs is now 4294967295.
◆ CID 1133090 (#1 of 1): Out-of-bounds read (OVERRUN)
9. overrun-local: Overrunning array ps of 64 4-byte elements at element index 4294967295 (byte offset 17179869180) using index --numPs (which evaluates to 4294967295).

```

Figure 3.5: Out-of-bound read found by *Coverity*

Although we know that the application is not protected by ASLR, DEP and SafeSEH while *elizabeht.dll* is loaded, we were not able to find a stack-based

buffer overflow using the static analysis of the library's source code. Nevertheless, in sub-section 4.2.2, we will take a look at the file format `.eli` and try to create simple examples of fuzzing templates.

3.3.3 `jennifer.dll`

As we do not have the source code available here, we will have to do RE. The chosen software to do so is `IDAPro` in its free version. `IDAPro` is a state-of-the-art disassembler and in its full paid version even a decompiler. This will allow us to take a look at the internals of the library as well as the file format.

First, we want to take a look at the moment of loading the library - `IDAPro` allows us to set breakpoints into the code of the DLL library while executing the main application. This way we can stop the execution flow at each exported function to find parts of `Mary` that take care of calling `LoadLibrary`. We could also do this from the other side - search all strings for "`jennifer.dll`" and use hardware breakpoints to stop execution whenever the application reads from the string's address. Both ways lead to a state after which the library is loaded. Parts of the code running before the load and after freeing the library are not interesting for us. Here we are able to find out that `Mary` tries to simulate ASLR for `jennifer.dll` by relocation. Before calling any function from the DLL, it loads and unloads the library a few times, and it also tries to load various system-related libraries. At least one of these libraries is often placed in a collision with the address `0x400000` where `jennifer.dll` has its `ImageBase`.

By continuing with RE we will find out, that the library enables its client applications to set a few callbacks. This presumption is later confirmed by finding unofficial header files specifying the API for working with the library. We have four callbacks in total - `Info`, `Error`, `Request` and `State`, while each of them is used in a completely different situation, has different input structure and different return values. These callbacks are able to alter the behaviour of the library in a significant way, as they can force the library to stop, continue where it last left or quit immediately. Also, each client application may have an entirely custom algorithm in these callbacks, and it should be noted that they are called a lot more often than expected - for example for each character in a particular string. Next, before any file is processed by the library two initialization and testing functions are called - `JenInitDll` and `JenTest`. Then there is one function which processes the whole input file, `JenProcessFile`.³ All these functions have, as parameters, pointers to custom structures that define many properties of the library - for example how deep to look for file type signature, temporary files directory and so on.

³*Names of functions changed appropriately to avoid application disclosure.*

3.3. In-depth inspection of selected parts

During the analysis of the library, we have tried to use the only existing available exploit found online. This exploit leads to the creation of a specially crafted `.jen` file which when processed by the library, causes a stack-based buffer overflow. The library implements its own `strcpy` function which copies bytes from the source address to the destination address until it reaches a null byte. This implementation is therefore equally dangerous as the standard one from `libc` because it does not check the number of bytes already written. The BO found by the authors of the exploit is rather an educational example - a locally allocated fixed size buffer which gets filled by using the aforementioned `strcpy` function and the input string is directly supplied by the user in the input file. There is not a single check for the size of the data and the size of available memory. As `jennifer.dll` is not being maintained for many years and the company that developed it does not exist anymore, the success of the exploit was expected. However, the authors of `Mary` found a way to protect it from the given buffer overflow. By implementing a checking logic in one of the callbacks, they are able to count the number of bytes that should be copied to the given fixed size local buffer, before passing this user input to `JenProcessFile`, where the vulnerability gets exploited. This way they are able to detect such malicious inputs and raise an exception which gets caught by `Mary` itself. `Mary` then reacts appropriately as if the input file is invalid - simply by showing a warning dialogue window informing the user that it was unable to process the input.

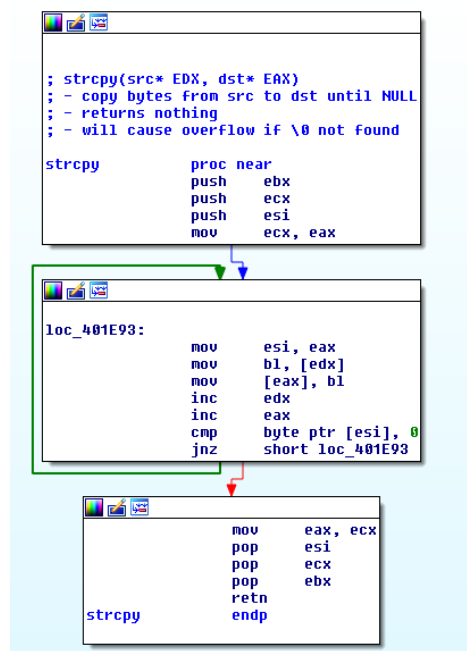


Figure 3.6: `strcpy` implementation by `jennifer.dll`

3. ANALYSIS

During reverse engineering of the library, we have confirmed that *jennifer.dll* is truly not under further development nor has it been fixed for many years and that the bugs in it stay there. However, we can also say that the authors of *Mary* are well aware of the security vulnerabilities that were found in the library and they try to mitigate their effects by any possible means - either by simulating ASLR or by checking against known exploits. Also, during our RE procedure, we have gained knowledge about the internal structure of the `.jen` file format. This structure was later confirmed by finding an unofficial documentation for the format. For this reason, we have decided to create a fuzzing template based on the file created by the exploit. This way we can try to search for similar mistakes automatically - this process will be described in 4.2.2.

Assessment

This is the last chapter of this thesis where we analyze the given application. Here we will try to assess all of the defects we have found. This chapter represents the end of the practical part of the thesis.

First, we want to briefly summarize found problems. This is intended for impatient readers that skipped the previous sections and jumped straight here. After that, we will utilize the knowledge gained in previous sections and assess discovered problems.

4.1 Defects

The first weakness found in the application is the unsecured download of the installer and lack of control checksums. Under certain conditions, this leads to the possibility of replacing the installer with a customized binary. Next is the absence of ASLR, DEP and SafeSEH for certain libraries. These security features mitigate the effects of buffer overflow attacks, which may lead to the change of the execution flow, instructions injection and complete change of actions in the affected application.

4.1.1 Unsecured download

An unsecured download is caused mostly by the way that some network protocols are designed. Because of the possibility of injecting routing information into routing tables of routers we do not own using the BGP protocol, or due to the possibility of poisoning ARP caches of other devices in local networks, we are able to manipulate and alter the path that each packet travels.[26][60] This is not always possible because of several reasons, but the design of such protocols fundamentally allows this. Next, because of the possibility of packing an existing application into another one, unpacking it later and executing the unpacked application, we can create a fake installer, which in addition to

actually installing the original application, does whatever its creator wants it to do.

We will try to examine the found problem in a local area network situation using ARP cache poisoning and a customized binary file which will include the original installer. We need access to the victim's local network - this access can be, for example in the case of local wireless networks, achieved by one of the methods mentioned in the subsection 2.1.2.5. In our example, we will suppose we already have this access, whether it is a wireless or a wired network. Next, we want to discover the IP address of the default gateway and inform all other devices in the network that the MAC address associated with this IP address is our own MAC address. This way we will be able to redirect all outbound traffic to our computer. Such a redirect will allow us to process the traffic, manipulate it and also re-route it. On the computer receiving all the traffic, we will start a configured `nginx` web server, which will serve the previously downloaded vendor's web copy, but with our customized binary as the installer. Finally, we will just change the destination IP address for packets heading to the vendor's web to our IP address and handle all incoming download requests.

4.1.2 Buffer Overflow

We are searching for a stack-based buffer overflow. This BO might be present when *elizabeth.dll* or *jennifer.dll* is loaded as these libraries are compiled with old and nowadays obsolete options, and therefore lack essential security features. In all other cases, the application is protected by the combination of ASLR, DEP and SafeSEH which represent a solution we will not try to bypass.

These libraries are loaded into the memory in case of working with `.eli` or `.jen` files. When processing `.jen` files, ASLR is simulated using high-likely-relocation, but this does not protect the application entirely. Therefore in the case of an application used by 500 million people, it still makes sense to look for an existing BO. These libraries get freed when the application is done with the input file.

A sophisticated and advanced analysis of these libraries has been done, which in the case of *jennifer.dll* led to reverse engineering and testing of existing exploits. In the case of *elizabeth.dll*, this led to an in-depth static analysis, source level debugging and the analysis of the source code itself. In subsection 4.2.2, we are focusing on fuzzing these libraries. This means choosing an appropriate fuzzer, creating its configuration and running it.

4.2 Practical assessment

We will test the unsecured download by redirecting the installer download request from a local network to our web server. This web server will be returning a customized binary containing the original installer as a resource. Buffer overflow will be tested by running a configured fuzzer trying to find an input leading the application to a crash. In case we detect such a crash, we will try to abuse it to change the execution flow.

4.2.1 Unsecured download

This will take place in a simulated network using *VirtualBox*. We will emulate three machines connected together - vendor's server (SRV), victim user (USR) and an attacker (ATK). USR will be downloading the installer binary from SRV, while ATK will be poisoning its ARP cache. In a real world we do not, of course, have SRV in our local network, but attacking default gateway instead of SRV would have the same effect. SRV and USR will be running *Ubuntu 16.04* while ATK will be running *Kali 2017.2* - a Linux distribution designed specifically for security researchers. Omitting DNS or default gateways in the network configuration below has no effect on the final functionality.

- **SRV** 192.168.1.1/24, no default gateway, no DNS server
- **USR** 192.168.1.2/24, no default gateway, no DNS server
- **ATK** 192.168.1.3/24, no default gateway, no DNS server

4.2.1.1 Traffic interception

Under normal circumstances (i.e. no ongoing MitM attack) the installer file `mary540.exe` will be downloaded from SRV.

```
victim@victim-VirtualBox:~$ curl http://192.168.1.1/mary540.exe | md5sum
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
   Dload  Upload  Total   Spent    Left    Speed
100 2033k  100 2033k    0     0  43.8M    0  --:--:--  --:--:--  --:--:--  45.1M
aecddf902c027e52fd1696a56de739c9  -
```

Figure 4.1: Downloading installer from SRV under normal circumstances

Ettercap was chosen for the ARP poisoning attack. *Ettercap* is a multipurpose tool allowing us to perform various MitM attacks, hijack TCP connections, sniff passwords and much more. First, we let *Ettercap* scan the network which will give us a list of active devices. Then SRV gets selected as target 1 (T1) and USR as target 2 (T2) - in case we want to poison the whole network and not just a particular device, we leave T2 unassigned. This instructs *Ettercap* to poison all other devices except for T1. Now it is enough to start a MitM

4. ASSESSMENT

ARP poisoning attack and let *Ettercap* send appropriate packets. In the *Connections* tab, we can now see all the connections between *USR* and *SRV*.

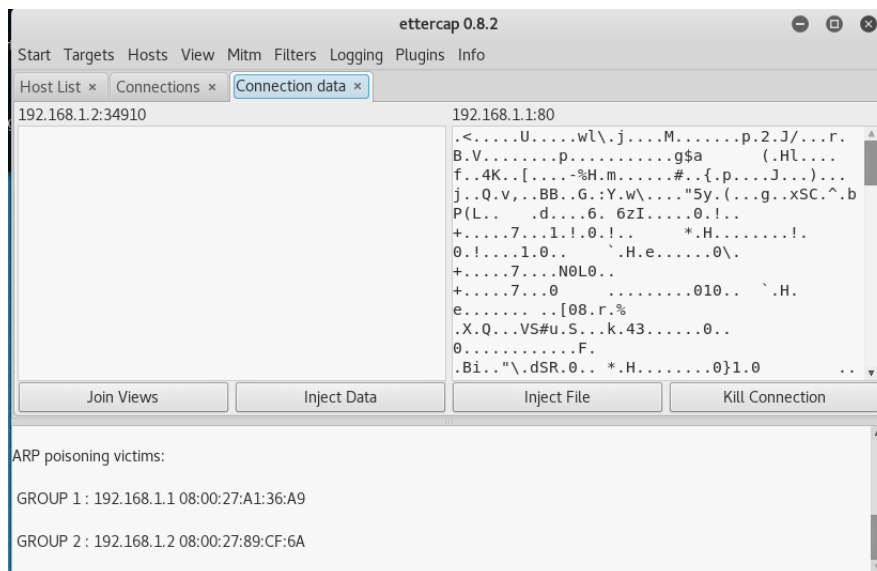


Figure 4.2: Intercepted connection

Next, we use two `iptables` commands taking care of web traffic redirection at the *ATK* station. The first command changes the destination IP addresses before the routing decision is made by the kernel so that HTTP packets from *USR* can reach local `nginx` at *ATK*. The latter command changes the source IP of outbound packets after the routing decision has been made so that *USR* still thinks it is communicating with *SRV*.

- `iptables -t nat -A PREROUTING --src 192.168.1.2 --dport 80 -p tcp --dst 192.168.1.1 -j DNAT --to-destination 192.168.1.3`
- `iptables -t nat -A POSTROUTING --src 192.168.1.3 --sport 80 -p tcp --dst 192.168.1.2 -j SNAT --to 192.168.1.1`

Finally, we start `nginx` in its default configuration and place the local copy of vendor's web into `/var/www/html`. Here we replace the original installer with our crafted binary. If required by the situation we could use the `more-set-headers` `nginx` module to manipulate outgoing HTTP headers to exactly match headers from *SRV*. Also, in case the vendor's web would be updated very frequently, and it would be complicated to keep our local version up-to-date, we could use the `nginx-lua-module` to patch `nginx` to serve only the installer from the local file and pass everything else to the vendor's web, effectively acting as a reverse proxy. In our case, none of the above is required. Now we

can download the file at USR again and receive a completely different file from ATK instead of SRV.

```
victim@victim-VirtualBox:~$ curl http://192.168.1.1/mary540.exe | md5sum
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left     Speed
100 2340k  100 2340k    0     0  55.4M      0  --:--:--  --:--:--  --:--:--  55.7M
97fa0128174750efa6be473c1eb0ca61 -
```

Figure 4.3: Tampered binary downloaded after poisoning ARP cache

```
root@kali:~# curl localhost/mary540.exe | md5sum
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left     Speed
100 2340k  100 2340k    0     0  2340k      0  0:00:01  --:--:--  0:00:01  78.8M
97fa0128174750efa6be473c1eb0ca61 -
```

Figure 4.4: Tampered binary stored locally

4.2.1.2 Binary tampering

The fake installer will carry the original installer as a resource which will get unpacked into a temporary file and executed.[61] The original installer is digitally signed, so direct tampering with this binary is not an option. Before the execution of the original installer, the attacker has space to do whatever needs to be done. If desired actions require Administrator privileges, the user might be notified about it based on the UAC settings. The fake installer will not have a digital signature, and therefore such a UAC notification will warn the user that the software comes from an unknown developer. This might cause some suspicion in the user and lead to cancellation of the installation. In case that the privileges of the user running the installer will be sufficient, we can handle this situation without the user being notified about anything strange. A UAC pop-up will show after our custom code is finished, just because of the original installer itself, thus showing the original vendor as the software developer. In our example, we are not executing any custom code as we do not really want to harm the user, but just point out possible options.

First, we make the resource available using the combination of `FindResource`, `LoadResource` and `LockResource` calls. Next, we ask the system for an available name of a temporary file and write the contents of the resource into it. We also rename this file to have an `.exe` extension to make it executable for other system functions. Finally, we call `ShellExecute` to launch the newly created file. Here, we use the `runas` option, which even though that it is undocumented, enables us to elevate privileges of a binary we are unable to change. This simple demonstration shows that unsecured download of the

installer may easily lead to the execution of attacker's code, without the user noticing anything.

4.2.2 Buffer overflow

We have created numerous fuzzing templates for both libraries, and our fuzzing tests are still running as of today. Fuzzing is a very time-consuming technique, as a vast number of generated inputs is evaluated as invalid by the application and therefore the tested part of the code is not very large. For this reason, we have decided to use smart-fuzzing. Here the fuzzer has some hints about the internal structure of the generated data, or even knows it entirely, which is the optimal case. This way the fuzzer can generate much more valid inputs and cover much larger part of the code base.

As a fuzzing software, we have decided to use *Peach* fuzzer. *Peach* is old, mature, proven, offers many configuration options, is both mutation and generation based and if our thesis should continue in a commercial environment *Peach* offers a paid pro version which includes many templates for common input types, advanced support and even more configuration options. Sample fuzzing templates for *Peach* were created during our thesis. These templates should serve as an example of working with the fuzzer, rather than a comprehensive and thorough description of examined file types.

4.2.2.1 Peach

Peach is configured using the so-called PIT files.[62] These are XML files which describe the structure of generated data, way of interaction with the tested application, methods of memory access checking and failure detection, log mechanism and other properties that can be configured for a fuzzing task. In our case, the generated data are going to be written to a file and *Mary* is going to be launched from a command line. Here, it will be given such parameters to process the given file using ALG-A without requiring any further interaction with the user. Crashes are detected using the `windbg` debugger and *Peach* takes care of replicating the crash along with logging everything important.

Most complicated part of the configuration is, therefore, the Data Model which describes the structure of the generated data. *Peach* is missing quite a few parameters in its free version however despite this fact it still offers an impressive range of options to try. Here follows a list of some of them to get an idea of what should a sophisticated fuzzer offer.

- **Data types** Numbers, strings, flags, blobs, ...
- **Relations** Marking e.g. specific number as a size-of/count-of/offset-of/... some other part of generated data

- **Fix-ups** Calculation of CRC-32/SHA1/MD5/... of some other part of generated data
- **Transformations** Compression/Decompression of data block using bz2/gzip, encryption/decryption of data block using AES/3DES/....

Unfortunately, the configuration is not very straightforward due to several facts. Whether it is missing/misleading/unpublished documentation, huge differences between versions 2.x and 3.x (*Peach* has been completely rewritten from Python to C#, has different parameters, options, etc.) or lack of proper community and support from the authors, it complicates any advanced usage. It should be however noted, that all these issues are resolved when using the paid version of *Peach* which logically gets much more attention from the authors. Therefore despite these facts, *Peach* is still the most appropriate tool to use as it offers much more configuration possibilities than other fuzzing software.

After creating the PIT files, executing the fuzzing task is a question of one simple command. For speeding up the process, we can use tools like *ImDisk* to create a RAM disk in the memory and write created files into it without waiting for often slow hard drives.

4.2.2.2 Fuzzing *elizabeth.dll*

First, we will take a look at fuzzing *elizabeth.dll*. As *.eli* is a highly optimized format, it turned out to have custom integer encoding. This is by definition a significant issue for any fuzzing software which is not aware of the encoding and does not support it. *Peach* can generate specific numbers using the relations, fix-ups or transformers as stated above, but it has absolutely no clue that integers written to a generated file must be encoded in a specific way. Therefore, the library is going to decode different numbers than *Peach* intended to write and the file might end up invalid. Unfortunately, this does not currently have a solution within our reach. After contacting the authors of *Peach*, it has been confirmed that no easy way how to achieve custom number encoding exists in the free version. An option for partially bypassing the encoding was the *eval* transformer applying given *Python* expression on selected numbers, allowing the user to modify the data on-the-fly. However, this transformer has either been marked as obsolete or has some new undocumented usage, but in any case, it currently does not work. *Peach* just does not start when trying to include this transformer in the PIT file. As so, to minimize the effect of the encoding algorithm, only 8-bit numbers are used in places where encoded numbers should be. This way the probability of generating correctly encoded number is the highest. It should be noted that the paid version of *Peach* has a variable-width-number type which makes things a little bit easier, and also that the authors are able to create a customized

version of *Peach* including this encoding. Thus, in a commercial environment, this issue can be resolved by using the paid version and purchasing a patch.

Next, besides being highly optimized, the file format is also very complicated. Also, the documentation does not entirely match the generated and valid files, therefore creating the templates is not very straightforward. Besides, several created templates crashed *Peach* itself (reported to the authors with no answer), so a wrapper script detecting the crash and starting *Peach* again had to be created.

Fuzzing with created templates is still running and was able to find multiple input files that crash a bundled client application. However, no input crashing *Mary* has been found yet. Also, none of the inputs that crash the client application can lead to the change of the execution flow. During the analysis of *elizabeth.dll*, we were able to find one issue, theoretical buffer overflow that can be caused thanks to the lack of ASLR, DEP and SafeSEH. However, no such buffer overflow has been found yet, and no proof of concept code has been developed. As more than a year has been spent on this analysis, it might seem that *elizabeth.dll* is safe to use. However, since it is a large, hard to maintain software project it is probably only a matter of time until future code edits lead to such a buffer overflow and it gets discovered - recommendations in 5.2 minimize the chances that it will be worth even looking for.

4.2.2.3 Fuzzing jennifer.dll

Here the situation is better, opposed to the previous case. Apart from a few tiny details, we can describe the internal structure of an example *.jen* file by our PIT file. The internal structure is not so complicated and no special integer encoding is required. Our PIT file is based on the previously mentioned file generated by the exploit - this exploit is only prevented thanks to the internal logic in *Mary*, while *jennifer.dll* still remains vulnerable to it. Its fuzzing might therefore create a file which will not get detected by *Mary*.

One problem we were facing here is the fact that *.jen* uses only the lowest 16-bit of the otherwise 32-bit CRC-32 value. As *Peach* is only able to generate 32-bit long CRC-32, we had to fix this using an *expression fix-up* at the price of some additional data generated into the file. This could have been also fixed by a particular transformer designed for shortening numbers, but although it is documented, it does not work. This has been also reported without any answer.

Same as with the previous library the fuzzing is still running as of today. And same as with the library earlier, it has not been able to find a valid buffer overflow leading to an application crash yet. Although we know that

the library is vulnerable, it is protected by the additional checks in its client application, which make it secure for now. But here, the risk of existing buffer overflow is even higher than in case of the former library - as *jennifer.dll* is not maintained anymore, it is possible that the callback logic in *Mary* will not be able to prevent the next BO found (if any). Thus, we will take a more detailed look at protection methods and recommendations in 5.2.

Mitigation & Recommendations

In this last chapter, we will focus on recommendations and advice for *Mary's* vendor who has been informed about the results of the thesis. We are going to examine possible protections and suggest the best one for a given situation. These protections should influence the users of *Mary* as less as possible, and they should also include minimal expenses connected with them.

5.1 Unsecured network communication

In the life cycle of our reviewed application are two situations which require some network communication. The first one is the download of the installer and the second one is displaying a commercial when using the product without a license. Both of these situations use some kind of unencrypted communication. Unencrypted communication leads to the possibility of hijacking the connections and inserting data delivered to the JS engine of our application, or even worse, delivering a tampered installer of our application. As mentioned before we have focused on a forged installer as it provides the attacker with more theoretical options and possibilities.

5.1.1 Protection methods

- **Checksums** The vendor might provide users with control checksums of downloaded files. Thanks to checksums, users would be able to check that the data they have downloaded is the file that was originally published by the vendor. Currently, they do not have such a possibility, even though this brings no additional costs for the vendor. This protection method only applies to the fake installer defect, as it requires the user to manually check that the published checksum matches the checksum of the downloaded file.
- **HTTPS** All the unencrypted network communication could be easily

secured by using HTTPS instead of plain HTTP. HTTPS, when correctly implemented, also ensures that no traffic hijacking is possible, thus eliminates all the attacks like ARP poisoning, BGP re-routing and so on. On the other hand, it might create additional fees connected with obtaining a trusted certificate, but thanks to services like *Let's Encrypt* these fees can be reduced to a very reasonable amount. Also, these expenses can be minimized using several CDN providers who provide SSL free of charge. Both of these should be negligible for a company producing software as famous as *Mary* is.

5.1.2 Recommendations

No network communication can be made 100% secure by its very definition. However, currently, it is believed that today's encryption standards should be strong enough for a regular user to keep his secrets private. Checksums are a nice-to-have feature but should be provided as an additional check and always in combination with HTTPS. Otherwise, the attacker might simply change the checksum value delivered to the user and fake it appropriately. Also, these checks require the user to participate in the process which is always bad by its design. For these reasons, we would recommend a full switch to HTTPS and its usage for all required communication.

Several emails have been exchanged with the vendor regarding this topic. During the time of working on this thesis, the vendor has already switched its public web to HTTPS. Every request now delivered to the standard HTTP port TCP 80 is replied with a 302 HTTP redirect to its secured version. Although this might seem like a solution, it only means that the attacker has to use a tool like `sslstrip` in addition to `ettercap`. The redirect maintains backward compatibility, but it is definitely not enough to stop a sophisticated attacker. Disabling HTTP solely would help this situation well. Also, the vendor is aware of the unencrypted communication happening in the commercial pop-up, but currently has private reasons to stick to their current technology. They are, however, working with several third parties on an update which will allow a full switch to HTTPS for this communication as well.

5.2 Buffer overflow

Although we have not found a valid buffer overflow vulnerability, the application is, in its current form, still providing space for any BO to appear due to future changes. This is caused by the lack of ASLR, DEP and SafeSEH for two libraries.

5.2.1 Protection methods

- **ASLR** ASLR makes buffer overflows much more complicated due to the fact, that it loads libraries and executables at different addresses everytime they are loaded/started. This ensures that supplying our exploit with a valid return address cannot be done as easily as without it. Getting the return address for an ASLR disabled application is usually (unless for example, relocation occurs) as simple as executing the application at a target platform, printing the addresses around the buffer with shellcode and using one of them. To minimize segmentation fault errors and illegal instructions, attackers usually use a simple NOP-sled technique which leads the CPU directly to the prepared instructions. With ASLR, this is not possible, as the address at which the application is loaded changes every time the application is started. Thus determining the address at the attacker's computer makes no sense. Bypassing ASLR usually requires some other vulnerability to be present in the application. This vulnerability (e.g. formatting string vulnerability[11]) then reveals a valid return address which is used by the rest of the exploit. Also, system libraries are usually loaded during the boot of the system and therefore their address stays the same until the reboot. But generally bypassing ASLR is a non-trivial action complicating the exploitation process.[63][10][59]
- **DEP** DEP is a Microsoft Windows name for the NX/XD bit in CPU architectures. This technology provides a way to mark specific parts of the program (stack, heap, selected pages, etc.) as not executable. This is checked by the CPU itself and causes a CPU exception in case an address from non-executable space is loaded into the PC. DEP is usually used to mark the stack pages as non-executable, which means the attacker is not able to launch shellcode from there. This method is very effective, but it has to be combined with ASLR. Otherwise, ROP techniques might be used to bypass it. Even when the stack is non-executable, but we have the knowledge of a valid return address, we might be able to use existing sequences of instructions to deactivate DEP. However, DEP makes the exploitation of ASLR supported applications harder.[63][10][59]
- **SafeSEH** SEH works on a principle of building a list of special functions which are called when an exception occurs. Each of these functions has the ability to determine if it wants to handle the risen exception or not. SEH based exploits rewrite the address of such handlers and cause an exception - the SEH mechanism then redirects the execution flow to inserted shellcode instead of a regular exception handler. SafeSEH protects against loading a custom exception handler by checking that the address of the handler was registered as a valid exception

handler before. This way it is much harder for exploits overwriting the `EXCEPTION_REGISTRATION` block to succeed.[32][40][59]

5.2.2 Recommendations

Here we have to stress out one already mentioned fact. In the case that at least one loaded library does not support ASLR then the whole application is still not protected by it.[59] This library will get loaded at the same address every time (unless relocation happens) and therefore instructions from this library might be used for ROP technique and controlling the execution flow. In case of *Mary*, the libraries without ASLR are *elizabeth.dll* and *jennifer.dll*.

- **elizabeth.dll** This library is compiled using old and obsolete MSVC which simply does not support ASLR. The author is well aware of this for more than five years, has been notified about possible security impacts and nothing has changed. There is, therefore, no reason to think that this situation will solve itself somehow. On the other hand, *elizabeth.dll* has available source codes, and everybody is able to recompile it. This might bring up some compatibility issues, but it should be possible to resolve them, which is not the case of a missing ASLR. Also, when the rest of the compilation parameters stays the same, these issues should be minimal. Therefore, the ideal solution seems to be the recompilation of the library with ASLR, DEP and SafeSEH switched on along with fixing arising issues.
- **jennifer.dll** In case of *jennifer.dll*, the situation is much worse. Recompilation is not an option here as the source code is not public and there is no way of contacting the authors as the company no longer exists. Also, the library contains known unfixed security vulnerabilities that are patched using callback handler logic in *Mary*. On the other hand, the file format `.jen` is nowadays very rarely used. Performing a quick search on Google brings only one valid result throughout the first five pages. Also, its performance parameters are long outdated, and it has generally been replaced by newer and better formats. For these reasons, it makes sense to drop support for it and abandon this library entirely. The user impact should be minimal based on the infrequent usage of the `.jen` format.

We have contacted the authors of *Mary* mainly because of the unsecured communication, which presents an actual threat. From their quick reaction and perfect communication, it comes out that security is a priority taken very seriously by them. Also, because no buffer overflow has been found in these libraries and that our fuzzing tests are still running, we are saving the information regarding the compilation parameters for a later time. Particularly for

a moment when a valid BO is found by the fuzzing if such a moment happens. Now it is legitimate to presume that the authors are well aware of these compilation parameters. Still, they have reasons that make them keep the libraries included in their current form.

Conclusion

This thesis had several objectives. First, it was getting familiar with methods and software used in reverse engineering of computer software. Next, we were supposed to use these techniques and software to perform a vulnerability assessment of the provided application, with focus on network communication. Finally, we had to document all found weaknesses and give measures leading to their mitigation.

In the first chapter, we have just extended the introduction to make sure everybody understands what will be this thesis's aim. We have outlined the structure of the thesis, expected results and formulated the purpose of it in a detailed way.

The second chapter has been dedicated to the research. We have studied chosen attack vectors, suitable methods and up-to-date software. Here we focused on reverse engineering. To be able to perform a sophisticated review we have also examined other areas of computer security. Most notably networking, black-box fuzzing and static analysis. All this knowledge has been supported by finding and working with professional software tools designed for such purposes.

The beginning of the assessment has been performed in the third chapter. Following the research, we have used previously acquired skills to find out detailed information about the studied application, reverse engineer it and put it through an in-depth static analysis. Given actions directed us to the detection of missing protection features of several libraries, and unsecured download of the application's installer.

Next chapter has been dedicated to continuing with the assessment. We have created example fuzzing templates for file formats processed by the defective libraries and fuzzed these libraries using the Peach fuzzer. Also, we have

CONCLUSION

simulated a scenario in which a new user downloads the application from the vendor's webpage. Using HTTrack, nginx and Ettercap we were able to perform an ARP poisoning attack, leading to a download of a tampered installer.

In the last chapter, we have provided information about possible protection methods and suggested the optimal way for fixing found weaknesses.

During the creation of this thesis, we gained significant knowledge, improved our computer security skills and assessed the given application using appropriate methods. The vendor of the reviewed application has been notified of all discovered issues, and some of them got already fixed.

Bibliography

- [1] Kvasnicka, T. *Improving Web Server Content Caching Performance*. Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2015.
- [2] McClure, S.; Scambray, J.; et al. *Hacking bez tajemství*. Computer Press, third edition, 2003, ISBN 80-7226-948-8.
- [3] Jirkal, M. *Bezpečnostní studie aplikace*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2013.
- [4] Žentek, J. *Bezpečnostní studie aplikace*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2014.
- [5] Operating System Market Share Worldwide [online]. [cit. 2017-12-21]. Available from: <http://gs.statcounter.com/os-market-share>
- [6] Russinovich, M. E.; Ionescu, A.; et al. Microsoft Windows Security. *The Microsoft Press Store [online]*, March 2012, [cit. 2017-12-21]. Available from: <https://www.microsoftpressstore.com/articles/article.aspx?p=2228450>
- [7] Howard, M.; LeBlanc, D. *Bezpečný kód*. Computer Press, first edition, 2008, ISBN 978-80-251-2050-7.
- [8] Aleph One. Smashing The Stack For Fun And Profit. *Phrack [online]*, November 1996, [cit. 2017-12-21]. Available from: <http://phrack.org/issues/49/14.html>
- [9] Hatch, B.; Lee, J.; et al. *Hacking bez tajemství - Linux*. Computer Press, first edition, 2003, ISBN 80-7226-869-4.
- [10] Erickson, J. *Hacking: The Art of Exploitation*. No Starch Press, second edition, 2008, ISBN 978-1-59327-144-2.

- [11] Foster, J. C.; Osipov, V.; et al. *Hacking - Buffer Overflow*. Grada Publishing, first edition, 2007, ISBN 978-80-247-1480-6.
- [12] Lin, C. C. Understanding the PC and the IR. 2003, [cit. 2017-12-26]. Available from: https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Overall/pc_ir.html
- [13] Intel. *Intel 64 and IA-32 Architectures Software Developer Manuals [online]*. [cit. 2017-12-26]. Available from: <https://software.intel.com/en-us/articles/intel-sdm>
- [14] Kernighan, B. W.; Ritchie, D. M. *The C Programming Language*. Prentice Hall PTR, second edition, 1988, ISBN 978-0131103627.
- [15] ALEX. The stack and the heap. *Learn CPP [online]*, August 2007, [cit. 2017-12-26]. Available from: <http://www.learncpp.com/cpp-tutorial/79-the-stack-and-the-heap/>
- [16] Krishnan, M. R. Heap: Pleasures and Pains. *MSDN [online]*, February 1999, [cit. 2017-12-26]. Available from: <https://msdn.microsoft.com/en-us/library/ms810466.aspx>
- [17] Managing Memory with Multiple Heaps [online]. [cit. 2017-12-26]. Available from: https://caligari.dartmouth.edu/doc/ibmcxx/en_US/doc/libref/concepts/cumemmng.htm
- [18] Vanhoef, M. Understanding the Heap and Exploiting Heap Overflows. 2013, [cit. 2017-12-26]. Available from: <http://www.mathyvanhoef.com/2013/02/understanding-heap-exploiting-heap.html>
- [19] Buffer Overflow CVE [online]. [cit. 2017-12-26]. Available from: <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer+overflow>
- [20] Bhamidipati, S. The Art of Reconnaissance - Simple Techniques. Technical report, SANS Institute, August 2001, [cit. 2017-12-23]. Available from: <https://www.sans.org/reading-room/whitepapers/auditing/art-reconnaissance-simple-techniques-60>
- [21] Long, J. *Google hacking*. Zoner Press, first edition, 2005, ISBN 80-86815-31-5.
- [22] Ettercap and middle-attacks tutorial [online]. [cit. 2017-12-23]. Available from: <https://pentestmag.com/ettercap-tutorial-for-windows/>
- [23] Aircrack-ng [online]. [cit. 2017-12-26]. Available from: <https://www.aircrack-ng.org/doku.php?id=Main>
- [24] Dostálek, L.; Kabelová, A. *Velký průvodce protokoly TCP/IP a systémem DNS*. Computer Press, fifth edition, 2008, ISBN 978-80-251-2236-5.

-
- [25] Bigelow, S. J. *Mistrovství v počítačových sítích*. Computer Press, first edition, 2004, ISBN 80-251-0178-9.
- [26] King, J.; Lauerman, K. ARP Poisoning Attack and Mitigation Techniques. Technical report, Cisco, January 2016, [cit. 2017-12-26]. Available from: https://www.cisco.com/c/en/us/products/collateral/switches/catalyst-6500-series-switches/white_paper_c11_603839.html
- [27] Handley, M.; Rescorla, E. Internet Denial-of-Service Considerations. RFC 4732, RFC Editor, December 2006. Available from: <https://tools.ietf.org/html/rfc4732>
- [28] Junaid, M.; Mufti, D. M.; et al. Vulnerabilities of IEEE 802.11i Wireless LAN CCMP Protocol. Technical report, University of Engineering and Technology, Taxila, Pakistan, February 2006. Available from: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.188.704&rep=rep1&type=pdf>
- [29] An offline Wi-Fi Protected Setup brute-force utility [online]. [cit. 2017-12-26]. Available from: <https://github.com/wiire-a/pixiewps>
- [30] Vanhoef, M.; Piessens, F. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. Technical report, imec-DistriNet, KU Leuven, November 2017. Available from: <http://papers.mathyvanhoef.com/ccs2017.pdf>
- [31] Lyon, G. F. *Nmap Network Scanning: Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure.Com LLC, first edition, 2008, ISBN 978-0-9799587-1-7.
- [32] Reverse Engineering course at FIT CTU. [cit. 2017-12-26]. Available from: <http://bk.fit.cvut.cz/en/predmety/00/00/00/00/00/00/04/70/36/p4703606.html>
- [33] Chikofsky, E. J.; Cross, J. H. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, January 1990, [cit. 2017-12-26]. Available from: <http://win.ua.ac.be/~lore/Research/Chikofsky1990-Taxonomy.pdf>
- [34] Pietrek, M. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. *MSDN [online]*, March 1994, [cit. 2017-12-26]. Available from: <https://msdn.microsoft.com/en-us/library/ms809762.aspx>
- [35] Danehkar, A. Injective Code inside Import Table. [cit. 2017-12-26]. Available from: <http://www.ntcore.com/files/inject2it.htm>

- [36] igorsk. Reversing Microsoft Visual C++ Part II: Classes, Methods and RTTI. September 2006, [cit. 2017-12-26]. Available from: http://www.openrce.org/articles/full_view/23
- [37] Eagle, C. *The IDA PRO BOOK*. No Starch Press, second edition, 2011, ISBN 978-1-59327-289-0.
- [38] Schwarz, B.; Debray, S.; et al. Disassembly of Executable Code Revisited. Technical report, University of Arizona, Tucson, USA, October 2002. Available from: <https://www2.cs.arizona.edu/~debray/Publications/disasm.pdf>
- [39] Vijayvargiya, A. Writing a basic Windows debugger. *Code Project [online]*, [cit. 2017-12-27]. Available from: <https://www.codeproject.com/Articles/43682/Writing-a-basic-Windows-debugger>
- [40] Pietrek, M. A Crash Course on the Depths of Win32 Structured Exception Handling. *Microsoft Systems Journal [online]*, January 1997, [cit. 2017-12-27]. Available from: <https://www.codeproject.com/Articles/43682/Writing-a-basic-Windows-debugger>
- [41] Collberg, C.; Thomborson, C.; et al. A Taxonomy of Obfuscating Transformations. Technical report, The University of Auckland, Auckland, New Zeland, 1997/2009. Available from: <https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf>
- [42] Neystadt, J. Automated Penetration Testing with White-Box Fuzzing. *MSDN [online]*, February 2008, [cit. 2017-12-27]. Available from: <https://msdn.microsoft.com/en-us/library/cc162782.aspx>
- [43] Hillman, M. 15 minute guide to fuzzing. *MWR InfoSecurity [online]*, August 2013, [cit. 2017-12-27]. Available from: <https://www.mwrinfosecurity.com/our-thinking/15-minute-guide-to-fuzzing/>
- [44] Böhme, M.; Pham, V.-T.; et al. Directed Greybox Fuzzing. Technical report, National University of Singapore, November 2017. Available from: <https://acmccs.github.io/papers/p2329-bohmeAemb.pdf>
- [45] Weinberg, G. M. Fuzz Testing and Fuzz History. February 2017, [cit. 2017-12-27]. Available from: <http://secretsofconsulting.blogspot.cz/2017/02/fuzz-testing-and-fuzz-history.html>
- [46] Rawat, S.; Jain, V.; et al. VUzzer: Application-aware Evolutionary Fuzzing. Technical report, Vrije Universiteit Amsterdam, March 2017. Available from: http://sharcs-project.eu/m/filer_public/48/8c/488c5fb7-9aad-4c87-ab9c-5ff251ebc73d/vuzzer_ndss17.pdf

- [47] Chronozon - An evolutionary knowledge-based fuzzer. [cit. 2017-12-27]. Available from: <https://github.com/CENSUS/choronzon>
- [48] Garg, P. Fuzzing: Application and File Fuzzing. *InfoSec Institute [online]*, January 2012, [cit. 2017-12-27]. Available from: <http://resources.infosecinstitute.com/application-and-file-fuzzing/>
- [49] Clang 6 documentation. [cit. 2017-12-27]. Available from: <https://clang.llvm.org/docs/>
- [50] Static Code Analysis. [cit. 2017-12-27]. Available from: https://www.owasp.org/index.php/Static_Code_Analysis
- [51] Wheeler, D. A. Flawfinder. [cit. 2017-12-27]. Available from: <https://www.dwheeler.com/flawfinder/>
- [52] A curated list of static analysis tools, linters and code quality checkers for various programming languages. [cit. 2017-12-27]. Available from: <https://github.com/mre/awesome-static-analysis>
- [53] Hicken, A. False Positives in Static Code Analysis. *Parasoft blog [online]*, February 2013, [cit. 2017-12-27]. Available from: <https://blog.parasoft.com/false-positives-in-static-code-analysis>
- [54] Friedl, S. An Illustrated Guide to the Kaminsky DNS Vulnerability. *Steve Friedl's Unixwiz.net Tech Tips [online]*, July 2008, [cit. 2018-1-2]. Available from: <http://unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html>
- [55] Son, S.; Shmatikov, V. The Hitchhiker's Guide to DNS Cache Poisoning. Technical report, The University of Texas at Austin, September 2010. Available from: https://www.cs.cornell.edu/~shmat/shmat_securecomm10.pdf
- [56] Wagner, R.; Bryner, J. Address Resolution Protocol Spoofing and Man-in-the-Middle Attacks. Technical report, SANS Institute, June 2006. Available from: <https://www.sans.org/reading-room/whitepapers/threats/address-resolution-protocol-spoofing-man-in-the-middle-attacks-474>
- [57] Cheese. TCP Session Hijacking. Technical report, Packetstorm, January 2010. Available from: <https://packetstormsecurity.com/files/85017/TCP-Session-Hijacking.html>
- [58] Lin, M. An Overview of Session Hijacking at the Network and Application Levels. Technical report, SANS Institute, January 2005. Available from: <https://www.sans.org/reading-room/whitepapers/ecommerce/overview-session-hijacking-network-application-levels-1565>

BIBLIOGRAPHY

- [59] Corelan Team. Exploit writing tutorial part 6: Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR. *Corelan [online]*, September 2009, [cit. 2018-1-3]. Available from: <https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/>
- [60] Gavrichenkov, A. Breaking HTTPS with BGP hijacking. Technical report, Qrator Labs, 2015. Available from: <http://www.blackhat.com/docs/us-15/materials/us-15-Gavrichenkov-Breaking-HTTPS-With-BGP-Hijacking-wp.pdf>
- [61] Kumar, P. How to embed an exe inside another exe as a resource and then launch it. *Geeks with Blogs [online]*, October 2009, [cit. 2018-1-3]. Available from: <http://geekswithblogs.net/TechTwaddle/archive/2009/10/16/how-to-embed-an-exe-inside-another-exe-as-a.aspx>
- [62] Deja vu Security. Peach PIT. [cit. 2018-1-3]. Available from: <http://community.peachfuzzer.com/v3/PeachPit.html>
- [63] Sutherland, G. How does ASLR and DEP work? [cit. 2018-1-3]. Available from: <https://security.stackexchange.com/questions/18556/how-do-aslr-and-dep-work>

Acronyms

DVD	Digital Versatile Disc
USB	Universal Serial Bus
OS	Operating System
DHCP	Dynamic Host Configuration Protocol
UAC	User Account Control
GUI	Graphical User Interface
CPU	Central Processing Unit
PC	Program Counter
ASLR	Address Space Layout Randomization
DEP	Data Execution Prevention
SEH	Structured Exception Handling
IPMI	Intelligent Platform Management Interface
SUID	set user ID upon execution
SGID	set group ID upon execution
BIOS	Basic Input-Output System
UEFI	Unified Extensible Firmware Interface
SMTP	Simple Mail Transfer Protocol
POP	Post Office Protocol
SSH	Secure Shell

A. ACRONYMS

HTTP Hypertext Transfer Protocol

DNS Domain Name System

MS-RPC Microsoft Remote Procedure Call

IDS Intrusion Detection System

IPS Intrusion Prevention System

MitM Man-in-the-Middle

AP Access Point

ARP Address Resolution Protocol

ICMP Internet Control Message Protocol

BGP Border Gateway Protocol

FTP File Transfer Protocol

SMB Server Message Block

VPN Virtual Private Network

TCP Transmission Control Protocol

HTTPS HTTP Secure

SFTP SSH File Transfer Protocol

DoS Denial-of-Service attack

UDP User Datagram Protocol

XML Extensible Markup Language

NIC Network Interface Controller

DDoS Distributed Denial-of-Service

WEP Wired Equivalent Privacy

WPA Wi-Fi Protected Access

WPS Wi-Fi Protected Setup

SSID Service Set Identifier

VLAN Virtual LAN

IP Internet Protocol

TTL Time-to-Live
ABI Application Binary Interface
API Application Programming Interface
MS Microsoft
PE Portable Executable
DLL Dynamic-link Library
IAT Import Address Table
RTTI Run-Time Type Information
NX/XD Non eXecute / eXecute Disable
VM Virtual Machine
AI Artificial Intelligence
ISP Internet Service Provider
MAC Media Access Control
NFS Network File System
SQL Structured Query Language
IEEE Institute of Electrical and Electronics Engineers
GNU GNU's Not Unix!
ROP Return oriented programming
MSVC Microsoft Visual C++
CRC Cyclic Redundancy Check
SHA Secure Hash Algorithm
MD Message-Digest Algorithm
AES Advanced Encryption Standard
DES Data Encryption Standard
RAM Random Access Memory
JS JavaScript
CDN Content Delivery Network
SSL Secure Sockets Layer

Contents of enclosed CD

```
src ..... the thesis LATEX source codes directory
├── DP_kvasnicka_tomas_2017.tex ..... the thesis text in TEX format
├── mybibliographyfile.bib ..... the thesis bibliography in TEX format
├── FITthesis.cls ..... the thesis template in TEX format
├── iso690.bst ..... the thesis bibliography template in TEX format
├── images ..... directory with images used in this thesis
└── text ..... the thesis text directory
    ├── DP_kvasnicka_tomas_2017.pdf ..... the thesis text in PDF format
```