Java Applet Client Security

Year 2006/2007

Dan Lund Christensen
Department of Computer Science
Aalborg University

Faculty of Engineering and Science
University of Aalborg(AAU)

Department of Computer Science

ABSTRACT:

This report documents the attempt to raise security for clients against a threat model, with these three attacks; Man In The Middle (MITM), phishing, and tampering attacks.
The analysis describes security mechanisms and existing technology to prevent the three attacks in the threat model. It was discovered in the analysis that there exists technology to prevent MITM and phishing attacks, but no security mechanism exist to prevent tampering of Java client programs.
Instead of designing and implementing existing technology as security mechanisms to prevent MITM and phishing attacks, the project should be, to design and implement a possible tamper-proofing security mechanism for Java client programs. In the design and implementation chapters, documentation of the prototype tamper-proofing security mechanism is found.
The test shows that the security mechanism is not sufficient, to ensure that the client was actual tamper-proof. However, when the prototype was tested, a nice property was found in the embedding of an illegal byte array operation, which could make it possible to allow one to trust the client for a small period of time.

TITLE:
EN: Java Applet Client Security
DK: Java Applet Klient Sikkerhed

PROJECT PERIOD:
Master Thesis,
September 1st 2006 - June 12st 2007

Author:
Dan Lund Christensen,
dalc@cs.aau.dk

SUPERVISOR:
Ivan B. Damgård
ivan@daimi.aau.dk
Josva Kleist
kleist@cs.aau.dk

NUMBER OF COPIES: 6

NUMBER OF PAGES: 56

Preface

This report is the master's thesis written by Dan Lund Christensen from the SW10 semester at the Department of Computer Science, Aalborg University.

The goal of the project is to analyze, which security mechanisms a client implemented in Java will require to provide security against the attacks defined in the threat model. As no security mechanism existed against the tampering attack, the project then changed to design and implementation project of the security mechanism to prevent this attack. Finally the designed and implemented tamper-proofing security mechanism has been tested, to observe whether it actual was secure against the tampering attack.

The report is split into six parts arranged as chapters.

The first chapter is the introduction of the problem. The introduction contains the requirements to the client, the threat model of the attacks to prevent, and a summary of what the client should be secure against.

The second chapter is an analysis of the security mechanisms, that can prevent the attacks from the threat model.

The third chapter is the design of the tamper-proofing security mechanism, where the design decisions are documented.

The fourth chapter is the implementation of the tamper-proofing security mechanism, where the implementation decisions are documented and which problem have been encountered and how they were solved.

The fifth chapter contains the test-case and test results of the tamper-proofing security mechanism, which is used for concluding whether the tamper-proofing security mechanism is secure against the tampering attack.

The sixth chapter is the conclusion of the entire Java Applet Client Security project.

Reading Instructions

The report can be read by other students from computer science with basic knowledge of data security. Bibliography references are marked with [x], where the x is the reference

that can be looked up in the bibliography at the end of the report before appendix. Just after the bibliography is a list of acronyms used. The first time an acronym appears in the report it has this structure Secure Socket Layer (SSL), and later when used the acronym will appear like this SSL.

Acknowledgements

I would like to thank the people who made it possible for me to take courses at Århus University and get external supervision. These people are Uffe Kjærulff who helped me through the bureaucracy at Aalborg University, my external supervisor Ivan B. Damgård, and Josva Kleist for being the additional supervisor required by Aalborg University regulations.

Signature

_____

Dan Lund Christensen

Contents

Chapter 1

Introduction

The topic of this master thesis is Java applet client security. The problem to research is the fact that clients function, in what shall be considered a hostile environment. The goal is then to implement security mechanisms in a client prototype, which makes it less prone to specific attacks. The question whether it is possible to implement a set of security mechanisms that makes the client function securely in hostile environments? The function of the client is to input data into a database and know the user that did. These requirements to the solution are first written down. The requirements to the solution are followed by a discussion of the security the solution shall provide.

1.1 The Solution

Before selecting specific security mechanisms, the actual requirements to the solution are formalized. The solution that consists of a client and a server is to support the list of requirements below. Each entry on the list has parentheses at the end. In the parentheses is a guess of the security mechanism or functionality to use for fulfilling the requirement.

Requirement 1: The user authenticate the server and authenticate herself to the server. (User authentication)

Requirement 2: The user can submit data to a server using an input form, after completing the authentication successfully.(Server receives data from input forms)

Requirement 3: Communication between client and server is confidential.(Communication encryption)

Requirement 4: Communication between client and server is authentic.(Transaction validation)

Requirement 5: It is easy to deploy new input forms.(Java applets)

Requirement 6: The server authenticate the client program running on the client.(Tamper-proofing and client validation)

Requirement 7: The security mechanisms are hard to circumvent.(Security through obscurity)

These requirements support the following scenario. The user authenticates herself and the server while the server authenticates the client program. The client type in the relevant data and sends it to the server. The server sends a confirmation that data has been received.

By observing the list of requirements with regards to security mechanisms, one would come up with the following list of security mechanisms; security through obscurity, user authentication, communication encryption, transaction validation, tamper-proofing, and client validation.

1.2 Security

To discuss security, one should first find the relevant attacks to be in the threat model. To do this one can use the list of security mechanisms that the solution description introduces. The attacks to select for the threat model will be; MITM attacks, phishing and tampering, these attacks are afterward put into the categories; External, Insider, Network, Off-line, and On-line (EINOO) where relevant. The EINOO categories were introduced in the data security course at DAIMI, Århus University, and no literature exists that describes these categories in greater detail.

1.2.1 Threat Model

This section defines the threat model with the relevant attacks. Each attack has a short description.

- Man In The Middle:
  The passive MITM attack uses sniffing to obtain network packages, afterward analyzing the network packages to obtain information, about what have been sent between the client and server. However, there are active MITM attacks, where the attackers have the intent of interrupting or change the information that flow between server and client. The attacker can do it by altering the data send to the server or client.

- Phishing:
  This attack is for obtaining personal information[DTH06]. The attacker mimics the official web server and its content, in order to trick the user into trust the website and use it to input personal information.

- Tampering:
  This is when the re-engineer change the client to circumvent the security mechanisms. It is possible because the client executable should be assumed public accessible.

1.2.2 EINOO Categories

The attacks from the threat model are sorted into the EINOO categories below.

- External:
  The attack is possible without use of insider privileges.
- Insider:
  The attack is possible with insider privileges.
- Network:
  The attacks that are possible against the network.
- Off-line attack:
  The attack is possible without real-time access to the system.
- On-line attack:
  The attack is possible with real-time access to the system.

The goal is to place the attacks within the relevant categories to clarify the attack; who can perform the attack, do it target the network, or is it an attack that requires on-line access to the system.

It is possible to place one attack within one or more attack categories. Where to place each attack is shown in Table 1.1.

Categories	Attacks
External	MITM, phishing, tampering
Insider	MITM, phishing, tampering
Network	MITM
Off-line	phishing, tampering
On-line	MITM

Table 1.1: In this table the attacks are put into the relevant categories.

The MITM attack is an attack, that either an external or insider party can perform on the network, but require the system to be on-line for the attack to be successful. The phishing attack is an attack done by either an external or insider party, where the attacker trick an insider into leak personal using a phony Graphical User Interface (GUI), the phishing attack does not require the system to be on-line for success. The last attack from the

threat model is the tampering attack. The attacker using this attack can either be an external or insider party, and the attack does not require on-line access to the system.

The conclusion is to consider both the external and insider parties as hostile. Only the MITM attack requires access to the network and that the system is on-line. The phishing and tampering attack can be completed without access to an on-line system, as long they have a version of the client program. To be resistant against tampering attacks it will require the re-engineer to restart the attack with each new client release. Re-engineering of programs is always possible because computers have to interpret the program.

1.3 Summary

The policies for the security mechanisms to enforce are the requirements listed in Section 1.1, and the security mechanisms are; security through obscurity, user authentication, communication encryption, transaction validation, tamper-proofing and client validation. The security mechanisms chosen shall in theory make previous listed attacks hard to deploy against the system. The argumentation for the security is done with a foundation in theory, from the perspective of each security mechanism. To handle it practically would require implementation errors being in the scope of this project which it is not, because even a good programmer cannot promise no implementation errors, which is also stated in [Vir91]. However, no architectural flaws should be present in the design and implementation.

Chapter 2

Analysis

This chapter contains an analysis of these necessary security mechanisms; security through security, user authentication, communication encryption, transaction validation, tamper-proofing and client validation, to fulfill the requirements and scenario in Section 1.1. The idea is to describe the security mechanisms, and compare the security mechanisms against already existing technology. In the analysis of each security mechanism is a discussion of which security the security mechanism offer. The two security mechanisms communication encryption and transaction validation is sub-security mechanisms to the secure channel security mechanism. The tamper-proofing and client validation security mechanisms should be understood as one security mechanism, because the two security mechanisms possible will rely on the same technology or idea.

2.1 Security through Obscurity

The problem with security through obscurity is that the re-engineer can un-cover all the information present in the program because computers are deterministic. In this section one can view the re-engineering of an example program, to observe if it is possible for a security mechanism to rely on hiding information in programs, because if re-engineering can reveal the hidden information, the attacker can use the information to compromise the system. The analysis is done with a practical perspective, which is to test the tools available for re-engineering and obfuscation. Obfuscation is interesting because of its use as protection against re-engineering. The conclusion of the analysis should be a process or tool to hide information in programs.

The tool to use for disassembling is `javap` program released with The SUN Java Developer Kit (JDK) and the tool to use for obfuscation is the ProGuard[1] program.

[1] http://proguard.sourceforge.net

2.1.1 Disassembling

Disassemblers are a program that requires a program file as input, and then output a file containing program instructions. The important thing in this project is that the step of translating a program file to some understandable file format is impossible, however, because it cannot be impossible it should be difficult.

In this project the Java programming language is the programming chosen for implementing the prototype, and then it is important to know what information the Java class file stores. This states what information can be deduced from class files in the disassembling process. In the programming language C the binaries that correspond to class files contain instructions and memory addresses, compared to Java everything from Java source code is present in class files except comments.

As a result of the information the class file stores, the disassemblers can output an instruction file with variable names, this are seen in the following two code listings, the Java source file for the program is in Appendix B Listing B.1 and the disassembling output of the Java class file is in Appendix B Listing B.2.

2.1.2 Run-Time Analysis

Re-engineers often use debuggers to do run-time analysis which consist of monitoring program memory and printing stack-traces, with the result that attackers can see program flow and data in program memory. An example where monitoring of program memory can be useful is when a security mechanism is hidden within class object, and the creation of that object happen when the program starts, then if the garbage collector do no collect this object while the program is running. The attacker knows that a security mechanism protecting the program from start to end and that this object is a likely candidate of objects that contain the security mechanism. To avoid this problem the disposing and creation of objects that contain security mechanisms shall be done several times while the program is running. The solution of creating and disposing of objects is still vulnerable to another attack, because the creation of objects of the same type will resolve in objects with the same structure in program memory. The attack would then be to search the heap for objects with a specific structure. Stack-traces have the same property that allows to search for specific call stack-trace patterns.

2.1.3 Obfuscation

The obfuscation process is to decrease the readability of byte-code, the evaluation of readability of byte-code is done with a comparison on readability between two instances of the same example program and then obfuscate one of the instances. After the disassembling of the two program instances one have two results to compare.

ProGuard obfuscation program consist of these three steps:

1. Shrinking byte-code.(dead code removal)
2. Optimizing byte-code.(analyzes and optimizes the byte-code of methods)
3. Obfuscating byte-code.(renames variables, methods, classes, etc.)

The disassembling of the obfuscation program instance is shown in Appendix B Listing B.3, and a comparison is done with the disassembling of the normal program instance shown in Appendix B Listing B.2. The comparison of the two listings show similar instruction sequences, but renaming is done on variable and method names with the exception of the program entry point, which ProGuard require set with the `-keep` option in the ProGuard configuration file shown in Appendix B Listing B.4. The only decrease of readability this offer is that the variables and methods names are less meaningful. One should also set the option `-allowaccessmodification` which is done in this trial obfuscation of the example program. It allow modifications to the access modifiers which has the result that the protected field variable in Appendix B Listing B.2 line 5 is the public field variable in Appendix B Listing B.3 line 4. By allowing the access modifiers to change do decrease readability, because it makes writing search patterns for searching tools difficult, while the search patterns cannot rely on a non-changing access modifier.

The use of ProGuard on bigger programs like the Ant project's `ant.jar` the obfuscation process remove 117 classes, and the new size of `ant.jar` was 149.658 bytes instead of the original size of 1.289.806 bytes. The conclusion is that the obfuscation process done on bigger projects can output smaller programs that use fewer resources for running.

2.1.4 Security

It can be observed on the disassembling listings Appendix B Listing B.2 and Listing B.3 that code obfuscation makes readability decrease. The byte-code obfuscation only gives limited security, because the security is still rests on an obscurity problem that is solvable. However, our goal of the obfuscation process is that hidden information cannot be found and decrease the readability to make circumvention of security mechanisms hard. It can be concluded that the obfuscation offered by ProGuard will not be enough to obtain these properties. The next step would then be to change the key and its position with each client release, which makes a believed to be a harder re-engineering problem. This re-engineering problem is believed to be harder, because new key information has to be found in each client release, compared to non-changing key information. The conclusion is that re-engineering problems are solvable, but using a tool such as ProGuard can help on performance. The final conclusion is that the project cannot only rely on an obfuscation tool like ProGuard for security.

The result is then to establish trust on the obfuscation of key information in programs, can only be successful for short periods of time starting after the program is made public. The time period have to be less than the time it takes to re-engineer the key information.

2.2 User Authentication

The process of validating the authenticity of a user is a requirement of most client software. The goal is to implement a user authentication mechanism, which make it problematic to deploy a successful phishing attack. Both the user authentication form and the data input forms can be vulnerable to phishing attacks. In [DT05] it is claimed that the solution described makes phishing attacks hard. This is why an analysis of the Trusted Path solution is done.

2.2.1 Trusted Path

The idea described in [DT05] is that authentication data, sent between client and server does not reveal any personal information. This is combined with dynamic security skins that ensure the client of the server's identity and vice versa. Dynamic security skins work as follows. Each user selects a specific image from a collection of images, the image is laid over the authentication form with a per session generated visual hash, a visual hash is an auto-generated picture using a regular hash value as input. The client compute a visual hash laid over the authentication form, and the server compute a visual hash laid over the input form. The visual hash is laid over the authentication form and the input form, the user then compare whether the two visual hashes are the same. If the visual hashes are not the same, it is likely a phishing attack taking place. The result is that phishing attacks becomes hard to perform, because to make the look-a-like input form, the attacker has to know the user selected image and the computed visual hash. Figure 2.1 is an example of how this solution could look, observe how the authentication form is a separate program from the data input form.

Phishing attacks are easy to perform within the content frame of the browser, but hard outside the content frame of the browser. This is because the content frame of the browser, is controlled by web-server providing the web-page. The authentication security mechanism makes use of a program outside the content frame of the browser to make the deployment of phishing attacks hard. The security the outside authentication program offer is then appended to the data input form in the browser's content frame using visual comparison.

The prototype in [DT05] consists of the two components Secure Remote Password protocol (SRP) and dynamic security skins. The SRP is the underlying authentication protocol that handles the user validation, which Section 2.2.2 describes SRP in greater detail. The dynamic security skins are the use of visual hashes on the authentication and data input form that create a trusted path, because each user has a unique selected picture and session generated visual hash laid over the authentication form, which it can compare with the visual hash of server provided data input form. The goal of using dynamic skins is to prevent phishing attacks, by making it hard for an attacker to make copies of the forms, because the attacker has to imitate each user's unique image and visual hash on the forms.

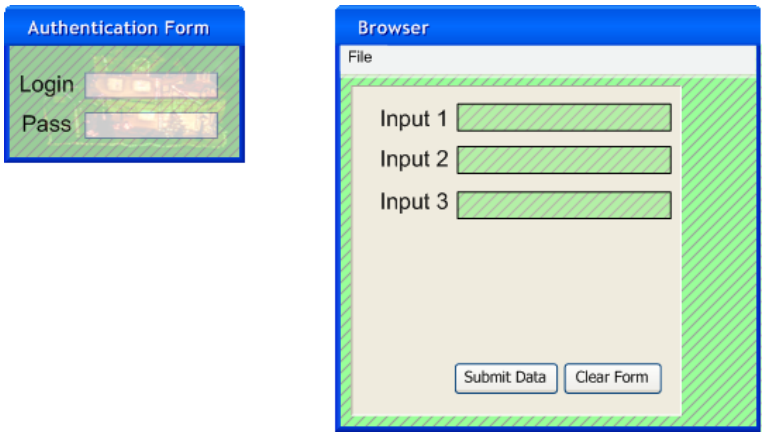

Figure 2.1: This shows two programs and how they are connected through visual graphics, which are the solution [DT05] propose.

Dynamic Security Skins

The client using the developed security mechanism should not be vulnerable to automated attacks. The user authentication scheme should be able to obtain this property, because each user personalizes her client with dynamic security skins. However, the client using the developed security mechanism would still be vulnerable to personal attacks. This is stated in the stories in [MS05]. One possible personal attack would be to spy on the user, and observe which image she uses for personalization. The attacker can just compile a client with that image, and replace the user's client software with the newly compiled version that has the same visual hash computation function.

In theory the dynamic security skins makes automated attacks hard, which targets clients using the proposed security mechanism. If dynamic security skins are not implemented properly, the attackers would be able to make an automatic attack that gives her knowledge of which personal image each user use.

The idea of the solution in [DT05] uses a trusted login form to assign trust to browsers content frame. This can be either a separate program or a plug-in to a browser. The solution in [DT05] uses a plug-in to a Firefox, because that browser's plug-in window is independent of the hostile content frame.

2.2.2 Authentication Protocols

There are multiple protocols for authentication and [DT05] use SRP which is relative new protocol. Comparing SRP with SSL would the most obvious choice, because SSL is widely used for internet commerce. Next are the descriptions of SRP and the SSL protocol.

Secure Remote Password protocol

The development of the SRP has happened in four versions, which have been called SRP-1, SRP-2, SRP-3, and SRP-6. The version to be implemented in the security mechanism is SRP-6[Wu02]. The fundamental literature for SRP is [Wu98].

SRP authentication is secure against sniffing[Wu98], because the information revealed, cannot be used to make a new authentication. The client and server uses a mutual shared secret, which is made by the client the first time it communicates with the server. The mutual shared secret is a password, which is used with user identity and a random salt to compute the verifier. The server can then use the verifier to authenticate the user.

SRP is an authentication and key-exchange protocol, which is designed to resist both passive and active MITM attacks.

The description of the SRP-6 user validation handshake is also found in[Wu02]. The description is an explanation of Figure 2.2.

	Client		Server
1.		$\xrightarrow{I}$	(lookup s, v)
2.	$x = H(s, I, P)$	$\xleftarrow{s}$	
3.	$A = g^a$ $u = H(A, B)$	$\xrightarrow{A}$ $\xleftarrow{B}$	$B = 3v + g^b$ $u = H(A, B)$
4.	$S = (B - 3g^x)^{a+ux}$		$S = (Av^u)^b$
5.	$M_1 = H(A, B, S)$	$\xrightarrow{M_1}$	(verify M_1)
6.	(verify M_2)	$\xleftarrow{M_2}$	$M_2 = H(A, M_1, S)$
7.	$K = H(S)$		$K = H(S)$

Figure 2.2: The SRP-6 authentication handshake. The figure can be found in [Wu02](page. 5)

In Figure 2.2 numerous variables needs to be introduced:

I: Identity, the user name.

P: Password, the users password.

s: Random salt, a random numeric value.

v: Verifier, the verifier is the computed on the server with the using this equation $v = g^{H(s,I,P)}$.

H(): Hash function, in this case one should use Secure Hash Algorithm-1 (SHA-1).

a,b,u: Random values.

K: The computed session key.

The group under which SRP operates is defined by a large safe prime N and a primitive root g, because SRP uses arithmetic operations and the data submitted are bytes. A function that converts byte arrays into integers is necessary. The verification of M_1 and M_2 are a check if M_1 is equal to M_2. A result from the authentication process is, that both the client and the server computes the same session key K. One can then use the session key to encrypt every transaction between client and server.

When initializing the SRP identity of a user at the server, the user selects a random salt s and computes a verifier $v = g^{H(s,I,P)}$ and sends both values to the server. The server can then later on identify the user using the random salt and the verifier. The security in SRP is that the verifier and random salt are sent once with the first authentication. If the attacker obtains the random salt and the verifier, it would require an expensive dictionary attack to obtain the password.

Secure Socket Layer

SSL is a widely used protocol for secure communication, because most web browsers are programmed with SSL support[Tan03, pages 956-960], when used in web browsers it is called HyperText Transmission Protocol, Secure (HTTPS). It is developed by Netscape and SSL is in version 3.0 which is from about 1996. It is used for authentication, negotiation of encryption algorithms and cryptographic keys. Its architecture enables it for use in other applications that do not use HTTPS to communicate, because the architecture is such that a layer is added between the application layer and the Transmission Control Protocol (TCP) layer. The SSL layer then encrypts data from the application layer to the TCP layer, and decrypts data from the TCP layer to the application layer. The SSL layer also starts the SSL handshake used for authentication, when a client requests a secure socket. SSL is a layer consisting of multiple protocols, it support multiple cryptographic algorithms. The SSL handshake protocol is the important part when it comes to this project, because is can handle the client and server authentication and key exchange for the encryption of application data. The abstraction level is higher compared to the abstraction level of the SRP description, because SSL supports multiple cryptographic algorithms and SRP is one cryptographic algorithm.

The SSL handshake can be seen in Figure 2.3 which is also described here. The client starts setting up the secure socket to a server by sending a `ClientHello` message to the server in question. The server then reply with a `ServerHello` message that can be appended some optional information: `Certificate`, `CertificateRequest`, and `ServerKeyExchange`. When the client receives the `ServerHello` message depending on which information is in the message, it can do one or more of the following options: `Certificate`, `ClientKeyExchange`, `CertificateVerify`, `changecipherspec`, and `Finished`. In the `ClientHello` and `ServerHello` messages contains protocol relevant information used to negotiate SSL version, cryptographic algorithms, and compression algorithms. The result of the negotiation is set by the `changecipherspec` action. The negotiation result is then send to the server, so the client and server has the same cipher specification, followed by the client being `Finished`, and the server set the cipher specification with `changecipherspec` then the server is `Finished`. Now application data can be transmit-

ted between authenticated client and server, and at the same time the application data is encrypted with the session key.

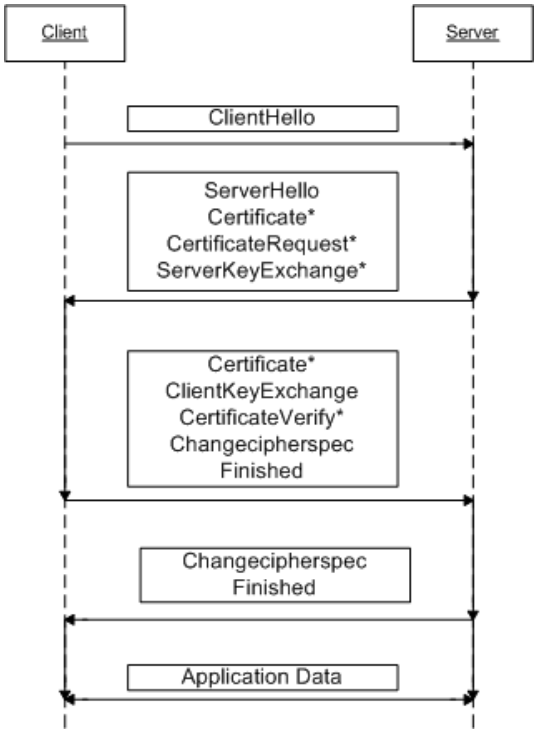

Figure 2.3: The SSL authentication handshake. The figure can be found in [FKK96]. The star indicate optional or situation dependent messages.

The normal use of SSL is the following. The clients initiate by requesting a secure socket from a server with a `ClientHello` message. The `ServerHello` message is delivered with the server certificate and completes `ServerKeyExchange`. The Client then do `CertificateVerify`, complete `ClientKeyExchange`, call `changecipherspec` and send `Finished` to the server. The server then calls `changecipherspec` and send `Finished`. However, in this scheme the client is not validated, because many application do not rely on people having certificates generated by a Certificate Authority (CA). Additional program in then often introduced that require a login and password from the user, that is sent over the established secure socket to the server for authentication.

2.2.3 Comparison of SRP and SSL

SRP and SSL are two protocols that are very similar, they both support authentication and key-exchange functionality. However, the authentication is done in two different ways. SSL relies on both parties having certificates issued by a CA, where the SRP relies on the client having a user-name and password and the server having a random salt and password verifier. The password verifier is issued by the client and stored on the server when the user is created. Both SSL and SRP has a weakness, SSL relies on that each client has a certificate issued by a CA, or extra software that use the secure socket to

make a user-name and password authentication. SRP has to rely on that it communicates with the right server, when the verifier is first send to the server.

At one point there is a big difference, which are the maturity and use of SSL compared to SRP. SSL is a well tested communication layer that consists of multiple protocols that use Internet Protocol (IP) and TCP. Most browsers and programming libraries offer SSL functionality, here is a list of just a few products that support SSL: Internet Information Services (IIS), Apache, Internet Explore, Firefox, Netscape, and Payment Systems. SRP has been test implemented with a couple of programs such as a version of telnet and JBoss. SRP is in the draft phase of being made a ciphersuite in Transport Layer Security protocol (TLS) and SSL[Wu98][TWMP06], to add user-name and password authentication supported in TLS and SSL.

2.2.4 Security

The security for this security mechanism should be the same as stated in [DT05], which states that a phishing attack is hard to complete successfully. The first login information send in the user authentication security are not send encrypted, however, the use of the SRP protocol for this ensure no information leak besides user-name. The use of SRP does not reveal any more information than the user-name, the attacker is only then given the option of on-line dictionary or brute-force attacks on the verifier the server store. This is easy to detect by the server and take appropriate measures, which could be to ban the IP address of the attacker.

2.3 Secure Channel

The communication between the client and server shall be done over a secure channel. The definition of a secure channel, is a communication media where the attacker watches the traffic between two parties and do not learn any information about the content of the conversation. At the same time they validate the integrity of the traffic the other party sends, and then if the attacker alters any traffic the receiver will detect it.

The secure channel then consist of two security mechanisms, the first security mechanism is communication encryption, the second security mechanism is transaction validation for validating the integrity of the traffic. Security arguments for both communication encryption and transaction validation are the final section.

2.3.1 Communication Encryption

The requirement to the client software is that it is not vulnerable to the following attacks: MITM, phishing, and re-engineering. To make the software solution resistant against passive MITM, one has to encrypt the data send between client and server. To encrypt

communication securely a secure crypto-system is necessary, multiple of assumed secure crypto-systems exists Advanced Encryption Standard (AES) and Rivest, Shamir, and Adleman (RSA) are just a couple of them.

2.3.2 Possible Techniques to Use

This security mechanism is trivial to implement, because the Java Collections Framework (JCF) contain packages with name `java.security` and `javax.crypto` that offer complete implementations of AES, RSA, and other cryptographic functionalities. Additional these packages is widely used and well documented.

It is even more trivial to implement communication encryption because the standards such as SSL and TLS offer communication encryption, and the JCF contain packages that supports SSL and TLS.

2.3.3 Transaction Validation

The transaction validation is solutions that do a check for changes on each transaction between client and server.

The idea is to validate each transaction from client to server with a check of a message digest. It increases security against the active MITM attacks, by ensuring no bits are flipped in the transaction. However, it is important to use a key for generating the signature or Message Authentication Code (MAC), because then security of the scheme would rely on the security of the key. The input to the validation function is the data, the key and the digest, and then it detects alterations between data and the digest. For calculating a signature or MAC one will use a cryptographic key, in one scheme a secret key is shared among the client and the server to generate the MAC, and the other scheme use a public key digital signature.

In this project two possible methods for obtaining such cryptographic keys are considered, 1) to use the session key computed in user authentication see Section 2.2.2, 2) to hide the key within client code. The first method is similar to the one used in SSL to generate MACs. The quality of the second method is questionable when hiding cryptographic keys within client code. In accordance to the possibility of re-engineering and tampering, the SSL approach will be the most secure. And the same could be done with the digital signature scheme.

2.3.4 Possible Techniques to Use

The following is a discussion of MAC and Digital Signature. Argumentation for which technique to use will be presented. First requirements are stated for the client validation process.

1. A check is done on each transaction.
   The check is deemed necessary on each transaction, because there is a risk that any transaction can be attacked.

2. Low usage of bandwidth.
   Is a requirement that bandwidth usage and computational resources are minimized, because each client has to submit a check value with each transaction. An example if the bandwidth usage is not low, the solution would scale badly with consideration to server bandwidth load.

MAC[Sti06, pages 140-152] and digital signing[Sti06, chapter Digital Signature] are two quite similar methods. First MAC is discussed. A MAC is attached to each transaction and is a hash computed from the transaction content. The method that creates the MAC has to use a key, because if no key is used, one only has to re-engineer the hash function.

Signing each transaction with a digital signature is equal to the hash function used for MACs. The function for digital signing and the hash function for MACs should both be public. And the security of the schemes is stated by the security of the key.

The decision of using MAC or Digital Signatures can based on performance, it is faster to compute a MAC than a signature. This is why MAC is to be used for the transaction validation security mechanism. When implementing a MAC scheme one should use standards, the standard to use is Keyed-Hash Message Authentication Code (HMAC) described in [Sti06, page 143], which is a MAC algorithm that takes transaction and 512 bit key as input, it was adopted by Federal Information Procession Standards (FIPS) as a standard in March, 2002.

The conclusion is to use HMAC for doing transaction validation, because HMACs are faster to compute than digital signatures. Both SSL and TLS use HMAC for transaction validation, then makes sense to use SSL for secure channels from the transaction validation perspective.

2.3.5 Security

The security of this security mechanism relies on the HMAC scheme being secure and that the session key generated is secure. The security is good with regards to active MITM attacks, because the receiving party can detect changes to the transaction. Because SSL offers both communication encryption and transaction validation, it would be an obvious choice for managing secure channels. The user authentication security mechanism requires SRP, and SSL is being made to support SRP.

2.4 Tamper-Proofing and Client Validation

The security mechanisms to place in the client software handles specific tasks, however, there should be a security mechanism to protect them against tampering. This security mechanism is to make the client software tamper-proof, which allow the server to detect if there have been tampering with the client software, and at the same time make sure that it only accepts input from official client releases.

The digital watermarking method found in [PKK+00] has promising properties. The method inserts a dynamic data structure into the program code, which is not easy to extract without leaving residues of the data structure behind. The dynamic data structure is the watermark. The watermark or residues in the program can the owner use to check ownership of the program in question. The problem is to check this property, a dump of the program running is necessary, which the client should send to the verifying party in this case the server. However, this dump of the running program can leak personal information to the server, because the dump could contain a password, private variables, and etc. the server should not have access to. The result is techniques that require program dumps are of no use.

Another solution would be trusted computing which is a solution where the computers contain trusted hardware or is added a dongle with trusted hardware. The trusted hardware contains small programs to make integrity checks or store parts of the program itself. These hardware solutions are outside the project scope, because focus of this project is on a strictly software solutions.

2.4.1 Possible Techniques to Use

The following is a discussion of these software techniques checksum, MAC, and Digital Signature for tamper-proofing and client validation. Argumentation for which technique to use will be presented. First one should state the requirements for the tamper-proofing and client validation security mechanism, which are the same requirements listed in Section 2.3.3.

1. A check is done on each transaction.
   The check is deemed necessary on each transaction, because there is a risk that any transaction can be send from an un-official client.

2. Low usage of bandwidth.
   The requirement that bandwidth usage should be low is, because each client has to submit a check value with each transaction. If the bandwidth usage is not low, the solution would scale badly with consideration to server bandwidth load.

MAC[Sti06, pages 140-152] and digital signing[Sti06, chapter Digital Signature] are two quite similar methods that was first introduced in Section 2.3.3 for transaction validation.

The difference is to use the second method for obtaining the key. The second method is where the key is stored in the client program, which is then unique with each client release. Then instead of being called MAC it should be called Client Authentication Code.

Signing each transaction to the server with a digital signature is the other possibility. The function for digital signing and the hash function for MACs should both be public. The security of the scheme then relies on the security of the key, and then it has the same problem with consideration to key handling.

The problem is, that re-engineering is relative easy, if the key is kept in the same position in each release of the client. One thing would then be, to change the position of the key, or better, divide the key and store it at multiple positions.

To make the re-engineering problem harder, one could inject the security mechanisms into existing classes/objects of the client program, to avoid that the information is hidden in an Easter egg[NM03], then using the existing program control flow to produce the key information. Each puzzle piece is a function call and calls to all functions are necessary to complete the puzzle. The key is never to be assembled in memory. Two schemes are explained in the following two sections that use the idea of splitting keys for either digital signatures or Client Authentication Code (CAC)s. One more scheme is explained in the third section, which use checksum computation of the program for both integrity and client release identity.

Digital Signing

As stated prior, the private key should never be allowed to be in one class/object. An idea is to use secret sharing do split the key into pieces, afterward hiding each piece of the secret key in existing code. The public key is (n,e) and the encryption is $f(x) = x^e \, mod n$, then using the exponential math rules $(x^s)^t = x^{s*t}$ and $x^s x^t = x^{s+t}$. The exponent e is split in multiple calculations to get $f(x)$.

Addition Split of Exponent *equation 1.1* $x^e \, mod n = x^{e_1} + x^{e_2} + x^{e_i} \, mod n, where e = e_1 + e_2 + e_i$

Multiplication Split of Exponent *equation 1.2* $x^e mod n = ((x^{e_1})^{e_2})^{e_i} mod n, where e = e_1 * e_2 * e_i$

This is computational harder than performing the Addition Split of exponents.

Exponent Disruption The exponent e value can also be split into one or more of the following arithmetic operations $e = t + v$,$e = t - v$,$e = t * v$,$e = t/v$, and $e = t \, mod \, v$.

Where $i < 0$ and s values have to be more than 1 for a change to happen, because as the equation state $s * 1 = s$ nothing happens and 0 is an illegal value of s.

Required Functions This now gives two different groups of functions:

1. Encryption functions
2. Compute exponent functions

One could make decoy functions, which make it harder to the re-engineer, however decoy functions has to be called. So the attacker cannot use a dead code removal tool to remove decoy functions. How these encryption and decoy functions can appear in Java is seen in Appendix A.

CAC

Here the idea is that the key is a block of pseudo-random generated bytes, that following array of bytes is split up into the wanted number of key blocks, each block is placed in the program. The CAC is computed on the input data that are sent from the client to the server. The HMAC algorithm seen in Algorithm 1 is an obvious choice for computing CACs, because it is designed for the purpose of computing MACs using keys.

Algorithm 1 HMAC algorithm from [Sti06].

$HMAC_{Key}(x)$
$ipad = 0x360x36 \cdots 0x36$
$opad = 0x5C0x5C \cdots 0x5C$
return $HashFunction((Key \oplus opad) \| SHA - 1((Key \oplus ipad) \| x))$

The hash function can be any secure one-way hash function. SHA-1 is a good candidate of such a one-way hash function, because current it is a very used hash function.

2.4.2 Checksum

Compared to the other two solutions the checksum solution have one advantage, which is the checksum is calculated from the program, and if changes are made to the program the checksum will reveal it. In the other two solutions altering the program is not detected. However, there is no guarantee that the no changes are made to the checksum function within the program, a solution on how to implement a checksum function is presented in [SLS+05] and its name is Pioneer. In the Pioneer solution the checksum is calculated such that the checksum function cannot be altered without detection. The problem with Pioneer is that it is actual a rootkit, users do not like having rootkits installed, this was observed with the Sony's copyright solution that relied on Sony installing a rootkit, which

was used for copyright protection[Woe05]. Because of the argumentation following in the section Security, further analysis of [SLS+05] is done in Chapter 3

2.4.3 Security

The problem with digital signatures and CAC is that even though one handles the key carefully, which is splitting up the key and placing key parts in different positions. It would still be possible to decompile the client, the security then relies on the security through obscurity.

The attacker can decompile the client and then add malicious code where she wants and re-compile the client, which results in a client that correctly signs or compute CACs and invokes the attacker's malicious code as specified. This attack is not detectable with the digital signatures and CAC tamper-proofing security mechanisms, it would require additional tamper-proofing mechanisms to prevent the described attack. The attack has two different applications, the first application is for the attacker to replace the original client with the re-engineered client, to trick users into submit personal data with the re-engineered client that then could leak the personal data. The second application is for the attacker to force illegal input to the server.

The checksum scheme is very different from the digital signature and CAC schemes. The computing the checksum from the program and the checksum will deviate if changes have been made to the program. This property is not in the digital signature or CAC schemes. The problem is that the client can compute the hash once, and then each time the server request the checksum the client can send the precomputed checksum.

The use of the checksum method is the only way to ensure client identity and tamper resistance if designed correctly. An example of such a design is the Pioneer solution described in [SLS+05]. The tamper-proofing security mechanism should then be able to obtain security, such that attackers both external and insiders cannot change the program without detection. The goal is then to design it such that no circumvention of the tamper-proofing mechanism is possible.

2.5 Summary

The security mechanisms suggested for client security are; security through obscurity, secure channel, user authentication, tamper-proofing and client validation. From the descriptions of the security mechanisms in the analysis, the following can be concluded about the security they offer. The user authentication security mechanism makes phishing attacks hard for any attacker and only allows users of the system in. The SRP protocol allow only dictionary and brute-force attacks on user-names and passwords to be done on-line. The secure channel security mechanism makes it hard to attack the network. Security cannot be argued for the tamper-proofing and client validation security mechanism, because the design of the security mechanism controls the security. However,

if the design of the tamper-proofing and client validation only rely on security through obscurity, the client program will only be secure against off-line re-engineering attacks for small periods of time.

The security mechanisms security through obscurity, secure channel and user authentication are more or less trivial to implement, because solutions already exists that do this. The security mechanism tamper-proofing and client validation is selected for design, implementation and test, because the solution that exists uses very low-level programming to ensure trust in the client. The tamper-proofing and client validation security mechanism is to be based on checksums as used the related work [SLS+05] and [SPvDK04], which is described further in Chapter 3. In the rest of the report, when one read tamper-proofing security mechanism, one should understand it as being the tamper-proofing and client validation security mechanism.

Chapter 3

Related Work

The related works are solutions that bootstrap trust using software and no hardware. The bootstrapping of trust is how to initiate trust on an un-trusted machine. The two solutions Pioneer[SLS+05] and SoftWare-based ATTestation (SWATT)[SPvDK04] use the same method to bootstrap trust. The method is explained using the Pioneer solution as an example. The summary is a description of, what to use from these two solutions in the design of a new tamper-proofing security mechanism.

3.1 Pioneer

The method used in Pioneer to bootstrap trust is where the client is an un-trusted system that uses a trusted server system to attest the software running on the client. The following terminology is defined, the trusted server is called a verifier and the un-trusted client is called a prover.

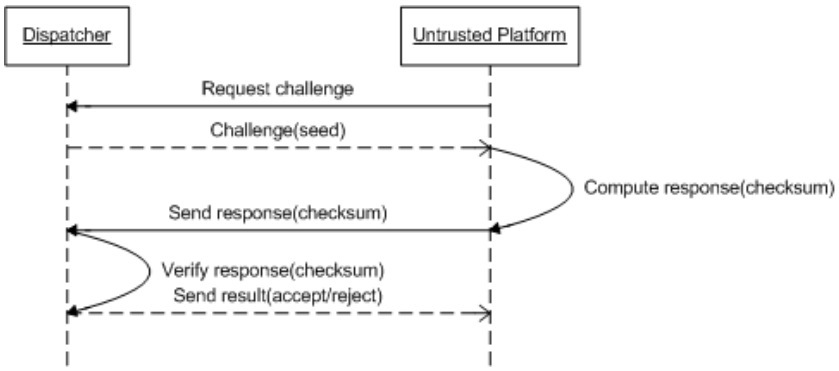

Figure 3.1: This shows challenge/response protocol that Pioneer and SWATT uses.

The data flow between the verifier and the prover can be seen on Figure 3.1. The prover request a challenge from the verifier, the verifier then respond with a challenge. The prover use the challenge to compute the checksum of the verification function running and

the software to attest, the checksum is then sent to the verifier. When the verifier receives the checksum, it checks that it can compute the same checksum using a copy of the verification function and the software to attest. If then the verifier accepts the checksum, the verification function on the client can stop and start the attested software. The attested software then runs un-interrupted, because while it is running system interrupts are disabled. The above data flow between the verifier and the prover ensure that a valid checksum can only be returned with negligible probability, before the prover has obtained the challenge from the verifier.

This challenge/response protocol does not itself bootstrap trust to the verification function or the software to attest. To obtain trust more measures are necessary, which bootstrap trust in the verification function. The idea is to optimize the verification function to smallest possible running-time so no faster verification functions exist. By knowing the computer hardware specification and most importantly the clock cycle of the Central Processing Unit (CPU), and that the verification function is implemented in assembler. Make it possible calculate estimated running-time for the program instructions. If the prover execute the verification function and send the computed checksum to the verifier within the estimated running-time, the verification function can be trusted and used to attest the software. This is because any changes to the verification function would increase the running-time.

The challenge/response protocol combined with Pseudorandom Memory Traversal (PRMT) forces the client to request a challenge from the verifier, before starting the computation of the checksum, making it hard to do precomputation. If precomputations where possible a cheating prover could return a valid checksum within estimated running-time and resulting in a failure of the bootstrapping of trust.

The challenge is a seed for the PRMT function, which actual uses the seed to seed a Pseudorandom Bit Generator (PRBG). The PRBG generate on the fly a memory address to one data block of all the data blocks storing the verification function and the software to attest. The data block is then input to a nested hash function, this process is then repeated until a hash has been computed on all data blocks. When a hash has been computed of the last data block, the outputted hash value is the checksum responded to the verifier. However, because the memory addresses for the data blocks are generated using a PRBG, there is problem, which is to ensure that all data blocks are touched at-least once. Coupon Collector's Problem solve this problem of touching all data blocks with high probability, by stepping through $n * ln(n)$ number of data blocks, where n is the number of blocks used for storing the verification function and the software to attest.

The verification function has to compute different checksums when the order of the data blocks are different, requiring a strongly ordered nested hash function. This is done by changing between the two Boolean operators AND and XOR, where the new hash function result are merged with the result of previously merged hash function results.

The Pioneer solution makes use of three things to bootstrap trust, which are a challenge/response protocol, timeout and knowledge of hardware platform.

3.2 Summary

The method of bootstrapping trust used in Pioneer and SWATT, require the knowledge of the hardware specification of the client computer or embedded device. This is not valid assumption in the tamper-proofing security mechanism to design in this project, because the client/user cannot be relied upon not to lie about her hardware specification. The assumption is valid in Pioneer and SWATT, because the client is the one who gains from providing a correct hardware specification.

The idea of using a challenge/response protocol to make precomputations hard is a good property to adopt into the design of the tamper-proofing security mechanism. The reason is that it does not build upon an assumption wrong for this project.

Chapter 4

Design

This is the proposal to a design of a tamper-proofing security mechanism that bootstrap trust to client software.

The tamper-proofing security mechanism is to use a checksum for client software verification, because the key hiding tamper-proofing schemes in Section 2.3.3 also requires a checksum to ensure client software integrity. The checksum can be used for both client software identity and integrity. The related work uses checksum functionality to make precomputations hard.

The design of the tamper-proofing security mechanism has similarities with the related work described in Chapter 3. The similarity is the problem area and the difference is the environment in which they work, this solution has to run on a Java Virtual Machine (JVM) and the Pioneer and SWATT are programmed in assembler. The JVM is a requirement because the client should be able to input data using Java applets.

4.1 The Tamper-Proofing Prototype

This section describes the prototype and what requirements there are to the prototype besides being developed in Java.

The challenge/response protocol described in Chapter 3 is in the prototype design so precomputations are hard

The prototype is to use the challenge/response protocol described in Chapter 3 for making the task of doing precomputations hard. The challenge/response protocol uses the client/server topology. This topology is a widely used architecture that enables other applets to embed the tamper-proofing security mechanism without additions to the topology.

The selected topology requires the prototype to consist a client and a server with the following security mechanism interaction:

1. Client:
   The client is an applet that contains the tamper-proofing security mechanism, which receives a challenge and computes a checksum that it sends to the server.

2. Server:
   The server program is to service the tamper-proofing security mechanism in the client applet, by sending a challenge and receive a checksum that it verifies.

The tamper-proofing security mechanism should be such that no obvious attack patterns are present for copy and replay attacks. The copy attack is where the invalid client stores a copy of the valid client in memory for checksum computations, then it can answer correctly to validation requests, even through it is actual an invalid client program running on the client machine. The replay attack is where the attacker can replay a previous validation, and then convince the server that she is actual running a valid client. If the below listed properties cannot be broken with re-engineering, a successful re-engineering attack is classified as hard to do:

1. Re-engineering attempts has to be restarted with each release of the client.

2. The tamper-proofing security mechanism cannot be circumvented within the stated time-frame.

It is not possible to bootstrap trust using the same method from Chapter 3, because of the requirement of implementing the client in Java. Another argument is that the server cannot be sure of the hardware specification of the client. Because no other solution implemented in software is known to bootstrap trust. Trust is assumed if the two listed re-engineering requirements are enforced and if no obvious copy or replay attacks exist. The challenge/response protocol, the byte array operations and the client validation checksum are measures to put trust into the client. The byte array operations are byte operations on the memory storing the client program. The goal of adding byte array operations to the client program is, then if the program flow change from the intended program flow, the checksum computed on the client program would be invalid. This should ensure that the program runs as intended. This should give the client program the initial trust, but this checksum cannot be send with the input data also, this would leave the server open to accept input from any one who knows a just computed checksum. It will then require the input submission to use another mechanism to authenticate that input comes from a previous validated client. The security mechanism for this is going to be the CAC scheme analyzed in Section 2.4, even through it is known that the security would rely strictly on a security through obscurity problem. Right now its assumed that the time-frame for the initial client validation to be 20 seconds and the time-frame for accepting input data is 10 minutes, these time-frames is thought of as the maximum time-frames where it should be possible for a legal user to do the scenario of authenticating and inputting data successfully.

The challenge/response protocol for the prototype can be seen in Figure 4.1. This is the modified version of the protocol described in Chapter 3, which are to make it hard to do precomputations before the client receives the challenge.

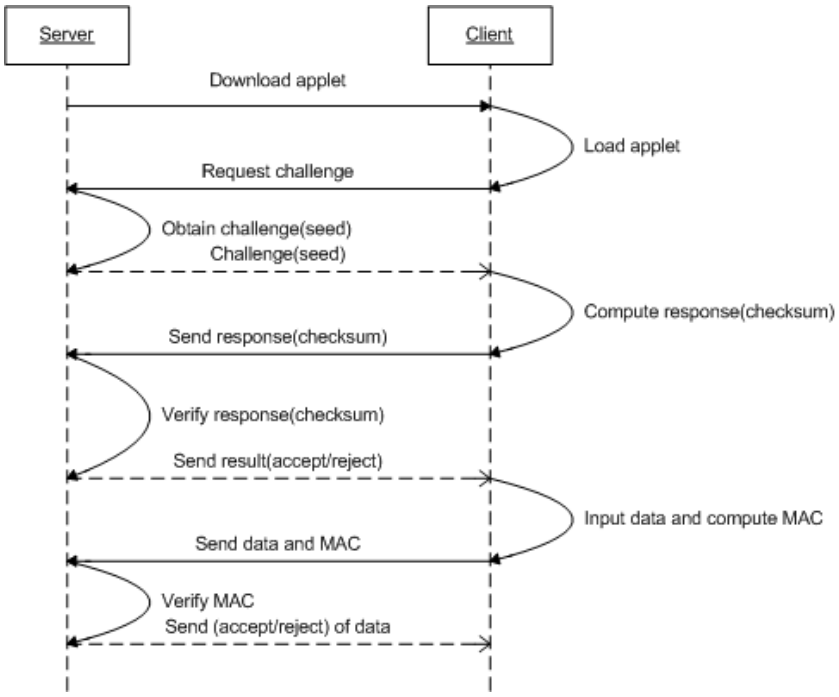

Figure 4.1: This shows the challenge/response protocol to use in the prototype.

The compilation and deployment of the client is shown in Figure 4.2A. The merge of tamper-proofing and client code, the adding of byte array operations to the merged code is done once per user. The compilation of the tamper-proofing and client code is done per user in the following order, first compile it with the `javac` compiler next obfuscate it with `ProGuard`, now all the client programs one per user are ready for deployment on the web-server. The process of obtaining and running the client is shown in Figure 4.2B. The client computer downloads the client from the server and starts the program just after having downloaded it, this is necessary because the re-engineer can start when she has received the client program. The client is then run until it is terminated, what happen while it runs is shown on Figure 4.3.

The obfuscation of the client is done to reduce readability of the client program. When the tamper-proofing mechanism is embedded into the client applet and have been compiled once, it should be re-compiled using an obfuscator. In this project the obfuscator to use is ProGuard.

4.2 Checksum Computation

The design of client software validation is described here. The client software validation process is the computation of a checksum a server is to verify. The PRMT generate a pseudo-random ordered data block of the verification function and the client software. The checksum is then the result from an HMAC computation with the seed as key and pseudo-random ordered data block as input. The one-way hash function to use in HMAC

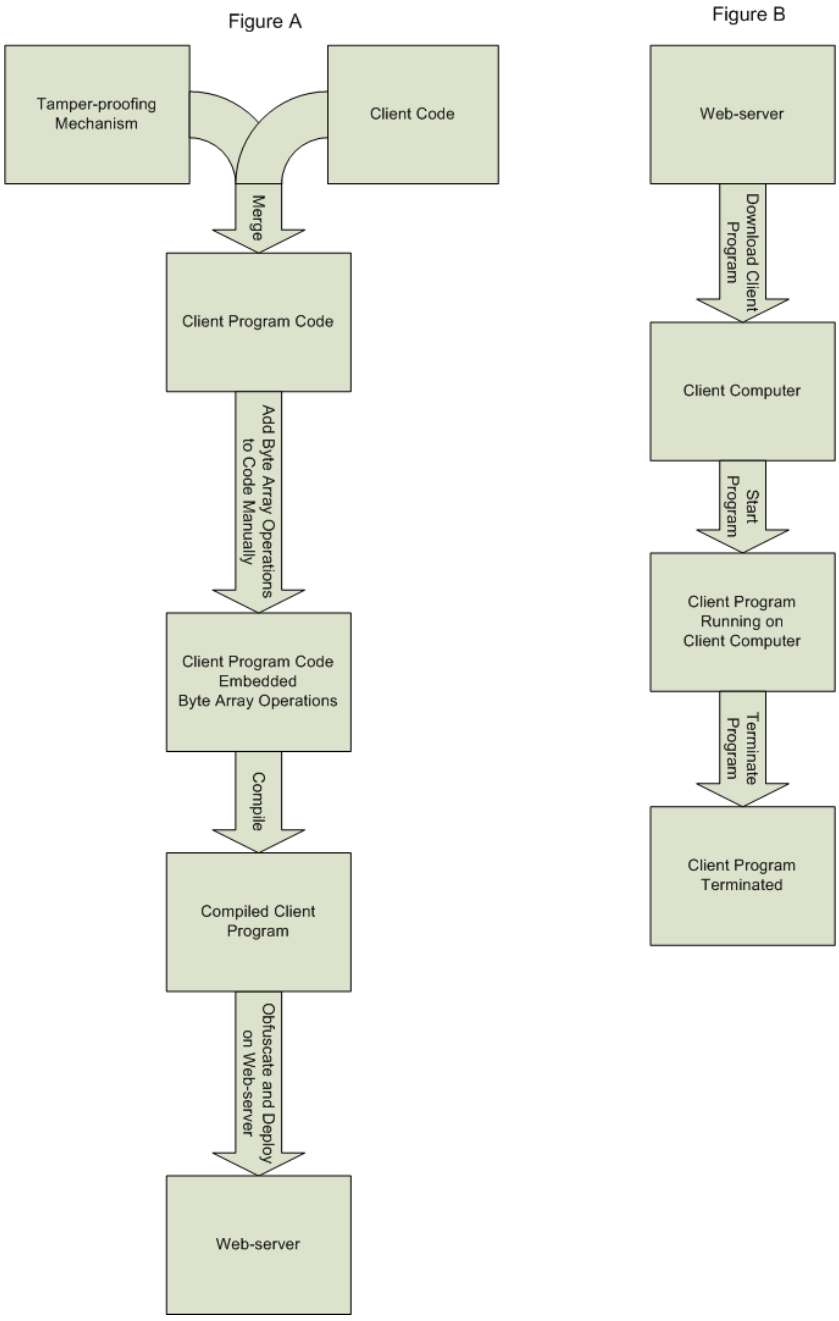

Figure 4.2: This figure shows the compilation and loading of the client.

is in this project SHA-1, however, the one-way hash function have to be easy to replace with another one-way hash function, when the security the one-way hash function offer is not sufficient.

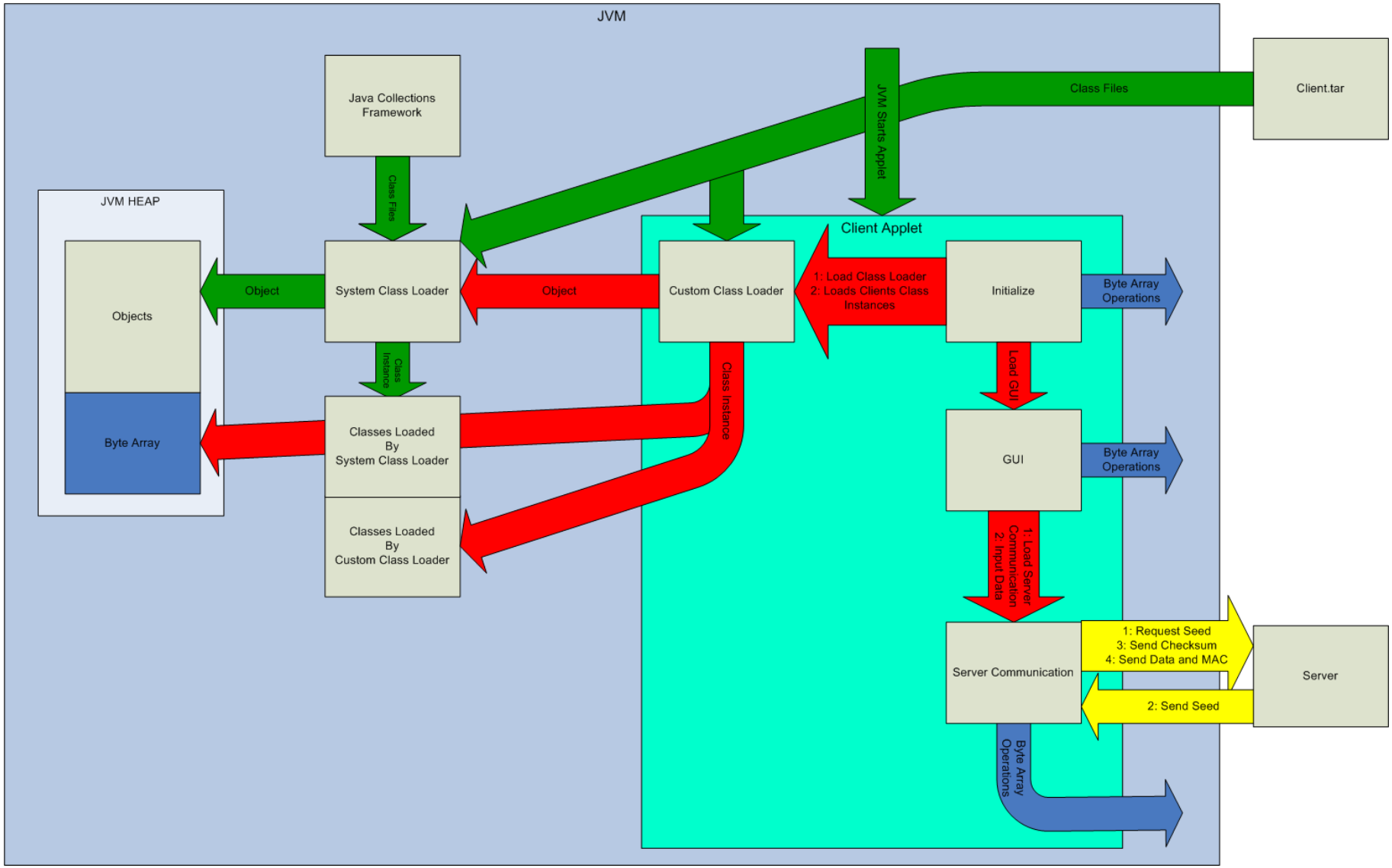

Figure 4.3: A richpicture of what happen when the client is running. The green arrows are external to the client applet, which are invocations by the JVM or external input to the client applet. The red arrows are invocation by the client applet. The blue arrows show where the byte array operations can be invoked, to change the loaded classes byte array present on the JVM heap, represented with the same blue color. The yellow errors show the communication between the client applet and the server. The arrows with numbered actions is where the invocation order of the action is important, the invocation have to be in increasing order.

4.2.1 Hash Function

The hash function to use in HMAC is SHA-1. There exist attacks against SHA-1, however, it is not relevant when SHA-1 is used in HMAC as is stated in [Bel05].

The one-way hash function can be changed with each client release, if this is enforced, it would require analysis by the re-engineer to discover the hash function being used.

The HMAC algorithm can be seen in Algorithm 2, the algorithm requires the following input data(x) and a key. The HMAC algorithm allow changing the value of *ipad* and *opad*. The key to use is the challenge obtained from the server, which the PRMT function uses as seed. The server is to use the class `java.security.SecureRandom` for generating pseudo-random challenges. The input data is the pseudo-random ordered data block generated using PRMT.

The SHA-1 implementation to use is the one provided by `java.security.MessageDigest`.

Algorithm 2 HMAC algorithm from [Sti06].

$HMAC_{Key}(x)$
$ipad = 0x360x36 \cdots 0x36$
$opad = 0x5C0x5C \cdots 0x5C$
return $SHA-1((Key \oplus opad) \| SHA-1((Key \oplus ipad) \| x))$

4.2.2 Pseudorandom Memory Traversal

PRMT is an idea from Pioneer[SLS+05] and SWATT[SPvDK04] explained in Chapter 3. The idea is to use a challenge/response protocol to limit possible precomputations of the checksum. Using the challenge to seed the PRMT algorithm that then selects data for the pseudo-random ordered data block. In Figure 4.4 is seen how, the client program is split into blocks and added to one pseudo-random ordered data block, through three steps. The checksum computations on the pseudo-random ordered data block then influence the checksum, which is the response in the challenge/response protocol. Precomputations of the checksum are limited, because it requires the seed to know the order of the data block for the checksum computation.

In Pioneer[SLS+05] and SWATT[SPvDK04] the order in which to read data blocks are decided by a PRBG, it generates the memory addresses that points to data blocks to read, and the order is dependent of the seeding of the PRBG. The challenge/response protocol and PRMT ensures that possible precomputations of the checksum are limited.

In this project the memory addresses that point to data blocks are replaced with indexes to a byte array. On the client the PRBG to use is `java.security.SecureRandom` and challenge obtained from the server is the seed. The seed to request from the server consist of 20 bytes, which is the same length as the checksum returned from the SHA-1 hash function. The client uses the same seed as the key for the HMAC algorithm. The reason to use the seed for both is, that it do not reduce security with consideration to

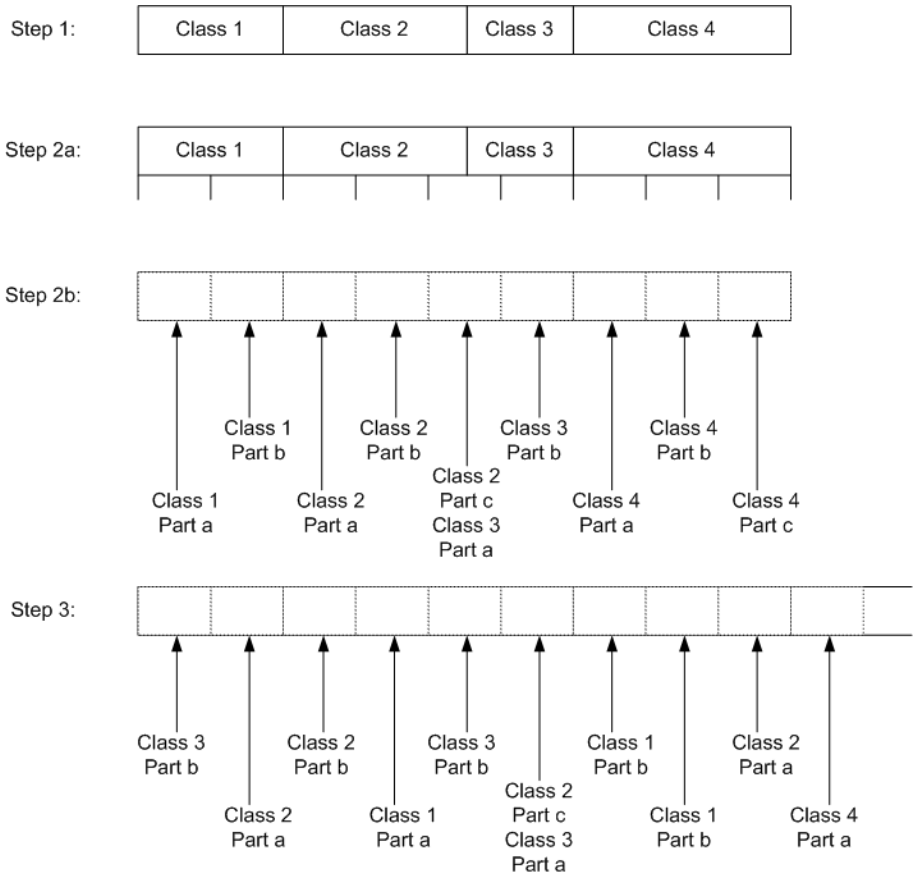

Figure 4.4: This figure describes how the byte array containing the static program data, is split into pieces of the same size and are pseudo-random ordered.

the limited precomputation property, but do reduce running-time, because less time is spend on sending one seed instead of two.

4.3 Byte Array Operations

The idea of the tamper-proofing security mechanism is that the byte array of loaded classes can be subject to legal and illegal operations. If the program is initialized as intended, the byte array is subject to only legal operations, and then the checksum of the byte array is valid. The byte array is subject to illegal operations when the program flow is altered. The re-engineer cannot tell the difference between legal and illegal operations unless she knows the legal program flow, which is the program flow of intended client use.

The operations on the byte array are on one or more bytes. There are two types of operations, type one seen on Figure 4.5 and type two seen on Figure 4.6.

There are three phases when using operations on the byte array for checksum verification, which are the three descriptions below:

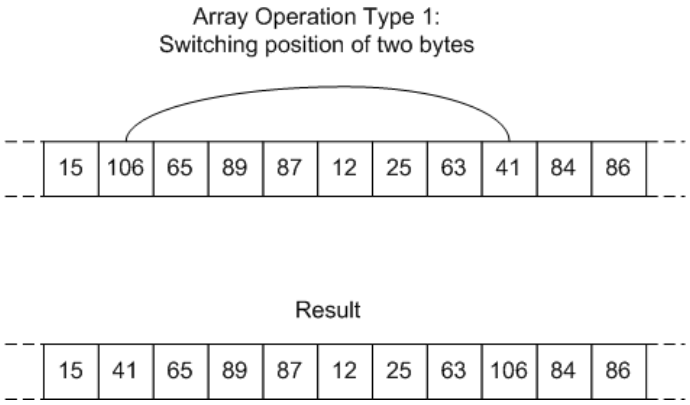

Figure 4.5: The operation type one is a swapping of byte values. The byte value at index 2 is swapped with the byte value at index 9.

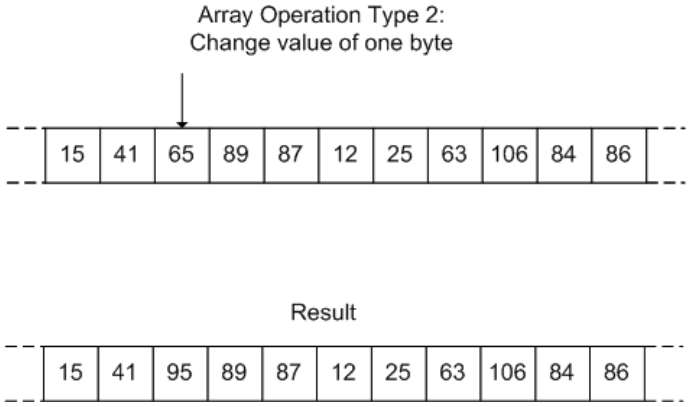

Figure 4.6: The operation type two is an insertion of a byte value. The byte value at index 3 is changed from 65 to 95.

Generating operations: It is important how the operations input are generated, if the input can be anticipated the effect is limited. However, when each input is generated with a pseudo-random source, the input is then hard for an attacker to find. An operation is defined to be when one byte is altered to a new value. The operations are generated using a PRBG for generating the indexes and new byte values. The generation of indexes shall not be larger than the size of the client program. Generating the operations with the use of PRBG makes it possible that the same index is generated more than once. The result is that the operations are strongly ordered, which make alterations of the program flow influence the outputted checksum, even if only legal operation was done. The source of pseudo-random bits will be the Java class `java.security.SecureRandom`.

Placing byte array operations in code: Operations can be placed anywhere in client program source code, where it is legal to insert a statement. The important thing is to know, which operations to use for the valid checksum. It is the placement of the operations that divides them into the two groups of legal and illegal operations. The illegal operations of the byte array have the purpose of confusing the re-engineer, which should be tricked to believe that an illegal operation is actual

a legal operation. The re-engineer would then do the illegal operation on the byte array before computing the checksum. If illegal operations are done on the byte array, the result should be an invalid checksum. In this project the placement of operations is done manually, though an automatic placement program would be relevant for production use.

Verify byte array subjected to byte array operations: It is important the server knows all legal operations, and the order they are performed on the client. If this is not the case, the server cannot verify the checksum it receives from the client, to be valid.

The verification function and the software to attest are embedded with the operations, and the operations perform byte operations on the byte array storing loaded client program, which makes up the verification function and the software to attest. The operations embedded into the client program are to make sure the client program runs as intended. The verification function and the software to attest contain both legal an illegal operations, it is then not enough for the re-engineer to find all operations, this requires her to know which operations are legal and which are illegal. A successful re-engineering attack of the tamper-proofing security mechanism is that it is possible to find all legal byte array operations.

4.4 CAC on Input Data

The byte array operations are to validate the client at initialization, but the tamper-proofing security mechanism requires a check with each input, to authenticate that the input is send from a valid client. This security mechanism design will be of the CAC scheme described in Section 2.4.

The CAC scheme uses the HMAC algorithm with a key and data. The key is static data to place at different positions in the code, and the data are the input data the user enter into the client. The CAC is then sent with the input data, where the server then verify the CAC knowing the key and the input data.

The three phases with regards to byte array operations are also necessary when designing the CAC scheme.

Generating key data: The generation of the key data is important, because the attacker should not be able to guess key data. Using `java.security.SecureRandom` for generating the key data will make it hard for the attacker to guess key data. The key can have a maximum length of 512 bits(64 bytes). The CAC scheme uses multiple keys, and the order in which the CAC is computed influence the output.

Placing key data: The key parts can be located anywhere in the client source code, which the client send input data function can access. This can be field values or methods, if it uses methods these can then call other methods that contain parts of

a key part. This can make a very large and complex data structure to re-engineer, but then running-time of the client is affected. The complexity of the structure for storing the key parts, is set by an acceptable ratio between complexity and running-time.

Verify the CAC: Any complex structure for storing key parts the server has to model, for it to be possible for the server to verify the CAC.

4.5 Design for Java

The prototype targets the Java programming environment, which enables the client program being a Java Applet that is easy to deploy. This imposes Java sandboxing used by browsers to run applets. The other problem is the JVM do not allow making dumps of memory or access memory direct, which the checksum is computed from. The solution to the memory access problem is a custom class loader, which is a subclass of the original bootstrap class loader. The browsers running the applet over network do not allow the creation of a custom class loader. The bootstrap class loader is only allowed to run bytecode placed within the classpath. This allow applets within the classpath and viewed with the Applet Viewer released by Sun[Gon02] to load a custom class loader.

The idea is then to implement a client that can run in the Applet Viewer, which allow the creation of a custom class loader as long it is in the classpath. The argument is that the functionality required of the class loader does not reduce the security of the JVM and the browser, the required functionality could then be added to the existing class loaders that browsers use. The only problem about adding the extra functionality to the current applet class loader is that it uses double memory for each class loaded. This is not a good property, but it is either this or a re-design of Java to allow direct memory access. And even through it is not a good property, it do not allow for the applet sandboxing to be broken, which terminate the applet in case a memory bound is tried broken.

The creation of the custom class loader and how the loaded classes are stored in memory is done with regards to make copy attacks hard.

Custom Class Loader

The class loader's call chain is added the custom class loader, then when the bootstrap class loader cannot find a specific class, it then delegate the loading of the specific class to the custom class loader. To force that loading of all client classes is done by the custom class loader. Explicit calls to the overloaded method `findClass(String name)` is performed. The method is such that while loading class information from the class files, it stores a copy of the class information for the checksum computation. The design of the custom class loader is such that even if an attacker replace or alter the custom class loader with a malicious class loader, the attacker is not able to cheat the checksum computation, because the requirement of the custom class loader is it outputs a byte

array of the classes loaded into the JVM. Then if any new malicious class is loaded it can only be access from classes where the checksum is computed, it would resolve in an invalid checksum.

Storing Classes

For storing the loaded classes a singleton class is selected that contain a byte array. A class is added to the byte array when the `CustomClassLoader` loads the class in question. The idea for using a singleton class is to make it easy to do byte array operations on one or more bytes in the array, because the singleton class is loaded with the first call to the method `getInstance()`. The same method enables classes to easy get the reference to the object containing the array. The singleton class is named `MemStorage`.

Design of the how the array is handled should be with consideration to the copy memory attacks described in [SLS+05]. This document [LY99] show how the classes are loaded in the JVM, and this document [Tra] shows how the class loader is overloaded.

4.6 Prototype Class Design

The client design has the following components `Applet`, `MemoryManagement`, `Checksum`, and `Communication` under which is a description of each of the necessary classes:

Applet

The design of the applet consist of two components the `LoadApplet` class with a description below, and the GUI to input data to the server.

LoadApplet: In Figure 4.8 is seen the class to load the GUI. It creates an instance of the `CustomClassLoader` class and uses that to load the GUI classes. The `LoadApplet` class then creates an instance of the GUI and starts it. This class is an obvious point of attack, because this class is not loaded by the `CustomClassLoader`. However, the class `LoadApplet` could be removed if the `CustomClassLoader` functionality was added to the current applet class loader. The reason that this class can be removed from the prototype is that its function is to create the `CustomClassLoader`.

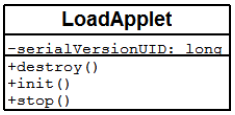

Figure 4.7: LoadApplet class diagram.

MemoryManagement

The memory management component is what loads and stores classes in the byte array, it is why these two classes are the exception of classes to store in the byte array. Meaning the `MemStorage` and `CustomClassLoader` cannot be in the byte array to use for the checksum computation. The result is that changes to these two classes will go un-detected.

CustomClassLoader: In Figure 4.8 is seen the class to use for loading classes into the JVM while storing them in the `MemStorage` class. The `CustomClassLoader` should be constructed with a reference to the parent class loader, and a string that contains the name of the client archive file. The private field value `jarfile` is for storing the name of the client jar file. The private field integer offset is the current offset for where to insert the next loaded class into the `MemStorage` byte array. The protected method `findClass(String name)` is the overloading method of the corresponding class `java.system.ClassLoader` . The method `findClass(Sting name)` use the protected method `findClassBytes(String name)` to load the class file to the name of that class.

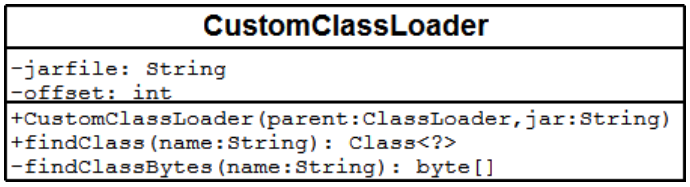

Figure 4.8: CustomClassLoader class diagram.

MemStorage: In Figure 4.9 is seen the singleton class for storing the byte array of loaded classes. The singleton class requires the private constructor, the private object reference, and the `getInstance()` method. The important thing about this object is that the byte array for storing the loaded classes is public. The reason is that byte array operations are done directly on the byte array. It is also easy to get the object reference using the `getInstance()` method.

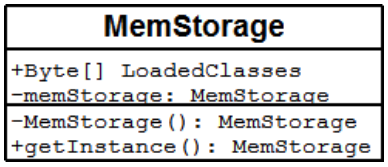

Figure 4.9: MemStorage class diagram.

Checksum

These are the class designs to use for the checksum computation and server verification.

Verifier: In Figure 4.10 is seen the class to place on the server, to use for verifying the checksum the client computes. When the `Verifier` is instantiated a seed is generated and it computes a corresponding reference checksum. The `verify` method then check its parameter checksum against the reference checksum, then the `Verifier` changes state if the two checksums match. The fields that represent the state of the `Verifier` instance are `dataReceived`, `softwareValid` and `time`. The `softwareValid` Boolean is set to true if the client has sent a matching checksum to the server. The `dataReceived` Boolean stores the status whether the client has sent data to the server, if true data has been received else false. The `time` stores the time from when the `Verifier` instance was created, the instantiation of `Verifier` is done when a client request a seed. The `time` field is for controlling the expiration of the `Verifier` instance.

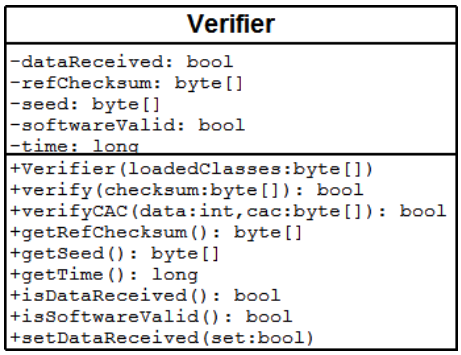

Figure 4.10: Verifier class diagram.

Hashing: In Figure 4.11 is seen the class to place on both the server and the client, and its use is for computing checksums. The only methods for users of the class to remember is the public methods `computeChecksum` and `computeHMAC`, the `computeChecksum` method use the three other methods for the client validation checksum computation. The method `computeHMAC` computes the HMAC using the one-way hash function in the method `computeHash`. This only requires the `computeHash` method to be reprogrammed if another one-way hash function should be necessary. The `computeHMAC` method calls the `computePRMTBlock` method to get a pseudo-random ordered byte array data blocks. To make the order of the blocks pseudo-random, it uses the `java.security.SecureRandom` PRBG. The CAC computation uses the `computeHMAC` method.

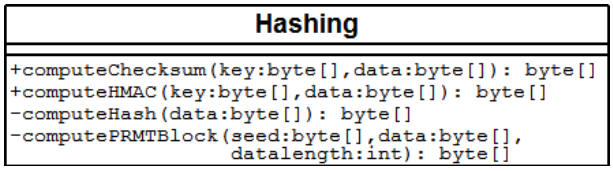

Figure 4.11: Hashing class diagram.

Communication

For communication between client and server the challenge/response protocol design uses Remote Method Invocation (RMI).

Client: In Figure 4.12 is seen the class that provides the client applet with an Application Programming Interface (API) to the challenge/response protocol. The software design of the tamper-proofing mechanism uses these methods as API for communicating with the server.

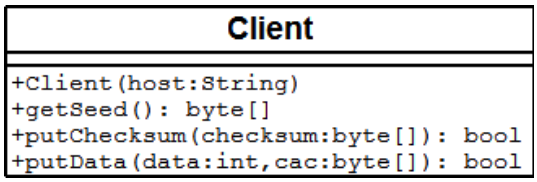

Figure 4.12: Client class diagram.

Server: In Figure 4.13 is seen the class design for the server, which should provide a service to the clients. The server is to answer seed requests, receive checksums and receive data from clients. The server class creates an instance of the `Verifier` for each seed request. The server also has to store a valid copy of the client program and the byte array operations done on the client program, which then makes it possible to compute a reference checksum for matching with checksums from clients. The server also has the list of CAC key parts to validate that input data originates from a valid client.

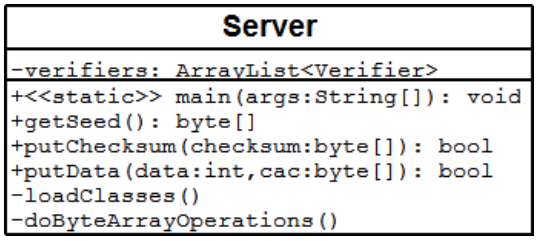

Figure 4.13: Server class diagram.

ChallengeResponse: In Figure 4.14 is seen the interface that list of the RMI communication methods.

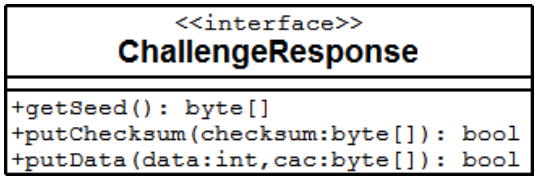

Figure 4.14: ChallengeResponse interface class diagram.

4.7 Summary

The design proposed for the tamper-proofing security mechanism uses a challenge/response protocol between the client and server. The challenge is a seed that the server sends to the client, which then computes the response checksum that it sends to the server. The checksum computation uses the HMAC algorithm. The HMAC algorithm use PRMT to generate input data which consist of pseudo-random ordered data blocks. The HMAC algorithm is keyed with the challenge and uses the SHA-1 hash function.

The input is an array of pseudo-random ordered blocks of the verification function and the software to attest. However, because the design of Java do not allow accessing memory directly, it is necessary to overload the class loader with a custom made class loader. The custom class loader then stores the classes it loads, in a byte array, while the actual class is loaded into the JVM. Byte array operations are done on the stored classes and these operations are embedded in the classes' methods to ensure, that loaded classes actually run. The use of byte array operations are to bootstrap trust into the verification function and software to attest, because it is assumed the re-engineer cannot find the set of legal byte array operations within the given time-frame.

The checksum verification is done before data is send from the client to the server. To make sure that the data originate from valid client software, the server verifies the CAC computed using the input data and is keyed with static data stored in the client program. The re-engineer is then forced to find all legal byte array operations and the static key data for the CAC computation.

Chapter 5

Implementation

This chapter documents the implementation of the previous described design. The documentation also contains what implementation problems have been encountered, the mechanism used, and how the final program is structured.

The client implemented is an input form that validates against the server before allowing the user to input data. The prototype uses an integer value to symbolize input data, the client compute a CAC on the input data and the key parts are stored in the client program. The CAC then accompanies the input data to the server, where the server then validates the CAC before accepting the input data.

The tamper-proofing security mechanism consists of two actual security mechanisms, which are the byte arrays operations with initial client validation check and the CAC scheme for checking that a valid client was the origin of the input data.

5.1 Pseudorandom Bit Generator

It was the plan to use `java.security.SecureRandom` for PRBG where pseudo-random bits was necessary. A problem occurred while implementing the verification of the initial checksum, because the checksum computed on the client and the server have to be the, same and was not. The Cryptographic Secure Pseudorandom Bit Generator (CSPRBG) used, has to output the same pseudo-random bits using the same seed, which the `SecureRandom` class did not. Because of this problem, a new CSPRBG was implemented. The new implemented CSPRBG class is also called `CSPRBG`, the constructor take the seed as an argument. The method used for generating a pseudo-random byte is shown in Listing 5.1. It uses the SHA-1 hash function with the seed as input, the least significant byte of the hash, is the pseudo-random byte returned by the method. The computed hash is then assigned to be the seed, it then uses as input with the next call of the method.

```
1 public byte generateByte()
2 {
```

```
3     md.update(seed);
4     seed = md.digest();
5     md.reset();
6     return seed[0];
7   }
```

Listing 5.1: The method `generateByte`.

5.2 Byte Array Operations

The two types of byte array operations added to the client code can be seen in Listing 5.2 and Listing 5.3. It is only the values and indexes that changes, and the location in the client controls whether it is an illegal byte array operation or legal.

```
1 ...
2 MemStorage.getInstance().loadedClasses[5436] = -85;
3 ...
```

Listing 5.2: The type one byte array operation as seen in the source code.

```
1 ...
2 byte b = MemStorage.getInstance().loadedClasses[3436];
3 MemStorage.getInstance().loadedClasses[3436] = MemStorage.
     getInstance().loadedClasses[6783];
4 MemStorage.getInstance().loadedClasses[6783] = b;
5 ...
```

Listing 5.3: The type two byte array operation as seen in the source code.

The client code is added the following legal and illegal byte array operations and the location is described with class names.

Legal Byte Array Operations

```
Type     Index        Index/Value
ByteArray.java:
0        3353         58
0        7013         -26
InputForm.java:
1        3436         6783
0        5436         -85
0        387      100
Client.java:
0        4814         -59
Hashing.java:
0        142      -29
0        5055         119
CSPRBG.java
0        4980         120
```

Illegal Byte Array Operations

```
InputForm.java:
0       3354        78
Hashing.java:
1       4452        7529
```

In total there are 11 byte array operations and of those 2 are illegal.

5.3 Client Authentication Code

After validating the client to be authentic, the server use the CAC scheme to verify that the input data, that clients sends, comes from valid clients. The CAC is an HMAC computed with a key stored in the client. How the key is stored in the client code can be seen in these two examples Listing 5.4 and Listing 5.5. The server validates the CAC by reading the CAC key from a file with the name `cackey.txt`.

```
1 ...
2 private byte[] b = {52};
3 ...
4 public byte[] getB() {
5   return b;
6 }
7 ...
```

Listing 5.4: An example snippet where the CAC key part is stored in a field variable.

```
1 private byte[] value() {
2   byte[] g = {69};
3   return g;
4 }
```

Listing 5.5: An example snippet where the CAC key part is retrieved with a method call.

5.4 The Implementation of Classes

This section describes the classes as they have been implemented. These new classes have been added since the design; `ByteArray`, `InputForm`, `CSPRBG`, and a server `MemStorage`. The reason to add these classes are, because the design required additional classes for the prototype to function as intended. Class diagrams, of the classes that does not deviate from the design, are left out of the implementation description. To see the class diagrams, one can look them up in the design. The description of all the classes is split into three groups with these package names; `common`, `client`, and `server`. The `common` package contain classes, that both the client and the server use. All the classes the client requires, which are not in the `common` package are in the `client` package. All the classes the server requires, which are not in the `common` package are in the `server` package. The three packages; `common`, `client`, and `server` are within the package `dk.phiber.tamperproofing`.

Common

Here is the description of the classes in the package `dk.phiber.tamperproofing.common`.

ByteArray: In Figure 5.1 the class that provides operators for manipulating byte arrays is seen. The following operators are supported `append`, `and` and `xor`, the `and` and `xor` are Boolean operators that can operate with byte arrays. The append operation return a byte array where the `input1` byte array is appended the `input2` byte array.

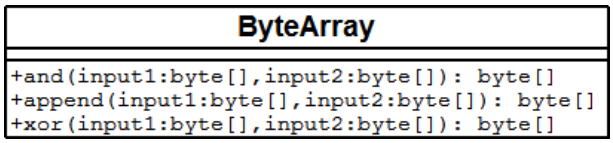

Figure 5.1: `ByteArray` class diagram.

ChallengeResponse: The `ChallengeResponse` interface defines the RMI methods for communication. No changes have been made to its class structure since the design.

CSPRBG: The `CSPRBG` provide the method `computePRMTBlock` in the `Hashing` class with a CSPRBG that output the same pseudo-random bits inputted the same seed. The structure of the class can be seen on Figure 5.2.

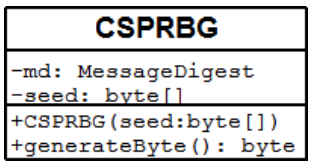

Figure 5.2: `CSPRBG` class diagram.

Hashing: The `Hashing` class is used for computing the checksum, the changes from the design to implementation is, that it do not use the `java.security.SecureRandom` but instead uses `CSPRBG` and the `computeHMAC` method has been made public. It uses `CSPRBG` instead because it produces the same pseudo-random bits with the same seed, and the `computeHMAC` has been made public because the CAC computation uses it. No changes have been made to its class structure since the design.

Client

Here is the description of the classes in the package `dk.phiber.tamperproofing.client`.

Client: The `Client` class that provides the API for RMI communication to the client and it structures is the same as in the design. No changes have been made to its class structure since the design.

CustomClassLoader: In Figure 5.3 is seen the custom class loader used for loading the classes not in the JCF, and at the same time stores class files in the `MemStorage` class shown in Figure 4.9. The access modifiers are changed from protected to public on the `findClass` method and from protected to private on the `findClassBytes` method.

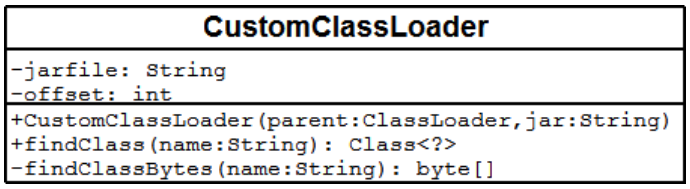

Figure 5.3: `CustomClassLoader` class diagram.

InputForm: In Figure 5.4 is shown the class to use for drawing and controlling the GUI. The class `LoadApplet` loads the class `InputForm`. The GUI consists of one input field and two buttons, the action listeners for these two buttons are subclasses. The `SubmitButtonListener` class contain the actions to perform when the user push the submit button, which are to compute a CAC of the input, and send the CAC and the data to the server. The `ClearButtonListener` class contains the action of clearing the content of the input field.

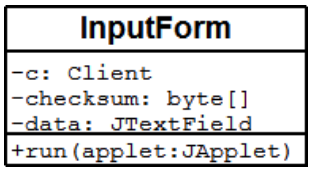

Figure 5.4: `InputForm` class diagram.

LoadApplet: The `LoadApplet` class is used for loading the GUI class `InputForm`. No changes have been made to its class structure since the design.

MemStorage: The `MemStorage` class is used for storing loaded classes. No changes have been made to its class structure since the design.

Server

Here is the description of the classes in the package `dk.phiber.tamperproofing.server`.

MemStorage: In Figure 5.5 is seen the `MemStorage` class to use on the server. The class is added so each verifier object does not need its own byte array of the program to validate. This is why this class is not a singleton class, because more instances of the class could be necessary in the future.

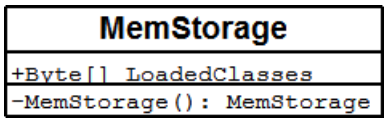

Figure 5.5: `MemStorage` class diagram.

Server: In Figure 5.6 is seen the class providing the RMI methods that the client invokes. It implements the methods in the `ChallengeResponse` interface, which stores the definitions of the RMI methods, the client/server communication requires. No changes have been made to its class structure since the design. Since the design a field variable of type server was added `MemStorage`, then client program is only loaded once even through there is multiple verifiers.

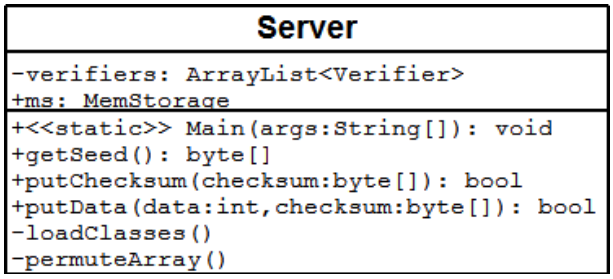

Figure 5.6: `Server` class diagram.

Verifier: The `Verifier` class is used for storing the state of a specific client validation. The server create and instance of the verifier for each seed request from a client. Then if a valid checksum is returned using the defined seed, the state is changed to software being valid and the client is now allowed to send data to the server. No changes have been made to its class structure since the design.

5.5 Running the Prototype

This section explains what configurations that are within the prototype source code, and how to start the server and the client programs.

5.5.1 Configurations in the Prototype

This implementation of the tamper-proofing prototype, requires properties being set manually in the source code. This section is to explain where these configurations exists in the source code. Some of these configurations are important if program obfuscation is done, this is because the ProGuard does not interpret string references to classes. Each description below represents a class where one or more configurations are set:

MemStorage: The class stored in `dk.phiber.tamperproofing.client.MemStorage` and the one stored in `dk.phiber.tamperproofing.server.MemStorage` are both constructed with the knowledge of what the required number of bytes is for storing the client. This is set in the constructor seen in Listing 5.5.1, the size assigned shall be a multiplication of data block size of 512 bytes.

```
1 MemStorage() {
2     loadedClasses = new byte[10240]; // 10240 = 21 * 512
3 }
```

Server: The order in which the classes are loaded into `MemStorage` is important. The order the class names are stored in the string array `names`, are the order the classes are loaded into the server `MemStorage` class. The order the class names are assigned the string array `names` have to be the same order the classes are loaded on the client. In Listing 5.5.1 the assignment of the string array `names` can be seen. In case of obfuscation, it is necessary to translate class names to the obfuscated class names, because the obfuscation program cannot detect that these strings actually are class names.

```
1  ...
2  String[] names = new String[8];
3  names[0] = "dk.phiber.tamperproofing.client.InputForm";
4  names[1] = "dk.phiber.tamperproofing.client.
       InputForm$ClearButtonListener";
5  names[2] = "dk.phiber.tamperproofing.client.
       InputForm$SubmitButtonListener";
6  names[3] = "dk.phiber.tamperproofing.client.Client";
7  names[4] = "dk.phiber.tamperproofing.common.Hashing";
8  names[5] = "dk.phiber.tamperproofing.common.CSPRBG";
9  names[6] = "dk.phiber.tamperproofing.common.ChallengeResponse";
10 names[7] = "dk.phiber.tamperproofing.common.ByteArray";
11 ...
```

If the client jar file has another name than `client.jar` it is important that file name is renamed to that. The line where the file name can be altered is seen in Listing 5.5.1, it is placed in the `loadClasses` method, where it is possible to alter the file name.

```
1 ...
2 File jar = new File("client.jar");
3 ...
```

If the byte array operation file has another name than `operations.txt` it is important to rename the file name. See Listing 5.5.1 to see how the line looks within the method `permuteArray`, where it is possible to alter the file name.

```
1 ...
2 fi = new FileInputStream("operations.txt");
3 ...
```

The time-frame where the server accepts the initial client validation checksum is 20 seconds from when the seed is requested. It is possible to change this value of

20000 milliseconds to some other value. See Listing 5.5.1 to see how the line looks within the method `putChecksum`, where it is possible to chane the client validation time-frame.

```
1 ...
2 if(d.getTime() < (verifier.getTime() + 20000)) {
3 ...
4 }
```

The time-frame where the server accepts one input with CAC is 10 minutes from when the seed is requested. It is possible to change this value of 600000 milliseconds to some other value. See Listing 5.5.1 to see how the line looks within the method `putData`, where it is possible to change the input time-frame.

```
1 ...
2 if(verifier.verifyCAC(cackey, value, checksum) && d.getTime() <
      (verifier.getTime() + 600000)) {
3 ...
4 }
```

The two time-frames of 20 seconds for client validation and 10 minutes for input submission are two guesses of the minimal time-frames, where it is possible for users that behave as intended with high probability to successful validate the client and send input data with a valid CAC to the server.

Client: `ChallengeResponse` interface is defined by a string that can be necessary to change if obfuscation is used. See Listing 5.5.1 to see how the line looks within the constructor method `Client`, where it is possible to change the name to the obfuscated name.

```
1 ...
2 stub = (ChallengeResponse)registry.lookup("ChallengeResponse");
3 ...
```

InputForm: The input form loads the rest of the class definitions into the JVM in the right order. These strings of class names would also be necessary to change if obfuscation is used. See Listing 5.5.1 to see how the lines look within the method `run`, where it is possible to change the order and the class names to the obfuscated class names.

```
1 ...
2 CustomClassLoader ccl = (CustomClassLoader)this.getClass().
     getClassLoader();
3 ccl.findClass("dk.phiber.tamperproofing.client.
     InputForm$ClearButtonListener");
4 ccl.findClass("dk.phiber.tamperproofing.client.
     InputForm$SubmitButtonListener");
5 ccl.findClass("dk.phiber.tamperproofing.client.Client");
6 ccl.findClass("dk.phiber.tamperproofing.common.Hashing");
7 ccl.findClass("dk.phiber.tamperproofing.common.CSPRBG");
8 ccl.findClass("dk.phiber.tamperproofing.common.
     ChallengeResponse");
```

```
9  ccl.findClass("dk.phiber.tamperproofing.common.ByteArray");
10 ...
```

The client is constructed with the IP of the server to connect to. See Listing 5.5.1 to see how the line looks within the `Client` constructor, where it is possible to change to another server IP.

```
1 ...
2 c = new Client("127.0.0.1");
3 ...
```

LoadApplet: Remember to rename the class to load in the HyperText Markup Language (HTML) file in case of obfuscation, or when the client program is stored in a archive file with another name. This is only necessary if one manually obfuscate the class name of the ProGuard entry point. See Listing 5.5.1 to see how the line looks, where one can change the class name of the class to load or the name of the client archive file.

```
1 ...
2 <applet code=dk.phiber.tamperproofing.client.LoadApplet.class
     archive="client.jar" width="200" height="200" >
3 ...
```

The custom class loader is constructed using a string containing the name of the client archive file, if the name of the archive changes this shall be renamed. See Listing 5.5.1 to see how the line looks within the method `init`, where it is possible to rename the name of the archive file.

```
1 ...
2 ccl = new CustomClassLoader(CustomClassLoader.class.
     getClassLoader(), "client.jar");
3 ...
```

The loading of the input form is done through a string definition. If obfuscation is used it can be necessary to rename both the class name and the method name `run`. See Listing 5.5.1 to see how the lines looks within the method `init`, where it is possible to rename the class name and method name.

```
1 ...
2 Class<?> c0 = ccl.findClass("dk.phiber.tamperproofing.client.
     InputForm");
3 Method m = null;
4 Object cl = c0.newInstance();
5 m = c0.getMethod("run", new Class[] {JApplet.class});
6 m.invoke(cl, new Object[] {this});
7 ...
```

5.5.2 Starting the Server

The function of the server is to validate the client and receive input from the client. A requirement is then, that the server is running before the client is started. The server

require that these two files are present `operations.txt` seen in Appendix E Listing E.1 and `cackey.txt` seen in Appendix E Listing E.2, which are the two files that contains byte array operations and the CAC key parts. The server use RMI for communication so the `rmiregistry` service has to be started manually before starting the server.

To start the server use the Ant build file in Appendix C, then to start the server enter this command `ant run.server`.

5.5.3 Starting the Client

The function of the client is to compute a checksum and send it to the server with the use of a challenge requested from the server. When the client checksum has been validated, the user is allowed to submit data onte to the server. The data is accompanied with a CAC that has the purpose, that the server can validate the CAC to come form a valid client.

To start the client use the Ant build file in Appendix C, then to start the client enter this command `ant run.client`. What this command does is to start the Sun `AppletViewer` with this policy in Appendix D Listing D.2 running the Appendix D Listing D.1. However, one shall make sure that the server is running or the client will output an error.

5.6 Summary

The server does not allow the user to submit data(an integer) to the server, before the client has to send a valid client validation checksum. The server afterward accepts input when the client sends an input with a valid CAC. The implementations of the client validation checksum and the CAC scheme store the key information in the client program.

It will be possible to extract the information for computing the client validation checksum and the CAC, because the information is stored in the client program. The goal of the security mechanisms is to ensure trust for a small period of time, where the server can assume that the data are send from a valid client.

Chapter 6

Testing

This chapter documents the security test of the tamper-proofing security mechanism that are proposed in Chapter 4. To test the security of the tamper-proofing security mechanism, the following two scripted attacks should fail for the security to be successful. The scripted attack has the goal of retrieving all legal byte array operations, and the attack requires a running-time within the time-frame of 20 seconds, where the server still accepts software validation checksums or the attack is unsuccessful. The second scripted attack has the goal of locating all CAC key parts from the client, and the attack require a running-time within the time-frame of 10 minutes, where the server still accepts input data.

In this project no financial resources are present for employing external parties to do the test. Only a project biased person is available to perform the test, however, the knowledge of the implementation of the security mechanism can compensate for being project biased. For the same reason the attacks are deployed against a client program that is not obfuscated with ProGuard, because the knowledge of the implementation would make the effect of the obfuscation limited. It is also a stronger attack, because is should be faster to perform when no interpretation of the obfuscated code is required.

6.1 Test-case

This is the test-case for testing the security of the tamper-proofing security mechanism. For the test to be unsuccessful the secrets hidden in the client program can be found within the time-frames, and then it is possible to mount an attack where the attacker can convince the server that the client is valid or that the input is sent from a valid client. The test-case is then to extract the hidden information within the time-frames and without running the client, and if this is possible the test is unsuccessful.

The goals of the attacks are on the list below:

- Extract all legal byte array operations and the class loading order.

- Extract all CAC key parts.

To mount a successful attack against the checksum validation process is to convince the server that the client is valid. The attacker has to obtain the order in which the classes are loaded into the byte array and all the legal byte array operations.

To mount a successful analysis attack against the CAC scheme is to convince the server that the input is sent from a valid client. The attacker has to obtain the CAC key parts, and the order in which they are used to compute the CAC.

To mount a successful combination attack against the server, is possible if only the CAC information can be extracted successfully. The attack consist of two steps: 1) to run the original client normally to convince the server it is valid, 2) to stop the client and compute the CAC with the extracted CAC key parts and the chosen user input without using the official client. The input will appear to come from a valid client when it actually does not.

To extract the hidden information the scripts found in Appendix F have been created to extract static method call information, byte array operation accesses to the `MemStorage` class, and the class loading order done by the `CustomClassLoader`. The script uses the three programs `javap`, `awk` and `sed`, to compose the three extraction processes byte array operations, method call lists, and class loading order. The byte array operations extraction process extracts all byte array operations both legal and illegal. To conclude which byte array operations are illegal, one uses the method calls, to observe which byte array operation is present in methods not being called or being called after the checksum has been computed. The class loading process extracts the order in which the `CustomClassLoader` loads classes, which controls the order of the data stored in the byte array, that byte array operations uses. The extraction processes of extracting the CAC key parts can be done observing the method calls, because all method calls within the action listener for the submit button shows, how many key parts there are, and where to find the key parts.

If one of the three attacks above can be done successful within the time-frames for client checksum validation or for input data submission, the tamper-proofing security mechanism is un-successful.

To run the test, do the following: Open an Linux terminal in the source directory and type the command `ant client` to build the client and afterward type the command `./combinedextract.sh`, which then output two log files named `search.log`, `classloading.log` and `calllists.log`. The file `search.log` contains the byte array operations found in the client program. The file `classloading.log` contains a list of the loading order of the client classes. The file `calllists.log` contains a list of all methods in the client, and with each method is a list of the methods it calls.

6.2 Test Results

The test has been performed as specified, and the three log files can be seen in Appendix G Listing G.1, Listing G.2 and Listing G.3. The test result section is split into two subsections. The first subsection contains the conclusion on the extraction of byte array operations and client class loading order. The second subsection contains the conclusion on the extraction of CAC key parts.

The running-times from five runs of the test script `combinedextract.sh` are the following numbers in seconds: 2.251, 2.304, 2.222, 2.182, and 2.232, which gives an average running-time of 2.2382 seconds. The tool used for timing the running-time is the Linux command `time`, the command is invoked like this `time ./combinedextract.sh`.

The following hardware and software configuration was used. The hardware specification is an IBM R40 laptop with 1.2 GHz Pentium 4 Mobile processor, 512 MB PC-133 Random-Access Memory (RAM), and HD with 4800 RPM. The operating system is Xubuntu version 7.04 and JDK version 1.6.0_01.

6.2.1 Extraction of Byte Array Operations

It is easy to extract all byte array operations present in the code. This can be seen in `search.log` shown in Appendix G Listing G.2, where all 11 byte array operations are present. There are two type one byte array operations and nine type two byte array operations. To recognize the difference between the two byte array operations use these two patterns. The type one pattern is the sequence of four `sipush` instructions. The type two pattern is the sequence of a `sipush` instruction followed by a `bipush` instruction.

To make the attack of finding the set of legal byte array operations, one has to know the order in which the classes are loaded. By analyzing the `classloading.log` in Appendix G Listing G.3. In the analysis the only line excluded from the log file is loading of `InputForm`, because we know that this is the first class loaded. The first class in the `MemStorage` byte array is then `InputForm` and the rest of the classes are loaded as shown in Listing 6.2.1 from top to bottom.

```
1     #8; //String dk.phiber.tamperproofing.client.
          InputForm$ClearButtonListener
2     #10; //String dk.phiber.tamperproofing.client.
          InputForm$SubmitButtonListener
3     #11; //String dk.phiber.tamperproofing.client.Client
4     #12; //String dk.phiber.tamperproofing.common.Hashing
5     #13; //String dk.phiber.tamperproofing.common.CSPRBG
6     #14; //String dk.phiber.tamperproofing.common.ChallengeResponse
7     #15; //String dk.phiber.tamperproofing.common.ByteArray
8  ...
```

The next step is to recognize the difference between legal and illegal byte array operations. Only one illegal byte array operation can be found analyzing Appendix G Listing G.2,

that byte array operation is a type two byte array operation present in Listing 6.2.1. The byte array operation is illegal with no effect, because it is done after the checksum computation. Because two illegal byte array operations are present in the code, there is still one byte operation to find. The illegal byte array operation not found is the type one byte array operation where array index 4452 is switched with index 7529. The illegal byte array operation can been seen in Listing 6.2.1. Analyzing the listings Appendix G there are no means to separate this byte array operations out from the legal ones, unless the re-engineer analyze the context of where byte array operations are placed. To enforce security one could now add so many byte array operations to the client that the right byte array combination can only be guessed with negligible probability within the time period where software validation is allowed. This will require many byte array operations to be added to the client, to assume that analyzing the context of each byte array operations would be more time consuming, because analysis of the program context will require more manual work by the re-engineer.

```
1 ...
2    587: sipush  3354
3    590: bipush  78
4 ...
```

```
1 ...
2    23:  sipush  4452
3    35:  sipush  4452
4    44:  sipush  7529
5    55:  sipush  7529
6 ...
```

It is then important to find how this property is obtained by the illegal byte array operation. This is done by analyzing the implementation. From the byte array operation's position in Listing G.2 it can be concluded that the permutation is in the class `Hashing` and the method `computeChecksum`. The class `InputForm` calls the method `computeChecksum` with input, so the `computeChecksum` does the type one byte array operations twice. The effect of the type one byte array operation is then cancelled out. This is a difficult problem to find without context analysis of the client program. The attackers best option is to run the program to get a valid client validation checksum.

6.2.2 Extraction of CAC Key Parts

To find the CAC key parts a different method is used than the one for finding byte array operations. To find byte array operations one could observe accesses to the `MemStorage` class. This method is not possible with the CAC scheme, but a method that does work is finding where the input data is send from in the client, because that will be close to where the CAC computation is. When one has found the CAC computation, references to the different key parts will be there. The location of the CAC computation in the disassembled class files Listing G.1 can be seen in Listing 6.2.2.

```
1 ...
```

```
2  InputForm$SubmitButtonListener.dis
3     5:   invokespecial   #3; //Method value:()[B
4  common/Hashing."<init>":()V
5  common/Hashing.getB:()[B
6  common/Hashing.computeHMAC:([B[B)[B
7  common/Hashing.computeHMAC:([B[B)[B
8  client/InputForm;)[B
9  common/ByteArray.xor:([B[B)[B
10 client/Client;
11 client/Client.putData:(I[B)Z
12 ...
```

If one observes the lines, one can assume that the two calls to the method `computeHMAC` is the actual computation of the CAC. The two calls will then allow the assumption that there are two CAC key parts, and that the attacker can obtain these two key parts before the CAC computation. The only method calls from client code in question is the creation of a `Hashing` object, and two methods `getB` and `value`, then these two methods would be likely key part holders. At the same time it can be concluded that the two methods return values are of type byte array, which is the same type as the key for the `computeHMAC` method should be. A safe assumption is that this CAC scheme cannot be assumed secure, because it is possible to do manually within the time-frame of 10 minutes with few key parts.

6.3 Summary

In general it can be concluded that the security does not hold, because the CAC scheme cannot sufficiently hide its key parts. This will leave the client susceptible to the combined attack in the test-case. Where the attacker uses a valid client for the client validation checksum, and then uses an invalid client to send input to the server. Another important thing to notice is that the byte array operations scheme is successful in hiding one illegal byte array operation within the set of legal byte array operations. It can be a possibility to use the same method for hiding CAC key parts, the method is to make the key parts dependant on how the program runs. One possibility could be to add methods that access and change the key part, but these methods have no direct relation to the CAC computation, except a common interest in one or more key parts. This new method requires the attacker to analyze all accesses to CAC key parts. It will resolve in a similar problem that gives the illegal byte array operation its hiding property, which is that the re-engineer have to observe how many times the program call each method. Because a method that change the CAC key part with one call, can change the key part to something else after multiple calls.

Chapter 7

Conclusion

The initial goal of this thesis was to develop a suite of security mechanisms to prevent the following attacks; MITM, phishing and tampering attacks. This suite should then be easy to embed into the client software making it resistant against these attacks.

It was then found that good security mechanisms existed against MITM and phishing attacks. The goal of the project was then changed from developing a suite of security mechanisms, to instead design and implement a tamper-proofing security mechanism for Java client software.

The tamper-proofing security mechanism design and implementation was in the test discarded as being a qualified solution for allowing servers to trust that data comes from un-tampered client. While testing it, it was found that an illegal byte array operation had the property, which enforces the attacker to analyze the context where the byte array operation is placed, or to guess the correct combination of the byte array operations. The attacker is only allowed one guess using a one-time passwords mechanism, and because a uniquely keyed client is compiled to each user. The attacker will not be more powerful even if she has more than one user's client release, because the task of extracting byte array operations will result in different outputs. The most obvious way to validate the client will be to run the un-tampered client. This then pushes the client validation from the program initialization phase to the data input phase, and the data input phase security is handled by the CAC scheme.

The design of the CAC scheme was not sufficient, because it is possible to manually to find the CAC key parts by analyzing the output of the test script. The property obtained by the illegal byte array operation is possible to extend to the design of a new secure CAC scheme. This will make it possible to design a tamper-proofing security mechanism, which actual can offer security and trust within a stated period of time.

The final conclusion is that it should be possible to make a tamper-proofing security mechanism that provides trust in clients for small periods of time.

Bibliography

[Bel05] Mihir Bellare. Attacks on sha-1. Technical report, Dept. of Computer Science & Engineering at University of California, 2005.

[DT05] Rachna Dhamija and J. D. Tygar. The battle against phishing: Dynamic security skins. In *SOUPS '05: Proceedings of the 2005 symposium on Usable privacy and security*, pages 77–88, New York, NY, USA, 2005. ACM Press.

[DTH06] Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 581–590, New York, NY, USA, 2006. ACM Press.

[FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The ssl protocol - version 3.0. Technical report, Netscape, 1996.

[Gon02] Li Gong. *Java Platform Security Architecture*. Sun Microsystems, Inc., 2002.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, Inc., 2nd edition, 1999. ISBN: 0201787911.

[MS05] Kevin D. Mitnick and William L. Simon. *The Art of Intrusion*. Wiley Publishing, Inc., 2005. ISBN: 0764569597.

[NM03] Gleb Naumovich and Nasir Memon. Preventing piracy, reverse engineering, and tampering. *Computer*, 36(7):64–71, 2003.

[PKK+00] Jens Palsberg, S. Krishnaswamy, Minseok Kwon, D. Ma, Qiuyun Shao, and Y. Zhang. Experience with software watermarking. In *ACSAC*, pages 308–316, 2000.

[SLS+05] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, October 2005.

[SPvDK04] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Swatt: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[Sti06] Douglas R. Stinson. *Cryptography - Theory and Practice*. Chapman & Hall/CRC, 3rd edition, 2006. ISBN: 1584885084.

[Tan03] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 4th edition, 2003. ISBN: 0130384887.

[Tra] Greg Travis. Understanding the java classloader. http://www.panix.com/ mito/articles/articles/classloader/j-classloader-ltr.pdf.

[TWMP06] D. Taylor, Thomas Wu, N. Mavrogiannopoulos, and T. Perrin. Using srp for tls authentication. Technical report, IETF, 2006.

[Vir91] Chonchanok Viravan. Fault investigation and trial. Technical Report SERC-TR-104-P, 1991.

[Woe05] Lorraine Woellert. Sony's copyright overreach. November 2005. http://www.businessweek.com/technology/content/nov2005/tc20051117_444162.htm.

[Wu98] Thomas Wu. The secure remote password protocol. In *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, 1998.

[Wu02] Thomas Wu. Srp-6: Improvements and refinements to the secure remote password protocol, 2002. http://srp.stanford.edu/srp6.ps.

Chapter 8

Acronyms

AAU AAlborg University

AES Advanced Encryption Standard

API Application Programming Interface

CA Certificate Authority

CAC Client Authentication Code

CPU Central Processing Unit

CSPRBG Cryptographic Secure Pseudorandom Bit Generator

EINOO External, Insider, Network, Off-line, and On-line

FIPS Federal Information Procession Standards

GUI Graphical User Interface

HMAC Keyed-Hash Message Authentication Code

HTML HyperText Markup Language

HTTPS HyperText Transmission Protocol, Secure

IIS Internet Information Services

IP Internet Protocol

JDK Java Developer Kit

JCF Java Collections Framework

JVM Java Virtual Machine

MAC Message Authentication Code

MITM Man In The Middle

PRBG Pseudorandom Bit Generator

PRMT Pseudorandom Memory Traversal

RAM Random-Access Memory

RMI Remote Method Invocation

RSA Rivest, Shamir, and Adleman

SHA-1 Secure Hash Algorithm-1

SRP Secure Remote Password protocol

SSL Secure Socket Layer

SWATT SoftWare-based ATTestation

TCP Transmission Control Protocol

TLS Transport Layer Security protocol

Appendix A

Functions

An example of how encryption and decoy functions can appear in Java.

```
// Addition in Exponent
// Variant One
public int addSign(int toSign){
  int kp = 4; // Keypart
  BigInteger n = ; // Declare modulus
  return toSign.pow(kp).mod(n);
}
// Variant Two
public int addSign(int toSign, BigInteger n){
  int kp = 4; // Keypart
  return toSign.pow(kp).mod(n);
}
// Variant Three
int kp = 4; // Keypart
public int addSign(int toSign, BigInteger n){
  return toSign.pow(kp).mod(n);
}
// Variant Four
int kp = 4; // Keypart
BigInteger n = ; // Declare modulus
public int addSign(int toSign, BigInteger n){
  return toSign.pow(kp).mod(n);
}
// Multiplication in Exponent
// Variant One
public int mulSign(int toSign){
  int kp = 4; // Keypart
  BigInteger n = ; // Declare modulus
  return toSign.pow(kp).mod(n);
}
// Variant Two
public int mulSign(int toSign, BigInteger n){
  int kp = 4; // Keypart
  return toSign.pow(kp).mod(n);
}
// Variant Three
int kp = 4; // Keypart
public int mulSign(int toSign, BigInteger n){
  return toSign.pow(kp).mod(n);
}
// Variant Four
int kp = 4; // Keypart
BigInteger n = ; // Declare modulus
public int mulSign(int toSign, BigInteger n){
  return toSign.pow(kp).mod(n);
}
```

```
47 // Arithmetic Functions
48 public int add(int number1, int number2){
49   return (number1 + number2)
50 }
51 public int sup(int number1, int number2){
52   return (number1 - number2)
53 }
54 public int mul(int number1, int number2){
55   return (number1 * number2)
56 }
57 public int div(int number1, int number2){
58   return (number1 / number2)
59 }
60 public int mod(int number1, int number2){
61   return (number1 % number2)
62 }
```

Listing A.1: An example of how encryption and decoy functions can appear in Java.

Appendix B

Obfuscation Example

This appendix show disassembling of a small Java program which source is in Listing B.1. The disassembling result of the un-obfuscated Java program is in Listing B.2. The disassembling result of the obfuscated Java program is in Listing B.3. The configuration file for obfuscating the Java program with ProGuard is in Listing B.4.

```
1  package dk.phiber;
2
3  public class Disassemble {
4
5    public int Public = 0;
6    protected int Protected = 0;
7    private int Private = 1000000;
8    /**
9     * @param args
10    */
11   public static void main(String[] args) {
12     // TODO Auto-generated method stub
13     Disassemble disassemble = new Disassemble();
14     disassemble.Public = disassemble.Private - disassemble.Protected;
15     disassemble.pub(1000000);
16     disassemble.pro("Fisk is good");
17     disassemble.pri(5);
18
19   }
20   public void pub(int i) {
21     String s = "Fishing";
22     System.out.println("Addition: " + i + Public);
23     this.pro(s);
24   }
25   protected void pro(String str) {
26     System.out.println(str + Private);
27   }
28   private int pri(int i) {
29     return Protected + 5 + i;
30   }
31 }
```

Listing B.1: The Java source for the small Java program.

```
1  Compiled from "Disassemble.java"
2  public class dk.phiber.Disassemble extends java.lang.Object{
3  public int Public;
4
5  protected int Protected;
6
```

```
7  public dk.phiber.Disassemble();
8    Code:
9      0: aload_0
10     1: invokespecial #12; //Method java/lang/Object."<init>":()V
11     4: aload_0
12     5: iconst_0
13     6: putfield  #14; //Field Public:I
14     9: aload_0
15     10:  iconst_0
16     11:  putfield  #16; //Field Protected:I
17     14:  aload_0
18     15:  ldc #18; //int 1000000
19     17:  putfield  #19; //Field Private:I
20     20:  return
21
22 public static void main(java.lang.String[]);
23   Code:
24     0: new #1; //class dk/phiber/Disassemble
25     3: dup
26     4: invokespecial #27; //Method "<init>":()V
27     7: astore_1
28     8: aload_1
29     9: aload_1
30     10:  getfield  #19; //Field Private:I
31     13:  aload_1
32     14:  getfield  #16; //Field Protected:I
33     17:  isub
34     18:  putfield  #14; //Field Public:I
35     21:  aload_1
36     22:  ldc #18; //int 1000000
37     24:  invokevirtual #28; //Method pub:(I)V
38     27:  aload_1
39     28:  ldc #32; //String Fisk is good
40     30:  invokevirtual #34; //Method pro:(Ljava/lang/String;)V
41     33:  aload_1
42     34:  iconst_5
43     35:  invokespecial #38; //Method pri:(I)I
44     38:  pop
45     39:  return
46
47 public void pub(int);
48   Code:
49     0: ldc #45; //String Fishing
50     2: astore_2
51     3: getstatic #47; //Field java/lang/System.out:Ljava/io/PrintStream;
52     6: new #53; //class java/lang/StringBuilder
53     9: dup
54     10:  ldc #55; //String Addition:
55     12:  invokespecial #57; //Method java/lang/StringBuilder."<init>":(Ljava/lang/
           String;)V
56     15:  iload_1
57     16:  invokevirtual #59; //Method java/lang/StringBuilder.append:(I)Ljava/lang/
           StringBuilder;
58     19:  aload_0
59     20:  getfield  #14; //Field Public:I
60     23:  invokevirtual #59; //Method java/lang/StringBuilder.append:(I)Ljava/lang/
           StringBuilder;
61     26:  invokevirtual #63; //Method java/lang/StringBuilder.toString:()Ljava/lang/
           String;
62     29:  invokevirtual #67; //Method java/io/PrintStream.println:(Ljava/lang/String
           ;)V
63     32:  aload_0
64     33:  aload_2
65     34:  invokevirtual #34; //Method pro:(Ljava/lang/String;)V
66     37:  return
67
68 protected void pro(java.lang.String);
69   Code:
70     0: getstatic #47; //Field java/lang/System.out:Ljava/io/PrintStream;
71     3: new #53; //class java/lang/StringBuilder
```

```
72     6: dup
73     7: aload_1
74     8: invokestatic  #75; //Method java/lang/String.valueOf:(Ljava/lang/Object;)
           Ljava/lang/String;
75     11:  invokespecial #57; //Method java/lang/StringBuilder."<init>":(Ljava/lang/
           String;)V
76     14:  aload_0
77     15:  getfield  #19; //Field Private:I
78     18:  invokevirtual #59; //Method java/lang/StringBuilder.append:(I)Ljava/lang/
           StringBuilder;
79     21:  invokevirtual #63; //Method java/lang/StringBuilder.toString:()Ljava/lang/
           String;
80     24:  invokevirtual #67; //Method java/io/PrintStream.println:(Ljava/lang/String
           ;)V
81     27:  return
82
83 }
```

Listing B.2: The disassembling result of the small un-obfuscated Java program.

```
1  public class dk.phiber.Disassemble extends java.lang.Object{
2  public int a;
3
4  public int b;
5
6  public dk.phiber.Disassemble();
7    Code:
8      0: aload_0
9      1: invokespecial #11; //Method java/lang/Object."<init>":()V
10     4: aload_0
11     5: iconst_0
12     6: putfield  #7; //Field a:I
13     9: aload_0
14     10:  iconst_0
15     11:  putfield  #8; //Field b:I
16     14:  aload_0
17     15:  ldc #23; //int 1000000
18     17:  putfield  #9; //Field c:I
19     20:  return
20
21 public static void main(java.lang.String[]);
22   Code:
23     0: new #1; //class dk/phiber/Disassemble
24     3: dup
25     4: invokespecial #12; //Method "<init>":()V
26     7: dup
27     8: astore_1
28     9: aload_1
29     10:  getfield  #9; //Field c:I
30     13:  aload_1
31     14:  getfield  #8; //Field b:I
32     17:  isub
33     18:  putfield  #7; //Field a:I
34     21:  aload_1
35     22:  ldc #23; //int 1000000
36     24:  invokevirtual #14; //Method a:(I)V
37     27:  aload_1
38     28:  ldc #22; //String Fisk is good
39     30:  invokevirtual #15; //Method a:(Ljava/lang/String;)V
40     33:  return
41
42 public final void a(int);
43   Code:
44     0: ldc #21; //String Fishing
45     2: astore_2
46     3: getstatic #10; //Field java/lang/System.out:Ljava/io/PrintStream;
47     6: new #5; //class java/lang/StringBuilder
48     9: dup
49     10:  ldc #20; //String Addition:
```

```
50     12:  invokespecial #13; //Method java/lang/StringBuilder."<init>":(Ljava/lang/
           String;)V
51     15:  iload_1
52     16:  invokevirtual #16; //Method java/lang/StringBuilder.append:(I)Ljava/lang/
           StringBuilder;
53     19:  aload_0
54     20:  getfield  #7; //Field a:I
55     23:  invokevirtual #16; //Method java/lang/StringBuilder.append:(I)Ljava/lang/
           StringBuilder;
56     26:  invokevirtual #18; //Method java/lang/StringBuilder.toString:()Ljava/lang/
           String;
57     29:  invokevirtual #17; //Method java/io/PrintStream.println:(Ljava/lang/String
           ;)V
58     32:  aload_0
59     33:  aload_2
60     34:  invokevirtual #15; //Method a:(Ljava/lang/String;)V
61     37:  return
62
63 public final void a(java.lang.String);
64   Code:
65     0: getstatic #10; //Field java/lang/System.out:Ljava/io/PrintStream;
66     3: new #5; //class java/lang/StringBuilder
67     6: dup
68     7: aload_1
69     8: invokestatic  #19; //Method java/lang/String.valueOf:(Ljava/lang/Object;)
           Ljava/lang/String;
70     11:  invokespecial #13; //Method java/lang/StringBuilder."<init>":(Ljava/lang/
           String;)V
71     14:  aload_0
72     15:  getfield  #9; //Field c:I
73     18:  invokevirtual #16; //Method java/lang/StringBuilder.append:(I)Ljava/lang/
           StringBuilder;
74     21:  invokevirtual #18; //Method java/lang/StringBuilder.toString:()Ljava/lang/
           String;
75     24:  invokevirtual #17; //Method java/io/PrintStream.println:(Ljava/lang/String
           ;)V
76     27:  return
77
78 }
```

Listing B.3: The disassembling result of the small obfuscated Java program.

```
1  -injars disassemble.jar
2  -outjars odisassemble.jar
3  -libraryjars java/jre/lib/rt.jar
4  -printmapping proguard.map
5  -overloadaggressively
6  -defaultpackage ''
7  -allowaccessmodification
8
9  -keep public class dk.phiber.Disassemble {
10   public static void main(java.lang.String[]);
11 }
```

Listing B.4: The ProGuard configuration file.

Appendix C

The Ant Build File

The Ant build file used for building and running the prototype can be seen in Listing C.1.

```
<project name="ClientSecurity" default="default" basedir=".">
  <description>The first build file.</description>
  <property name="build" location="build"/>
  <property name="dist" location="dist"/>
  <property name="src" location=""/>
  <property name="package" location="${build}/dk/phiber/tamperproofing"/>
  <property name="build.client" location="${package}/client"/>
  <property name="build.common" location="${package}/common"/>
  <property name="build.impclasses" location="${package}/impclasses"/>
  <property name="build.permutation" location="${package}/permutation"/>
  <property name="build.server" location="${package}/server"/>
  <property name="jar.client" location="${dist}/client"/>
  <property name="jar.common" location="${dist}/common"/>
  <property name="jar.impclasses" location="${dist}/impclasses"/>
  <property name="jar.permutation" location="${dist}/permutation"/>
  <property name="jar.server" location="${dist}/server"/>
  <target name="default" depends="compile,server,client"/>
  <target name="compile">
    <mkdir dir="${build}"/>
    <javac srcdir="${src}/" destdir="${build}"/>
  </target>
  <target name="clean">
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
  <target name="server" depends="compile">
    <mkdir dir="${dist}"/>
    <copy file="operations.txt" todir="${dist}" overwrite="true"/>
    <copy file="cackey.txt" todir="${dist}" overwrite="true"/>
    <jar destfile="${jar.server}.jar" basedir="${build}"
    includes="**/server/*.class,**/common/*.class,**/client/*.class">
      <manifest>
        <attribute name="Main-Class"
        value="dk.phiber.tamperproofing.server.Server"/>
      </manifest>
    </jar>
  </target>
  <target name="client" depends="compile">
    <mkdir dir="${dist}"/>
    <jar destfile="${jar.client}.jar" basedir="${build}" includes="**/client/*.
        class,**/common/*.class"/>
    <copy file="applet.html" todir="${dist}" overwrite="true"/>
    <copy file="java.policy.applet" todir="${dist}"
    overwrite="true"/>
  </target>
  <target name="permutation" depends="compile">
```

```
46      <mkdir dir="${dist}"/>
47      <jar destfile="${jar.permutation}.jar" basedir="${build}"
48      includes="**/permutation/*.class">
49        <manifest>
50          <attribute name="Main-Class"
51          value="dk.phiber.tamperproofing.permutation.PermutationFile"/>
52        </manifest>
53      </jar>
54    </target>
55    <target name="run.server" depends="compile,server">
56      <!-- <exec dir="${dist}" executable="rmiregistry" os="Linux"/> -->
57      <java dir="${dist}" jar="${jar.server}.jar" fork="true">
58        <jvmarg line="-Djava.rmi.server.codebase=file:${jar.server}.jar"/>
59      </java>
60    </target>
61    <target name="run.permutation" depends="compile,permutation">
62      <!-- <exec dir="${dist}" executable="rmiregistry" os="Linux"/> -->
63      <java dir="${dist}" jar="${jar.permutation}.jar" fork="true">
64        <arg line="20 8422"/>
65      </java>
66    </target>
67    <target name="run.client" depends="compile,client">
68      <exec dir="${dist}" executable="appletviewer" os="Linux">
69        <arg line="applet.html -J-Djava.security.policy=${dist}/java.policy.applet"/>
70      </exec>
71    </target>
72    <target name="javadoc">
73      <javadoc destdir="doc" defaultexcludes="yes" author="true" use="true"
            windowtitle="Java security mechanism">
74        <fileset dir="." defaultexcludes="yes">
75          <include name="dk/phiber/tamperproofing/**"/>
76        </fileset>
77        <link href="http://java.sun.com/javase/6/docs/api/"/>
78      </javadoc>
79    </target>
80  </project>
```

Listing C.1: The Ant build file used for building and running the prototype.

Appendix D

Applet Viewing

The HTML file used to load applet into Sun AppletViewer can be seen in Listing D.1.

```
1 <html>
2 <body>
3 <applet code=dk.phiber.tamperproofing.client.LoadApplet.class
4 archive="client.jar" width="200" height="200" >
5 </applet>
6 </body>
7 </html>
```

Listing D.1: The HTML file that is used to load the applet into Sun AppletViewer.

The policy file used by Sun AppletViewer to load the client can be seen in Listing D.2. The reason this policy file is used is to allow the client to overload the applet class loader with the custom class loader.

```
1 /* AUTOMATICALLY GENERATED ON Tue Apr 16 17:20:59 EDT 2002*/
2 /* DO NOT EDIT */
3
4 grant {
5   permission java.security.AllPermission;
6 };
```

Listing D.2: The policy file used by Sun AppletViewer to load the client.

Appendix E

Server Key Files

The byte array operations file used in the prototype is seen in Listing E.1.

```
0 3353 58
0 7013 -26
1 3436 6783
0 5436 -85
0 387 100
0 4814 -59
0 142 -29
0 5055 119
0 4980 120
```

Listing E.1: The byte array operations file used in the prototype.

The CAC key file used in the prototype is seen in Listing E.2.

```
52
69
```

Listing E.2: The CAC key file used in the prototype.

Appendix F

The Test Extraction Scripts

This script Listing F.1 is responsible for calling all the other scripts listed below:

- `disassembler.sh` shown in Listing F.2
- `searchdisassembledfiles.sh` shown in Listing F.3
- `cleanlog.sh` shown in Listing F.5
- `constructcalllists.sh` shown in Listing F.6
- `cleancalllists.sh` shown in Listing F.7
- `classload.sh` shown in Listing F.4

```
#!/bin/sh

./disassembler.sh

./searchdisassembledfiles.sh > search.log
./cleanlog.sh search.log

./constructcalllists.sh > calllists.log
./cleancalllists.sh calllists.log

./classload.sh classloading.log
```

Listing F.1: The script that combine all the other scripts.

```
#!/bin/sh

CLASS="dist/"
DIRECTORY="disassembler"
mkdir -p $DIRECTORY
for i in Client CustomClassLoader InputForm InputForm\$ClearButtonListener
    InputForm\$SubmitButtonListener LoadApplet
do
javap -c -classpath $CLASS dk.phiber.tamperproofing.client.$i > $DIRECTORY/$i.dis
done
for i in ByteArray ChallengeResponse CSPRBG Hashing
do
javap -c -classpath $CLASS dk.phiber.tamperproofing.common.$i > $DIRECTORY/$i.dis
```

```
13 done
```

Listing F.2: The script that disassembles all class files.

```
1  #!/bin/sh
2
3  cd disassembler
4
5  for i in Client.dis CustomClassLoader.dis InputForm.dis InputForm\
       $ClearButtonListener.dis InputForm\$SubmitButtonListener.dis LoadApplet.dis
       ByteArray.dis ChallengeResponse.dis CSPRBG.dis Hashing.dis
6  do
7  echo ##############################################
8  echo $i
9  echo ##############################################
10 awk '/(MemStorage.?loadedClasses)|((public)|(private)|(protected))/,/(bipush|baload
       |iload)|(Code:)/ {print $0}' $i
11 done
```

Listing F.3: The script that searches all disassembled class files for byte array operations.

```
1  #!/bin/sh
2
3  cd disassembler
4
5  for i in Client.dis CustomClassLoader.dis InputForm.dis InputForm\
       $ClearButtonListener.dis InputForm\$SubmitButtonListener.dis LoadApplet.dis
       ByteArray.dis ChallengeResponse.dis CSPRBG.dis Hashing.dis
6  do
7  echo $i >> ../tmp.txt
8  sed 's/^[0-9]*.*ldc\(.*dk\.phiber\.tamperproofing\..*\..*\)$/hhhhhhh\1hhhhhhh/' <
       $i >> ../tmp.txt
9  done
10
11 awk '/hhhhhhh/,/hhhhhhh/ {print $0}' ../tmp.txt > ../$1
12 sed 's/hhhhhhh\(.*\)hhhhhhh$/\1/' < ../$1 > ../tmp.txt
13
14 cat ../tmp.txt > ../$1
```

Listing F.4: The scripts that extracts the class loading order done by `CustomClassLoader`.

```
1  #!/bin/sh
2
3  sed 's/^[0-9]*.*getfield.*//' < $1 > clean.tmp
4  sed 's/^[0-9]*.*invoke.*//' < clean.tmp > $1
5  sed 's/^[0-9]*.*baload.*//' < $1 > clean.tmp
6  sed 's/^[0-9]*.*iload.*//' < clean.tmp > $1
7  sed '/^$/d' < $1 > clean.tmp
8  sed 's/^[0-9]*.*aload.*//' < clean.tmp > $1
9
10 #cat clean.tmp > $1
11 rm clean.tmp
```

Listing F.5: The script that clean the output from the searchdisassembledfiles.sh script for garbage information.

```
1  #!/bin/sh
2
3  cd disassembler
4
5  for i in Client.dis CustomClassLoader.dis InputForm.dis InputForm\
       $ClearButtonListener.dis InputForm\$SubmitButtonListener.dis LoadApplet.dis
       ByteArray.dis ChallengeResponse.dis CSPRBG.dis Hashing.dis
6  do
```

```
7 echo $i
8 awk '/((public)|(private)|(protected))|(\/\/Method)/,/($)/ {print $0}' $i
9 done
```

Listing F.6: The script that construct method call lists foreach method.

```
1 #!/bin/sh
2
3 sed 's/^[0-9]*.*java.*//' < $1 > clean.tmp
4 sed '/^$/d' < clean.tmp > $1
5 sed 's/\(^[0-9]*.*dk\/phiber\/tamperproofing\/\)\(.*\)/\2/' < $1 > clean.tmp
6
7 cat clean.tmp > $1
8 rm clean.tmp
```

Listing F.7: The scripts that clean the output from the constructcalllists script for garbage information.

Appendix G

Test Results

These are the log files calllists.log Listing G.1, search.log Listing G.2, and classloading.log Listing G.3 that the test script produces.

```
Client.dis
client/MemStorage;
public byte[] getSeed();
public boolean putChecksum(byte[]);
public boolean putData(int, byte[]);
CustomClassLoader.dis
client/MemStorage;
InputForm.dis
public dk.phiber.tamperproofing.client.InputForm();
client/Client.getSeed:()[B
client/MemStorage;
client/InputForm;)V
client/InputForm;)V
client/MemStorage;
client/MemStorage;
client/MemStorage;
client/MemStorage;
client/MemStorage;
common/Hashing."<init>":()V
common/Hashing.computeChecksum:([B[B)[B
common/Hashing."<init>":()V
client/MemStorage;
common/Hashing.computeChecksum:([B[B)[B
client/Client.putChecksum:([B)Z
client/MemStorage;
InputForm$ClearButtonListener.dis
InputForm$SubmitButtonListener.dis
   5: invokespecial #3; //Method value:()[B
common/Hashing."<init>":()V
common/Hashing.getB:()[B
common/Hashing.computeHMAC:([B[B)[B
common/Hashing.computeHMAC:([B[B)[B
client/InputForm;)[B
common/ByteArray.xor:([B[B)[B
client/Client;
client/Client.putData:(I[B)Z
LoadApplet.dis
public dk.phiber.tamperproofing.client.LoadApplet();
public void init();
public void stop();
public void destroy();
ByteArray.dis
public dk.phiber.tamperproofing.common.ByteArray();
public static byte[] and(byte[], byte[]);
```

```
45 public static byte[] xor(byte[], byte[]);
46 client/MemStorage;
47 client/MemStorage;
48 public static byte[] append(byte[], byte[]);
49 ChallengeResponse.dis
50 CSPRBG.dis
51 public dk.phiber.tamperproofing.common.CSPRBG(byte[]);
52 client/MemStorage;
53 public byte generateByte();
54 Hashing.dis
55 public dk.phiber.tamperproofing.common.Hashing();
56 public byte[] computeHMAC(byte[], byte[]);
57 client/MemStorage;
58 common/ByteArray.xor:([B[B)[B
59    65:  invokespecial #12; //Method computeHash:([B)[B
60 common/ByteArray.xor:([B[B)[B
61    74:  invokespecial #12; //Method computeHash:([B)[B
62 common/ByteArray.append:([B[B)[B
63 common/ByteArray.append:([B[B)[B
64    84:  invokespecial #12; //Method computeHash:([B)[B
65 public byte[] computeChecksum(byte[], byte[]);
66 client/MemStorage;
67 client/MemStorage;
68 client/MemStorage;
69 client/MemStorage;
70 client/MemStorage;
71    80:  invokespecial #21; //Method computePRMTBlock:([B[BI)[B
72    103: invokespecial #21; //Method computePRMTBlock:([B[BI)[B
73 common/ByteArray.append:([B[B)[B
74    121: invokevirtual #22; //Method computeHMAC:([B[B)[B
75 public byte[] getB();
```

Listing G.1: This is the extracted method call lists.

```
1  Client.dis
2  public class dk.phiber.tamperproofing.client.Client extends java.lang.Object{
3  dk.phiber.tamperproofing.common.ChallengeResponse stub;
4  public dk.phiber.tamperproofing.client.Client(java.lang.String);
5    Code:
6     15:  sipush  4814
7     18:  bipush  -59
8  public byte[] getSeed();
9    Code:
10 public boolean putChecksum(byte[]);
11   Code:
12 public boolean putData(int, byte[]);
13   Code:
14 CustomClassLoader.dis
15 public class dk.phiber.tamperproofing.client.CustomClassLoader extends java.lang.
       ClassLoader{
16 public dk.phiber.tamperproofing.client.CustomClassLoader()   throws java.io.
       FileNotFoundException;
17   Code:
18 public dk.phiber.tamperproofing.client.CustomClassLoader(java.lang.ClassLoader,
       java.lang.String)   throws java.io.FileNotFoundException;
19   Code:
20 public java.lang.Class findClass(java.lang.String)   throws java.lang.
       ClassNotFoundException;
21   Code:
22 public byte[] findClassBytes(java.lang.String);
23   Code:
24
25
26 InputForm.dis
27 public class dk.phiber.tamperproofing.client.InputForm extends java.lang.Object{
28 public dk.phiber.tamperproofing.client.InputForm();
29   Code:
30 public void run(javax.swing.JApplet);
31   Code:
```

```
32     117: sipush  5436
33     120: bipush  -85
34     277: sipush  387
35     280: bipush  100
36     451: sipush  3436
37     463: sipush  3436
38     472: sipush  6783
39     483: sipush  6783
40     566: putfield  #2; //Field checksum:[B
41
42
43     580: pop
44     587: sipush  3354
45     590: bipush  78
46  InputForm$ClearButtonListener.dis
47  public void actionPerformed(java.awt.event.ActionEvent);
48    Code:
49  InputForm$SubmitButtonListener.dis
50  public void actionPerformed(java.awt.event.ActionEvent);
51    Code:
52  LoadApplet.dis
53  public class dk.phiber.tamperproofing.client.LoadApplet extends javax.swing.JApplet
        {
54  public dk.phiber.tamperproofing.client.LoadApplet();
55    Code:
56  public void init();
57    Code:
58  public void stop();
59    Code:
60  public void destroy();
61    Code:
62  ByteArray.dis
63  public class dk.phiber.tamperproofing.common.ByteArray extends java.lang.Object{
64  public dk.phiber.tamperproofing.common.ByteArray();
65    Code:
66  public static byte[] and(byte[], byte[]);
67    Code:
68  public static byte[] xor(byte[], byte[]);
69    Code:
70     8: sipush  3353
71     11:  bipush  58
72     167: sipush  7013
73     170: bipush  -26
74  public static byte[] append(byte[], byte[]);
75    Code:
76  ChallengeResponse.dis
77  public interface dk.phiber.tamperproofing.common.ChallengeResponse extends java.rmi
        .Remote{
78  public abstract byte[] getSeed()   throws java.rmi.RemoteException;
79  public abstract boolean putChecksum(byte[])   throws java.rmi.RemoteException;
80  public abstract boolean putData(int, byte[])   throws java.rmi.RemoteException;
81  }
82  CSPRBG.dis
83  public class dk.phiber.tamperproofing.common.CSPRBG extends java.lang.Object{
84  public dk.phiber.tamperproofing.common.CSPRBG(byte[]);
85    Code:
86     25:  sipush  4980
87     28:  bipush  120
88  public byte generateByte();
89    Code:
90  Hashing.dis
91  public class dk.phiber.tamperproofing.common.Hashing extends java.lang.Object{
92  public dk.phiber.tamperproofing.common.Hashing();
93    Code:
94  public byte[] computeHMAC(byte[], byte[]);
95    Code:
96     19:  sipush  5055
97     22:  bipush  119
98  public byte[] computeChecksum(byte[], byte[]);
99    Code:
```

```
100     23:  sipush  4452
101     35:  sipush  4452
102     44:  sipush  7529
103     55:  sipush  7529
104     70:  sipush  142
105     73:  bipush  -29
106  public byte[] getB();
107    Code:
```

Listing G.2: This is the extracted byte array operations from the client program.

```
1    #8; //String dk.phiber.tamperproofing.client.InputForm$ClearButtonListener
2    #10; //String dk.phiber.tamperproofing.client.InputForm$SubmitButtonListener
3    #11; //String dk.phiber.tamperproofing.client.Client
4    #12; //String dk.phiber.tamperproofing.common.Hashing
5    #13; //String dk.phiber.tamperproofing.common.CSPRBG
6    #14; //String dk.phiber.tamperproofing.common.ChallengeResponse
7    #15; //String dk.phiber.tamperproofing.common.ByteArray
8    #9; //String dk.phiber.tamperproofing.client.InputForm
```

Listing G.3: This is the extracted client classes loading order.