Implementation and evaluation of security protocols in e-commerce applications

by

Hugh Davenport

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Engineering
in Software Engineering.

Victoria University of Wellington
2013

Abstract

There are a large amount of programs in the development process in todays technology environment, and many of these involve some type of security needs. These needs are usually not dealt with in a sensible way and some even don't bother with any analysis. This thesis describes a solution of implementing a secure protocol, and gives an evaluation of the process along with the techniques and tools to aid a secure design and implementation process. This allows others to take this knowledge into account when building other applications which have a need for security development.

Acknowledgments

I would like to acknowledge my supervisors Dr. Kris Bubendorfer, and Dr. Ian Welch. They gave me the motivation to finish this thesis after a long break due to full time work.

I would also like to acknowledge my office mate, Ben Palmer, who researched the protocol that I am implementing for this Masters thesis.

I would like to acknowledge the graduate students in the School of Engineering and Computer Science at Victoria University of Wellington. Especially those who came from the room called "Memphis". They were great company while working on this thesis, and will remain good friends in working life.

I would like to say thanks to my friends and family for sticking by me while I was working on this, and also for motivating me to finish after the long break. I know it can be difficult to enjoy moments when I am stuck at a computer working.

I would also like to acknowledge my employers, who allowed me to take a few weeks off to finish this thesis, even though I was very busy at the time.

Most of all, I acknowledge lolly snake (shown in figure 1), without which this thesis could not of been completed.

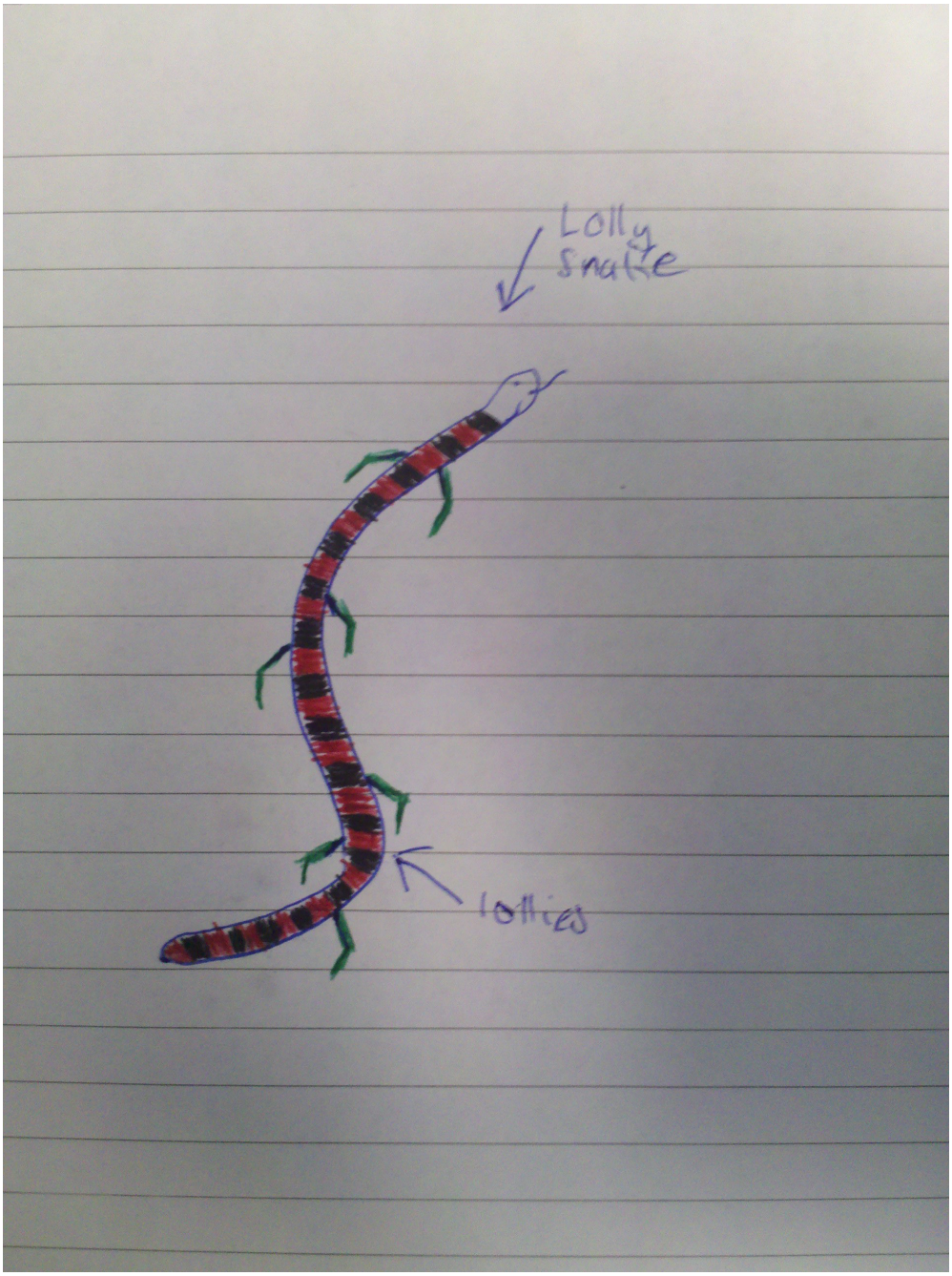

Figure 1: Picture of a Lolly Snake. Licensed under CC-BY-SA. © Sam Bonner. Valued at 437.25 words.

Contents

List of Figures

Chapter 1

Introduction

1.1 E-commerce

E-commerce security traditionally concerns itself with the authenticity of parties involved and the integrity of the data exchanged.

Authen:w ticity is achieved through SSL certificates over the HTTP protocol (making it the HTTPS protocol) [49, 50, 11, 12, 13]. This allows the client to know that it is talking to the server it is thinking it is talking to. There are down-sides to the HTTPS scheme, one example is how to determine the trustworthiness of a given certificate.

Integrity of the data on the server side is normally handled through back-end security measures.

As e-commerce moves from single client-seller models to situations where chains of supplier and resellers are used, additional security measures are required. In particular, provenance.

1.2 Tagged Transaction Protocol

In contrast, the Tagged Transaction Protocol [66] concentrates on the digital provenance of products. The protocol allows for an honest customer to verify that the product they have purchased is from the authentic supplier regardless of any resellers who have handled the product. The protocol allows verification that products have been sold only once and that resellers have legitimately obtained the product from the supplier through some reseller chain. This includes any *dishonest* resellers.

The simple approach to this is that the chain of resellers is recorded and the customer checks each transaction to verify its legitimacy. This process does not allow the reseller to be anonymous, and the customer can choose different resellers based on this knowledge for the next purchase, which is not acceptable to most resellers.

The tagged transaction protocol uses a third party, the Tag Generation Centre (TGC), to issue *tags* along with any purchase. This in effect creates an implicit reseller chain for any product, while providing anonymity for the resellers in the chain. The TGC can also be used by customers to verify the tag received in a purchase. This verification may only involve the clients checking that the tag is signed by the TGC and holds the correct information, but more detailed verifications could take place, while assuring the anonymity of parties in the chain.

The Tagged Transaction Protocol has been mathematically verified but at the time of this work had not been implemented.

1.3 Research Aims

The aims of this research is to implement a prototype of the security protocol discussed in Chapter 3, and to compare and discuss the different processes for developing secure software and their respective benefits and disadvantages. The comparison helps us determine engineering best practices for implementing security protocols.

1.4 Thesis Outputs

The outputs of this thesis are described in this section. They are described in greater detail in further chapters, and Chapter 7 discusses how they were used to obtain the research aims above.

1.4.1 Prototype of the Security Protocol

The prototype was developed in the Java programming language, using the BouncyCastle cryptographic library [86]. The prototype was implemented using secure design practises in mind, and was made to be extendable for alternative implementations.

1.4.2 Comparison of Secure Development Processes

A comparison between different secure design approaches is made. Designs such as CLASP (Comprehensive, Lightweight Application Security Process) [65], TSP-Secure (Team Software Process - Secure) [81], SDL (Secure Development Lifecycle) [30], and SecSDM (Secure Software Develop-

ment Methodology) [23] are presented, along with advantages and disadvantages between them.

In addition to complete processes for developing secure software, a selection of additional tools or partial processes is presented with a discussion on how they can fit into the entire development process.

The SecSDM approach was then used to show how to validate the implementation of the prototype against security requirements. This approach was then compared with an intuitive approach of implementation and a discussion on the advantages and disadvantages of each approach is made.

1.4.3 Performance Evaluation of the Prototype

The performance measure of the protocol is carried out to gain knowledge on where bottle-necks lie, and how long common tasks in the protocol take.

The measurements took the total time of the protocol running through a test process, and also the time taken for each of the components. The evaluation of these results is discussed about usability and possible further testing.

1.5 Structure of Thesis

This thesis describes the process of developing an implementation of a secure protocol. The main aspects of the thesis is to evaluate between different processes that could possibly be used when designing and implementing secure software.

The thesis will be split into several chapters which outline different parts of the process and background material. A short outline of the chapters is as follows:

This chapter introduces the thesis, gave some brief background material, and some motivation for the research. An introduction of the research in question was given, along with the aims of the research and the thesis outputs.

Chapter 2 covers a more detailed description of the Tagged Transaction Protocol, as well as a survey of related work.

Chapter 3 expands on the background to the security protocol that will be implemented. Points that will be covered include: Digital provenance; Cryptographic functions and libraries; Secure development techniques; and other minor points.

Chapter 4 describes the design and implementation process of the security protocol. This includes any reasoning behind each decision in the process.

Chapter 5 outlines the verification and validation techniques used to show security and validity of the implementation against the specifications of the protocol.

Chapter 6 presents an analysis of the performance of the implementation of the protocol. This includes an discussion on what the results represent.

Chapter 7 summarises this thesis.

Chapter 2

Related Work

This chapter outlines the protocol used for this project followed by some related works. Section 1.2 in chapter one gives motivation for the protocol mentioned here.

The chapter will be laid out as follows:

- Section 2.1 presents the tagged transaction protocol.
- Section 2.2 presents a selection of related implementations.

2.1 Tagged Transaction Protocol

This section presents the background material for the security protocol that has been implemented for this project. The protocol is called the *Tagged Transaction Protocol* [66].

- Section 1.2 in Chapter 1 discussed motivation for the protocol
- Section 2.1.1 will present a model of the relationship domain used in the protocol

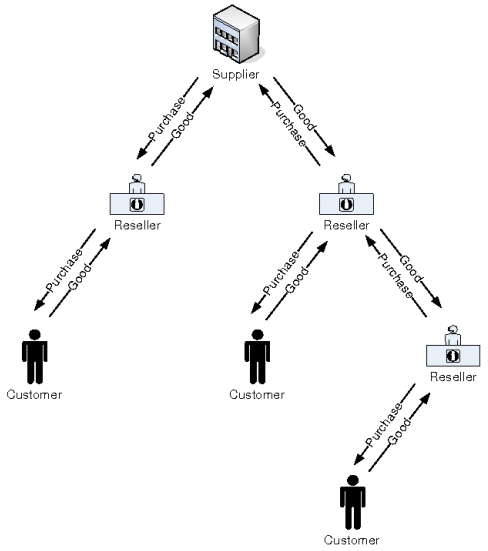

Figure 2.1: Relationships of the Tagged Transaction Protocol [66]

- Section 2.1.2 will present a model of the security domain used in the protocol
- Section 2.1.3 will discuss possible threats to the model of the protocol
- Section 2.1.4 will present the details of how the protocol works

2.1.1 Domain Model

Figure 2.1 shows the basic domain model for the protocol [66]. The protocol participants include a supplier, a chain of one or more resellers, and a customer. These roles in detail are:

Supplier

Party that originally creates the digital product and hold all rights for the item.

Reseller

Party that purchases the product of any other reseller, and sells the product to any customer (which in turn could be a reseller).

Customer

Party that is interested in purchasing the digital product of a reseller and may become a reseller.

2.1.2 Security Model

Malicious participants could be a possible threat to this domain They may try and break the anonymity of the system. Possible actions malicious resellers could take are shown below.

The customer will wish to verify that the first two actions have not taken place, but the system will also need to protect them from the third action. If a customer learns the identity of a reseller further up the chain, they may *take out the middleman* to get a better deal. The protocol has an option to provide anonymity.

Spoofing

Claiming to be a supplier, or attempting to subvert protocol to appear to be the supplier.

Counterfeiting

Selling a digital product that the reseller has not purchased. This could include any of the following:

- Fabrication: Attempting to build a tag from scratch (after possibly seeing the structure of the tag)
- Cloning: Trying to resell a product multiple times.
- Network Sniffing: Copying a legitimate licence from the network and selling to customer.

Identity Revelation

The customer learns the identity of resellers (or possibly the supplier) that are not neighbours in the chain.

2.1.3 Threat Model

It is assumed that any malicious participant is polynomial bounded. It is also assumed that the reseller of product x will not collude with the supplier of product x, but may collude with any other supplier to try and convince the customer that the other supplier is the supplier of x. It is also assumed that the customer does not collude with the reseller.

These assumptions do not imply that both the customer and the supplier are *honest*. The supplier may try and find the identity of the customer and vice versa, while being polynomial bound.

There may also be a third party member that tries to impersonate a reseller r and act in such a way that the protocol fails and discredits r.

2.1.4 Protocol Details

This section outlines the details of the protocol.

- Introduce any definitions of global variables, and the tag structure.
- Outline the process of registering a product in the system.
- Outline the process of the supplier generating a tag to send to a reseller.
- Outline the process of a reseller generating a tag to send to a customer or another reseller.

Definitions

This section defines the constructs used in the protocol, then describes the details of each component of the protocol and how it fits together.

The protocol requires encryption, digital signatures and a secure message digest. The background to these constructs is presented in Section 3.1.

The original protocol description specifies a modified elgamal algorithm for signing [68]. This protocol could be replaced with any digital signing system that is secure against existential forgeries in the adaptive chosen message attack mode.

In this implementation, the elgamal encryption scheme is used for encrypting data [18], and the secure hash (SHA) family of secure message digests is used for message digests [54]. More details of the actual implementation of the protocol is described in Chapter 4.

The modified elgamal signature scheme is used because it has been proved secure in the random oracle model against existential forgeries in adaptive

chosen message attacks [68]. The differences between this modified form and the original signing scheme are described below.

For signing and encrypting data, the following notation is used throughout this thesis. For signing data D with a private key sk, the notation for the signed content is $\{D\}_{sk}$. For encryption data D for the owner of the public key pk, the notation for the encrypted data is $\{D\}_{pk}$.

For all the key generation, signing, verification, encryption and decryption operations, the mathematical computations are calculated modulo a large prime p in the integer group $\mathbb{Z}_p$ closed under multiplications unless specified otherwise.

The keys described in this document use the following notation. For public keys, pk will be used, and for private keys, sk will be used. The relationship between the public and private keys is $pk = g^{sk}$ mod p, where g is a generator for the group $\mathbb{Z}_p$.

For the elgamal algorithm, there are the parameters p, q, and g. The p and g values are described above. The q value is a large prime such that $q|p-1$ (q divides $p-1$).

The TGC will have a key pair that is used both for signing and encrypting, and will be referred to as pk_{TGC} and sk_{TGC} for the public and private key respectively. The supplier will generate a key pair for each item supplied x, and these will be referred to as pk_x and sk_x. The reseller r will generate a one time key pair for each tag it requests (which it may or may not receive), and is referred to as $pk_{tag,r}$ and $sk_{tag,r}$.

The reseller has a separate one time key pair to prevent the TGC (or other parties) being able to link multiple transactions to a single reseller. If a single key pair were used, the TGC will be able to use a simple one to one map between public key and reseller.

The protocol uses a zero knowledge proof of knowledge of the one time

resellers private key $sk_{tag,r}$. The details of this proof are described in detail in the generation of the reseller tags. A commitment value is used by the reseller along with the public key $pk_{tag,r}$. This commitment value uses the notation $a_r = g^{z_r} \bmod p$, where z_r is a random private value chosen by the reseller.

The tags in the system have the following outline. The tag contains the following four values $\{A = pk_x, B = L_x, C = pk_{tag,r} = g^{sk_{tag,r}} \bmod p, D = a_r = g^{z_r} \bmod p\}$. The values have the following meanings:

- $A = pk_x$ is the public key of the product x, which is generated by the supplier of product x during the registration of product x.
- $B = L_x$ is the licence for the product x, which is generated by the supplier of product x, when the original tag in this chain is generated.
- $C = pk_{tag,r}$ is the one time reseller public key for this tag. It is related to the private key as shown above.
- $D = a_r$ is the commitment value used by the reseller for the zero knowledge proof of the private key $sk_{tag,r}$.

The licence L_x has the following outline. The licence contains the following three values $L_x = \{id = H(x), tagno, Licence\}_{sk_x}$. The values have the following meanings:

- $id = H(x)$ is the id of the item x, generated by the hash function H.
- $tagno$ is a unique field that is used to detect replay.
- $Licence$ is a supplier dependant licence that is sent to the customer. The implementation details of this licence are not specified in the protocol.

The specific hash function H is not specified in the protocol and would

be implementation specific. This implementation uses the SHA family of message digest functions [54]. The implementation will be described in greater detail in Chapter 4. The licence L_x is signed by the private key sk_x of the product x.

Some communications may be made over an anonymous channel. These communications can be made anonymous by using an onion routing service [82] such as TOR [15]. The use of any anonymity is not specified in the protocol, and is therefore implementation specific.

The communications that are be made anonymous are be the communications between the supplier and the TGC, and any reseller and the TGC. This anonymity prevents the TGC from knowing the identity of the supplier or the reseller. The optional anonymous channels are shown as dotted lines in Figures 2.2 and 2.3.

Registration of Products

The registration phase of the protocol is described in this section. Registration is the process by which supplier notifies the TGC that they have a product they wish to supply using the protocol. The TGC can then perform an out of band check to validate that the supplier is authorised to supply that product. The TGC can also choose to use a first in first served for registration. If anonymous communications are used, the TGC *must* follow a first in first registered approach as they can not verify an identity they do not know.

This phase of the protocol only involves the supplier and TGC entities. The reseller or the customer has no interaction in this phase.

The setup for this phase is for the supplier to calculate the id of the item as $id = H(x)$, where H is a implementation specific hash function, then the

supplier must generate a key pair pk_x and sk_x for the item x. These keys use the modified elgamal algorithm as described above.

The supplier then sends an encrypted registration request to the TGC containing the values of the id and the public key for the item x. The TGC then performs any checks necessary, and if the registration succeeds a signed receipt is sent back to the supplier. This receipt contains the id and the public key, and is signed by the TGC private key sk_{TGC}. This gives the supplier proof that the product x was registered for them, in case a rogue TGC does not follow the specifications correctly.

The registration request sent to the TGC is as follows:

$$\{id = H(x), pk_x\}_{pk_{TGC}}$$

At this point, the supplier has successfully registered the item x. If any problems have occurred, then the supplier will have to try the registration process again, though if an item with the same hash has already been registered this will always fail.

Supplier Generating Tags

This section details the process of a supplier generating a tag for the protocol. This situation occurs when a reseller requests purchase of an item x that the supplier has registered with the TGC.

An overview of this stage is shown in Figure 2.2, which was displayed in the original paper describing the protocol.

A reseller starts the process by sending a purchase request to the supplier. This request contains the hash of the item x as the id ($id = H(x)$), also a one time public key and a commitment value.

The one time public key is the public half of a key pair which the reseller

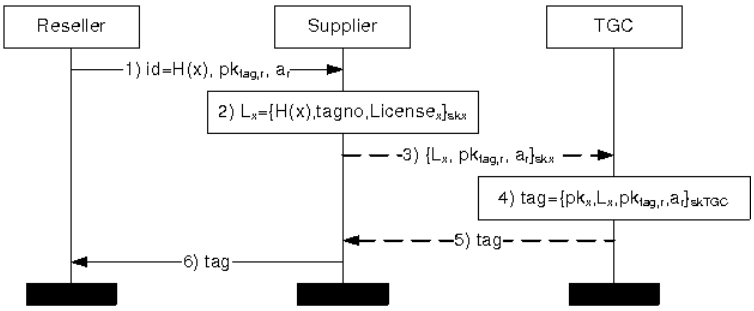

Figure 2.2: Supplier Generating Tag with TGC

must generate for use in the received tag. The key is referred to as $pk_{tag,r}$, while the private half is $sk_{tag,r}$. They relate to each other as per usual, $pk_{tag,r} = g^{sk_{tag,r}} \bmod p$.

The commitment value is generated by the reseller as a commitment token for use in a zero knowledge proof that will be performed when the reseller attempts to generate a tag. The cryptographic section earlier in this chapter (Section 3.1) introduces zero knowledge proofs. The specific details of how the proof is performed is described later in this section when the reseller generated tags are outlined.

The commitment value is related to a private random value z_r, which the reseller must generate. The commitment value is calculated as $a_r = g^{z_r} \bmod p$.

In summary, the request sent by the reseller contains the following:

$$\{id = H(x), pk_{tag,r}, a_r\}.$$

After the supplier receives this purchase request, it creates a tag request to send to the TGC. The tag request contains a licence L_x, and the public key and commitment value sent by the reseller ($pk_{tag,r}$ and a_r). This request is signed by the supplier's private key for item x (sk_x).

The licence construct L_x contains a licence that the end user can use with

the product, along with some fields to aid the protocol. L_x contains the following: $id = H(x)$; $tagno$; and $Licence$. The id is the id of the item being licenced, $tagno$ is a unique field that is used to detect replay, and $Licence$ is an implementation specific licence. L_x is signed by the private key sk_x.

This tag request sent to the TGC is as follows:

$$\{L_x = \{id = H(x), tagno, Licence\}_{sk_x}, pk_{tag,r}, a_r\}_{sk_x}$$

After the TGC receives the tag request it checks that the $tagno$ field is not the same as any other requests for the item x. This protects against replay attacks on the licence, so the supplier must generate a new licence and sign it for each request.

The TGC then creates a tag, signs it, and returns it. This tag is then forwarded on to the reseller that sent the purchase request. The actual item x may also be sent here, but is not specified in the protocol description.

The tag contains the following items:

$$tag = \{pk_x, L_x, pk_{tag,r}, a_r\}_{sk_{TGC}}$$

The reseller should then check that the tag contains the correct information, and the signatures for both the licence and the complete tag are valid and signed by sk_x and sk_{TGC} respectively. This check could also be done by the supplier as well to save extraneous traffic, but is not specified in the protocol.

Reseller Generating Tags

This section describes the specifications for the stage where the reseller attempts to generate a tag based on a purchase request from another reseller or a customer.

This stage involves a zero knowledge proof between the reseller and the

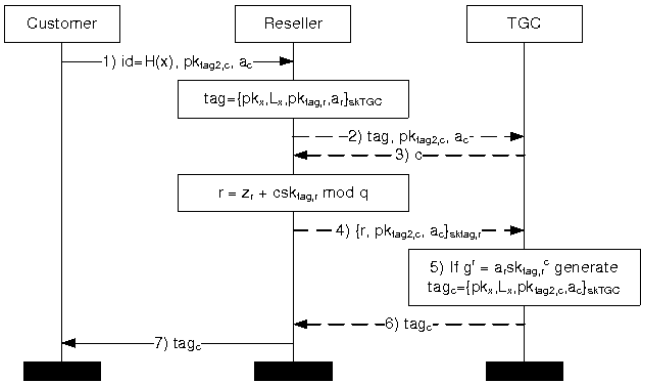

Figure 2.3: Reseller Passing Tag to Customer

TGC. Section 3.1.2 outlines a general description of what a zero knowledge proof is, while this section merely outlines the key points required to understand this protocol.

This zero knowledge proof is used to determine whether the reseller has access to the private key. This stops forging and also can be used to detect replays. Details of this proof will be given as well as an example of replay detection.

An overview of this stage is shown in Figure 2.3, which was taken from the original paper describing the protocol.

This stage occurs when a reseller or a customer (c) sends a purchase request for item x to a reseller (r) that resells item x. The purchase request contains the hash of the item, a one time public key and a commitment value.

The one time public key is the public half of a key pair which the reseller must generate for use in the received tag. The key is referred to as $pk_{tag2,c}$, while the private half is $sk_{tag2,c}$. They relate to each other as per usual, $pk_{tag2,c} = g^{sk_{tag2,c}} \bmod p$.

The commitment value is generated by the reseller/customer as a com-

mitment token for use in a zero knowledge proof that will be performed when the reseller attempts to generate a tag. The cryptographic section described earlier in this chapter (Section 3.1) introduces zero knowledge proofs. The specific details of how the proof is performed is described below.

The commitment value is related to a private random value z_c, which the reseller must generate. The commitment value is calculated as $a_c = g^{z_c} \mod p$.

In summary, the request sent by the reseller contains the following:

$$\{id = H(x), pk_{tag2,c}, a_c\}.$$

After the reseller r receives the request, it then creates a tag generation request to send to the TGC. This tag request contains the original tag received by r when they purchased x (which contains the one time public key and commitment value for r), along with the one time public key and commitment value for the client. The data sent to the TGC is as follows:

$$\{tag = \{pk_x, L_x, pk_{tag,r}, a_r\}_{sk_{TGC}}, pk_{tag2,c}, a_c\}$$

When the TGC receives the request, it checks that the one time public key has not been used before for the $tagno$ field found in L_x. The TGC then initiates a zero knowledge proof of knowledge of a discrete logarithm [76] so that the reseller r proves to the TGC that it has knowledge of the secret key $sk_{tag,r}$ such that $pk_{tag,r} = g^{sk_{tag,r}} \mod p$, using a commitment value $a_r = g_{z_r} \mod p$.

The zero knowledge proof (ZKP) starts by the TGC sending a challenge value c which the TGC generates. The TGC will then expect the response r such that $g^r = a_r pk_{tag,r}^c$.

The reseller calculates $r = z_r + csk_r \mod q$ and sends it to the TGC and with the one time public key and the commitment value of the customer

c, with all the data signed by the one time private key of the reseller. The data sent back to the TGC is as follows:

$$\{r = z_r + csk_r \bmod q, pk_{tag2,c}, a_c\}_{sk_{tag,r}}$$

The TGC then checks that the result is signed, the one time public key and commitment value of the customer are the same, and that the result r is as expected. This proof is demonstrated by the following equations:

$$g^r = g^{z_r + csk_{tag,r} \bmod q} \tag{2.1}$$

$$g^r = g^{z_r} g^{csk_{tag,r}} \tag{2.2}$$

$$g^r = a_r pk_{tag,r}^c \tag{2.3}$$

Equation 2.1 is basic substitution of $r = z_r + csk_{tag,r} \bmod q$. Equation 2.2 uses the power rules to rearrange the equation into two separate powers. Equation 2.3 uses the fact that $a_r = g^{z_r} \bmod p$ and $pk_{tag,r} = g^{sk_{tag,r}} \bmod p$. Equation 2.3 is also the equation that the TGC verifies, and given the previous two equations, it follows that the reseller would give the correct r value only with negligible chance of cheating if they knew the private key $sk_{tag,r}$.

If the proof passes, then the TGC stores the transcript c, r against the item hash ($H(x)$), the public key ($pk_{tag,r}$) and commitment value (a_r). The TGC then issues a tag with the following content:

$$tag = \{pk_x, L_x, pk_{tag2,c}, a_c\}_{sk_{TGC}}$$

The reseller then forwards this on to the customer, who should check that the tag contains the correct information, the licence is signed by sk_x and the tag is signed by sk_{TGC}. This check could also be done by the reseller but is not specified in the protocol.

If the proof fails then the TGC does not do anything, and returns an error result.

If the tag is submitted again to the TGC, then the TGC can detect replay and discover the private key $sk_{tag,r}$. This is done because the TGC will have the first transcript c_1, r_1 already and now should have a new transcript c_2, r_2. Both these transcripts use the same commitment value a_r. The following equations can be used to extract the one time secret key $sk_{tag,r}$.

$$g^{r_1}/g^{r_2} = a_r g^{c_1 sk_{tag,r}}/a_r g^{c_2 sktag,r} = g^{sk_{tag,r}(c_1-c_2)} \tag{2.4}$$

$$g^{r_1-r_2} = g^{sk_{tag,r}(c_1-c_2)} \tag{2.5}$$

$$log_g\ g^{r_1-r_2} = log_g\ g^{sk_{tag,r}(c_1-c_2)} \tag{2.6}$$

$$r_1 - r_2 = sk_{tag,r}(c_1 - c_2) \tag{2.7}$$

$$sk_{tag,r} = (r_1 - r_2)/(c_1 - c_2) \tag{2.8}$$

Equation 2.4 uses the ratio of the two g^r results, expands that out, and cancels out the common a_r term. Equation 2.5 rewrites that result in a form that is ready to take the log. Equation 2.6 shows the log_g of both sides, and Equation 2.7 shows the result after cancelling the log and the exponent. Equation 2.8 rearranges so that $sk_{tag,r}$ is the target. This shows how the TGC can use the two transcripts c_1, r_1 and c_2, r_2 with a common commitment value a_r to determine the secret key $sk_{tag,r}$.

2.2 Related Work

This section presents two security protocols implementations from a selection of many security protocol descriptions. The two presented here were chosen as they were the only ones that focussed on the implementation of a security protocol rather than a description of a security protocol. The two systems are as follows:

- Section 2.2.1 presents the Paradiso system [47, 48]
- Section 2.2.2 presents the Potato system [1, 69]

These systems all allow for the reseller model, where a customer can on-sell a product they purchase of a supplier. This model is similar to how the Tagged Transaction Protocol works. The Tagged Transaction Protocol has the extra benefit of having a verifiable solution, and allowing the supplier and all but the immediate reseller to be anonymous.

2.2.1 Paradiso

The Paradiso system allows customers to purchase reseller rights along with products from the suppliers [47, 48]. This provides the reseller model, and has the benefit that the reseller does not need to contact the supplier after purchasing the reseller rights for a certain number of copies of the product.

Paradiso runs on media devices with a trusted computing module (TCM), which allows enforcing contracts between the supplier and customer. The device can be something like the iPod, or the Zune. The device will have it's own private key stored in the secure hardware of the TCM. All interaction with the Paradiso system, and the content being transferred makes use of the secure hardware of the TCM.

To gain reselling rights, the device sends a request to the supplier along with it's public key. When payment is made for the content, the supplier sends the customer the content and any rights encrypted with a newly created AES key. All of this is encrypted using the customers public key and sent to the customers device. The reselling rights and license is signed by the suppliers private key. To verify the license, the player confirms that it has been signed by a valid suppliers.

The Paradiso system has many appealing features, though it has a disadvantage of relying on the TCM. In an ideal situation, all devices would have this TCM, and this would allow reselling of content without further input from the supplier.

2.2.2 Potato

Potato is a music redistribution system developed by Fraunhofer and the company 4FriendsOnly [1, 69]. Users of the system purchase the content, and also get reseller rights that give them credit when they on-sell.

There is a signed media format (SMF) presented by Nutzel et. al. [26]. This format has all the content encrypted with the AES symmetric encryption scheme. The encryption key is then encrypted by the private key of the last customer. Any further purchases need to decrypt the content by using the public key of the last buyer. The format has the following fields: encrypted media; encrypted AES key; public key of last buyer; signed licence. The licence comes from an accounting server, in this case the Potato system.

Potato allows distribution through P2P networks, but has the disadvantage of requiring interaction with a central web server run by trusted third party. Potato also has an interesting use of a reward system.

Chapter 3

Techniques and Tools

This chapter outlines some techniques and tools that are useful when developing secure software.

The chapter will be laid out as follows:

- Section 3.1 presents a discussion of cryptography
- Section 3.2 discusses a range of different development techniques for secure software.

3.1 Secure Use of Cryptographic Mechanisms

Cryptography is an important underlying mechanism of security engineering. This section aims to present to the reader the background knowledge of cryptography and how it is implemented in security engineering.

Cryptographic functions are the underlying constructs to security protocols, and how strong the functions are how strong the protocol can *ever* be. That is, if a cryptographic function is found to be weak, all protocols

employing this function have a vulnerability due to the use of this function. There are several organisations that evaluate the security of primitive functions and rate how strong they are. As well as that, many individuals who have access to the source for the function can try and exploit it. One process in security engineering is to take all this information and make an informed decision on what functions to use and what not to use, based on what the situation requires.

- Section 3.1.1 presents a discussion on key sizes
- Section 3.1.2 outlines a background to zero knowledge proofs
- Section 3.1.3 presents a discussion on sources of random material, and their importance in cryptography
- Section 3.1.4 presents a selection of cryptographic libraries

3.1.1 Choice of Key Size

Key sizes are an important topic in the field of security. When the key size goes up, the computing power needed to break the cryptography using those keys goes up exponentially. The computation needed to perform the cryptographic functions also goes up exponentially. This means that choosing a key size is a trade-off between time and security.

As an alternate system, elliptic curve keys require smaller key sizes for similar security of larger traditional keys.

Having said all this, the question still arises. "What is the right size key?"

The right key size is really determined by how much security you need versus the computational power. Standard key sizes that have been used in the field are 1024, 2048, and more recently 4098. But which is best? The cryptography using 1024 keys may be broken in years to come, so

may not be the best solution. The 2048 size is a comfortable size for both security and computational power, but how long will that last. The 4098 size provides good security while being slower computationally.

Another aspect that would need to be addressed is how long the keys are intended to be used. If the keys are only intended to be used for a short period, the 2048 size may be the best, as it is unlikely that the key will be broken in that period. But if the keys are intended to be used for a moderate time (say 10–20 years), then it could be possible for the 2048 bit key to be broken in that time. The 4098 size may possibly be the best fit here.

3.1.2 Zero Knowledge Proof

This section describes the idea of a zero knowledge proof. A zero knowledge proof is when a verifier (with no knowledge of the secret) verifies that another party knows the secret value. The secret value here can be basically anything, but in cryptography it usually has some mathematical background.

Zero knowledge proofs were first presented in 1985 by Goldwasser et. al. [25]. Since then many different variations for different proofs have been researched. This section will not describe the many different types, just a brief overview on how a zero knowledge proof works. A specific proof was described in Section 2.1.4 when the protocol specifications for this thesis was outlined.

A proof must satisfy three properties: Completeness; Soundness; and Zero-knowledge.

Completeness is satisfied if an honest verifier can be convinced that an honest prover knows the secret.

Soundness requires that no dishonest prover can convince an honest verifier, except for a small probability of error.

Zero-knowledge requires that no dishonest verifier can learn more than the fact that the prover knows the secret.

A small example in an abstract setting is described below, first published as an example of how to explain zero knowledge proof to your children (laypeople) [70]. In this example there are two parties, Peggy (the prover), and Victor (the verifier). Peggy has the knowledge of a secret word that will open a door in a cave shaped like a circle. This cave has only one entrance. There are two paths leading to the hidden door, and to pass through the door the secret word must be uttered.

Victor wants to verify that Peggy does indeed know the secret word, but Peggy does not want to tell him outright. So they come up with the following way to prove that she knows the word. Victor waits outside the cave and Peggy goes in, randomly choosing one of the two paths (Victor can not see which path she takes). Victor then goes into the cave so he can see the two paths, and calls out to Peggy to come out on the path that he chooses.

At this point there is a 50% chance that Peggy went down the path that Victor specified and can come back without using the secret word. On the other hand, there is a 50% chance that she needs to utter the secret word to come back along the correct path. To be certain, Victor could test Peggy as many times as he wants (or until she gets tired of walking back and forth), and if she never gets it wrong, chances are that she knows the secret word (the *proof* part of the zero knowledge proof). There is still a small probability that she guesses correctly each time, but this probability (or soundness error) can be reduced exponentially by more tests. Note that Victor still does not know the secret word, hence the *zero knowledge* part of a zero knowledge proof.

3.1.3 Randomness

This section presents a brief discussion on random data. The field of randomness relates back to the field of mathematics, and is an important part of cryptography. Note that a definition of randomness is described in the next subsection.

Without random data, most of the assumptions made in cryptography would be null and void. For example, if no random data was used to create a key pair, with the same seed value an adversary can create *the exact same key pair*. This is not desirable.

With some extra randomness, it is hard to create the exact same key pair. If the random data is insufficient, then statistical analysis can be performed to attempt to generate the same key pair given only the public key and/or the seed value. This shows a good motivation on why randomness is important in cryptography. This is one example of motivation, but the same argument could be applied to more areas of cryptography.

This section will give a brief introduction to the idea of randomness, then go on to talk about sources of random material (and how they may be built into the operating system or as a userspace program), and finally a brief discussion of online sources of random material and whether they are suitable for cryptographic purposes.

What is Randomness?

Kolmogorov defines a sequence of bits as random if and only if the length of the sequence is shorter than any algorithm that could produce the sequence [39]. This definition ensures that a random sequence can not be compressed, which implies that there can be no statistical measures to deduce any bits in the sequence.

Randomness can be tested for sufficient randomness and some tests for this are outlined in RFC 4086 [17] and FIPS 140-2 [53].

Randomness can be split into two major different variations: true randomness (which includes environmental randomness and sources like dice); and pseudo randomness (generated by an algorithm).

True randomness includes sources such as any environmental noise for example: atmospheric noise, or thermal noise from resistors. It also includes physical systems such as dice, or roulette wheels, as long as these systems are *fair*. This source of randomness can be obtained with purpose made hardware random number generators, or manually by rolling a dice.

Pseudo randomness is a type of randomness that is generated by a computer algorithm. The rest of this section will outline a few algorithms that can be used. The algorithms presented below are a specific type of pseudo random number generator algorithms designed for cryptography. These types of generators are called Cryptographically-Secure-Pseudo Random Number Generators (CSPRNG).

Randomness Sources in Major Operating Systems

First of all, the source of randomness for each of the major operating systems should be mentioned.

BSD systems and derivatives such as Mac OS X use a system called Yarrow [38] which combines multiple sources of randomness by determining the quality of the inputs. A successor to the Yarrow algorithm is Fortuna [20].

Linux uses a system that provides randomness from any available random source, not limited to but can include hard disk access among other things [28]. Linux's system can also take advantage of any connected hardware random sources.

Microsoft systems use the Microsoft Cryptographic Application Programming Interface, specifically the method CryptGenRandom [44].

Userspace Sources of Randomness

Aside from support in the operating systems, there are also userspace tools that can provide pseudo randomness. These systems include EGD, prngd, HAVEGE, randomsound, rng-tools.

EGD: The Entropy Gathering Daemon is a daemon for systems that do not have a source of randomness in the operating system [89]. It works by running userspace commands to collect various sources of unpredictability from the underlying system.

Pseudo Random Number Generator Daemon (prngd) is an EGD compatible program [35]. prngd differs from EGD in that it is a non blocking program that seeds the random number generator in the OpenSSL library.

HArdware Volatile Entropy Gathering and Expansion (HAVEGE) is a tool that takes the unpredictability of the underlying system as a source of randomness [78]. The throughput of the HAVEGE algorithm is significantly faster than the underlying systems source of randomness, and can also act as an additional source of randomness for those systems, which makes it very useful for cryptographic systems that require a large supply of random data quickly.

Randomsound is a utility that uses the Linux sound system ALSA to gather randomness [79]. It works by taking the low order bit of the ADC output in the sound card, removing any biases and adds it to the Linux random pool /dev/random.

The program rng-tools is an additional tool used by the Linux random pool to obtain randomness from external sources. It allows using external

sources such as a hardware random number generator to add randomness to the internal pool used by /dev/random.

Each of these tools needs to take into consideration how they collect the random entropy, and how they may use mathematics to extend the size of the entropy.

It should be noted that the process of extending the entropy makes the source of randomness less reliable as it gets further from the definition of true random. This means that these techniques are used as a trade-off between security and time (or entropy size, which contributes to the major time computation in cryptography).

This project uses the tool HAVEGE as the source of randomness is quickly obtained. The results in chapter 6 show the difference in timing between running the protocol not using HAVEGE, and running the protocol while using HAVEGE.

Online Sources of Randomness

As a final note, a mention of online sources such as `random.org` and `rand.org`. These sources may not be as secure for cryptographic purposes unless encryption is placed around the traffic received from the site.

3.1.4 Cryptographic Libraries

There are a multitude of libraries that implement the major algorithms for cryptographic use. It is best practice to use a library that has already implemented an algorithm rather than implementing it from scratch for a new project. This saves your implementation from having unnecessary bugs that the libraries that are developed by a wider group of developers don't have.

These libraries may have limitations such as language bindings, or algorithms implemented. The choice of a particular library to use would come down to how *trusted* is it in the security field?, does it implement the needed algorithm?, and does it have bindings for the language used?

Note that the language used may be determined by the library choice or vice versa.

This section will not detail any of the libraries or constraints they may have, but is purely a list of some well known libraries.

These cryptographic libraries (in no particular order) include: OpenSSL [87], GNU crypto [84], bouncycastle [86], cryptlib [27], gnutls [85], beecrypt [91], gcrypt [83], nettle [45], polarssl [55].

This list does not include any cryptographic libraries built into the language itself.

The implementation of this project used the bouncycastle cryptographic library.

3.1.5 Summary

This section presents a summary of the techniques discussed in this chapter. The main content of this summary will focus on how the techniques can relate to the implementation of the protocol described in Section 2.1.

The actual description of the implementation is not covered until chapter 4, but this section outlines the details that would need a decision on the specifications of the protocol.

The details that are not described in the protocol specifications include the following:

- Networking between entities
- Anonymous networking between entities and the TGC
- Hash function(s) to use
- Licences for objects (the $Licence$ field in L)

The networking used in this implementation is described in chapter 4. This describes the technique used to communicate, the advantages and disadvantages of that technique, and some alternate techniques with benefits.

The anonymous networking is not specified as such, and any implementation of an onion routing [82] (or garlic routing [14]) may be used. An example implementation would be TOR [15]. This implementation uses no anonymous communications built into the system, but the use of such a routing protocol could be used in a layer underneath the protocol with no (or minimal) changes to the implementation.

The hash function used is a combination of SHA-256 and SHA-512, both from the Secure Hash Algorithm (SHA) family 2nd Generation [54]. The reasons for this choice are the high standard of the algorithms and the trust held by the algorithms in the security field. In later releases of the protocol, the SHA-3 family could be used when it is finalised [52].

The licences used in the system are not defined, and are intended to be supplier dependant so the supplier would have to extend the library that this implementation provided. More details of this are described in chapter 4.

3.2 Secure Software Design Lifecycles

There are several security development cycles that share roots with the well known software design lifecycles such as the Waterfall [73] and Spiral [5] models, or a iterative approach [41]. A key feature of these frameworks is that they integrate security into every stage of the cycle. This hopefully leads to more secure software.

To aid comparison between different lifecycles, each lifecycle will be split into several logical phases. These phases create a common ground for each lifecycle to enable discussion and comparison of these lifecycles.

- Security Training
- Requirements Gathering
- Design
- Implementation
- Verification
- Release

Security training is essentially a phase that happens before a project starts, though some lifecycles explicitly include it. The training here can be quite general or it can be project specific.

Requirements gathering is an important part of any lifecycle, whether or not it is security based. In a security based lifecycle, it is where any security needs are decided upon, and what level of security is satisfactory (this may involve a trade-off of some kind). The gathering process may involve some kind of risk/threat analysis and uses this information to build a model of the security needed.

The design phase is the phase where all the details on the how the project is

going to be implemented are defined. Any design documents are created during this phase. This phase is quite closely related to the requirements phase as the requirements create a starting block from which to build on.

The implementation phase is where the actual coding takes place. It should follow the design phase closely to make sure all the requirements are met.

The verification phase is where some kind of analysis takes place to make sure that the implementation is correct in regard to the security requirements and any specifications. It may involve use of static analysis tools or test cases or similar. These verification tests may be implemented parallel to the implementation phase, or as a separate phase.

The release phase is the phase that happens after the project is finished. It may involve creating a plan on how to handle any vulnerabilities that arise after the product is released. In this phase, any vulnerabilities that are found, will be fixed. This can require that the lifecycle to be repeated on a smaller scale for each set of vulnerabilities.

The above phases work in a general sense that can have specifics applied to it with each individual lifecycle. Any lifecycle style can be used, such as a traditional waterfall or spiral model, or an agile approach.

There are a wide variety of different security frameworks in current practice. Many of them have similar aspects and some have a few unique features. This section presents a selection of security frameworks, and discusses advantages and disadvantages of using them in the area of this project. Each discussion also details how the particular framework can be applied to this project.

- Section 3.2.1 presents complete lifecycles
- Section 3.2.2 presents auxiliary tools that can combine to make a complete lifecycle

3.2.1 Complete Lifecycles

This subsection will present some complete secure software design lifecycles, discuss merits and limitations, and compare similarities. This will lead into the next subsection which presents some techniques that do not make up a full lifecycle, but can be used with others to build a custom lifecycle.

The lifecycles presented in this section are as follows:

- Team Software Process - Secure (TSP-Secure) [9]
- Comprehensive, Lightweight Application Security Process (CLASP) [62]
- Security Development Lifecycle (SDL) [30]
- Secure Software Development Methodology (SecSDM) [23]

TSP-Secure

Team Software Process Secure (TSP-Secure) is a software lifecycle methodology [9]. It is an extension of the Team Software Process (TSP) methodology [81] (developed by the Software Engineering Institute at Carnegie Mellon University). It combines use of coding standards, checklists, and tools to verify output.

TSP-Secure is being developed as a part of the *survivable security engineering* research done at CERT® [80]. CERT is part of the Software Engineering Institute at Carnegie Mellon University. This research involves many other security tools and methodologies that will be described later in this section.

The main idea behind TSP-Secure is the use of a checklist, which is a

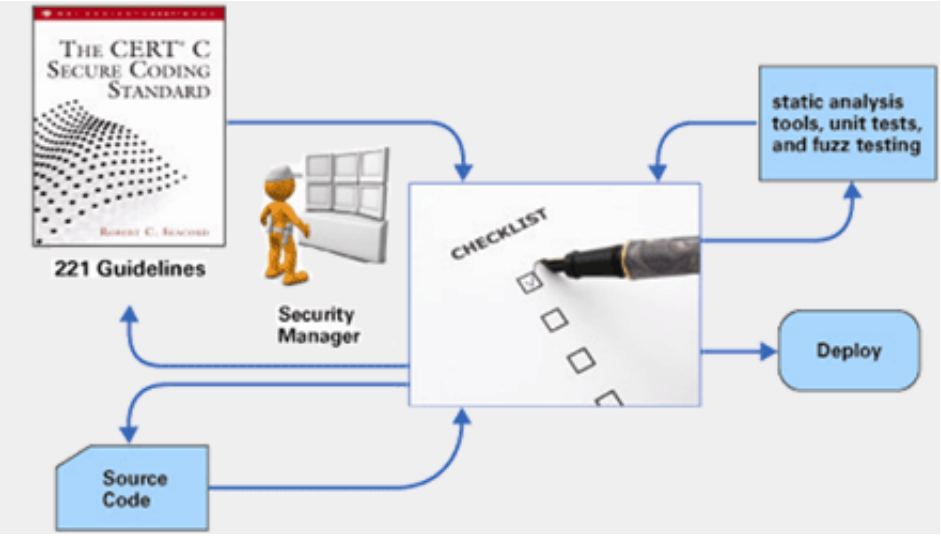

Figure 3.1: TSP Secure

project specific *working document*. This checklist contains everything that needs to be done in each phase, and is modified to create new additions or check things off at each step of the process.

The checklist, while being project specific, can include items that may be present in more general checklists or guidelines. For example, looking at Figure 3.1, the CERT® C Secure Coding Standard is a general set of guidelines, but can have a heavy influence on the items used in any given project specific checklists used in TSP-Secure.

Figure 3.1 shows an overview of how TSP-Secure works. In this image, it shows use of the CERT® C Secure Coding Standard, which is part of the CERT Secure Coding Standards collection [8]. Possible tools that can be used for the verification and testing process are presented and discussed in Section 3.2.2.

Below is a more in depth outline on how the checklist may be modified during each phase of the lifecycle.

The first step in the process will be to generate an initial start to the checklist, usually by determining the security requirements based on the project specifications. This step produces an initial checklist that can be used throughout the rest of the steps. This step will fit into the requirements phase. This phase and the remainder of the phases will include a security manager overseeing additions and completion of the checklist to ensure everything is done properly.

The design phase will involve deciding what libraries, languages, and other design aspects to be used in the security project to meet the requirements. These decisions are noted on the checklist, and can be aided by a set of security guidelines, like those in the CERT® C Secure Coding Standard book [8]. This phase is illustrated by the arrow loop in the top left of the figure, from checklist to the coding standards and back.

The implementation phase involves the actual coding of the project. This is where the checklist is *checked* off as something is implemented, and some more items are added (they will be used in the verification stage). Throughout this phase, reference to the coding standards may take place to ensure proper code is produced. This phase is illustrated by the bottom left arrow loop in the figure, as well as the top left loop.

The verification phase involves using various tools to ensure that the implementation meets the requirements correctly. These tools can include static analysis tools, unit testing tools, fuzz testing, and other similar techniques. When something has been verified, the appropriate entry on the checklist is checked off. Tools that can be used for this purpose are presented in Section 3.2.2. The top right loop in the figure illustrates this phase.

The release phase in TSP-Secure involves just deploying it (shown on the right hand side of the image). After release, if vulnerabilities are found, this process can be repeated as necessary.

This checklist approach will be useful for validating the development process as it can generate a new checklist and check things off that have been done. Addition of new checklist items may have a bias through towards what was done during the development process though.

CLASP

The Comprehensive, Lightweight Application Security Process (CLASP) is a software security lifecycle with a focus on web development. CLASP is a sub-project of a larger project, the Open Web Application Security Project (OWASP) [62]. OWASP is a security framework which incorporates many sub projects, tools, and methodologies. Section 3.2.2 will discuss other projects in OWASP and how they can be used to work together.

It is designed so that it can be integrated with other lifecycles with relative ease. The general idea is that any security needs are identified and addressed at an early stage in the lifecycle, and thoroughly tested and verified for appropriate security requirements.

CLASP has a set of best practices that it follows, and they are as follows:

1. Institute awareness programs
2. Perform application assessments
3. Capture security requirements
4. Implement secure development practices
5. Build vulnerability remediation procedures
6. Define and monitor metrics
7. Publish operational security guidelines

These practices are intended to be a broad overview of how a project is tackled and because of this, they tend to be quite vague about how some techniques are to be used. In contrast, TSP-Secure has a concrete checklist to follow during each phase, which is well-defined and the only vagueness in TSP-Secure is the choice of tools to use.

These practices contain several different techniques which are used throughout the different phases in the lifecycle.

1. The institute awareness program is a program that brings security to all the members of a development team through training [61]. This practice belongs to the security training phase.

2. Performing application assessments include the following techniques: identifying, implementing, and performing security tests [59]; threat modelling [63]; and performing a source level security review [64]. The testing technique is in the verification phase, the threat modelling, in both the requirements and design phases, and the security review fits into the verification phase.

3. Capturing security requirements involves a decision, deciding on the security requirements for the project [58]. This fits into the requirements gathering phase.

4. Implement secure development practices involves defining or using an existing set of secure development practises. This fits into the security design phase of a lifecycle.

5. Building vulnerability remediation procedures involves building a set of procedures to aid dealing with vulnerabilities discovered before or after a release of a product [57]. This fits into the release phase of a lifecycle.

6. Metrics are useful throughout all phases in a lifecycle. This practice involves defining a set of metrics that can allow judgement on how secure something is, then using these metrics to monitor development.

7. The operational security guidelines are intended for documentation purposes after a product is released. It fits into the release phase of a lifecycle.

Because CLASP is not precisely defined, it is not very helpful for validating out development process. If the specifications for CLASP were more detailed, its metric system proves to be very useful for the validating process.

SDL

Security Development Lifecycle (SDL) is a Microsoft designed software development lifecycle, with an emphasis on security. The basis of the lifecycle is a series of phases which integrate security at each step. This ensures that security is placed in the application as early as it needs to be and no last minute additions to the code base *should* be necessary. The phases are as follows: security training; requirements gathering; design; implementation; verification; and release. There is one additional phase after the cycle is complete, i.e. a response to any vulnerabilities that occur post-release. This phase fits into the release phase for comparison.

SDL is a detailed lifecycle definition, but lacks the support of concrete techniques such as pre-made forms for requirements, tools for testing/verifying. It is more concrete than CLASP, and about on a par with TSP-Secure (which is very detailed for requirements and design phase, but not so much in implementation and verification).

Security training is useful for a team based environment to bring all members up to the same standard, and establish a team wide process that can be followed out for any security concerns. The training must also cover technical aspects such as current vulnerabilities, and possible future vulnerabilities in the area of the project. Information on how to gather infor-

mation about security matters must also be delivered in training.

Requirements gathering involves identifying key security needs for the application, and to what standard these needs must be met. This involves analysing risk and threats to the application and the severity of these risks/threats, then using this information to determine what needs will be met and to what level. This can mean balancing between usability and security, whichever is more important to the specific application.

The design stage is the stage where a base idea is formed and developed. It involves performing a threat analysis in greater detail than the requirements stage to narrow down all the requirements for the overall project and starts building a structure for the project.

The implementation stage is where the programme becomes a concrete application. For this stage, best practices need to be followed to ensure security is held and no unwanted vulnerabilities are introduced. Tools that need to be used are identified here, and libraries that should or should not be used, and throughout the stage, static analysis should be performed to spot any potential vulnerabilities.

The verification stage is where the implementation meets the requirements and design. Basically the application should be functioning by this stage, so it needs to be tested against the requirements decided upon. This can be done by some testing framework (mentioned in more detail in section 3.2.2), an independent tester, in-house testing or manual inspection of code to verify requirements.

The release stage gathers together any details that may be needed if an incident happens, a final review of the product, then file away all information needed. The product is now released to the users.

The response stage occurs if a security incident happens post-release. This is where all the information filed away at the release stage is needed, get-

ting the response team up to knowledge with the product. A fix is then made, and released to either the single user, or all users as a public fix.

SDL's descriptive process would make it an ideal candidate for validation of the development process. The lack of tools to aid the validation is one disadvantage.

SecSDM

SecSDM is a process that is based on SDL aimed at implementing security into any software development life cycle [23]. It builds on the authors previous work [24] which gives a general outline of security software development. SecSDM is available in both paper form, and as an application to guide through the process.

SecSDM builds on the good practices of SDL and continues the philosophy of a very detailed lifecycle. SecSDM is based largely on various international standards on security.

The main benefit for SecSDM is the use of the concrete form for requirements and design, but it lacks the choice of libraries and tools for testing and verification. The verification process in SecSDM is done by a technique of backtracking through the design of the project, and this technique is also useful for the release phase where vulnerabilities may become apparent.

SecSDM involves the following stages: Investigation; Analysis; Design; Implementation; Maintenance. The investigation and analysis phases map to the requirements gathering phase; the design and implementation phases map to the phases of the same name respectively; and the maintenance phase maps to the release phase.

SecSDM has the benefits of SDL, as well as a concrete form for require-

ments that link all the way to the implementation. This makes it an excellent system for use with validation of the development process.

SecSDM was chosen to validate the design for this project because of its good practices, and has the foundation of international engineering standards. The lifecycle also has a strong focus right from the requirements gathering through to implementation while other lifecycles may only focus on one particular phase or may be too vague for use in validation.

Summary

Table 3.1 presents a summary of the complete lifecycles covered in this section. It shows each phase of the lifecycle, and what techniques it provides for ensuring security.

From this table, the following conclusions can be drawn:

TSP-Secure has good support for requirements gathering using SQUARE, but has an issue in that the specific tools needed for completing the implementation and verification need to be decided on.

CLASP is very vague in its description, as most of the description is just a set of guidelines on how to go about a secure project with no specifics on procedures that can be used.

SDL has a very detailed process, but lacks concrete tools/forms/checklists that some other lifecycles have.

SecSDM has the benefits of SDL's detailed process, and also has a custom form/checklist to work through the requirements gathering to the implementation stage in the lifecycle. This gives all round good support, and the use of the form makes it a valuable tool for validation of a custom lifecycle as well. For this reason, SecSDM was chosen to validate the development process taken, and will be described in more detail in Chapter 5.

Phase	TSP-Secure	CLASP	SDL	SecSDM
Concreteness	Medium	Low	Medium	Medium-High
Training	General Training	Institute Security Awareness Program [61]	Project specialised training program	General Training
Requirements	SQUARE [43]	Capturing Requirements [58], Threat Modelling [63]	Risk analysis, determine security thresholds, risk assessment	Asset impact analysis, linking to common threats to produce risks. Risk analysis. [33]
Design	CERT® Coding Standards [8] (or similar), Checklist	Threat Modelling [63]	Threat modelling, analyse attack surface	Mapping risks to security services and mechanisms [34]
Implementation	Static Analysis, Checklist	Secure Development Practices [60]	Determine required tools/libraries, static analysis	Mapping software libraries to security mechanisms.
Verification	OCTAVE [2], Unit Testing, Fuzz Testing, Checklist	Security Testing [59], Code Review [64], Relating back to Security Policies and Requirements	Dynamic analysis, fuzz testing, review attack surface	Backchecking through other phases
Release	Deploying, possible repetition of process	Vulnerability remediation procedures [57], operational security guidelines	incident response plan, final security review	Possible smaller repetitions of process

Table 3.1: Summary of complete software security lifecycles

3.2.2 Auxiliary Tools

This subsection will present any methodologies and tools that do not make up a complete lifecycle in themselves, but when they are used with other components (or replace a phase in an existing lifecycle) they can make up a fully secure software design lifecycle.

The subsection divided so that methodologies or tools that fit under one category (or lifecycle phase) are grouped together. There may be some overlap with some tools, and this is mentioned when this occurs.

Requirements Phase

For the requirements phase, a possible methodology that can be used is the Security Quality Requirements Engineering (SQUARE) process [43].

SQUARE is a development process to help integrate security early in the development life cycle [43]. It involves nine steps: defining requirements; identifying security goals; developing supporting documents; risk analysis; elicitation techniques; generating security requirements; categorising requirements; prioritising requirements; and inspecting requirements.

SQUARE, like TSP-Secure (mentioned in previous subsection), is also a part of the *survivable security engineering* research done at CERT® [80]. CERT also develops OCTAVE, a secure coding standard, and other things. OCTAVE is discussed in the evaluation section, and the coding standard will be discussed in the implementation section.

Design Phase

The design phase can use a range of tools and methodologies that aid the designer in the process. This section covers some modelling tools, some

potential frameworks, and some security guidelines that can be applied.

One modelling tool that is useful in designing security applications is UMLSec [37], which is an extension of the popular Unified Modelling Language [32]. The extension makes use of the standard UML extension mechanics such as *Stereotypes* and *Constraints*.

Some possible frameworks that can be used in the design stage are: the Open Web Application Security Project (OWASP) [62]; Enterprise JavaBeans (EJB) [56]; or the Spring project [36] (including Spring Security [46]).

Open Web Application Security Project (OWASP) is a security framework geared towards web applications. It contains several different tools that aid security engineering. There are tools to *protect* from security related design and implementation flaws, tools to *detect* security related implementation flaws, and tools to integrate security into a software development life cycle. A few OWASP projects are listed below:

- OWASP Secure Coding Practices
- OWASP Development Guide
- OWASP Enterprise Security API (ESAPI)
- OWASP Application Security Verification Standard (ASVS) Project
- OWASP Code Review Guide
- OWASP Testing Guide
- OWASP Comprehensive, Lightweight Application Security Process (CLASP)
- And many more.

While on a whole the OWASP framework fits in the design phase, some of the projects can be useful in other phases. The secure coding practices, de-

velopment guide, and the enterprise security API are useful in the implementation phase. The ASVS project, the code review guide, and the testing guide are useful in the verification phase. The CLASP project, which was mentioned in Section 3.2.1, is a complete lifecycle in itself.

Enterprise JavaBeans (EJB's) are types of applications that aim at server-side modular enterprise applications [56]. A bean can be thought of as a single module that can make up a part of a application. These style of applications tend to be web orientated.

There are two major types of beans: session beans and message driven beans. Session beans can be either state full, stateless, or singleton. The particular one to use depends on the situation needed to model; *does it need state?* Message driven beans are those driven by a message being sent, not a method being invoked. This is done by a bean "subscribing" to any messages that it wants to be notified about, then it will be triggered when any of these messages are sent.

Spring is a framework for developing enterprise Java products, and makes large use of JavaBeans without the project source being dependant on Spring.

Spring Security is a tool used for projects using Spring to incorporate security in projects. Spring Security is itself a project that uses Spring, to provide security to other projects using Spring. The tool uses a simple XML based project file to define security needs for the project. It is mostly useful for web based applications, much like OWASP.

Implementation Phase

In the implementation phase, the security of any libraries used need to be evaluated. In Chapter 5, a section covering cryptographic libraries is presented.

Another issue is avoiding introducing coding errors. For this reason, the implementation and verification phases are quite closely connected. This section will introduce a few tools and guidelines that aid good implementation.

Before implementation starts, a concise coding style needs to be determined. A few well known standards include the GNU Coding Standards [22] or the Linux Kernel Coding Style [88].

Coding Standards are an integral part of the development process. Standards can come in many forms, but the principle is the same - they are a set of guidelines on how the code is to be developed. Standards may include, among other things, whether certain functions need to be used, and where they are to be used.

Another set of standards, aimed at security development is the CERT® Secure Coding Standards [8]. These are a collection of standards for a few popular programming languages such as C, C++, and Java. They are used in some of the processes developed by CERT, and can be used as a good starting ground for many developers. One important aspect of the CERT Secure Coding Standards is that they focus on secure programming, something that many other standards do not cover.

As mentioned above, the CERT® Secure Coding Standards are being developed as a part of the *survivable security engineering* research done at CERT® [80] at Carnegie Mellon University along with SQUARE, TSP-Secure, and OCTAVE (mentioned below).

Checkstyle is a tool that can be applied to source code to determine whether the coding standard is being followed correctly [7]. This is useful while implementing in order to check that everything is running on track.

A similar tool, but one that is aimed at finding common vulnerability and bug patterns, is FindBugs [29]. FindBugs is a tool that uses static analysis

to find *bugs* in source code. Bugs here can be anything from performance bugs, bad practice, correctness, security vulnerabilities, and dodgy hacks. It can accept user defined lists of bugs, as well as using the packaged list. This tool is useful during implementation of a security project to keep the code clean of bugs (especially possible security vulnerabilities). It can be applied during each build, or when code is submitted to a repository.

Verification Phase

The verification phase is used to test against security vulnerabilities, verify that the application follows the requirements and specifications, and evaluate the security of the application.

This section will present a framework for evaluation, OCTAVE [2], and also some tools that can be used to test and verify against the implementation.

Operationally Critical Threat, Asset, and Vulnerability Evaluation (OCTAVE) is a collection of tools, techniques and methods for risk-based information security strategic assessment and planning. Three methods that are a part of OCTAVE are: the original OCTAVE; OCTAVE-s (aimed at smaller organisations); and OCTAVE-Allegro (streamlined).

As mentioned above, OCTAVE and its partners are being developed as a part of the *survivable security engineering* research done at CERT® [80] at Carnegie Mellon University.

Another process that can aid evaluation is the Orange Book (mentioned in the training phase). and this can be used to assess how secure the application is.

The rest of this section presents the following tools: ESC/Java2; JUnit; and JTest.

ESC/Java2 is a static analysis tool. Static analysis tools are tools that take into account the raw source code of the program to find either errors, vulnerabilities, verification on conditions, or many other features. More information on static analysis can be found [51]. Other static analysis tools include Checkstyle, and FindBugs (both mentioned above).

ESC/Java2 is an *Extended Static Checker* for the Java programming language. It is the second version of the software. It can be used to assert statements that should be correct at run-time, but checks it at *compile-time*. It can be used to prove pre- and post-conditions, as well as loop invariants. In the back-end, ESC/Java uses the Simplify theorem prover [10] as an automated theorem prover. ESC/Java is not considered complete, and false positives or negatives can possibly occur.

ESC/Java uses an extension to the Java Modelling Language (JML) [42] to specify what constraints are needed to be checked. This allows a developer to put simple *comments* in their source code and run a tool against it to have more peace of mind. These constraints could be security related, such as the works by Hubbers [31] and Schubert [77].

An issue with checkers like ESC/Java, is that the developer needs to specify manually the constraints on the program which uses up a valuable developer resource, time. There are some tools that provide a solution, *automatically* generate the conditions. Some of these tools are the Daikon Invariant Checker [19], and Houdini [21] (which unfortunately has not been developed since 2001).

The other tools presented are types of unit testing tools. This means that *unit tests* can be defined for each component of a project, and the tools used to test it. These unit tests refer to what the project requirements and specifications are and may be decided in the design phase.

JUnit is a testing framework useful for testing small unit cases in a project. Basically a specific unit test case is written to test a single logical part of

the project, for each component that needs testing. This can be useful to test security vulnerabilities by ensuring the return results do not leak any private information, and also for testing whether the implementation follows the specification. JUnit is specific to the Java programming language. For other languages, there are various other projects.

JTest is a tool that uses static analysis to generate automatically JUnit test cases [67]. These automatically generated tests should be reviewed by someone with a knowledge of the system to ensure quality tests. This automatic process speeds up the testing process significantly as the developer is not required to write up test cases manually.

Chapter 4

Design and Implementation

This chapter outlines the implementation of the protocol, and the process taken during implementation. This chapter will be split as follows:

- Section 4.1 outlines the general structure of the implementation.
- Section 4.2 describes the implementation of the Tag Generation Centre.
- Section 4.3 describes the implementation of the supplier.
- Section 4.4 describes the implementation of the reseller.
- Section 4.5 describes the implementation of the tag.
- Section 4.6 describes the implementation of a licence.
- Section 4.7 describes the implementation of the hash object.
- Section 4.8 describes the implementation of the cryptographic functions in the system
- Section 4.9 describes the implementation of the networking in the protocol.

4.1 Structure of Implementation

This section outlines the structure of the implementation of the protocol. The protocol specifications are described in Section 2.1.

The protocol involves the following entities: The Tag Generation Centre (TGC); a group of suppliers; a group of resellers; and a group of customers. In this implementation, the customers are treated as resellers. Each of these entities are described later in this section.

The protocol also defines the tag, and the implementation of a tag is described in Section 4.5.

Outside of the scope of the protocol itself, is the implementation of the networking between the entities. The implementation of this networking is described in Section 4.9.

4.2 Tag Generation Centre Implementation

This section describes the implementation of the Tag Generation Centre (TGC). The TGC performs the following functions:

- Generating the Elgamal parameters p, q, and g.
- Registering suppliers products.
- Issuing tags to suppliers.
- Issuing tags to resellers based on a zero knowledge proof.

The implementation of these functions and the backing object are described in the rest of this section. The networking to interact between the TGC and other entities is described in Section 4.9.

4.2.1 Backing Implementation

The backing of the TGC involves a Java class with the following fields:

- specifications (ElGamalParameterSpec)
- privateKey (ElGamalPrivateKey)
- publicKey (ElGamalPublicKey)
- registeredItems (Map<Hash, ElGamalPublicKey>)
- zeroKnowledgeProofs (Map<Tag, BigInteger[]>)
- issuedTags (Map<Hash, Set<Integer>>)

The specifications field is an object that holds the values of the ElGamal parameters p, q and g. ...

The private and public key are the private and public key respectively for the TGC. These keys are used for signing, encrypting and decrypting data sent to/from the TGC.

The registeredItems is a map between a Hash object and a public key. The Hash object is a hash value of an object, described at .... The public key is the public key for the product, which is sent by the supplier when registration occurs. This map is used to detect replay registrations.

The zeroKnowlegdeProofs map is a map between a Tag object and an array of BigIntegers. The tag object is the tag being used in the zero knowledge proof, and the array holds two items. The first entry in the array is the challenge c sent by the TGC, and the second entry, which is only available after the proof is completed, holds the response value r sent by the reseller.

Detection for replay of a zero knowledge proof is detected in two ways: First if a reseller initiates a tag request and the map has an entry for the original tag, the request is denied; second if a reseller tries to complete

a zero knowledge proof by sending the return value r, and that value already exists in the array, then the request is denied as the proof has already been completed before.

The issuedTags map holds a set of *tag numbers* for each item registered.

4.2.2 Generating Elgamal Parameters

This section outlines the method for generating the ElGamal parameters. As mentioned above, the parameters are p, q, and g. These parameters are global to the entire system, and are used for creating keys and performing the zero knowledge proof.

To generate these values, a modified form of the bouncy castle is used. The modifications made are for more checks to be made against the generated values. The additional checks are detailed below. Without these checks the system doesn't work correctly.

The rest of the process uses just the straight bouncy castle library to do all the key generations, signing, verifying, encryption and decryption.

The additional checks are to be done instead of the code found in the DHParametersHelper class, the method being replaced is the selectGenerator method. The additional checks are as follows:

- Checks that $g^{p-1} \bmod p = 1$
- Checks that $g^2 \bmod p \neq 1$
- Checks that $g^q \bmod p = 1$
- Checks that $g^p \bmod p = g$

If any of these checks fail, the selected generator is discarded and another is selected to be tested.

The original code only checks that the following two conditions are not met:

- $g^2 \bmod p \neq 1$
- $g^q \bmod p = 1$

These checks are incorporated in the new checks, no loss of security occurs, only increased robustness to any factorisation attacks.

4.2.3 Registering Suppliers Products

This section outlines the method used to register products with the TGC.

The parameters to this method are one encrypted registration request. A registration request is an object that holds a hash value and a public key for the item.

The process for registration are as follows:

1. Attempt to decrypt the registration, throws exception if fails
2. Ensures decrypted registration is valid, throws exception if invalid
3. Checks the registeredItems map for the hash value in the request, throws exception if one exists (meaning that this item has already been registered)
4. Checks that the public key in the request is a valid ElGamal key using the same parameters that were generated by the TGC
5. If all checks pass, the registration is added to the registeredItems map, an empty set is added to the issuedTags map, and then a signed receipt is returned to the supplier.

This method is marked as synchronized to avoid two attempts registering an item at the same time.

4.2.4 Issuing Tags to Suppliers

This section outlines the method used to generate tags based on requests from suppliers.

The parameters to this method are one signed tag request. A tag request is either a supplier or a reseller tag request. In this method, the supplier tag request is given. A supplier tag request contains a signed licence, a public key and a commitment value (BigInteger).

The process for generating the tag is as follows:

1. Checks that the signed tag request is in fact a supplier tag request, throws exception if not

2. Checks the public key in the request is a valid ElGamal key using the same parameters that the TGC generated

3. Ensures that the licence is signed by the same key that was used in the registration stage

4. Checks that the licence field tagno has not been seen before for this item by checking the issuedTags map for the presence of this tagno

5. Adds the tagno to the set contained in the issuedTags map and then issues a signed tag

This method is also marked as synchonized to avoid two attempts of generating a tag with the same tagno in the licence.

4.2.5 Issuing Tags to Resellers

This section outlines the method uses to generate tags for resellers. It includes the TGC side of the zero knowledge proof. In the reseller implementation details below, the full details of the zero knowledge proof is presented.

There are two methods used for generating a tag for resellers. The first method takes a encrypted request for a tag, does some housekeeping and returns a challenge for the zero knowledge proof. The second method takes the result of the zero knowledge proof along with the original tag request, does some replay detection, verifies the proof and returns a tag if everything passes.

The first method has the following process:

1. Decrypts the tag request, throwing an exception if fails
2. Checks that the decrypted tag request is a value reseller tag request, throwing an exception if not
3. Checks that the original tag is a valid tag that was signed by the TGC
4. Checks that the public key used in the request is a valid ElGamal key using the same parameters that were generated by the TGC
5. Checks that the original tag has not been used to generate a tag before by checking the zeroKnowledgeProofs map for the key
6. Generate a challenge value c, and put it in the zeroKnowledgeProofs map
7. Return the challenge c

The reseller then computes the result of the zero knowledge proof $r = z_r + csk_r \bmod q$. The reseller then calls the next method with the customers

public key and commitment value and the result r. This data is all signed by the resellers private key for that tag.

The second method has the following process:

1. Retrieves the entry from zeroKnowlegdeProofs map with the original tag as the key
2. Checks that the zero knowledge proof has already been started (i.e. a challenge has been sent)
3. Checks that the result of the zero knowledge proof has not already been sent (the second element in the array is null)
4. Add the result value to the array, to detect replay attacks
5. Checks that the following formula holds: $g^r \bmod p = a_r pk^c_{tag,r}$
6. If all tests pass, return a signed tag.

Both of these methods are set to be synchronized to stop any attacks due to the concurrent nature of the network.

4.3 Supplier Implementation

This section describes the implementation of the suppliers in the protocol. Suppliers can also act as resellers, but this section only outlines the functions of a supplier.

The functions that a supplier performs are:

- Registering items with the Tag Generation Centre (TGC)
- Requesting tags for resellers purchases

The implementation of these functions as well as the backing implementation are described in the rest of this section.

The networking to interact with the TGC and the resellers is described in Section 4.9.

The supplier class is marked as abstract and designed so that custom implementations can be made to extend this class. The only restriction is that the core functions of the supplier are marked as final and can't be overridden by custom implementations.

4.3.1 Backing Implementation

This section describes the backing implementation of the supplier object.

The supplier object has the following fields:

- tgcPublicKey (PublicKey)
- specifications (ElGamalParameterSpec)
- items (Map<Hash, Item>)
- keys (Map<Hash, key pair>)
- receipts (Map<Hash, SignedRegistration>)
- tagnos (Map<Hash, Set<Integer>>)

The tgcPublicKey field contains the public key of the Tag Generation Centre, and the specifications field contains the parameters p, q, and g. These values are initialised when the object is created.

The items field maps the hash of an item to the actual item object. This is used for when the actual item is sent along side the tag, the supplier can find the item by looking up the hash key.

The keys map contains the key pair used for each item. The public key is used for the tag and registration, while the private key is used to prove that we are the legitimate supplier of the item that was registered with the TGC

The receipts map holds all the registration receipts that can be used to prove that the TGC actually registered the item. This can be used to show that a rogue TGC is registering an item with multiple suppliers, which is against the specification.

The tagnos map holds a set of tagnos used for each item.

4.3.2 Initialisation

When the object is created, the TGC public key and the parameters for the ElGamal algorithm are retrieved from the TGC. This uses a network connection to the TGC as described in Section 4.9.

After the TGC information has been retrieved, the custom method initialise is called, this method should setup anything that the custom implementation of the supplier needs, for example registering all the items that the supplier owns.

4.3.3 Registering Items

This section outlines the process of registering an item with the TGC.

The method parameter is the item object. The process is as follows:

1. Generate a hash for the item, and store the hash-item pair in the items map

2. Check that the item hasn't been registered before, by checking the receipts map
3. Generate a key pair for use with the item, and store it in the keys map
4. Create a registration request using the hash of the item and the generated public key
5. Encrypt the request using the TGC's public key
6. Send the request to the TGC, and await the reply
7. If an exception occurs, let the user know and stop processing here
8. If no exception occurs, check that the registration receipt returns matches the request sent, and is signed by the TGC
9. Add the receipt to the receipts map
10. Add an empty set to the tagnos map for this item

This method is intended to be called by the custom implementations through the initialize method, but this is not a restriction. The only restriction is that this method can't be overridden, to protect the protocol from being broken by bad implementations.

4.3.4 Requesting Tags

This section outlines the process of requesting a tag from the TGC after a purchase request from a reseller.

This method is called with a parameter of a purchase request. A purchase request is a request from a reseller that contains a item id, a public key

and a commitment value. This is sent by a reseller using the networking implementation described in Section 4.9.

The process of this method is as follows:

1. Check that the item is sold by the supplier by checking there is an entry in the receipts map using the hash value as a key

2. Generate a licence, containing a unique tagno field

3. Sign the licence using the private key found in the keys map

4. Create a supplier tag request using the licence and the resellers public key and commitment value

5. Sign this request and send it to the TGC

6. If an exception is thrown, let user and reseller know

7. If no exception is thrown, check the returned tag to see if it matches the request sent, and is signed by the TGC

8. Send the tag along with the item to the reseller.

When generating licences, a abstract method generateLicence is called. This method is meant to have a custom implementation for each suppliers licence generation code. The created tagno is checked in the tagnos map, and if not there it is used, and added to the map to ensure the TGC accepts subsequent registrations. More information about licence implementation is described in Section 4.6.

Another thing to note, if this supplier also acts as a reseller, first an attempt to resell a product is made before attempting to supply it. The details of reselling a product are described in the next section.

4.4 Reseller Implementation

This section describes the implementation of the resellers in the protocol. Note that the customers in the system are treated as resellers.

The functions of a reseller are:

- Purchasing items from upstream in the chain (either suppliers or resellers)
- Generate tags for customers (involving a zero knowledge proof)

The implementation of these functions as well as the backing implementation are described in the rest of the section

4.4.1 Backing Implementation

This section describes the implementation of the backing object of a reseller.

The reseller object has the following fields:

- tgcPublicKey (PublicKey)
- specifications (ElGamalParameterSpec)
- items (Map<Hash, Item>)
- tags (Map<Hash, Queue<SignedTag>)
- keys (Map<Tag, key pair>)
- zValues (Map<Tag, BigInteger>)

The tgcPublicKey field contains the public key of the Tag Generation Centre, and the specifications field contains the parameters p, q, and g. These

values are initialised when the object is created.

The items map stores all the items that the reseller current has to offer.

The tags map stores a queue of tags to use when reselling. If the queue is empty, the reseller must purchase a new supply of tags from upstream.

The keys map holds the key pairs generated for each tag, while the zValues map stores the z values generated for each tag. The z value is used to compute the commitment value and the result of the zero knowledge proof.

4.4.2 Purchasing items upstream

This section describes the process and implementation of a reseller purchasing items from upstream sources, such as suppliers or other resellers.

The details of the network protocol implementation between the reseller and the upstream source is described in more detail in Section 4.9.

The parameters for this method are the hash of the item, and the connection to the reseller that the purchase is being requested from. The reseller in this case could also be a supplier.

The process for purchasing the item is as follows:

1. Generate a z value
2. Compute the commitment value $a = g^z \bmod p$
3. Generate a key pair for use with the received tag
4. Create a purchase request for the item using the item id, the generate public key, and the commitment value
5. Send the request to the upstream reseller/supplier

6. If any exception is encountered, tell the user and stop processing (any private data is discarded)
7. Check that the tag matches the request and is signed by the TGC
8. Add the returned item to the items map
9. Add the returned tag to the tags queue
10. Add the z value to the zValues map
11. Add the key pair to the keys map

This method is intended to be called when the reseller is out of stock, but can also be called when stock is plentiful as the returned tags are added to a queue.

4.4.3 Generating tags for customers

This section describes the implementation of a reseller requesting tags for a customer purchase. This includes a description of the zero knowledge proof implementation in the reseller.

The parameter to this method is a purchase request from a customer. The details on how this request is sent through the network is described in Section 4.9. A purchase request is an object that contains the item hash, a public key, and a commitment value.

The process of generating a tag is as follows:

1. Check that the item is sold by us, and is in stock, otherwise throw an exception
2. Retrieve a tag from the tag queue, removing at the same time

3. Generate a reseller tag request based on the information in the purchase request and the retrieved tag
4. Encrypt the reseller tag request using the TGC public key
5. Sent the request to the TGC, throwing an exception if failing
6. The TGC should return a challenge value c for the zero knowledge proof
7. Retrieve the z value and key pair for the retrieved tag from the zValues and keys maps respectively
8. Compute the result of the zero knowledge proof $r = z_r + csk_r \bmod q$
9. Send the result and the original tag request to the TGC
10. If TGC returns an exception, add the retrieved tag back to the queue then inform the user and customer
11. If TGC returns a tag, check it's validity and that it matches the request and is signed by the TGC
12. Send the generated tag and the item to the customer

This method is used for generating tags based on a purchase request from a customer. It involves a zero knowledge proof between the reseller and the TGC.

4.5 Tag Implementation

This section describes the implementation of the tags in the system. A tag is the central part to the system, and is necessary for the protocol to work. The structure of the tag is as follows:

- itemPublicKey (PublicKey)
- licence (SignedLicence)
- resellerTagPublicKey (PublicKey)
- resellerTagCommitmentValue (BigInteger)

The itemPublicKey is a field for the public key of the item. This is sent to the TGC in the registration phase between the supplier and the TGC. It is contained in all the subsequent tags.

The licence is a licence structure that is signed by the item private key. It should be verified that the licence is signed by the key be testing against itemPublicKey. The licence is generated when the supplier requests a tag from the TGC due to a purchase request from a reseller. The licence contains a tagno field which is used to detect replay. The details of the licence implementation is described in Section 4.6.

The resellerTagPublicKey and resellerTagCommitmentValue are the public key and the commitment value for the reseller for this tag. They are used in the zero knowledge proof between the reseller and the TGC

The tag can be signed and encrypted, details of how this is implemented are described in Section 4.8

4.6 Licence Implementation

This section describes the implementation of the licences in the system. The licence is a signed token to verify that the item is licenced for the end user. The structure of the licence object is as follows:

- itemID (Hash)

- tagno (Integer)
- licence (byte[])

The itemID shows which item this licence is valid for. It corresponds to a hash value of the item. The hash object is described in Section 4.7.

The tagno field is a unique value to each licence. It is used by the TGC to verify that this licence is not being replayed.

The licence field is the actual licence itself, in a custom form implemented by the supplier implementation. It is simply an array of bytes.

4.7 Hash Implementation

The hash object is used to identify items in the system, without sending the full item across the network. The hash object is — as it's name suggests — a hash value of the item.

The specifications of which hash function to use was not present in the original protocol specifications, so the function used in this implementation may be replaced by another hash function.

The implementation of the hash object uses a series of message digests of the item to avoid any collisions. The algorithms used are the SHA-512 and the SHA-256 hashes.

Additional hashes could be used by extending this class with a custom implementation, and releasing that extension to all users of the system.

4.8 Cryptographic Functions

This section outlines the implementation of the various cryptographic functions in the system. The underlying functions use the bouncy castle cryptographic API, with the exception of signing and verifying signatures, which use a slightly modified bouncy castle implementation.

Various objects in the system can be signed or encrypted, and these objects then become wrapped in a new class (either SignedObject or EncryptedObject respectively). The rest of the section outlines how these classes are interacted with.

4.8.1 Signing

The SignedObject class holds the signed object, and the signature. It has a method getEnclosingObject(). The class is abstract and is intended to be extended for each specific type of object needing signed. This system implements the following classes: SignedLicence; SignedTag; SignedTagRequest. These subclasses have methods such as getEnclosingLicence() depending on their type.

A helper class SigningUtils is used for all signing and verifying of objects. The class has two public methods, sign and verify.

The sign method takes the object to be signed and a private key, and returns a subclass of SignedObject (depending on the object passed in). Note that the private key here is not sent over the network in any way, but has the vulnerability to be sniffed by inspecting the internals of the Java virtual machine.

The verify method takes a SignedObject and a public key. The return value is true iff the object is signed by the public key, false otherwise.

4.8.2 Encryption

The EncryptedObject class holds the encrypted byte array of the object. The class is abstract and is intended to be extended for each type of object needing encryption. This means that the encrypted data sent over the network does have the type of the underlying data attached. This is a tradeoff between ease of use of the system and protecting the data. The actual contents of the underlying object are not sent in the clear.

A helper class EncryptionUtils is used for all encryption and decryption of objects. The class has two public methods encrypt and decrypt.

The encrypt method takes and object and a public key. It encrypts the object and returns the results in a subclass of EncryptedObject (depending on the passed object).

The decrypt method takes an EncryptedObject and a private key. The return result is either null if the decryption failed, or the original object in the correct type. Note that the private key here is not sent over the network, but has the vulnerability of being sniffed by inspection of the internals of the Java virtual machine.

4.9 Network Implementation

This section describes the implementation of the networking between different entities in the system. The networking specification is out of the scope of the protocol.

The implementation of the networking between different entities in the tagged transaction protocol were not specified in the original specifications for the protocol. This means that the networking is an implementation decision to make, and will be implementation dependant.

The implementation of the network requires the design of how objects are represented in a network stream. Because of the choice of language, it is possible for this implementation to make use of Java's serialization technology to represent objects as streams that can be sent over the network. This implementation is not the most efficient way, but makes extending the system for more objects trivial.

Another way to go about the representation of objects is to define a byte level representation of each object used in the system. This approach means that an efficient design can be used to limit unnecessary bandwidth. A side effect of this is that either future objects need to have a reserved structure, or a redesign must take place if further objects are introduced to the system at a later point.

This prototype implementation makes use of the first option of using Java's built in serialization technique. This technique allows one to *serialize* an object into a stream that can be sent over the network for example. If this approach is shown to effect the results badly due to increased overhead, a redesign may be needed for future prototypes.

Each object that will require sending over the network is declared as serializable, and a helper class NetworkUtils takes care of the encoding and decoding of the objects. The use of a helper class keeps the main structure of the library unaltered, and in the event of a change to the design of the networking, minimal changes will be needed.

To declare an object serializable, the class just needs to implement the Serializable interface. This does not require any additional methods. A field serialVersionUID is used to track different versions of the object so that both ends of the network stream know that the object relates to the version they are aware of (or not if that is the case). Also any fields that are not to be sent over the network can be declared transient. This means that that field will not be encoded in the stream.

The NetworkUtils helper class takes care of the following functions: opening connections to remote entities, and receiving connections from remote entities. These functions will be described separately but it should be noted that they provide roughly a reverse functionality of the other. The NetworkUtils requires that any object that is to be sent implements the Serializable interface, otherwise will inform the user of the error.

Each entity in the system has two superclasses that identify how they are to be used for networking. For example, the Supplier class has two superclasses, a SupplierClient and a SupplierServer. This is the same for the Reseller and the TagGenerationCentre as well. The client class is used to connect to remote entities, and the server class is the backing model of the entity (it has all the code for the functionality). The server class will be described later in this section.

4.9.1 Client Side

When an entity wishes to make a connection to a remote entity (say a local Reseller to a Supplier at host "foo" on port 12345. The Reseller class creates a new Supplier object by calling the SupplierClient constructor with the arguments "foo" and 12345, or whatever the host and port combination are. This client class creates a new helper class NetworkUtils that actually makes the underlying connection, and then waits for messages to send.

When that reseller wishes to call a method with the supplier, it calls the method as per usual, and the client code does the work. The client code takes the request, converts it into a Method Object (part of the reflection API in Java), and sends that to the helper class along with the arguments for the method. The helper class then sends that on to the remote supplier (details on how it is received are below). The helper class then either receives the result, or an exception, and passes that back through to the

client. The client takes the result and passes it back to the original reseller class, and if appropriate the exception. If the exception is due to a network error, the client class needs to deal with that in an appropriate way. This implementation simply informs the user of the error and cleans up.

4.9.2 Server Side

The server side of the connection is the appropriate Server class, for example SupplierServer. This class is used for the actual implementation, and any extensions should extend this class to be able to use to networking underneath.

In the example above, the reseller attempts to connect to the supplier. The SupplierServer (or extension thereof) takes the connection and starts a new NetworkUtils helper to receive the messages. Any errors in the network are simply ignored on the server end of the communication.

When the reseller calls a method on the supplier client, the request is sent to the server. The NetworkUtils helper takes the message from the stream, reads in the required arguments from the stream, then invokes the method on the SupplierServer object, getting a result or an exception. This result or exception is sent back to the client by sending a boolean whether it is the result or not, then either the result or the exception depending on what is needed.

Chapter 5

Verification and Validation

This chapter discusses the verification and validation of the implementation of the protocol. These two concepts are similar in nature and need defining. The definition used in this thesis is as follows:

- Verification is an attempt to ensure that the product is built correctly, in the sense that the output products of an activity meet the specifications imposed on them in previous activities.
- Validation is an attempt to ensure that the right product is build, that is, the product fulfils its specific intended purpose.

When there are applied to this project: verification is the process of verifying that the protocol specifications are met in the implementation, and validation is the process of ensuring the users security requirements are met.

The verification stage is simply using some technique to show that the implementation meets the protocol specifications presented in chapter 3. The validation stage is an attempt to show that the implementation is secure, to some degree of secureness. Both these processes can not be exact, and

as states above are just an *attempt*, but the purpose is to attempt to show to the fullest understanding that they are both satisfied.

This chapter will be split in two, Section 5.1 will discuss verification of the protocol specifications, and Section 5.2 will discuss the validation of the security of the implementation.

5.1 Verification of Protocol Implementation

This subsection outlines the procedure followed to validate the implementation correctly implements the specifications of the security protocol presented in Section 2.1.

The following techniques have be used in the validation procedure: A suite of unit tests for small components of the protocol; an internal review of the implementation by the creator of the protocol. In addition, an external review of the implementation against the specifications of the protocol could be possible, but has not been done due to time constraints of this project.

The section is split as follows: Section 5.1.1 describes the unit testing procedure used; and Section 5.1.2 describes the internal review procedure.

5.1.1 Unit Testing

This section presents the various unit tests that are applied to the implementation to show that the specifications are followed correctly.

The purpose of each test is to execute a small part of the protocol's process in a controlled manner an observe the outcome. Some of these tests are

expected to pass, and some are expected to fail. The pass and fail criteria will be described in detail for each test.

These tests should cover all of the different stages of the protocol, and if they all pass or fail as expected, then this validates that the implementation of the protocol is correct with regards to the test cases. The next section shows how the test cases can be validated independently to show that they are suitable as test cases.

Each test case will describe the following:

- Protocol setup procedures
- Inputs for test (either static or dynamic)
- Expected outcome of the test
- Pass and fail criteria for the test

The design and implementation details are outlined in more detail in Chapter 4, but in some situations some design and implementation details have been clarified.

These unit tests will be described in more detail in the rest of the section.

Supplier Item Registration

This test performs a simple registration of an item from one supplier entity with the tag generation centre. This step should pass if an item of the same ID has not already been registered (see next test for this situation). This test involves the following steps:

1. Calculating the ID of the item.
2. Generating a key pair for use with this item

3. Register id using the public key with the TGC in an encrypted message.

The TGC can do additional checks outside of the protocol to ensure that the supplier is the rights holder of the product, but this would not work if anonymity is used.

The implementation of the protocol used in this project does not include any out of band checks on identity, and simply uses a first in first registered model.

The ID of the item is calculated as a SHA-512 hash of the product data, and to minimise the collision possibilities a SHA-256 hash is also appended. The bouncy castle library is used to generate the hash.

The key pair generated is an Elgamal key pair, using the parameters p, q, and g which are generated by the TGC.

To get the protocol up to the stage of this test the following setup is needed to be done:

1. TGC Generates ElGamal parameters p, q, and g
2. TGC Generates key pair for signing and encryption with the TGC.
3. The Elgamal parameters and the public key for the TGC are made publicly available.

At this stage, the supplier is ready to register an item. This test makes the assumption that an item with the same ID has not been registered yet. The pass criteria is that the TGC will allow the registration to happen. If this test fails, the TGC will reject the registration. This test is expected to pass.

The input parameters for the test are the parameters p, q, and g which the TGC generated, the TGC public key, and the item being registered. This test doesn't have any dependency on the item being registered other than

that the item (or one with the same ID value) has not been registered in the past as per the assumption above. For that reason, a static item has been used with description of "Test item".

This test performed as expected in the implementation, that is the TGC accepted the registration.

Supplier Impersonation (Item Registration)

This test performs an attempt at impersonating a supplier during the item registration stage of the protocol. This test should *fail*. This involves the following steps:

1. Obtain a product ID, either by legitimate purchase or intercepting.
2. Generate key pair to use with item
3. Attempt to register item using the new public key with the TGC

Obtaining the product ID by intercepting can happen either by intercepting the original registration or by intercepting a tag. As the original registration is encrypted for the TGC, no valuable data can be obtained. This encryption can only be broken by a flaw in the Elgamal protocol, or if the TGC private key is obtained.

This leaves only intercepting a tag, or a legitimate purchase. If either of these occur, it should be noted that this can only happen if the product has *already* been registered.

When an attempt to register the item with the TGC happens, as the product with the same ID has already been registered, this step fails.

This shows that this test *must* fail iff the protocol is implemented correctly and the encryption of the original registration is not broken.

The protocol setup for this test is as follows:

1. TGC Generates ElGamal parameters p, q, and g
2. TGC Generates key pair for signing and encryption with the TGC.
3. The Elgamal parameters and the public key for the TGC are made publicly available.
4. Any supplier has registered an item i
5. Any reseller has requested a tag through the supplier for item i
6. The TGC accepted the tag request and a tag was generated.
7. This tag was obtained by the attacker.

At this stage, the attacker can try to register an item using the item id from the intercepted tag, or the legitimately obtained tag.

The pass criteria for this test is that the TGC will allow the registration, and the fail criteria is that the TGC will reject the registration. It is assumed that the item i has already been successfully registered, which is a prerequisite for any tags to be generated for it. This test is expected to fail, as the item has already been registered.

The input parameters for this test are the parameters p, q, and g from the TGC, the TGC public key, the ID of i (the id of the being registered). Note that the attacker could attempt to use the public key from the intercepted tag with the same effect of generating a new key pair, but the attacker would not know the original private key. If the registration was accepted, the attacker would not be able to abuse this registration without knowledge of the private key relating the public key that was registered. Both a generated key pair, and reusing the original public key are expected to fail.

The test performed as expected in this implementation, that is the TGC rejected the registration.

Zero Knowledge Proof

This test performs the zero knowledge proof algorithm used in the protocol. This involves the following steps:

1. Customer initiates transaction with reseller.
2. Reseller requests tag from TGC using their old tag.
3. TGC starts zero knowledge proof by sending random challenge c
4. Reseller responds with proof
5. TGC checks proof with expected value and issues tag

This process proves that the reseller is actually the owner of the original tag. The correct response is $r = z_r + csk_{tag,r}$, where z_r is the value used to produce the commitment value ($a = g^{z_r}$), and $sk_{tag,r}$ is the one time private key for the resellers tag. The response is sent to the TGC along with original tag request, all signed by the private key for the resellers tag.

The response is checked by the TGC using the knowledge that the private key $sk_{tag,r}$ relates to the public key with the following function, $pk_{tag,r} = g^{sk_{tag,r}}$. The response value should be correct in the following equation, $g^r = a_r pk^c_{tag,r}$

This test is actually part of the reseller generated tag test which occurs in any transaction with chain length greater than one.

The protocol setup for this test is as follows:

1. TGC Generates global parameters p, q, and g

2. TGC Generates a key pair for signing and encryption
3. TGC shares the parameters p, q, g, and the public key
4. A supplier has registered an item i
5. The reseller has purchased item i off the supplier and received a tag t
6. A customer requests purchase of item i from the reseller
7. The reseller sends a tag request to the TGC on behalf of the customer, using tag t

At this stage, the protocol is in the middle of the reseller generated tag function. The zero knowledge proof must be completely successfully for the TGC to accept the proof and send a signed tag.

The pass criteria for this test is the TGC accepts the proof and returns a signed tag for the item being requested. The fail criteria is that the TGC rejects the proof for some reason. This test is expected to pass.

The input parameters for this test are the global parameters p, q, and g, the public TGC key, the commitment value a_r, and the one time public key $pk_{tag,r}$. The reseller knows the secret value z_r that relates to a_r by the function $a_r = g^{z_r}$, and also the one time secret key $sk_{tag,r}$

The test performed as expected in this implementation, that is the TGC accepted the proof and returned a signed tag.

Zero Knowledge Proof (Replay Attack)

This test is similar to the test above, but the reseller is trying to use the same tag in a zero knowledge proof.

Using two zero knowledge proof transactions with values c_1, r_1, and c_2, r_2, the one time secret key ($sk_{tag,r}$) for the resellers tag can be discovered by computing the following.

$g^{r_1}/g^{r_2} = CB^{c_1}/CB^{c_2} = B^{c_1-c_2}$ and $log_g B = r_1 - r_2/c_1 - c_2 = sk_{tag,r}$

The protocol set up for this test is as follows:

1. TGC Generates global parameters p, q, and g
2. TGC Generates a key pair for signing and encryption
3. TGC shares the parameters p, q, g, and the public key
4. A supplier has registered an item i
5. The reseller has purchased item i off the supplier and received a tag t
6. A customer requests purchase of item i from the reseller
7. The reseller sends a tag request to the TGC on behalf of the customer, using tag t
8. The reseller and the TGC carry out a zero knowledge proof successfully
9. Another customer requests purchase of item i from the reseller
10. The reseller sends a tag request to the TGC, using tag t again

At this stage, the reseller is about to attempt to perform a zero knowledge proof using that tag they have used previously.

The pass criteria for this test is the TGC accepts the proof and returns a signed tag. The fail criteria for this test is that the TGC rejects the proof as a replay, and can produce the one time secret key for the resellers tag. This test is expected to fail.

The input parameters for this test are the usual global parameters p, q, and g, the TGC public key, the commitment value a_r, and the one time public key $pk_{tag,r}$. The reseller knows the secret value z_r such that $a_r = g^{z_r}$, and also the one time private key $sk_{tag,r}$. The TGC also has a record of the previous zero knowledge proof transaction values c_1, and r_1 stored against the item.

The test performed as expected, that is the TGC rejected the proof and discovered the resellers private key for that tag.

Zero Knowledge Proof (Forged Attack)

This test is similar to the above two, except the attacker has no knowledge of the secret value z or the secret key $sk_{tag,r}$ and instead attempts to guess and forge the response value r. This is unlikely to be correct, but there is a very minute chance none the less.

The protocol set up for this test is as follows:

1. TGC Generates global parameters p, q, and g
2. TGC Generates a key pair for signing and encryption
3. TGC shares the parameters p, q, g, and the public key
4. A supplier has registered an item i
5. A reseller has purchased item i off the supplier and received a tag t
6. The attacker intercepts this tag t
7. A customer requests purchase of item i from the attacker
8. The attacker sends a tag request to the TGC on behalf of the customer, using tag t

At this stage, the attacker is ready to attempt a zero knowledge proof with no knowledge of the secret value z_r or the secret key $sk_{tag,r}$.

The pass criteria for this test is that the TGC will accept the proof and return a signed tag. The fail criteria is that the TGC will reject the proof as incorrect. It is expected that this test will fail.

The inputs are the usual p, q, g, the TGC public key, the commitment value a_r, and the one time public key $pk_{tag,r}$. The response value r will be a random number of the appropriate length. Because of the dynamic nature of the r input, this test will be repeated several times to ensure a wide selection of inputs are tested. Note that there is a very small chance that a random input produces the correct response.

Licence Replay Attack

This test shows an impersonation attempt from a reseller attempting to be a supplier by attempting to replay a legitimate licence. This test is expected to fail. This can happen with the following steps:

1. A customer initiates transaction with reseller (they believe to be the supplier)
2. Reseller (impersonating the supplier) sends a generated licence along with the customers request, all signed by the items private key, to the TGC.
3. If everything passes, the TGC returns a signed tag.

In the second step, the generated licence includes a tagno field. This field must be unique to the item, otherwise the TGC will reject the transaction. This stops licence replay attacks. The only other way to send a licence is if it is signed by the items private key, which is protected by the true supplier.

The protocol setup for this test is as follows:

1. TGC Generates global parameters p, q, and g
2. TGC Generates a key pair for signing and encryption
3. TGC shares the parameters p, q, g, and the public key
4. A supplier has registered an item i
5. A reseller has purchased item i off the supplier and received a tag t
6. The attacker intercepts the suppliers tag request which contains licence l
7. A reseller requests purchase from attacker for item i, believing them to be supplier.
8. The attacker sends a tag request to TGC using the licence l.

At this stage, the attacker is attempted to get the TGC to issue a tag for supplying to the reseller. This requires the attacker to send a licence, the public key and commitment value supplied by the reseller to the TGC all signed by the items private key. As the attacker doesn't have the private key, they can only replay the knowledge they have, which is the public key and commitment value of the original reseller. The TGC will check this and will result in not issuing a tag.

The pass criteria is the TGC will issue a tag for the attacker, and the fail criteria is the TGC will refuse to issue a tag due to a replay attack. It is expected that this test will fail.

The inputs are the usual p, q, g, the TGC public key, the intercepted tag request containing the licence l, and the public key and commitment value from the reseller.

Supplier Impersonation

This test shows a reseller attempting to impersonate a supplier based on information from the intercepted registration.

The protocol setup for this test is as follows:

1. TGC Generates global parameters p, q, and g
2. TGC Generates a key pair for signing and encryption
3. TGC shares the parameters p, q, g, and the public key
4. A supplier has registered an item i
5. Attack intercepts the registration
6. A reseller requests purchase of item i from attacker, believing it to be the supplier

At this stage the attacker is requesting a tag from the TGC by impersonating the supplier. To request a tag, they need a licence and the public key and commitment value from the reseller, all signed by the private key of the item. As the attacker doesn't have the private key, they will not be able to sign the triple.

The pass criteria is the TGC issuing a tag, and the fail criteria is the TGC rejecting the request due to incorrect or no signature or no licence. It is expected that this test will fail.

The inputs are the usual p, q, g, the TGC public key, and the public key and commitment value from the reseller.

Tag Replay Attack

This test shows a replay attack when the attacker tries to resell an already sold tag. This assumes that the tag has been received during a legitimate transaction, and has already been used in a transaction since receiving it. This test is expected to fail. The steps involved in this test are as follows:

1. Purchase a product from a reseller upstream, receiving a tag.
2. Resell that product to a customer downstream, using the received tag.
3. Attempt to resell the product to another customer downstream, using the same tag.

The attempt to resell the second time should fail because after a successful trade with the tag, a zero knowledge proof would've taken place and any replay attempt on this will fail with the TGC.

The protocol setup for this test is as follows:

1. TGC Generates global parameters p, q, and g
2. TGC Generates a key pair for signing and encryption
3. TGC shares the parameters p, q, g, and the public key
4. A supplier has registered an item i
5. A reseller has purchased item i off the supplier
6. A customer requests purchases item i off the reseller
7. The reseller requests tag from TGC and carries out zero knowledge proof, receives tag t
8. Reseller sends tag t to customer with item i

9. The attacker intercepts this tag t
10. A customer requests purchase of item i from the attacker
11. The attacker sends a tag request to the TGC on behalf of the customer, using tag t

At this stage the attacker is requesting a tag from the TGC using an intercepted tag t. This tag has already been used before to generate a new tag, and as such a zero knowledge proof has already taken place. The TGC should detect this and reject the request for a new tag.

The pass criteria is that the TGC will issue a tag, and the fail criteria is that the TGC will reject the tag due to a zero knowledge proof already having taken place. It is expected this test will fail.

Reseller Purchase

This test is a slightly larger test that shows the protocol running with a chain length of one, that is a supplier and a customer (who may then resell). This tests whether the supplier can generate a tag successfully. This test is expected to pass.

The steps in this test are as follows:

1. Generate a one time key pair.
2. Generate a private z value.
3. Calculate the commitment value
4. Send a request to the supplier with the one time public key and the commitment value.
5. Receive tag.

The key pair generated is an elgamal key pair using the bouncy castle library. The z value is a random number with bit length the same as parameter q. The commitment value is equal to g^z mod p.

Chain of Two

This test is a slightly larger test again. This tests the protocol running with a chain length of two, that is a supplier, and reseller, and a customer (who can then resell on if they wish). This test should test all aspects of the protocol, and can be extended to show that all combinations would pass iff this test passes (if enough computational resources are available).

This test is expected to pass, and shows that the implementation of the protocol is assumed to be correct.

5.1.2 Internal Review

The internal review consisted of a review by the designer of the original protocol. This was done as an iterative process throughout the development process, and as a final check once the prototype was complete.

This review showed that the implementation correctly implemented the protocol as per the designers specifications. This review also doubled as a security audit of the code base that the implementation used.

5.2 Validation

This section presents the validation of the secureness of the implementation. This was done as a two pronged approach. The first part was using the secure development lifecycle (SDL) to validate secureness through the

process taken to produce the implementation. The second technique for validation is to determine possible vulnerabilities that could be present.

The SDL used to validate the process is the SecSDM lifecycle [23]. This lifecycle was chosen as it has strong ties to international standards in security, and has a very strong focus on designing security. An overview of SecSDM was presented in Section 3.2.1, and a more detailed outline will be presented later in this chapter in Section 5.2.1.

Using this validation technique to validate a design process is useful as it shows that best practices were used throughout the development process.

In a more team based development project, the intuitive approach discussed in Chapter 4 may not be suitable, and a traditional SDL is more applicable. Because of the individual development aspect of this project, the intuitive approach is shown to work similar to a traditional SDL.

The section will be split into several sub-sections as follows:

- Section 5.2.1 will present the SecSDM lifecycle.
- Section 5.2.2 will discuss the application of SecSDM to this project, outlining each phase in the lifecycle.

5.2.1 The SecSDM lifecycle

The lifecycle used for validation of the design process is the SecSDM tool, which was presented in Section 3.2.1, along with other possible lifecycles. SecSDM was chosen because of the strong backing of the ISO standards for security. These standards show good security engineering practises.

The SecSDM tool is used by both a paper based system, and a software tool. In the TTP prototype, the software tool was used. The phases used in the lifecycle are as follows:

1. Investigation
2. Analysis
3. Design
4. Implementation
5. Maintenance

These phases and the general outline of the lifecycle is outlined and discussed below in the section. The application of these phases is discussed in Section 5.2.2.

The rest of the section will outline each stage in detail. Note that the notation A.1 refers to figure 1 in appendix A, A.2 is figure 2 in appendix A, and so on. The notation B.1, and B.2 refer to figures 1 and 2 in appendix B respectively.

Investigation

The investigation stage decides on asset needing protection, and the impact those assets have. Then the associated threats are determined, and the risks derived from those risks. The final risks are sorted into an order determined by the value of the asset impact, likelihood of the threat occurring, and the level of vulnerability of the risk.

Figures A.1, A.2, A.3, A.4, A.5 and A.6 show the form based process of the investigation stage. Figure A.1 shows the process of determining the assets and the impact value. Figure A.2 shows the process of specifying the level of likelihood for common threats. Figure A.3 shows the process of determining the most critical risks through the asset/threat relationships. Figure A.4 shows the process of specifying the level of vulnerability for

each of the selected risks. Figure A.5 and A.6 show the risk analysis process that outcomes an ordered list of risks with an associated value.

This stage in the SecSDM lifecycle was derived from work done by Whitman and Mattord [90] and the ISO/IEC TR 13335-3:1998 [33]

Analysis

The analysis stage takes the ordered risks from the investigation phase and decides what security services are needed to protect against the risks.

Figure A.7 shows the form for this stage. Some common security services are shown for the common threats. The next step is to use the information about the threats to decide on what services are needed for each threat, and to what level. Levels include basic, standard, and extra strong.

Design

The design phase maps the security services needed to protect the risks to concrete security mechanisms. This technique is derived from the ISO standard 7498-2 [34].

Figure A.8 shows a table mapping the security services from the analysis stage to a set of security mechanisms. The rest of the form (not shown) for this stage is decided which of these mechanisms (or all) are needed to support the security service required. In the form, a level of service is entered. Levels include basic, standard, and extra strong.

Figure A.9 shows the final summary table for this stage. This table is marked corresponding the the results of the entries on the forms for this stage without the level of service.

Implementation

In the implementation phase, the security mechanisms are mapped to a software library that will fulfil the mechanism. In the SecSDM example form, only the .Net language is used. This thesis discusses various libraries of different languages that can be used in Section 3.1.4

Figure A.10 shows the form used for this stage. For each mechanism from the design stage, specify which risks they protect, and note the libraries used to provide the mechanism.

Maintenance

The maintenance stage doesn't have an associated form with it, but simply using the information gathered throughout the process to fix vulnerabilities when they arise. The fixing process may include performing another smaller iteration for each vulnerability.

5.2.2 Application of SecSDM to TTP

This section discusses the application of SecSDM to the TTP prototype. This will entail what happened in each stage of the lifecycle, choices made at each phase, and reasoning behind any choices made.

The section will be laid out so that each stage of the lifecycle is described separately, then a summary and general discussion will follow. The stages covered from the SecSDM lifecycle are:

1. Investigation
2. Analysis
3. Design

4. Implementation
5. Maintenance

Investigation

As mentioned earlier in the chapter, the investigation stage is where the assets needing protection are decided. From those assets, a ordered list of risks are derived. This section shows how the process was followed for this project.

We start with deciding on the assets needing protection. Figure B.1 shows the SecSDM form filled out with all the important assets for this project. The most important assets are shown at the bottom of the image along with the asset impact value.

The assets decided upon are as follows:

- TGC Private Key (Critical)
- Supplier Private Key (High)
- Reseller Private Key (Medium)
- Reseller Z-value (Medium)
- Tags (Low)

The reasoning for the final selection is because all of those assets are needed for successful running of the protocol, and if some of those assets for compromised then the security of the protocol is void.

Figure B.2 shows the common threats and the likelihood of them occurring. The reasoning behind this is because they seemed the most adequate levels for each of the threats for this system.

Figure B.3 shows the ordered selection of risks chosen from the asset/threat combinations. The ordering is shown by the letters A to H, where A is the most critical risk, and H is the least. This set of risks and levels were decided by working out how likely a threat is against a given asset.

Figure B.4 shows the levels of vulnerabilities for each of the risks chosen in the previous step. The reasoning behind this is because the level of vulnerability shows how badly a certain risk can damage the protocol system.

The next set of figures show the risk assessment for the chosen risks. Each risk is given a table to complete, and it will give a value of the risk which is a combination of the associated asset impact value, the likelihood of the associated threat, and the level of vulnerability for that particular risk. The value is in the range 0 to 8.

Figure B.5 shows the assessment of risks A and B; Figure B.6 shows the assessment of risks C, D, and E; Figure B.7 shows the assessment of risks F, G, and H. The circle in the figures show that risks assessed value.

Figure B.8 shows a summary of the risks and the values.

Analysis

As mentioned earlier in this chapter, the analysis stage is where security services are assigned for each of the risks. Figure B.9 shows the completed form for this stage.

To get these results, the typical threats shown in the top of the image were transferred down to the bottom of the image depending on what threats each risk related to. The mapping between risks and threats are shown above in Figure B.3.

The SecSDM form suggests that each mechanism have labels B, S, and ES for basic, standard, and extra strong support respectively. In this form, I

have just used an X to mark which service is needed, and all mappings are treated equally.

Design

The design stage is where security *mechanisms* are mapped to the security services assigned in the analysis stage. These mappings are according to ISO 7498-2 [34].

The following group of figures show the mechanisms needed for each risk. Figure B.10 shows risks A and B; Figure B.11 shows risks C and D; Figure B.12 shows risks E and F; Figure B.13 shows risks G and H.

The SecSDM form suggests that each mechanism have labels B, S, and ES for basic, standard, and extra strong support respectively. In this form, I have just used an X to mark which mechanism is needed, and all mappings are treated equally.

These mappings were applied directly from the mappings shown in Figure A.8 earlier in the chapter.

Figure B.14 shows a summary of these mechanisms for each risk. Instead of an X as the form says, I have used background shading for viewing purposes.

Implementation

This stage did not have a completed form as the SecSDM lifecycle focused on .NET components, while this implementation used Java. The blank form is available in figure A.10.

Section 3.1 shows a range of possible options of different cryptographical libraries, and the choice this implementation used.

Chapter 6

Performance Analysis

This chapter outlines the experiments performed and the results gathered and a discussion of their relevance. The tests outlined in this chapter focus on performance only, while chapter 5 covers the verification and validation tests for the design and implementation.

The chapter is be split as follows:

- Section 6.1 will outline the experimentation setup
- Section 6.2 will present the results of the experiments
- Section 6.3 will summarise the findings.

6.1 Experimentation Setup

This section outlines the experimental setup of the performance results. This setup will include descriptions of the operating environment, the test framework, and the test input and output formats.

6.1.1 Operating Environment

This section outlines the environment that the tests were run in. This includes the test machine specifications, operating system, Java versions, and also a brief discussion on aspects of the environment such as the the load of the machine and the source of randomness.

There were three machines used in the testing. The machines were stock Dell Optiplex GX780's. They had the following specifications:

- Intel Core 2 Duo
- 4GB DDR Memory
- Running Debian Squeeze (stable)
- Linux Kernel 2.6.32-5
- Java version OpenJDK Runtime Environment 1.6.0_18 (IcedTea6 1.8.7) (6b18-1.8.7-2 squeeze1) (Server VM build 14.0-b16)

The tests were performed with a network consisting of only localhost. Each test was run by itself, with minimal load along side the test (achieved by running a minimal set of services). The reasoning behind this decision was to focus on the performance of the implementation rather than the network throughput and effects that it had on the protocols performance.

The source of randomness in the Java library was the NativePRNG with a SHA1PRNG helper. This means that the /dev/{u,}random files are used as the main source of randomness. To keep the entropy levels in these files at a usable level, the HAVEGE daemon is used [78]. HAVEGE is detailed more in Section 3.1.3.

6.1.2 Test Framework

The testing performed was a simple set of transactions with a chain length of two. This means that there are four entities in the system: The TGC; a supplier; a reseller; and a customer.

The purpose of this test is to perform all functions of the protocol, including network connectivity (only on localhost though). The inputs to the test are different key sizes to use, and the outputs are the timings involved for the test to run.

The test is run for the key sizes within the range of 256 bits to 4098 bits, with an increment of 256 bits. The lower bound was chosen as 256 as one of the ElGamal encryption algorithm requires a size of at least 192 bits. The upper bound was chosen to provide a value that will be useful for several years after publication. The increment was chosen to give enough granularity but not too much unnecessary noise. This means that any strange results should be seen if any occur.

Each key size is run through the test framework 100 times, which was shown to be enough to get a p-value of 0.05.

The timings are collected by using the unix utility time and recording the *user* time for each run of each key size test. The user time is the time spent using the CPU and gives a better estimate of the running time with no load. The results from this may not be accurate if the machine comes under especially large load. This is unlikely to happen, as these machines were only used for running the tests.

The process taken for the test is as follows:

1. TGC Generates ElGamal parameters p, q, and g
2. TGC Generates key pair for signing and encryption

3. TGC becomes public using a free port on localhost, sharing the parameters and the public key

4. The supplier and reseller connects to the TGC and retrieves the parameters and public key. They then become public by using free ports on localhost.

5. The supplier registers a test item with the TGC, after generating a key pair for use with that item

6. The reseller purchases the item off the supplier, which involved a tag request from the supplier to the TGC. The reseller receives this tag

7. The reseller verifies that the tag is correct.

8. The customer purchases the item off the reseller, which involves the reseller using the existing tag to request a new tag with the TGC and perform a zero knowledge proof. The customer then receives the new tag

9. The customer verifies that the tag is signed by the TGC and the hash and licence correspond to the received item

6.2 Performance Results

This section presents the results for the tests outlined in section 6.1. Graphs of the results appear in-line. Each figure will be discussed in this section.

A note on the image titles. They may contain a computer name such as M108, M109, and M10B. These are simply the names of the computer that test was run on to generate that graph. As mentioned in section 6.2.1, the performance of all the machines were similar and the graph for two of the machines are ommitted in most of the results. These are available on

request.

6.2.1 Total time taken

Figures 6.1, 6.2, and 6.3 show the total time taken for the experiment on each machine. Along the x axis the key sizes are shown, and along the y axis the time taken in seconds is shown. The time axis is presented in a logarithmic scale.

The actual data is shown as black circles, and the average and one standard deviation around the average are shown by lines coloured blue and red respectively.

Figures 6.1, 6.2, and 6.3 show that the time taken to execute the test transactions is sub-exponential in nature, and that the data is quite widely spread. The logarithmic nature is expected from the protocol as it is using exponential functions for the cryptographic parts. The wide spread of data points on the other hand was not expected. This lead to further analysis to try and identify the cause by timing each stage in the protocol. Further figures shown in this section show each of the stages in order to break down each sub-stage of the protocol.

Another point to note is that the data looks similar on each machine. Due to this, the extra figures for each section of the protocol will only show one machine, although analysis was done on multiple machines.

6.2.2 Initialisation time

Figure 6.4 shows the time taken for the TGC to initialise. This step in the protocol is a one off cost when the TGC is first started. The bottom axis

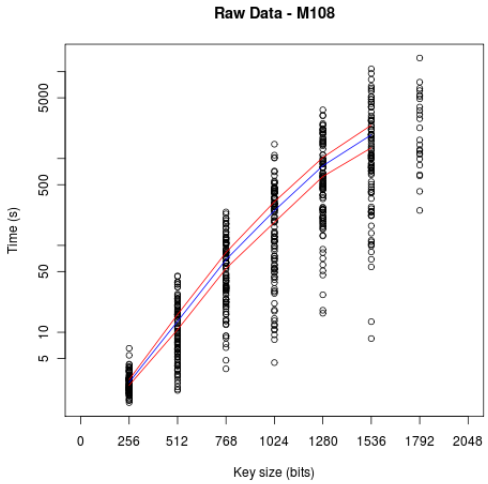

Figure 6.1: Raw results with 99% confidence interval, M108

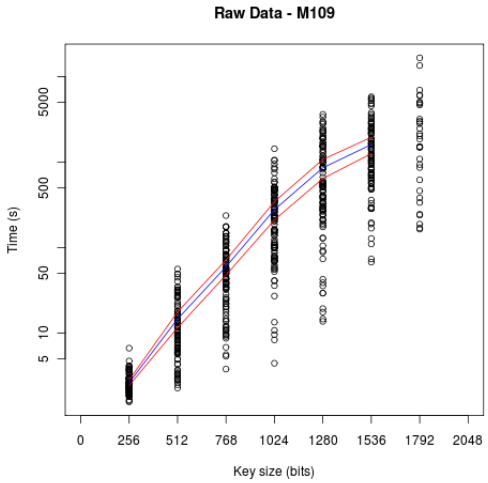

Figure 6.2: Raw results with 99% confidence interval, M109

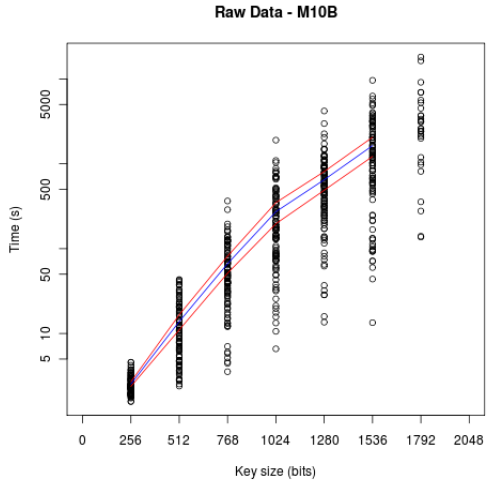

Figure 6.3: Raw results with 99% confidence interval, M10B

shows the key sizes, and the side axis shows the time taken in seconds. Again, the side axis is shown with a logarithmic scale.

Figure 6.4 shows sub-exponential growth and the large spread of the total time. It should be noted that the time is very close to the total time taken though, and could be shown in later figures that this is the stage that is creating the large spread and times for the protocol. This is due to the time taken for the Elgamal encryption algorithm to create the parameters. After these parameters are created, further cryptographic operations should be simpler.

6.2.3 Supplier product registration time

Figure 6.5 shows the time taken for a supplier to register a product with the TGC. This is recorded as the round trip time from the suppliers perspective. The bottom axis shows the key sizes, and the side axis shows the time taken in milli-seconds. The time scale is logarithmic.

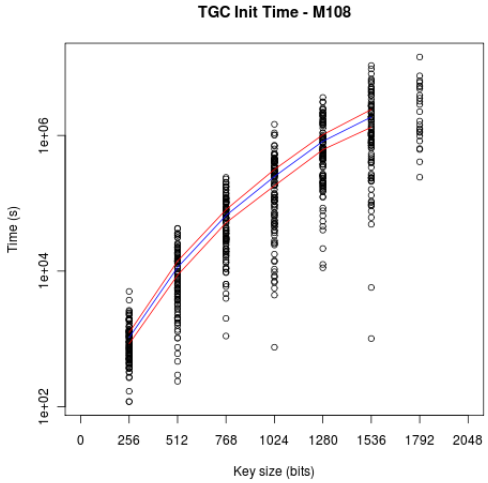

Figure 6.4: TGC Init results with 99% confidence interval, M108

Figure 6.5 shows slightly sub-exponential time as well, but significantly lower times than the initialisation stage. This is a good sign, as it shows that the protocol has low times for the more common functions. This shows that it takes around one to two seconds with a key size of 1024.

6.2.4 Reseller purchase product from supplier time

Figure 6.6 shows the time taken for a reseller to purchase an item from a supplier. This is recorded as the round trip time from the resellers perspective. The axis's are the same as for the register time.

Figure 6.6 shows an exponential function, but with lower times than the registration step. This stage is more common than the registration step as it will be done multiple times, but the product is only registered once. This shows that it takes around a quarter of a second with a key size of 1024.

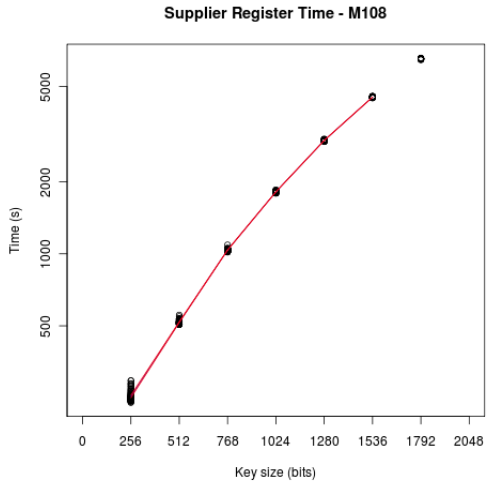

Figure 6.5: Supplier register results with 99% confidence interval, M108

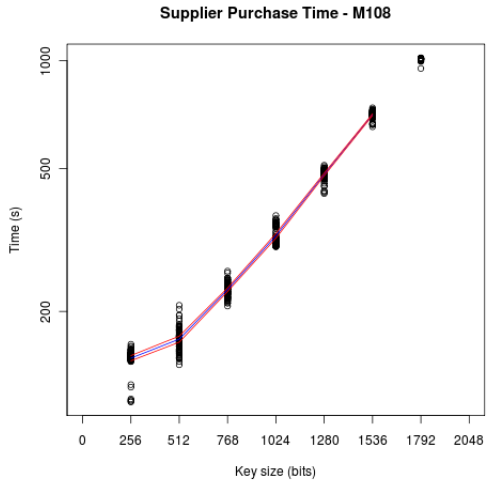

Figure 6.6: Supplier Purchase results with 99% confidence interval, M108

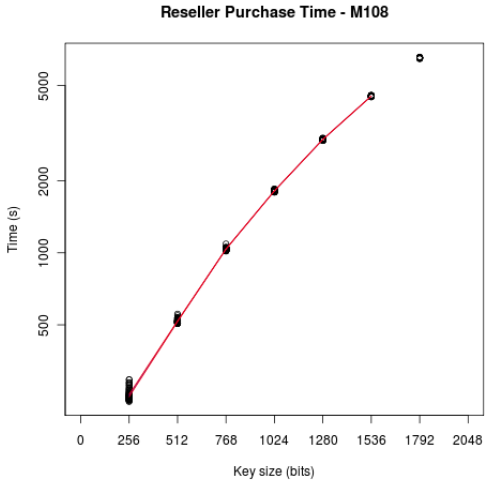

Figure 6.7: 1st Reseller Purchase results with 99% confidence interval, M108

6.2.5 Reseller purchase product from reseller time

Figure 6.7 shows the time taken for a reseller to purchase an item from another reseller. This is recorded as the round trip time from the perspective of the reseller that is purchasing the product. The axis's are the same as above.

This figure shows a slightly sub-exponential function with higher time costs than purchasing from a supplier. This is not the best outcome, as this stage is repeated for each reseller in the chain, whereas the supplier purchase is only done with the top reseller. The time is still not massive given that, taking around one second with a key size of 1024.

This stage is also the time that would be expected from an end user using the system.

6.2.6 Varying other environmental variables

This section provides some more results from which conclusions about different setups may be drawn.

The results are showing the different costs of running different parts of the protocol, both using the HAVEGE daemon to gather entropy, and without.

Figure 6.8 shows the time taken for the protocol to run the one time key parameter generation. This relates to figure 6.4, showing the time taken for the initialisation of the TGC. Figure 6.8 was run without the HAVEGE daemon running to gather random entropy. This image shows a fairly linear growth with some outliers. These outliers always appeared as the first run of the batch. This behaviour seemed like there was some anomalies with using the Java system, by having outliers which may be caused by the VM startup cost. This startup cost most likely includes the time to fill the entropy pool for the random functions needed to generate the parameters.

Figure 6.9 is also showing the initialisation time, but this time with the HAVEGE daemon running. As with figure 6.8, this figure shows a fairly linear growth with similar outliers which may be due to the setup cost and the entropy pool needing to be filled the first run. When HAVEGE is run, then the time taken is marginally smaller.

Figures 6.10 and 6.11 show the time take for the OpenSSL tool set to generate the parameters. This test was done to try and take out the overhead of using the Java Bouncycastle libraries for the initialisation step. Again this shows a linear relationship, though with no outliers. The values are about the same, around the 1 second mark.

These extra figures confirm our believe that there may be some anomalies with using the Java system, by having outliers which may be caused by the VM startup cost. This is because when we take the Java system out of the procedure, the outliers are not present.

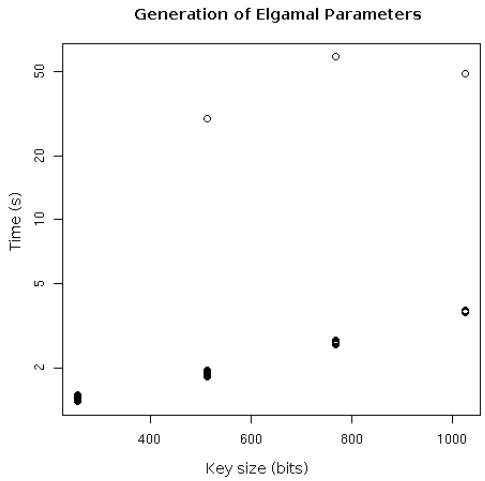

Figure 6.8: Time taken for protocol to run (one time gen) - without HAVEGE

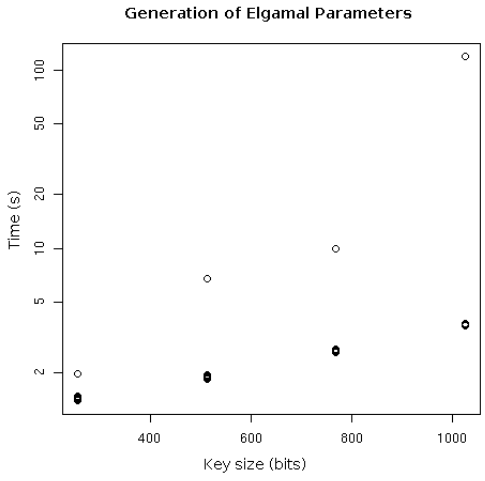

Figure 6.9: Time taken for protocol to run (one time gen) - with HAVEGE

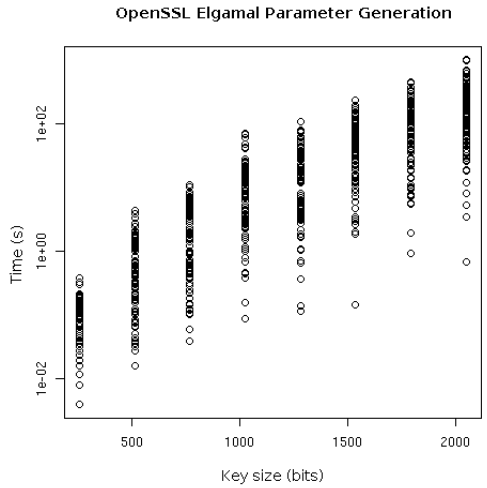

Figure 6.10: Time taken for openssl to generate key parameters - without HAVEGE

The HAVEGE daemon does produce minor benefits, but may be more noticeable for more long running servers. These tests were not run for great lengths so the results may not be applicable to real world usage.

6.3 Summary of Results

The results shown in section 6.2 show us the following:

- Large one time start up cost for initialising the TGC
- Anomalies with the Java cryptographic library
- Exponential times for each stage in the process
- Moderate timings for transferring tags between entities

Those results are the main points that will be summarised. A more detailed discussion is in section 6.2.

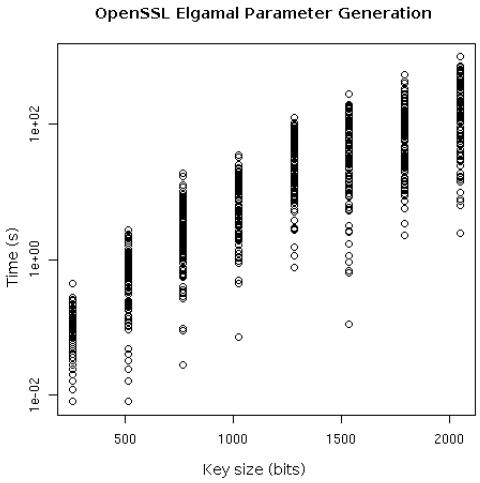

Figure 6.11: Time taken for openssl to generate key parameters - with HAVEGE

The one time start up cost is one of the biggest downfalls of the performance of this implementation. Though it is just a one time cost when the system is first initialised. Improvements on the underlying cryptographic library, possibly using a different language, and many other factors could make an improvement on the timing of this stage. Section 6.2.6 describes how the conclusions of the use of Java was causing outliers to be present, and shows that taking Java out of the generation procedure for the initialization these outliers were not present.

Each stage was shown to be exponential in time, which was to be expected as the cryptographic functions rely on exponential functions to work. Choosing an adequate key size as a time cost and security tradeoff is important. A key size of 1024 bits showed reasonable times for each stage in the system, with faster computers the key size could possibly be increased.

The times for transferring tags between entities using a 1024 bit key size showed performance of about one to two seconds round trip time. This

delay is acceptable to users dealing with security transactions.

A note should be made about possible degradation of the performance when anonymity is used, and also network speeds. The experiments here focussed only on running the tests on a single machine with no network traffic.

If a second computer is involved, then the cryptographic load is shared between the two systems, and some network latency would be introduced. The size of the tag being transferred is rather small, and the size of the product would be the main cost of network bandwidth. Further tests could be run to determine what the speed would be when using two (or more) machines over a network, and what impact different types of networks introduce.

My hypothesis is that the time taken is mainly in the cryptographic functions, and would be shared between the two systems which would make the overall *protocol* time smaller, but the introduced network latency would likely bring it up to the same level. I think that the type of network would not play a large part in the timing costs, but would still need taking into account when deciding on key sizes.

The issue of anonymity is solved by the use of a network such as the TOR network [15]. Further tests could be run to see the impact of using such a network would be. Ries et. al. have looked into a comparison the latency of different types of anonymity networks [72].

Chapter 7

Conclusions and Future Work

This thesis describes the process of developing an implementation of a secure protocol. The main aspects of the thesis is to evaluate between different processes that could possible be used when designing and implementing secure software.

This chapter is structured as follows: Section 7.1 will reiterate the research aims, and Section 7.2 reiterates the thesis outputs and discuss how the aims were met;Section 7.3 discusses possible areas for future work.

7.1 Research Aims

The aims of this research were to implement a prototype of the security protocol discussed in Chapter 3, and to compare and discuss the different processes for developing secure software and their respective benefits and disadvantages. The comparison helps us determine engineering best practices for implementing security protocols. These were used in this implementation.

7.2 Thesis Outputs

The outputs of this thesis are described in this section. They are described in greater detail in further chapters, and Chapter 7 discusses how they were used to obtain the research aims above.

7.2.1 Prototype of the Security Protocol

The prototype was developed in the Java programming language, using the BouncyCastle cryptographic library [86]. The prototype was implemented using secure design practises in mind, and was made to be extendable for alternative implementations.

The outline of the development procedure is presented in chapter 4.

7.2.2 Comparison of Secure Development Processes

A comparison between different secure design approaches is made. Designs such as CLASP (Comprehensive, Lightweight Application Security Process) [65], TSP-Secure (Team Software Process - Secure) [81], SDL (Secure Development Lifecycle) [30], and SecSDM (Secure Software Development Methodology) [23] were presented, along with advantages and disadvantages between them.

In addition to complete processes for developing secure software, a selection of additional tools or partial processes is presented with a discussion on how they can fit into the entire development process.

The comparison was discussed in chapter 3.

The SecSDM approach was then used to show how to validate the implementation of the prototype against security requirements. This approach

was then compared with an intuitive approach of implementation and a discussion on the advantages and disadvantages of each approach is made.

The discussion of the SecSDM approach was made in chapter 5. The value of the use of SecSDM was useful in validating the security of the actual implementation of the prototype. It allows us to confirm that engineering best practices were followed.

7.2.3 Performance Evaluation of the Prototype

The performance measure of the protocol is carried out to gain knowledge on where bottle-necks lie, and how long common tasks in the protocol take.

The measurements took the total time of the protocol running through a test process, and also the time taken for each of the components. The evaluation of these results is discussed about usability and possible further testing. To obtain the aims mentioned above, this research had the following outputs:

- A prototype of the security protocol, in the form of a modular library
- A comparison of different processes for developing secure software
- An example of using parts of the SecSDM [23] process for developing secure software
- A measurement of the protocol performance, with an evaluation

The prototype was developed in the Java programming language, using the BouncyCastle cryptographic library [86]. The prototype was implemented using secure design practises in mind, and was made to be extendable for alternative implementations.

A comparison between different secure design approaches is made. Designs such as CLASP (Comprehensive, Lightweight Application Security Process) [65], TSP-Secure (Team Software Process - Secure) [81], SDL (Secure Development Lifecycle) [30], and SecSDM (Secure Software Development Methodology) [23] are presented, along with advantages and disadvantages between them.

In addition to complete processes for developing secure software, a selection of additional tools or partial processes is presented with a discussion on how they can fit into the entire development process.

The SecSDM approach was then used to show how to validate the implementation of the prototype against security requirements. This approach was then compared with an intuitive approach of implementation and a discussion on the advantages and disadvantages of each approach is made.

The performance measure of the protocol is carried out to gain knowledge on where bottle-necks lie, and how long common tasks in the protocol take.

The measurements took the total time of the protocol running through a test process, and also the time taken for each of the components. The evaluation of these results is discussed about usability and possible further testing.

These outputs are described in further chapters, and discussions of how they were met are in Chapter 7. The following section gives the structure of the thesis.

7.3 Future Work

This section presents some ideas for possible expansion and further research. Further research into the area is important to expand knowledge on different ideas where they may be out of scope with this research.

A major area of possible research is to expand the comparison of design processes to other lifecycles than SecSDM and the intuitive approach. This would open pathways for future implementations of security based approaches with different ideas on what makes a good design process.

Another area of expansion is to extend the experimental process to include other factors such as network latency, and latency due to use of anonymous networks.

As this implementation was just a prototype, further design decisions may be made on the protocol which would require new implementations, or extensions of this prototype.

Appendix A

Blank SecSDM Forms

INVESTIGATION PHASE	
STEP 1 : Information Asset Identification and Valuation (Whitman & Mattord, 2003). For example: customer orders, service requests, employee salaries, EDI documents, etc.	
STEP 1a : Information Asset Identification	
Which information assets are the most **critical** to the **success** of the proposed software application?	1. 2. 3.
Which information assets pertaining to the proposed software application generate the most **revenue**?	1. 2. 3.
Which information assets pertaining to the proposed software application generate the most **profitability**?	1. 2. 3.
Which information assets pertaining to the proposed software application would be the most **expensive to replace**?	1. 2. 3.
Which information assets pertaining to the proposed software application would be the most **expensive to protect**?	1. 2. 3.
Which information assets pertaining to the proposed software application would be the most **embarrassing** or cause the greatest **liability if revealed**?	1. 2. 3.
STEP 1b : Information Asset Valuation	
From the information provided above, select and prioritise the FIVE most important information assets pertaining to the proposed software application and indicate their respective Asset Impact Values (where 0=negligible, 1=low, 2=medium, 3=high, 4=critical) by placing an 'X' in the appropriate cells (one 'X' per asset).	

Information Assets	Asset Impact Value				
	0 Negligible	1 Low	2 Medium	3 High	4 Critical
Asset A ……..					
Asset B ……..					
Asset C ……..					
Asset D ……..					
Asset E ……..					

Figure A.1: SecSDM Investigation Stage, Step 1

INVESTIGATION PHASE				
STEP 2 : Threat Identification and Assessment (Whitman & Mattord, 2003) *A wide variety of threats face an organisation's information and it's information systems. Each of these threats has the potential to attack any of the information assets previously identified. The following questions need to be addressed:* ∞ *Which threats represent the most danger to the information assets pertaining to the proposed software application?* ∞ *How much would it cost to recover from a successful attack ?* ∞ *Which of the threats would require the greatest expenditure to prevent ?* *In order to answer these questions, for each of the common threats listed below, estimate it's level of likelihood, frequency and potential impact to the information assets pertaining to the proposed software application (by placing an 'X' in the appropriate cell).*				
Common Threats (ISO/IEC TR 13335-3:1998)	**Level of Likelihood/Frequency/Impact**			
	LOW	**MED**	**HIGH**	**N/A**
Theft of information (Deliberate e.g. Illegal information disclosure)				
Use of system by unauthorised users (Deliberate or Accidental e.g. Unauthorised access by competitors)				
Use of system in an unauthorised manner (Deliberate or Accidental e.g. Unauthorised data collection)				
Masquerading of user identity (Deliberate e.g. Malicious Hacking)				
Malicious software attacks (Deliberate or Accidental e.g. viruses, worms, DoS)				
User errors (Deliberate or Accidental e.g. Invalid/inaccurate data entry)				
Repudiation (Deliberate e.g. Denial of having performed transaction)				
Technical software failures or errors (Deliberate or Accidental e.g. Bugs, code problems, unknown loopholes)				
Other …… ……				
Other …… ……				

Figure A.2: SecSDM Investigation Stage, Step 2

INVESTIGATION PHASE					
STEP 3 : Risk (Asset/Threat) Identification (Whitman & Mattord, 2003) *By identifying the most critical asset/threat relationships one is able to ascertain the risks most likely to impact the proposed software application.* *Identify the EIGHT most critical risks (asset/threat relationships) by placing a letter from 'a' to' h' in the appropriate cell.*					
ASSET/THREAT RELATIONSHIP					
Common Threats (ISO/IEC TR 13335-3:1998) (refer to Step 2)	**Information Assets (refer to Step 1)**				
	Asset A	**Asset B**	**Asset C**	**Asset D**	**Asset E**
Theft of information					
Use of system by unauthorised users					
Use of system in an unauthorised manner					
Masquerading of user identity					
Malicious software attacks					
User errors					
Repudiation					
Technical software failures or errors					
Other ……					
Other ……					

Figure A.3: SecSDM Investigation Stage, Step 3

INVESTIGATION PHASE			
STEP 4 : Determine Level of Vulnerability *In order to determine the level of vulnerability for each risk (asset/threat relationship 'a' to 'h') as identified in Step 3, you need to consider the likelihood that the risk may materialise, taking the current situation and controls into account. This can be done by plotting the level of vulnerability for each risk (asset/threat relationship 'a' to 'h') by placing an 'X' in the appropriate cell.*			
Risks (refer to Step 3)	**Level of Vulnerability**		
	LOW	**MEDIUM**	**HIGH**
	The asset is quite well protected against the threat, therefore the vulnerability is low.	*The asset is exposed to some degree and is not that well protected, therefore the vulnerability is medium.*	*The asset is exposed to a large degree and is not well protected at all, therefore the vulnerability is high.*
Risk A (Asset/Threat Relationship 'a')			
Risk B (Asset/Threat Relationship 'b')			
Risk C (Asset/Threat Relationship 'c')			
Risk D (Asset/Threat Relationship 'd')			
Risk E (Asset/Threat Relationship 'e')			
Risk F (Asset/Threat Relationship 'f')			
Risk G (Asset/Threat Relationship 'g')			
Risk H (Asset/Threat Relationship 'h')			

Figure A.4: SecSDM Investigation Stage, Step 4

INVESTIGATION PHASE

STEP 5 : Risk Assessment (matrix from ISO/IEC TR 13335-3:1998)

In order to determine the actual measure of risk, the relevant Asset Impact Value, level of vulnerability and the likelihood of the threat must be considered for each risk identified in Step 3. The appropriate row in the matrix is identified by the Asset Impact Value of the particular asset (simply carry this over from Step 1). The appropriate column is identified by the likelihood, frequency and potential impact of the particular threat and the level of vulnerability. Simply transfer the level of vulnerability as determined in Step 4, and the likelihood, frequency and potential impact of the particular threat, as determined in Step 2. This must be performed for each risk as identified in Step 3 by placing an 'X' in the appropriate cell.

RISK A (as per Asset/Threat Relationship 'a' in Step 3):

……..

……..

		Likelihood/Frequency/Potential Impact of Threat								
		LOW			MEDIUM			HIGH		
		Level of Vulnerability			Level of Vulnerability			Level of Vulnerability		
		LOW	MED	HIGH	LOW	MED	HIGH	LOW	MED	HIGH
Asset Impact Value	Negligible 0	0	1	2	1	2	3	2	3	4
	Low 1	1	2	3	2	3	4	3	4	5
	Medium 2	2	3	4	3	4	5	4	5	6
	High 3	3	4	5	4	5	6	5	6	7
	Critical 4	4	5	6	5	6	7	6	7	8

RISK B (as per Asset/Threat Relationship 'b' in Step 3):

……..

……..

		Likelihood/Frequency/Potential Impact of Threat								
		LOW			MEDIUM			HIGH		
		Level of Vulnerability			Level of Vulnerability			Level of Vulnerability		
		LOW	MED	HIGH	LOW	MED	HIGH	LOW	MED	HIGH
Asset Impact Value	Negligible 0	0	1	2	1	2	3	2	3	4
	Low 1	1	2	3	2	3	4	3	4	5
	Medium 2	2	3	4	3	4	5	4	5	6
	High 3	3	4	5	4	5	6	5	6	7
	Critical 4	4	5	6	5	6	7	6	7	8

Figure A.5: SecSDM Investigation Stage, Step 5

INVESTIGATION PHASE		
STEP 6 : Risk Prioritisation *For each risk identified in Step 5, transfer the risk description and the risk value to the table below. This step completes the risk analysis part of the Investigation Phase.*		
	Risk Description (Asset/Threat relationship)	**Risk Value**
Risk A		
Risk B		
Risk C		
Risk D		
Risk E		
Risk F		
Risk G		
Risk H		

Figure A.6: SecSDM Investigation Stage, Step 6

ANALYSIS PHASE					
STEP 7 : Identification of Security Services *Typically the following threats would require the services as indicated in the table below.*					
Threats	**Security Services**				
	Identification & Authentication	**Access Control**	**Data Confidentiality**	**Data Integrity**	**Non-repudiation**
Theft of information	X	X			
Use of system by unauthorised users	X	X			
Use of system in an unauthorised manner		X	X		
Masquerading of user identity	X	X			
Malicious software attacks		X		X	
User errors		X		X	
Repudiation	X				X
Technical software failures or errors				X	
STEP 7 : Identification of Security Services *With this knowledge, map each of the EIGHT risks (asset/threat relationships) identified in the Investigation Phase to the envisaged services. For each risk, more than one service may be identified. Indicate the level of protection required by placing a B (for Basic), S (for Standard) or ES (for Extra Strong) in the relevant cells. Not all services will be required to address each individual risk.*					
Risks	**Security Services**				
	Identification & Authentication	**Access Control**	**Data Confidentiality**	**Data Integrity**	**Non-repudiation**
Risk A					
Risk B					
Risk C					
Risk D					
Risk E					
Risk F					
Risk G					
Risk H					

Figure A.7: SecSDM Analysis Stage

DESIGN PHASE					
STEP 8: Mapping of Security Services to Security Mechanisms (according to ISO 7498-2) Typically the security services are implemented through the security mechanisms as indicated in the table below.					
Security Mechanisms	**Security Services**				
	Identification & Authentication	**Access Control**	**Data Confidentiality**	**Data Integrity**	**Non-repudiation**
Encipherment/ Encryption	X		X	X	
Digital Signatures	X			X	X
Access Control Mechanisms		X			
Data Integrity Mechanisms				X	X
Authentication Exchange Mechanisms	X				
Traffic Padding			X		
Routing Control			X	X	
Notarisation				X	X

Figure A.8: SecSDM Design stage (sample table)

DESIGN PHASE

STEP 9 : Summary of Findings

In the table below indicate which security services and mechanisms would be required to address the specified risks (A, B, C, D, E, F, G, H). Simply place an 'X' in the appropriate cells.

Security Mechanisms	Security Services																			
	Identification & Authentication				Access Control				Data Confidentiality				Data Integrity				Non-repudiation			
Encipherment/ Encryption	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H
Digital Signatures	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H
Access Control Mechanisms	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H
Data Integrity Mechanisms	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H
Authentication Exchange	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H
Traffic Padding	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H
Routing Control	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H
Notarisation	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H

Figure A.9: SecSDM Design stage (final output)

IMPLEMENTATION PHASE				
STEP 10 : Mapping of Security Mechanisms to .Net and Other Components *For each security mechanism, indicate the risks (A, B, C, D, E, F, G, H) for which it is relevant, by placing an 'X' next to the appropriate risk (transferred directly from Step 9). In addition, identify the specific .Net or other components through which the security mechanisms will be implemented.*				
			.Net security components	**Other components**
Security Mechanisms	Encipherment/ Encryption	Risk A Risk B Risk C Risk D Risk E Risk F Risk G Risk H		
	Digital Signatures	Risk A Risk B Risk C Risk D Risk E Risk F Risk G Risk H		
	Access Control Mechanisms	Risk A Risk B Risk C Risk D Risk E Risk F Risk G Risk H		
	Data Integrity Mechanisms	Risk A Risk B Risk C Risk D Risk E Risk F Risk G Risk H		
	Authentication Exchange Mechanisms	Risk A Risk B Risk C Risk D Risk E Risk F Risk G Risk H		
	Traffic Padding	Risk A Risk B Risk C Risk D Risk E Risk F Risk G Risk H		
	Routing Control	Risk A Risk B Risk C Risk D Risk E Risk F Risk G Risk H		
	Notarisation	Risk A Risk B Risk C Risk D Risk E Risk F Risk G		

Figure A.10: SecSDM Implementation stage

Appendix B

Completed SecSDM Forms

INVESTIGATION PHASE	
STEP 1 : Information Asset Identification and Valuation (Whitman & Mattord, 2003). For example: customer orders, service requests, employee salaries, EDI documents, etc.	
STEP 1a : Information Asset Identification	
Which information assets are the most **critical** to the **success** of the proposed software application?	1. TGC Keys
	2. Tags
	3.
Which information assets pertaining to the proposed software application generate the most **revenue**?	1. Products of the system
	2.
	3.
Which information assets pertaining to the proposed software application generate the most **profitability**?	1. Products of the system
	2.
	3.
Which information assets pertaining to the proposed software application would be the most **expensive to replace**?	1. TGC Keys
	2. Supplier Keys
	3
Which information assets pertaining to the proposed software application would be the most **expensive to protect**?	1
	2.
	3.
Which information assets pertaining to the proposed software application would be the most **embarrassing** or cause the greatest **liability if revealed**?	1. TGC Private Key
	2. Supplier Private Key
	3. Reseller Keys or Z value

STEP 1b : Information Asset Valuation

From the information provided above, select and prioritise the FIVE most important information assets pertaining to the proposed software application and indicate their respective Asset Impact Values (where 0=negligible, 1=low, 2=medium, 3=high, 4=critical) by placing an 'X' in the appropriate cells (one 'X' per asset).

Information Assets	Asset Impact Value				
	0 Negligible	1 Low	2 Medium	3 High	4 Critical
Asset A TGC Private Key					X
Asset B Supplier Private Key				X	
Asset C Reseller Private Key			X		
Asset D Reseller Z value			X		
Asset E Tags		X			

Figure B.1: SecSDM Investigation Stage, Step 1, Impact Value of Assets

INVESTIGATION PHASE

STEP 2 : Threat Identification and Assessment (Whitman & Mattord, 2003)
A wide variety of threats face an organisation's information and it's information systems. Each of these threats has the potential to attack any of the information assets previously identified. The following questions need to be addressed:

- *Which threats represent the most danger to the information assets pertaining to the proposed software application?*
- *How much would it cost to recover from a successful attack ?*
- *Which of the threats would require the greatest expenditure to prevent ?*

In order to answer these questions, for each of the common threats listed below, estimate it's level of likelihood, frequency and potential impact to the information assets pertaining to the proposed software application (by placing an 'X' in the appropriate cell).

Common Threats (ISO/IEC TR 13335-3:1998)	**Level of Likelihood/Frequency/Impact**			
	LOW	**MED**	**HIGH**	**N/A**
Theft of information (Deliberate e.g. Illegal information disclosure)	X			
Use of system by unauthorised users (Deliberate or Accidental e.g. Unauthorised access by competitors)		X		
Use of system in an unauthorised manner (Deliberate or Accidental e.g. Unauthorised data collection)			X	
Masquerading of user identity (Deliberate e.g. Malicious Hacking)	X			
Malicious software attacks (Deliberate or Accidental e.g. viruses, worms, DoS)		X		
User errors (Deliberate or Accidental e.g. Invalid/inaccurate data entry)				X
Repudiation (Deliberate e.g. Denial of having performed transaction)	X			
Technical software failures or errors (Deliberate or Accidental e.g. Bugs, code problems, unknown loopholes)		X		
Other ……..……..……..……..……..……... ……..……..……..……..……..……...				
Other ……..……..……..……..……..……... ……..……..……..……..……..……...				

Figure B.2: SecSDM Investigation Stage, Step 2, Likelihood of Common Threats

INVESTIGATION PHASE					
STEP 3 : Risk (Asset/Threat) Identification (Whitman & Mattord, 2003) *By identifying the most critical asset/threat relationships one is able to ascertain the risks most likely to impact the proposed software application.* *Identify the EIGHT most critical risks (asset/threat relationships) by placing a letter from 'a' to' h' in the appropriate cell.*					
ASSET/THREAT RELATIONSHIP					
Common Threats (ISO/IEC TR 13335-3:1998) (refer to Step 2)	**Information Assets (refer to Step 1)**				
	Asset A	**Asset B**	**Asset C**	**Asset D**	**Asset E**
Theft of information	D	F	H	H	
Use of system by unauthorised users	C	E	G	G	
Use of system in an unauthorised manner	A				
Masquerading of user identity					
Malicious software attacks	B				
User errors					
Repudiation					
Technical software failures or errors	C	E	G	G	
Other ……					
Other ……					

Figure B.3: SecSDM Investigation Stage, Step 3, Asset Threat Relationships

INVESTIGATION PHASE			
STEP 4 : Determine Level of Vulnerability *In order to determine the level of vulnerability for each risk (asset/threat relationship 'a' to 'h') as identified in Step 3, you need to consider the likelihood that the risk may materialise, taking the current situation and controls into account. This can be done by plotting the level of vulnerability for each risk (asset/threat relationship 'a' to 'h') by placing an 'X' in the appropriate cell.*			
Risks (refer to Step 3)	**Level of Vulnerability**		
	LOW	**MEDIUM**	**HIGH**
	The asset is quite well protected against the threat, therefore the vulnerability is low.	*The asset is exposed to some degree and is not that well protected, therefore the vulnerability is medium.*	*The asset is exposed to a large degree and is not well protected at all, therefore the vulnerability is high.*
Risk A (Asset/Threat Relationship 'a')		X	
Risk B (Asset/Threat Relationship 'b')		X	
Risk C (Asset/Threat Relationship 'c')	X		
Risk D (Asset/Threat Relationship 'd')	X		
Risk E (Asset/Threat Relationship 'e')	X		
Risk F (Asset/Threat Relationship 'f')	X		
Risk G (Asset/Threat Relationship 'g')		X	
Risk H (Asset/Threat Relationship 'h')		X	

Figure B.4: SecSDM Investigation Stage, Step 4, Risk Vulnerabilities

INVESTIGATION PHASE

STEP 5 : Risk Assessment (matrix from ISO/IEC TR 13335-3:1998)

In order to determine the actual measure of risk, the relevant Asset Impact Value, level of vulnerability and the likelihood of the threat must be considered for each risk identified in Step 3. The appropriate row in the matrix is identified by the Asset Impact Value of the particular asset (simply carry this over from Step 1). The appropriate column is identified by the likelihood, frequency and potential impact of the particular threat and the level of vulnerability. Simply transfer the level of vulnerability as determined in Step 4, and the likelihood, frequency and potential impact of the particular threat, as determined in Step 2. This must be performed for each risk as identified in Step 3 by placing an 'X' in the appropriate cell.

RISK A (as per Asset/Threat Relationship 'a' in Step 3):

……..

Denial of Service of the Tag Generation Centre (Unintended Use)

……...

		Likelihood/Frequency/Potential Impact of Threat								
		LOW			MEDIUM			HIGH		
		Level of Vulnerability			Level of Vulnerability			Level of Vulnerability		
		LOW	MED	HIGH	LOW	MED	HIGH	LOW	MED	HIGH
Asset Impact Value	Negligible 0	0	1	2	1	2	3	2	3	4
	Low 1	1	2	3	2	3	4	3	4	5
	Medium 2	2	3	4	3	4	5	4	5	6
	High 3	3	4	5	4	5	6	5	6	7
	Critical 4	4	5	6	5	6	7	6	(7)	8

RISK B (as per Asset/Threat Relationship 'b' in Step 3):

……..

Denial of Service of the Tag Generation Centre (Malicious Intent)

……...

		Likelihood/Frequency/Potential Impact of Threat								
		LOW			MEDIUM			HIGH		
		Level of Vulnerability			Level of Vulnerability			Level of Vulnerability		
		LOW	MED	HIGH	LOW	MED	HIGH	LOW	MED	HIGH
Asset Impact Value	Negligible 0	0	1	2	1	2	3	2	3	4
	Low 1	1	2	3	2	3	4	3	4	5
	Medium 2	2	3	4	3	4	5	4	5	6
	High 3	3	4	5	4	5	6	5	6	7
	Critical 4	4	5	6	5	(6)	7	6	7	8

Figure B.5: SecSDM Investigation Stage, Step 5, Risks A and B

RISK C (as per Asset/Threat Relationship 'c' in Step 3):

Tag Generation Centre Key Altered

		Likelihood/Frequency/Potential Impact of Threat								
		LOW			MEDIUM			HIGH		
		Level of Vulnerability			Level of Vulnerability			Level of Vulnerability		
		LOW	MED	HIGH	LOW	MED	HIGH	LOW	MED	HIGH
Asset Impact Value	Negligible 0	0	1	2	1	2	3	2	3	4
	Low 1	1	2	3	2	3	4	3	4	5
	Medium 2	2	3	4	3	4	5	4	5	6
	High 3	3	4	5	4	5	6	5	6	7
	Critical 4	4	5	6	(5)	6	7	6	7	8

RISK D (as per Asset/Threat Relationship 'd' in Step 3):

Tag Generation Centre Private Key Stolen

		Likelihood/Frequency/Potential Impact of Threat								
		LOW			MEDIUM			HIGH		
		Level of Vulnerability			Level of Vulnerability			Level of Vulnerability		
		LOW	MED	HIGH	LOW	MED	HIGH	LOW	MED	HIGH
Asset Impact Value	Negligible 0	0	1	2	1	2	3	2	3	4
	Low 1	1	2	3	2	3	4	3	4	5
	Medium 2	2	3	4	3	4	5	4	5	6
	High 3	3	4	5	4	5	6	5	6	7
	Critical 4	(4)	5	6	5	6	7	6	7	8

RISK E (as per Asset/Threat Relationship 'e' in Step 3):

Supplier Key Altered

		Likelihood/Frequency/Potential Impact of Threat								
		LOW			MEDIUM			HIGH		
		Level of Vulnerability			Level of Vulnerability			Level of Vulnerability		
		LOW	MED	HIGH	LOW	MED	HIGH	LOW	MED	HIGH
Asset Impact Value	Negligible 0	0	1	2	1	2	3	2	3	4
	Low 1	1	2	3	2	3	4	3	4	5
	Medium 2	2	3	4	3	4	5	4	5	6
	High 3	3	4	5	(4)	5	6	5	6	7
	Critical 4	4	5	6	5	6	7	6	7	8

Figure B.6: SecSDM Investigation Stage, Step 5, Risks C, D, and E

RISK F (as per Asset/Threat Relationship 'f' in Step 3):

Supplier Private Key Stolen

		Likelihood/Frequency/Potential Impact of Threat								
		LOW			MEDIUM			HIGH		
		Level of Vulnerability			Level of Vulnerability			Level of Vulnerability		
		LOW	MED	HIGH	LOW	MED	HIGH	LOW	MED	HIGH
Asset Impact Value	Negligible 0	0	1	2	1	2	3	2	3	4
	Low 1	1	2	3	2	3	4	3	4	5
	Medium 2	2	3	4	3	4	5	4	5	6
	High 3	(3)	4	5	4	5	6	5	6	7
	Critical 4	4	5	6	5	6	7	6	7	8

RISK G (as per Asset/Threat Relationship 'g' in Step 3):

Reseller Key or Z value Altered

		Likelihood/Frequency/Potential Impact of Threat								
		LOW			MEDIUM			HIGH		
		Level of Vulnerability			Level of Vulnerability			Level of Vulnerability		
		LOW	MED	HIGH	LOW	MED	HIGH	LOW	MED	HIGH
Asset Impact Value	Negligible 0	0	1	2	1	2	3	2	3	4
	Low 1	1	2	3	2	3	4	3	4	5
	Medium 2	2	3	4	(3)	4	5	4	5	6
	High 3	3	4	5	4	5	6	5	6	7
	Critical 4	4	5	6	5	6	7	6	7	8

RISK H (as per Asset/Threat Relationship 'h' in Step 3):

Reseller Private Key or Z value Stolen

		Likelihood/Frequency/Potential Impact of Threat								
		LOW			MEDIUM			HIGH		
		Level of Vulnerability			Level of Vulnerability			Level of Vulnerability		
		LOW	MED	HIGH	LOW	MED	HIGH	LOW	MED	HIGH
Asset Impact Value	Negligible 0	0	1	2	1	2	3	2	3	4
	Low 1	1	2	3	2	3	4	3	4	5
	Medium 2	2	(3)	4	3	4	5	4	5	6
	High 3	3	4	5	4	5	6	5	6	7
	Critical 4	4	5	6	5	6	7	6	7	8

Figure B.7: SecSDM Investigation Stage, Step 5, Risks F, G, and H

INVESTIGATION PHASE		
STEP 6 : Risk Prioritisation *For each risk identified in Step 5, transfer the risk description and the risk value to the table below. This step completes the risk analysis part of the Investigation Phase.*		
	Risk Description (Asset/Threat relationship)	**Risk Value**
Risk A	Denial of Service of the Tag Generation Centre (Unintended Use)	7
Risk B	Denial of Service of the Tag Generation Centre (Malicious Intent)	6
Risk C	Tag Generation Centre Key Altered	5
Risk D	Tag Generation Centre Private Key Stolen	4
Risk E	Supplier Key Altered	4
Risk F	Supplier Private Key Stolen	3
Risk G	Reseller Key or Z value Altered	4
Risk H	Reseller Private Key or Z value Stolen	3

Figure B.8: SecSDM Investigation Stage, Step 6, Summary

ANALYSIS PHASE

STEP 7 : Identification of Security Services
Typically the following threats would require the services as indicated in the table below.

Threats	Security Services				
	Identification & Authentication	Access Control	Data Confidentiality	Data Integrity	Non-repudiation
Theft of information	X	X			
Use of system by unauthorised users	X	X			
Use of system in an unauthorised manner		X	X		
Masquerading of user identity	X	X			
Malicious software attacks		X		X	
User errors		X		X	
Repudiation	X				X
Technical software failures or errors				X	

STEP 7 : Identification of Security Services
With this knowledge, map each of the EIGHT risks (asset/threat relationships) identified in the Investigation Phase to the envisaged services. For each risk, more than one service may be identified. Indicate the level of protection required by placing a B (for Basic), S (for Standard) or ES (for Extra Strong) in the relevant cells. Not all services will be required to address each individual risk.

Risks	Security Services				
	Identification & Authentication	Access Control	Data Confidentiality	Data Integrity	Non-repudiation
Risk A		X	X		
Risk B		X		X	
Risk C	X	X		X	
Risk D	X	X			
Risk E	X	X		X	
Risk F	X	X			
Risk G	X	X		X	
Risk H	X	X			

Figure B.9: SecSDM Analysis Stage

DESIGN PHASE

STEP 8 : Mapping of Security Services to Security Mechanisms (according to ISO 7498-2)
For each risk, identify the specific mechanisms that would be implemented to support the security services required. Indicate the level of protection required by placing a B (for Basic), S (for Standard) or ES (for Extra Strong) in the relevant cells. Not all security services and mechanisms will be required to address each individual risk.

RISK A (as per Asset/Threat Relationship 'a' in Step 3):
……
Denial of Service of the Tag Generation Centre (Unintended Use)
……..

Security Mechanisms	Security Services				
	Identification & Authentication	Access Control	Data Confidentiality	Data Integrity	Non-repudiation
Encipherment/ Encryption			X		
Digital Signatures					
Access Control Mechanisms		X			
Data Integrity Mechanisms					
Authentication Exchange					
Traffic Padding			X		
Routing Control			X		
Notarisation					

RISK B (as per Asset/Threat Relationship 'b' in Step 3):
……
Denial of Service of the Tag Generation Centre (Malicious Intent)
……..

Security Mechanisms	Security Services				
	Identification & Authentication	Access Control	Data Confidentiality	Data Integrity	Non-repudiation
Encipherment/ Encryption				X	
Digital Signatures				X	
Access Control Mechanisms		X			
Data Integrity Mechanisms				X	
Authentication Exchange					
Traffic Padding					
Routing Control				X	
Notarisation				X	

Figure B.10: SecSDM Design Stage, Risks A and B

RISK C (as per Asset/Threat Relationship 'c' in Step 3):
……
Tag Generation Centre Key Altered
……..

Security Mechanisms	Security Services				
	Identification & Authentication	Access Control	Data Confidentiality	Data Integrity	Non-repudiation
Encipherment/ Encryption	X			X	
Digital Signatures	X			X	
Access Control Mechanisms		X			
Data Integrity Mechanisms				X	
Authentication Exchange	X				
Traffic Padding					
Routing Control				X	
Notarisation				X	

RISK D (as per Asset/Threat Relationship 'd' in Step 3):
……
Tag Generation Centre Private Key Stolen
……..

Security Mechanisms	Security Services				
	Identification & Authentication	Access Control	Data Confidentiality	Data Integrity	Non-repudiation
Encipherment/ Encryption	X				
Digital Signatures	X				
Access Control Mechanisms		X			
Data Integrity Mechanisms					
Authentication Exchange	X				
Traffic Padding					
Routing Control					
Notarisation					

Figure B.11: SecSDM Design Stage, Risks C and D

RISK E (as per Asset/Threat Relationship 'e' in Step 3):
……
Supplier Key Altered
……..

Security Mechanisms	Security Services				
	Identification & Authentication	Access Control	Data Confidentiality	Data Integrity	Non-repudiation
Encipherment/ Encryption	X			X	
Digital Signatures	X			X	
Access Control Mechanisms		X			
Data Integrity Mechanisms				X	
Authentication Exchange	X				
Traffic Padding					
Routing Control				X	
Notarisation				X	

RISK F (as per Asset/Threat Relationship 'f' in Step 3):
……
Supplier Private Key Stolen
……..

Security Mechanisms	Security Services				
	Identification & Authentication	Access Control	Data Confidentiality	Data Integrity	Non-repudiation
Encipherment/ Encryption	X				
Digital Signatures	X				
Access Control Mechanisms		X			
Data Integrity Mechanisms					
Authentication Exchange	X				
Traffic Padding					
Routing Control					
Notarisation					

Figure B.12: SecSDM Design Stage, Risks E and F

RISK G (as per Asset/Threat Relationship 'g' in Step 3):
……
Reseller Key or Z value Altered
……..

Security Mechanisms	Security Services				
	Identification & Authentication	Access Control	Data Confidentiality	Data Integrity	Non-repudiation
Encipherment/ Encryption	X			X	
Digital Signatures	X			X	
Access Control Mechanisms		X			
Data Integrity Mechanisms				X	
Authentication Exchange	X				
Traffic Padding					
Routing Control				X	
Notarisation				X	

RISK H (as per Asset/Threat Relationship 'h' in Step 3):
……
Reseller Private Key or Z value Stolen
……..

Security Mechanisms	Security Services				
	Identification & Authentication	Access Control	Data Confidentiality	Data Integrity	Non-repudiation
Encipherment/ Encryption	X				
Digital Signatures	X				
Access Control Mechanisms		X			
Data Integrity Mechanisms					
Authentication Exchange	X				
Traffic Padding					
Routing Control					
Notarisation					

Figure B.13: SecSDM Design Stage, Risks G and H

DESIGN PHASE

STEP 9 : Summary of Findings

In the table below indicate which security services and mechanisms would be required to address the specified risks (A, B, C, D, E, F, G, H). Simply place an 'X' in the appropriate cells.

Security Mechanisms	Security Services																			
	Identification & Authentication				Access Control				Data Confidentiality				Data Integrity				Non-repudiation			
Encipherment/ Encryption	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H
Digital Signatures	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H
Access Control Mechanisms	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H
Data Integrity Mechanisms	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H
Authentication Exchange	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H
Traffic Padding	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H
Routing Control	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H
Notarisation	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H	E	F	G	H

Figure B.14: SecSDM Design Stage, Summary

Bibliography

[1] 4FRIENDSONLY. New technologies for virtual goods. http://wwww.4fo.de/en/potato.htm, accessed in Oct 2010.

[2] ALBERTS, C. J., AND DOROFEE, A. *Managing Information Security Risks: The OCTAVE Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[3] BLAKE-WILSON, S., NYSTROM, M., HOPWOOD, D., MIKKELSEN, J., AND WRIGHT, T. Transport Layer Security (TLS) Extensions. RFC 3546 (Proposed Standard), June 2003. Obsoleted by RFC 4366 [4].

[4] BLAKE-WILSON, S., NYSTROM, M., HOPWOOD, D., MIKKELSEN, J., AND WRIGHT, T. Transport Layer Security (TLS) Extensions. RFC 4366 (Proposed Standard), Apr. 2006. Obsoleted by RFCs 5246 [13], 6066 [16], updated by RFC 5746 [71].

[5] BOEHM, B. A Spiral Model of Software Development and Enhancement. *SIGSOFT Softw. Eng. Notes 11*, 4 (1986), 14–24.

[6] BROWN, M., AND HOUSLEY, R. Transport Layer Security (TLS) Authorization Extensions. RFC 5878 (Experimental), May 2010.

[7] BURN, O.

[8] CERT. Secure Coding Standards. http://www.cert.org/secure-coding/scstandards.html, accessed in Dec 2010.

[9] CERT. TSP-Secure. `http://www.cert.org/secure-coding/secure.html`, accessed in Dec 2010.

[10] DETLEFS, D., NELSON, G., AND SAXE, J. B. Simplify: a theorem prover for program checking. *J. ACM 52*, 3 (2005), 365–473.

[11] DIERKS, T., AND ALLEN, C. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), Jan. 1999. Obsoleted by RFC 4346 [12], updated by RFCs 3546 [3], 5746 [71].

[12] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), Apr. 2006. Obsoleted by RFC 5246 [13], updated by RFCs 4366 [4], 4680 [74], 4681 [75], 5746 [71].

[13] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746 [71], 5878 [6].

[14] DINGLEDINE, R. The Free Haven Project: Design and Deployment of an Anonymous Secure Data Haven. Master's thesis, MIT, MA, USA, June 2000.

[15] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: the second-generation onion router. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 21–21.

[16] EASTLAKE 3RD, D. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066 (Proposed Standard), Jan. 2011.

[17] EASTLAKE 3RD, D., SCHILLER, J., AND CROCKER, S. Randomness Requirements for Security. RFC 4086 (Best Current Practice), June 2005.

[18] EL GAMAL, T. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology* (New York, NY, USA, 1985), Springer-Verlag New York, Inc., pp. 10–18.

[19] ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering* (New York, NY, USA, 1999), ACM, pp. 213–224.

[20] FERGUSON, N., AND SCHNEIER, B. *Practical Cryptography*, 1 ed. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[21] FLANAGAN, C., AND LEINO, K. R. M. Houdini, an Annotation Assistant for ESC/Java. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity* (London, UK, 2001), Springer-Verlag, pp. 500–517.

[22] FREE SOFTWARE FOUNDATION. GNU Coding Standards. `http://www.gnu.org/prep/standards`, accessed in Oct 2010.

[23] FUTCHER, L., AND VON SOLMS, R. SecSDM: A Model for Integrating Security into the Software Development Life Cycle. In *Fifth World Conference on Information Security Education*, L. Futcher and R. Dodge, Eds., vol. 237 of *IFIP International Federation for Information Processing*. Springer Boston, 2007, pp. 41–48.

[24] FUTCHER, L., AND VON SOLMS, R. Guidelines for Secure Software Development. In *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology* (New York, NY, USA, 2008), SAICSIT '08, ACM, pp. 56–65.

[25] GOLDWASSER, S., MICALI, S., AND RACKOFF, C. The knowledge complexity of interactive proof-systems. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing* (New York, NY, USA, 1985), STOC '85, ACM, pp. 291–304.

[26] GRIMM, R., AND NUTZEL, J. Potato system and signed media format – an alternative approach to online music business. In *Proceedings of the 3rd International Conference on Web Delivering of Music* (Germany, Sept. 2003), WEDELMUSIC '03, Fraunhofer Publica [http://publica.fraunhofer.de/oai.har], pp. 23–26.

[27] GUTMANN, P. cryptlib Encryption Toolkit. `http://www.cs.auckland.ac.nz/~pgut001/cryptlib/`, accessed in Jul 2012.

[28] GUTTERMAN, Z., PINKAS, B., AND REINMAN, T. Analysis of the linux random number generator. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2006), SP '06, IEEE Computer Society, pp. 371–385.

[29] HOVEMEYER, D., AND PUGH, W. Finding Bugs is Easy. *SIGPLAN Not. 39*, 12 (2004), 92–106.

[30] HOWARD, M., AND LIPNER, S. *The Security Development Lifecycle*. Microsoft Press, Redmond, WA, USA, 2006.

[31] HUBBERS, E., OOSTDIJK, M., AND POLL, E. Implementing a Formally Verifiable Security Protocol in Java Card. In *Proceedings of the First International Conference on Security in Pervasive Computing, volume 2802 of Lecture Notes in Computer Science* (2003), Springer-Verlag, pp. 213–226.

[32] HUNT, J. *The Unified Process for Practitioners: Object-Oriented Design, UML and Java*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.

[33] ISO 13335-3:1998. Information technology – guidelines for the management of it security – part 3: Techniques for the management of it security. ISO, Geneva, Switzerland.

[34] ISO 7498-2:1989. Information processing systems – open systems interconnection – basic reference model – part 2: Security architecture. ISO, Geneva, Switzerland.

[35] JÄNICKE, L. PRNGD - Pseudo Random Number Generator Daemon. `http://prngd.sourceforge.net/`, accessed in Jul 2012.

[36] JOHNSON, R., HOELLER, J., ARENDSEN, A., RISBERG, T., AND KOPYLENKO, D. *Professional Java Development with the Spring Framework*. Wrox Press Ltd., Birmingham, UK, UK, 2005.

[37] JUERJENS, J. *Secure Systems Development with UML*. SpringerVerlag, 2003.

[38] KELSEY, J., SCHNEIER, B., AND FERGUSON, N. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *Proceedings of the 6th Annual International Workshop on Selected Areas in Cryptography* (London, UK, 2000), SAC '99, Springer-Verlag, pp. 13–33.

[39] KOLMOGOROV, A. N. On tables of random numbers. *Sankhyā Ser. A 25* (1963), 369–375. Reprinted in [40].

[40] KOLMOGOROV, A. N. On tables of random numbers. *Theor. Comput. Sci. 207*, 2 (Nov. 1998), 387–395. Reprint of [39].

[41] LARMAN, C., AND BASILI, V. Iterative and incremental developments. A brief history. *Computer 36*, 6 (june 2003), 47 – 56.

[42] LEAVENS, G. T., AND CHEON, Y. Design by Contract with JML, 2006.

[43] MEAD, N. R., AND STEHNEY, T. Security Quality Requirements Engineering (SQUARE) Methodology. In *SESS '05: Proceedings of the 2005 workshop on Software engineering for secure systems—building trustworthy applications* (New York, NY, USA, 2005), ACM, pp. 1–7.

[44] MICROSOFT. CryptGenRandom function. `http://msdn.microsoft.com/en-us/library/windows/desktop/aa379942(v=vs.85).aspx`, accessed in Jul 2012.

[45] MÖLLER, N. Nettle: a low-level cryptographic library. `http://www.lysator.liu.se/~nisse/nettle/nettle.html`, accessed in Jul 2012.

[46] MULARIEN, P. *Spring Security 3*. Packt Publishing, 2010.

[47] NAIR, S. K., GERRITS, R., CRISPO, B., AND TANENBAUM, A. S. Turning teenagers into stores. *Computer 41* (2008), 58–62.

[48] NAIR, S. K., POPESCU, B. C., GAMAGE, C., CRISPO, B., AND TANENBAUM, A. S. Enabling drm-preserving digital content redistribution. In *Proceedings of the Seventh IEEE International Conference on E-Commerce Technology* (Washington, DC, USA, 2005), CEC '05, IEEE Computer Society, pp. 151–158.

[49] NETSCAPE COMMUNICATIONS CORP. SSL 0.2 protocol specification. `http://www.mozilla.org/projects/security/pki/nss/ssl/draft02.html`, Feb 1995.

[50] NETSCAPE COMMUNICATIONS CORP. SSL 3.0 protocol specification. `http://www.mozilla.org/projects/security/pki/nss/ssl/draft302.txt`, Nov 1996.

[51] NIELSON, F., NIELSON, H. R., AND HANKIN, C. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[52] NIST. Tentative timeline of the development of new hash functions. http://csrc.nist.gov/groups/ST/hash/timeline.html, accessed in Jul 2012.

[53] NIST COMPUTER SECURITY DIVISION. Federal information processing standards 140-2.

[54] NIST COMPUTER SECURITY DIVISION. Federal information processing standards 180-2.

[55] OFFSPARK. Polar SSL. http://polarssl.org/, accessed in Jul 2012.

[56] ORACLE. Enterprise JavaBeans Technology. http://www.oracle.com/technetwork/java/index-jsp-140203.html, accessed in Dec 2010.

[57] OWASP FOUNDATION. Build vulnerability remediation procedures. http://www.owasp.org/index.php/Category:BP5_Build_vulnerability_remediation_procedures, accessed in Feb 2010.

[58] OWASP FOUNDATION. Capture security requirements. http://www.owasp.org/index.php/Category:BP3_Capture_security_requirements, accessed in Feb 2010.

[59] OWASP FOUNDATION. Identify, implement, and perform security tests. http://www.owasp.org/index.php/Identify,_implement,_and_perform_security_tests, accessed in Feb 2010.

[60] OWASP FOUNDATION. Implement secure development practices. http://www.owasp.org/index.php/Category:BP4_Implement_secure_development_practices, accessed in Feb 2010.

[61] OWASP FOUNDATION. Institute security awareness program. http://www.owasp.org/index.php/Institute_security_awareness_program, accessed in Feb 2010.

[62] OWASP FOUNDATION. Open Web Application Security Project. http://www.owasp.org, accessed in Oct 2010.

[63] OWASP FOUNDATION. Perform security analysis of system requirements and design (threat modeling). http://www.owasp.org/index.php/Perform_security_analysis_of_system_requirements_and_design_(threat_modeling), accessed in Feb 2010.

[64] OWASP FOUNDATION. Perform source-level security review. http://www.owasp.org/index.php/Perform_source-level_security_review, accessed in Feb 2010.

[65] OWASP FOUNDATION. CLASP security principles. https://www.owasp.org/index.php/CLASP_Security_Principles, accessed in Jul 2012.

[66] PALMER, B., BUBENDORFER, K., AND WELCH, I. A protocol for anonymously establishing digital provenance in reseller chains (short paper). In *Financial Cryptography* (2011), G. Danezis, Ed., vol. 7035 of *Lecture Notes in Computer Science*, Springer, pp. 85–92.

[67] PARASOFT. Java static analysis, code review, unit testing, runtime error detection. http://www.parasoft.com/jsp/products/jtest.jsp/, accessed in Jul 2012.

[68] POINTCHEVAL, D., AND STERN, J. Security proofs for signature schemes. In *EUROCRYPT '96: Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology* (1996), Springer-Verlag, pp. 387–398.

[69] POTATO SYSTEM. Potato system - about us. `http://www.potatosystem.com/info/en/imprint`, accessed in Sept 2010.

[70] QUISQUATER, J.-J., GUILLOU, L., ANNICK, M., AND BERSON, T. How to explain zero-knowledge protocols to your children. In *Proceedings on Advances in cryptology* (New York, NY, USA, 1989), CRYPTO '89, Springer-Verlag New York, Inc., pp. 628–631.

[71] RESCORLA, E., RAY, M., DISPENSA, S., AND OSKOV, N. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), Feb. 2010.

[72] RIES, T., STATE, R. AND PANCHENKO, A. Comparison of low-latency anonymous communication systems — practical usage and performance. In *Australasian Information Security Conference (AISC 2011)* (Perth, Australia, 2011), C. Boyd and J. Pieprzyk, Eds., vol. 116 of *CRPIT*, ACS, pp. 77–86.

[73] ROYCE, W. W. Managing the Development of Large Software Systems: Concepts and Techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering* (Los Alamitos, CA, USA, 1987), IEEE Computer Society Press, pp. 328–338.

[74] SANTESSON, S. TLS Handshake Message for Supplemental Data. RFC 4680 (Proposed Standard), Oct. 2006.

[75] SANTESSON, S., MEDVINSKY, A., AND BALL, J. TLS User Mapping Extension. RFC 4681 (Proposed Standard), Oct. 2006.

[76] SCHNORR, C. P. Efficient identification and signatures for smart cards. In *EUROCRYPT '89: Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology* (New York, NY, USA, 1990), Springer-Verlag New York, Inc., pp. 688–689.

[77] SCHUBERT, A., AND CHRZĄSZCZ, J. ESC/Java2 as a Tool to Ensure Security in the Source Code of Java Applications. In *Software Engineering Techniques: Design for Quality*, K. Sacha, Ed., vol. 227 of *IFIP International Federation for Information Processing*. Springer Boston, 2007, pp. 337–348.

[78] SEZNEC, A., AND SENDRIER, N. HAVEGE: A user-level software heuristic for generating empirically strong random numbers. *ACM Trans. Model. Comput. Simul. 13*, 4 (Oct. 2003), 334–346.

[79] SILVERSTONE, D. randomsound — alsa sound card related entropy gathering daemon. `http://www.digital-scurf.org/software/randomsound`, accessed in Jul 2012.

[80] SOFTWARE ENGINEERING INSTITUTE, CARNEGIE MELLON UNIVERSITY. CERT. `http://www.cert.org/`, accessed in Feb 2011.

[81] SOFTWARE ENGINEERING INSTITUTE, CARNEGIE MELLON UNIVERSITY. Team software process. `http://www.sei.cmu.edu/tsp/`, accessed in Feb 2011.

[82] SYVERSON, P. F., GOLDSCHLAG, D. M., AND REED, M. G. Anonymous connections and onion routing. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 1997), SP '97, IEEE Computer Society, pp. 44–56.

[83] THE FREE SOFTWARE FOUNDATION. Libgcrypt. `http://www.gnu.org/software/libgcrypt/`, accessed in Jul 2012.

[84] THE FREE SOFTWARE FOUNDATION. The GNU Crypto project. `http://www.gnu.org/software/gnu-crypto/`, accessed in Jul 2012.

[85] THE FREE SOFTWARE FOUNDATION. The GNU Transport Layer Security Library. `http://www.gnu.org/software/gnutls/`, ac-

cessed in Jul 2012.

[86] THE LEGION OF THE BOUNCY CASTLE. bouncycastle.org. `http://www.bouncycastle.org`, accessed in Jul 2012.

[87] THE OPENSSL PROJECT. OpenSSL: The Open Source toolkit for SSL/TLS. `http://www.openssl.org/`, accessed in Jul 2012.

[88] TORVALDS, L. Linux Kernel Coding Style. `http://lxr.linux.no/source/Documentation/CodingStyle`, accessed in Oct 2010.

[89] WARNER, B. EGD: The Entropy Gathering Daemon. `http://egd.sourceforge.net/`, accessed in Jul 2012.

[90] WHITMAN, M. E., AND MATTORD, H. J. *Principles of Information Security*, 3rd ed. Course Technology Press, Boston, MA, United States, 2007.

[91] X-WAY RIGHTS BV, AND DEBLIER, B. Beecrypt. `http://beecrypt.sourceforge.net/`, accessed in Jul 2012.