Secure communications protocols and the protection of cryptographic keys

Nikos MAVROGIANNOPoulos
Secure communications protocols and the protection of cryptographic keys

Nikos MAVROGIANNOPOULOS

Jury:
Prof. dr. ir. D. Vandermeulen, chair
Prof. dr. ir. B. Preneel, supervisor
Prof. dr. ir. I. Verbauwhede
Prof. dr. ir. F. Piesens
Prof. dr. ir. B. Schoenmakers
(Technische Universiteit Eindhoven)
Prof. dr. C. J. Mitchell
(Royal Holloway, University of London)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

June 2013
Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotocopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the publisher.

D/2013/7515/69
Acknowledgements

Having the opportunity to work in the Computer Security and Industrial Cryptography (COSIC) group of Katholieke Universiteit Leuven has been a pleasure and an honour. It was a great pleasure and also personal challenge to work in the field of cryptography and collaborate with experts in the field, an opportunity given to me by my promotor Bart Preneel, to whom I will always be grateful.

Foremost, I would like to thank my parents for encouraging me to pursue this work, Andreas Pashalidis for being insistent on understanding every small detail of our work—something that eventually uncovered many limitations of my knowledge,—Nessim Kissri for always providing constructive and encouraging comments and for taking up the work when I had given up, Filipe Beato and Oscar Reparaz for the interesting discussions on an extreme variety of topics and proof-reading this thesis, Raoul Strackx for his constructive comments on this thesis and Anthony Van Herrewege for his fine Dutch translation of the abstract.

Undoubtedly, my work in COSIC would not have been the same without Pela Noë, who always was there to help with any issue that occurred.

Finally, I would like to thank the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen) for the SEC-SODA project, which funded the research on the topics of this thesis.
Abstract

The thesis deals with open problems in the area of two-party secure communications protocols and cryptographic key protection. A secure communications protocol ensures the authenticity of both parties and protects the confidentiality, integrity and freshness of the information that is transmitted. For this purpose, both parties require long term cryptographic keys that are used for mutual entity authentication and to establish session keys.

We evaluate the methods used for long-term key protection in software, and in combinations of software and hardware (e.g., smart cards). We further investigate whether these methods are sufficient to prevent key extraction when the protected long-term keys are combined with secure communications protocols. We provide evidence that the currently used methods are not sufficient to ensure the overall security of a system when the protected keys are used in conjunction with an arbitrary protocol, and demonstrate this with an attack on the smart card-based public key Kerberos (PKINIT) protocol. We further attempt to ensure the overall security of a system by enhancing Shoup’s simulatability-based static corruptions model to account for threats related to long-term key protection. Under this model we examine the Diffie-Hellman variant of the PKINIT protocol and show that a fix to the protocol, we proposed, is secure. Furthermore, we develop a software-based long-term key protection scheme for the Linux-kernel, that is used to protect long-term keys from application compromise. It relies on the isolation mechanisms available in the operating system, and is designed to avoid all pitfalls and known attacks from previous designs. Finally, we study a family of attacks on secure communications protocols, namely the cross-protocol or multi-protocol attacks. These attacks exploit interactions between different protocols that share the same keys. While the threat of such attacks was often highlighted, only few practical attacks exist for secure communications protocols. We survey the known attacks and countermeasures, and present a new attack on the widely deployed TLS (also known as SSL) protocol that relies on interactions between the Diffie-Hellman ciphersuites and their elliptic curve counterparts. Moreover, to mitigate this attack and related attacks we propose countermeasures for the TLS protocol.
Deze thesis handelt over veilige communicatieprotocollen met twee partijen en cryptografische sleutelbescherming. Een veilig communicatieprotocol heeft de volgende eigenschappen: beide partijen worden geauthentiseerd t.o.v. elkaar en de confidentialiteit, integriteit en versheid van de overgezonden informatie worden beschermd. Beide partijen maken hiervoor gebruik van een lange termijn cryptografische sleutel; op basis van deze sleutels worden de partijen geauthentiseerd en worden sessiesleutels gegenereerd.

We evalueren de methodes die gebruikt worden voor langetermijnbescherming van sleutels in software en in een combinatie van hard- en software (bv. een chipkaart). Verder onderzoeken we of deze methodes het uitlekken van sleutels kunnen voorkomen als de langetermijnsleutels gecombineerd worden met veilige communicatieprotocollen. We tonen aan dat de huidige methodes niet volstaan om de volledige veiligheid van een systeem te garanderen als sleutels gebruikt worden samen met een willeurig protocol. We illustreren dit met een aanval op het publieke sleutel Kerberos (PKINIT) protocol dat gebaseerd is op chipkaarten. Verder trachten we de volledige veiligheid van een systeem te garanderen door Shoups simuleerbaarheidsgebaseerde statische corruptiemodellen te verbeteren zodat ze rekening houden met gevaren gerelateerd aan langetermijnsleutelbescherming. Op basis van deze modellen onderzoeken we de Diffie-Hellman variant van het PKINIT protocol en tonen aan dat een verbetering, door ons gesuggereerd, veilig is. Verder ontwikkelen we een softwarematig beschermingsalgoritme voor langetermijnsleutels voor de Linux kernel, dat gebruikt wordt om langetermijnsleutels te beschermen tegen aanvallen vanuit de software applicaties. Het algoritme maakt gebruik van de isolatiemechanismes die beschikbaar zijn in het besturingssysteem en is ontwikkeld om alle aanvallen tegen vorige versies te weerstaan. Ten slotte bestuderen we een familie van aanvallen op veilige communicatieprotocollen, namelijk cross-protocol- en multi-protocolaanvallen. Deze aanvallen buiten interacties tussen verschillende protocollen uit die gebruik maken van dezelfde sleutels. Hoewel de aandacht reeds vaak op het potentiële gevaar van zulke aanvallen gevestigd werd, bestaan er slechts enkele praktische aanvallen. We onderzoeken de bestaande aanvallen en tegenmaatregelen en presenteren een
nieuwe aanval op het wijdverspreide TLS (ook gekend als SSL) protocol. Deze aanval maakt gebruik van interacties tussen het Diffie-Hellman protocol over de groep $\mathbb{Z}_p^*$ en over de groep van elliptische krommen over een eindig veld. Ook stellen we maatregelen voor om deze en gerelateerde aanvallen tegen te gaan.
## Contents

Abstract iii  
Contents vii  
List of Figures ix  
List of Tables xi  
List of Abbreviations xiii

### I General Overview

1 Introduction 2  
1.1 Motivation and scope 2  
1.2 Cryptography and secure communications 3  
1.3 Outline 5

2 Preliminaries 7  
2.1 Cryptography 7  
2.2 Secure communications protocols 11  
2.3 Long-term key protection 13  
2.3.1 Obfuscation and white-box cryptography 14  
2.3.2 Subsystem isolation 18  
2.4 Cross-protocol attacks 27  
2.4.1 Existing cross-protocol attacks 28  
2.4.2 Preventing cross-protocol attacks 35

3 Contributions 39  
3.1 Cryptographic key protection 39  
3.1.1 Obfuscation and white-box cryptography 39  
3.1.2 Subsystem isolation 40  
3.2 Cross-protocol attacks 43
4 Conclusions and open issues

4.1 Cryptographic key protection ............................................. 45
  4.1.1 Obfuscation and white-box cryptography ......................... 45
  4.1.2 Subsystem isolation .................................................. 46

4.2 Cross-protocol attacks .................................................. 47

Bibliography

II Publications

5 List of publications

  5.1 A taxonomy of self-modifying code for obfuscation ............... 63
  5.2 A Linux kernel cryptographic framework .............................. 97
  5.3 Security implications in Kerberos by the introduction of smart cards ......................................................... 121
  5.4 Towards a secure Kerberos key exchange with smart cards . . 135
  5.5 A cross-protocol attack on the TLS protocol ....................... 159

A A Linux kernel cryptographic framework (extended version) ...... 185

Curriculum vitae

  211
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>An analogous of symmetric cryptography.</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>An analogous of the asymmetric (public key) cryptography idea.</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>The various layers in the Internet protocols suite.</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>The basic operations of a security protocol.</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>The different layers available in a software system.</td>
<td>20</td>
</tr>
<tr>
<td>2.6</td>
<td>The Wide Mouth Frog protocol.</td>
<td>23</td>
</tr>
<tr>
<td>2.7</td>
<td>The first smart card protocol in [20].</td>
<td>25</td>
</tr>
<tr>
<td>2.8</td>
<td>A simplified run of the Kerberos protocol.</td>
<td>29</td>
</tr>
<tr>
<td>2.9</td>
<td>Ticket data prior to encryption for Kerberos V4 and Kerberos V5.</td>
<td>29</td>
</tr>
<tr>
<td>2.10</td>
<td>The format of a Kerberos V4 ticket.</td>
<td>31</td>
</tr>
<tr>
<td>2.11</td>
<td>The authentication part of the GSM protocol.</td>
<td>32</td>
</tr>
<tr>
<td>2.12</td>
<td>The contents of the ServerKeyExchange message.</td>
<td>33</td>
</tr>
<tr>
<td>2.13</td>
<td>The Wagner and Schneier attack on SSL 3.0.</td>
<td>35</td>
</tr>
</tbody>
</table>
List of Tables

2.1 A summary of the ‘Extended Key Usage’ field of all web server certificates in the 2010 EFF SSL Observatory data. . . . . . . . . . 37
2.2 A summary of the ‘Key Usage’ field of RSA web server certificates in the 2010 EFF SSL Observatory data. . . . . . . . . . 37
### List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3/8</td>
<td>The authentication and key agreement algorithms of GSM.</td>
</tr>
<tr>
<td>A5</td>
<td>The encryption algorithm used in GSM.</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface.</td>
</tr>
<tr>
<td>BAN</td>
<td>Burrows Abadi and Needham. A logic of authentication.</td>
</tr>
<tr>
<td>CBC</td>
<td>Cipher Block Chaining mode. An encryption algorithm mode of operation.</td>
</tr>
<tr>
<td>CNG</td>
<td>Cryptography API Next Generation. A cryptographic API used in Microsoft Windows.</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit.</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard. An encryption algorithm established by the U.S. National Bureau of Standards (NBS) in 1976, based on a cipher designed in IBM.</td>
</tr>
<tr>
<td>DH</td>
<td>Diffie-Hellman algorithm. A key establishment method.</td>
</tr>
<tr>
<td>DHE</td>
<td>Diffie-Hellman Ephemeral. A term of the TLS protocol to describe a DH key establishment to generate an ephemeral key.</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name System.</td>
</tr>
<tr>
<td>DNSSEC</td>
<td>Domain Name System Security Extensions. A set of specifications by IETF for securing certain kinds of information provided by the DNS.</td>
</tr>
</tbody>
</table>
DOS  The Disk Operating System. An operating system popular in the 80’s.

DSA  Digital Signature Algorithm. One of the algorithms used in DSS.

DSS  Digital Signature Standard. A digital signature standard established by the U.S. National Institute of Standards and Technology (NIST).

ECB  Electronic CodeBook mode. The simplest encryption mode in a block cipher.


ECDSA  Elliptic Curve Digital Signature Algorithm. One of the algorithms used in DSS.

EFF  The Electronic Frontier Foundation is an international non-profit digital rights group.

EMV  Originally stood for Europay, MasterCard and Visa. A global standard for inter-operation of smart cards of the banking sector.

ETSI  European Telecommunications Standards Institute.


GSM  Global System for Mobile communications. A set of standards developed by the ETSI for digital cellular protocols.

HSM  Hardware Security Module.

HTML  HyperText Markup Language.

HTTP  HyperText Transfer Protocol.

IETF  Internet Engineering Task Force.

IKE  Internet Key Exchange. IPsec’s key entity authentication and key establishment sub-protocol.

IMAP  Internet Message Access Protocol.

IND-CCA  Ciphertext indistinguishability under chosen ciphertext attack. A property of encryption schemes.
IND-CCA2 Ciphertext indistinguishability under adaptive chosen ciphertext attack. A property of encryption schemes.

IPC Inter-Process Communication.

IPsec IP Security protocol. A secure communications protocol used on the IP Internet layer.

IV Initialization Vector.

KDC Key Distribution Center.


MLS Multi-Level Security. Used to describe systems that provide access control mechanisms with multiple levels of user privileges.

OS Operating System.

PCI Peripheral Component Interconnect. A computer bus for attaching hardware devices.

PIN Personal Identification Number. The term is often used to describe a short password.

PKI Public Key Infrastructure. The set of procedures and policies required to securely distribute and manage public key certificates.

PKINIT Public Key Cryptography for Initial Authentication. An extension of the Kerberos protocol.

PKIX The Public Key Infrastructure working group of IETF.

POP3 Post Office Protocol 3.

POSIX Portable Operating System Interface.


SIM Subscriber Identity Module. A smart card used in mobile phones to authenticate to the provider.

SMTP Simple Mail Transfer Protocol.
SSH  Secure SHell. A secure communications protocol widely used for remote access.

SSL  Secure Sockets Layer. An older name of TLS.

SUS  Single UNIX Specification.

TLS  Transport Layer Security. A secure communications protocol widely used in the world wide web.

TPM  Trusted Platform Module. A secure co-processor in a computer that can store cryptographic keys. It is specified by the Trusted Computing Group.

ZK   Zero Knowledge.
Part I

General Overview
1 Introduction

1.1 Motivation and scope

While often a privilege of military organizations or secret societies in the past, secure communications protocols are increasingly used in several aspects of our daily lives. Online activities such as shopping, remote system access, and voice communications are some of their main applications.

A secure communications protocol describes the steps that need to be followed in order to ensure certain properties of a communications channel, such as entity authentication, message integrity and confidentiality. The protocol typically makes use of cryptographic algorithms, such as key establishment, authentication and encryption algorithms combined in certain way to ensure the previous properties. Briefly, a two-party secure communications protocol utilizes any available credentials\(^1\) of the participants to authenticate each other, and establish session keys. The session keys, valid only for a single communication session, are used to encrypt and protect the integrity of any subsequent exchange of messages. The participant’s credentials are called long-term keys, and are typically valid for multiple sessions.

\(^1\)We use the term credentials to indicate any type of identity proof. In cryptographic algorithms credentials are typically secret, private, or public keys.
The field of secure communications protocols is a diverse field consisting of multiple specialized topics, e.g., encryption algorithms, key establishment methods, etc. In this thesis we focus on two of them. The first topic is long-term key protection. We evaluate the methods used for long-term key protection in software, and in combinations of software and hardware. We further investigate whether these methods are sufficient to prevent key extraction when combined with an arbitrary protocol. The second topic is attacks on secure communications protocols. We discuss a family of attacks on secure communications protocols, namely the cross-protocol attacks, and techniques to mitigate them.

1.2 Cryptography and secure communications

The need for secure communications is probably as old as human history. Written reports of simple substitution ciphers used for secret communications date back to 4th century BCE [49, 84], with the military manual “Poliorketika” (translation: “On siegcraft”) by Aeneas, possibly a Greek general. In that manual, various techniques and devices to conceal a message to be delivered by a courier are described. Another device realized in classical antiquity is the ‘Scytale’; possibly used by Spartan field officers to authenticate messages originating from the homeland [84]. However, message concealment or secret writing was not restricted to military activities. The art of secret writing is one of the arts recommended for women, in an Indian book focused on human sexual behavior [102].

During and after the era of the Roman empire simple forms of cryptography were still used to secure communications, e.g., Caesar’s cipher known to us by Suetonius [96]. The invention of cryptanalysis as an analysis of the frequencies of letters is attributed to the Arab mathematician Al-Kindi [93] in the 9th century CE, but more advanced systems that are resistant to letter frequency analysis only appear at the dawn of middle ages. In 1518, a German abbot, Johannes Trithemius published his work “Polygraphiae” [98] in which he described the first known polyalphabetic cipher. Giovan Battista Bellaso, an Italian lawyer, published in [11] a polyalphabetic cipher\(^2\) that made use of an additional secret, introducing the notion of the key. The design of ciphers, however, evolved from an art of the skilled to a science, possibly in the 19th century after the Dutch linguist Auguste Kerckhoffs set the following principles (translation from French by Fabien Petitcolas) for a cryptographic system [58].

1. The system must be substantially, if not mathematically, undecipherable;

\(^2\)This cipher is known today as the Vigenère cipher.
2. The system must not require secrecy and can be stolen by the enemy without causing trouble;

3. It must be easy to communicate and retain the key without the aid of written notes, it must also be easy to change or modify the key at the discretion of the correspondents;

4. The system ought to be compatible with telegraph communication;

5. The system must be portable, and its use must not require more than one person;

6. Finally, given the circumstances in which such system is applied, it must be easy to use and must neither stress the mind nor require the knowledge of a long series of rules.

The second principle and the fact that the cipher is separated from the secret, set the foundations of modern cryptography.

Until the first half of the 20th century polyalphabetic substitution ciphers similar to the Battista cipher were still in use over the world, however, they were occasionally broken [55]. An advanced mechanized variant of a polyalphabetic substitution cipher, known as the Enigma cipher machine, was invented by the German electrical engineer Arthur Scherbius shortly after World War I. It was used extensively by the Axis powers to secure diplomatic and military communications before and during World War II, but despite its complexity it was also successfully cryptanalyzed [93]. These cryptanalytic advances, in combination with the invention of computers impacted modern cipher design.

The father of information theory Claude Shannon, in 1949, examined the mathematical structure of secrecy systems, and described [88] properties of what is today perceived as a modern cipher. He further showed that perfect secrecy is possible at the cost of key material as long as the size of the plaintext, i.e., the one time pad [12]. Since then, several cryptographic primitives were developed for the purposes of secure communications. Some prominent examples are the Data Encryption Standard (DES), the invention of asymmetric (public key) cryptography with the Diffie-Hellman key establishment method [36], and others.

However, the requirements for secure communications extend past the notion of secrecy, which is offered by ciphers. They were explicitly set in the Internet era, in documents describing protocols for secure online transactions [24, 103, 32]. They can be summarized as follows.\(^3\)

\(^3\)Note that on earlier work [103] there are additional requirements set for a protocol such as traffic analysis prevention and denial of service detection. The former is now considered only in specialized applications and the latter is typically an implicit requirement.
• Confidentiality; the exchanged messages must not be visible to an eavesdropper.

• Message integrity; any modification to the exchanged messages should be detected.

• Entity authentication; the communicating parties should be assured of each other’s identity.

These requirements form the implicit or explicit expectations from a secure communications protocol. They are fulfilled by modern protocols by combining the available cryptographic primitives, such as ciphers, in certain ways.

Today several secure communications protocols exist, many of which are international standards available for public use. Some examples include the Transport Layer Security (TLS), more commonly known as the SSL protocol [42, 34], which dominates the world wide web secure communications, the Secure Shell (SSH) protocol [110], used for remote system access, and the IPsec protocol [56].

1.3 Outline

The rest of this thesis is organized in two parts. The first part provides background information on existing work on the topics of cryptographic key protection and the cross-protocol family of attacks in secure communications protocols. Then, it discusses our contributions and provides conclusions and possibilities for future work. The second part contains our detailed publications related to the topics of this thesis.
“It is only one who is thoroughly acquainted with the evils of war that can thoroughly understand the profitable way of carrying it on.”

Sun Tzu – The art of war

2

Preliminaries

2.1 Cryptography

Cryptography is the scientific field that provides not only the building blocks required for a secure communications protocol, but also the theory behind a secure communications protocol design and evaluation.

While the term of ‘theory of secure communications protocols’ may not be authoritative we believe it incorporates the efforts of certain related fields of computing science and formal methods on cryptography. It includes the work on several aspects of communications protocols, for example, two party protocols that are the focus of this thesis, multi-party protocols, protocols that perform secure multi-party computations [109], etc. It is a relatively new field that is also advancing quickly while trying to match the pace secure communications protocols evolve; it also provides the set of tools and methods to describe and analyze modern protocols. The field first developed by creating precise definitions for the threats and adversaries in a network [37], and later with the creation of generic frameworks that model algorithms and their composition into protocols [21, 47]. These frameworks allow the generation of proofs on protocol components by using simulation, where each component must simulate an ideal abstraction. Once all components of a protocol can be simulated the overall security is ensured by a composition theorem. That way,
the desired security goals of a secure communications protocol can be verified against the set of threats handled by the framework.

At the same time, the field of cryptography provides algorithms and methods that offer, for example, encryption, integrity protection, or entity authentication. The cryptographic primitives are typically categorized as symmetric or pre-shared key, and asymmetric or public key. In the symmetric key algorithms all participants possess the same key and use it to encrypt, decrypt or authenticate data. An analogous system in the non-digital world can be seen in Fig. 2.1.

Asymmetric or public key algorithms consisting of a pair of keys, a public and a private key. The public key, as its name suggests, may be publicly available; in an encryption algorithm it allows anyone to encrypt a message to the holder of the corresponding private key. The private key is then used by the holder for decryption. The authentication operation is similar; a digital signature is generated by the holder of the private key and anyone can verify the signature using the public key. An analogous of asymmetric signatures in the physical world is shown in Fig. 2.2.

Asymmetric algorithms are a modern invention. They were first realized in 1976 by Diffie and Hellman in the now called Diffie-Hellman (DH) key establishment.
method [36], which allowed secure key establishment between two parties, in the presence of a passive eavesdropper, even when no prior key establishment has taken place. Later, other methods followed, e.g., the now widespread RSA method [82], which is used mainly for digital signatures, something that allowed the migration of traditional concepts such as signatures and notaries to the digital world. That is, a trusted third party can sign documents, proofs of identity (commonly called certificates), and anyone may verify their authenticity. Such a setup, i.e., the set of procedures and policies required to securely distribute and manage certificates, is typically called the Public Key Infrastructure (PKI).

Today asymmetric cryptography is widely deployed and used for entity authentication, digital signatures and session key establishment. The various existing infrastructures are application specific, e.g., the DNSSEC key hierarchy, or the EMV banking protocol’s hierarchy; in some protocols the hierarchy may consist of commercial entities that serve this purpose for a fee, or local authorities, e.g., within the scope of a corporation or department. In contrast, symmetric key cryptography is commonly used for the purposes of confidentiality and message integrity.

![Image of asymmetric cryptography concept]

**Figure 2.2:** An analogous of the asymmetric (public key) cryptography idea. The private key is in the form of a seal handle which is used to produce sealed documents, while the public key, is the form of the seal, which is public knowledge. (The seal picture is from the Dumfries & Galloway Museums collection).

In addition to division in symmetric and asymmetric, cryptographic primitives
can also be categorized based on their purpose. While there are several categories and sub-categories, we restrict our following brief description to the most relevant for the purposes of secure communications.

**Authenticated key establishment.** Key establishment algorithms enable two peers to agree on a key that is only known to them, over a public channel. DH [36], or RSA-encryption are few of the methods deployed in modern protocols. The known methods are based on certain mathematical problems that are asymmetric in nature. That is, a function $f$ that is easy to calculate, but the effort required to invert it is significantly larger. For example, the security of the DH system relies on the asymmetry in the performance of modular exponentiation on the multiplicative group $\mathbb{Z}_p^*$ (which is relatively easy) and the calculation of discrete logarithms (which is very difficult in $\mathbb{Z}_p^*$ for a large prime $p$).

A significant factor in key establishment is entity authentication. An unauthenticated key establishment method like DH ensures that a key has been established, but the identity of the party that this key has been established with, is unknown. For that, modern protocols utilize digital signatures, or encryption based methods, using a known long-term key or a trusted third party to authenticate the messages of the peer in a key establishment [14, 34].

**Digital signatures.** Digital signatures were one of the first applications of asymmetric cryptography. The most widely deployed algorithm today is RSA [82], but ElGamal-based [43] signature schemes such as DSS [78] or GOST R 34.10-2001 [45] have also seen considerable deployment.

Modern digital signature algorithms operate on data to be signed, by extracting a fingerprint of the data (typically by utilizing hash algorithms), and applying the private key based signature on the fingerprint. The verifier of the signature performs the same process to obtain the data fingerprint and then verifies it using the public key.

**Data encryption.** Encryption is the process of encoding messages to prevent eavesdropping; decryption is the inverse process. They are the oldest cryptographic primitives and hence the best known in the field of cryptography. Encryption algorithms are also called ciphers, and map plaintext (the data to be encrypted), to ciphertext (the encrypted data). Early encryption algorithms based their security on the secrecy of the algorithms. Modern ciphers such as DES, or AES [30], following Kerckhoffs’ principles, are public, and their security depends on the secrecy of a key.
Data integrity. While encryption algorithms provide confidentiality, they do not provide integrity of the transmitted data. Depending on the type of cipher in use, modifications of the ciphertext may result in predictable modifications on the plaintext. For example in stream ciphers that utilize the exclusive-or operation to combine ciphertext with the keystream, a bit-flip on the ciphertext results in a bit-flip on the plaintext. In secure communication protocols this is countered by using integrity mechanisms that utilize keys, such as Message Authentication Code (MAC) algorithms, e.g., HMAC [61], or encryption modes of operation that incorporate message authentication, e.g., AES-GCM [72].

2.2 Secure communications protocols

There are many security protocols that utilize cryptographic primitives and target distinct applications. The focus of this thesis is secure two-party communications (or sessions) over the Internet. A secure communications protocol ensures the authenticity of both parties and protects the confidentiality, integrity and freshness of the information that is transmitted. For this purpose, both parties require long term cryptographic keys that are used for mutual entity authentication and to establish session keys. Note that we will use the term ‘Internet secure communications protocols’ to describe any protocols published by the Internet Engineering Task Force (IETF) with the intention of providing data confidentiality, integrity and entity authentication.

The Internet secure communications protocols, such as TLS [34], IPsec [56] or Kerberos [77], provide a secure channel between a client and server. They perform entity authentication and key establishment, where the server authenticates the client and vice-versa, and establish fresh session keys. They can be classified as protocols that use pre-shared keys, where each party holds a copy of a common secret key, and protocols that operate using public keys, possibly over a public key infrastructure (PKI). The public key infrastructure manages certificates that typically contain a (locally unique) identifier, a public key and possibly few additional data; the certificates are digitally signed by a trusted third party called the Certification Authority (CA). The public key variants of the protocols are being the norm today.

These protocols are part of our everyday lives, and while mostly unnoticed, they are used in e-commerce and corporate environments. Each targets a different layer of the protocols stack (see Fig. 2.3), e.g., TLS and Kerberos are positioned over the transport layer, while IPsec is at the Internet layer, which indicates the types of networking applications they cover. The layers

---

1 The client and server roles may be arbitrary. Typically the client is the initiator of the session.
of the Internet protocols suite consist from (i) the link layer that contains the communication technologies used in a local network, (ii) the internet layer (IP) which establishes inter-networking of local networks, (iii) the transport layer which handles the reliable communication, and (iv) the application layer which contains the applications that run over the data communication channel.

![Diagram of the various layers in the Internet protocols suite.](image)

**Figure 2.3: The various layers in the Internet protocols suite.**

The major body of these communications protocols was designed based on best practices in protocol design [3] at their time. However, as time passed, new attacks were devised, new cryptographic algorithms were invented, and more tools were made available, such as formal protocol analysis [9, 91, 23] that led to the revision of several protocol design decisions. Almost all known protocols have undergone a phase of major rewrite. Examples can be seen in the IPsec transition from IKEv1 to IKEv2, the Kerberos update from V4 to V5, the SSL 2.0 protocol transition to SSL 3.0 (and now TLS 1.2) etc.

A secure communications protocol’s operation can be summarized in Fig. 2.4. Initially a combined authentication and key establishment step is performed. In this step the long-term keys, either pre-shared, or public keys, part of a public key infrastructure, are used for authentication of each of the two parties. The output of this step is a number of temporal keys, called session keys, that are used during the confidentiality and integrity step. The latter step is where the actual secure communication takes place, and data messages from both parties are exchanged securely. Note that several protocols may include more options in session establishment, e.g, by including a session resumption operation to optimize sessions established after an initial one, as in TLS, or by allowing the generation of child sessions, as in IKEv2.
2.3 Long-term key protection

When secure communications protocols are used in real world systems, various methods are employed by applications to protect the cryptographic keys involved. There are applications, that read the (possibly encrypted) keys from the storage subsystem, copy them in memory and use them for cryptographic operations throughout the program execution. Other applications protect their secrets via obfuscation, and others utilize external hardware for the cryptographic operations, e.g., smart cards or Hardware Security Modules (HSMs). This indicates that not all applications are designed to protect against the same threats. A web server developer would consider as a major threat an adversary that accesses the system through a software vulnerability, while an obfuscated application developer may consider the major threat to be the user of the software, who potentially extracts the keys from the software and accesses a proprietary network using other software (e.g., the Skype telecommunications application).

In this section we discuss existing methods of cryptographic key protection in software systems. We include in this context various protection techniques used in software systems, including white-box cryptography and obfuscation, as well as asset isolation.

The systems we consider in this study are single level security software systems, which describe most widely deployed computer systems, i.e., the Linux kernel based systems, Mac OS X, Windows etc. We use the term single level security system to distinguish from Multi-Level Security (MLS) systems. In that model, software is comprised of an operating system running in a different protection ring\(^2\) from normal applications. This requirement ensures that normal applications cannot access data available to the operating system. In

\(^2\)We use the term protection ring to assume a strict separation of data and code between different access control levels. This terminology was described during the design of Multics
addition, processes may also be isolated from each other and access controls may restrict the data each process accesses.

We classify the available protection methods for cryptographic keys against an adversary as follows.

- **No protection**, the keys are stored unprotected in the application memory; note that we do not distinguish between encrypted or plain keys in the storage subsystem since in both cases the keys are exposed in the application’s memory, making any distinction applicable to a very limited number of threats;
- **White-box cryptography/obfuscation**, the cryptographic keys are intermixed with the application;
- **Isolation**, the keys are not present in the application memory; the application is using an external process or hardware to perform cryptographic operations.

Depending on the adversary considered, these classes offer different levels of protection. The protection offered by the first class is minimal and thus not considered in the subsequent analysis. The next sections elaborate on the protection levels and the adversaries considered in the latter two classes.

### 2.3.1 Obfuscation and white-box cryptography

Obfuscation, used for the purpose of protecting secrets in software systems has a long history. Early DOS programs used it to hide their copy protection functionality, and shareware programs obfuscate their internal structures to prevent modification and tampering [46]. White-box cryptography, on the other hand, is a new field introduced with Chow et al.’s DES white-box cipher [26]. It is a specialized obfuscation method for cryptographic key material, which is intended to resist key extraction and also ensures certain cryptographic properties under a formal definition [106].

#### The adversarial model

The threat addressed by the current obfuscation techniques is an adversary who has the software under his control, and all the required permissions for executing it. That is, an adversary with debugging, disassembling and emulation tools available who seeks to reverse-engineer a program in order to
recover some assets (e.g., the protected keys). The goal of the obfuscation and white-box techniques is to prevent the recovery of these assets.

An important note about these techniques is that the adversary is able to execute the program. That is, even if obfuscation and white-box cryptography could prevent the adversary from extracting the keys from the software in which they are embedded, they fall short to an adversary who is content with simply accessing the obfuscated cipher. While the keys may remain protected, access to the functionality of the (obfuscated) software is sufficient to use the available cipher. For that reason, obfuscation techniques may be more useful in asymmetric applications where the software user is provided with only one of the cryptographic primitives, i.e., either encryption or decryption but not both. However, whether asymmetric applications of white-box cryptography are possible is an open question [106].

**Generic obfuscation**

The notion of software obfuscation was formalized by Barak et al. [5]. The authors’ formalization depends on an obfuscator satisfying the “virtual black box” property, which guarantees that an adversary with access to an obfuscated program should not learn anything more than is possible via oracle access to the original, unobfuscated program. Note that we use the term oracle to denote a system which can be queried to provide information about something that otherwise would be unknown. In this context the oracle provides the output of the original program. In other words, access to the program code should not provide more information than can be gained by observing the program’s outputs. Under this definition, generic obfuscation is proven to be impossible [5].

More formally, obfuscation is the mapping a program $P$ into another, semantically equivalent program $O(P)$. Barak’s formalization utilizes probabilistic Turing Machines, which may be seen as an abstraction of real programs.

**Definition 1.** An obfuscator $O$ is a probabilistic algorithm that accepts a Turing Machine $P$ as input and outputs the Turing Machine $O(P)$, which satisfies the following properties.

- (functionality) For every Turing Machine $P$, $O(P)$ is a Turing Machine that computes the same function as $P$.
- (polynomial slowdown) The length and the running time of $O(P)$ are at most polynomially larger than the corresponding values of $P$.

---

3In fact this was one of the initial approaches to public key cryptography [35].
4That is, Turing Machines with an additional tape that contains random data.
• *(virtual black box property)* Anything that can be computed in polynomial time from \(O(P)\) can also be computed in polynomial time given oracle access to \(P\).

Note that we say that \(x\) polynomially larger than \(y\) if \(x = O(y^d)\), for some \(d \in \mathbb{N}^+\). This definition, although it captures the expected functionality of obfuscation of an arbitrary program, is proven to be of limited applicability, as it is impossible to satisfy [5].

**Domain-specific obfuscation**

In order to overcome the impossibility result Lynn et al. weaken the notion of obfuscation in order to construct obfuscators for a class of programs for access-control [63]. Their notion of obfuscation, named obfuscation in the random oracle model, removes the requirement of semantic equivalence of the obfuscated program and replaces it with a requirement for approximate equivalence. That is, the obfuscator definition is replaced with the following.

**Definition 2.** An obfuscator \(O\) is a probabilistic algorithm that accepts a Turing Machine \(P\) as input and outputs the Turing Machine \(O(P)\), which in addition to the polynomial slowdown and the virtual black box property satisfies the following.

• *(approximate functionality)* For every Turing Machine \(P\), \(O(P)\) is a Turing Machine that computes the same function as \(P\) except with negligible probability.

Under this definition positive results of obfuscation in the domain of access control are shown [63].

**White-box cryptography**

As we saw, the impossibility result on a general-purpose obfuscator, i.e., an obfuscator that can protect any arbitrary program, cannot be extended to domain-specific constructions, e.g., obfuscators for access control [63]. In the white-box cryptography field [106], the open question is whether there can be obfuscation of a cipher and a key under Barak’s definition.

Saxena et al. in [86] formalize the *white-box property* as the ability of an obfuscated cipher, when running under the control of the adversary, to provide the same security guarantees (security notions) as a cipher that is available through a black-box setting, i.e., the adversary has oracle access to it.

More specifically, Saxena’s white box property is defined using a family of probabilistic Turing machines, which model the white-box encryption algorithm
in combination with a security notion, e.g., ciphertext indistinguishability (IND-CCA). A security notion describes a “game” the adversary plays that involves a cryptosystem (e.g., an encryption algorithm); if the adversary cannot get an advantage in this game, the cryptosystem is said to satisfy the security notion. Without getting into details (see [106, 86] for a detailed description), the white-box property is defined as follows.

**Definition 3.** An obfuscated cipher $O$, satisfies the white-box property on a particular security notion $sn$, and a cipher $P$, if

$$|\max(Adv_{WB}^{sn}_{A,O(P)}(k)) - \max(Adv_{BB}^{sn}_{A}(k))| \leq n(k),$$

where $n$ is a negligible function and $k$ a security parameter.

The $Adv_{BB}$ function outputs the probability that the adversary will win the game described by the security notion $sn$ in a black-box setting, i.e., without access to the obfuscated program. Correspondingly, $Adv_{WB}$ outputs the probability for the adversary to win the same game described by the notion $sn$ with additional access to $O(P)$.

In this model, Saxena et al. show that certain properties cannot be provided by a white-box cipher, e.g., indistinguishability under adaptive chosen ciphertext attack (IND-CCA2).

**Applicability to real world software**

Despite the negative results, obfuscation and white-box cryptography are used in the software industry to protect secrets. They are usually used to raise the adversary’s cost of reverse-engineering by the cost required for deobfuscation. Wyseur notes in [107]:

“With respect to security: as far as we know, no white-box implementation in a real world product has suffered from a key recovery attack, despite the cryptanalysis results that have been published. This shows that there is a clear gap between theory and practice . . .”

This uncovers an inconsistency between the formal evaluation of obfuscation and white-box techniques with real world practice. Even though the existing techniques are shown to have weaknesses in the existing models, and attacks are discovered [76, 75, 108], these attacks do not materialize into (publicly known) attacks on real world applications. That indicates that the existing formal models fall short into measuring the factors that are accounted in real world techniques.
2.3.2 Subsystem isolation

Isolation as a protection method for certain assets is a well-known technique in various fields. The most prominent example seen throughout human history is the quarantine, the practice of isolating suspected carriers of epidemic diseases from the rest of the population, in order to prevent a pandemic. In computer science, isolation is used as a security mechanism, e.g., to prevent privilege escalation \[81\], to prevent spread of computer viruses, or to prevent the extraction of secrets by storing them in HSMs \[83\]. In this section we summarize the existing isolation techniques used for cryptographic key protection and discuss open issues.

Hardware-based isolation

One way to isolate cryptographic keys from the software that uses them is through the usage of HSMs, smart cards \[90\] and other hardware tokens \[83\]. Their purpose is to provide a protected platform, either from the users themselves or from third parties that somehow obtain the device, to perform cryptographic operations without exposing the keys to a possibly untrustworthy environment (e.g., a terminal). Example devices are public key smart cards, the well-known SIM cards, the Trusted Platform Module \[100, 99\], PCI-cards etc.

Depending on the threats a device defends against, it may be susceptible to a variety of attacks that extract the cryptographic keys, including physical attacks, i.e., attacks that reverse engineer the hardware module and extract the stored keys \[2\], cloning \[18\], power analysis and more. Hence, the devices are typically classified based on the physical protection offered to the stored keys, on several levels of protection \[79, 52\]. For the purposes of this thesis, we assume a budget threshold for the above physical attacks to be impractical for the target adversaries. Our main concern is protection against extraction of the cryptographic keys using the available operations on the module in isolation.

Note, that historically the term smart card \[90\] has been used to describe various tokens that vary from a simple data holder to tokens with keypads and screens, tokens which provide the whole application stack (e.g., the EMV \[38\] cards), etc. In this thesis, when referring to a hardware security module or smart card we only assume the ability of the device to perform certain cryptographic operations on data provided by the user. These operations, e.g., encryption or signature generation, will utilize the stored keys without exposing them. Other types of smart cards or modules are beyond the scope of this thesis.
Software-based isolation

Because the deployment of application-specific hardware security modules is sometimes prohibitive, either financially or due to other constraints, several software approaches to isolation exist. They are based on isolation mechanisms provided by the software systems. In the operating systems we consider (see Sect. 2.3), the available isolation mechanisms can be divided into the following categories (see also Fig. 2.5).

- **Process-level**: Each process running on a system is isolated from other processes and its resources are protected via access control mechanisms enforced by the operating system.

- **Operating system-level**: The operating system is isolated from the rest of the system by running in a different CPU protection ring than its processes. Its resources are protected by a combination of CPU and operating system mechanisms.

- **Hypervisor-level**: The operating system is run on top of a hypervisor that is isolated from it by running on the highest CPU protection ring.

Examples of process-level isolation for the purposes of cryptographic keys protection are the privilege separation method used in OpenSSH [81], the CNG API implementation [74] in the Windows OS, the Lite Security Module [73] (LSM), etc. The key idea of process isolation is to use an isolated process to hold the cryptographic keys and provide operations on them as service to other processes using inter-process communication (IPC). The previously mentioned frameworks utilize the operating system’s access control mechanisms to protect access to the stored keys, e.g., by running the isolated process under a separate dedicated user. The cryptographic key isolation imposed by the software frameworks is indistinguishable to the users of the frameworks from a hardware security module, and in fact the CNG and LSM implementations provide their operation under the same API used for HSMs.

Both the process-based and OS isolation approaches rely directly or indirectly on the operating system’s access control mechanisms, and thus require a level of trust in the operating system. When the operating system cannot be trusted, e.g., due to its complexity, sensitive assets such as cryptographic keys could be moved outside its reach, to a hypervisor similarly to the system in [95]. In such systems the kernel transfers control to a dedicated process running under the hypervisor to provide operations on the cryptographic keys. The advantage is that the dedicated process could be kept intact even when the operating system is compromised.
The adversarial model

In the rest of this thesis we refer to any isolation mechanism, either software or hardware, simply as ‘security module’. Indeed, all types of isolation mechanisms have a common purpose, namely to prevent the exposure of keys, and share common threats. These threats arise from the fact that they are expected to protect the keys even if they are the only uncompromised subsystem. This can be modeled by an adversary who obtains temporary access to the module, meaning that all operations available from the module to the system, may at some point, be available to the adversary. Hence, he may access temporarily cryptographic operations on the stored keys using data of his choice (e.g., mount an adaptive chosen-plaintext attack).

The temporary access of the adversary indicates that at some point that access should be revoked, either because of detection or due to a system upgrade. That is important because if the adversary could obtain access to the system on a permanent basis, he has capabilities identical to the system and no isolation technique can mitigate that.

Nevertheless, one may question the practical relevance of this type of adversary. We argue that this adversary models both on-line and off-line attacks.

In the on-line world, system compromises are a commonly occurring issue [85], and this adversary literally matches the possible attacks on security modules within these systems. For example a web server that uses TLS [34] in combination with a security module, when compromised provides the adversary
with the same access as modeled. If the server uses a security module that protects the keys from such an adversary, once the compromise is terminated, e.g., after a scheduled software upgrade or an adversary detection, the stored keys can be trusted again.

Perhaps unsurprisingly, off-line threats are also modeled with the same adversary. Systems where security modules such as smart cards are distributed to the users, share the same threats. That is, an adversary may obtain temporary access to the modules, even though the means are different. For example, smart cards can be shared temporarily by the users or used in tampered terminals, actions that are equivalent to an adversary with temporal access. Strict policies against sharing may not be completely effective to eliminate such threats, since user studies have shown that users share credentials such as passwords in spite of any prohibition \[89\]. The fact that a smart card is a physical object that can be taken back would hardly improve this user behavior. Furthermore, even in environments where users are security conscious and will not lend their cards, unauthorized “borrowing” or usage in a tampered terminal cannot be easily prevented, and even worse they cannot be detected by the owner once the card is returned.

**Evaluating the security of security modules**

Since security modules are the only access point for cryptographic operations using the protected keys, it is important that the operations they provide are selected in a way that no combination of them is sufficient to expose the keys. This requirement, although simple to formulate has been shown hard to prove or assure in real world security modules \[6, 16, 18, 27\].

One of the reasons behind that is that typical security modules are created to operate with multiple protocols that have different requirements on operations. For example modules like IBM’s CCA \[50\] and modules compliant with the PKCS #11 API are described in documents of more than 500 pages and 1500 pages respectively, which reveals the level of complexity involved. Their design is complicated not only because of the numerous protocols involved, but also because of the different variants of existing algorithms used in protocols and the inclusion of questionable operations required by obscure protocols. Ensuring the protection of cryptographic keys when there is a proliferation of available operations is a Herculean task, and unsurprisingly several serious flaws have been found in prominent security modules \[6, 16, 27\] that allow the keys to be derived by a combination of operations.

\[^5\text{PKCS #11 [83] is an API, published by RSA Security designed to suit requirements of hardware security modules, where separation of stored cryptographic keys and cryptographic operations is mandated by the module itself.}\]
The current approach to ensure the protection level of the security modules, is to verify the modules' security claims (e.g., inability to extract the key) by formalizing the available operations and using a model checker. This approach has been followed in a few cases [31, 17] and helped not only to identify issues in modules but also to fix them. However, the goal of a formal proof of key safety in an arbitrary security module is on-going work, and currently we only have indications of security using model checkers in designs that support only a limited number of operations.

An alternative approach to ensure the protection of the keys is to delegate the security of the module to the algorithms used. That is, to fix the supported operations of the security module to a minimum subset that the algorithms are designed to protect, when controlled by an adversary and prevent any interaction between keys of different algorithms. For example encryption and decryption operations provide an attacker with the ability of adaptive chosen-ciphertext and chosen-plaintext attacks, attacks that symmetric cryptographic algorithms like AES [30] are designed to resist. Public key cryptosystems such as the RSA-based ones [54], when used for encryption are also believed to be resistant to these attacks. In addition, signing operations require the underlying cryptosystems to be secure against adaptive chosen-message attacks [44]; and cryptosystems such as RSA-based, DSA or ECDSA [78] are believed to be fulfill this requirement. Furthermore, one should prevent that keys interact in order to avoid any risk of vulnerability and maintain the security reductions for these schemes [54]. Despite the apparent advantages of this approach, it is not known whether it is practical, i.e., whether existing protocols can take advantage of such a security module.

**Combining security modules with secure communications protocols**

Ensuring the protection of the stored keys in a security module is an important task that benefits the users of such modules. Unfortunately, it is not sufficient to protect a secure communications protocol that utilizes such modules in the presence of the modeled adversary. We demonstrate that using the following example. Imagine a security module that provides only the AES encryption operation on a stored key. Since AES is considered to be practically secure under adaptive chosen plaintext attacks, the protected key should be considered secure even if the security module in which it is stored falls in the hands of the adversary for a limited time. Consider now the Wide Mouth Frog protocol in Fig. 2.6. It is a key transport protocol that was shown to be secure [19] against Dolev-Yao type adversaries (that control the network), using BAN logic.

The protocol is initiated by $A$ who sends to the trusted party $S$ a timestamp $T_A$, the identity of the party $B$ he wishes to communicate with and a freshly
generated key $K_{AB}$, encrypted using the key $A$ shares with $S$ (which is stored in his secure module). After receiving that message $S$ verifies that the timestamp is recent and forms a message for $B$ that contains a timestamp, $A$'s identity and the key sent by $A$, encrypted using the key $S$ shares with $B$.

The protocol can be abused by an adversary that temporarily obtains $A$’s security module as follows. After obtaining the module the adversary generates multiple messages that contain future timestamps that extend to the period of a few years, the identity of $B$ and a random key. Then he encrypts all these messages using the module and returns the module. Now, without the possession of the module the adversary can initiate sessions at the pre-selected times in the future.

This example illustrates that combining secure communications protocols with security modules is not a straightforward task. Even if a secure protocol is combined with a secure module, the resulting system may be insecure. There are few attempts to ensure the security of the resulting system, which we discuss in the next paragraphs. Note that in certain cases we will use the term smart card to be consistent with the terminology used originally at the described work, even though it is a synonym to the security module term.

**BAN logic.** Abadi et al. enhanced BAN logic [19] to account for smart card related threats in [20]. The BAN logic is a logic of beliefs used to prove certain
properties in authentication protocols. The smart card extension is the first known formal treatment of smart card threats written when the notion of a smart card was not clearly defined. As such the protocols that are studied utilized smart cards that differ from smart cards that resemble the features of a modern smart card, to smart cards that include batteries (to maintain a clock), screen and keypad. They consider a Dolev-Yao network adversary with the additional abilities for smart card theft and terminal compromise. The latter is a main concern, possibly because at that time smart cards were simple containers of data that were read by the terminal, and they dealt with the issue explicitly in the studied protocols, by including verification of the terminal.

Without getting into the details of the BAN logic we describe the additions required for the smart card modeling. In their model, the user and the smart card are different entities that share a secret (the PIN). The original logic is enhanced with a notion for secure channels and the notion of timely channel, a channel which was established recently, and this logic is being used to verify and prove correct delegation-based protocols, i.e., protocols that allow delegation of authority such as the user’s card signing the terminal’s credentials for certain time to authorize the terminal to act on his behalf. The main and most advanced protocol is a secure communications protocol, shown in Fig. 2.7. There are four roles in that protocol, a trusted certificate authority, the user $U$, the smart card $S$ and the terminal $T$. The smart card contains a public and private key pair $K_S$ and a certificate signed by the authority cert$_S$. The terminal also possesses a public key pair $K_T$ and cert$_T$. The protocol briefly relies on the smart card signing a certificate containing a temporary public key of the terminal in order for the terminal to operate on behalf of the user.

The channel notion of the extended logic is then used to prove desirable properties of the protocol.

**The Shoup-Rubin game-based method.** Shoup and Rubin proposed in [92] a game-based model to capture the usage of smart cards with symmetric keys in key distribution protocols involving a trusted party. The threat model assumed is a static corruptions model (Dolev-Yao adversary) extended to model smart card theft, terminal tampering, etc. It is one of the first works to discuss the security implications and gains of using smart cards to protect cryptographic keys in general purpose protocols. The formal model used is similar to the Bellare and Rogaway model [10], which assumes a security parameter $k$, a number of hosts $n$, and a trusted server $S$. Each host is given a smart card with long-term key $K_i$, $1 \leq i \leq n$ and $S$ is given key $K$. The smart card is modeled as a stateless probabilistic oracle. On input $x$ it returns $f(k, x)$, a function of the key. Each host $i$ may communicate with a host $j$ by using a process $\Pi(i, j, u)$, where $u$ is a process identifier to allow more than one
connection. The adversary is a polynomial time probabilistic algorithm that initializes and interacts with the system. The interaction is a query/answer based communication with the processes and the server. The “transcript” is a list that contains the queries and answers ordered in time. The allowed queries to a process are delivery of a message, and response, as well as three special queries (1) for a process to reveal its session key, and (2) for a smart card to reveal its long-term key and (3) to access the smart card oracle. If the latter special queries are used, the process or the smart card are considered to be opened. During the interaction each process may output a message indicating acceptance, which indicates that a session key was established.

In this model a process that accepted is defined to contain a fresh session key if the following conditions hold.

---

Figure 2.7: The first smart card protocol in [20].
• The process has not been opened;
• the host or its partner smart cards have been not opened;
• the partner’s smart card is not accessed between the time of a host’s first and last query.

At the end of the interaction the adversary plays a game. It may quit and score zero, or select a process holding a fresh session key. In the latter case the adversary is provided with the processes’ session key, and a randomly generated string and must guess which one is which. If the guess is right the adversary scores 1, otherwise -1. The definition of security in this model is the following.

1. If the adversary faithfully forwards all messages between the processes and the server, the two processes accept and share the same session key;
2. The expected value of the score of the adversary is negligible as a function of the security parameter $k$.

Under this model the authors propose a key establishment protocol for smart cards and prove it secure. The model is the first known to combine the study of security modules with protocols and to assume a proper threat model. The model, however, is restricted to key establishment protocols with a trusted third party.

Inductive verification. Bella uses mathematical induction in [8] as the main tool to prove the protocol’s security goals with respect to smart cards that protect symmetric keys. The possible threats are modeled as a set of events, defined by inductive rules. The security goals are then proved using induction on the set. The adversary model is similar to the model of Shoup-Rubin, with an additionally modeled threat which the author calls “data bus failure.” This threat models smart card tampering in a way that messages from the card to the reader are modified or removed. The author then models the Shoup-Rubin protocol in [92] and uses the ‘Isabelle’ tool [80] to prove its security claims.

Resettable zero knowledge. Canetti et al. in [22] introduce the notion of resettable zero knowledge (rZK). This notion expresses an improvement over the classical notion of zero knowledge. Protocols under rZK remain secure even after a prover is reset to its initial state and re-uses the same random numbers. The resetting property in this notion is of particular importance to certain smart cards (e.g., smart cards that do not support atomic updates) due to their nature of being under the complete control of the adversary. A protocol that satisfies the rZK definitions is suitable for a cryptographic protocol that utilizes smart cards.
Limitations of the current verification approaches

As previously discussed, there are two issues that existing verification approaches for security modules try to address. The first is to ensure protection against key extraction, i.e., there is no combination of operations that can be used to extract the key. The other is to ensure that combining a security module with the protocol is secure, and there are no values that can be extracted from the module that would compromise the security of past or future sessions. The two issues are typically handled separately, but as demonstrated, both are required to ensure the security of a module.

However, the existing methods for verification when combined with protocols are of limited applicability. The BAN logic based method is designed for protocols that use delegation. Hence, its applicability to current real world protocols that, typically do not involve delegation, is limited. The Shoup-Rubin and the Inductive verification approaches are both limited to long-term symmetric keys, something that makes them unsuitable for the current protocols that typically use long-term asymmetric keys. Furthermore, the resettable zero knowledge notion may be interesting in the case of certain restricted smart cards, or for the design of future protocols, but does not apply to any existing protocols that do not involve algorithms with very strong guarantees such as resettable zero knowledge.

A generic approach to security module verification in combination with a protocol is thus an open problem.

2.4 Cross-protocol attacks

The term ‘cross protocol’ attack is used to describe attacks that exploit interactions between different protocols or sub-protocols.6 A prominent example is the HTML forms [97] attack which deceives a web (HTTP) browser into delivering content to an unrelated protocol such as mail (SMTP, IMAP, POP3). New attacks of this family are occasionally discovered, e.g., a cross-protocol attack between the HTTP and Websockets protocol [48].

While the threat of such attacks has often been highlighted, only few practical attacks existed for secure communications protocols. Kelsey et al. were the first to publish a study describing possible threats [57] and showing that the current design and verification methodology was not sufficient to handle

---

6Because a modern communications protocol may often be a shell protocol that negotiates protocol options that could be entirely different, we use the term ‘sub-protocol’ to describe those options.
the issue. Their argumentation was based on the presentation of two secure protocols which are rendered insecure once their long-term keys are shared. Furthermore, they propose guidelines for preventing such attacks. Other work further expands on the topic and describes cross-protocol attacks on proposed, fictional and modified real world protocols [1, 29].

Previous studies highlight the dangers of key sharing between different and unrelated protocols, but this practice is not widespread (possibly as a result of that work) and no real world attacks are known. On the other hand cross-protocol attacks between different versions or different sub-protocols within the same protocol exist. Wagner et al. were the first to present in [104, 105] a cross-protocol attack on the SSL 3.0 protocol by utilizing interactions between different ciphersuites (sub-protocols). Barkan et al. showed a cross-protocol attack applicable to GSM (mobile) networks [7] that was based on the fact that GSM mobiles can be made to use the same key for different ciphers. Yu et al. show in [111] an attack against servers that support both V4 and V5 versions of the Kerberos protocols. All attacks are possible because keys are typically re-used in different versions of protocols or in different sub-protocols of a protocol.

In the next paragraphs we provide more details on the existing attacks on deployed protocols and continue with a discussion about the current consensus on the prevention of these attacks.

2.4.1 Existing cross-protocol attacks

The attack on Kerberos V5

Kerberos is an entity authentication and key distribution protocol originally proposed in 1988. Its main goal is to establish fresh session keys between users and servers and, as a result, enable users to log into multiple servers that belong to a common infrastructure. To this end, a user first requests an electronic ‘ticket’ from a central Key Distribution Center (KDC). That ticket enables the user to log into a server that is part of the infrastructure. The main idea is depicted in Fig. 2.8.

There are two versions of the protocol, namely V4, which is the initial 1988 protocol [94] and V5 [60, 77] which corrected several issues of the original. One of their differences is the way the KDC ticket is encrypted. In both versions the ticket is encrypted using the key shared between the server and the KDC, using a fixed IV, i.e., a sequence of zero bytes in des-cbc-md57, but additional

7Note that the attack applies to the Kerberos V5 protocol as published in 1993 [60], which at the time was the current version.
Cross-Protocol Attacks

The Kerberos protocol

Server  User  KDC

\[
\text{ticket request} \quad \rightarrow \quad \left( \text{ticket} \right)_{K_s} \quad \rightarrow \quad \left( \text{ticket} \right)_{K_u}
\]

Figure 2.8: A simplified run of the Kerberos protocol. \( K_s \) is the key shared between KDC and the server, and \( K_u \) the key shared with the user.

formatting is applied in Kerberos V5. The new and old formats are shown in Fig. 2.9. The new format consists of the encrypted ticket data prefixed with random data and a checksum (which in the most current versions of the protocol is an HMAC), while the old format is a direct encryption of the ticket data. That change was to prevent the ticket from being used by an adversary as an oracle to obtain arbitrary data encryptions from the KDC using the key shared between the KDC and the server.

\[
\text{random data} \quad \text{checksum} \quad \text{ticket data} \quad \text{pad} \quad \{ \text{Kerberos} \}_{V5}
\]

\[
\text{ticket data} \quad \text{pad} \quad \{ \text{Kerberos} \}_{V4}
\]

Figure 2.9: Ticket data prior to encryption for Kerberos V4 and Kerberos V5.

Yu et al. in [111] noticed that in an implementation of Kerberos where both V4 and V5 protocols were supported, the same key \( K_s \) was being used to encrypt all tickets between the KDC and the server. Because certain parts of the ticket data could be manipulated by an adversary (e.g., when it controls another KDC server in the network), the ticket obtained by the V4 KDC could be used as an oracle by the adversary to obtain an encryption of data of his choice in
the V4 server with the key $K_s$ and then utilize them with the Kerberos V5 protocol.

In particular, the adversary obtains a $K_s$ encrypted ticket from the Kerberos V4 server with a client name of “a234567XXXXXXXX.” Given the format of the V4 ticket data in Fig. 2.10, under the des-cbc-md5 cipher the first block consists of the flags and the fixed part of the string and results in ciphertext $C_0$, and the $X = “XXXXXXXX”$ part will be the second CBC-encrypted block $C_1 = E_{K_s}(X \oplus C_0)$. Because the IV used for this encryption is a fixed sequence of zero bytes $C_0$ is an ECB encryption of the provided data. Then by repeating the ticket granting process, but this time by replacing “XXXXXXXX” with the value of $C_0 \oplus M$, where $M$ is a message of the adversary’s choice, the adversary can obtain the encryption of any messages of his choice with the key $K_s$. That is because of the cancellation property of the exclusive or used in CBC; hence $C_1$ will be $E_{K_s}(M)$.

While the the Kerberos V5 protocol is immune to this attack because of the random data prefix in the ticket (which serves as a random IV), the combination of the V4 and V5 server makes both versions vulnerable to the attack.

The attack on GSM

GSM is a set of protocols defined by European Telecommunications Standards Institute (ETSI) for mobile networks [40]. Among others it provides authentication of the user and encryption of the communication. The original protocol defined the (secret) encryption algorithms A5/1, the strong encryption algorithm of the standard, A5/2, a deliberately weakened encryption algorithm, and later variants include A5/3, an encryption algorithm based on KASUMI [39]. The protocol further utilizes the A3 and A8 authentication and key agreement algorithms, which may vary from network to network. The weak cipher A5/2 was widely implemented into mobile phones to allow interoperability with networks that were capable only of A5/2.

The protocol works as shown in Fig. 2.11. The network sends the client $RAND$, a 128-bit nonce, and the client replies with the 32-bit $SRES$ value, that depends on its authentication key $K_i$ (typically of 128-bits). The network authenticates the client by verifying the $SRES$ value to match with its own generated value, using its copy of $K_i$, and both parties generate a key $K_c = A8(K_i, RAND)$ to be used in the subsequent communication.

Barkan et al. in addition to an efficient key recovery attack on A5/2 in [7] describe three cross-protocol attacks, that are based on the fact that the same key $K_i$ is used to generate keys of any of the available A5 ciphers.
The first attack is a client impersonation attack and requires the adversary to operate as man-in-the-middle between the network and the client. The adversary is forwards the `RAND` message received from the network to the client and the reply `SRES` to the network. At that point the adversary (who doesn’t know the generated key $K_c$) asks the client to encrypt using A5/2. The protocol doesn’t authenticate this request so the client complies believing the request comes from the network. Then the adversary uses the cryptanalytic results for A5/2 to recover $K_c$. That $K_c$ value can be used by the adversary to impersonate the client in his current connection with the network.

The second attack is a cipher downgrade attack, and the adversary again operates as a man-in-the-middle between the network and the client. At this time the adversary modifies the capabilities of the client (which are not authenticated) to only advertise support for the A5/2 cipher. If the server agrees the adversary will be able to recover the key used in A5/2 and eavesdrop

---

**Figure 2.10: The format of a Kerberos V4 ticket.** The fields marked as string are null terminated strings of length \( \leq 40 \).

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flags</td>
<td>[0:7]</td>
</tr>
<tr>
<td>client name</td>
<td>(string)</td>
</tr>
<tr>
<td>client instance</td>
<td>(string)</td>
</tr>
<tr>
<td>client realm</td>
<td>(string)</td>
</tr>
<tr>
<td>client address</td>
<td></td>
</tr>
<tr>
<td>session key</td>
<td></td>
</tr>
<tr>
<td>lifetime</td>
<td>[15:23]</td>
</tr>
<tr>
<td>timestamp</td>
<td></td>
</tr>
<tr>
<td>service name</td>
<td>(string)</td>
</tr>
<tr>
<td>service instance</td>
<td>(string)</td>
</tr>
<tr>
<td>pad</td>
<td>(( \leq 7 ) bytes)</td>
</tr>
</tbody>
</table>
any conversations.

The last attack stems from the fact that many networks don’t perform the authentication process often and re-use the key generated in the last session. It is an attack that compromises past sessions established using A5/1 or A5/3. The adversary in this attack seeks to discover the key $K_c$ used by the client in those sessions; in a setup similar to the first attack, he asks the client to re-authenticate and re-uses the $RAND$ value that was used in the target session. Because $K_c$ only depends on the network provided $RAND$, the client generates the same $K_c$ as in the previous session and is asked by the adversary to utilize A5/2. At this point the adversary uses the cryptanalytic results for A5/2 to recover $K_c$ which is used in turn to decipher the past sessions.

**The Wagner and Schneier attack on SSL 3.0**

Wagner and Schneier describe in [104] a server impersonation attack on the SSL 3.0 [42] protocol. Although this attack proved to be impossible in practice due to an incorrect interpretation of the protocol [71], the underlying idea is worth recalling because it highlights the threats due to interaction between sub-protocols. The attack transforms a server into an oracle that signs messages submitted by the adversary. In particular the server is used by the adversary to sign DH group parameters, which are presented to the client as RSA parameters.
This allows the recovery of the client’s secret by the adversary and eventually to the establishment of a secure session between the adversary and the client. In that session the client is convinced that the adversary is the server he intended to connect to.

The attack is based on the observation that the digital signature in a DH key establishment does not cover any identifier of the negotiated ciphersuite. According to the SSL 3.0 protocol [42] when the DH key establishment method has been negotiated, the group parameters and the session public key are sent by the server in the ‘ServerKeyExchange’ message as shown in Fig. 2.12a. The signature on that message is calculated on the algorithm parameters, and the nonces exchanged by both parties. The crucial observation is that the negotiated key establishment method is not part of this signature.

\[
\begin{array}{c|c}
\mathcal{L}_p & \text{length of } p (\mathcal{L}_p) \\
 & p \\
\mathcal{L}_g & \text{length of } g (\mathcal{L}_g) \\
 & g \\
\mathcal{L}_{Y_s} & \text{length of } Y_s (\mathcal{L}_{Y_s}) \\
 & Y_s \\
\end{array}
\begin{array}{c|c}
\mathcal{L}_m & \text{length of } m (\mathcal{L}_m) \\
 & m \\
\mathcal{L}_e & \text{length of } e (\mathcal{L}_e) \\
 & e
\end{array}
\]

(a) Diffie-Hellman         (b) RSA-EXPORT

Figure 2.12: The contents of the ServerKeyExchange message in Diffie-Hellman and RSA-EXPORT key establishment methods in TLS and SSL 3.0 protocols. Each row represents a 2-byte (16-bit) field, unless the length is explicitly given. All indicated lengths are in bytes.

This omission allows an adversary to re-use a signed ‘ServerKeyExchange’ packet in another session, with another key establishment method, by initiating a parallel connection to the server. The attack deceives a client who advertises
a ‘TLS_RSA_EXPORT’ ciphersuite and expects temporary RSA parameters in the ‘ServerKeyExchange’ message, into receiving DH parameters from a ‘TLS_DHE_RSA’ ciphersuite. Note that, the RSA-EXPORT key establishment requires the server to generate a temporary 512-bit long RSA key pair and include it in the ‘ServerKeyExchange’ message. In both DH and RSA-EXPORT the parameters are signed using the same RSA key. Note that, although, the server could in theory use different keys for different ciphersuites (sub-protocols) this is neither mandated by the protocol, nor a standard practice.

The attack assumes that the client reads and verifies the signature, and then reads the RSA parameters (see Fig. 2.12b) one by one, yielding the following scenario. The client verifies the signature, reads the RSA modulus $m$, which corresponds to the prime of the DH group $p$, and then reads the RSA exponent $e$ field which corresponds to the group generator $g$. Therefore, the client encrypts the pre-master secret $k$ as $k^g \mod p$ and includes it in its ‘ClientKeyExchange’ message. Since $p$ is a prime number and $g$ is known, it is very easy to compute the $g$-th root of $k^g$ to recover $k$, which allows the adversary to impersonate the server. Note that the ‘Finished’ messages that provide handshake message modification detection using message hashes encrypted and authenticated with the session keys, cannot detect this attack since the adversary recovers the pre-master secret.

The authors noticed that the SSLRef 3.0b1 implementation was immune to the attack and attributed the failure to a paranoid sanity check of this particular implementation. However, careful examination of the TLS packet parsing reveals that the failure of the attack is due to the serialized way TLS packets need to be parsed. The variable length vectors [42] used in the structure definition in Fig. 2.12 require an implementation to read the vector length before reading data, hence an implementation can only start parsing the packet serially, i.e., from start to end without being able to read a field before reading the previous one. In the RSA case, a client would have to read the modulus length, then the modulus, and the same for the exponent and signature fields. If the DH ‘ServerKeyExchange’ packet, which contains one additional field, is substituted, that field will be read instead of the signature and verification fails.

Even though the Wagner and Schneier attack fails due to a parsing error, it demonstrates the idea of a cross-protocol attack utilizing two of the SSL 3.0 key establishment methods, the DH key establishment and the RSA-EXPORT key establishment.
2.4.2 Preventing cross-protocol attacks

Kelsey et al. [57] provide a list of rules that can reduce the risk of cross-protocol attacks, to protocols that follow these rules. They are summarized below.

1. The re-use of keys across protocols must not be allowed. If a single key must be present then this key should certify the other dedicated keys for each protocol.

2. Each operation using the key must include an identifier of the specific application, protocol, version, or step in the protocol.

3. (in signature protocols) Include a fixed unique identifier in a fixed place in the authentication protocol.
4. (in protocols with encryption) In public and secret-key encryption operations and key-derivation operations, the unique identifier should be used in a way that makes the message impossible to decrypt without using the right unique identifier.

5. Smart cards should be aware of the above techniques.

The last three rules are clarification of the first two in specific cases. Cremers in [29] is also stressing the need for item 2, by requiring a context aware tagging scheme. However, even if the suggestions seem reasonable they may not be easy to deploy in practice. Below we summarize some issues.

**Lack of universal identifiers.** There is no notion of universal identifiers to be used in protocol messages that uniquely specify the protocol, the version and the session. Even protocols designed within the same organization, e.g., the IETF protocols, do not adhere to a unique identification rule, demonstrating the practical difficulties of an extensive and globally managed identifier space. However, this lack of universal identifiers may not be of great importance as the practice of key re-use within unrelated protocols is not widespread.

On the other hand, key re-use is often encountered, and not prohibited by the relevant protocol specifications, e.g., in [34], within different protocol versions, or when negotiating different cryptographic primitives (e.g., key establishment) on the same protocol. That practice may simplify the protocol specification and implementation by preventing a proliferation of keys that may not be easy to manage, at the risk of being vulnerable to cross-protocol attacks. For that we believe that the point (2) above is especially relevant in modern protocols despite the lack of a universal protocol identifier.

**Restricting key usage.** While using keys for specific purposes is today a reality, e.g., the GNUPG program generates different subkeys for signing and encryption, signed by a master key, and PKIX [28] certificates include key usage fields, there could be issues when the determination of the key purpose is delegated to end-users.

Alves-Foss indicates in [1] that if protocol identifiers are to be embedded in public key certificates, e.g., by using the extended key usage field of PKIX, these will have to be set by their owners, who may not understand the underlying risks and incorrectly tag their certificates. Over a decade after his comment, we can see in Table 2.1 and Table 2.2 that almost half of the HTTPS server certificates do not properly set their key purpose, and the majority of RSA keys, despite the recommendations against that [54], do not restrict the intended key usage either (e.g., to either signing or encryption).
Table 2.1: A summary of the ‘Extended Key Usage’ field of all web server certificates in the 2010 EFF SSL Observatory data. That field allows to enable specific flags in a certificate that mark the intended purpose of its key.

<table>
<thead>
<tr>
<th>Extended key usage</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPS server flag</td>
<td>2 160 032</td>
<td>53.7</td>
</tr>
<tr>
<td>No flags</td>
<td>1 861 734</td>
<td>47.3</td>
</tr>
</tbody>
</table>

Table 2.2: A summary of the ‘Key Usage’ field of RSA web server certificates in the 2010 EFF SSL Observatory data.

<table>
<thead>
<tr>
<th>Public key algorithm</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA encrypt</td>
<td>19 770</td>
<td>0.49</td>
</tr>
<tr>
<td>RSA sign</td>
<td>21 005</td>
<td>0.52</td>
</tr>
<tr>
<td>RSA both</td>
<td>3 978 821</td>
<td>98.99</td>
</tr>
</tbody>
</table>

**Formal verification.** While it is possible to formally verify the security of a secure communications protocol [23, 53], the current models of verification cannot address cross-protocol attacks. That is mainly because the formalization of a protocol (e.g., in [53, 15]) includes a single variant and version of the protocol rather than all the available options. That way interactions between different versions of the protocol and interactions between sub-protocols are not captured. More recent work [13] provides mechanized proofs of security in the TLS handshake, and explicitly considers cross-protocol attacks. It is however, limited to the RSA and DH set of TLS ciphersuites, and its positive results are limited to interactions between them. Nevertheless, that work indicates that more and more aspects of protocol security verification are captured by modern models.
Consider also that very philosophical and witty answer of Diogenes to the man who asked, ‘How shall I avenge myself on my enemy?’ ‘By becoming a good and honest man.’ ”

Plutarch – Morals

3

Contributions

3.1 Cryptographic key protection

Our initial attempt to approach the field of cryptographic key protection was through obfuscation and white-box cryptography. However, the limited applicability of that field in modern applications, encouraged us to investigate to alternative approaches to key protection such as subsystem isolation. The next paragraphs summarize our contributions to these fields.

3.1.1 Obfuscation and white-box cryptography

To cope with the inconsistency between the formal evaluation of obfuscation and white-box techniques, and the real world practice of utilizing obfuscation techniques, we follow an alternative approach to the evaluation of existing obfuscation techniques. In order to quantify their protection level, we propose a classification of the adversary’s capabilities based on the limitations of the available attack tools (see Sect. 5.1 and [66]), similarly to the approach initially followed by Aucsmith in [4] to evaluate tamper-resistance. Our approach specifically targets obfuscation techniques that utilize self-modification.
The main idea behind this decision is that in a typical developer’s environment the available debugging and disassembling tools are not perfect and have limitations that are exploited by obfuscation techniques such as self-modifying code. Several of these limitations are due to the tool’s reliance on static analysis techniques of the executable, as discussed in [101] from the perspective of reverse engineering. Based on these facts we devise gradually increasing adversary capabilities that map on the existing debugging and disassembling tools, and vary from simple static analysis techniques, such as the disassembler, to adversaries with an ideal debugger that has no imposed limitations. Based on these adversaries, we devise attributes based on the existing obfuscation practices in order to perform a taxonomy of obfuscation techniques that employ self-modification. Our results uncover the various different protection levels offered by these techniques.

We considered adversaries with capabilities that are gradually increasing, starting with the availability of a simple disassembler, to the availability of an ideal debugger as in [64] or specialized tools designed to attack a particular method such as a tool to circumvent self-checking code. That analysis is based on a modeling of the imperfection of today’s real world tools available for reverse-engineering. Hence a subjective factor in the modeling of “difficulty” for reverse-engineering is introduced, in a way that our study depends on the current-day technology. This dependence, however, provides the ability to assess methods that are used in practice, but due to the negative theoretical results (see Sect. 2.3.1), would otherwise be indistinguishable.

3.1.2 Subsystem isolation

Development of a kernel-based isolation system.

An important limitation of the Linux-kernel based systems, that we attempted to address, is the lack of a framework to access cryptographic accelerators in user-space. Other operating systems, e.g., OpenBSD, had already solved the issue with the introduction of the /dev/crypto device [59], which allows access to cryptographic accelerators via the ioctl system call. While porting this framework to the Linux kernel we realized the advantage of converting this device to a security module. That would transform it from a device that merely provides access to a cryptographic accelerator, to a device that is able to manage keys and provide operations to keys without exposing them, in addition to any acceleration capabilities.

The main design goal was to use the framework to protect applications from accidentally exposing keys through a vulnerability. Therefore, we reworked
the original design of the /dev/crypto device to construct a software security module (see Sect. 5.2 and [70]), to take advantage of the separation of Linux kernel from user-space in order to protect the keys. The keys are accessed from the user-space applications using a new device (/dev/ncr) and the ioctl system call, but only indirectly, by using references. To ensure that no combination of operations is sufficient to expose the protected keys, the allowed cryptographic operations were limited to the operations the supported algorithms were designed to protect from. Furthermore, care was taken to avoid all pitfalls and known attacks from previous designs [27, 17].

Once the implementation was complete, we discovered that it outperformed any user-space security module, that utilized process level isolation, in the same system. We attribute that performance benefit to the additional overhead of the inter-process communication, compared to the user-space to kernel-space communication.

An attack on public key Kerberos (PKINIT).

Despite the fact that smart cards and other security modules are widely used in combination with security protocols (e.g., for authentication), they are usually handled only as a deployment issue in security protocol design. However, as previously demonstrated, even though the protocol and the security module may be both considered secure, their combination is not necessarily secure.

We further demonstrate this fact by presenting an attack on a protocol mainly used with smart cards, the public key Kerberos (PKINIT) protocol, and propose a simple fix to the protocol (see Sect. 5.3 and [67]). In addition we argue the need for certain protocol properties in protocols that use smart cards for entity authentication. These properties would not only ensure the secrecy of past and future sessions, but also prevent any impersonation of the user, or other parties to him outside the time-frame of the compromise, assuming that the adversary is not able to extract any long-term secrets from the card. A summary of the properties is shown below.

- **SC forward secrecy:** Session keys that were established with a user’s smart card over a non-corrupted terminal remain secret, even if an adversary later obtains access to that user’s smart card.

- **SC backward secrecy:** Session keys that are established with a user’s smart card over a non-corrupted terminal remain secret, even if an adversary had previously accessed the user’s smart card.
• **SC key-compromise impersonation**: An adversary with access to a user’s smart card, should not be able to impersonate other entities to that user, as long as the user’s terminal is not corrupted.

• **Possession-based authentication**: An adversary with access to a user’s smart card, should be able to impersonate that user only for as long as he has access to his card.

All of the properties above ensure that for smart card-related threats, the damage caused by any adversary is limited to the time it had access to the card, e.g., a tampered terminal can only authenticate as a user while the card is inserted, but not later. Once card access is revoked no further damage can occur, and the card can still be relied on. We believe that this is a reasonable expectation stemming from the fact that smart cards are possession-based authentication tokens; non-possession must lead to the inability to authenticate.

**A model for the verification of smart card-based key establishment.**

Proposing desirable properties of security protocols when combined with smart cards, is a first step in the security evaluation of a protocol. Another important step is the developing of a formal model under which the properties are verified. Unfortunately, despite the fact that smart cards have been introduced to a multitude of protocols, to the best of our knowledge, only few works provide a formal treatment of smart card-based protocols against an adversary that is able to attack a user’s terminal independently from his smart card (see, for example, [92, 8]). These treatments are limited in scope and do not provide the necessary functionality to prove real world protocols secure.

For this purpose, we have enhanced (see Sect. 5.4 and [68]) Shoup’s simulatability-based static corruptions model [91] to account for threats arising from smart card usage. The notion of security in this model depends on an ideal system, with an ideal key establishment that is by definition secure and a real system that describes the actual protocol and participant interactions. A proof of security in this model shows that any attack on the real system can be simulated in the ideal system. Under this model we examine the DH variant of the public key Kerberos protocol (PKINIT) and show that the fixed version of the protocol, as proposed in [67], secure.

The new model’s applicability is not restricted to the Kerberos protocol and applies to any protocol that can be expressed in Shoup’s model. Given its broad scope, the simplicity behind the original model, and the fact that it incorporates a composition feature [91] we believe that the smart card version
of the model is a practical tool that can be used to verify other real world smart card-based protocols.

3.2 Cross-protocol attacks

In Sect. 5.5 and [71], we describe a cross-protocol attack applicable to the TLS protocol 1.2 [34] and its earlier revisions. It can be seen as an adaptation of the Wagner and Schneier attack on SSL 3.0 for the TLS protocol. The attack presents valid elliptic curve Diffie-Hellman (ECDH) parameters signed by a server to a client that incorrectly interprets these parameters as valid plain DH parameters. The attack enables an adversary to successfully impersonate a server to a client after obtaining $2^{40}$ signed elliptic curve keys from the original server. Furthermore, we examine the practical impact of this attack and our findings suggest that although the attack complexity is high for attacking a specific client, the attack may be practical in a scenario where the adversary is interested in attacking any random client.

The attack depends on the server supporting the explicit elliptic curves option. This option is not supported in the tested open-source implementations making them not vulnerable to this attack. This fact suggests that for now the implementation of the explicit elliptic curves protocol option should be avoided unless a counter-measure is in place.

The described attack can be countered by verifying the server’s DH public key for known invalid values. The TLS protocol, however, should not over-rely on these tests. There could be other values that can be used to mount variants of our attack. It is important that the trust in the peer’s signature needs to extend to a trust in the signed parameters and for that we propose a fix to the protocol that makes it immune to cross-protocol attacks [71, 69].

The fix we propose in [71] modifies the TLS server key establishment message to include not only the nonces from the current session but also identifiers of the currently used combination of ciphers (ciphersuite) and other negotiated parameters. The fix is negotiated as an extension and maintains backwards compatibility in the protocol. An updated version of the fix is described in [69] and includes the ability for the server to detect a client that supports the fix, even when under a downgrade attack.

While our attack is an “online” attack, i.e., it requires the original server to be active during the attack, we show that extensions of the TLS protocol such as “False start” [62] that reduce protocol round-trips by sending the encrypted client’s application data before the full handshake is complete, are at risk. The
attack may be used to recover the encryption key, and thus the encrypted data, offline.
“And therefore (proceeded Socrates), before he makes up his mind with what enemy to go to war, a statesman should know the relative powers of his own city and the adversary’s, so that, in case the superiority be on his own side, he may throw the weight of his advice into the scale of undertaking war; but if the opposite he may plead in favor of exercising caution.”

Xenophon — The Memorabilia

4

Conclusions and open issues

4.1 Cryptographic key protection

4.1.1 Obfuscation and white-box cryptography

While obfuscation techniques are used to protect real world applications, their security properties are often evaluated using ad hoc methods, rather than a formal model. One of the reasons is, that under the existing formal models obfuscation of an application is deemed impossible, and thus they are unable to differentiate between existing techniques.

Our classification methodology in [66] provides a tool to assess the protection level of obfuscation techniques, that is not affected by the negative theoretical results from Barak and Saxena [5, 86]. While our analysis is only applicable to techniques that utilize self-modification, we believe it is a first step in the analysis of other obfuscation techniques based on a modeling of the imperfection of today’s real world tools available for reverse-engineering. The extension of this methodology for other obfuscation techniques, e.g., white-box cryptography, is a challenging task that would enable the assessment of more obfuscation methods used in practice.
Nevertheless, obfuscation need not only to be seen as an ad hoc technique that cannot offer any provable level of security. As we have seen, while it is now known to be impossible to create a method to obfuscate an arbitrary application, there are domains where obfuscation is successfully used. For example, the UNIX login program applies a computationally expensive one-way function to the user provided password and compares with a list of stored values [41]. That can be shown to provide obfuscation under the random oracle definition of obfuscation (see Def. 2). Today most operating systems apply similar methods, and this clearly shows that, in spite of the negative results, obfuscation techniques can be used to successfully solve problems in certain application domains.

### 4.1.2 Subsystem isolation

Our implementation of a cryptographic framework for the Linux kernel shows that the design and development of an isolated subsystem in a modern operating system is a non-trivial task that requires an appropriate balance between performance, functionality, and portability. Our design aims for performance without sacrificing the requirements set for key protection, and takes into account known attacks and pitfalls of previous designs to implement its defense mechanisms. That led to a security module that offers the necessary protection to prevent the exposure of the stored cryptographic keys. However, such a design, due to its reliance on the Linux kernel internals, is limited to that operating system. A portable security module would be required to operate in user-space using portable APIs for inter-process communication, e.g., POSIX or SUS. Nevertheless, we have shown that the current IPC APIs are not sufficient to provide comparable performance to a user-space security module with a kernel-based one. Whether efficient IPC APIs are possible in today’s operating systems is an open problem.

On the other hand, we have shown that the availability of a security module is insufficient for the protection of the stored cryptographic keys. There are weaknesses in the currently used methods for security module verification, and even if security modules could be proven secure in isolation, when combined with a secure communications protocol the previous proof is no longer applicable to the overall system. We have identified protocols with issues, i.e., the Kerberos PKINIT variant, when used in combination with smart cards. While the discovery of such issues in an arbitrary protocol may not be as impressive, it is interesting to note that this protocol fails at its primary use case, which is the utilization of Kerberos with smart cards.\(^1\) It is also interesting

---

\(^1\)PKINIT is the protocol used in the Microsoft Active Directory when smart card authentication is required.
that this use-case is never mentioned in the PKINIT protocol description [112]. The latter omission resulted in security proofs for the PKINIT protocol [25, 15] with assumptions that did not reflect the actual protocol deployment. It cannot be known whether mentioning the use-case explicitly in the protocol would have resulted in proofs on a setting closer to the deployed protocol, but nevertheless it would have properly informed an interested third party reading the protocol description and evaluating the existing proofs.

Furthermore, we propose a new method for the verification of secure communications protocols in combination with security modules and use that model to show that the modified version of the PKINIT protocol we proposed in [67] is secure. The model is based on Shoup's simulatability-based static corruptions model [91] and accounts for threats arising from smart card usage. However, it is important to note, that while a verification method for security modules provides a scheme to verify these modules with respect to some preset threats, the effort required for that task is often neglected. Typically such schemes can be used by experts in the field of the theory of computing and allow for sketchy proofs when the brevity is required, e.g., in order to be read by another human, or result in lengthy proofs that may not be easy to be read or verified by another human. For that it is important in addition to such methods, to provide specialized tools similar to [15] that could also allow protocol designers and other interested parties to easily verify such protocols with minimal effort.

4.2 Cross-protocol attacks

Cross-protocol attacks exist as a result of the practice of re-using the same key in different protocols. While this practice is not widespread in distinct and unrelated protocols, it is very common on different versions of the same protocol or when the negotiation of different cryptographic primitives (e.g., key establishment) is allowed by the protocol. One of the reasons could be the fact that it simplifies the protocol specification by preventing a proliferation of keys, at the risk, however, of being vulnerable to cross-protocol attacks.

For that, the previously discussed counter-measures must be considered during a protocol design phase, in addition to any formal verification. This is because while there are several methods for the formal verification of real world protocols, their modeling rarely includes interactions between different versions of the protocol and interactions between sub-protocols. That is indicated in the proof of security on the TLS-DHE combination of ciphersuites in [53], where the authors explicitly exclude such attacks, and in fact our cross-protocol attack in [71] applies to the same set of ciphersuites that are proven secure. Therefore,
it is apparent that the extension of existing formal models to include cross-protocol attacks is an issue that needs to be addressed.

In [71] we have presented a new cross-protocol attack on TLS, an important Internet secure communications protocol. We believe however, that pragmatic solutions to the issue are of equal or more interest. Even though we have attempted to solve the issue in the context of the TLS protocol [69], a practical and universal solution, if possible even, is a challenging task. Whether this attack applies to other Internet protocols, such as IPsec, SSH, or Kerberos, is an interesting and, we believe, yet unanswered question.

It is also important to note that Wagner and Schneier cross-protocol attack on the TLS protocol was known over a decade ago [104, 105]. That attack had a flaw which rendered it impractical and affected only clients that supported the deliberately weakened RSA-EXPORT ciphersuites, and probably for these reasons it was ignored by the TLS working group in later revisions of the protocol [32, 33, 34]. Nevertheless, the main contribution of the Wagner and Schneier attack was documenting a weakness of the TLS handshake protocol that renders a server to an oracle that signs messages for an adversary. That observation was used by our attack to successfully attack another set of ciphersuites, demonstrating that known issues should be solved even if they do not result in an immediate practical attack.
Bibliography


[24] CCIT. Data communication networks: Open systems interconnection (OSI); security, structure and applications - security architecture for open systems interconnection for CCIT applications, 1991. pages 4


[38] EMV. Book 2: Security and key management, 2008. pages 18


[50] IBM. CCA basic services reference and guide for the IBM 4758 PCI and IBM 4764 PCI-X cryptographic coprocessors, Oct. 2006. pages 21


[55] Kasiski, F. Die Geheimschriften und die Dechiffrierkunst. 1863. pages 4


[83] RSA Laboratories. PKCS #11: Cryptographic token interface standard v2.30, 2009. pages 18, 21


[91] Shoup, V. On formal models for secure key exchange, 1999. IACR ePrint archive 1999/012. pages 12, 42, 47


[99] Trusted Computing Group. TCG software stack (TSS) specification version 1.2 level 1, 2006. pages 18


Part II

Publications
“‘Alcinous’, answered Ulysses, ‘there is a time for making speeches, and a time for going to bed;’”

Homer – Odyssey

5

List of publications

Part of this thesis


Other publications

Conferences/Workshops


Submissions to Internet Engineering Task Force (IETF)


5.1 A taxonomy of self-modifying code for obfuscation

Publication data


Contributions

Principal author together with Nessim Kisserli.
A taxonomy of self-modifying code for obfuscation

Nikos Mavrogiannopoulos    Nessim Kisserli
Bart Preneel
Katholieke Universiteit Leuven
ESAT/SCD/COSIC
Kasteelpark Arenberg 10 Bus 2446
B-3001 Leuven-Heverlee
Belgium

August 2011

Abstract

Self-modifying code is frequently used as an additional layer of complexity when obfuscating code. Although it does not provide a provable level of obfuscation, it is generally assumed to make attacks more expensive. This paper attempts to quantify the cost of attacking self-modified code by defining a taxonomy for it and systematically categorising an adversary’s capabilities. A number of published methods and techniques for self-modifying code are then classified according to both the taxonomy and the model.

1 Introduction

There are many reasons to protect a program’s implementation details. Commercial entities generally do so in an attempt to preserve the secrecy of algorithms and protocols that are considered to provide a competitive advantage. While crackers and cyber criminals share this goal, they also seek to confound defense systems, postpone their detection and eventual tracing. The technique by which a program $P$ is transformed into another, semantically equivalent yet harder to understand program $P'$, is known as obfuscation, the notion of which was first formalized by Barak et al. [6].
The term obfuscation is applied to a broad range of techniques, covering both syntactic and semantic transformations with results of varying efficacy. Obfuscation methods include the simple renaming of variables, stripping of comments, flattening of the control-flow graph, replacement of instructions with equivalent ones, including use and emulation of entire instruction sets, and dynamic code creation. The latter approach, when changing existing code, is referred to as self-modifying code, and is of particular relevance to obfuscation because of the inherent additional complexity it introduces. This has also been seen as a barrier to its adoption as conventional programming pattern.

Unfortunately for those entrusting secrets to obfuscation, Barak et al. in [6] established its impossibility under a formal definition. However, despite theoretical shortcomings, obfuscation and self-modifying code are used in practice to raise the barrier for adversaries by commercial entities [40, 14]. Their use also figures prominently in grayer areas of security such as bypassing anti-virus engines [28, 44, 43] and intrusion detection systems [41].

Given that obfuscation is no panacea, those wishing to use it must weigh its deployment cost against the effort required of an adversary to defeat it. The former can be quantified, to some extent, in terms of initial cost of developing the obfuscation, subsequent program execution slowdown, increased file size, and complications to user support. The increased complexity of combining self-modifying code with obfuscation and its effect on an adversary cannot currently be adequately ascertained due to a lack of systematic categorization and precisely defined terminology with which to characterize it.

A taxonomy clearly defines relevant terms enabling the unambiguous, reproducible categorization of items it purports to describe. It is in this spirit that we present our taxonomy of self-modifying code for obfuscation and apply it to a number of published methods. We classify these based on the attributes of a toolset available to an adversary (defined in Section 3.1.1). We do not, however, attempt to capture more subjective and variable aspects such as the adversary’s skill or familiarity with low-level details of the execution environment. Neither the associated cost of developing a tool, nor the technical expertise required to effectively use it are specified; these vary greatly over time as available reverse-engineering and debugging tools become both more effective and easier to use.

Because self-modifying code weakens the distinction between program code and data, we begin the paper with an introduction to code and its use on modern computers, followed by an overview of self-modifying code. The core of the document, Section 3, defines code obfuscation, describes our classification methodology, and includes several real-world examples of obfuscation techniques using self-modifying code. Further discussion of the taxonomy and observations on its relation to the surveyed examples can be found in Section 4.
We conclude in Section 5.

2 The nature of self-modifying code

2.1 An architectural overview

To convey the nature of self-modifying code we briefly examine modern computer architecture. Typical processors based on the von Neumann central processing unit (CPU) architecture execute instructions from main system memory. These instructions are different for each processor and their size varies from a few to several bytes [24, 3]. Although in a typical CPU architecture the main memory may be divided in sections or segments, this distinction is often only logical and the CPU will execute control transfers (i.e. jumps) to memory addresses in different sections. In the Intel x86 [24] family of CPUs\(^1\) for example, applications are executed using the “flat” memory layout. Memory is reserved, into which the application’s code and data are copied, before the system transfers control to it. The memory the application “sees” is virtual, its layout imposed by the system, and initially, only partially mapped to the underlying physical memory. A typical memory layout, under an operating system such as GNU/Linux, is shown in Figure 1. The code section (often referred to as text) holds the application instructions while the BSS\(^2\) and data sections hold uninitialized and initialized application variables respectively. Two special memory sections are the “heap” and “stack”. The former being a variable memory section from which the application’s requests for extra memory are fulfilled. The stack memory section serves multiple purposes including the storage of operating system environment variables, command line arguments, compiler-specific variables, and current program execution context values (such as the saved instruction pointer) used by the CPU’s function call instruction during execution.

Despite the separate depiction of code section from the remainder of the memory sections in Figure 1 such segregation is purely logical. Code placed in any other section will be executed by the processor\(^3\). This lack of distinction between code and data allows programs to store data in memory for later

---

1. Other architectures such as MIPS [45] or ARM [3] operate in a similar fashion, but details may vary.
2. Its name (Block Started by Symbol) is a relic of history with little bearing on its use today.
3. New generation CPUs such as the x86-64, ARMv6 [3, 25] support access permissions for memory pages allowing system and applications to make a distinction between executable memory and memory containing data.
interpretation as instructions, or in other words, it allows self-modification of programs.

2.2 Global overview of self-modifying code

Our paper’s main focus is self-modifying code for obfuscation purposes. However, the following paragraphs briefly illustrate other real-world applications of self-modifying code to provide a broader overview of the technology.

Self modifying code for optimization and extensibility A trend in software that peaked with Sun’s Java language, was the notion of just-in-time compilation [5], usually of abstract machine instructions. Unlike a compiler which translates code to CPU instructions, the just-in-time compiler is part of an interpreter and operates by generating code at run-time. That idea was used by Massalin in [31] to design a proof of concept operating system using self-modification for optimization. Even some mainstream programming languages allow applications to replace and execute their code via code-generation. These include Perl through its eval function, COBOL with the ALTER keyword and a C language extension called “backtick” C [35] allowing dynamic code generation.
Debugging, anti-debugging and tamper resistance  Several debuggers modify the target application’s code. The debugger by Vanegue et al. [49], for example, injects its code into the debugged process. Other designs such as gdb [42] also modify the target’s code to implement breakpoints, by replacing the target code with a software interrupt instruction (on x86 architectures). Anti-debugging techniques [13, 37] often take advantage of such self-modifying behavior to thwart debugging. Additionally, self-modifying code has been used to improve software tamper resistance techniques [20].

Code injection and control flow modification  The lack of distinction between code and data allows for code injection attacks such as “Stack smashing”, “Format string attacks” and many others [2, 32]. In these attacks, the injected code is written to a program’s memory as input data to which control is subsequently transferred. The majority of attacks of this kind are prevented today at the hardware level by marking writable memory pages as non-executable [21, 3, 25]. Attack techniques evolved however, circumventing the protection offered by marked pages by modifying a program’s control flow in so-called code-injection attacks such as return-to-libc and return oriented programming [9, 39, 8],

3 Self-modification for code obfuscation

Self modification has a long history of use in software obfuscation. Early DOS programs used it to hide their copy protection functionality, and shareware programs obfuscated their internal structures against modification [22]. It was even used to prevent software from executing on a competitor’s operating system [38] by storing the code in an obscure format and correctly extracting it at run-time.

In the next sections, we introduce a taxonomy for obfuscation techniques using self-modifying code. We describe the addressed threat model, and discuss our classification method before applying it to the published techniques we survey.

3.1 Code obfuscation

3.1.1 Threat model

The notion of obfuscation was formalized by Barak et al. [6]. Their formalization depends on an obfuscator satisfying the “virtual black box” property, which guarantees that an adversary with access to an obfuscated program should not
learn anything more than is possible via oracle access to the original, unobfuscated program. In other words, access to the program code should not provide more information than can be gained by observing the program’s outputs. Under this definition, obfuscation is proved impossible [6]. Combined with work showing self-modified software to be semantically equivalent to non-self-modified software [7], self-modifying techniques are not expected to provide a provable level of obfuscation.

However, self-modification for obfuscation is usually expected to raise the adversary’s cost of reverse-engineering by the cost required for deobfuscation. This is what we attempt to quantify in this study, based on a classification of attack tools, similarly to the approach in [4]. The threat in this case is an adversary with debugging, disassembling and emulation\(^4\) tools available and seeks to reverse-engineer a program. Additionally, the adversary is assumed to have all required information for executing the software in question\(^5\), but not necessarily any knowledge of the protection method used.

### 3.1.2 Adversary’s capabilities

In a typical developer’s environment [10] the available debugging and disassembling tools are not perfect and have limitations that can be exploited by self-modifying code. The limitations from a reverse engineering perspective

\(^4\)We will not distinguish between a debugger and an emulator as there is no fundamental difference in their capabilities for the purposes of reverse engineering.

\(^5\)For this reason protection methods based on secret information, such as a password, will be ignored.

<table>
<thead>
<tr>
<th>Category</th>
<th>Capability</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Disassembler</td>
<td>Merely obtain an Assembly language description of the program, derived through static analysis.</td>
</tr>
<tr>
<td>II</td>
<td>Debugger with no ability to handle self-modifying code</td>
<td>A debugger limited by its reliance on static analysis to obtain control flow information.</td>
</tr>
<tr>
<td>III</td>
<td>Debugger that handles self-modifying code</td>
<td>A debugger that operates correctly when parts of code are modified.</td>
</tr>
<tr>
<td>IV</td>
<td>Specialized tools</td>
<td>The adversary is able to obtain tools specifically targeting the protection technique.</td>
</tr>
</tbody>
</table>

Table 1: Adversary’s tool capabilities
are discussed in [49]. Of particular relevance here is their reliance on a static analysis of the executable to obtain control flow information.

Not all debuggers have this limitation, thus we model an adversary described by a list of gradually increasing capabilities. This is shown in Table 1, and models adversaries with different capabilities ranging from a simple disassembler, to the availability of an ideal debugger as in [30] or specialized tools designed to attack a particular method such as a tool to circumvent self-checking code [48].

In the first category we model an adversary with static analysis tools, such as a disassembler. Category II models the typical debugging tools available to software developers, such as gdb [42]. Those tools are limited by their reliance on static analysis for information about the debugged program and are thus unable to handle self-modifying code. Category III models the ideal debugger in [30], that has no practical issues when handling self-modifying code. The last category models an adversary able to create or obtain specialized tools with which to attack the binary. We note however that a protection method’s complexity may make it more efficient to create such tools than to mount a lower-category attack as illustrated in Section 4.2.

3.2 Attributes of the taxonomy

This section explores the use of self-modifying code for obfuscation. We survey a number of published techniques, academic and otherwise, and classify them according to the common attributes described below. Taken together, we call the combination of an obfuscation technique’s attributes its protection profile.

The presentation of each attribute in the taxonomy is kept intentionally succinct, while a discussion of some of the more nuanced aspects, as well as the relation of the attributes to the selected adversary’s capabilities in Table 1 is provided in section 4.

3.2.1 Ways of concealment

The body of a program can be obfuscated in different ways, including:

- **In slices**: The program is treated as a collection of parts, possibly separately or differently obfuscated. In practice, slices are generally program functions.

---

6In our tests, even single stepping instructions overwritten by others was not detected by the debugger which appears to cache code, incorrectly displaying the modified pages.
• As a singleton: The program is treated as a single block to be obfuscated as a whole.

For our purposes, we will consider any part of a program as a program slice.

3.2.2 Encoding

Code can be encoded in many different ways, ranging from simple substitutions to more complex transformations, such as encryption or compression. Methods vary according to the granularity of encoding, space and run-time overheads. Replacing individual instructions with random data for example, while highly granular, requires a reverse mapping of data back to the original instructions. Although a keyed block cipher only requires a key, its use of fixed-sized data blocks reduces its granularity.

3.2.3 Visibility and exposure

We use the term visibility to describe the amount of code obfuscated. It includes complete visibility, when no part of the program is obfuscated, partial visibility, when parts are obfuscated but others remain in clear, and no visibility when a program is fully obfuscated. Intuitively, a visible slice is one which may be read (and executed) whereas a slice with no visibility is unreadable until deobfuscated.

We further classify exposure over the lifetime of a program as complete, or temporary, depending on whether program slices are re-encoded after use or not. A completely exposed program slice is one which is decoded for execution but not subsequently re-encoded. A temporarily exposed program slice is re-encoded after use, minimizing its exposure to an adversary. A program’s exposure type may partially depend on the encoding scheme used.

3.3 Methods

3.3.1 Kanzaki’s method

In order to increase the barrier to understanding software, Kanzaki et al. [27, 26] describe a method of obfuscating program instructions by overwriting them with dummy ones. The idea can be summarized as having parts of software which, at run-time, restore other component’s dummy code with both the original code, and code for restoring the dummy instructions once the original has executed. The proposed system works on Assembly code and consists of:
1. Determining the positions to be protected\(^7\), as well as those of the Hiding and Restoration routines.

2. Replacing original instructions with dummies.

3. Creating and inserting Hiding and Restoration routines.

4. Complication of the routines.

The first step is the most complex one due to the need to determine the positions of the Hiding and Restoration routines. To determine those positions the following rules are given:

- All paths leading to dummy instructions must include a restoration routine.
- No path between a restoration routine and a dummy instruction can include a restoration routine.
- All paths between a hiding routine and a dummy instruction must contain a restoration routine.
- All paths from a dummy instruction to the end of the program must include a hiding routine.

The replacement of instructions is done in a way that preserves the length of the instruction to be replaced. This avoids any need for relocation\(^8\) of the program once the instructions are replaced. After the instructions are hidden they are marked using an Assembly label, and two routines are generated. One will restore the instruction under label to the original, and the other will hide it again.

As a final step, an obfuscation of the transformation routines is performed. Those are modified to refer indirectly to the memory address to be modified, and in addition some modification might be applied. The modifications proposed are to store a different value of the label and add a sequence of instructions (such as addition, subtraction, etc.) that in the end will result to the correct memory address.

\(^7\)This may be random or specified.

\(^8\)Programs have virtual addresses, and jumps within programs refer to those. If program memory is modified by the insertion of some bytes, all references to addresses must be updated.
3.3.2 Madou’s method

Madou et al. propose a rewriting engine [29] that will rewrite functions based on a template before execution. A pseudo-random number generator (PRNG) is used to encode the rewriting templates, and techniques such as opaque variables\(^9\) from [17] are relied on to hide the PRNG seed. In effect the PRNG is used as a stream cipher with the seed as key. The rewriting engine is embedded in the program.

In detail, this method replaces functions of a program with a template containing the function with some instructions replaced by randomly generated data. Uses of the function are updated to reference code calling the rewriting engine with details of the function’s entry point and an “edit script”. This is data in memory containing information required for reconstructing the original code using the template. The rewriting engine uses the edit script and the function’s location to modify the template, recreating the original code.

The authors distinguish between One-Pass mutations and Cluster-Based mutations.

**One-Pass Mutations** With this technique each function in the program has its own template and edit script. The function’s initial code calls the rewriting engine with information about the template and function address. Before executing, the initial function is rewritten and the rewriting code is itself overwritten by the original code.

\(^9\)Opaque variables in this context are variables that have some property that is known to the obfuscator but the deobfuscator cannot deduce. See [17] for more information. It is not known whether variables with this property exist.
Cluster-Based Mutations In this approach functions are grouped according to their code similarity and a common template is generated for functions in the same class. As before, each unique function is replaced with calls to the rewriting engine using the corresponding edit script and entry point. However since the template code is shared the original function is generated in memory and the call to the rewriting engine updated to point to it.

In order to avoid analysis of the edit scripts, they are encrypted using a seeded PRNG. The key is generated at run-time as an opaque variable to prevent its easy recovery.

<table>
<thead>
<tr>
<th>Summary:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation:</td>
</tr>
<tr>
<td>• No public implementation is available.</td>
</tr>
<tr>
<td>Can be used to hide:</td>
</tr>
<tr>
<td>• The program in slices.</td>
</tr>
<tr>
<td>Encoding:</td>
</tr>
<tr>
<td>• Replaces instructions with random data. The decoding instructions (template) are protected using a stream cipher.</td>
</tr>
<tr>
<td>Exposure:</td>
</tr>
<tr>
<td>• Code is partially visible and after decoding is completely exposed.</td>
</tr>
</tbody>
</table>

3.3.3 Shell-code hiding

A defense against unintentional code injection is program input validation, ensuring accepted data conforms to certain rules or formats, such as alphanumeriness etc.

To counter that in [56] the authors explore ways to hide code for ARM CPUs [3] in instructions that are firmly alpha-numeric, in order to masquerade code in a text-looking format that passes any validation check for alphanumeric characters. The authors face the problem of having a limited number – only 13 – of ARM instructions that are within the printable limits of ASCII characters. This limited set of instructions lacks arithmetic operations and any branch instruction, hence causing problems in writing any non-trivial piece of code, such as a worm or shell-code\(^\text{10}\). However by using self-modifying code the instruction set is increased with the set of bytes that can be written as an exclusive or (XOR) of a printable character with another one, since instruction XOR is

\(^{10}\)Shell-code is code that is intended to be injected to running applications on a machine, with the goal of executing a shell.
within the printable character set. An illustration of the idea on how to generate null bytes required for the following instruction\textsuperscript{11} using ARM assembly, is shown below:

\begin{verbatim}
mov r0, #0 ; byte representation: 0xe3 0xa0 0x00 0x00
\end{verbatim}

To construct this instruction, the authors used the following code, adapted from \cite{56}, simplified and added comments for clarity:

\begin{verbatim}
; Load in register 1 the 16 bits that are in label .Linstr + 2 bytes.
; r1=0x0180
ldrh  r1, .Linstr+2

; XOR register 1 with the value 0x0180
; and place output in register 1.
; r1=0x0000
eor  r1, r1, #384

; store in label .Linstr the value of register 1
strh  r1, .Linstr
.Linstr:
.byte 0xe3, 0xa0
.byte 0x80, 0x01 ; new value = 0x00, 0x00
\end{verbatim}

The latter code generates the new instruction in place of the old instruction in line 9 and executes it. All the instructions used in the code snippet above, such as \texttt{LDRH}, \texttt{EOR} and \texttt{STRH} are within the printable character set. This technique is used by the authors to generate code that is entirely textual.

Summary:
Implementation:
- No public implementation is available.

Can be used to hide:
- The entire program as a singleton.

Encoding:
- Replaces instructions with ones that have a corresponding code in the printable character set, or ones that can be recovered using instructions from the printable set.

Exposure:
- Code is partially visible and after decoding becomes completely exposed.

\textsuperscript{11}For simplicity in this example it is assumed that only null bytes cannot be used.
3.3.4 Methods used in Burneye

Written by team-teso [46], Burneye\textsuperscript{12} is a tool to obfuscate ELF binaries [47], optionally password protecting them and imprinting them such that they only run on specifically fingerprinted hosts. We will restrict our description on the aspects of the tool that hide code and involve self-modifying code. The tool provides alternative options for protecting code, that can be combined.

- Simple scrambling: A Galois linear feedback shift register is used to encipher the original code.
- Encryption with a secret: The contents of the program are encrypted using the RC4 [36] cipher with a user-supplied password as keying material. The actual key derivation mechanism uses salting to discourage rainbow table attacks.
- Fingerprinting: Various characteristics of the system on which the binary is to be executed are collected and the code is encrypted with this information as key.

The first option, encrypts the executable with a simple cipher and no secret, and the transformation is reversed at execution time. The second and third option use a different cipher, with a secret part to encrypt the binary. The secret part is required for the execution phase, and in second option is given by a password prompt, while at third option the running system’s characteristics are used for generating a key. Although there are cases where the last two options will increase the security level, in our model they provide no extra security since we assume knowledge of the secret information by the adversary.

A similar to the first option in operation method is described in [50, Section 12.17], by Viega et al. That method uses the RC4 cipher to encrypt an executable program and decrypts it at run-time.

\textsuperscript{12}The rootkit used during the infamous compromise of three Debian servers in 2003 was obfuscated using Burneye.
Summary:
Implementation:
- A public implementation is available in [46].
Can be used to hide:
- The entire program as a singleton.
Encoding:
- Uses a linear feedback shift register (LFSR) based or RC4 stream cipher to encrypt code.
Exposure:
- Code is non-visible and after decoding becomes completely exposed.

3.3.5 UPX compression

In [34], Oberhummer and Lazlo describe UPX, a multi-platform executable file compressor. UPX stands for the Ultimate Packer for eXecutables and is mainly used for reducing the size of executables in restricted systems, but is also used by viruses and other executables to obfuscate their code. UPX can use a variety of compression algorithms, such as UCL [33], LZMA\textsuperscript{13} or the proprietary NRV [33] algorithms. The default and fastest decompressor being NRV.

Prior to compression, UPX processes code (called filtering) to increase the compression level. That processing is architecture specific. On x86, for example, relative addresses in JMP and CALL instructions are replaced by absolute, causing a larger byte match for the compressor. Following this, the UPX packer tool compresses the target executable using the specified compression algorithm, and prepends the unpacker to the executable.

At run-time the unpacker takes control, decompresses the original program to memory, reverses the filtering process, and transfers control to it.

\textsuperscript{13}LempelZivMarkov chain algorithm.
Summary:
Implementation:
- A public implementation is available in [34].

Can be used to hide:
- The entire program as a singleton.

Encoding:
- Uses a compression algorithm, i.e., LZMA, UCL or NRV, to obfuscate code.

Exposure:
- Code is non-visible and after decoding becomes completely exposed.

3.3.6 Methods used in Shiva

Shiva is an GNU/Linux ELF executable [47] encryptor written by Shawn Clowes and Neil Mehta first presented at CanSecWest in 2003 [15] to forward the state of the art in binary obfuscation on UNIX platforms. While the original presentation gives a high-level overview of Shiva, details of its workings were first publicly presented by Chris Eagle at BlackHat 2003 [19].

Shiva encrypts a program in blocks using the Tiny Encryption Algorithm (TEA) [55] and decrypts them on-demand to ensure the whole binary is never fully exposed in memory for dumping to disk. In addition, it allows the TEA encrypted blocks to be further encrypted using AES [18] and decrypted with a runtime user-supplied password. Several other anti-debugging techniques are used.

The code of the original program is being replaced by encoded data consisting of three blocks:
- Block 1, the Shiva code.
- Block 2, data used by Shiva.
- Block 3, The protected code in TEA encrypted blocks (possibly wrapped in a further AES encryption if password protected).

The outer encoding is done using simple instructions such as XOR and ADD operations. At program start-up the binary is decoded producing the three blocks in memory. After this step the Shiva code, takes control and starts decryption on demand.
Shiva encrypts not only the instructions used by the program’s code, but also encodes all metadata available through the ELF file information. It uses anti-debugging techniques such as jumping into the middle of instructions, polymorphic code generation, and also implements its own runtime environment with memory maps and on-demand paging. Unused memory is filled with the INT 3 software interrupt\(^{14}\) and when called, will trigger code that will do decryption and mapping of the missing page for execution to resume.

The memory mapping is implemented using encrypted blocks containing meta-information about the original program’s memory layout as normally given by the ELF’s program header table. These meta-blocks are encrypted with different keys, each of which is reconstructed dynamically, used, then cleared. The key-reconstruction functions are themselves differently obfuscated for each binary.

Finally, Shiva replaces some instructions in the original program with the INT 3 software interrupt storing the original operands and emulating the original instruction via the ptrace interface during the interrupt. This robs reverse engineers of the certainty of having captured all instructions in a code block.

<table>
<thead>
<tr>
<th>Summary:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation:</td>
</tr>
<tr>
<td>• No public implementation is available.</td>
</tr>
<tr>
<td>Can be used to hide:</td>
</tr>
<tr>
<td>• The entire program as a singleton.</td>
</tr>
<tr>
<td>Encoding:</td>
</tr>
<tr>
<td>• Relies on both simple encodings (XOR and ADD) and cryptographically strong methods, such as block ciphers TEA and AES.</td>
</tr>
<tr>
<td>Exposure:</td>
</tr>
<tr>
<td>• Code is non-visible and is only temporarily exposed.</td>
</tr>
</tbody>
</table>

### 3.3.7 Methods used in Cryptexec

Cryptexec is an x86 code obfuscator, published by Vrba [51] in phrack 63. It encrypts a number of program functions using the CAST cipher [1] in ECB\(^{15}\) mode with a given key. Similarly to Shiva in Section 3.3.6 it tries to prevent expanding the executable code in memory. For that during decoding it operates like a virtual machine decrypting and executing the encrypted code.

---

\(^{14}\)The INT 3 interrupt in a x86 GNU/Linux system causes the executed program to receive a SIGTRAP signal. That signal is handled by appropriate program code.

\(^{15}\)Electronic Codebook Mode. The simplest mode to use a block cipher.
instruction by instruction. It maintains its own private stack space from which it allocates two execution contexts, one for the encrypted code, the other for the decoder. Decoding is performed as follows:

- The decoder fetches the next instruction from the encrypted code.
- The data is decrypted.
- The plaintext is disassembled, yielding the original instruction.
- The instruction is executed using the encrypted code context.

Cryptexec provides an API to access encrypted functions thus making it easy to provide access to the encrypted parts of the executable. In addition it ensures that during decoding no more than two single instructions are decrypted in memory at any time.

### Summary:

**Implementation:**
- A public implementation is available in [52].

**Can be used to hide:**
- The program in slices.
- The entire program as a singleton.

**Encoding:**
- Relies on the cryptographic block cipher CAST in ECB mode.

**Exposure:**
- Code is non-visible and is only temporarily exposed.

### 3.3.8 CSPIM

CSPIM is a modification of the Cryptexec method published by Vrba in [54]. Instead of obfuscating the original program instructions, the code to be obfuscated is compiled to the MIPS I instruction set [45] and encrypted using the RC5 cipher in ECB mode using a 32-bit block size. The MIPS I instruction set is selected due to its simplicity, thus requiring a simpler emulator, and the small block size leads to a fine-tuned decryption granularity. During run-time a virtual machine decrypts and emulates the encrypted code instruction by instruction. The operation is very similar to Cryptexec, but with an added emulation layer, that allows for portable obfuscated code that will be interpreted by a platform depended virtual machine.
Summary:
Implementation:
• A public implementation is available in [53].
Can be used to hide:
• The program in slices.
• The entire program as a singleton.
Encoding:
• Relies on the cryptographic block cipher RC5 with a 32-bit block size in ECB mode.
Exposure:
• Code is non-visible and is only temporarily exposed.

3.3.9 Aucsmith’s tamper resistant software

Aucsmith discusses in [4] the issue of software tamper resistance and proposes an implementation to achieve it. We give a quick overview of the method and how self-modifying software is used within this framework.

The threat model in this application is explained using three categories:

1. The malicious threat originates outside the system. The perpetrator is using the network to “get in” the system.

2. The perpetrator has access of software running on the system. The attacker is bounded by the operating system. In this category the perpetrator is someone who had at one time access to the system.

3. The perpetrator has complete access to the system. He is the “owner” of the system. This is further split into:

   (a) No specialized analysis tools, such as debuggers and disassemblers are available.

   (b) Software tools such as debuggers and disassemblers are involved.

   (c) Hardware analysis tools, such as processor emulators are available.

The categories this framework targets against are 2 and 3. The 3rd category is claimed to be handled up to attacks involving hardware analysis tools. The basic principles this implementation is based on are

• To disperse secrets in time and space.
• Obfuscation of interleaved operations. This is having the full functionality split and executed by successive iterations or rounds of the code. To prevent discovery of the interleaved components the usage of self-modifying code is suggested.

• Have an installation unique code such as having each instance of software unique elements.

• Interlocking trust. The correct performance of components should depend on the correct performance of others.

The mixture of those principles is suggested to guarantee tamper resistance.

The architecture is composed of “Integrity Verification Kernels” (IVKs), that follow the above principles and an “Interlocking Trust Mechanism”. The former are “protected” code segments, that ensure the integrity of the process. The Interlocking Trust Mechanism is a mechanism to allow IVKs verifying other IVKs. Without getting into the details of the mechanisms we will describe their usage of self-modifying software.

The IVKs are described to be encrypted and self decrypt their body during execution. The cipher used should ensure that as decryption of a code section occurs, other code sections should be encrypted. It is also suggested to re-use memory locations for different operations.

This is the first study that advices the usage of self-modifying code to protect software. There are no formal results such as a metric on how security can be evaluated based on the threat level they protect against, but nevertheless it is an important source of ideas for software protection.

Summary:
Implementation:
• No public implementation is available.

Can be used to hide:
• The entire program as a singleton.

Encoding:
• Relies on cryptographically strong methods.

Exposure:
• Code is non-visible and is only temporarily exposed.
3.3.10 Cappaert’s method

Cappaert et al. in [12, 11] proposes ways to avoid static analysis of executable code, focused on code encryption. The authors distinguish between bulk decryption, where a loader does the decryption of all the executable code at run-time, and on-demand decryption mode. The main focus of the proposal is the on-demand decryption mode, where functions are decrypted at run-time, and re-encrypted them after usage.

In this scheme “crypto guards” are introduced. Those are routines that will decrypt a function’s code based on the integrity of other function’s code. A cryptographic hash is used to obtain the key from another function’s code. This cryptographic hash will provide from arbitrary length data a fixed size byte array that can be used as a key with a symmetric cipher. Each function is protected with a key that is derived from the code of the caller, and re-encrypted when the function returns.

The authors of this scheme faced the problem of having multiple callers of a function. In that case they suggest on deriving a key from all the callers. However in this scheme the real caller of the encrypted function will be in clear whilst the other callers will be probably encrypted. This has the disadvantage of having to decrypt all the possible callers, in order to derive the decryption key\(^{16}\).

In addition to limiting the available code to a debugger, this approach offers protection against modification of the binary code. Any change in a function will cause an incorrect decryption of some other function.

<table>
<thead>
<tr>
<th>Summary:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation:</td>
</tr>
<tr>
<td>- No public implementation is available.</td>
</tr>
<tr>
<td>Can be used to hide:</td>
</tr>
<tr>
<td>- The program in slices.</td>
</tr>
<tr>
<td>Encoding:</td>
</tr>
<tr>
<td>- Relies on cryptographically strong methods.</td>
</tr>
<tr>
<td>Exposure:</td>
</tr>
<tr>
<td>- Code is non-visible and is only temporarily exposed.</td>
</tr>
</tbody>
</table>

\(^{16}\)The “Surreptitious Software” book [16] described a modified version of the Cappaert el al. algorithm called the \text{OB}F\text{CKSP}. This tries to solve the multiple callers problem by moving the decryption of a function \(F\) to the callers of the caller. The code of all the direct callers of \(F\) is being used as decryption key of \(F\).
4 Discussion

In the following sections we elaborate on the use of self-modifying code for obfuscation in the context of the presented taxonomy and the structure of the methods surveyed, summarized in Table 2. We discuss the concealment, encoding methods, visibility and exposure, and their effect on the defense against different adversaries.

4.1 Ways of concealment

The method of concealment, whether a singleton or in slices, has no direct effect on the overall quality of obfuscation. A weak relation exists however, between the method and the degree of exposure. Methods that do not use a virtual machine to emulate the obfuscated code, such as Kanzaki’s, Aucsmith’s and Cappaert’s [27, 12, 4], operate on code slices in order to offer temporary exposure. However virtual machine based methods such as Cryptexec, CSPIM and Shiva [51, 54, 15], demonstrate that singleton encoding methods can be combined with temporary exposure, by interpreting arbitrary (Turing-complete) programs, instruction by instruction.

Hence we see that even if the ways of concealment have no impact on the obfuscation, some techniques might impose a particular concealment method due to operational requirements.

4.2 Encoding

The encoding method perceives the notion of code obfuscation, as a complex transformation of code to be stored in a binary file. To consider a method “secure”, devising a decoding method should be more costly than mounting an attack of a higher adversary category. For example, a decoder using a constant value XOR, after the method is realized may be cheaper to attack creating a specialized tool which recovers the clear code for disassembly than mounting an attack with a category III debugger (see Table 1 on page ). Encoding methods, if used in isolation, by definition protect against the adversary of category I, i.e., the static disassembler.

4.2.1 Simple methods

The simple encoding methods, such as instruction replacement and XOR or addition modulo integers, are invertible transformations used to obfuscate code.
They are relatively understandable and used to conceal small amounts of code, generally supplementing other protection methods.

4.2.2 Complex methods

Complex encodings such as encryption or code compression can protect the code completely from static analysis tools, such as disassemblers of category I. Almost all techniques using complex methods employ efficient algorithms such as the RC4 [36] stream cipher, block ciphers in ECB\(^{17}\) mode, or compression methods with a fast decompressor. Methods using cryptography favor speed of decryption over cipher strength as testified by the prevalent use of “weak” modes of operation such as ECB. The cryptographic algorithms used need only be sufficiently strong to ensure breaking the cipher is harder than reverse engineering the cipher and discovering the key. The algorithm’s speed is accorded far more importance. The inclusion of instructions for high-performance encryption in newer CPUs [23] might render methods previously deemed prohibitively expensive [11] feasible.

4.3 Visibility

Visibility is a measure of method’s protection against static analysis. We distinguished two types of visibility, partial and none. In the former, parts of the code are obfuscated leaving others in the clear, while the latter type ensures all code is obfuscated.

Methods offering partial visibility leave a percentage of code in the clear. Consequently, relevant information may yet be extracted, even by a category I adversary with a disassembler. A high-level instruction such as the `printf` libc function call may contain a dozen or more assembly instructions and obfuscating a few may not sufficiently hide the code’s functionality. Replacing 10% of the instructions at random with dummies gives an adversary a 90% probability of finding a valid one at each check. By increasing the percentage of replaced instructions however, schemes like Kanzaki’s and Madou’s require duplication of large amounts of code.

Thus methods offering partial visibility can only defend probabilistically against an adversary seeking to reverse engineer the obfuscated code. Methods offering

\(^{17}\)ECB exposes some structure of the plaintext as identical plaintext blocks yield the same encrypted blocks. In practice this has not been problematic as the decoder remains an easier target of attack [19].
no visibility obfuscate the entire program code. While the effectiveness of obfuscation transformations varies, as abstract methods they offer full protection against static code analysis.

Visibility whether partial or none applies to category I adversary, by definition.

4.4 Exposure

Exposure is a measure of a method’s protection during program execution. Although all surveyed techniques obfuscate a program’s code, some deobfuscate it in its entirety for execution, exposing it completely, while others do so only temporarily. We discuss differences between the two approaches and the impact on their levels of protection, assuming the usage of an encoding method.

Table 2: A summary of the taxonomy of self-modifying code for code obfuscation. The methods marked with ♣ have a public implementation available. The check mark(✔️) or commenting text is used when a technique suits the corresponding attribute.

<table>
<thead>
<tr>
<th>Method</th>
<th>Concealment</th>
<th>Encoding</th>
<th>Visibility</th>
<th>Exposure</th>
</tr>
</thead>
<tbody>
<tr>
<td>In slices</td>
<td>Singleton</td>
<td>Simple</td>
<td>Complex</td>
<td>None</td>
</tr>
<tr>
<td>Kanzaki</td>
<td>✔️</td>
<td>Code replaced by random data</td>
<td>Depends on the percentage of replaced instructions</td>
<td>Code decoded and re-encoded after execution</td>
</tr>
<tr>
<td>Madou</td>
<td>✔️</td>
<td>Code replaced by random data, decoding using PRNG</td>
<td>Depends on the percentage of replaced instructions</td>
<td>✔️</td>
</tr>
</tbody>
</table>

86
Table 2: (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Concealment</th>
<th>Encoding</th>
<th>Visibility</th>
<th>Exposure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>In slices</td>
<td>Singleton</td>
<td>Complex</td>
<td>None</td>
</tr>
<tr>
<td>Shell-code hiding</td>
<td></td>
<td></td>
<td></td>
<td>Code that generates new code</td>
</tr>
<tr>
<td>Burneye ♣</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LFSR or RC4 encryption algorithms</td>
<td>✓</td>
</tr>
<tr>
<td>UPX ♣</td>
<td>✓</td>
<td></td>
<td></td>
<td>LZMA, UCL or NRV compression algorithms</td>
</tr>
<tr>
<td>Shiva</td>
<td>✓</td>
<td></td>
<td></td>
<td>TEA or AES encryption algorithms</td>
</tr>
</tbody>
</table>
Table 2: (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Concealment</th>
<th>Encoding</th>
<th>Visibility</th>
<th>Exposure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cryptexe</td>
<td>✓</td>
<td>✓</td>
<td>CAST</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>encryption algorithm</td>
<td>✓</td>
</tr>
<tr>
<td>CSPIM</td>
<td>✓</td>
<td>✓</td>
<td>RC5</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>encryption algorithm</td>
<td>✓</td>
</tr>
<tr>
<td>Aucsmith</td>
<td>✓</td>
<td></td>
<td>Any</td>
<td>Code decoded and re-encoded after execution</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>encryption algorithm</td>
<td>✓</td>
</tr>
<tr>
<td>Cappaert</td>
<td>✓</td>
<td></td>
<td>Any</td>
<td>Code decoded and re-encoded after execution</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>encryption algorithm</td>
<td>✓</td>
</tr>
</tbody>
</table>
4.4.1 Complete exposure

Several methods only protect code against static attacks, before execution. If concealed as a singleton, program code is decoded in its entirety and stored in memory, available to an adversary. If, on the other hand, it is concealed in slices, only those decoded for execution are exposed. As the program continues to run, more slices are exposed until it is available in its entirety, as code is executed\(^\text{18}\).

Such methods prevent disassemblers from displaying any useful information and hamper debuggers with no support for self-modifying code. In the days of anti-virus software statically scanning binaries for known-signatures, viruses relied on such tricks to escape detection [43].

Completely exposed programs can be made available in clear to an adversary, even of category II, using a memory dump\(^\text{19}\). If the memory dump is taken after decoding, the full program code will be available to the adversary. For the reasons above, we note that the protection level of methods that completely expose code, is against adversaries with tools of category I.

4.4.2 Temporary exposure

Methods that only allow temporary exposure such as Cryptexec, Kanzaki, Cappaert’s etc., take a piecemeal approach to encoding programs. Either instructions or whole functions are recovered at runtime, on-demand, and re-encoded immediately after use, ensuring that at no single moment of execution is the entire code available in the clear. In addition to preventing disassemblers from displaying any useful information, this method performs constant self-modification, for as long as protected functions are called. Moreover, it prevents adversaries at any point during execution from obtaining an overview of the whole program. Only the executing function or part and required data are available in clear.

For this reason we claim that the protection of this method is against adversaries with tools of category II or lower. Category III adversaries are also discouraged by depriving them from a direct global overview of the code.

\(^{18}\)Non-executed code is obviously not decoded.

\(^{19}\)Operating systems typically provide ways to take a memory dump of a process for debugging purposes. The GNU/Linux operating system, for example, saves a program’s registers and memory in a core file –suitable for examination with a debugger– upon receiving a SIGSEGV signal.
Table 3: The taxonomy attributes and their effects on protection level against adversaries. The adversaries’ categories correspond to Table 1 on page and the check mark(✓) indicates protection against this adversary.

<table>
<thead>
<tr>
<th>Adversary</th>
<th>Description</th>
<th>Encoding</th>
<th>Visibility</th>
<th>Exposure</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Disassembler</td>
<td>Reverse engineering is easy</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>II</td>
<td>Debugger with no ability to handle self-modifying code</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>III</td>
<td>Debugger that handles self-modifying code</td>
<td></td>
<td>Prevents global overview</td>
<td></td>
</tr>
<tr>
<td>IV</td>
<td>Specialized tools</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.5 Relation to the adversaries

Table 3 summarizes the effects of our taxonomy’s attributes on the different adversaries defined in Section 3.1.1. The included remarks correspond to the discussions in the previous sections. The concealment criterion is omitted as it has no direct bearing on the overall security level, as discussed in Section 4.1.

5 Conclusion

We have presented a taxonomy of self-modifying code for obfuscation and a model characterizing different levels of attackers in terms of the types of tools required to mount each attack. We used the model and taxonomy towards a primary goal of establishing a terminology and framework within which more precise discussions of self-modification for obfuscation may be pursued. Further secondary results include better motivating an obfuscator’s choice of com-
ponents and their use in a protection scheme, as well as highlighting under-developed areas of analysis for the simultaneous exploitation by obfuscators, and implementation by attackers. Taken together, the combination of attributes listed in the taxonomy, namely concealment, encoding, visibility, and exposure, constitute an obfuscation technique’s protection profile. We show, perhaps unsurprisingly, that a protection profile combining complex encoding, no visibility, and temporary code exposure provides the highest protection against analysis from the defined adversaries.

As table 2 indicates, many of the obfuscation techniques employ robust protection profiles, requiring specialized tools to attack them. However, the fact that most have been successfully attacked highlights the ultimate futility of obfuscation. In a cat and mouse game, programs are obfuscated involving complex methods such as self-modifying code, to hide pieces of code for a period of time before they become irrelevant, and attacks with varying attack tools are performed to uncover the obfuscated code. Ultimately specialized tools are implemented to uncover obfuscated programs, but the custom nature of the tools may mean a change in the obfuscation’s protection profile necessitates a rewrite of the attacking tool. This can be seen at the attack on Shiva [19] which was subsequently modified by its authors to foil the custom deobfuscator.

We see our findings do not contradict the impossibility result of Barak et al. [6], nor the semantic equivalence of self modified software with non-self modified software [7]. We base our analysis on a modelling of the imperfection of today’s real-world tools available for reverse-engineering. Hence a subjective factor in the modelling of “difficulty” for reverse-engineering, is introduced, in a way that this study depends on the current-day technology. This dependence however provides the ability to assess methods that are used in practice, despite the negative theoretical results.

For that we have seen that obfuscation raises the bar of protection against our modelled adversaries of categories I and II. An adversary of category I, that is an adversary with static-analysis tools, that inherently has no ability to follow code transformations, and cannot reverse engineer programs encoded using a complex encoding method without devising a specialized tool that reverses the transformation. Likewise an adversary of category II, i.e., dynamic-analysis tools with restrictions that due to bugs or limitations fail to follow code transformations, also require specialized tools to evade methods that offer temporal exposure. Those specialized tools by our definition would come at the expense of requiring an adversary of a higher category, being category III or IV.

Thus we have made apparent that current-day protection techniques are depending on the scarce availability of tools that resemble an “idealized” debugger and the difficulty of devising a custom decoding mechanism. The wider
availability of category III tools that pose no issues in following self-modifying
code would certainly render many of the discussed techniques irrelevant and
possibly drive the technology to protection methods that exploit different tool
limitations and restrictions.

6 Acknowledgments

The authors would like to thank Andreas Pashalidis, and the anonymous ref-
erees for their comments which improved the manuscript. This work was
supported in part by the Institute for the Promotion of Innovation through
Science and Technology in Flanders (IWT Vlaanderen) SBO project, the Re-
search Council K.U.Leuven: GOA TENSE (GOA/11/007), and by the IAP
Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy).

References

[1] C. Adams. The CAST-128 Encryption Algorithm. RFC 2144 (Informa-
tional), May 1997.
1996.
ARM, 2009.
ings of the First International Workshop on Information Hiding, pages
June 2003.
and K. Yang. On the (im)possibility of obfuscating programs. In Lecture
self-modifying programs. In 7th IEEE International Conference on
Software Engineering and Formal Methods - SEFM 2009, Hanoi Viet Nam,
11 2009.
[8] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good in-
structions go bad: Generalizing return-oriented programming to RISC. In


5.2 A Linux kernel cryptographic framework

Publication data


An extended version of the paper is available in Appendix A.

Contributions

Principal author.
A Linux kernel cryptographic framework: Decoupling cryptographic keys from applications

Nikos Mavrogiannopoulos* Miloslav Trmač† Bart Preneel*

March 2012

Abstract

This paper describes a cryptographic service framework for the Linux kernel. The framework enables user-space applications to perform operations with cryptographic keys, while at the same time ensuring that applications cannot directly access or extract the keys from storage. The framework makes use of the higher privilege levels of the operating system in order to provide this isolation. The paper discusses the relevant security requirements and expectations, and presents the design of the framework. A comparison with alternative designs is also provided.

1 Introduction

There are many types of attacks against software applications, such as code injection, control flow modification [23, 1], etc., attributed to software vulnerabilities. Their elimination has been proven to be a Sisyphean task and today, software vulnerabilities are mitigated using a combination of defensive programming [21] and retroactive patching. As a result, there is a risk of an attacker ‘taking control’ of an application exposed to a public network (such as a web server), maybe for some limited time until the vulnerability is known and the system is updated.

The cost of such compromise may be significant for the stakeholders. One example are on-line shops today that typically use TLS [10] to authenticate to the client and setup a secure communications channel. The secure channel is setup using a server certificate and a corresponding private key. Because in typical web server applications the private key resides in the same address

* Katholieke Universiteit Leuven, ESAT/SCD/COSIC.
† Red Hat.

98
space\textsuperscript{1} as the server, its compromise is equivalent to a compromise of the private key. Thus, an adversary with the ability to compromise the web server could retrieve the private key and obtain the ability to masquerade as the original shop to its clients. This would be possible even after such a compromise is no longer possible (e.g., caused by an automatic software upgrade, or by installing a newer operating system). This might be catastrophic for the reputation of the shop.

A defense mechanism for the above attack is decoupling cryptographic keys from the software that uses them. This is typically achieved using hardware security modules or smart cards \cite{22}, that store the cryptographic keys in a medium that allows operations without exposing them. That approach effectively raises the protection of cryptographic keys from the cost of an application compromise, to the cost of compromising a hardware module. However, there is a possibility to increase the protection levels without involving new hardware. Operating systems (OS) are typically designed with security in mind \cite{14}, and provide protection mechanisms such as segregation of the OS with user applications and isolation of independent processes. These mechanisms can be used to decouple cryptographic keys from applications, providing an additional layer of security to a system where hardware security modules are not practical or are too costly (e.g. low-cost servers, or mobile phones).

Today, frameworks offering isolation of cryptographic keys from applications exist \cite{19, 18}, based on the separation of processes and the access control mechanisms offered by the OS. In this paper we present a cryptographic framework that decouples cryptographic keys from applications based on the separation between the OS and its processes. Apart from the description, we discuss the difficulties this approach imposes as well its advantages and disadvantages comparing to an alternative design. Moreover, in order to ensure the safety of the stored keys, we base the design of the framework on principles that delegate the protection of stored keys to the strength of the supported algorithms.

The next section provides a background on existing work. In Section 3 we formalize the threats on current operating systems and set the basic requirements for our framework. In Section 4 we describe our framework on a high level and present its architecture. A security analysis is provided in Section 5. In Section 6 a performance analysis is provided and the following section discusses the differences between our approach and alternatives. Section 8 concludes.

\textsuperscript{1}By the term ‘address space’ we mean memory accessible in a process. Each process has its own address space.
2 Background and related work

Hardware security modules (HSM) and smart cards are the main medium used today for decoupling cryptographic keys from software and are often required by governmental standards [20, 13]. Their operation is based on a logical and physical separation of cryptographic keys from the applications using them. For example a typical PKCS #11\(^2\) smart card or a TPM module [22] allow for cryptographic operations (e.g., RSA decryption) using stored keys, without exposing them.

Because the deployment of hardware security modules is not always feasible, a few software-only approaches to separation already exist, based on existing isolation mechanisms\(^3\). The CNG API [19] is a redesign of the old Microsoft CryptoAPI, that allows the decoupling of cryptographic keys from applications. It uses a special ‘key isolation process’ executing with the permissions of a system user which provides services, such as cryptographic key storage and operations to other applications. A similar approach is used by another framework, the lite security module [18] (LSM). It provides a PKCS #11 API to applications in order to access a central daemon. The daemon executes under a system user and provides key storage and operations to the other applications. Both frameworks allow persistent keys that cannot be exported.

The protection level of security modules, however, is not easy to assess. That is because the available operations should be selected in way that no combination of them is sufficient to obtain or reconstruct the stored keys. This requirement, although simple to formulate it has been shown in practice that it is hard to achieve in real-world security modules [5, 7]. The goal of a formal proof of safety in security modules is on-going work, and currently we only have indications of security in restricted designs [9, 6] that support few operations. Hence in our design we make sure that the cryptographic operations implemented and made available are not sufficient to extract the stored keys. We will use a minimal set of operations carefully chosen for attack resistance properties.

The term security module is used in this paper to describe any framework or hardware that provides operations on cryptographic keys without exposing them.

\(^2\)A cryptographic API that provides logical separation between keys and operations. It is mainly used to access operations on hardware security modules and smart cards.

\(^3\)Software that provides emulation to a security module but provides no isolation between cryptographic keys and the application was not considered in our study.
3 Model and Requirements

Our model of an OS is based on the Linux kernel design. In that, software is comprised of an OS running on a different protection ring\(^4\) than normal applications. This requirement ensures that normal applications cannot access data available to the OS.

A user called administrator, is able to directly control the OS executing in the highest protection ring and there is at least one user that operates solely on a lower protection ring. The applications running on a system are assumed to be running with the privileges of a normal user. We use the common terms user-space to reference the lower privilege rings and kernel-space for the higher privilege ring the OS operates.

3.1 Threats

In this model of an OS, applications operate in user-space and utilize cryptographic keys. We assume that application vulnerabilities might exist for some time until the system is patched. Those give adversaries a time window to execute malicious code at the same ring as the applications running in the system. Thus our main threats are:

- An adversary that has taken control for limited time of a user-space application and is trying to recover the protected keys; the adversary might be collaborating with another legitimate user;

- An adversary that is a legitimate system user and wants to extract raw keys; in contrast to the former adversary, this one is not restricted by time.

We want to protect keys in the presence of such adversaries. It should be noted that in this model, there is no distinction between operating system access and administrative access. This is due to the design of the modelled system.

3.2 Requirements

A list of high level requirements for the framework concerning the cryptographic keys to be protected, follows.

1. Users should be allowed to perform cryptographic operations using the keys, but the key material should not be extractable;

\(^4\)We use the term protection ring to assume a strict separation of data and code between different access control levels.
Table 1: Features and supported algorithms.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Supported Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key generation</td>
<td>Symmetric keys and Private/public key pairs</td>
</tr>
<tr>
<td>Encryption and Decryption</td>
<td>RSA PKCS #1 v1.5, RSA PKCS #1 OAEP, AES-CBC, AES-CTR, AES-ECB, 3DES-CBC, 3DES-CTR, 3DES-ECB, CAMELLIA-CBC, CAMELLIA-CTR, CAMELLIA-ECB, RC4</td>
</tr>
<tr>
<td>Signing and verification</td>
<td>DSA, RSA PKCS #1 v1.5, RSA PKCS #1 PSS, HMAC-MD5, HMAC-SHA1, HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, HMAC-SHA-512, MD5, SHA1, SHA-224, SHA-256, SHA-384, SHA-512</td>
</tr>
<tr>
<td>Key wrapping</td>
<td>AES-RFC3394, AES-RFC5649</td>
</tr>
<tr>
<td>Key derivation</td>
<td>Diffie-Hellman as in PKCS #3</td>
</tr>
</tbody>
</table>

2. Keys should be protected from all known side-channel attacks;
3. All cryptographic material such as keys must not be accessible from applications that use them, or from any other application, either directly or through a process memory dump;
4. Keys should be transferable to other systems in a secure way.

Our requirements for the features on the implementation were set to include the required functionality for TLS 1.2 [10] protocol implementation, and for its interface to be translatable to the PKCS #11 API. The operations and algorithms implemented are summarized in Table 1.

4 Architecture

Our framework, named NCR, is implemented as a self-contained Linux kernel module that provides access to cryptographic operations. It supports a synchronous API for communication with the user-space based on the `ioctl()` system call. We emulate the hardware isolation of cryptographic operations from actual keys using the ring separation between the OS and user-space as shown in Figure 1. Our goal is to completely decouple cryptography from the processes that use it.

`ioctl()` is a primitive communication interface between kernel and user-space. It typically works by allowing the caller to specify the operation and a structure with the operation input data.
4.1 Usage of keys

Applications should be allowed to perform operations using a set of keys in a way that only the outcome will be available to them. For this reason keys should be accessible and referenced as abstract objects, without exposing their contents. However, not all keys should be treated the same. Security protocols, typically operate using long term keys, used for a key exchange that outputs short term keys valid only for the current session (session keys). Typical cryptographic protocols such as the Internet security protocols [10, 15], satisfy the known session keys property of [4], which ensures that the knowledge of past session keys does not affect protocol security. Thus, in order to avoid protocol specific key derivation algorithms being present in the framework, we explicitly distinguish keys as ‘long term’ and ‘session’ keys. This distinction strikes a balance between pragmatism and need for key protection. Operations on long term keys will be protected by the proposed framework, while session keys will be generated and used in user-space. The distinction is not arbitrary. Session keys are valid for limited time and protect data in the application address space. Hence, if one is able to extract those keys during their validity period, it is not unrealistic to claim that the data they protect could also be extracted.

4.2 Framework components

Our design consists of two basic components, one is the ‘cryptographic backend’ and the NCR component that provides operations to user-space applications. Each component is described in detail in the following paragraphs, and a high level interplay of the components is shown in Figure 2.
4.2.1 Cryptographic backend component

This component of the design is based partially on the existing Linux kernel asynchronous symmetric cipher and hash API, and partially on libtomcrypt and libtommath. The latter libraries were used to implement the asymmetric cryptography backend that is not part of the Linux kernel cryptographic API. The libraries were modified for the Linux kernel and were used for the implementation of RSA, DSA and Diffie-Hellman algorithms. The Linux random number generator was used to provide randomness.

The main purpose of the cryptographic backend component is to provide the main NCR component with a set of algorithms to support the required operations as in Table 1.

**Figure 2:** Component view of the framework. The entry and exit points for cryptographic keys are marked with double arrows.
4.2.2 NCR component

The NCR component contains the main implementation of the provided features. It operates on key objects that are only available within the component. The key objects are marked on generation or import with policy flags that indicate exportable status, wrappable status, or wrapping status. This approach allows for key objects of a gradual security level. The highest being the keys that have no policy flags set and thus cannot be exported nor wrapped with another key, and the lowest being keys that are marked as exportable. We refer to keys with no policy flags as safe keys.

To enforce the security flags throughout the component, key objects can enter and leave the component through specific points only, the ‘Key storage’ and possibly the ‘Key wrapping’ subsystem as shown in Figure 2. Keys marked as exportable can also use the ‘Key export’ subsystem. The security flags are checked and enforced by the above subsystems, which also ensure that keys cannot be decoupled from their flags.

Key objects A mandatory access control mechanism is used to restrict access to cryptographic keys. Each key object on creation is given an owner and a group and the access policy is being applied by the subsystem that is performing the operations, the abstract session interface. The usage of a mandatory access control instead of the familiar in Linux discretionary has the advantage of preventing the transfer of keys from users under some outsider’s control to other unauthorized users of the system.

Abstract session interface This interface provides a generic way for the diverse supported algorithms to be used. The algorithms operate on input data and provide encryption, decryption, signature generation, signature verification as well as message digest operations. These operations are mapped to three actions, or ioctl() calls, being NCRIO_SESSION_INIT, NCRIO_SESSION_UPDATE and NCRIO_SESSION_FINAL. The NCRIO_SESSION_INIT action initializes the algorithm in the proper mode, i.e., encryption mode or decryption mode, and associate the operation with a key object. Subsequent NCRIO_SESSION_UPDATE actions operate on input data, i.e., by hashing or encrypting, and might generate output data depending on the operation. The NCRIO_SESSION_FINAL will terminate the operation and if required output data (e.g. in signature generation). To avoid the need of multiple system calls for simple operations, i.e., encryption of a given data block we introduced NCRIO_SESSION_ONCE, which combines the functionality of the above operations in a single system call.

Key wrapping refers to the technique of encrypting a key with another.
**Key wrapping and unwrapping**  The key wrapping subsystem allows transferring keys securely to other systems by encrypting a key with another. The subsystem accepts key encryption keys if they are marked as wrapping keys. The marking of a key as wrapping or unwrapping is a privileged operation (requires administrative access). An integrity check is performed by all the algorithms supported by the subsystem to ensure that random data cannot be unwrapped to a valid key. The type and associated algorithm with the key are included with the wrapped data to distinguish between different keys. The calls for wrapping and unwrapping keys are `NCRIO_KEY_WRAP` and `NCRIO_KEY_UNWRAP`. Because this operation separates the keys from their associated flags, it is a privileged operation.

**Key storage**  A difference with user-space security module frameworks is that kernel-space provides no ability of data storage. For this reason we adopted a design where the storage is delegated to the user-space applications. The protection of safe keys is being done by encrypting and authenticating everything with a system-wide master key. The keys, meta-data such as flags and an access control list are encrypted and authenticated, and handed out as binary data to user-space. Hence, the key storage facility –provided by the framework’s user– is agnostic on the actual keys it stores. It is not able to recover any key, and can only pass it to kernel-space for loading. No naming scheme is imposed by the framework and the user can adopt any convenient scheme for the stored key objects. The corresponding calls for storing and loading keys for storage are `NCRIO_KEY_STORAGE_WRAP` and `NCRIO_KEY_STORAGE_UNWRAP`.

**System-wide master key**  As previously discussed a master key is required to allow the key storage subsystem to operate. The only requirement for this master key is that it has to be loaded by a privileged user –the administrator. In a typical deployment we expect that the key will be provided either as part of the boot process or protected using the filesystem access control mechanism. It should be noted that the overall system security is only as strong as this master key.

**Key derivation**  Unlike the ciphers and other algorithms supported by the ‘Abstract session interface’, the Diffie-Hellman key exchange, cannot fit the

---

7Storing the system key in a hardware security module, when available, might seem to increase the defenses of the framework even against an adversary with administrative access. However, although such protection would prevent him from recovering the storage key, it would allow him to recover all the locally stored long-term keys. Because we expect long-term keys to be locally stored, the framework would not protect against this adversary.
NCRIO_SESSION_INIT, NCRIO_SESSION_UPDATE and NCRIO_SESSION_FINAL formality of operations. To solve that issue, PKCS #11 defines a way to operate it in software by using a key derivation API, which we adopt. This key derivation API is a single action NCRIO_KEY_DERIVE that accepts an input key, an output key and parameters.

Key generation  This subsystem supports two actions NCRIO_KEY_GENERATE and NCRIO_KEY_GENERATE_PAIR. The former action generates a random key of a given bit size to be used in MAC and symmetric ciphers. The latter generates a public and private key pair suitable for an algorithm such as RSA, DSA or Diffie-Hellman. All generated keys are tied to a single algorithm, and special parameters are allowed to fine-tune the key generation, i.e., by specifying bit lengths for symmetric keys, or prime and generator for Diffie-Hellman keys etc. Special flags can be given to keys during generation time. They are split in operation and policy flags as in Table 2. However, not all combinations are allowed, to prevent interactions between different algorithms (e.g., prevent a key to be used for RSA signing and encryption).

<table>
<thead>
<tr>
<th>Operation flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encryption</td>
<td>The specified key can be used only for encryption</td>
</tr>
<tr>
<td>Decryption</td>
<td>The specified key can be used only for decryption</td>
</tr>
<tr>
<td>Signature</td>
<td>The specified key can be used for signing only</td>
</tr>
<tr>
<td>Verification</td>
<td>The specified key can be used for verifying signatures only</td>
</tr>
</tbody>
</table>

**Table 2:** Flags used to mark keys. More than one flag may be applicable to a key.

<table>
<thead>
<tr>
<th>Policy flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrapping</td>
<td>The specified key can be used for wrapping other keys</td>
</tr>
<tr>
<td>Unwrapping</td>
<td>The specified key can be used for unwrapping other keys</td>
</tr>
<tr>
<td>Wrappable</td>
<td>The key can be wrapped by other keys</td>
</tr>
<tr>
<td>External</td>
<td>The key was not generated within the framework</td>
</tr>
<tr>
<td>Exportable</td>
<td>The key can be exported from the component</td>
</tr>
</tbody>
</table>

Key Import and Export  Keys that are explicitly flagged as exportable can be exported using the NCRIO_KEY_EXPORT action. Symmetric keys are exported in raw format, and public and private keys encoded using ASN.1 DER encoding rules.
5 Security analysis

In this section we discuss how this implementation defends against attacks on cryptographic key isolation frameworks and side-channel attacks. In Appendix A we demonstrate through a human-directed proof how a particular set of security objectives, based on our requirements, are met, in the context of a TLS server application.

5.1 Cryptographic keys protection

Several cryptographic APIs that enforce isolation of keys, have issues that result in modification of key attributes [7, 6] by an adversary. We counter those issues by enforcing keys and attributes to be handled as a single unit in all operations. When for example, keys are exported from the framework, i.e., for storage, the exporting function will authenticate the data [12] to prevent modifications. This authentication depends only on the system key, preventing attacks as in [6].

5.2 Safe keys

Safe keys have no policy flags and only allow cryptographic and storage operations. The supported ciphers are known to be secure against chosen-ciphertext and chosen-plaintext attacks, and the signature algorithms against adaptive chosen-message attacks [11]. This ensures that cryptographic operations alone will not reveal the key. The storage operations as discussed above encrypt and authenticate the key itself, as well as all metadata using the system’s master key. This way unauthorized modification or decryption by all types of adversaries is prevented. In addition to the file system’s access controls that will be used to store the key, the storage key format includes and access control list that specifies the users allowed to unwrap it. Thus, an adversary that has successfully attacked one user account can not use it to load another user’s keys.

5.3 Wrappable keys

Wrappable keys cannot be exported in raw but have to be encrypted by another designated key available to the framework. It is used as a mechanism to allow transfer of keys to other systems in a secure way, i.e., without revealing them in the process. In [7] Clulow describes several attacks to PKCS #11 wrappable
keys which are countered by making wrapping and unwrapping privileged operations. To defend against a (stronger) adversary that might obtain and modify such keys, we do not allow arbitrary wrapping mechanisms but only algorithms that include an integrity check [12].

5.4 Exportable keys

Keys marked as exportable cannot be protected by this framework. They should be considered as having the same security level as any keys available to user-space.

5.5 Side channel attacks

Adversaries of such a framework have incentive to take advantage of side-channel attacks such as the ones described in [3, 17], to use timing and other information obtained by operations in order to extract the raw keys. For the purposes of this project we implemented the counter measures for the RSA algorithm, known as RSA blinding [17]. To counter timing attacks in the Linux kernel symmetric cipher subsystem, constant time ciphers should be added. This was not performed as part of our prototype but would be required for a real-world deployment.

6 Performance Analysis

In this section we show that the performance cost of decoupling of cryptographic keys from user-space processes. We compare the NCR framework (/dev/ncr) against a key isolation framework based purely in user-space. Moreover we include performance results of non-isolation frameworks such as the Linux port of the OpenBSD framework8 [16] (/dev/crypto), and a user-space implementation of the AES. The latter two would reveal the performance “loss” by the usage of a key isolation framework.

For the benchmark we created a utility that submitted requests for encryption on each framework of chunks of data with different size and a fixed key. The AES cipher in CBC mode, and the NULL (dummy) cipher were used in our tests. The same implementation of AES was used in all frameworks under test, and all were tested on the same hardware and Linux kernel version 3.0.0.

8The Linux port of the OpenBSD framework was selected because of its performance advantage [8] over the native Linux cryptographic API (AF_ALG).
For a submission of a chunk of data for encryption, the /dev/crypto interface requires 3 system calls, the same as the NCR framework. However NCR allows for submission of requests using a single system call, the NCRIO_SESSION_ONCE. We called the latter ‘/dev/nr single’ variant and included it to our benchmark to assess the optimizations obtained by reducing the number of system calls.

The only existing user-space key isolation framework that was available for the same system was LSM [18], but its design is based on the sockets API, requiring TCP/IP processing and data copying between the processes. A comparison with the NCR or the /dev/crypto frameworks that are zero-copy⁹ would be negatively affected by such overhead. For that reason we implemented a user-space benchmark that would operate using the zero-copy principle. It involves two applications sharing memory and using semaphores for synchronization of operations.

![Figure 3: Performance comparison of /dev/nr, /dev/crypto, and a user-space crypto server on the NULL cipher.](image)

In Figure 3 we show a comparison of all frameworks using the NULL cipher and in Figure 4, a comparison using the AES cipher. Since the NULL cipher performs no operations, the former results reveal the overhead of the framework and its maximum performance limit. Comparing to the /dev/crypto interface we see that the NCR implementation has no noticeable performance difference. This illustrates the fact that decoupling of cryptographic keys from processes does not introduce any performance cost to a kernel framework. We can also

⁹By zero-copy we mean that no data are copied from the memory of the requesting application to memory of the serving application or kernel.
Figure 4: Performance comparison of `/dev/ncr`, `/dev/crypto`, a user-space isolation framework and a user-space non-isolation framework, on the AES-CBC cipher.

see from the performance of NCR with a single system call, that the impact of the reduction of system calls from 3 to 1 is important. That is because system calls require context switches (switch execution from user-space to kernel-space) which come at a considerable cost.

The user-space based isolation framework shows very poor performance compared on the others. Profiling of the benchmark applications showed that this performance difference is due to the usage of semaphores for synchronization of the two processes. In the NULL cipher case, 50% of the execution time on each process was spent in synchronization. That is because semaphores in Linux although they are based in user-space futexes, do not completely avoid context switches, causing the observed poor performance. The difference between user-space and kernel-space implementations is reduced to $\approx 10\%$ when a software algorithm such as AES is involved, even though, on the NULL cipher the difference exceeds 50%. Thus over the Linux kernel, a user-space approach to cryptographic key isolation would have disadvantage on high-rate or high-bandwidth transactions.

Both isolation frameworks, NCR and the userspace, are in a disadvantage comparing to userspace non-isolation framework as shown in Figure 4. Although the performance difference seems to become insignificant as larger data chunks are used, large chunks of data might not be the typical use-case of such a frame-
work. For that we would suggest that a key isolation framework should be used after the benefits of isolation are weighted against the performance loss.

7 Comparison with alternative approaches

The differences between different framework designs, such as based on the separation of processes or the separation with the OS are not limited to performance. A separation framework might have additional goals from the requirements we set in Section 3. For example the CNG framework [19] abstracts the usage of hardware security modules, smart cards or software isolation using the same interface. This is not easily implemented in a framework residing in the OS, since functionality such as access to smart cards\textsuperscript{10} is not typically available. Portability is another concern. A cross-platform solution cannot rely on a particular kernel interface.

On the other hand, there are advantages of an OS based framework. This is mainly performance as illustrated in the previous section, as well as efficient access to cryptographic accelerators [16] and the secure bootstrapping property [2]. Cryptographic accelerators, when not implemented in the CPU instruction set are only available through a kernel-space driver. A kernel-space cryptographic framework is in advantage of using the driver directly without the cost of context switches. In a secure bootstrapping architecture each system module verifies the modules it loads (or uses). For example, the system loader would verify the OS and the OS would verify the loaded applications and so on. Having the cryptographic framework in the OS allows for a clear and simple design since the OS would already contain the functionality needed to perform signature verification, i.e., the algorithms, and this functionality has already been verified by the system loader as part of the OS. We provide a summary of the advantages and disadvantages in Table 3.

8 Concluding remarks

In the previous sections we have presented the implementation of a cryptographic framework for the Linux kernel. The main goal of the framework is to decouple cryptographic keys from the applications using them, by confining the keys to kernel-space. This confinement is achieved through the enforcement of rules on the way keys are generated and utilized in order to protect against

\textsuperscript{10}For example parsing PKCS #15 structures or PKCS #11 modules requires an addition of large subsystems to the OS.
known attacks on cryptographic APIs. The API is practical, in the sense that it can be used to fully implement cryptographic protocols such as TLS 1.2 [10], and can be used to provide services under the PKCS #11 API. It is available to all system users and the administrator role, although required, is limited to maintenance aspects of the framework, such as loading the master keys, and the generation of wrapping keys.

Our performance evaluation showed that the throughput of the framework is equivalent to another kernel-based cryptographic framework that does not decouple keys from applications, indicating that there are no issues in the design that hinder performance. Moreover we show that alternative designs consisting of user-space only components will have performance degradation, caused by the inter-process communication overhead. Thus we believe that our software implementation provides a practical additional layer of protection for cryptographic keys.

9 Acknowledgments

The authors would like to thank Steve Grubb for the inception of this project, Jan Filip Chadima for implementing a user-space interface to the NCR framework, Stephan Müller, Tomáš Mráz, Óscar Repáraz, Andreas Pashalidis, Nessim Kissler, Jan Cappaert and the anonymous referees for valuable comments and feedback. This work was supported in part by the Research Council K.U.Leuven: GOA TENSE, by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy) and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen) SBO project.
References


A Application to TLS key protection

This section describes in detail how the proposed framework could be used to protect the private keys used to identify a TLS [10] server. The approach is inspired by the security target format defined by the common criteria standard [13].

A.1 Problem definition

The TLS 1.2 protocol uses a public-private key pair to identify the server and avoid man-in-the-middle (MITM) attacks: the public key is included in a certificate signed by a trusted third party and provided to the client as part of the TLS connection establishment process. The protocol can use either a RSA key pair, or a Diffie-Hellman key pair. Although the algorithm and key exchange method choice affects internal operation of the TLS handshake, for our purposes we can treat both kinds of keys in the same way: the private key must be available to the TLS server, and disclosure of the private key to an
adversary allows him to successfully mount a MITM attack impersonating the legitimate server.

As Section 4.1 describes, protection of session key material does not provide a meaningful improvement in security, and an adversary is necessarily able to mount a MITM attack, or an equivalent information disclosure attack, during the time the TLS server is under the adversary’s active control.

We therefore consider the following components of the problem of protecting a TLS server.

**T1** Threat to be mitigated: An adversary could impersonate the TLS server for an unbounded time. This is based on the capabilities of the first adversary described in Section 3.1.

**T2** Necessary functionality: The TLS server must have enough access to the private key identifying the server (referred to simply as “the key” from now on) to be able to serve legitimate requests.

We also assume that the TLS private key is managed by the administrator.

### A.1.1 Security objectives

To handle the above-described problem, we have chosen a set of specific objectives that we want the TLS server deployment to meet fully.

**O1** The TLS server process can remain compromised only for limited time.

**O2** Only the TLS server process and processes started by the administrator can perform operations using the key.

**O3** An adversary who controls the TLS server can not export the key to a different computer.

**O4** Processes of the administrator are trusted not to reveal the key.

**O5** The TLS server has enough access to the key to be able to perform RSA decryption, RSA or DSA signing and Diffie-Hellman shared secret establishment, depending on TLS key exchange method.

These objectives constitute one possible approach to handling the problem set out in Section A.1: O1–O4 are together designed to mitigate threat T1, and O5 to provide functionality required by T2.

Whether, and how effectively, the chosen objectives handle the problem set, is a question that must be assessed by human judgement.\(^\text{11}\)

\(^\text{11}\)For example, the above set of objectives ignores ways to mount threat T1 without attacking the computer on which the TLS server runs, such as discovering the private key from the
A.2 Requirements and policies

Finally, we design a set of requirements on the cryptographic module described in this article and policies that must be enforced by the server’s administrators by system configuration or setting up human processes. The requirements and policies were chosen in a way that allows us to demonstrate that all objectives were met in a semi-formal way.

We use the following requirements which are met by the module and were verified by manual inspection of the source code:

R1 The only ways to make a pre-existing key available as a /dev/ncr key object is to load it using the key storage component, unwrap it, and to import it from a plaintext representation.

R2 The wrapping and exportable flag restrictions are correctly enforced.

R3 No other user than administrator is able to set the wrapping or exportable flag of an existing key.

R4 No user other than administrator is able to make a copy of an existing non-exportable key with one or both of the flags set if they were not set on the original without having access to a key with the wrapping flag set.

R5 A key wrapped for storage identifies users allowed to unwrap or to perform operations using the key, and this restriction is correctly enforced.

The following policies are the responsibility of system administrators:

P1 Any TLS server compromise is discovered and fixed in a bounded amount of time.

P2 The TLS server runs under a dedicated user account.

P3 The key, wrapped for storage, is stored in a file that is accessible to the user account of the TLS server.

P4 The key is not wrapped, using the storage master key or any other key, in files that are accessible to any other user account, except perhaps for the administrator.

P5 The key wrapped for storage in P3 above identifies the user account of the TLS server, and no other account, as allowed to unwrap the key.

P6 The storage wrapping master key is available at most to the kernel and processes running as administrator on the TLS server computer.

P7 Processes of the administrator are trusted not to reveal the key.

public key by brute force attacks, or obtaining a copy of the key from a key escrow system.
The key wrapped for storage in P3 above does not have the exportable flag set.

The TLS server does not have access to any key with the wrapping flag set.

The key is not stored in any file unencrypted, nor encrypted with a key available to processes not running as administrator.

A.3 Meeting security objectives

We can now demonstrate that the requirements and policies from Section A.2 are sufficient to implement the objectives we have chosen in Section A.1.1:

We start with two auxiliary claims:

L1 If a key does not have the wrapping or exportable flag set, then the TLS server can not wrap using the key, or export the key, respectively: ensured by combination of R2, R3, R4, P9.

L2 An adversary who controls the TLS server can not extract the key material to user space:
  - The key is not available exported in plaintext: P8, L1, P10.
  - The key can not be extracted through the key wrapping mechanism: P9, L1.
  - The key can not be extracted using the storage wrapping mechanism: P6.

Moving on to the objectives:

- O1 is implemented by policy P1.
- To demonstrate O2 is implemented, consider:
  - Only the TLS server process and processes started by administrator can load the key into a /dev/ncr key object: R1, P4, P2, and P10.
  - An adversary who controls the TLS server can not make the key object available to other user accounts on the same computer: R5, P5, P9, L1.
  - An adversary who controls the TLS server can not extract the key material to user space: L2.
- To demonstrate O3 is implemented, consider:
  - The adversary can not extract the key material to user space: L2.
The adversary can not use the key wrapping mechanism to transfer the key material: P9, L1.

The adversary can not use the storage wrapping mechanism to transfer the key material: P6.

- O4 simply becomes policy P7.
- O5: The key is made available to the server by P3.
5.3 Security implications in Kerberos by the introduction of smart cards

Publication data


Contributions

Principal author together with Andreas Pashalidis.
Security implications in Kerberos by the introduction of smart cards

Nikos Mavrogiannopoulos Andreas Pashalidis
Bart Preneel
Katholieke Universiteit Leuven
ESAT/SCD/COSIC – IBBT
Kasteelpark Arenberg 10 Bus 2446
B-3001 Leuven-Heverlee
Belgium

May 2012

Abstract

Public key Kerberos (PKINIT) is a standardized authentication and key establishment protocol which is used by the Windows active directory subsystem. In this paper we show that card-based public key Kerberos is flawed. In particular, access to a user’s card enables an adversary to impersonate that user even after the adversary’s access to the card is revoked. The attack neither exploits physical properties of the card, nor extracts any of its secrets. We propose protocol fixes and discuss properties that authentication and/or key establishment protocols should provide in order to be better equipped against the threats that arise due to the usage of smart cards.

1 Introduction

Consider the following real-world scenario. On his way to a customer, Bob realizes that he forgot his smart card in the drawer of his desk. Knowing that he needs a document that is stored on his company’s file server, which he can access only using the card, he calls his colleague Alice for help. He explains his situation, and asks her to retrieve the document using his smart card from the drawer, and email it to him. He also tells her the required PIN. Alice follows his instructions, and Bob’s meeting goes well. On the next day, after thanking
Alice for her help, Bob takes back his card and, as a security precaution, also changes its PIN.

In the above scenario, Bob expects Alice and, for that matter, anyone else, to be unable to impersonate him from the moment he regains control of his card. Bob’s expectation is reasonable, and stems from the fact that smart cards are possession-based authentication tokens; non-possession must lead to the inability to authenticate.

In this paper, we describe an attack against the Diffie-Hellman (DH) variant of the public key Kerberos protocol [33], that enables Alice to continue impersonating Bob even after he has regained control of his card and changed his PIN. Our attack, which assumes that smart cards are used for user authentication, shows that, contrary to reasonable expectation, smart card-based public key Kerberos fails to provide authentication based on ‘something you have’. Moreover, it demonstrates that deployments where smart cards are protected with a PIN for the purposes of two-factor authentication, in fact do not provide any authentication factor. This is because Alice is able to continue impersonating Bob even after a PIN change. Our attack neither exploits the physical properties of the card, nor extracts any of its secrets. Note that smart card-based public key Kerberos is a deployed protocol [12, 14, 21], and has been proven secure under certain models [2, 8, 11, 26]. Also note that implementations that conform to the specification [33] are required to support the DH variant.

The models under which public key Kerberos was shown to be secure do not support the outsourcing of certain functions to a removable smart card which is subject to attack independently from the user’s terminal. Finally, we propose a fix to the protocol, and discuss desirable protocol properties.

The rest of this paper is organized as follows. The next section surveys related work. Section 3 describes our attack and proposes a protocol fix. Section 4 discusses certain protocol properties and explains why they are particularly desirable when smart cards are used as secure computation devices. Section 5 discusses whether existing formal analysis are sufficient to identify the attack. Finally, section 6 concludes.

2 Background and related work

Kerberos is an authentication and key distribution protocol originally proposed in 1988, and has a long history of attacks and updates (see, for example, [2, 11] and the references therein). Its main goal is to enable users to log into multiple servers that belong to a common infrastructure. To this end, a user first requests an electronic ‘ticket’ from a central Key Distribution Center (KDC).
The ticket then enables authenticated users to log into any server that is part of the infrastructure.

The focus of this paper is the DH variant of the public key Kerberos protocol (PKINIT), as specified in [25, 33]. This protocol naturally lends itself to an implementation where the user’s private signing key is stored in a smart card [12, 14]. In the resulting combined protocol, the smart card functions as a signature device that signs the user’s ticket requests. However, introducing a smart card to a security protocol presents risks. This is because the system is now divided into a larger number of entities, each of which is subject to attack [19, 28]. A large body of literature discusses, for example, attack strategies that involve corrupting the user’s terminal [1, 6, 9, 31].

Several models, such as the BAN logic [10, 20], process calculus-based logics [13], and complexity-theoretic analysis techniques (e.g. [4]) capture protocol security guarantees on authentication and key exchange. The adversary they consider is typically a Dolev-Yao-style adversary [17], i.e. is assumed to control the entire network, but is not able to corrupt participants. Other models that consider stronger adversaries, i.e. ones that may also corrupt participants by extracting their private keys, have been used to capture the notion of perfect forward secrecy [16, 7, 29].

The public key Kerberos protocol, including its DH variant, has been proven secure under a Dolev-Yao-style adversary model [2, 8, 11, 26]. However, the adversary models used in the above studies do not consider adversaries with attack abilities such as temporary access to a user’s card. That is, the currently used models for protocol verification do not capture our attack and, more generally, their positive results of protocol verification do not carry over to a smart card setting.

Despite the fact that smart cards have been introduced to a multitude of protocols, to the best of our knowledge, only few works provide formal treatments of smart card-based protocols against an adversary that is able to attack a user’s terminal independently from his smart card (see, for example, [3, 18, 30]). Our attack underlines the necessity of such models in verifying a protocol’s security guarantees when smart-cards are involved.

3 Our attack

This section briefly revisits the DH variant of public key Kerberos with smart cards [12, 21, 25], describes our attack, and proposes a fix. The attack was disclosed to the IETF Kerberos mailing list in June 2011 [23].
3.1 DH public key Kerberos with smart cards

The protocol specifies three types of player: users, servers and a central entity called the Key Distribution Center (KDC). A user is equipped with a terminal and a smart card. The card contains an asymmetric key pair and a certificate, signed by an authority, that binds the public key to the user’s identity. Moreover, the card provides an interface over which the terminal can ask the card to sign messages using the private key. Note that the card may require a PIN in order to respond to requests from the terminal.

One of the main goals of the protocol is to establish fresh session keys between users and servers. Figure 1 provides a high-level overview of the protocol. When the user decides to log into a server (step 0 in the Figure) his terminal constructs an AS_REQ message as specified in [25] and then calculates its hash value \( h \). Then it chooses a DH group, randomly generates an ephemeral DH secret \( x \in Z_p \), and computes \( g^x \), where \( p \) is a large prime and \( g \) is the generator of the group. The terminal also chooses a nonce \( n \) and stores the current time. It then provides the values \( g^x, n, \tau \) and \( h \) to the smart card for signing (step 1). Note that, depending on the implementation, it may only provide a hash of these values to the smart card. If the smart card is PIN-enabled, then the user must provide his PIN prior to this operation. Depending on the implementation, the PIN is inserted either to the terminal or the smart card reader. Figure 1 shows the case where the user provides his PIN to the terminal in step 0, and where this gets sent to the card in step 1.

The signature \( \sigma \), output by the smart card (step 2), is then used by the terminal to construct an augmented version of the AS_REQ message, which we denote by AS_REQ*. This message, which contains \( g^x, n, \tau, \sigma \) and AS_REQ as a substructure, is sent to the KDC (step 3) which, among other things, verifies the signature.\(^1\) If verification succeeds, then the KDC chooses a random \( y \in Z_p \), computes \( g^y \) and the ephemeral secret \( \kappa = g^{xy} \), and constructs a response AS_REP according to [25]. This message contains a ticket which is encrypted with \( \kappa \). Finally, it augments this message with fields containing the value of \( g^y \) and \( n \) signed with server’s private key. The resulting message, denoted AS_REP*, is sent to the terminal (step 4).

Using the value \( g^y \) and its ephemeral secret \( x \), the terminal recovers the key \( \kappa \) and is therefore able to decrypt the ticket. This ticket enables the terminal to complete the subsequent message exchange with the desired server (step 5). At the end of this exchange, which follows the standard [25] and does not involve

\(^1\)Note that \( n \) and \( \tau \) are used for replay protection. In particular, the KDC stores the received nonce \( n \) until a particular expiry time, which is calculated based on \( \tau \). All AS_REQ* messages that (a) are received until the time of expiry and (b) contain the same nonce, are considered replays and are rejected.

125
the smart card, the user and the server will have established a session key. The exact details of this exchange are not relevant to our attack.

Figure 1: The DH variant of public key Kerberos with smart card (simplified).

3.2 The attack

Our attack is based on the observation that the KDC has no means to verify whether or not an incoming \texttt{AS.REQ} message is fresh. That is, while the KDC checks that the timestamp $\tau$ indicates approximate current time, this does not guarantee that the \texttt{AS.REQ} message was constructed recently. In fact, the \texttt{AS.REQ} message could have been generated in the distant past.

In order to mount our attack, the adversary first obtains access to a victim’s smart card. This can be done either by compromising the victim’s terminal, or by stealing the card and its PIN. The adversary then fabricates an \texttt{AS.REQ} message, calculates its hash $h$, chooses a nonce $n$, a random DH secret $x$, and chooses a timestamp $\tau$ indicating a particular future point in time. It then sends the values $g^x$, $\tau$, $n$, and $h$ to the card in order to obtain the signature $\sigma$. Using this signature, the adversary constructs an \texttt{AS.REQ} message.

Note that this fabricated \texttt{AS.REQ} message will be accepted by the KDC as a genuine ticket request from the victim at time indicated by $\tau$. Since neither the victim’s himself, nor his smart card is required in the remainder of the protocol, the message enables the adversary to impersonate the victim to the KDC at time $\tau$. With the ticket in the KDC’ response, the adversary will further be able to impersonate the victim to the server of his choice until the ticket expires.

Note that, in order to be able to impersonate a victim at, say, approximately 20:00 of every Monday in a two-year period, the adversary must fabricate about 104 \texttt{AS.REQ} messages as described above and, for each such message, obtain a signature from the card. In other words, a few minutes of access to a victim’s
card are sufficient for the adversary to be able to impersonate the victim, on a regular basis, for years.

3.3 Proposed fixes to the protocol

If the user’s smart card was required in a later protocol stage, our attack may have been avoided. The RSA public key encryption variant of public key Kerberos, for example (also specified in [33]), does not appear to be affected by our attack because, in this variant of Kerberos, the smart card is required to decrypt the KDC’ reply. Thus, the adversary would not be able to decrypt replies from the KDC without access to the smart card. However, switching to this Kerberos variant may not be desirable because it allows violation of the key separation principle: the same RSA key pair is used for both signing and decrypting [22, 33]. Moreover, switching to this Kerberos variant is not an option for smart cards that are based on the DSA or ECDSA cryptosystems [32, 33], since they not support decryption.

We propose two fixes, one that completely defends against the attack at the cost of an additional message exchange, and a fix that limits the effectiveness of the attack to a specific time period. None of the proposed fixes requires already deployed cards to be replaced, assuming that the cards conform to widely used standards [24, 27], and can therefore can sign arbitrary data. They also do not require involvement of the user’s smart card after the initial message is generated, but instead provide the means for the KDC to verify the freshness of incoming \texttt{AS\_REQ}\textsuperscript{*} messages.

The first fix requires the KDC to initially send a nonce $n_S$ to the terminal. This nonce is then added to the data signed by the smart card and included in \texttt{AS\_REQ}\textsuperscript{*} (see step 1 in Figure 1). On reception of the \texttt{AS\_REQ}\textsuperscript{*} message, the KDC must also ensure that the number $n_S$ in the message matches the number it generated. Note that this fix requires a message from the KDC to the terminal carrying the nonce, and a change to the \texttt{AS\_REQ}\textsuperscript{*} message to accommodate the additional nonce.

An alternative fix does not change the two-message negotiation of Kerberos but requires the client to obtain a fresh ‘cookie’ from the KDC at a regular intervals, selected by the KDC (e.g. once per day). The cookie is randomly generated by the KDC periodically and is used to ensure the freshness of the incoming \texttt{AS\_REQ}\textsuperscript{*} messages. For that, the cookie must be covered by the signature of the \texttt{AS\_REQ}\textsuperscript{*} message. Note that now an adversary can successfully launch our attack within the cookie validity period.

Finally, it is perhaps worth mentioning that, albeit in reference to a previous version of Kerberos, weaknesses by the use of timestamps were already pointed
out, and replacing them with nonces was already suggested, more than twenty years ago [5].

4 Desirable properties for smart card protocols

Authentication and session key establishment protocols that make use of smart cards are exposed to threats that arise due to fact that certain functions are performed by the terminal while others take place on the smart card. The most important threats are adversaries that (a) have temporary access to smart cards, and (b) compromised user terminals. Smart card-based protocols should resist these threats by automatically recovering when a temporary compromise is over. That is, if the adversary’s access to the smart card is revoked, or when a non-compromised terminal is used, then the protocol should provide the same level of security as if no compromise ever happened.

In the following, we list four protocol properties that are aimed to address the above threats. Although they do not constitute a formal security model, we believe that they offer a practical check list for protocol designers and implementers. Note that we list only properties that address the smart card-specific threats. As such, our list complements the list of desirable protocol properties from [7].

The following properties ensure that a temporary compromise only affects the secrecy of sessions that fall inside the time window of the compromise.

- **SC perfect forward secrecy**: Session keys that were established with a user’s smart card over a non-corrupted terminal remain secret, even if an adversary later obtains access to that user’s smart card, as long as the adversary does not extract any long-term secrets from the card.

- **SC backward secrecy**: Session keys that are established with a user’s smart card over a non-corrupted terminal remain secret, even if an adversary had previously accessed the user’s smart card although, without extracting any long-term secrets.

Note that perfect forward secrecy as defined in [7, 16] requires session keys to remain secret even if the adversary has knowledge of long-term keys. **SC perfect forward secrecy** is, therefore, a weaker notion; however, we argue that it is more relevant in practice to smart card-based protocols because the cost of obtaining access to a victim’s smart card and simply performing computations with it, is typically much lower than the cost of extracting its long-term secrets. Of

2While we use the term ‘smart card’, the discussion in this section also applies to protocols that make use of other security tokens or modules.
course, if a protocol provides the stronger notions of forward/backward secrecy, then it is likely to also satisfy the above notions.

The following properties ensure that a temporary compromise is not adequate for the adversary to impersonate legitimate users outside the time window of the compromise.

- **SC key-compromise impersonation**: An adversary with access to a user’s smart card, but without having extracted the long-term secrets stored in the card, should not be able to impersonate other entities to that user, as long as the user uses a non-corrupted terminal.

- **Possession-based authentication**: An adversary with access to a user’s smart card, but without having extracted the long-term secrets stored in the card, should be able to impersonate that user only for as long as it has access to his card.

*SC key-compromise impersonation* is a weaker notion than key-compromise impersonation as defined in [7], because the adversary is not given the user’s long-term secrets. Finally, possession-based authentication captures the ‘ownership’ of the card, in particular that performing operations with the keys stored in the card implies physical possession of the card. A protocol that does not offer the above properties, fails to meet reasonable expectations of any system that is supposed to provide authentication based on ‘something you have’.

## 5 Existing verification methods

The existing models used for the verification of public key Kerberos [2, 8, 11, 26] consider a Dolev-Yao style adversary where, in addition to the network, the adversary may also control statically corrupted parties. Since they do not consider the smart card setting, these models do not capture an adversary with temporary access to a smart card.

Shoup and Rubin present such an approach in [30]. Their model considers an adversary that can perform operations with a user’s smart card, and the applicability on the Kerberos protocol can be easily demonstrated, showing that existing protocol verification methods are sufficient to model the requirements of smart card-based protocols.
6 Concluding remarks

This paper demonstrates that seemingly straight-forward protocol extensions, such as the migration of the user’s private key from his computer to a smart card, can have severe implications to the security of the overall system. Even though previous work has argued for the need of a separate model for smart card protocols [28, 30], the case of public key Kerberos shows that the usage of smart cards is sometimes still handled as a deployment issue rather than a fundamental change to the protocol. As a result, the proofs of security for public key Kerberos, which were based on the protocol specification without smart cards, become effectively irrelevant.

We showed that the DH variant of public key Kerberos, when used with smart cards, enables an adversary with access to a user’s smart card, to impersonate that user even after access to his smart card is revoked. We also proposed fixes for the protocol. Clearly there is need to formally verify smart card-based public key Kerberos, with our proposed fixes, under a smart card-aware model such as [30]. Such a treatment would not only verify, or disqualify, the effectiveness of our fix, but may also uncover other unidentified issues in the overall protocol.

More generally, there is a need to formally verify all smart card-based protocols, especially those that have been implemented and deployed, such as TLS with PKCS#11 smart cards [15, 27], under an appropriate model.

7 Acknowledgments

The authors would like to thank Jens Hermans for his comments which improved the manuscript. This work was supported in part by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen) SBO project, the Research Council K.U.Leuven: GOA TENSE (GOA/11/007), by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and in part by the European Commission through the support action with contract number 258630 GINI SA and CIP thematic network with contract number 270901 SSEDIC.

References


5.4 Towards a secure Kerberos key exchange with smart cards

Publication data


Contributions

Principal author together with Andreas Pashalidis.
Towards a secure Kerberos key exchange with smart cards

Nikos Mavrogiannopoulos Andreas Pashalidis
Bart Preneel
Katholieke Universiteit Leuven
ESAT/SCD/COSIC
Kasteelpark Arenberg 10 Bus 2446
B-3001 Leuven-Heverlee
Belgium

January 2013

Abstract

Public key Kerberos (PKINIT) is a standard authentication and key establishment protocol. Unfortunately, it suffers from a security flaw when combined with smart cards. In particular, temporary access to a user’s card enables an adversary to impersonate that user for an indefinite period of time, even after the adversary’s access to the card is revoked. In this paper, we extend Shoup’s key exchange security model to the smart card setting, and examine PKINIT in this model. Using this formalization, we show that PKINIT is indeed flawed, and provide a proof that a recently proposed fix leads to a secure protocol.

1 Introduction

It is well known that human users can be authenticated based on something they know (e.g. a password), something they have (e.g. a smart card), or some part of themselves (e.g. a fingerprint). Unfortunately, possession-based authentication systems are more complex and costly to implement, deploy, and maintain than knowledge-based authentication systems. This is because, while the user’s mind operates as the secure storage place for his password in a knowledge-based system, in a possession-based system hardware tokens and special software must be manufactured, distributed, and managed in the user’s
terminal. Analyzing the security of systems that are based on hardware tokens is also more challenging because security-critical functions are performed both by the user’s terminal and the token itself (see [22] for a detailed discussion of the implications of this situation). Moreover, hardware tokens are often protected by a Personal Identification Number (PIN); while this potentially leads to stronger security, often called ‘two-factor authentication’, it complicates the security analysis further.

Another complicating factor, which is often overlooked, is that users may share their tokens just as they share their passwords, even if explicitly told to not do so [23]. We assume the following mental model with respect to sharing. While a user will change his password in order to revoke the rights from those with whom the password was shared, a user that wishes to revoke the rights from those with whom his token was shared, simply retrieves his token and keeps it at a safe place. Possession-based authentication system propagate exactly this mental model, and, therefore, must guarantee that non-possession of the token leads to the inability to authenticate.

The system must also take into account the threat of an adversary that compromises the user’s terminal. Assuming that the token communicates through the terminal, such a compromise affects all users, irrespective of whether or not they share their tokens with others. A compromised terminal may trigger a large number of illegitimate communications with the token, and, unless the token has some mechanism to alarm the user, the user cannot detect the attack.

Apart from authenticating users, token-based authentication systems are also used to establish session keys between the user’s terminal and a remote server. These keys, which are used to protect the communication between the user and the server, must be secret, even to someone who had previously access to the token. The system must therefore protect these keys from threats they are exposed to whenever the token is within an adversarial environment (i.e. either an adversary temporarily ‘borrowed’ the token, or compromised the user’s terminal).

In order to address the above threats, we argued in [17] about the need of certain protocol properties for token-based authentication and key establishment systems. To ensure the properties, the adversary is assumed to not be able to extract the long-term secrets from the token (e.g., a tamper resistant token). The following two properties ensure that a temporary compromise does not lead to the ability for an adversary to impersonate users or servers outside the time window of the compromise.

- **SC key-compromise impersonation**: An adversary with access to a user’s token, should not be able to impersonate other entities to that user, as long as the user uses a non-corrupted terminal.
- **Possession-based authentication:** An adversary with access to a user’s token, should be able to impersonate that user only for as long as it has access to the token.

The following two properties ensure the secrecy of past and future sessions.

- **SC perfect forward secrecy:** Session keys that were established with a user’s token over a non-corrupted terminal remain secret, even if an adversary later obtains access to that user’s token.

- **SC backward secrecy:** Session keys that are established with a user’s token over a non-corrupted terminal remain secret, even if an adversary had previously accessed the user’s token.

These properties complement the list of the desirable properties for secure communications protocol from [7].

The focus of this paper is the possession-based authentication system that arises when using the Diffie-Hellman (DH) variant of public key Kerberos [27] with typical PKCS #11 [21] compliant smart cards. The informal examination in [17] shows that, unfortunately, this system does not provide the property of possession-based authentication; an adversary that has temporary access to a user’s smart card at some point in time, can impersonate that user for an indefinite period of time. In this paper, we formalize this work. In particular, we extend Shoup’s key exchange security model [24] to the smart card setting, and examine PKINIT in this model. Based on this, we show that PKINIT is indeed flawed and provide a proof that a similar to the proposed fix in [17] leads to a secure protocol.

The rest of this paper is organized as follows. The next section overviews related work and Section 3 revisits the PKINIT protocol. The attack and a fix is described in Section 4. Section 5 revisits Shoup’s static corruptions model [24] and presents our extended version of the model. This extended version accounts for the presence of smart cards in the context of key establishment protocols. Section 6 applies the model to the original and the fixed PKINIT protocols. Finally, Section 7 concludes.

## 2 Related work

Several models, such as the BAN logic [9, 16], process calculus-based logics [13], and complexity theory analysis techniques (e.g. [4]) capture protocol security guarantees on authentication and key exchange. The adversary they consider

1Such smart cards are used in the real world.
is typically a Dolev-Yao-style adversary [15], i.e., is assumed to control the entire network. The PKINIT protocol, including its Diffie-Hellman variant, has already been proven secure under that adversary model [2, 8, 12, 20].

However, the Dolev-Yao model does not consider adversaries with attack abilities such as temporary access to a user’s card. That is, the models used for the PKINIT protocol verification do not capture attacks that involve smart card access, similar to our introduction story. More generally, their positive results of protocol verification do not carry over to the smart card setting.

Despite the fact that smart cards have been introduced to a multitude of protocols, to the best of our knowledge, only few works provide formal treatments of smart card-based protocols against an adversary that is able to attack a user’s terminal independently from his smart card (see, for example, [10, 25, 11, 3]). In the following paragraphs we provide an overview of the existing methods and their limitations.

2.1 BAN logic

Abadi et al. enhanced BAN logic [9] to account for smart card related threats in [10]. The BAN logic is a logic of beliefs that is used to prove certain properties in authentication protocols. The smart card extension is the first known formal treatment of smart card threats written in time where the notion of a smart card was not clearly defined. As such the protocols that are studied vary from smart cards that resemble the features of a modern smart card, to smart cards that include batteries (to maintain a clock), screen and keypad. The adversaries they consider are the typical Dolev-Yao network adversary, but include the abilities for smart card theft and terminal compromise. The latter is a main concern of the study, possibly because it was written in a time where smart cards were simple containers of data that were read by the terminal, and the authors cope with it explicitly by including terminal verification in the studied protocols. In their model, the user and the smart card are different entities that share a secret (the PIN). The original logic is enhanced with a notion for secure channels and the notion of timely channel (a channel which was established recently), and this logic is being used to verify and prove correct delegation-based protocols, i.e., protocols that allow delegation of authority such as the user’s card signing the terminal’s credentials for certain time to authorize the terminal to act on his behalf.

Its applicability on protocols that do not involve delegation (i.e., most modern protocols) is limited.
2.2 The Shoup-Rubin game-based method

Shoup and Rubin proposed in [25] a game-based model to capture the usage of smart cards with symmetric keys in key distribution protocols involving a trusted party. The threat model assumed is a static corruptions model extended to model smart card theft, terminal tampering, etc. It one of the first works to discuss the security implications and gains of using smart cards to protect cryptographic keys. The formal model used is similar to the Bellare & Rogaway model [5], which assumes a security parameter $k$, a number of hosts $n$, a trusted server $S$. Each host is given a smart card with long-term key $K_{1 \leq i \leq n}$ and $S$ is given key $K$. The smart card is modeled as a stateless probabilistic oracle. On input $x$ it returns $f(k, x)$. Each host $i$ may communicate with a host $j$ by using a process $II(i, j, u)$, where $u$ is a process identifier to allow more than one connection. The adversary is a polynomial time probabilistic algorithm that initializes and interacts with the system. The interaction is a queries/answer based communication with the processes and the server. The “transcript” is a list that contains the queries and answers ordered in time. The allowed queries to a process are delivery of a message, and response, as well as three special queries (1) for a process to reveal its session key, and (2) for a smart card to reveal its long-term key and (3) to access an operation of the smart card oracle. If the latter special queries are used, the process or the smart card are considered to be opened. During the interaction each process may output a message indicating acceptance which indicates that a session key was established.

The Shoup-Rubin verification approach is limited to long-term symmetric keys, something that makes the unsuitable for the verification of several modern protocols that utilize long-term asymmetric keys.

2.3 Inductive verification

Bella in [3] uses the mathematical induction as the main tool to prove the protocol’s security goals with respect to smart cards that protect symmetric keys. The possible threats are modeled as a set of events, defined by inductive rules. The security goals are then proved using induction on the set. The an adversary model is similar to the model of Shoup-Rubin, with an additionally modeled threat which the author calls “data bus failure”. This threat models smart card tampering in a way that messages from the card to the reader are modified or removed. The author then models the Shoup-Rubin protocol in [25] and uses the “Isabelle” tool to prove its security claims.

As with the Shoup-Rubin verification approach, Inductive verification is limited to long-term symmetric keys.
2.4 Resettable zero knowledge

Canetti et al. in [11] introduce the notion of resettable zero knowledge (rZK). This notion expresses an improvement over the classical notion of zero knowledge. Protocols under rZK remain secure even after a prover is reset to its initial state and re-uses the same random numbers. The resetting property in this notion is of particular importance to certain smart cards (e.g., smart cards that do not support atomic updates) due to their nature of being under the complete control of the adversary. A protocol that satisfies the rZK definitions is suitable for a cryptographic protocol that utilizes smart cards.

That notion while it may be interesting for the design of future protocols that target certain smart card types, it does not apply to any existing protocols that do not involve algorithms with strong guarantees such as (resettable) zero knowledge.

2.5 Our approach

In this paper we enhance Shoup’s simulatability-based static corruptions model [24] to account for threats in a typical key establishment protocol arising from smart card usage. Under this model we prove the modified PKINIT protocol secure. The notion of security in this model depends on an ideal system, with an ideal key exchange that is by definition secure and a real system that describes the actual protocol and participant interactions. A proof of security in this model shows that any attack on the real system can be simulated in the ideal system. We use this model because it is dedicated to key exchange, and, as such suffices for our purposes. Moreover, due to its composition properties [24], it is a suitable tool to study protocol components in isolation.

3 Overview of the DH variant of PKINIT

Kerberos is an authentication and key distribution protocol originally proposed in 1988, and has a long history of attacks and updates (see, for example, [2, 12] and the references therein). Its main goal is to establish fresh session keys between users and servers and, as a result, enable users to log into multiple servers that belong to a common infrastructure. To this end, a user first requests an electronic ‘ticket’ from a central Key Distribution Center (KDC). The ticket then enables authenticated users to log into a server that is part of the infrastructure.
The main characteristic of all Kerberos variants, as well as the main difference with other Internet security protocols like TLS or IPSec, is a single round-trip message exchange between the client and the KDC. This short message exchange is insufficient to provide any kind of freshness indication in the client message due to the lack of any prior input from the server, but it is handled by including a client-generated timestamp to ensure freshness (despite early arguments against that [6]). Replays in this exchange are prevented by requiring the server to store the client generated nonce during the validity time of the timestamp.

The public key Kerberos protocol

![Diagram of the public key Kerberos protocol](image)

**Figure 1:** The DH variant of public key Kerberos with smart card (simplified).

Fig. 1 provides a high-level overview of the DH of PKINIT as specified in [19, 27] and specifically its variant where each party generates a fresh pair of DH keys. When the user decides to log into a server (step 0 in the Figure) his terminal constructs an AS_REQ message as specified in [19] and then computes its hash value $h$. Then it chooses a DH group, randomly generates an ephemeral DH secret $x \in \mathbb{Z}_p$, and computes $g^x$, where $p$ is a large prime and $g$ is the generator of the group. In that process the terminal chooses the 32-bit nonces $n_1$, $n_2$ and stores the current time $\tau$. It then provides the values $g^x$, $n_1$, $n_2$, $\tau$ and $h$ to the smart card for signing (step 1). Note that, depending on the implementation, it may only provide a hash of these values to the smart card. If the smart card is PIN-enabled, then the user must provide his PIN prior to this operation. Depending on the implementation, the PIN is inserted either to the terminal or the smart card reader. Fig. 1 shows the case where the user provides his PIN to the terminal in step 0, and where this is sent to the card in step 1.

The signature $\sigma$, output by the smart card (step 2), is then used by the terminal
to construct an augmented version of the \texttt{AS\_REQ} message, which we denote by \texttt{AS\_REQ^∗}. This message, which contains $g^x, n_1, n_2, \tau, \sigma$ and \texttt{AS\_REQ} as a substructure, is sent to the KDC (step 3) which, among other things, verifies the signature. If verification succeeds, then the KDC chooses a random $y \in \mathbb{Z}_p$, computes $g^y$ and the ephemeral secret $K = F(g^{xy}, n_1)$, and constructs a response \texttt{AS\_REP}. This message contains a ticket which is encrypted with $K$. Finally, it augments this message with fields containing the values of $g^y$ and $n_1$ signed with server’s private key. The resulting message, denoted \texttt{AS\_REP^∗}, is sent to the terminal (step 4).

Using the value $g^y$ and its ephemeral secret $x$, the terminal recovers the key $K$ and is therefore able to decrypt the ticket. This ticket enables the terminal to complete the subsequent message exchange with the desired server (step 5). The details of this exchange are not relevant to the attack. A more detailed view of the key exchange is shown below.

\begin{align*}
\text{User} & \rightarrow \text{KDC} : \tau, \text{ID}_{\text{Server}}, n_1, n_2, g, p, g^x, \text{cert}_{\text{User}}, \\
& \quad \text{sig}_{\text{User}}(\tau, n_1, n_2, g, p, g^x, \text{ID}_{\text{KDC}}) \\
\text{KDC} & \rightarrow \text{User} : g^y, \text{ID}_{\text{User}}, \text{cert}_{\text{KDC}}, \text{sig}_{\text{KDC}}(g^y, n_1), \\
& \quad \{n_2, \text{ID}_{\text{Server}}, K_{\text{Server}}\} K
\end{align*}

where $K = F(g^{xy}, n_1)$ and $F$ is a hash function based on SHA1 and $K_{\text{Server}}$ is the key shared between the client and the server.

\section{Attack and fix}

This section revisits the attack on the PKINIT protocol as described in [17] and proposes a fix.

\subsection{The attack}

This protocol naturally lends itself to an implementation where the user’s private signing key is stored in a smart card and indeed, PKINIT is typically used with smart cards in the Microsoft Windows Active Directory [18, 14]. However, in the combined protocol, there are three types of players: users, servers and a central entity called the Key Distribution Center (KDC). A user is equipped with a terminal and a smart card. The card contains an asymmetric key pair and a certificate, signed by an authority, that binds the public key to the user’s identity. Moreover, the card provides an interface over which the terminal can ask the card to sign messages using the private key. Note that the card may require a PIN in order to respond to signature requests from the terminal.
Careful inspection of the smart card deployment documentation for Kerberos [18, 14] reveals a flaw, which arises due to the smart card introduction. This flaw leads to a relatively trivial attack, that enables an adversary with only temporary access to a victim’s smart card, to impersonate the victim even after the adversary’s access to the card is revoked. The attack is based on the observation that the KDC has no means to verify whether or not an incoming \texttt{AS\_REQ}\(^∗\) message is fresh. That is, while the KDC checks that the timestamp \(\tau\) indicates approximate current time, this does not guarantee that the \texttt{AS\_REQ}\(^∗\) message was constructed recently. In fact, the \texttt{AS\_REQ}\(^∗\) message could have been generated in the distant past.

In order to mount the attack, the adversary first obtains access to a victim’s smart card. This can be done either by compromising the victim’s terminal, or by stealing the card and its PIN. The adversary then fabricates an \texttt{AS\_REQ} message, calculates its hash \(h\), chooses nonces \(n_1, n_2\), a random DH secret \(x\), and a timestamp \(\tau\) indicating a particular future point in time. It then sends the values \(g^x, \tau, n_1, n_2, h\) to the card to obtain the signature \(\sigma\). Using this signature, the adversary constructs an \texttt{AS\_REQ}\(^∗\) message.

Note that this fabricated \texttt{AS\_REQ}\(^∗\) message will be accepted by the KDC as a genuine ticket request from the victim at time indicated by \(\tau\). Since neither the victim’s himself, nor his smart card is required in the remainder of the protocol, the message enables the adversary to impersonate the victim to the KDC at time \(\tau\). With the ticket in the KDC’ response, the adversary will further be able to impersonate the victim to the server of his choice until the ticket expires.

Note that, in order to be able to impersonate a victim at, say, approximately 20:00 of every Monday in a two-year period, the adversary must fabricate about 104 \texttt{AS\_REQ}\(^∗\) messages as described above and, for each such message, obtain a signature from the card. In other words, a few minutes of access to a victim’s card are sufficient for the adversary to be able to impersonate the victim, on a regular basis, for years.

### 4.2 The fix

A fix that defends against the attack at the cost of an additional message is described below. In this paper, we refer to the resulting fixed protocol as ‘modified PKINIT’. In modified PKINIT, the KDC first sends a nonce \(n_0\) to the terminal. This nonce is then added to the data signed by the smart card and included in \texttt{AS\_REQ}\(^∗\) (see step 1 in Fig. 1). On reception of the \texttt{AS\_REQ}\(^∗\) message, the KDC also ensures that the signature covers \(n_0\). Apart from the message from the KDC to the client that transports the nonce, and a change
to the $\text{AS}_\text{REQ}^*$ message to accommodate the additional nonce. The resulting protocol is shown below.

\[
\begin{align*}
\text{KDC} & \rightarrow \text{User} : n_0 \\
\text{User} & \rightarrow \text{KDC} : \tau, \text{ID}_{\text{Server}}, n_1, n_2, g, p, g^x, \text{cert}_{\text{User}}, \\
& \quad \text{sig}_{\text{User}}(\tau, n_0, n_1, n_2, g, p, g^x, \text{ID}_{\text{KDC}}) \\
\text{KDC} & \rightarrow \text{User} : g^y, \text{cert}_{\text{KDC}}, \text{sig}_{\text{KDC}}(\text{ID}_{\text{User}}, g^x, g^y, n_1), \\
& \quad \{n_2, \text{ID}_{\text{Server}}, K_{\text{Server}}\}
\end{align*}
\]

where $K = F(g^{xy}, n_1)$ and $F$ is a hash function based on SHA1.

Note however, that the fact that the protocol’s initial message is sent by the server does not imply that this is no longer a client-initiated protocol. There are examples of client-initiated where the server is sending the first session message, e.g., the SSH [26] protocol. Of course depending on the underlying transport layer an additional initial client message may be required to initiate the session.

An issue that should be noted in the fix presented here, is the addition of $g^x$ and $\text{ID}_{\text{User}}$ values to the signature value of the KDC. The original signature only contained the public parameter of the server and the client’s nonce, making it applicable to a variety of sessions unrelated to this one that may share the same nonce (not an unlikely scenario given that this nonce is only 32-bits long). The lack of these values was first noted by Roy et al. [20] who nevertheless proved the security of the original scheme on the random oracle model based on the fact that the function $F$ is used to derive the shared keys. However, we believe that the disassociation of a server-generated signature with the session that it was intended to be used, is a bad practice [1]. For this reason, we include the values in our modified version (a fact that also simplifies our later proofs).

5 A model for static corruptions with smart cards

This section describes the security model in which we analyze PKINIT and the fix. Our model is an extension of Shoup’s model for session key establishment protocols [24] to the smart card setting. Note that we choose to extend the ‘static corruptions’ variant of Shoup’s model, and that we omit certain options that are not relevant for our arguments in this paper. For a complete description of all variants of Shoup’s model the reader is referred to [24].
The security model is based on simulatability of the real world within an ideal world. That is, protocols that are secure in this model behave ‘as if’ the interactions between participates would take place in the ideal world. This ideal world is part of the model’s definition and captures what it means for the protocol to be secure.

More specifically, security is defined by means of a ‘game’, i.e., a series of interactions, between an adversary and a ‘ring master’. The purpose of this game, which takes place either in the ‘real world’ or in the ‘ideal world’, is the generation of a transcript. In the ideal world, the interactions between the adversary and the ring master cause protocol instances to be initialized, session keys to be generated and assigned to these instances, keys to be revealed, etc. However, in the ideal world, these protocol instances are ‘virtual’ because no security protocol is ever executed (and hence no protocol messages are ever exchanged); session keys are instead generated by the ring master, and information about these keys can be leaked to the adversary exclusively over well-defined interactions.

In the real world, the adversary also interacts with the ring master. In this world, however, the ring master controls ‘real’ protocol participants that execute the actual protocol. Session keys are indeed established by real protocol instances, and the only way for the adversary to drive these instances, is by means of sending protocol messages to the ring master. These messages are forwarded by the ring master to the protocol instances chosen by the adversary, and responses are forwarded to the adversary.

In both worlds, the adversary causes protocol events to occur, i.e., session keys to be established, sessions to be aborted, keys to be revealed, etc. These events are written to a transcript. The security definition requires that all real-world adversaries are ‘simulatable’ in the ideal world. That is, adversaries that interact, through the ring master, with real protocol participants, must not be able to cause a series of events that is impossible to be caused by an adversary that operates in ideal world which, by definition, is secure. Whether or not this ‘simulatability’ requirement is fulfilled, is verified by means of a ‘distinguisher’. This distinguisher is given a transcript that originates from an adversary/ring master game, and must decide whether the game took place in the ideal or the real world. Only if the distinguisher has no advantage over random guessing is the protocol considered secure.

5.1 The ideal world

As explained above, in the ideal world, the adversary interacts with the ring master by means of different queries, defined below. During this interaction,
the ring master creates state for ‘users’ and ‘user instances’, denoted by $U_i$ and $I_{i,j}$, respectively, where $i, j \in \mathbb{N}$. The state associated with users and user instances is generated during this interaction. While the state associated with users models participants that may execute a given session key establishment protocol, the state associated with user instances models individual executions the protocol. User instances must be ‘initialized’ before they can establish session keys. The ring master associates with every initialized instance a particular state; possible states are active, isolated and finished. The ring master may also reject certain queries; whenever this happens, the adversary has violated some consistency constraint which renders the particular query to be illegal.

The purpose of the adversary’s interaction with the ring master is the generation of a transcript, i.e., a sequence of entries. Initially, the transcript is empty. The queries the adversary may issue during the game are as follows.

- **InitUser($i, ID$)**: On reception of this query where $ID$ is a bit string, the ring master rejects the query if there exists an $(\text{InitUser}(i, \cdot))$ entry in the transcript. Otherwise, it assigns the identifier $ID$ to $U_i$ and appends the entry $(\text{InitUser}, i, ID)$ to the transcript. No information is returned to the adversary.

- **InitUserInstance($i, j, role, PID$)**: On reception of this query where $role \in \{0, 1\}$ and $PID$ is a bit string, the ring master rejects the query if the transcript does not contain any $(\text{InitUser}, i, \cdot)$ entry, or if it contains an $(\text{InitUserInstance}, i, j, \cdot, \cdot)$ entry. Otherwise, it sets the instance to be active, and assigns to it the role $role$ and the partner identifier $PID$ to the instance. This partner identifier is to be seen as the identifier of the user with which this instance shall expect to establish a session key. The ring master then appends the entry $(\text{InitUserInstance}, i, j, role, PID)$ to the transcript, and no information is returned to the adversary. Note that the role $role$ breaks symmetry and can be seen as an indicator of whether the instance should behave as a client or server (equivalently, as an initiator or a responder).

**Remark 1:** Two instances may be ‘compatible’ with respect to a transcript. Informally, we say that instances are ‘compatible’ if they expect to establish a session key with each other. Formally, two instances $I_{i,j}$ and $I_{i',j'}$ are said to be compatible with respect to a transcript, if there exist entries $(\text{InitUser}, i, ID)$, $(\text{InitUser}, i', ID')$, $(\text{InitUserInstance}, i, j, role, PID)$, and $(\text{InitUserInstance}, i', j', role', PID')$ such that $role \neq role'$, $ID = PID'$, and $ID' = PID$.

- **AccessSC($i$)**: On reception of this query, the ring master rejects the query if the transcript does not contain any $(\text{InitUser}, i, \cdot)$ entry. Otherwise, it appends the entry $(\text{AccessSC}, i)$ to the transcript.
Remark 2: Note that AccessSC(i) is only a placeholder in the ideal world. The idea behind it is to simulate unauthorized access to smart cards, and its purpose is to enable the compromise mode in the Start query (see below).

- **Abort(i, j):** On reception of this query, the ring master rejects the query if there exists no (InitUserInstance, i, j, ·, ·) entry in the transcript. Otherwise, it appends the entry (Abort, i, j) to the transcript and sets $I_{i,j}$ to be finished. No information is returned to the adversary.

- **Start(i, j, mode[.key]):** On reception of this query, where mode $\in \{create, connect(i', j'), compromise\}$ and key is a bit string, the ring master rejects the query if there exists no (InitUserInstance, i, j, ·, ·) entry in the transcript, or if $I_{i,j}$ is finished. Otherwise, it proceeds as follows.

  - If mode = create, then the ring master generates a key $K_{i,j}$ uniformly at random, assigns it to $I_{i,j}$, sets $I_{i,j}$ to be isolated, and appends the entry (Start, i, j) to the transcript. No information is returned to the adversary.

  - If mode = connect(i', j'), then the ring master rejects the query if the two instances $I_{i,j}$ and $I_{i',j'}$ are not compatible. The ring master further rejects the query if the instance $I_{i',j'}$ is not isolated. The ring master then assigns the key $K_{i',j'}$, which was previously assigned to $I_{i',j'}$, to the instance $I_{i,j}$. Finally, the ring master sets both $I_{i,j}$ and $I_{i',j'}$ to be finished, and appends the entry (Start, i, j) to the transcript. No information is returned to the adversary.

  - If mode = compromise, then the adversary must also specify the parameter key. On reception of this query, the ring master rejects the query if there exists no (InitUserInstance, i, j, ·, ·) entry in the transcript. Otherwise, it selects this entry. The ring master then checks whether or not $I_{i,j}$ has compatible instances. If it has, then the ring master checks whether or not the first (InitUserInstance, i', j', ·, ·) entry, where $I_{i',j'}$ is a compatible instance, appears before the selected entry. If it does, then the ring master deselects the (InitUserInstance, i, j, ·, PID) entry selected above and selects this (InitUserInstance, i', j', ·, ·) entry instead. The ring master then checks whether or not the peer’s the smart card was illegitimately accessed after the instance that was initialized first, was initialized. That is, the ring master checks whether or not the transcript contains an (AccessSC, i') entry after the entry selected above. If not, then the ring master rejects the query.

  Finally, the ring master then checks whether or not the identifier PID corresponds to an initialized user, i.e., if there exists an (InitUser, ·, ·, ·).
such that $PID = ID'$; if it does, then the ring master also rejects the query. If the query is not rejected, then the ring master assigns the key $key$ to the instance $I_{i,j}$, sets the instance to be finished, and appends the entry $(Start, i, j)$ to the transcript. No information is returned to the adversary.

**Remark 3:** In simpler words the ring master accepts the $Start$ with $mode = compromise$ query only when the peer is not an assigned user or if $AccessSC$ has been issued on the peer after any of the current or the peer’s instance have been initialized.

- $Application(f)$: On reception of this query, where $f$ is the description of a function that takes as parameters a string and a set of keys, the ring master evaluates $f$ on input the string $R$ and the set of keys $\{K_{i,j}\}$ that have been assigned to instances during the game so far. The ring master appends the entry $(Application, f, f(R, \{K_{i,j}\}))$ to the transcript and returns the value $f(R, \{K_{i,j}\})$ to the adversary.

- $Implementation(comment)$: On reception of this query, where $comment$ is a bit string, the ring master appends the entry $(Implementation, comment)$ to the transcript. No information is returned to the adversary.

**Remark 4:** To capture the threats of adversaries with temporary access to smart cards, the capabilities of the ideal world adversary are extended compared to the ‘static corruptions’ variant of Shoup’s model [24]. The idea of our extension is that the adversary may explicitly access a participant’s smart card by means of the $AccessSC$ query. An adversary that causes an ongoing peer session to accept by doing so after the session was initialized, is considered benign in our model. This follows the intuition that, as long as an adversary has illegitimate access to a victim’s smart card, it is no surprise if the victim can be impersonated. An adversary that causes a peer instance to accept without accessing the participant’s smart card while their session is ongoing, on the other hand, this considered malicious. The distinction between these two adversary types is captured by the consistency constraints that the ring master enforces on reception of a $Start(\cdot, \cdot, compromise, \cdot)$ query. Note that modifying the constraints that apply to ‘compromise’ queries follows the general approach of the Shoup-Rubin model [25].

### 5.2 The real world

Unlike in the ideal world, users and user instances are not simply placeholders in the real world. Instead, users correspond to real participants in a given session key establishment protocol, and are given long-term cryptographic keys and smart cards. User instances are probabilistic Turing machines that are
activated for a particular execution of the protocol. As such, they accept messages and, for each incoming message, they output another message as well as a status indication from the following set.

- **continue**: The instance expects at least one more message.
- **accept**: The instance is finished and a session key has been established.
- **reject**: The user instance is finished without having established a session key.

Apart from users and user instances, in the real world there also exists a trusted third party (TTP). This TTP may operate off-line and has a public/private key pair. Users are required to register with this TTP before their instances can establish session keys. As in the ideal world, an adversary interacts with the ring master for the purposes of generating a transcript. The queries the adversary may issue to the ring master, are as follows.

- **InitUser**(*i, ID*)**: On reception of this query where *ID* is a bit string, the ring master rejects the query if there exists an (InitUser, *i, ·*) or an (Implementation, register, ·, ·, ·) entry in the transcript. Otherwise, *U_i* is registered with the TTP, his long term state is generated and his smart card is personalized and given to him. The ring master appends the entry (InitUser, *i, ID*) to the transcript, and no information is returned to the adversary.

- **Register**(*i, ID, request*): On reception of this query where *ID* and *request* are bit strings, the ring master rejects the query if there exists an (InitUser, *i, ·*) or an (Implementation, register, ·, ·, ·) entry in the transcript. Otherwise, it forwards *request* to the TTP, which registers the identity *ID*, and generates a response *response*. The ring master then appends the entry (Implementation, register, *ID*, request, *response*) to the transcript, and returns *response* to the adversary.

- **InitUserInstance**(*i, j, role, PID*): On reception of this query where *role* ∈ {0, 1} and *PID* is a bit string, the ring master rejects the query if the transcript does not contain any (InitUser, *i, ·*) entry, or if it contains an (InitUserInstance, *i, j, ·, ·, ·*) entry. Otherwise, it sets the instance to be active, assigns to it the role *role* and the partner identifier *PID* to the instance, and appends the entry (InitUserInstance, *i, j, role, PID*) to the transcript. No information is returned to the adversary.

- **DeliverMessage**(*i, j, inMsg*): On reception of this query where *inMsg* is a bit string, the ring master rejects the query if the transcript does not contain any (InitUserInstance, *i, j, ·, ·*) entry. Otherwise, it sends the message *inMsg* to instance *I_i,j*. After processing this message, the instance outputs
the message \textit{outMsg} and the status \( \tau \). The ring master appends the entry \((\text{Implementation}, \text{deliverMsg}, i, j, \text{inMsg}, \text{outMsg}, \tau)\) to the transcript. If \( \tau \neq \text{continue} \), the ring master sets \( I_{i,j} \) to be \textit{finished}. If \( \tau = \text{accept} \), the ring master also appends the entry \((\text{Start}, i, j)\) to the transcript; if \( \tau = \text{reject} \), it appends the entry \((\text{Abort}, i, j)\) instead. Finally, the ring master returns \textit{outMsg} to the adversary.

- \textit{AccessSC}(i, \text{request}): On reception of this query where \textit{request} is a bit string, the ring master rejects the query if the transcript does not contain any \((\text{InitUser}, i, \cdot)\) entry. Otherwise, it sends \textit{request} to \( U_i \)'s smart card. After processing this message, the smart card outputs the response \textit{response}. The ring master appends the entries \((\text{Implementation}, \text{accessSC}, i, \text{request}, \text{response})\) and \((\text{AccessSC}, i)\) to the transcript. Finally, the ring master returns \textit{response} to the adversary.

- \textit{Application}(f): On reception of this query, the ring master proceeds as in the ideal world, with the exception that the set of keys \( \{K_{i,j}\} \) refer to the keys that were actually established by the protocol.

Note that there exists no \textit{Implementation} query in the real world. We are now ready to state the soundness and security definitions.

\textbf{Definition 1.} A smart card-based protocol is efficient and sound if real world user instances terminate after a polynomially bounded number of incoming messages, and, whenever the adversary faithfully delivers the generated messages between two compatible user instances, these instances accept and share the same session key.

\textbf{Definition 2.} A smart card-based protocol is secure if for every efficient real world adversary, there exists an ideal world adversary that generates a computationally indistinguishable transcript.

\textbf{Remark 5:} In order to maintain the generality of our model, we did not introduce entities in the real world definition that are specific to PKINIT. While the key distribution server (KDC) is such an entity, it is handled just as another user that may also possess a smart card. While it is not mandatory that all users have smart cards, this approach naturally covers situations where the KDC stores its keys in a hardware security module (which operates similarly to a smart card).

\textbf{Remark 6:} Note that even though the PKINIT protocol has the notion of time and timestamps, they are not needed in the model. The “time” in the real or ideal world can be assumed to be given by a numeration of the transcript.
6 Results

This section presents our results, namely that PKINIT is not secure in the model described in Section 5, and that the fix proposed in [17] is. Our proof assumes that the Computational Diffie-Hellman (CDH) problem is hard, that the used signature scheme is unforgeable, and the random oracle model.

**Theorem 1.** PKINIT with smart cards is not secure.

**Proof.** This proof is an application of the distinguishing approach described in point 5 of section 3.4 in [24]. Consider the real world transcript shown in Fig. 2. According to entries 1 and 2, the adversary first initializes two users, 10 and 20, with identifiers ‘Alice’ and ‘KDC’, respectively. The adversary then accesses Alice’s smart card in order to obtain the signature $\text{signature}$ on the structure $\text{signedData}$. The adversary chooses $\text{signedData}$ such that the concatenation $\text{signedData} | \text{signature}$ yields the fabricated $\text{AS}_{\text{REQ}}^*$ message described in Section 4.1. The adversary afterwards initializes an instance 200 for the KDC (see entry 5). Subsequently, the adversary sends the fabricated $\text{AS}_{\text{REQ}}^*$ message to KDC (see entry 6). Since this message is identical to a genuine message from Alice, the KDC accepts and establishes the session key $K_{20,200}$ (see entry 7). In entry 8, the adversary forces his guess of this session key into the transcript and then it asks the ring master to dump all established keys to the transcript. This results in the single established key $K_{20,200}$ to be written in entry 9.

We now describe a distinguisher that is able to determine, with non-negligible probability, whether a transcript of the above form originates from the real or the ideal world. The distinguisher simply indicates ‘real world’ if and only if the value of $\text{guess}$ in entry 8 equals the session key $K_{i,j}$ reported in entry 9. This distinguisher is actually correct except with negligible probability. In order to see this, observe that, as explained in Section 4.1, the real world adversary knows the session key established by the KDC’s instance. Therefore, with certainty, it holds that $\text{guess} = K_{i,j}$. There exist only two potential approaches to construct an ideal world transcript of this form where $\text{guess} = K_{i,j}$ holds; the first approach is by random guessing and this approach works with negligible probability. The other approach is to compromise the KDC’s instance. Issuing a $\text{Start}(20, 200, \text{compromise}, \text{key})$ query such that a $(\text{Start}, 20, 200)$ entry appears on position 8 in the transcript. This is, however, impossible for the ideal world adversary because this query is rejected unless either (a) the user identifier ‘Alice’ is not assigned to any user, or (b) Alice’s smart card is accessed after the instance $I_{20,200}$, or any of its compatible instances, is initialized, and neither of these is true in this transcript.

**Theorem 2.** Modified PKINIT with smart cards is secure in the random oracle model, assuming that the Computational Diffie-Hellman (CDH) problem is
hard with a security parameter $k_d$, the signature scheme is unforgeable with a security parameter $k_s$, signing keys exist only on smart cards, and nonces\(^2\) are sufficiently long and random with a security parameter $k_n$.

Before proceeding with the proof, we restate the definition of the modified PKINIT protocol.

\[ U_i' \rightarrow U_i : n_0 \]
\[ U_i \rightarrow U_i' : \tau, \text{ID}_\text{Server}, n_1, n_2, g, p, g^x, \text{cert}_i, \text{sig}_i(\tau, n_0, n_1, n_2, g, p, g^x, \text{ID}_i') \]
\[ U_i' \rightarrow U_i : g^y, \text{ID}_i, \text{cert}_i', \text{sig}_i'(\text{ID}_i, g^x, g^y, n_1), \{n_2, \text{ID}_\text{Server}, K_\text{Server}\} K \]

**Proof.** We describe a simulator that transforms any real-world adversary into an ideal-world adversary such that no distinguisher has non-negligible advantage in determining whether a transcript originates from the interaction of the real-world adversary or our simulator.

Our simulator simulates real-world users and protocol instances towards the real-world adversary, and responds to queries of the form InitUser, Register, InitUserInstance, DeliverMessage, AccessSC and Application correspondingly. Towards the ideal-world ring master, our simulator behaves as follows. InitUser, Register, InitUserInstance, AccessSC, and Application queries are forwarded to the ideal-world ring master without the superfluous parameters. Register(·, ·, ·)

\(^2\)It is, however, important to note that while the security parameters $k_s$ and $k_d$ are controlled by the user of the protocol by selecting longer key sizes and DH groups, the lengths of nonces are fixed in the protocol to 32-bits.
queries are forwarded as \((\text{Implementation}, \text{register}, \cdot, \cdot, \text{response})\) queries, where \text{response} is generated by our simulator. \text{DeliverMessage} queries are forwarded as \((\text{Implementation}, \text{deliverMsg}, i, \cdot, \cdot, \text{inMsg}, \text{outMsg}, \tau)\) queries, where \text{inMsg} denotes the real-world adversary's message, and \text{outMsg} and \tau denote the response message and status generated by the simulator's simulation of user \(U_i\), respectively. Moreover, if an \((\text{Abort}, i, j)\) entry appears in the real-world transcript, our simulator issues an \text{Abort}(i, j) query in the ideal world.

In order to cause the appearance of \((\text{Start}, i, j)\) entries in the correct positions of the ideal-world transcript while ensuring that the sets of established session keys remain computationally indistinguishable in both worlds (this requirement is important because otherwise a distinguisher can gain an advantage by examining the output of \text{Application} queries), our simulator proceeds as follows. Suppose a \((\text{Start}, i, j)\) entry appears on the real-world transcript, indicating that instance \(I_{i,j}\) just received the last message it expected, accepted, and established a session key.

**Case A:** If \(I_{i,j}\)'s partner identifier \(PID\) does not correspond to an initialized user (if this holds in the real world transcript, it does so in the ideal world transcript as well), then our simulator issues a \text{Start}(i, j, \text{compromise}, \text{key}) query, where \text{key} is the actual key established by \(I_{i,j}\) in the real world. Note that this query results in the required \((\text{Start}, i, j)\) transcript entry. Moreover, it ensures that the key established by \(I_{i,j}\) in the ideal world is identical to its real-world counterpart.

**Case B:** Otherwise, i.e., if there exists an \((\text{InitUser}, i', ID')\) entry in the transcript such that \(ID' = PID\), then our simulator checks whether or not there exists an \((\text{AccessSC}, i')\) entry in the transcript after the \((\text{InitUserInstance}, i, j, \text{role}, PID)\) entry.

**Case B.1:** If such an entry exists, then our simulator issues a \text{Start}(i, j, \text{compromise}, \text{key}) query, where \text{key} is the actual key established by \(I_{i,j}\) in the real world. This query results in the required \((\text{Start}, i, j)\) transcript entry. In effect, this means that the adversary accessed the smart card after \(n_0\) was generated and sent and thus is able to compromise the session.

**Case B.2:** Otherwise, i.e., if such an \((\text{AccessSC}, i')\) entry does not exist, then observe that, at least one instance \(I_{i',j'}\) that is compatible with \(I_{i,j}\) exists, except with negligible probability. That is, the nonce \(n_0\) was sent before any “malicious" \text{AccessSC} by the adversary and the values \(g^x, g^y\) and the ID match. This follows from the structure of modified PKINIT, the assumption that the signature scheme is unforgeable, and the assumption that nonces are sufficiently long. Our simulator checks whether or not there exists an \((\text{AccessSC}, i')\) entry in the transcript after the \((\text{InitUserInstance}, i', j', \text{role'}, PID')\) entry that corresponds to the first compatible peer instance \(I_{i',j'}\).
Case B.2.1: If such an entry exists, then our simulator issues a \textit{Start}(i, j, \text{compromise}, \text{key}) query, where \text{key} is the actual key established by \text{I}_{i,j} in the real world. This results in the required \text{(Start, i, j)} transcript entry.

Case B.2.2: Otherwise, i.e., if such an \text{(AccessSC, i′)} entry does not exist, then our simulator checks whether or not at least one of \text{I}_{i,j}'s compatible instances has accepted (and therefore are \textit{isolated} in the ideal world).

Case B.2.2.1: If none of the compatible instances has accepted, then our simulator issues a \textit{Start}(i, j, \text{create}) query. As a result, a random key \text{K}_{i,j} is assigned to \text{I}_{i,j} in the ideal world. Although this key is almost certainly not identical to its real world counterpart, it is computationally indistinguishable from it in the random oracle model (see section 5.3.3 in [24]).

Case B.2.2.2: If at least one of the compatible instances has accepted, then our simulator selects one of these real world instances, denoted \text{I}_{i′,j′}, by extracting the keys from all of them, and comparing it to the key established by \text{I}_{i,j}. If a match is found, then it issues a \text{Start}(i, j, \text{connect}(i′, j′)) query. Note that this results in the required \text{(Start, i, j)} transcript entry, and causes the keys \text{K}_{i,j} and \text{K}_{i′,j′} to be identical in the ideal world, just as they are in the real world. Our simulator therefore preserves the computational indistinguishability of the key sets and hence the transcripts. Note that, except with negligible probability, our simulator can always find a matching peer \textit{isolated} instance and that this instance is unique. This follows from the construction of the PKINIT protocol, the assumed security of the signatures, and assuming that nonces are sufficiently long. If, for example, the adversary had illegitimately accessed the peer’s smart card, then its \text{AccessSC}(i′, ·) queries must have been issued \textit{before} the \text{InitUserInstance}(i′, j′, \text{role′, PID′}) query because otherwise our simulator would have branched to Case B.2.1. Hence, except with negligible probability, the adversary must have faithfully forwarded protocol messages between \text{I}_{i,j} and \text{I}_{i′,j′} and that, as a result, their keys \text{K}_{i,j} and \text{K}_{i′,j′} are identical. \hfill \Box

7 Conclusions

In this paper we extended Shoup’s security model to include threats in the smart card setting, and examined the DH variant of the public key Kerberos protocol (PKINIT) in the new model. The new model incorporates smart card

---

In the original PKINIT protocol, this assertion does not hold. In particular, using the attack described in Section 4.1, an adversary can cause \text{I}_{i,j} to establish a key that is distinguishable from a random one; more precisely, the adversary knows the exact value of that key, as shown in Theorem 1. Moreover, since this adversary does not issue an \text{AccessSC}(i′) query after the \text{(InitUser, i′, ID)}, our simulator cannot simulate this adversary as this would require issuing an illegal \text{compromise} query.
threats such as temporary access and terminal tampering, and is shown to capture the attack described in [17]. Furthermore, we show that the PKINIT protocol fix of [17] is secure in the that model.

The new model’s applicability is not restricted to the Kerberos protocol and applies to any protocol that can be expressed in Shoup’s model [24]. Given the broad scope, and also the simplicity behind the original model, we believe that the smart card version of the model would be a practical tool that can be used to verify other real world smart card-based protocols.

What the proof of security of the modified PKINIT protocol achieves, is to transform the intuition of security in the smart card setting due to the additional nonce, to a formal assurance of security. On the other hand, a formal assurance of security may not constitute a sufficient reason to introduce a protocol change, which is not trivial to deploy. We believe, however, that given the fact that main use-case of the PKINIT protocol in Microsoft Windows Active Directory is in combination with smart cards, the modification is a reasonable trade-off to ensure security in the smart card setting.

References


5.5 A cross-protocol attack on the TLS protocol

Publication data


Contributions

Principal author.
A Cross-Protocol Attack on the TLS Protocol

Nikos Mavrogiannopoulos*      Frederic Vercauteren*
Vesselin Velichkov†        Bart Preneel*

October 2012

Abstract
This paper describes a cross-protocol attack on all versions of TLS; it can be seen as an extension of the Wagner and Schneier attack on SSL 3.0. The attack presents valid explicit elliptic curve Diffie-Hellman parameters signed by a server to a client that incorrectly interprets these parameters as valid plain Diffie-Hellman parameters. Our attack enables an adversary to successfully impersonate a server to a random client after obtaining $2^{40}$ signed elliptic curve keys from the original server. While attacking a specific client is improbable due to the high number of signed keys required during the lifetime of one TLS handshake, it is not completely unrealistic for a setting where the server has high computational power and the attacker contents itself with recovering one out of many session keys. We remark that popular open-source server implementations are not susceptible to this attack, since they typically do not support the explicit curve option. Finally we propose a fix that renders the protocol immune to this family of cross-protocol attacks.

1 Introduction

The TLS protocol TLS is one of the major secure communications protocols on the Internet, used by a variety of applications such as web browsers, electronic mail, voice over-IP and more. The protocol derives from Netscape’s SSL 3.0 [14], and is the joint attempt, under the umbrella of IETF, to create a secure protocol for e-commerce. The first version of the protocol, TLS 1.0, fixed the known issues [4] in SSL 3.0 and introduced HMAC [17]. TLS 1.1 followed to address known attacks in CBC encryption mode [2, 26] and RSA [16].

*Katholieke Universiteit Leuven, ESAT/SCD/COSIC.
†University of Luxembourg.
Today the latest version is TLS 1.2 [11] but since none of the known weaknesses are classified as major, older versions of the protocol including SSL 3.0 are still in use.

The TLS protocol is an agile protocol that allows peers to negotiate their highest supported protocol version, as well as the combination of ciphers used in a session. That combination is called a ciphersuite; it typically determines the symmetric encryption cipher with its operational mode, the key exchange method and the message authentication algorithm. The various versions of the protocol added new ciphersuites, deprecated old ones, or kept the same set of the previous version.

Not all available ciphersuites in TLS are equally strong: the most prominent example were the ones marked as export ciphersuites. When strong cryptography was not allowed to be exported from USA, major TLS-enabled web browsers that originated in USA included ciphersuites known to be weak. These supported the RSA-EXPORT key exchange method, which used 512-bit RSA keys, and was combined with 40-bit or 56-bit symmetric encryption. Fortunately they were deprecated since TLS 1.1 in 2006 [10].

As a result of the various ciphersuites available in the protocol, a typical implementation includes several algorithms offering similar functionality. For example an implementation may support the Diffie-Hellman (DH) key exchange algorithm, as well as elliptic curve Diffie-Hellman (ECDH). This fact is exploited in our attack by taking advantage of interactions between the different ciphersuites. In particular we exploit the possibility that a client interprets signed ECDH key exchange parameters as plain DH parameters.

The Wagner and Schneier attack Wagner and Schneier describe in [27] a server impersonation attack on the SSL 3.0 [14] protocol. Although this attack turned out to be impossible in practice due to an incorrect interpretation of the protocol, the underlying idea is still worth recalling. The attack transforms a server into an oracle that signs messages submitted by the adversary. In particular the server is used by the adversary to sign DH parameters, which are presented to the client as RSA parameters. This allows the recovery of the client’s secret by the adversary and eventually to the establishment of a secure session between the adversary and the client. In that session the client is convinced that the adversary is the server he intended to connect to. We will use the term cross-protocol attack to describe this attack, as well as the family of attacks that rely on interactions between distinct key exchange methods. The term multi-protocol attack is also used in the literature to describe this family of attacks [7].

Even if the Wagner and Schneier attack turned out to be impossible to imple-
ment, it demonstrates that the TLS protocol violates the following principle set forth by Anderson and Needham in [1].

**Principle 3:** Be careful when signing or decrypting data that you never let yourself be used as an oracle by your opponent.

This weakness was ignored, possibly, because the only published attack could not be implemented, and required the client to request the deliberately weakened RSA-EXPORT key exchange method.

**Our attack** As the protocol evolved and various other key exchange methods such as SRP, PSK or ECDH [3, 13, 25] were added, the fact that the server can be used as an oracle becomes relevant again. In this paper we re-examine the Wagner and Schneier attack in the context of the latest TLS protocol version [11] and describe a new cross-protocol attack. The attack uses the interactions between DH and ECDH key exchanges. It is based both on the ability to transform a TLS server into an oracle that provides signed parameters, and on TLS implementations blindly trusting those signed parameters.

Our contributions in this paper are as follows.

- To our knowledge, our attack is the first server impersonation attack on the TLS protocol with complexity much lower than a cryptanalytic attack on the cryptographic primitives used;
- Our attack highlights a much larger family of cross-protocol attacks that the TLS protocol is vulnerable to, which was previously ignored;
- We show that although basic checks on DH protocol parameters help to mitigate simple attacks, they are not sufficient to completely protect the protocol.

**The adversary** The adversary in both attacks is a Dolev-Yao adversary [12], that has full control over the network communications.

**Paper organization** In Section 2 we present the Wagner and Schneier attack on SSL 3.0 and the incorrect assumption that renders the attack impossible. Then in Section 3 we present our attack on the protocol, and in Section 4 we discuss the impact of the attack on several implementations. Section 5 provides a simulation of the attack in a real world scenario, and in Section 6 we propose a fix that makes TLS immune to this family of attacks. Finally Section 7 concludes the paper.


Terminology This document assumes familiarity with the TLS protocol [11] and adopts its terminology. Furthermore, when we refer to Diffie-Hellman key exchange we denote with $g$ the generator of the multiplicative group modulo $p$, and with $Y_s$ and $Y_c$ the public values of the server and the client. The explicit elliptic curves supported by TLS are given by a Weierstrass equation of the form

$$y^2 = x^3 + ax + b \mod q.$$ 

Note that we use $q$ to denote the ECDH prime to distinguish it from the plain DH prime $p$. The coefficients $a$ and $b$ are the curve parameters represented as integers modulo $q$. The protocol works in a group generated by a base point $P$ (simply called base in the remainder of the paper). The cofactor is defined as the order of the curve (i.e. the number of points on the curve) divided by the order of the base point. A public ECDH share is of the form $Q = [k]P = (X, Y)$, with $k$ the private key, $Q$ the elliptic curve point obtained by scalar multiplication of $P$ by $k$ and $X$ (resp. $Y$) the $x$ (resp. $y$) coordinate of $Q$.

2 The Wagner and Schneier attack

Wagner and Schneier in [27] describe a cross-protocol attack (the authors refer to it as “key exchange algorithm rollback attack”) based on the observation that the digital signature in a DH key exchange does not cover any identifier of the negotiated ciphersuite. According to the SSL 3.0 protocol [14] when a DH key exchange has been negotiated, the group parameters and key exchange data are sent by the server in the ‘ServerKeyExchange’ message as shown in Fig. 1a. The signature on that message is calculated on the algorithm parameters, and the nonces exchanged by both peers. The crucial observation is that the negotiated key exchange method is not part of this signature.

This omission allows an adversary to re-use a signed ‘ServerKeyExchange’ packet in another session, with another key exchange method, by initiating a parallel connection to the server. The attack deceives a client who advertises a ‘TLS_RSA_EXPORT’ ciphersuite and expects temporary RSA parameters in the ‘ServerKeyExchange’ message, into receiving DH parameters from a ‘TLS_DHE_RSA’ ciphersuite. Note that, the RSA-EXPORT key exchange requires the server to generate a temporary 512-bit long RSA key pair and include it in the ‘ServerKeyExchange’ message. In both DH and RSA-EXPORT the parameters are signed using the RSA algorithm.

The attack assumes that the client reads and verifies the signature, and then reads the RSA parameters (see Fig. 1b) one by one, yielding the following
Figure 1: The contents of the ServerKeyExchange message in Diffie-Hellman and RSA-EXPORT key exchange methods. Each row represents a 2-byte (16-bit) field, unless the length is explicitly given. All indicated lengths are in bytes.
The Wagner and Schneier attack

Client

Adversary

Server

Figure 2: A message sequence chart describing the Wagner and Schneier attack on SSL 3.0. The dashed lines indicate a forwarded message.
....scenario. The client verifies the signature, reads the RSA modulus \( m \), which corresponds to the prime of the DH group \( p \), and then reads the RSA exponent \( e \) field which corresponds to the group generator \( g \). Therefore, the client encrypts the pre-master secret \( k \) as \( k^g \mod p \) and includes it in its ‘ClientKeyExchange’ message. Since \( p \) is a prime number and \( g \) is known, it is very easy to compute the \( g \)-th root of \( k^g \) to recover \( k \), which allows the adversary to impersonate the server. Note that the ‘Finished’ messages that provide handshake message modification detection using message hashes encrypted and authenticated with the session keys, cannot detect this attack since the adversary recovers the pre-master secret.

The authors noticed that the SSLRef 3.0b1 implementation was immune to the attack and attributed the failure to a paranoid sanity check of this particular implementation. However, tests we performed on other implementations which did not include such sanity checks also failed. Careful examination of the TLS packet parsing reveals that the failure of the attack is due to the serialized way TLS packets need to be parsed. The variable length vectors [14] used in the structure definition in Fig. 1 require an implementation to read the vector length before reading data, hence an implementation can only start parsing the packet serially, i.e., from start to end without being able to read a field before reading the previous one. In the RSA case, a client would need to read the modulus length, then the modulus, and the same for the exponent and signature fields. If the DH ‘ServerKeyExchange’ packet, which contains one additional field, is substituted, that field will be read instead of the signature and verification fails.

Even though the Wagner and Schneier attack fails, it demonstrates the idea of a cross-protocol attack utilizing two of the SSL 3.0 key exchange methods, the DH key exchange and the RSA-EXPORT key exchange.

3 A new cross-protocol attack

Since version 1.0 [9] the TLS protocol has been augmented with various other key exchange methods such as SRP, PSK or ECDH [3, 13, 25]. In this section we present a server impersonation attack on clients that support the DH key exchange and wish to connect to a server that supports, among others, the ECDH method.

In order to support the ECDH key exchange method, the ‘ServerKeyExchange’ message was augmented in [3] to allow for several elliptic curve ciphersuites supporting multiple sub-options. The sub-option relevant to this paper is the representation of the elliptic curve used. It allows for explicit prime curves, ex-
plicit curves of characteristic 2, or named curves. Depending on the negotiated
ciphersuite the structure containing the selected curve parameters is signed by
the server using an RSA or ECDSA key. The attack we present requires the
server to support the explicit prime curve option, and the client to support the
plain DH method. Because the only common signature algorithm in the ECDH
and DH key exchanges is RSA, the server is also required to possess an RSA
signing key.

3.1 Summary of the attack

In the explicit prime curve option, the server includes in its signed ‘ServerKey-
Exchange’ message the parameters of its elliptic curve and an ephemeral public
key to be used for this session (see Fig. 3a). The randomness of the public key
contributes to the feasibility of achieving a cross-protocol attack.

In the attack the adversary, after receiving the client’s Hello message, initi-
ates multiple connections to the server, until an ECDH ‘ServerKeyExchange’
is presented that satisfies two properties. The first is that the message can
be interpreted as a valid DH ‘ServerKeyExchange’ message, and secondly, the
adversary can recover the exchanged DH key. After a suitable message is re-
ceived, the adversary forwards it to the client, who verifies the (valid) signature
and proceeds with the handshake. Assuming the adversary can recover the ex-
changed DH key, the handshake with the client completes and thus the server
impersonation is successful. The attack is sketched in Fig. 4.

We first estimate the probability with which a valid ECDH key exchange mes-
sage can be interpreted as a valid DH key exchange message. Then we in-
vestigate how the adversary can recover the session key, either by explicitly
computing a discrete logarithm or forcing the session key to take a value in a
limited set. Finally, we compute the number of server connections required by
the simplest version of our attack.

3.2 Probability of valid key exchange message

Length requirements on key exchange parameters The attack success
depends on whether the signed ECDH parameters can be interpreted as DH
parameters. In Fig. 3 we contrast the contents of the ‘ServerKeyExchange’
packets in both cases. The ECDH parameters consist of the constant curve pa-
rameters followed by the randomly generated ECDH public key. For our attack
to succeed, we require that the $p$ field in the DH parameters extends past the
<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>curve_type [1]</td>
<td>prime length ( (L_q) )</td>
<td>EC prime ( q )</td>
</tr>
<tr>
<td>( L_q )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 1 + L_a )</td>
<td></td>
<td>length of ( a ) ( (L_a) )</td>
</tr>
<tr>
<td>( L_a )</td>
<td></td>
<td>EC curve parameter ( a )</td>
</tr>
<tr>
<td>( 1 + L_a )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 1 + L_b )</td>
<td></td>
<td>length of ( b ) ( (L_b) )</td>
</tr>
<tr>
<td>( L_b )</td>
<td></td>
<td>EC curve parameter ( b )</td>
</tr>
<tr>
<td>( 1 + L_{base} )</td>
<td></td>
<td>base length ( (L_{base}) )</td>
</tr>
<tr>
<td>( L_{base} )</td>
<td></td>
<td>base point</td>
</tr>
<tr>
<td>( 1 + L_{order} )</td>
<td></td>
<td>order length ( (L_{order}) )</td>
</tr>
<tr>
<td>( L_{order} )</td>
<td></td>
<td>order</td>
</tr>
<tr>
<td>( 1 + L_{cofactor} )</td>
<td></td>
<td>cofactor length ( (L_{cofactor}) )</td>
</tr>
<tr>
<td>( L_{cofactor} )</td>
<td></td>
<td>cofactor</td>
</tr>
<tr>
<td>( 2L_q + 1 )</td>
<td>public type [4]</td>
<td>public length ( (2L_q + 1) ) ( public type )</td>
</tr>
<tr>
<td>( 0 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Elliptic Curve Diffie-Hellman

(b) Diffie-Hellman

Figure 3: Contrasting the ‘ServerKeyExchange’ message contents with Diffie-Hellman and explicit elliptic curve Diffie-Hellman parameters, side-by-side. Each row represents a 2-byte (16-bit) field, unless the length is explicitly given. All the indicated lengths are in bytes and the numbers in brackets denote a constant value field.
The new cross-protocol attack

Figure 4: A message sequence chart describing our attack. The dashed lines indicate a forwarded message, and the consequent lines indicate multiple trials.
constant curve parameters\(^1\), which results in \(p\) having its least significant bytes and \(g\) and \(Y_s\) fully positioned in the space of the ephemeral elliptic curve public key and therefore have random contents. This means that multiple queries to such server would provide ‘ServerKeyExchange’ messages with ECDH parameters that if interpreted as DH parameters will have variable lengths for \(g\) and \(Y_s\) and there is a non-zero probability for these lengths to have valid values (i.e. add up to the remaining message length).

Before calculating this probability, we need to clarify when these length fields are positioned in the ephemeral public key space. As already mentioned, this depends on the length of \(p\), which is interpreted based on the contents of the curve_type and elliptic curve prime length (\(\mathcal{L}_q\)) as shown in Fig. 3a. Since in the explicit curves TLS option the curve_type byte contains the identifier 1, the length in bytes of \(p\) (\(\mathcal{L}_p\)) would be interpreted as:

\[
\mathcal{L}_p = 1 \| \mathcal{L}_q = 256 + \mathcal{L}_q,
\]  

where \(\|\) denotes concatenation. For the message to be parsed correctly and the \(g\) and \(Y_s\) lengths to be placed accordingly, \(\mathcal{L}_p\) must be larger than the length of all the fixed parameters and less than the total length of the ECDH parameters (minus the minimum length of the \(\mathcal{L}_q, g, \mathcal{L}_{Y_s}, Y_s\) fields being 6 bytes). The fixed parameters are the prime \(q\) and the fields marked as \(C\) in the ECDH message as shown in Fig. 3a. Then:

\[
C = \mathcal{L}_a + \mathcal{L}_b + \mathcal{L}_{\text{base}} + \mathcal{L}_{\text{order}} + \mathcal{L}_{\text{cofactor}} + 7
\]  

and we require:

\[
\mathcal{L}_q + C \leq \mathcal{L}_p \leq C + 3\mathcal{L}_q - 6
\]

Note that in the above equation, \(3\mathcal{L}_q\) represents the sum of the lengths of the prime \(q\), and the \(X\) and \(Y\)-coordinates of the ephemeral elliptic curve public key. Combined with Eqn. (1) we conclude that \(C\) should satisfy:

\[
262 - 2\mathcal{L}_q \leq C \leq 256.
\]

The constraints for \(C\) are not unrealistic, and if we consider randomly generated curves (i.e. with random \(a\) and \(b\)) the attack will work for any elliptic curve over a finite field of characteristic roughly between 300 and 400 bits. For instance, if we test the prime curves listed in [6] we see that they are fulfilled for the secp384r1 curve (although this is a named curve), as shown in Table 1.

\(^1\)If this requirement is not satisfied an attack may be possible on servers with constant curve parameters of certain form. We do not consider this attack in this paper.
Table 1: The sizes (in bytes) of the parameters in various named curves. The curve marked with gray fulfills the requirements for the attack: $262 - 2\mathcal{L}_q \leq C \leq 256$.

<table>
<thead>
<tr>
<th>Curve name</th>
<th>$\mathcal{L}_q$</th>
<th>$\mathcal{L}_a$</th>
<th>$\mathcal{L}_b$</th>
<th>$\mathcal{L}_{\text{base}}$</th>
<th>$\mathcal{L}_{\text{order}}$</th>
<th>$\mathcal{L}_{\text{cofactor}}$</th>
<th>$C$</th>
<th>$262 - 2\mathcal{L}_q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>secp192k1</td>
<td>24</td>
<td>1</td>
<td>1</td>
<td>$2\mathcal{L}_q + 1$</td>
<td>$\mathcal{L}_q$</td>
<td>1</td>
<td>83</td>
<td>214</td>
</tr>
<tr>
<td>secp192r1</td>
<td>24</td>
<td>$\mathcal{L}_q$</td>
<td>$\mathcal{L}_q$</td>
<td>$2\mathcal{L}_q + 1$</td>
<td>$\mathcal{L}_q$</td>
<td>1</td>
<td>129</td>
<td>214</td>
</tr>
<tr>
<td>secp224k1</td>
<td>28</td>
<td>1</td>
<td>1</td>
<td>$2\mathcal{L}_q + 1$</td>
<td>$\mathcal{L}_q$</td>
<td>1</td>
<td>95</td>
<td>206</td>
</tr>
<tr>
<td>secp224r1</td>
<td>28</td>
<td>$\mathcal{L}_q$</td>
<td>$\mathcal{L}_q$</td>
<td>$2\mathcal{L}_q + 1$</td>
<td>$\mathcal{L}_q$</td>
<td>1</td>
<td>149</td>
<td>206</td>
</tr>
<tr>
<td>secp256k1</td>
<td>32</td>
<td>1</td>
<td>1</td>
<td>$2\mathcal{L}_q + 1$</td>
<td>$\mathcal{L}_q$</td>
<td>1</td>
<td>107</td>
<td>198</td>
</tr>
<tr>
<td>secp256r1</td>
<td>32</td>
<td>$\mathcal{L}_q$</td>
<td>$\mathcal{L}_q$</td>
<td>$2\mathcal{L}_q + 1$</td>
<td>$\mathcal{L}_q$</td>
<td>1</td>
<td>169</td>
<td>198</td>
</tr>
<tr>
<td>secp384r1</td>
<td>48</td>
<td>$\mathcal{L}_q$</td>
<td>$\mathcal{L}_q$</td>
<td>$2\mathcal{L}_q + 1$</td>
<td>$\mathcal{L}_q$</td>
<td>1</td>
<td>249</td>
<td>166</td>
</tr>
<tr>
<td>secp521r1</td>
<td>66</td>
<td>$\mathcal{L}_q$</td>
<td>$\mathcal{L}_q$</td>
<td>$2\mathcal{L}_q + 1$</td>
<td>$\mathcal{L}_q$</td>
<td>1</td>
<td>339</td>
<td>130</td>
</tr>
</tbody>
</table>

**Probability estimate** If a server uses explicit elliptic curve parameters in the appropriate range, the attack is straightforward, even though it requires quite some effort from the adversary and the server. The adversary intercepts a client connection; upon receipt of the ‘ClientHello’, initiates multiple connections to the server. His goal is to obtain signed ECDH parameters that contain valid lengths for $g$ and $Y_s$.

From Fig. 3a we see that the maximum valid length of $g$ is given by

$$2\mathcal{L}_q - (256 - C) - 2 - 2 - 1,$$

where the consecutive terms in the above sum correspond to: the size of $X$ and $Y$-coordinate, the part taken up by $p$, the length field of $g$, the length field of $Y_s$ and the minimal length of $Y_s$. If we define this upper bound as $L := 2\mathcal{L}_q - 261 + C$, then we conclude that the valid lengths of $g$ satisfy

$$0 < \mathcal{L}_g \leq L.$$  \(4\)

For each valid $\mathcal{L}_g$ there is precisely one possible length of $Y_s$, namely

$$\mathcal{L}_{Y_s} = L + 1 - \mathcal{L}_g.$$  \(5\)

Since the lengths of $g$ and $Y_s$ are 16-bit values and assuming a uniform distribution of the bytes on the elliptic curve public key, the probability of the ECDH message being parsed successfully as DH parameters is:

$$P(\text{valid message}) = \frac{L}{2^{16}} \cdot \frac{1}{2^{16}} = \frac{L}{2^{32}}.$$  \(6\)

Since the lengths of $g$ and $Y_s$ are assumed random, the probabilities that these attain any given fixed value are independent. The uniform distribution assumption for the values read as the $g$ and $Y_s$ lengths may appear questionable.
Even though a random $X$-value has probability close to $1/2$ of being a valid $X$-coordinate and its uniformity assumption is plausible, this is not the case with the $Y$-coordinate. To confirm this assumption we calculated using [21] the estimated PDF, shown in Fig. 5, for the two lengths (of $g$ and $Y_s$) on the secp384r1 curve. The plots strengthen the assumption of uniformity. Furthermore, by simulating the attack on the secp384r1 curve, we calculated the empirical probability for a valid message and compared it with the theoretical probability computed by Eqn. (6), in Table 2. The theoretical values do not significantly deviate from the empirical ones.

![Figure 5: The estimated PDF for values read as the length of $g$ (blue), and the length $Y_s$ (red) on the secp384r1 curve. The length of $Y_s$ is calculated after a suitable length for $g$ is found.](image)

<table>
<thead>
<tr>
<th>Probability</th>
<th>Theoretical</th>
<th>Empirical</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid message</td>
<td>$1.9 \cdot 10^{-8}$</td>
<td>$2.0 \cdot 10^{-8}$</td>
</tr>
</tbody>
</table>

### 3.3 Recovering the session key

So far, we have explained how the adversary can use the server as an oracle to receive, with a certain probability, a signed message containing a triplet of
random numbers, which the client interprets as the DH parameters \((p, g, Y_s)\).
To prevent confusion with the role implied by these symbols (e.g. \(p\) may not be a prime) we will use the symbols \((n, \alpha, \beta)\) instead.

Upon receipt, the client calculates his DH share as \(Y_c \equiv g^x \equiv \alpha^x \mod n\), and the pre-master secret as \(Y_s^x \equiv \beta^x \mod n\). Recall from Eqn. (1) that the size of \(n\) is larger than 256 bytes or 2048 bits. To compute the session key, the adversary can either recover \(x\) from \(Y_c\) exploiting the fact that \(n\) is not necessarily prime or he can force specific values for the pre-master secret \(\beta^x\) by judiciously selecting the triplet \((n, \alpha, \beta)\) he forwards to the client. We now analyze both cases in detail.

**Computing \(x\)**  Recovering \(x\) from \(\alpha^x \mod n\) corresponds to computing a discrete logarithm in \(\mathbb{Z}_n\). If the factorization of \(n = \prod_{i=1}^{s} p_i^{e_i}\) is known or can be computed, this problem can be easily reduced (using the Chinese Remainder Theorem) to a number of discrete logarithm problems in \(\mathbb{F}_{p_i}\). Unfortunately, computing the factorization of \(n\) using the Number Field Sieve [19] (NFS) has similar complexity to computing discrete logarithms in \(\mathbb{F}_p\) with \(p\) the same size as \(n\). On the other hand, the complexity of the Elliptic Curve factoring method [20] (ECM) depends on the size of the smallest prime factor of \(n\), so if the \(p_i\) are much smaller than \(n\), ECM would outperform NFS easily. To analyze the applicability of ECM, recall that a number \(n\) has prime factors \(p_i\) all smaller than \(n^{1/u}\) with probability asymptotically equal to \(u^{-u}\) [8].

Two different attack scenarios now arise, namely online vs. offline. In the online scenario, the attacker has to compute \(x\) during the lifetime of the handshake. Since the typical handshake timeout is less than a minute, the attacker can only succeed if \(n\) is really smooth. Since \(n\) has more than 2048 bits, we would require \(u > 20\) for the prime factors to be small enough for the attack to succeed within the timeout. Hence, the probability that \(n\) satisfies these requirements is \(< 2^{-80}\).

In the offline scenario, the attacker has a much higher probability of succeeding. Indeed, now \(n\) can have one large prime factor (e.g. around 530 bits, the current DLP record [15]) and all other prime factors much smaller (e.g. around 240 bits, the current ECM record [5]). Since the size of \(n\) is never larger than 2500 bits, the probability that the attacker succeeds given a valid message is larger than \(8^{-8} = 2^{-24}\). This probability then needs to be multiplied with Eqn. (6) which is around \(2^{-25}\). The overall success probability therefore is around \(2^{-49}\).

The fact that the key can be recovered offline does not pose any threat to the TLS protocol since in the normal execution of the protocol, the application data are only transmitted after the verification of ‘Finished’ messages by both peers. However, extensions to the protocol such as the “False start” [18] that try to
reduce protocol round-trips by sending the client’s application data before the peer’s finished message is verified, are at risk. This attack may be used to obtain the encrypted client’s data and decrypt it by calculating the shared key offline.

**Computing pre-master secret** Recall that the ephemeral key $Y_s$ is given by part of the X and/or Y coordinate of the ephemeral public key generated by the server. It is thus possible and in fact not unlikely (see the next section), that $Y_s = 0, \pm 1$. As observed in [22], if the client accepts any of these values, the pre-master secret is very easy to compute since it will be equal to $0, \pm 1$.

Such values for $Y_s$ can be generalized slightly when there are other roots of unity of small order or roots of nullity if $n$ is composite, since then it becomes likely that the pre-master secret again equals 1 or 0. The main obstacle for this generalisation is the fact that the available length for $Y_s$ is rather small compared to the total length of $n$. We require roots of unity and/or nullity with compact representation modulo $n$. For roots of nullity it is easy to see that this is very unlikely: indeed, let $n = \prod_{i=1}^{s} p_i^{e_i}$, then the root of nullity with most compact representation modulo $n$ is given by $r = \prod_{i=1}^{s} p_i$. This shows that a small root of nullity is extremely unlikely. For roots of unity of order $k > 2$, the condition is that $n \mid \Phi_k(r)$, with $\Phi_k(x)$, the $k$-th cyclotomic polynomial, so for random $n$ this is again unlikely. Hence, we will mainly focus on the case $Y_s = 1$.

### 3.4 Attack success probability

In this section, we provide a lower bound on the adversary’s probability of success for the simplest attack where the adversary expects to find the value 1 in $Y_s$.

To compute the probability that $Y_s = 1$ for a valid message, we again refer to Fig. 3. For every fixed valid length $\mathcal{L}_g = k$ with $0 < k \leq L$ we can compute the probability that the message is valid and $Y_s = 1$. Recall that for $\mathcal{L}_g = k$, we have $\mathcal{L}_{Y_s} = L + 1 - k$. If we set $V_k$ to be the event that $\mathcal{L}_g = k$ and $\mathcal{L}_{Y_s} = L + 1 - k$, then the probability that $V_k$ and $Y_s = 1$ (all bytes of $Y_s$ equal to zero, except the least significant byte equal to 1) is therefore given by

$$P(V_k \land Y_s = 1) = P(V_k)P(Y_s = 1 | V_k) = \frac{1}{2^{32}} \cdot \frac{1}{2^{8(L+1-k)}}.$$
Summing over all $0 \leq k \leq L$ we thus obtain
\[ P(\text{valid message } \land Y_s = 1) = \frac{1}{2^{32}} \sum_{i=1}^{L} \frac{1}{2^{8i}} \approx 2^{-40}. \] (7)

This shows that an adversary needs roughly $2^{40}$ signed elliptic curve public keys from the server to obtain a message that satisfies the attack requirements. For the attack to proceed, we note that this message needs to be received during the lifetime of the TLS handshake with a client (this could be a random client).

4 Attack assumptions

The described attack relies on the following two assumptions.

1. The client software supports one of the ‘TLS\_DHE\_RSA’ ciphersuites and a DH public key ($Y_s$) with value 1 is accepted;
2. The server software supports one of the ‘TLS\_ECDHE\_RSA’ ciphersuites, with the ‘arbitrary\_explicit\_prime\_curve’ [3] option, has selected a curve of size between 300 and 400 bits and uses RSA as the signing algorithm.

First assumption The first assumption turns out to be true for several implementations. We tested TLS implementations that support the ‘TLS\_DHE\_RSA’ ciphersuites, and concluded that not all of them include sanity checks on the received DH parameters. We tested the invalid values 0, ±1 and summarized the obtained results in Table 3.

Table 3: The behavior of implementations when receiving various values as a Diffie-Hellman public key ($Y_s$).

<table>
<thead>
<tr>
<th>$Y_s$</th>
<th>NSS 3.12.6</th>
<th>OpenSSL 1.0.1</th>
<th>GnuTLS 3.0.18</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Y_s = 1$</td>
<td>Accept</td>
<td>Accept</td>
<td>Accept</td>
</tr>
<tr>
<td>$Y_s = 0$</td>
<td>Reject</td>
<td>Reject</td>
<td>Reject</td>
</tr>
<tr>
<td>$Y_s = -1$</td>
<td>Accept</td>
<td>Accept</td>
<td>Accept</td>
</tr>
</tbody>
</table>

We can see that there are implementations that blindly trust the signed parameters received by the server and do not check for invalid parameters\(^2\). Note that the value of 0 in NSS was rejected if it had minimal encoding but accepted otherwise.

\(^2\)Note that the authors of these implementations have been contacted and the described vulnerabilities have been fixed.
It can be argued that these are individual implementation bugs because the TLS protocol [11] includes recommendations for checking DH parameters. However, the simple checks for public values of \( \pm 1 \) and 0 are not explicitly mentioned and the implementer is directed to other documents instead.

Although there is merit in testing for obvious invalid parameters, the protocol should not rely on these tests. Such tests will rule out corner cases like the one our attack is using, but no matter how thorough they cannot detect values like roots of unity or nullity that can be used to mount a variant of our attack, nor ensure the proper generation of the parameters. It can be argued that the former values could be ruled-out by forcing the client to perform a primality check on the group modulus and to calculate the order of the generator. However, even if that was an acceptable time to security trade-off, the latter cannot be performed with the information provided in a TLS handshake. It is crucial that the trust in the peer’s signature needs to extend to a trust in the parameters, and that can only be achieved with a protocol fix.

**Second assumption** Although almost all servers on the Internet have an RSA public key, at the moment this paper was written, public data available about Internet servers such as the ‘Internet SSL Survey’ [24] did not include the required information to assess whether servers supporting arbitrary elliptic curves exist. Our belief is that because popular open source TLS protocol implementations such as NSS, OpenSSL and GnuTLS currently do not support arbitrary elliptic curves, the majority of Internet servers are unaffected by our attack.

5 *Feasibility of the attack*

Given the large number \( 2^{40} \) of signed messages the adversary requires from the server while the client’s handshake is on hold, we need to evaluate the feasibility of the attack in a real world scenario. For that we performed a simulation of the part of the attack involving the server and the adversary. However, since open-source implementations of TLS do not support explicit elliptic curves, in the simulation we use the named curve TLS option, which limits our choice of curves. In our simulation, the server is a web-server that supports TLS with the secp384r1 curve and holds a 1024-bit RSA key. The adversary performs multiple partial handshakes with the server, that are terminated once the signed ‘ServerKeyExchange’ message is received. The simulation was performed using two 24-CPU Intel Xeon (X5670) systems interconnected using Gigabit Eth-
ernet. One was used as an HTTPS server using the nxweb\textsuperscript{3} software with GnuTLS 3.0.20, and the other was used to initiate the adversary’s connections to the server. The software used to simulate the adversary is a modified version of the httpress\textsuperscript{4} tool, that measures the number of requests per second of the partial TLS handshake used in the attack. Both nxweb and httpress operate in a multi-threaded manner and were assigned all the available CPUs in each system.

We now consider two variants of the attack. The first targets a specific client, following closely our attack as in Fig. 4, and the second is an attack that targets any client.

5.1 Attacking a specific client

In this attack the adversary’s goal is to impersonate the server to a specific client of his choice. The adversary would then be required to perform $2^{40}$ connections to the server before the client times out. In our test setup the adversary was able to perform 3770 connections per second on average, resulting in an expected time of approximately 9 years to attempt all the $2^{40}$ connections. This is quite a long time given the TLS handshake timeout values in browsers (see Table 4), and human patience. However, typical high-load Internet servers operate in clusters and may support hardware acceleration of RSA signing operations. On such a combination that improves performance by a factor of 1000 the estimated time is 3 days. That is still a long time for a single interactive session, indicating that the described attack may not be practical today in attacking a specific targeted client. The attack simulation results are summarized in Table 5.

Table 4: TLS handshake timeout values in various browsers.

<table>
<thead>
<tr>
<th>Browser</th>
<th>Handshake timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chrome 20</td>
<td>20 secs</td>
</tr>
<tr>
<td>Firefox 10</td>
<td>30 secs</td>
</tr>
<tr>
<td>Internet Explorer 8</td>
<td>40 secs</td>
</tr>
<tr>
<td>Opera 12</td>
<td>40 secs</td>
</tr>
</tbody>
</table>

\textsuperscript{3}https://bitbucket.org/yarosla/nxweb/wiki/Home
\textsuperscript{4}https://bitbucket.org/yarosla/httpress/wiki/Home
Table 5: The resources required by the web server during the attack simulation.

<table>
<thead>
<tr>
<th>Web server</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitted data</td>
<td>4.7 MB/sec</td>
</tr>
<tr>
<td>Received data</td>
<td>1.8 MB/sec</td>
</tr>
<tr>
<td>Requests handled</td>
<td>3770 req/sec</td>
</tr>
</tbody>
</table>

5.2 Attacking a random client

On the other hand, there are cases where the adversary is not particularly interested in a specific client and impersonating the server to a random client is sufficient. In that scenario, the adversary hijacks every (distinct) client connection attempt and uses the interval before the connection times out to initiate multiple connections to the server. On every failed attempt he gives up on that client who sees a connection timeout in his browser. Using the numbers from the simulation, and assuming a 40 second connection timeout value, the adversary would expect to impersonate the server after $2^{23}$ distinct client connection attempts. For high-load servers this is a rather low number. For example, Google currently has 700 million unique visitors per day, so it would only take on average 17 minutes for the attack to succeed (if the increase in load can be absorbed).

6 A possible fix

Currently the signature on the ‘ServerKeyExchange’ message covering the key exchange parameters, ensures only the freshness of the message, but not whether it was intended for this particular key exchange. A simple fix may be to include the negotiated ciphersuite into the signature. However, the TLS protocol is complex and negotiates extensions that modify the key exchange in several ways. For example, in this particular attack we used the explicit curves option of TLS, which is negotiated using an extension. Such extensions should also be covered by the signature, to prevent any attack that uses the sub-options allowed by a key exchange algorithm.

For that we propose to modify the signature of the ‘ServerKeyExchange’ to include, in addition to explicit identifiers of the algorithms, all the previously exchanged messages. Our proposed signature for a ‘ServerKeyExchange’ message is shown in Fig. 6. It includes explicit indicators of the entity (server), the key exchange algorithm used, the handshake messages exchanged, and the parameters of the key exchange. This modification may be negotiated either
with an upgraded TLS version number, or by defining a new TLS extension similarly to the approach in [23], allowing backwards compatibility.

```plaintext
enum { server (0), client (1) } ConnectionEnd;

enum { dhe_dss (0), dhe_rsa (1),
       ec_diffie_hellman (2) } KeyExchangeAlgorithm;

struct {
  select (KeyExchangeAlgorithm) {
    case dhe_dss:
    case dhe_rsa:
      ServerDHPParams params;
    case ec_diffie_hellman:
      ServerECDHParams params;
  }
} Parameters;

struct {
  Parameters params;
  digitally-signed struct {
    ConnectionEnd entity;
    opaque handshake_messages<1..2^24-1>;
    KeyExchangeAlgorithm kx_algorithm;
    Parameters params;
  }
} ServerKeyExchange;
```

Figure 6: The proposed format for the ServerKeyExchange message signature. Note that we follow the TLS protocol message description. In particular, the type opaque is used to indicate bytes containing uninterpreted data and arrays of variable length, specified with the `<floor..ceiling>` notation, are preceded by a number of bytes containing the length of the array.

A drawback of this change is that it requires caching of the previously exchanged messages until the ‘ServerKeyExchange’ message is sent. This however, should be an insignificant cost for today’s servers.
7 Conclusions

In this paper we presented a new attack on the TLS protocol that exploits the fact that a client can interpret signed explicit elliptic curve Diffie-Hellman (DH) key exchange parameters as valid plain DH parameters. It is a cross-protocol attack similar in nature to the attack introduced by Wagner and Schneier. The attack enables impersonation of the server and is much more efficient than breaking any of the involved cryptographic primitives.

Nonetheless, the presented attack depends on the server supporting the explicit elliptic curves option. This option is not supported in the tested open-source implementations making them resistant to this attack. This fact suggests that for now the implementation of the explicit elliptic curves protocol option should be avoided unless a counter-measure like our proposed fix is in place.

A limiting factor of the attack in servers that support the explicit elliptic curve option is that it requires the initiation of $2^{40}$ sessions within the timeframe of the client session. We show that this may be prohibitive for attacking a specific client, but if the target is attacking any random client, the attack should be feasible on a cluster of servers.

Moreover, we show that extensions of the TLS protocol such as the “False start” [18] that reduces protocol round-trips by sending the encrypted client’s application data before the full handshake is complete, are at risk. The attack may be used by an adversary to recover the encryption key offline and access the encrypted data.

The described attack can be countered by verifying the server’s DH public key for known invalid values. The TLS protocol, however, should not over-rely on these tests. There could be other values that can be used to mount variants of our attack but cannot be easily detected. We believe that the trust in the peer’s signature needs to extend to a trust in the signed parameters.

It is also worth noting that our attack may not be the only possible cross-protocol attack on the TLS protocol. Due to the fact that the protocol in its current form allows a server to be used as an oracle by an adversary, other attacks that explore different algorithm interactions may also be possible. For that we proposed a fix to the protocol that defends against this family of attacks.
8 Acknowledgments

The authors would like to thank David Wagner, Kenny Paterson, Andreas Pashalidis, Yngve Nysaeter Pettersen, Eric Rescorla, Adam Langley, Bodo Moeller, Marsh Ray, and the anonymous referees for their comments which improved this manuscript. Moreover we would like to thank Koen Simoens and Elmar Tischhauser who contributed in the formulation of this attack. This work was supported in part by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen) SBO project, the Research Council KU Leuven: GOA TENSE (GOA/11/007), by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy).

References


Addendum

Roots of nullity

The term roots of nullity [28] is used to describe numbers that are roots of zero in a ring $R$ of integers modulo $m$. That is, the numbers $r$ such that $\exists x \in R : r^x = 0 \mod m$.

Probability of valid key exchange message

In subsection “Length requirements on key exchange parameters” the text “over a prime finite field of characteristic roughly between 300 and 400 bits” is replaced with “over a finite field of prime order roughly between 300 and 400 bits”.

A Linux kernel cryptographic framework
(extended version)
A Linux kernel cryptographic framework:
Decoupling cryptographic keys from applications*
[extended version]

Nikos Mavrogiannopoulos
Dept. of Electrical Engineering/COSIC
Katholieke Universiteit Leuven

Miloslav Trmač
Red Hat

Bart Preneel
Dept. of Electrical Engineering/COSIC
Katholieke Universiteit Leuven

May 2013

Abstract

This paper describes a cryptographic service framework for the Linux kernel. The framework enables user-space applications to perform operations with cryptographic keys, while at the same time ensuring that applications cannot directly access or extract the keys from storage. The framework makes use of the higher privilege levels of the operating system in order to provide this isolation. The paper discusses the relevant security requirements and expectations, and presents the design of the framework. A comparison with alternative designs is also provided.

1 Introduction

There are many types of attacks against software applications, such as code injection, control flow modification [30, 1], etc., attributed to software vulnerabilities. Their elimination has been proven to be a Sisyphean task and today, software vulnerabilities are mitigated using a combination of defensive programming [27] and retroactive patching. As a result, there is a risk of an
attacker ‘taking control’ of an application exposed to a public network (such as a web server), maybe for some limited time until the vulnerability is known and the system is updated.

The cost of such compromise may be significant for the stakeholders. One example are on-line shops today that typically use TLS [11] to authenticate to the client and setup a secure communications channel. The secure channel is setup using a server certificate and a corresponding private key. Because in typical web server applications the private key resides in the same address space as the server, its compromise is equivalent to a compromise of the private key. Thus, an adversary with the ability to compromise the web server could retrieve the private key and obtain the ability to masquerade as the original shop to its clients. This would be possible even after such a compromise is no longer possible (e.g., caused by an automatic software upgrade, or by installing a newer operating system). This might be catastrophic for the reputation of the shop.

A defense mechanism for the above attack is decoupling cryptographic keys from the software that uses them. This is typically achieved using hardware security modules or smart cards [29], that store the cryptographic keys in a medium that allows operations without exposing them. That approach effectively raises the protection of cryptographic keys from the cost of an application compromise, to the cost of compromising a hardware module. However, there is a possibility to increase the protection levels without involving new hardware. Operating systems (OS) are typically designed with security in mind [18], and provide protection mechanisms such as segregation of the OS with user applications and isolation of independent processes. These mechanisms can be used to decouple cryptographic keys from applications, providing an additional layer of security to a system where hardware security modules are not practical or are too costly (e.g. low-cost servers, or mobile phones).

Today, frameworks offering isolation of cryptographic keys from applications exist [24, 23], based on the separation of processes and the access control mechanisms offered by the OS. In this paper we present a cryptographic framework that decouples cryptographic keys from applications based on the separation between the OS and its processes. Apart from the description, we discuss the difficulties this approach imposes as well its advantages and disadvantages comparing to an alternative design. Moreover, in order to ensure the safety of the stored keys, we base the design of the framework on principles that delegate the protection of stored keys to the strength of the supported algorithms.

The next section provides a background on existing work. In Section 3 we for-

---

1By the term ‘address space’ we mean memory accessible in a process. Each process has its own address space.
malize the threats on current operating systems and set the basic requirements for our framework. In Section 4 we describe our framework on a high level and present its architecture. A security analysis is provided in Section 5. In Section 6 a performance analysis is provided and the following section discusses the differences between our approach and alternatives. Section 8 concludes.

2 Background and related work

Hardware security modules (HSM) and smart cards are the main medium used today for decoupling cryptographic keys from software and are often required by governmental standards [26, 17]. Their operation is based on a logical and physical separation of cryptographic keys from the applications using them. For example a typical PKCS #11\(^2\) smart card or a TPM module [29] allow for cryptographic operations (e.g., RSA decryption) using stored keys, without exposing them.

Because the deployment of hardware security modules is not always feasible, a few software-only approaches to separation already exist, based on existing isolation mechanisms\(^3\). The CNG API [24] is a redesign of the old Microsoft CryptoAPI, that allows the decoupling of cryptographic keys from applications. It uses a special ‘key isolation process’ executing with the permissions of a system user which provides services, such as cryptographic key storage and operations to other applications. A similar approach is used by another framework, the lite security module [23] (LSM). It provides a PKCS #11 API to applications in order to access a central daemon. The daemon executes under a system user and provides key storage and operations to the other applications. Both frameworks allow persistent keys that cannot be exported.

The protection level of security modules, however, is not easy to assess. That is because the available operations should be selected in way that no combination of them is sufficient to obtain or reconstruct the stored keys. This requirement, although simple to formulate it has been shown in practice that it is hard to achieve in real-world security modules [5, 7].

The goal of a formal proof of key safety in security modules is on-going work, and currently we only have indications of security in restricted designs [10, 6] that support few operations. Thus, in our design we make sure that the cryptographic operations implemented and made available are not sufficient to

---

\(^2\)A cryptographic API that provides logical separation between keys and operations. It is mainly used to access operations on hardware security modules and smart cards.

\(^3\)Software that provides emulation to a security module but provides no isolation between cryptographic keys and the application was not considered in our study.
extract the stored keys. We will use a minimal set of operations carefully chosen for attack resistance properties. E.g., a signing operation using algorithms that are resistant to adaptive chosen-message attacks [13].

The term security module is used in this paper to describe any framework or hardware that provides operations on cryptographic keys without exposing them.

3 Model and Requirements

We chose our model based on the current single level\(^4\) operating system design. In that model software is comprised of an operating system running in a different protection ring\(^5\) than normal applications. This requirement ensures that normal applications cannot access data available to the operating system only. We also assume that attacks on the operating system are far less likely to succeed than attacks on specific software. This assumption is supported by [30] which concludes that “Application vulnerabilities exceed OS vulnerabilities”.

A user called administrator, is able to directly control the OS executing in the highest protection ring and there is at least one user that operates solely on a lower protection ring. The applications running on a system are assumed to be running with the privileges of a normal user. We use the common terms user-space to reference the lower privilege rings and kernel-space for the higher privilege ring the OS operates.

3.1 Threats

In this model of an OS, applications operate in user-space and utilize cryptographic keys. We assume that application vulnerabilities might exist for some time until the system is patched. Those give adversaries a time window to execute malicious code at the same ring as the applications running in the system. Thus our main threats are:

- An adversary that has taken control for limited time of a user-space application and is trying to recover the protected keys; the adversary might be collaborating with another legitimate user;

\(^4\)Single level is used here to distinguish between Multi-Level Security (MLS) systems.

\(^5\)We use the term protection ring to assume a strict separation of data and code between different access control levels. This terminology was described during the design of Multics in [32] and today it is a standard component of Operating Systems and CPUs [16, Privilege Levels p. 5–9].
• An adversary that is a legitimate system user and wants to extract raw keys; in contrast to the former adversary, this one is not restricted by time.

We want to protect keys in the presence of such adversaries. It should be noted that in this model, there is no distinction between operating system access and administrative access. This is due to the design of the modelled system.

3.2 Requirements

<table>
<thead>
<tr>
<th>Feature</th>
<th>Supported Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key generation</td>
<td>Symmetric keys and Private/public key pairs</td>
</tr>
<tr>
<td>Encryption and Decryption</td>
<td>RSA PKCS #1 v1.5, RSA PKCS #1 OAEP, AES-CBC, AES-CTR, AES-ECB, 3DES-CBC, 3DES-CTR,</td>
</tr>
<tr>
<td></td>
<td>3DES-ECB, CAMELLIA-CBC, CAMELLIA-CTR, CAMELLIA-ECB, RC4</td>
</tr>
<tr>
<td>Signing and verification</td>
<td>DSA, RSA PKCS #1 v1.5, RSA PKCS #1 PSS, HMAC-MD5, HMAC-SHA1, HMAC-SHA-224, HMAC-SHA-256,</td>
</tr>
<tr>
<td></td>
<td>HMAC-SHA-384, HMAC-SHA-512, MD5, SHA1, SHA-224, SHA-256, SHA-384, SHA-512</td>
</tr>
<tr>
<td>Key wrapping</td>
<td>AES-RFC3394, AES-RFC5649</td>
</tr>
<tr>
<td>Key derivation</td>
<td>Diffie-Hellman as in PKCS #3</td>
</tr>
</tbody>
</table>

A list of high level requirements for the framework concerning the cryptographic keys to be protected, follows.

1. Users should be allowed to perform cryptographic operations using the keys, but the key material should not be extractable;

2. Keys should be protected from all known side-channel attacks;

3. All cryptographic material such as keys must not be accessible from applications that use them, or from any other application, either directly or through a process memory dump;

4. Keys should be transferable to other systems in a secure way.

Our requirements for the features on the implementation were set to include the required functionality for TLS 1.2 [11] protocol implementation, and for its interface to be translatable to the PKCS #11 API. The operations and algorithms implemented are summarized in Table 1.
4 Architecture

In our design we want to emulate the hardware isolation of cryptographic operations from actual keys using the ring separation between the operating system and user-space as shown in Figure 1. Our goal is to completely isolate cryptography from the processes that actually use it. For that reason we implement a kernel service—in our prototype a Linux kernel module—that provides operations to cryptographic keys without exposing them. This would fulfill all of our requirements, as shown in Section 3.

Our framework, named NCR, is implemented as a self-contained Linux kernel module that provides access to cryptographic operations. It supports a synchronous API for communication with the user-space based on the `ioctl()` system call. We emulate the hardware isolation of cryptographic operations from actual keys using the ring separation between the OS and user-space as shown in Figure 1. Our goal is to completely decouple cryptography from the processes that use it.

![Figure 1: High level architecture](image)

4.1 Usage of keys

Applications should be allowed to perform operations using a set of keys in a way that only the outcome will be available to them. For this reason keys should be accessible and referenced as abstract objects, without exposing their contents. However, not all keys should be treated the same. Security protocols, typically operate using long term keys, used for a key exchange that outputs

---

6`ioctl()` is a primitive communication interface between kernel and user-space. It typically works by allowing the caller to specify the operation and a structure with the operation input data.
short term keys valid only for the current session (session keys). Typical cryptographic protocols such as the Internet security protocols [11, 20], satisfy the known session keys property of [4], which ensures that the knowledge of past session keys does not affect protocol security. Thus, in order to avoid protocol specific key derivation algorithms being present in the framework, we explicitly distinguish keys as ‘long term’ and ‘session’ keys.

This distinction strikes a balance between pragmatism and need for key protection. Operations on long term keys will be protected by the proposed framework, while session keys will be generated and used in user-space. The distinction is not arbitrary. Session keys are valid for limited time and protect data in the application address space. Hence, if one is able to extract those keys during their validity period, it is not unrealistic to claim that the data they protect could also be extracted.

To prevent attacks due to interaction between different algorithms the public and private key pairs are tied to a single algorithm, such as RSA or DSA, and we enforce the same rule for secret keys as well. For example a key that is tagged to be used with the AES algorithm cannot be used with RC4 algorithm. In addition we allow for some keys to be used as input to a cryptographic hash or MAC function, in order to be consistent with the usage of secret keys for key exchange as in TLS with pre-shared keys (TLS-PSK) [12].

4.2 Framework components

Our design consists of two basic components, one is the ‘cryptographic backend’ and the NCR component that provides operations to user-space applications. Each component is described in detail in the following paragraphs, and a high level interplay of the components is shown in Figure 2.

4.2.1 Cryptographic backend component

This component of the design is based partially on the existing Linux kernel asynchronous symmetric cipher and hash API, and partially on libtomcrypt and libtommath. The latter libraries were used to implement the asymmetric cryptography backend that is not part of the Linux kernel cryptographic API. The libraries were modified for the Linux kernel and were used for the implementation of RSA, DSA and Diffie-Hellman algorithms. The Linux random number generator [14] was used to provide randomness.

The main purpose of the cryptographic backend component is to provide the
main NCR component with a set of algorithms to support the required operations as in Table 1.

**Figure 2:** Component view of the framework. The entry and exit points for cryptographic keys are marked with double arrows.

### 4.2.2 NCR component

The NCR component contains the main implementation of the provided features. It operates on key objects that are only available within the component. The key objects are marked on generation or import with policy flags that indicate exportable status, wrappable status, or wrapping\(^7\) status. This approach allows for key objects of a gradual security level. The highest being the keys that have no policy flags set and thus cannot be exported nor wrapped with another key, and the lowest being keys that are marked as exportable. We refer to keys with no policy flags as safe keys.

\(^7\)Key wrapping refers to the technique of encrypting a key with another.
To enforce the security flags throughout the component, key objects can enter and leave the component through specific points only, the ‘Key storage’ and possibly the ‘Key wrapping’ subsystem as shown in Figure 2. Keys marked as exportable can also use the ‘Key export’ subsystem. The security flags are checked and enforced by the above subsystems, which also ensure that keys cannot be decoupled from their flags.

**Key objects** A mandatory access control mechanism is used to restrict access to cryptographic keys. Each key object on creation is tied to a particular algorithm and is given an owner and a group. The key access policy is being applied by the subsystem that is performing the operations, the abstract session interface. The usage of a mandatory access control instead of the familiar in Linux discretionary has the advantage of preventing the transfer of keys from users under some outsider’s control to other unauthorized users of the system.

**Abstract session interface** This interface provides a generic way for the supported diverse algorithms to be used. All of the supported algorithms operate on some input data and their functionality is summarized in Table 2. signature verification as well as message digest operations. These operations are mapped to three actions, or ioctl() calls, being NCRIO_SESSION_INIT, NCRIO_SESSION_UPDATE and NCRIO_SESSION_FINAL. The NCRIO_SESSION_INIT action initializes the algorithm in the proper mode, i.e., encryption mode or decryption mode, and associate the operation with a key object. Subsequent NCRIO_SESSION_UPDATE actions operate on input data, i.e., by hashing or encrypting, and might generate output data depending on the operation. The NCRIO_SESSION_FINAL will terminate the operation and if required output data (e.g. in signature generation). That way, a class of different looking operations is served using three basic actions, being compatible with the PKCS #11 API.

Moreover, to avoid the need of multiple system calls for simple operations, i.e., encryption of a given data block we introduced NCRIO_SESSION_ONCE, which combines the functionality of the above operations in a single system call, something that is shown in Section 6 to have impact in performance.

**Key wrapping and unwrapping** The key wrapping subsystem allows transferring keys securely to other systems by encrypting a key with another. The subsystem accepts key encryption keys if they are marked as wrapping keys. The marking of a key as wrapping or unwrapping is a privileged operation (requires administrative access). An integrity check is performed by all the algorithms supported by the subsystem to ensure that random data cannot be unwrapped to a valid key. The type and associated algorithm with the key...
<table>
<thead>
<tr>
<th>Algorithm type</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetric ciphers</td>
<td>Encryption, Decryption, Encryption &amp; Authentication, Decryption &amp; Verification</td>
</tr>
<tr>
<td>Asymmetric ciphers</td>
<td>Encryption, Decryption, Signature generation, Signature verification</td>
</tr>
<tr>
<td>Hash algorithms</td>
<td>Message digest generation</td>
</tr>
<tr>
<td>HMAC algorithms</td>
<td>MAC generation, MAC verification</td>
</tr>
</tbody>
</table>

**Table 2**: Operations available for each algorithm type.

are included with the wrapped data to distinguish between different keys. The calls for wrapping and unwrapping keys are `NCRIO_KEY_WRAP` and `NCRIO_KEY_-_UNWRAP`. Because this operation separates the keys from their associated flags, it is a privileged operation.

**Key storage**  A difference with user-space security module frameworks is that kernel-space provides no ability of data storage\(^8\). For this reason we adopted a design where the storage is delegated to the user-space applications. The protection of safe keys is being done by encrypting and authenticating everything with a system-wide master key. The keys, meta-data such as flags and an access control list are encrypted and authenticated, and handed out as binary data to user-space. Hence, the key storage facility—provided by the framework’s user—is agnostic on the actual keys it stores. It is not able to recover any key, and can only pass it to kernel-space for loading. No naming scheme is imposed by the framework and the user can adopt any convenient scheme for the stored key objects. The corresponding calls for storing and loading keys for storage are `NCRIO_KEY_STORAGE_WRAP` and `NCRIO_KEY_STORAGE_UNWRAP`.

**System-wide master key**  As previously discussed a master key is required to allow the key storage subsystem to operate. The only requirement for this master key is that it has to be loaded by a privileged user—the administrator. In a typical deployment we expect that the key will be provided either as part of the boot process or protected using the filesystem access control mechanism\(^9\).

\(^8\)Note that the Linux kernel can work without any writable filesystem and its subsystems should be operational without any facility for storage.

\(^9\)Storing the system key in a hardware security module, when available, might seem to increase the defenses of the framework even against an adversary with administrative access. However, although such protection would prevent him from recovering the storage key, it would allow him to recover all the locally stored long-term keys. Because we expect long-term keys to be locally stored, the framework would not protect against this adversary.
It should be noted that the overall system security is only as strong as this master key.

**Key derivation** Unlike the ciphers and other algorithms supported by the ‘Abstract session interface’, the Diffie-Hellman key exchange, cannot fit the NCRIO_SESSION_INIT, NCRIO_SESSION_UPDATE and NCRIO_SESSION_FINAL formality of operations. To solve that issue, PKCS #11 defines a way to operate it in software by using a key derivation API, which we adopt. This key derivation API is a single action NCRIO_KEY_DERIVE that accepts an input key, an output key and parameters.

**Key generation** This subsystem supports two actions NCRIO_KEY_GENERATE and NCRIO_KEY_GENERATE_PAIR. The former action generates a random key of a given bit size to be used in MAC and symmetric ciphers. The latter generates a public and private key pair suitable for an algorithm such as RSA, DSA or Diffie-Hellman. All generated keys are tied to a single algorithm, and special parameters are allowed to fine-tune the key generation, i.e., by specifying bit lengths for symmetric keys, or prime and generator for Diffie-Hellman keys etc. Special flags can be given to keys during generation time. They are split in operation and policy flags as in Table 3. However, not all combinations are allowed, to prevent interactions between different algorithms (e.g., prevent a key to be used for RSA signing and encryption) [19]).

**Table 3:** Flags used to mark keys. More than one flag may be applicable to a key.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encryption</td>
<td>The specified key can be used only for encryption</td>
</tr>
<tr>
<td>Decryption</td>
<td>The specified key can be used only for decryption</td>
</tr>
<tr>
<td>Signature</td>
<td>The specified key can be used for signing only</td>
</tr>
<tr>
<td>Verification</td>
<td>The specified key can be used for verifying signatures only</td>
</tr>
<tr>
<td>Wrapping</td>
<td>The specified key can be used for wrapping other keys</td>
</tr>
<tr>
<td>Unwrapping</td>
<td>The specified key can be used for unwrapping other keys</td>
</tr>
<tr>
<td>Wrappable</td>
<td>The key can be wrapped by other keys</td>
</tr>
<tr>
<td>External</td>
<td>The key was not generated within the framework</td>
</tr>
<tr>
<td>Exportable</td>
<td>The key can be exported from the component</td>
</tr>
</tbody>
</table>

**Key Import and Export** Keys that are explicitly flagged as exportable can be exported using the NCRIO_KEY_EXPORT action. Symmetric keys are exported
in raw format, and public and private keys encoded using ASN.1 DER encoding rules. For public keys the SubjectPublicKeyInfo field of an X.509 certificate [8] is used, and for private keys an encoding was used, that is compatible with other libraries such as OpenSSL and GnuTLS\textsuperscript{10}.

4.3 Extensibility

Since the \texttt{ioctl()} system call is limited to static structures and thus does not provide much room for extensibility, we created a hybrid-API based on \texttt{ioctl()} and the netlink\textsuperscript{11} attributes. That way we allow for a number of attributes to be specified per call to provide an extensible API.

5 Security analysis

In this section we discuss how this implementation defends against attacks on cryptographic key isolation frameworks and side-channel attacks. In Appendix A we demonstrate through a human-directed proof how a particular set of security objectives, based on our requirements, are met, in the context of a TLS server application.

5.1 Cryptographic keys protection

Several cryptographic APIs that enforce isolation of keys, have issues that result in modification of key attributes [7, 6] by an adversary. We counter those issues by enforcing keys and attributes to be handled as a single unit in all operations. When for example, keys are exported from the framework, i.e., for storage, the exporting function will not only encrypt but also authenticate the data [15] to prevent modifications. This authentication depends only on the system key, preventing attacks as in [6].

We summarize the actions allowed on each key type in Table 4, and discuss individual aspects of security of the different keys in the framework, in the following paragraphs.

\textsuperscript{10}An alternative would be to use PKCS #8\textsuperscript{28} encoding for private keys, but it was not used because it involved higher complexity than the current approach. This might be reconsidered in a future revision.

\textsuperscript{11}A framework for communication of user-space with kernel.
5.2 Safe keys

Safe keys have no policy flags and only allow cryptographic and storage operations. The supported ciphers are known to be secure against chosen-ciphertext and chosen-plaintext attacks, and the signature algorithms against adaptive chosen-message attacks [13]. This ensures that cryptographic operations alone will not reveal the key. The storage operations as discussed above encrypt and authenticate the key itself, as well as all metadata using the system’s master key. This way unauthorized modification or decryption by all types of adversaries is prevented. In addition to the file system’s access controls that will be used to store the key, the storage key format includes all metadata, e.g., the corresponding cipher and an access control list that specifies the users allowed to unwrap it. Thus, an adversary that has successfully attacked one user account can not use it to load another user’s keys.

The safety of those keys depend solely on the storage subsystem since it is the only subsystem that can be used to export them. The import and key generation subsystems are the only subsystems that can be used to create these keys.

5.3 Wrappable keys

Wrappable keys cannot be exported in raw but have to be encrypted by another designated key available to the framework. It is used as a mechanism to allow transfer of keys to other systems in a secure way, i.e., without revealing them in the process. In [7] Clulow describes several attacks to PKCS #11 wrappable keys. We give a summary below:

1. Wrapping with a weaker key;
2. Inserting a known wrapping key and using it to wrap the key;
3. Decryption of a wrapped key using the wrapping key;
4. Key Conjuring: unwrapping a random key and storing it as a key;
5. Secret Key Integrity: splitting the wrapped key and unwrapping each part separately in order to perform a brute-force attack;
6. Private key modification: Replacing wrapped blocks with different values.

We counter attacks 4-6 by not allowing arbitrary and simple wrapping mechanisms but only algorithms that include an integrity check. Currently the supported ones are the key wrapping methods described by RFC3394 and RFC5649 [31, 15, 25]. The 3rd item, of using the wrapping key to decrypt
the wrapped key, is countered by prohibiting encryption and decryption with keys marked as wrapping keys. The 2nd item is countered by having the addition of wrapping keys a privileged operation, thus prohibiting non-privileged users from inserting their own wrapping keys.

5.3.1 Keys marked as hashable

Keys that are marked as hashable only, allow in addition to operations on safe keys, using the raw key as input to a cryptographic hash or HMAC. Because of that there are might be cases where weaknesses of the available hashes can be used to recover the used key. To prevent that we allow hashable keys to be used only with algorithms that are known to guarantee the 1st preimage resistance property of a cryptographic hash function.

5.4 Exportable keys

Keys marked as exportable cannot be protected by this framework. They should be considered as having the same security level as any keys available to user-space.

<table>
<thead>
<tr>
<th>Allowed operations</th>
<th>Safe key</th>
<th>Wrappable key</th>
<th>Hashable key</th>
<th>Exportable key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cryptographic</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
<tr>
<td>Storage</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
<tr>
<td>Wrapping</td>
<td>⋆</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hashing</td>
<td></td>
<td></td>
<td>⋆</td>
<td></td>
</tr>
<tr>
<td>Export</td>
<td></td>
<td></td>
<td>⋆</td>
<td></td>
</tr>
</tbody>
</table>

5.5 Side channel attacks

Adversaries of such a framework have incentive to take advantage of side-channel attacks such as the ones described in [3, 22], to use timing and other information obtained by operations in order to extract the raw keys. For the purposes of this project we implemented the counter measures for the RSA algorithm, known as RSA blinding [22]. To counter timing attacks in the Linux
kernel symmetric cipher subsystem, constant time ciphers should be added. This was not performed as part of our prototype but would be required for a real-world deployment.

6 Performance Analysis

In this section we show that the performance cost of decoupling of cryptographic keys from user-space processes. We compare the NCR framework (/dev/ncr) against a key isolation framework based purely in user-space. Moreover we include performance results of non-isolation frameworks such as the Linux port of the OpenBSD framework\textsuperscript{12} [21] (/dev/crypto), and a user-space implementation of the AES. The latter two would reveal the performance “loss” by the usage of a key isolation framework.

For the benchmark we created a utility that submitted requests for encryption on each framework of chunks of data with different size and a fixed key. The AES cipher in CBC mode, and the NULL (dummy) cipher were used in our tests. The same implementation of AES was used in all frameworks under test, and all were tested on the same hardware and Linux kernel version 3.0.0.

For a submission of a chunk of data for encryption, the /dev/crypto interface requires 3 system calls, the same as the NCR framework. However NCR allows for submission of requests using a single system call, the NCRIO\_SESSION\_ONCE. We called the latter ‘/dev/ncr single’ variant and included it to our benchmark to assess the optimizations obtained by reducing the number of system calls.

The only existing user-space key isolation framework that was available for the same system was LSM [23], but its design is based on the sockets API, requiring TCP/IP processing and data copying between the processes. A comparison with the NCR or the /dev/crypto frameworks that are zero-copy\textsuperscript{13} would be negatively affected by such overhead. For that reason we implemented a user-space benchmark that would operate using the zero-copy principle. It involves two applications sharing memory and using semaphores for synchronization of operations.

In Figure 3 we show a comparison of all frameworks using the NULL cipher and in Figure 4, a comparison using the AES cipher. Since the NULL cipher performs no operations, the former results reveal the overhead of the framework and its maximum performance limit. Comparing to the /dev/crypto interface

\textsuperscript{12}The Linux port of the OpenBSD framework was selected because of its performance advantage [9] over the native Linux cryptographic API (AF_ALG).

\textsuperscript{13}By zero-copy we mean that no data are copied from the memory of the requesting application to memory of the serving application or kernel.
we see that the NCR implementation has no noticeable performance difference. This illustrates the fact that decoupling of cryptographic keys from processes does not introduce any performance cost to a kernel framework. We can also see from the performance of NCR with a single system call, that the impact of the reduction of system calls from 3 to 1 is important. That is because system calls require context switches (switch execution from user-space to kernel-space) which come at a considerable cost.

The user-space based isolation framework shows very poor performance compared on the others. Profiling of the benchmark applications showed that this performance difference is due to the usage of semaphores for synchronization of the two processes. In the NULL cipher case, 50% of the execution time on each process was spent in synchronization. That is because semaphores in Linux although they are based in user-space futexes, do not completely avoid context switches, causing the observed poor performance. The difference between user-space and kernel-space implementations is reduced to $\approx 10\%$ when a software algorithm such as AES is involved, even though, on the NULL cipher the difference exceeds 50%. Thus over the Linux kernel, a user-space approach to cryptographic key isolation would have disadvantage on high-rate or high-bandwidth transactions.

Both isolation frameworks, NCR and the userspace, are in a disadvantage comparing to userspace non-isolation framework as shown in Figure 4. Although the performance difference seems to become insignificant as larger data chunks
are used, large chunks of data might not be the typical use-case of such a framework. For that we would suggest that a key isolation framework should be used after the benefits of isolation are weighted against the performance loss.

7 Comparison with alternative approaches

The differences between different framework designs, such as based on the separation of processes or the separation with the OS are not limited to performance. A separation framework might have additional goals from the requirements we set in Section 3. For example the CNG framework [24] abstracts the usage of hardware security modules, smart cards or software isolation using the same interface. This is not easily implemented in a framework residing in the OS, since functionality such as access to smart cards is not typically available. Portability is another concern. A cross-platform solution cannot rely on a particular kernel interface.

On the other hand, there are advantages of an OS based framework. This is mainly performance as illustrated in the previous section, as well as effi-

\footnote{For example parsing PKCS #15 structures or PKCS #11 modules requires an addition of large subsystems to the OS.}
Table 5: Comparison of the different approaches for cryptographic key isolation.

<table>
<thead>
<tr>
<th></th>
<th>Kernel-space framework</th>
<th>User-space framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Combination with HSMs</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Portability</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Secure bootstrapping</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Efficient access to hardware cryptographic accelerators</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

cient access to cryptographic accelerators [21] and the secure bootstrapping property [2]. Cryptographic accelerators, when not implemented in the CPU instruction set are only available through a kernel-space driver. A kernel-space cryptographic framework is in advantage of using the driver directly without the cost of context switches. In a secure bootstrapping architecture each system module verifies the modules it loads (or uses). For example, the system loader would verify the OS and the OS would verify the loaded applications and so on. Having the cryptographic framework in the OS allows for a clear and simple design since the OS would already contain the functionality needed to perform signature verification, i.e., the algorithms, and this functionality has already been verified by the system loader as part of the OS. We provide a summary of the advantages and disadvantages in Table 5.

8 Concluding remarks

In the previous sections we have presented the implementation of a cryptographic framework for the Linux kernel. The main goal of the framework is to decouple cryptographic keys from the applications using them, by confining the keys to kernel-space. This confinement is achieved through the enforcement of rules on the way keys are generated and utilized in order to protect against known attacks on cryptographic APIs. The API is practical, in the sense that it can be used to fully implement cryptographic protocols such as TLS 1.2 [11], and can be used to provide services under the PKCS #11 API. It is available to all system users and the administrator role, although required, is limited to maintenance aspects of the framework, such as loading the master keys, and the generation of wrapping keys.

In addition to protecting cryptographic keys this framework provides user-space applications with access to cryptographic hardware accelerators in a similar
way as in [21]. This is especially important in embedded systems with limited processors, where the software implementation of a cryptographic algorithm can be an order of magnitude slower than the hardware one.

Our performance evaluation showed that the throughput of the framework is equivalent to another kernel-based cryptographic framework that does not decouple keys from applications, indicating that there are no issues in the design that hinder performance. Moreover we show that alternative designs consisting of user-space only components will have performance degradation, caused by the inter-process communication overhead. Thus we believe that our software implementation provides a practical additional layer of protection for cryptographic keys.

9 Acknowledgments

The authors would like to thank Steve Grubb for the inception of this project, Jan Filip Chadima for implementing a user-space interface to the NCR framework, Stephan Müller, Tomáš Mráz, Óscar Repáraz, Andreas Pashalidis, Nes-sim Kisserli, Jan Cappaert and the anonymous referees for valuable comments and feedback. This work was supported in part by the Research Council K.U.Leuven: GOA TENSE, by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy) and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen) SBO project.

References


This section describes in detail how the proposed framework could be used to protect the private keys used to identify a TLS [11] server. The approach is inspired by the security target format defined by the common criteria standard [17].
A.1 Problem definition

The TLS 1.2 protocol uses a public-private key pair to identify the server and avoid man-in-the-middle (MITM) attacks: the public key is included in a certificate signed by a trusted third party and provided to the client as part of the TLS connection establishment process. The protocol can use either a RSA key pair, or a Diffie-Hellman key pair. Although the algorithm and key exchange method choice affects internal operation of the TLS handshake, for our purposes we can treat both kinds of keys in the same way: the private key must be available to the TLS server, and disclosure of the private key to an adversary allows him to successfully mount a MITM attack impersonating the legitimate server.

As Section 4.1 describes, protection of session key material does not provide a meaningful improvement in security, and an adversary is necessarily able to mount a MITM attack, or an equivalent information disclosure attack, during the time the TLS server is under the adversary’s active control.

We therefore consider the following components of the problem of protecting a TLS server.

T1 Threat to be mitigated: An adversary could impersonate the TLS server for an unbounded time. This is based on the capabilities of the first adversary described in Section 3.1.

T2 Necessary functionality: The TLS server must have enough access to the private key identifying the server (referred to simply as “the key” from now on) to be able to serve legitimate requests.

We also assume that the TLS private key is managed by the administrator.

A.1.1 Security objectives

To handle the above-described problem, we have chosen a set of specific objectives that we want the TLS server deployment to meet fully.

O1 The TLS server process can remain compromised only for limited time.

O2 Only the TLS server process and processes started by the administrator can perform operations using the key.

O3 An adversary who controls the TLS server can not export the key to a different computer.

O4 Processes of the administrator are trusted not to reveal the key.
O5. The TLS server has enough access to the key to be able to perform RSA decryption, RSA or DSA signing and Diffie-Hellman shared secret establishment, depending on TLS key exchange method.

These objectives constitute one possible approach to handling the problem set out in Section A.1: O1–O4 are together designed to mitigate threat T1, and O5 to provide functionality required by T2.

Whether, and how effectively, the chosen objectives handle the problem set, is a question that must be assessed by human judgement.\textsuperscript{15}

A.2 Requirements and policies

Finally, we design a set of \textit{requirements} on the cryptographic module described in this article and \textit{policies} that must be enforced by the server’s administrators by system configuration or setting up human processes. The requirements and policies were chosen in a way that allows us to demonstrate that all objectives were met in a semi-formal way.

We use the following requirements which are met by the module and were verified by manual inspection of the source code:

R1. The only ways to make a pre-existing key available as a /dev/ncr key object is to load it using the key storage component, unwrap it, and to import it from a plaintext representation.

R2. The wrapping and exportable flag restrictions are correctly enforced.

R3. No other user than administrator is able to set the wrapping or exportable flag of an existing key.

R4. No user other than administrator is able to make a copy of an existing non-exportable key with one or both of the flags set if they were not set on the original without having access to a key with the wrapping flag set.

R5. A key wrapped for storage identifies users allowed to unwrap or to perform operations using the key, and this restriction is correctly enforced.

The following policies are the responsibility of system administrators:

P1. Any TLS server compromise is discovered and fixed in a bounded amount of time.

P2. The TLS server runs under a dedicated user account.

\textsuperscript{15} For example, the above set of objectives ignores ways to mount threat T1 without attacking the computer on which the TLS server runs, such as discovering the private key from the public key by brute force attacks, or obtaining a copy of the key from a key escrow system.
P3 The key, wrapped for storage, is stored in a file that is accessible to the user account of the TLS server.

P4 The key is not wrapped, using the storage master key or any other key, in files that are accessible to any other user account, except perhaps for the administrator.

P5 The key wrapped for storage in P3 above identifies the user account of the TLS server, and no other account, as allowed to unwrap the key.

P6 The storage wrapping master key is available at most to the kernel and processes running as administrator on the TLS server computer.

P7 Processes of the administrator are trusted not to reveal the key.

P8 The key wrapped for storage in P3 above does not have the exportable flag set.

P9 The TLS server does not have access to any key with the wrapping flag set.

P10 The key is not stored in any file unencrypted, nor encrypted with a key available to processes not running as administrator.

A.3 Meeting security objectives

We can now demonstrate that the requirements and policies from Section A.2 are sufficient to implement the objectives we have chosen in Section A.1.1:

We start with two auxiliary claims:

L1 If a key does not have the wrapping or exportable flag set, then the TLS server can not wrap using the key, or export the key, respectively: ensured by combination of R2, R3, R4, P9.

L2 An adversary who controls the TLS server can not extract the key material to user space:

- The key is not available exported in plaintext: P8, L1, P10.
- The key can not be extracted through the key wrapping mechanism: P9, L1.
- The key can not be extracted using the storage wrapping mechanism: P6.

Moving on to the objectives:

- O1 is implemented by policy P1.
• To demonstrate O2 is implemented, consider:
  – Only the TLS server process and processes started by administrator can load the key into a /dev/ncri key object: R1, P4, P2, and P10.
  – An adversary who controls the TLS server can not make the key object available to other user accounts on the same computer: R5, P5, P9, L1.
  – An adversary who controls the TLS server can not extract the key material to user space: L2.

• To demonstrate O3 is implemented, consider:
  – The adversary can not extract the key material to user space: L2.
  – The adversary can not use the key wrapping mechanism to transfer the key material: P9, L1.
  – The adversary can not use the storage wrapping mechanism to transfer the key material: P6.

• O4 simply becomes policy P7.

• O5: The key is made available to the server by P3.
Nikos Mavrogiannopoulos was born in 1979 in Athens, Greece. He received a Diploma in Mathematics from the University of Ioannina, Greece, in 2004 and a Master of Science degree in Information and Security technology from the Technische Universiteit Eindhoven (TU/e), the Netherlands, in 2006.

Before joining the KU Leuven’s Computer Security and Industrial Cryptography (COSIC) group in 2010, he worked as a software engineer in companies in Greece and the Netherlands. He contributed to the Free Software Foundation, by maintaining and enhancing GnuTLS [65], a cryptographic library that implements the TLS protocol.