END-TO-END SECURITY OF INFORMATION FLOW

IN WEB-BASED APPLICATIONS

A Dissertation
Presented to
The Academic Faculty

by

Lenin Singaravelu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2007

END-TO-END SECURITY OF INFORMATION FLOW

IN WEB-BASED APPLICATIONS

Approved by:

Dr. Calton Pu, Advisor
College of Computing
Georgia Institute of Technology

Dr. Jonathon Giffin
College of Computing
Georgia Institute of Technology

Dr. Ling Liu
College of Computing
Georgia Institute of Technology

Dr. Karsten Schwan
College of Computing
Georgia Institute of Technology

Dr. Hermann Härtig
Fakultät Informatik
Technische Universität Dresden

Date Approved: August 22, 2007

ACKNOWLEDGEMENTS

I have relied on guidance, support and advice of many people in producing this dissertation. First and foremost, I am deeply indebted to my advisor Dr. Calton Pu, for his guidance and support throughout the Ph.D. process. His knowledge and vision proved to be instrumental in my development as a researcher. I am also grateful for his flexibility in allowing me to explore different areas, which broadened my horizons.

My work in Dresden over two summers was the starting point for this dissertation. I want to thank Dr. Hermann Härtig for giving me the opportunity to work with him in the beautiful city of Dresden. I have to thank the members of my thesis committee, Dr. Jonathon Giffin, Dr. Hermann Härtig, Dr. Ling Liu, and Dr. Karsten Schwan, for their insightful comments which helped me improve my work. I want to thank Dr. Ling Liu and Dr. Karsten Schwan for their guidance throughout my Ph.D. I also want to thank Dr. Charles Consel for the opportunity to work with him in Bordeaux, France.

My fellow students at Georgia Tech., T.U. Dresden and Bordeaux made the Ph.D. process an enjoyable and fun-filled one. Presentation sessions at DISL group meetings, CERCS seminars, brainstorming sessions and critical analysis of research papers helped me gain a better understanding of research. Discussions on everything under the sun helped me unwind and relax, while at the same time exposing me to whole new worlds. Last, but not the least, I have to thank my parents for their love, support and belief in me throughout my life.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

LIST OF TABLES viii

LIST OF FIGURES ix

SUMMARY xii

CHAPTER 1 Introduction 1

1.1. Terms and Definitions 2

1.2. Security Problems in Web-Based Applications 3

1.2.1. Vulnerable End-Point Software 4

1.2.2. Misbehaving Intermediate Services 5

1.3. Thesis Statement and Contributions 6

1.4. Organization 7

CHAPTER 2 The AppCore Approach 9

2.1. Information Flow in Web-based Applications 9

2.2. Overview of the AppCore Approach 10

2.2.1. Design Goals 11

2.3. Constructing AppCores 12

2.4. Hardware Requirements 13

2.5. Evaluating AppCores 14

2.6. Applications of the AppCore Approach 15

2.6.1. A Simple AppCore for an E-Commerce Transaction Client 15

2.6.2. General Applicability 21

CHAPTER 3 Client-Side AppCore For https-Based Applications 23

3.1. Information Flow in https-based applications 23

3.2. Challenges and Opportunities in Protecting Information Flow 25

3.3. Components of AppCore 28
3.3.1. The *https* Proxy 29
3.3.2. Trusted Viewer 31
3.4. Implementation 32
3.5. Evaluation 34
3.5.1. Security Properties of BLAC 34
3.5.2. Performance 35
3.5.3. Complications with Real-World Web Servers 38
3.5.4. Revisiting Design Goals 39
3.6. Discussion 40
3.6.1. Applicability to other VMM-based approaches 40
3.6.2. Server-Side Support 41

CHAPTER 4 A Server-Side AppCore For Web Service Platforms 42

4.1. Design of Web Service Platforms 43
4.2. Security Problems in Web Service Platforms 45
4.3. Components of the AppCore 47
4.3.1. Components taken from Legacy WSP 47
4.3.2. Message Splicer 49
4.4. Architecture of ISO-WSP 49
4.5. Application Level Support for the AppCore 51
4.5.1. Secure Functional Interface: A Restricted Interface for Remote Access . 52
4.5.2. Modifications to Data Structures 53
4.5.3. Modifications to Code 55
4.6. Implementation 56
4.7. Evaluation 58
4.7.1. Security Properties of ISO-WSP 58
4.7.2. Performance of ISO-WSP 61
4.7.3. Cost of Porting Application-level Code 66
4.7.4. Revisiting Design Goals 67
4.8. Discussion 68

4.8.1. Applicability to Other WSPs ... 68
4.8.2. Cost of Splitting Applications ... 68

CHAPTER 5 End-to-End Security In The Presence of Misbehaving Intermediate Services ... 70

5.1. Information Flow in Web Service Compositions ... 70
5.1.1. Electronic Prescription Application ... 71
5.1.2. Security Requirements in an Electronic Prescription Application ... 72
5.2. Need for Fine-Grained, End-to-End Security Framework ... 74
5.3. Overview of WS-FESec ... 75
5.4. Addressing Integrity Requirements ... 76
5.4.1. Composition of Integrity Groups ... 77
5.4.2. Signing IntGs ... 77
5.5. Addressing Confidentiality Requirements ... 78
5.5.1. Composition of Confidentiality Groups ... 78
5.5.2. Key Distribution ... 78
5.5.3. Establishing Secret Context between WS_S and WS_R ... 79
5.5.4. Source Web Service's (WS_S) knowledge of Recipients (WS_R) ... 79
5.5.5. Encrypting ConfGs ... 79
5.6. Interaction between Integrity and Confidentiality Requirements ... 80
5.7. Evaluation ... 81
5.7.1. Security Properties of WS-FESec ... 81
5.7.2. Performance ... 83
5.7.3. Interaction with Composition Languages ... 85
5.7.4. Limitations of WS-FESec ... 86

CHAPTER 6 Related Work ... 87

6.1. Protection against Vulnerable Software ... 88
6.1.1. Malicious Software ... 88
6.1.2. Reducing Vulnerabilities in Software ... 88
6.1.3. Refactoring Software ... 90

6.1.4. Information Flow Restrictions ... 90
6.1.5. Browser Defenses ... 92
6.2. Protection from Misbehaving Intermediate Nodes ... 93
CHAPTER 7 Conclusion ... 95
7.1. Summary ... 95
7.2. Future Work ... 97
REFERENCES ... 100

LIST OF TABLES

Table 1: Modules in the Mozilla Browser [14] 16

Table 2. Complexity Comparison of E-Commerce Transaction Client AppCore and Mozilla. Level of shading indicates functional equivalence of components. 20

Table 3: Execution time for an E-Commerce Transaction 21

Table 4: Complexity Comparison of BLAC and a Linux-based software stack. The Linux kernel includes networking, file system, IO, memory management support. L4Env includes servers for bootstrapping the system and resource managers for memory and devices, IO manager, window manager and ABI support. 35

Table 5. Dataset for Measuring Page Access Times. Traces were generated using the WebScarab proxy [16] and the Internet Explorer web browser. Caching was enabled in the browser and the cache was cleaned before generating the trace. 36

Table 6. List of some WS-* extensions and the functionality provided by each of them [13] 45

Table 7. Comparison of Source Lines of Code (SLOC) and McCabe's Cyclomatic Complexity (MCC) of the T-WSP and the Axis2 WSP along with its extensions. All numbers were generated using the JavaNCSS tool [11]. 60

Table 8. Number of Lines of Code added or modified when porting legacy applications to ISO-WSP. 67

Table 9. Structure of an Electronic Prescription [9]. Superscripts indicate classes of information. Pt: Patient, Phy: Physician, Pharm: Pharmacy, Ins: Insurance, Drug: Drug. 72

LIST OF FIGURES

Figure 1. Overview of Web-Based Applications. In a traditional application, there is a single service provider processing a request. Web Service compositions typically have more than one service provider processing a request. 1

Figure 2. AppCore and TCB for an E-Commerce Transaction Client. Shaded boxes represent Trusted Components. Red lines denote untrusted components and untrusted communication links. 18

Figure 3. Stages in an https-based application; collectively referred to as a Session. Shaded boxes indicate communication over SSL. Partially shaded boxes indicate transition between SSL and plain-text modes. Numbers indicate sequence of operations. 24

Figure 4. Architecture of the BLAC system. High sensitivity information, as identified by the text patterns, is directed to a small trusted viewer and the rest of the information is directed to a legacy software stack. 33

Figure 5. Comparison of File Access Times. 36

Figure 6. CDF of HTML Page Access Time based on HTML pages from our traces. As expected, pages with fewer fragments (not shown on graph) had lower page retrieval times. 37

Figure 7. Web Services Stack. From W3C's Web Services Architecture specification [9]. 44

Figure 8. SOAP Processing Model in Apache Axis2. 44

Figure 9. Indirect Access in the Axis2 WSP. All handlers have access to service context, which contains, amongst others, parameters for the encryption key. A malicious handler can replace the encryption key as shown in the highlighted line. 46

Figure 10. Architecture of ISO-WSP. Shaded boxes represent Trusted Components. Trusted Components execute in a separate protection domain. 50

Figure 11. Partial Code Listing for a Payment Processing Service. Code generation process uses XSLT templates. Boxed area indicates modifications specified by developer. Highlighted areas indicate code reuse. 54

Figure 12. Securing Information Flow in ISO-WSP. The T-WSP replaces security-sensitive data items with protected data items or dummy data items before passing them on to the U-WSP. Secure Functional Interface (SFI) is discussed in Section 4.5.1 59

Figure 13. Remote Call Overhead. Java RMI Round Trip Time vs. Message Size 62

Figure 14. Performance Comparison of Parsers 63

Figure 15. Comparison of security processing costs for an incoming message. All numbers in microseconds. Numbers in parentheses indicate the 95% confidence interval. 64

Figure 16. Comparison Web Service Throughput with a mix of differing Security Configurations. Interpretation of configuration: WS-Security + Plain at 1, implies that 1% of calls were made to the WS-Security-enabled implementation and 99% of calls were made to the Plain text implementation. 65

Figure 17. Electronic Prescription Application. Patient contacts aggregator to identify the closest, cheapest or fastest delivery pharmacies. Shaded boxes indicate data items stored by each entity. Broken line indicates personal interaction. 71

Figure 18. Details for the CallbackReference Type 80

Figure 19. Performance with Multiple Signature per Message 84

Figure 20. Multiple Encryptions per message. Single Channel refers to encrypting all stock prices in a response message with the same key. In Multi-Channel, every stock price in the message is encrypted with a separate key, creating multiple security levels in the same message. Encryption keys are wrapped with RSA. 85

SUMMARY

Web-based applications and services are increasingly being used in security-sensitive tasks. Current security protocols rely on two crucial assumptions to protect the confidentiality and integrity of information: First, they assume that end-point software used to handle security-sensitive information is free from vulnerabilities. Secondly, these protocols assume point-to-point communication between a client and a service provider. However, these assumptions do not hold true with large and complex vulnerable end point software such as the Internet browser or web services middleware or in web service compositions where there can be multiple value-adding service providers interposed between a client and the original service provider.

To address the problem of large and complex end-point software, we present the AppCore approach which uses manual analysis of information flow, as opposed to purely automated approaches, to split existing software into two parts: a simplified trusted part that handles security-sensitive information and a legacy, untrusted part that handles non-sensitive information without access to sensitive information. Not only does this approach avoid many common and well-known vulnerabilities in the legacy software that compromised sensitive information, it also greatly reduces the size and complexity of the trusted code, thereby making exhaustive testing or formal analysis more feasible. We demonstrate the feasibility of the AppCore approach by constructing AppCores for two real-world applications: a client-side AppCore for https-based applications and an AppCore for web service platforms. Our evaluation shows that security improvements and complexity reductions (over a factor of five) can be attained with minimal modifications to existing software (a few tens of lines of code, and proxy settings of a browser) and an acceptable performance overhead (a few percent).

To protect the communication of sensitive information between the clients and service providers in web service compositions, we present an end-to-end security framework called WS-FESec that provides end-to-end security properties even in the presence of misbehaving intermediate services. We show that WS-FESec is flexible enough to support the lattice model of secure information flow and it guarantees precise security properties for each component service at a modest cost of a few milliseconds per signature or encrypted field.

CHAPTER 1

INTRODUCTION

Web-based applications are increasingly being used to perform security-sensitive tasks. The U.S. Census Bureau estimated that online retail sales generated over 110 billion dollars in revenue in 2006 [26]. It also estimates that e-commerce sales are growing at the rate of around 20% every year. According to a comScore Inc. study [6], nearly 40 million people in the U.S. used online banking services in 2005. The growing popularity of online applications has also resulted in increasing online fraud. A CyberSource study [7] reports that online fraud resulted in losses over $ 2.8 billion in 2005.

Figure 1 provides an overview of web-based applications. One variation from traditional web-based applications is the presence of more than one service provider for a single application. This situation arises in the domain of Service Oriented Computing (SOC), where standards such as SOAP and WSDL have facilitated the development of

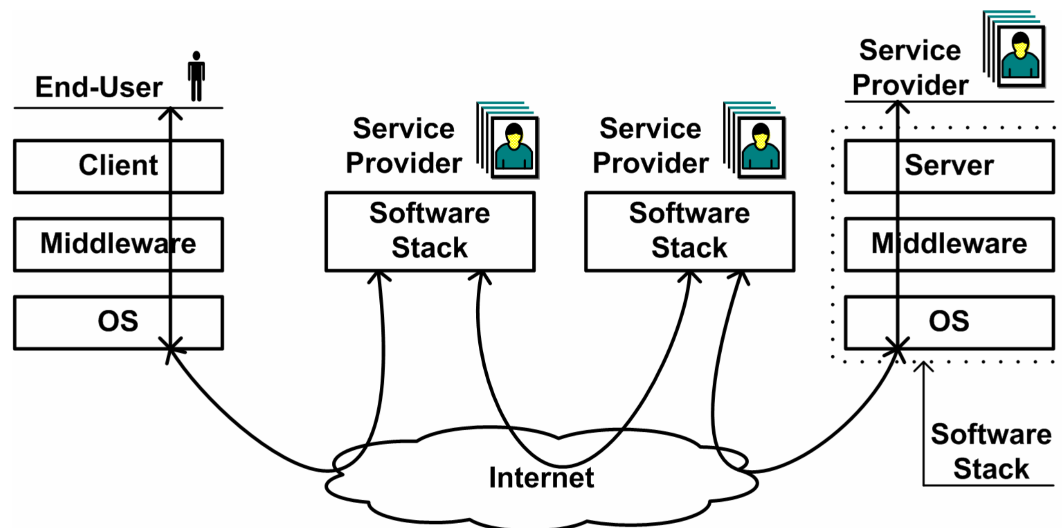

Figure 1. Overview of Web-Based Applications. In a traditional application, there is a single service provider processing a request. Web Service compositions typically have more than one service provider processing a request.

composite applications, called web service compositions, where service providers provide value added services by aggregating, filtering or annotating content from other service providers. A web-based application involves exchange of information between the end-user and the various service providers. In applications such as electronic commerce or online banking, a part of this information is deemed as security-sensitive and hence has to be protected from modification or unauthorized access.

Maintaining the confidentiality and integrity of sensitive information requires an end-to-end protection mechanism, where the ends are the *end-user* and the *service provider*. From the figure, we see that protection mechanism not only has to deal with attacks on the network, it also has to deal with attacks on end-point software stacks and misbehaving intermediate service providers.

1.1. Terms and Definitions

This dissertation deals with the following security requirements of web-based applications:

- *Integrity*: Only authorized entities can modify data or any modifications by unauthorized entities can be detected.
- *Non Repudiation of Origin*: The originator of the message is prevented from denying having sent the message.
- *Confidentiality*: Only authorized entities have access to data.
- *Confinement*: Data is distributed only to authorized entities.

In an interaction between a client and a server, any data item that requires one of the above defined security properties to hold is termed as a *sensitive* data item.

- *Trust*: Trust in a service is the expectation of a client that the service provides the above defined security properties for all its sensitive data items. An expanded trust model for web services, as defined in Hoffman et al. [71], relies on additional properties like usability, privacy, availability and reliability, audit and verification, user expectation

and feedback from the clients. As a starting point, we limit the definition of trust to satisfaction of security requirements.

System components are broadly classified into two categories: *Trusted Components* and *Untrusted Components*. Trusted components are entities that act in a trustworthy manner, i.e., they do not modify data inappropriately, they do not reveal data to unauthorized entities, and they perform their tasks correctly. We further assume that trusted components cannot be compromised. On the other hand, the behavior of untrusted components is not constrained in any way. However, in some instances, the behavior of untrusted components can be controlled by trusted components, so as to minimize the damage due to misbehaving components, e.g., sandboxing untrusted components.

From their respective definitions, it is clear that only trusted components should handle sensitive data items in their plain text or unprotected representations. Therefore, we would like to minimize the size and complexity of such components in an attempt to reduce the attack profile of the application and facilitate exhaustive testing or formal verification. Generally, untrusted components cannot be allowed access to sensitive data. However, untrusted components can access sensitive data that is protected with the help of trusted wrappers. *Trusted wrappers* are components that allow the reuse of untrusted components without compromising security objectives, e.g., SSL is a trusted wrapper that allows the use of an untrusted Internet to exchange information.

1.2. Security Problems in Web-Based Applications

Security protocols such as SSL [61], TLS [53], https [94] and WS-Security [15] have been developed to protect the confidentiality and integrity of sensitive information in web-based applications. These protocols protect data flow from the client machine to the server machine and vice versa, in the process assuming that the client and server software are trustworthy. The security protocols, except for WS-Security, provide coarse-grained, point-to-point protection, which is inadequate in the presence of intermediate

services. WS-Security, though designed with web service compositions in mind does not address end-to-end security issues, especially in the open environment envisioned by SOC.

1.2.1. Vulnerable End-Point Software

End-points in the current system, both the client and the server, run large and complex software that suffer from multiple vulnerabilities. On the client side, the Internet browser is commonly used to carry out security sensitive tasks such as performing e-commerce transactions and online banking. A full-fledged web browser is a large piece of software and is capable for performing a large number of tasks, e.g., the Mozilla browser contains over 1 million lines of code (LOC) and it can browse http, https sites and local filesystems. With appropriate plugins, it can also play multiple formats of audio and video. The browser suffers from multiple vulnerabilities including arbitrary code execution, information leak vulnerabilities and security system bypass [21][22]. Attackers have exploited these vulnerabilities in a variety of ways: installing extensions that snoop for logins and passwords to online banking sites [34] and compromising browsers' security features like the yellow lock symbol to carry out phishing attacks [23]. In addition, several other attacks have been demonstrated that allows an attacker to trick the user into uploading sensitive files [22], execute arbitrary code and bypass security restrictions like Zones in Internet Explorer [21].

With the rise in popularity of web services, web service platforms like Axis, Microsoft .NET, IBM Websphere, JBoss are being used on both the server and client ends to support web services. These web service platforms typically implement the web services framework as specified by W3C [30]. This includes basic messaging functionality and support for critical extensions like WS-Addressing, WS-Security. In addition, they also provide support for programmer defined extensions for logging, admission control or load balancing. Implementing all these functionalities have resulted

in a single, large multi-threaded middleware that is difficult to test or verify. For example, Axis2 contains about 23,000 LOC. In addition, Axis2 has to rely on external libraries for supporting extensions such as WS-ResourceFramework (20,000 LOC), WS-ReliableMessaging (11,000 LOC) amongst others. Axis2 also provides support for programmer defined extensions which can be instantiated, unloaded or reloaded at runtime. Web service platforms suffer from multiple vulnerabilities which allow an attacker to inject arbitrary code or bypass security mechanisms [19][20]. These vulnerabilities allow an attacker to gain access to sensitive data or modify system settings on the server and gain control of the platform.

1.2.2. Misbehaving Intermediate Services

Web service compositions introduce two challenges to end-to-end integrity and confidentiality. First, intermediate services in a composition are *selectively* allowed to look at or modify the results of the data producing web services. While this enables value-adding transformations like annotation, filtering, or translation, final data consumers will have to trust these intermediate services to maintain the integrity of content.

Secondly, web services operate in an open environment, i.e., (a) there are a large number of service providers on the Internet and (b) service providers operate at different levels of trust. One can assume transitivity and trust the next web service in the chain of invocation but given the large number of services on the Internet, and the various permutations and combinations in the way they are composed, “full transitivity” is not a feasible or a scalable alternative. On the other hand, for the same reasons, it is also not feasible to expect a data consumer to trust, or even be aware of all component services used in composing a particular result. Operating in an open environment also implies that services in a composition often operate at different security levels with respect to the data producing services. The confidentiality policy of a data producing web service dictates

the visibility of its information. Hence, all components of a message should not be available to all services in a composition.

Security protocols such as SSL and TLS are coarse-grained. They provide complete access to message contents to intermediate service and hence, misbehaving services can compromise the confidentiality or integrity of sensitive information. WS-Security provides support for fine-grained encryption and signatures; however, it does not address end-to-end security issues in open environments. For example, we present an electronic prescription system (Section 5.1.2) where the originator of sensitive information does not know of all potential recipients at message generation time and currently, WS-Security does not provide mechanisms to address this issue.

1.3. Thesis Statement and Contributions

This dissertation addresses challenges discussed in Section 1.1 in providing end-to-end security in web-based applications. The main thesis of this dissertation is that improving end-to-end security properties of information flow in web-based applications requires small and simple end-point software and extensions to existing security protocols that account for the new class of loosely-coupled composite applications. Towards this end, this dissertation makes the following contributions:

- First, to counter the problem of large and complex end point software, we present the AppCore approach [106] which uses manual analysis of information flow to split existing software into two parts: a small and simple trusted part that handles security-sensitive information and a legacy, untrusted part that handles non-sensitive information without access to sensitive information. This isolation of sensitive information avoids many common and well-known vulnerabilities in the legacy software that compromised sensitive information. Additionally, the AppCore approach also greatly reduces the size and complexity of the trusted code, thereby making exhaustive testing or formal analysis more feasible. The AppCore approach also supports the reuse of non-

sensitive legacy interfaces at user, programming and remote access levels, minimizing user discomfort and changes to existing software.

- Second, we demonstrate the practical feasibility of the AppCore approach by constructing AppCores for two real-world applications: a client-side AppCore for https-based applications [104] and a server-side AppCore for web service platforms (WSPs) [107]. The client-side AppCore uses an https proxy and a small and simple trusted viewer to handle sensitive information such as passwords, with a legacy browser handling non-sensitive other information. The server-side AppCore carries out secure processing of sensitive information and the legacy WSP performs the bulk of web service functionality. In each case, sensitive information is only accessible by a small and simple AppCore (over a factor of five reduction in software complexity), thereby protecting sensitive information from common vulnerabilities in the legacy software. Our evaluation shows that these security improvements can be attained with minimal modifications to existing software (a few tens of lines of code, and proxy settings of a browser) and an acceptable performance overhead (a few percent).

- Third, to protect the communication of sensitive information between the clients and service providers in web service compositions, we present an end-to-end security framework called WS-FESec [105]. WS-FESec uses fine-grained signatures and encryption in conjunction with developer input on security requirements of various parts of the message to provide end-to-end security in a loosely-coupled, open environment. We show that WS-FESec is flexible enough to support the lattice model of secure information flow and it guarantees precise security properties for each component service at a modest cost of a few milliseconds per signature or encrypted field.

1.4. Organization

The rest of the dissertation is organized as follows: Section 2 presents the AppCore approach to building small and simple components to handle security-sensitive

data. Section 2 also briefly describes the construction and implementation of a simple AppCore for e-commerce transaction client. Sections 3 and 4 discuss in detail the design and construction of AppCores for https-based client-side software and web service platforms. Each section also evaluates the AppCores according to the design goals laid out in Section 2. Section 5 discusses the WS-FESec, a fine-grained, end-to-end security framework for maintaining confidentiality and integrity in web service compositions. The section also describes how WS-FESec can be used to support the lattice model of information flow and satisfy the security requirements of the electronic prescription system. Section 6 discusses the related work and Section 7 concludes the dissertation and discusses future research directions.

CHAPTER 2

THE APPCORE APPROACH

The large size and high complexity of end-point software in security-sensitive applications has hindered their testability and resulted in systems with multiple security vulnerabilities. To combat this problem, we present the AppCore approach that refactors existing software into two parts: a small and simple part, called the AppCore, operating on security-sensitive data items and the rest of the legacy software that is used for all other operations. We expect the AppCore to be smaller and simple than the original software, thereby simplifying the analysis and testing process. At the same time, by reusing the legacy software to operate on security-insensitive data, we expect to reuse legacy code and interfaces.

We first discuss information flow in web-based applications and present the reasoning for refactoring software. Next we present an overview of the AppCore approach and the design goals for constructing AppCores. Then, we discuss the process of AppCore construction and present our evaluation methodology. Finally, we discuss the applications of the AppCore approach. We present a detailed example of an AppCore for an e-commerce transaction client and discuss the general applicability of the AppCore approach.

2.1. Information Flow in Web-based Applications

The selective use of security protocols like SSL in interactions between servers and clients indicates that information flow in web-based applications consists of atleast two distinct classes of data items: *security-sensitive* and *security-insensitive*. However, this distinction is not propagated to the middleware or the application layer. The same software is typically used to handle both classes of data. Therefore any vulnerability in

the software, irrespective of whether or not the vulnerable component needs access to security-sensitive data, can be used to compromise security-sensitive data, e.g., by exploiting a vulnerability in compressed HTML module in Internet explorer, an attacker installed a browser help object that sniffs for passwords for online banking sites before they can be encrypted with SSL/TLS [34].

Not all components of an application are essential to maintaining the confidentiality and integrity of information flow. Components are security-insensitive because of one of two reasons: first, these components are not involved in processing security-sensitive information, e.g., plugins or extensions to browsers are typically not invoked during a security-sensitive operation like online banking. Secondly, these components only process protected security-sensitive information, e.g., the HTTP protocol implementation in browsers and web service platforms process security-sensitive information, after they have been encrypted with SSL, TLS or WS-Security. Therefore, these components are not essential to maintaining the confidentiality or integrity of information flows.

There are two advantages to isolating components that handle security-sensitive data items in a small application that executes in a separate protection domain from the rest of the application: first, by limiting flow of security-sensitive data items to the small application, vulnerabilities in the rest of the application can no longer be used to compromise security-sensitive data items. Secondly, we reduce the size and complexity of software that has access to security-sensitive data. This makes the software more amenable to exhaustive testing and formal verification.

2.2. Overview of the AppCore Approach

The AppCore approach recognizes that not all components of an application are essential to maintaining the confidentiality and integrity of information flow. Currently, we assume that information flow in web applications is composed of two classes of data

items: security-sensitive and security-insensitive data items. Security-sensitive data items can be downgraded into security-insensitive data items via trusted wrappers. Trusted wrappers are components that allow the reuse of untrusted components without compromising confidentiality and integrity objectives. Trusted wrappers implement downgrading operations such as digital signatures, encryption and down sampling (e.g., down sampling of multimedia, one bit result of a comparison of secure password and untrusted input).

Components that require access to security-sensitive data items are called as *trusted components*. These components will have to be trusted as they can modify sensitive data items before they can be protected with trusted wrappers. These components are extracted from the original application and composed into an AppCore. The rest of the application is denoted as *untrusted* and is modified to invoke the AppCore for operating on security-sensitive data items. The AppCore is then executed in a separate protection domain from the rest of the application.

2.2.1. Design Goals

The construction of AppCores must take into account the following design goals:

1. *Limit flow of Security-Sensitive Information to Trusted Components*: This is the key security requirement that drives the construction of AppCores. However, note that untrusted components can be allowed access to protected security-sensitive information through the use of trusted wrappers.

2. *Minimize size and complexity of Trusted Components*: Reducing the size and complexity of security-sensitive components makes the code more amenable to exhaustive testing or formal verification.

3. *Allow use of legacy software for security-insensitive tasks*: Using the legacy application with its familiar user-interface for security-insensitive tasks minimizes

discomfort for the end-user. By using legacy software for security-insensitive tasks, developers and users have access to the complete set of features of the original software.

4. *Reuse interfaces as far as possible*: Since both the AppCore and the untrusted software will be involved in performing a task, they have to communicate with each other. Reusing existing interfaces allows for the AppCore to be easily integrated with the rest of the application. In the case that the AppCore has to interact with a remote entity, conforming to existing interfaces (e.g., SOAP over HTTP for web service platforms) allows AppCores to interoperate with legacy entities.

5. *Minimize Performance Loss*: Since our restructuring results in two sets of components (one trusted AppCore and the untrusted application) coordinating with each other to perform a task that was previously accomplished using a single set of components, there will be additional costs for communication and coordination. These overheads should be minimized to make AppCores viable.

2.3. Constructing AppCores

The process of extracting an AppCore from an existing application can be broadly divided into three stages: (1) analysis of the application to identify trusted components, (2) extracting the identified components and composing them into an AppCore and (3) modifying the original application to use the AppCore for security sensitive tasks.

The function of the analysis stage is to identify components that handle security-sensitive data items. If the application has reasonable documentation, this step can be accomplished by analyzing the documentation, e.g., the Mozilla browser for the e-commerce transaction client scenario (Section 2.6.1.1) is well documented and has clearly-defined modules making identification easier. Otherwise, we have to manually identify security-sensitive data items based on domain knowledge (e.g., private keys, password). Once identified, dataflow analysis can be used to track the flow of security-sensitive data items through the application (one such approach is discussed in [41]).

In the next stage, we extract the security-sensitive components and integrate them into a standalone AppCore. There are two factors that control component integration: First, to the greatest extent possible, we want to reuse the interfaces between the security-sensitive components and the rest of the application, and second, we want to constrain the security-sensitive components to perform only the requisite security-sensitive tasks. Reusing existing interfaces simplifies the reintegration of the AppCore with the application. Constraining security-sensitive components involves modifying or rewriting components to perform the necessary security-sensitive task with least amount of software, e.g., substituting Mozilla's NSS module with a bare-bones SSL library, MatrixSSL, provides size savings over two orders of magnitude. However, this could also break existing interfaces and increase the cost of reintegration. Thus we have two conflicting factors influencing component integration.

The final stage consists of going through the components in the original application and replacing the existing function calls to security-sensitive modules with calls to the new AppCore. If the original application has an extension architecture (e.g., browsers and web service platforms), this is a straightforward process as we can connect the application to the AppCore using extensions.

2.4. Hardware Requirements

Even though, we prescribe the use of AppCores to perform security-sensitive tasks, an attacker can employ interface spoofing attacks to trick an end-user into interacting with an untrusted application instead of the corresponding AppCore. Similarly, a remote entity can be tricked into revealing sensitive information to an untrusted application. The AppCore approach relies on hardware support as described in the specifications of the Trusted Computing Group [25][24] to limit flow of security-sensitive information. Specifically, it relies on authenticated booting to establish a chain of authentication for the executing software. The authentication chain can be used to

reassure a remote party about the application being executed (*remote attestation*). Locally, a user can compare the chain of trust with a secure copy (e.g., a copy on a USB key) before logging into the system. Once logged in, the window manger (a trusted component) uses the chain of trust to inform the user about the trustworthiness of the in-focus application. *Sealed storage* protects the confidentiality of the cryptographic keys that are foundations for the security of the rest of the system.

2.5. Evaluating AppCores

We evaluate AppCores along the lines of the design goals mentioned in Section 2.2.1. The goal of our evaluation is to show that the security improvements compare favorably with the performance overheads imposed by the AppCore. We evaluate the performance impact of AppCores by building a prototype system and running appropriate micro and macro benchmarks. Based on the design of each AppCore, we present arguments as to how we restrict the flow of sensitive information to the Trusted Components. We also show that the AppCore is invoked only when operating on sensitive data, thereby allowing the reuse of legacy interfaces and components.

In addition to the qualitative arguments describing security improvements, we also present quantitative arguments concerning the reduction in software complexity of Trusted Components. We use two well known software complexity metrics - Source Lines of Code (LOC), and McCabe's complexity metric (MCC) [85] to illustrate the reduction in software complexity. Many software engineering studies have shown that the LOC metric is roughly correlated with the number of defects [61][103], which may be relatively imprecise, but serves as a useful baseline measure [37]. MCC is based on McCabe's definition [85] of a control flow complexity metric. It is measured per function and it gives the number of distinct execution paths in a given function. Intuitively, it represents the minimum number of tests that need to be carried out on that function to verify control flow properties.

2.6. Applications of the AppCore Approach

The next two chapters provide detailed examples of the AppCore approach, one for a client-side application and another for web service platforms, which is an integral part of the software stack on both the client and the server sides. In this section, we discuss the broad applicability of AppCore approach based on our work [106] and examples from closely related work by others.

2.6.1. A Simple AppCore for an E-Commerce Transaction Client

The most popular tool for carrying out an e-commerce transaction is a browser. Browsers perform two critical functions for e-commerce – they display content, in a format determined by the merchant, to the user and they accept user input and pass them along to the merchant. Data transfers can be protected using a transport layer security protocol like SSL or TLS. In a typical e-commerce transaction, initially, the customer builds up a shopping cart, which can involve multiple rounds of merchant-customer interaction. Next, the customer decides to finalize the transaction. At this point most merchants use a transport layer security protocol to protect any further communication. Once a secure layer has been established, the customer provides the merchant with payment information or unique login information to retrieve a profile. The merchant verifies this information and finalizes the transaction.

The large code base of browsers and the support for extensions via tightly integrated (i.e. executing in the same address-space) plugins hinder effective testing of browsers. Browsers are sources of multiple vulnerabilities including arbitrary code execution and security bypass [21][22]. Browsers also suffer from spoofing vulnerabilities where the attacker is able to fool the user into mistaking an arbitrary site for a trusted site. Attackers have successfully exploited these vulnerabilities to install malicious plugins, and steal private information like passwords and credit card information. These vulnerabilities illustrate the risk in using a browser to carry out

Table 1: Modules in the Mozilla Browser [14]

Type	Example Modules
Main Browser	Browser, Portable Runtime, Display Widgets, New HTML Parser.
Security	Security (NSS & JSS).
Scripting	Javascript Engine, Rhino, Live Connect.
Security-Extras	Personal Security Manager, JS Security.
UI-Enhancements	Clipping & Compositing, Find as you type, ImageLib, accessibility.
Parsing-Extras	RDF, DOM, XML, XSLT, MathML.
Extras	I18N, URI Loader, Zlib, Qt support, Cookies, Plugins, Preferences, Update.

security-sensitive operations like online purchases. On the other hand, since a majority of merchants and consumers prefer to use the browser as a transaction tool, an effective solution must work within the framework of the browser.

2.6.1.1 Components of the AppCore

The security-sensitive data in an e-commerce transaction client is the user's payment and shipping information, the shopping cart that is displayed to the user, and the user's choice about the transaction. Upon analysis of the browser's components (listed in Table 1), we find that the following components are security-sensitive: the main browser, which contains the parser, the display, the security library and basic user-interface components. These components form a significant portion (over 50%, 500 KLOC) of the browser's code base. Composing them into an AppCore would result in a trusted application smaller than the original browser, but we can achieve better results by refining the selected components. We simplify the selected components by limiting their functionality. For example, constraining the language describing the shopping cart would allow for a smaller parser and display interface. Similarly, implementing only the most

used cryptographic mechanisms would significantly reduce the size of the security library. We now consider these components in more detail.

Browser Parser. This browser component has to understand a variety of HTML and XML standards with multiple variants. A shopping cart, for example, can be described in detail using sophisticated tags and templates. While this is an expected good usability feature, it introduces potential vulnerabilities due to growing code size. A cart described using the table tags in HTML would require a small and specialized parser. As shown in Section 2.6.1.3, this specialization helps us achieve significant reductions in the code size and complexity of the AppCore.

User Interface. There are many demands on a generic web browser; their functionality has to include the display of text and images in multiple formats, support novelties such as tabbed browsing and skins, etc. In contrast, a small interface designed for a shopping cart would focus on the unambiguous display of the cart content and a clear way to accept or decline the transaction. Our AppCore uses a text-box-based interface that displays the table and presents the user with a confirmation window. We were able to implement this functionality with less than 500 LOC. While our current interface is primitive, given the limited expressiveness of the cart data, we expect that even a more sophisticated interface would be orders of magnitude smaller than a generic browser.

Security Library. A library implementing either the SSL or TLS protocol is necessary to carry out secure transactions over the Internet. In our implementation, we use MatrixSSL [17], an SSL library with a small footprint originally developed for embedded systems. The library provides a minimal set of standard algorithms necessary for SSL. At less than 10 KLOC it is significantly smaller and less complex than SSL library in Mozilla (NSS module in Mozilla is over 180 KLOC).

We can see that in each case, we tradeoff functionality for a reduction in complexity. At one end of the spectrum is a large and complex browser that can be used

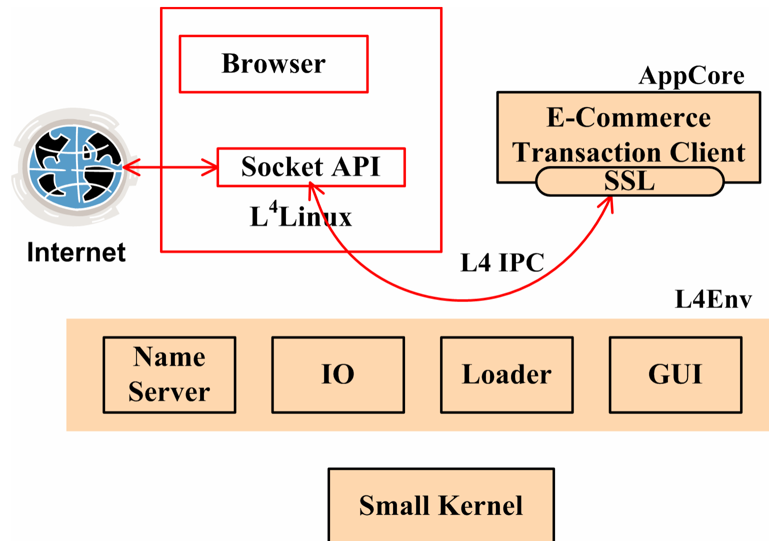

Figure 2. AppCore and TCB for an E-Commerce Transaction Client. Shaded boxes represent Trusted Components. Red lines denote untrusted components and untrusted communication links.

to accomplish multiple tasks. On the other end of the spectrum is a specialized client software that can perform only a single task. Selecting the appropriate tradeoff is dependent on technical factors such as size of code base that can be properly analyzed and tested and on non-technical factors such as user comfort levels. While designing metrics for evaluating tradeoffs is outside the scope of this thesis (Attack Surface Area [73] is an example of such metric), we feel that the E-commerce transaction client provides acceptable tradeoffs as it contains sufficient functionality for the user to carry out a non-trivial operation, but at the same time significantly reduces complexity.

2.6.1.2 Constructing the AppCore

The parser, security library and the user interface components are put together to form the AppCore for an E-Commerce Transaction Client. The user employs the AppCore to handle sensitive financial information and a legacy browser for the rest of the tasks. In an e-commerce transaction, the user first employs the browser to populate the shopping cart. When the user decides to check out, the web server contacts the E-Commerce Transaction Client AppCore instead of the legacy browser. The user then provides sensitive financial information to the AppCore, which encrypts the data using

SSL before transferring it to an untrusted network stack. We rely on authenticated booting and remote attestation to ensure that the user and the web server are communicating with the AppCore when dealing with financial information.

We implemented the E-Commerce Transaction Client AppCore on the Nizza security architecture [66][68]. Nizza is composed of four major parts: a small kernel; an execution environment consisting of trusted components (such as a name-server and window manager); an untrusted legacy standard OS with its applications; and security-sensitive applications. In our implementation, we use the L4 microkernel as the small kernel. The execution environment of the microkernel (called L4Env in Figure 2) provides essential system services such as naming, IO management and display management and hence has to be trusted. The browser runs as an untrusted application on a para-virtualized, untrusted commodity operating system (L^4Linux [67]). The E-Commerce Transaction Client AppCore is executed as trusted process, directly on top of the microkernel. The overall system is depicted in Figure 2. A point of interest is that the AppCore relies on an untrusted socket interface proxy to communicate with the external network. This is an example of the use of trusted wrappers: We use SSL in the AppCore to protect the confidentiality and integrity of the data before passing it over to the proxy. Hence we can afford to use an untrusted network stack for communication.

2.6.1.3 Evaluation

The Nizza architecture provides confidentiality and integrity of data at the operating system and middleware level. These properties are extended to the application layer by the E-Commerce Transaction Client AppCore. The AppCore maintains confidentiality and integrity of data by using a transport layer security protocol like SSL or IPSec to encrypt data before handing it over to untrusted components. The Nizza architecture also provides us with a trusted window manager that controls the top portion of the screen. The top of the screen, which acts as an unforgeable trust indicator, is used

Table 2. Complexity Comparison of E-Commerce Transaction Client AppCore and Mozilla. Level of shading indicates functional equivalence of components.

Component	LOC	Cumul. MCC
MatrixSSL	8,600	1,200
Mozilla-NSS	180,000	24,700
Custom Parser	200	35
HTML-Mozilla	19,000	3,100
AppCore	10,000	1,500
Mozilla v1.0	978,000	151,000

by the window manager to indicate the trust level of the in-focus application. We assume that the users are a part of the TCB, i.e., they are aware of the trust indicator when they give out sensitive data or perform security-sensitive operations. Since, untrusted applications cannot modify the trust indicator, interface spoofing attacks are minimized.

The main focus of the AppCore approach is to reduce the size of the trusted computing base at the operating system, middleware and the application layers. A smaller and simpler code base has two advantages – a smaller code base for testing and analysis and a smaller attack profile. Table 2 compares the software complexity of a generic browser-based approach and an AppCore-based approach. The AppCore simplifies the functionality of the security-sensitive code in two main components: a smaller SSL library and a very narrow subset of HTML to describe the cart. We observe the simplification of components provides considerable reductions in software size. The entire AppCore is about one hundredth the size of the Mozilla browser. Similar reductions are seen with the MCC metric. One concrete benefit of using simple Trusted Components is the lack of an extension architecture in the AppCore. The legacy browser still possesses an extension architecture, but the attacker can no longer exploit vulnerabilities in them to compromise sensitive information, as in [34].

Next, we measure the time to carry out a typical e-commerce transaction using the AppCore. The transaction client initiates an SSL connection and receives the cart from

Table 3: Execution time for an E-Commerce Transaction

Client-Server pairs	Linux-Linux	L^4Linux-L^4Linux	L4 - L^4Linux
Time (ms)	39.1	40.2	43.5
Stdev %	0.2	10.6	8.0

the web-server. It displays the cart and waits for the user's response. Depending on the user's response, the client either sends the user's payment and shipping information back to the server and finalizes the transaction or aborts it. Since we are interested in finding out the minimum time to execute the transaction, we eliminate user-interaction by assuming that the user always wants to accept the transaction. Table 3 lists the server-side execution time for a transaction over a loopback interface. Each column title represents the execution environment of the client and server respectively. The execution time for the trusted scenario (L4 - L^4Linux) is around 11 % slower than the Linux scenario. But in absolute terms, the execution time is less than 50 ms, which is insignificant when compared to user response time (order of seconds).

Finally, we discuss code and interface reuse in the AppCore. Since the AppCore is used only during the exchange of sensitive financial information, the user can employ legacy applications for other tasks, e.g., building up the shopping cart. However, we require the web server to use a different data format to carry out the transaction: specialized HTML instead of regular HTML. Hence the AppCore cannot be used with unmodified legacy web servers. We present a solution to this problem in the form of a more generalized AppCore for all https-based applications.

2.6.2. General Applicability

The AppCore approach is motivated by previous efforts on refactoring software with focus on security. Privilege Separation [90], PrivMan [76] and PrivTrans [41] are closest examples of the AppCore approach. All three attempt to refactor existing applications such that only a small portion of the code executes with high privileges. AppCore also has the same goal in mind; however, AppCore calls for further

simplification of the code executing with higher privileges providing significant complexity reductions.

AppCore approach has also been successfully applied to a VPN gateway (MikroSINA) [69][106] and an email signer (Enigmail Plugin) [106]. In the VPN gateway case, the AppCore approach reduced software complexity by a factor of 3, and in the email signer case, it reduced the complexity by a factor of 5. In a recent work, we also described mechanisms for building an AppCore for POP3 mail clients [104]. The AppCore construction process is similar to the one for the https-based clients (Chapter 3).

More recently, a similar approach has also been applied to device drivers, which typically operate in high privilege domains. MicroDrivers [62] attempts to refactor device drivers and splits them into a small kernel-mode component and a larger user-mode component that executes with lower privileges. In their study of network, SCSI and sound drivers, the authors noted that "critical functions", functions that need to execute in kernel mode, accounted for less than 30% of the functions. In their implementation of the e1000 network driver, the authors achieved a 75% reduction in size of code that executes in kernel mode.

CHAPTER 3

CLIENT-SIDE APPCORE FOR HTTPS-BASED APPLICATIONS

https is a URI scheme that augments the HTTP protocol with encryption (using SSL or TLS) and is widely used in many applications such as online banking and electronic commerce. The Web-Browser (browser for short) is the most commonly used client-side software for https-based applications. The user, typically a human, interacts with a Trusted Computing Base (TCB) that consists of an operating system, a window manager and various libraries in addition to a web-browser.

A typical web browser is a large and complex software that suffers from multiple security vulnerabilities. As shown in Section 2.6.1, we can design a simple AppCore to handle sensitive information. However, the solution requires the user to employ a functionally limited AppCore for a substantial duration. Information flow patterns in https-based applications provide opportunities to reduce the number of times a user has to interact with a functionally limited application. We first discuss information flow in https-based applications. Next we present challenges in protecting flow of sensitive information on the client side in https-based applications. We also discuss the opportunities afforded by https-based applications in designing a small and simple AppCore. We then discuss the components of a small AppCore for https-based application and present a prototype implementation, called BLAC, on top of the L4 microkernel [104]. Finally, we evaluate the security properties and performance of the resultant AppCore.

3.1. Information Flow in https-based applications

Figure 3 presents the stages in an interaction between a client and a server in a typical https-based application. We refer to the sequence of interactions as a session.

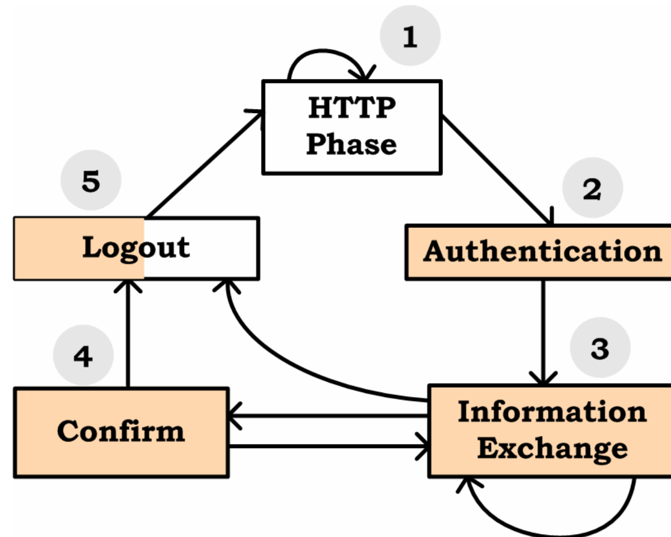

Figure 3. Stages in an https-based application; collectively referred to as a Session. Shaded boxes indicate communication over SSL. Partially shaded boxes indicate transition between SSL and plain-text modes. Numbers indicate sequence of operations.

Initially, the client and the server employ plain-text, HTTP-based communication. Therefore, information exchange is limited to non-sensitive information in this stage. This stage might be missing in servers that do not support the http URI. The client has to authenticate itself, typically using a login and a password, before it can access sensitive information. The authentication step is carried out using https URLs. The authentication information is encoded in a session identifier (the cookie), thereby enabling the user to access sensitive-data over multiple interactions without having to authenticate repeatedly. Upon successful authentication, the web server and the client exchange information over multiple interactions. At any point during the session, the user is allowed to logout, returning the interaction to unencrypted communication.

Each stage in a session follows the HTTP protocol. The client makes an HTTP request based on previous HTTP responses and/or user input. The server replies with an HTTP response. The client parses this response to generate more requests (e.g., for embedded images) or render it in a viewer. The client now waits for user input to generate a new request. In SSL enabled stages, requests and responses are transmitted over a secure channel (SSL or TLS). The client has to first establish an SSL connection

with the web server. The SSL connection could be a new connection or the restoration of a previously suspended connection. Next, the client transmits an HTTP request over the encrypted channel and receives an HTTP response. The SSL connection is then suspended or closed. From HTTP/1.0 onwards, more than one HTTP request and response can be exchanged before an SSL connection is closed or suspended.

Information flow in https-based applications can be classified into three categories: *Non-sensitive*, *Low-sensitivity* and *High-sensitivity* information. Of the three, the first one is accessed using http URIs and the later two are available only through https URIs. This three level classification has justifications in the realm of many online applications such as online banking and electronic commerce. In electronic commerce, the user is asked to choose shipping address, select a payment method, enter a new payment method (e.g., bank account or credit card information) and confirm the purchase. Clearly, information on a new payment method is more sensitive than shipping address. Since confirming a purchase modifies account status and places a financial burden on the user, we also treat user's choice on confirmation as high-sensitivity data. We argue that the user and the service provider want to use a more trustworthy software stack to handle such high-sensitivity information.

3.2. Challenges and Opportunities in Protecting Information Flow

There are two key challenges to protecting information flow on the client-side in https-based applications. First, current client-side software are large and complex, hindering analysis and testing. This results in Trusted Components with multiple critical security vulnerabilities. We already discussed the software complexity and vulnerability issues of browsers, a commonly used client-side application-level software for https-based applications, in Section 2.6.1.

The second challenge is the popularity and ubiquity of current software and interfaces. For example, web browsers are increasingly used in many online applications

including mail, spreadsheet and document editing. Users are very familiar with the user-interface (UI) and the functionality provided by the browser. Limiting the functionality or modifying the UI will reduce the appeal of the modified software. Similarly interfaces at various levels of the software stack are too widely used to modified or abandoned outright, e.g., API between OS kernels and application-level software can be enhanced with information flow primitives [70], but this approach will not be applicable to the large number of existing software. Information exchange protocols between service provider and client, e.g., HTTP protocol, too are well entrenched that solution requiring modifications to the protocol will have limited appeal.

Previous efforts have addressed the first challenge, large and complex software stacks, by proposing application-specific TCBs (e.g., Terra [64]). While this reduces the size and complexity of the TCB, it does not address complexity of middleware and application-level software. A simple version of AppCore for e-commerce transactions (Section 2.6.1 and [106]) extends this approach to all layers of the software stack by reducing complexity of Trusted Components of application-level software. However, reducing the complexity of Trusted Components requires modifications such as curtailing the functionality of the components and narrowing the interface exported by the components. Moreover, the E-Commerce Transaction Client AppCore required the use of the functionally-constrained AppCore for all https interactions. As opposed to transmitting financial information in e-commerce transactions, other https-based applications such as online banking can consist of a large number of user interactions, e.g., browsing accounts, looking at various reports and graphs, etc. In such a scenario, using functionally limited software for extended periods of time significantly inconveniences the user.

The Proxos approach [109] attempts to work around this problem by allowing software developers to specify trust in one of the interfaces – the OS-application (system call) interface – by splitting the interface into a trusted and untrusted part. The trusted

part of the interface is implemented by a trusted OS, whereas the untrusted part is implemented by a commodity operating system. By doing so, Proxos allows the reuse of current application-level software and systems software as far as possible, while at the same time minimizing modifications to the application-level software. However, large and complex client software such as browsers and middleware such as the display manager still need to be completely trusted.

Opportunities: Client-server also applications possess many properties that allow us to reduce complexity *and* reuse existing software and interfaces as far as possible. First, information flow in a session of interaction in client-server application contains data items with differing security and functionality requirements. As a concrete example, consider a session in an online banking application, where the user logs in, checks her account, modifies her account and logs off. One can argue that account status information is less sensitive than user authorization for account modifications, e.g., transfer of funds, ordering cheques or change of personal information. Some banks already recognize this difference in sensitiveness of information and ask the user for an authorization token, e.g., a one-time Transaction Authorization Number (TAN), for account modifications to be approved [8]. From a functionality point of view, account status information also requires a richer presentation format, e.g., spreadsheets and graphs, whereas account modification information typically shows up in a tabular format on a *confirm* page. Sensitive input consists of human input in the form of a sequence of keystrokes or mouse clicks.

We can leverage this difference in security and functionality requirements and construct small and simple components to handle security-sensitive information and handling the rest of the information in legacy components. Doing so allows us to reuse existing interfaces as far as possible.

Another opportunity arises from the functionality and security tradeoffs preferred by the user and the service provider. For example, the user might be willing to use legacy

components on her computer system to handle sensitive information, but prefers to use small and simple components to operate on all sensitive information when working on a public-access machine. Similarly, during a virus outbreak, the service provider might ask users to interact using smaller and simple components more extensively than otherwise. By providing the user (and the service provider, if necessary) with freedom in choosing the components, we can ensure that security requirements are balanced with the functionality requirements.

3.3. Components of AppCore

Previously, we noted that all data items exchanged using https do not have the same security requirements (Section 3.1). We leverage this difference in sensitiveness of data items to select between fully-functional, but complex legacy application-level software and small and simple but functionally constrained software. We allow users (or web servers) to determine the usage of the small and simple application-level software (called the Trusted Viewer) based on their security and functionality tradeoffs.

We use a novel approach to split the information flow between the two types of application level software. We use an https proxy, which has to be trusted, to determine the sensitiveness of incoming messages from the web server. Sensitiveness is determined by user-provided or web-server-provided patterns. Any HTTP response that contains any of these patterns is passed on to the Trusted Viewer and the rest of the messages are passed on to the legacy browser. We achieve this separation with minimal modifications to the legacy browser: we modify the https proxy settings of the browser. We also do not require any modifications to the HTTP protocol and therefore, we can interoperate with the large number of legacy clients and web servers. In the rest of this section, we describe the two components of the AppCore: the https proxy and the Trusted Viewer.

3.3.1. The *https* Proxy

The https proxy has three functions: multiplex and demultiplex requests and responses amongst the Trusted Viewer and the legacy browser, identify high-sensitivity information in http responses, and finally, accept configuration information from the TrustedViewer. Since we require our solution to work with legacy service providers, we cannot assume that incoming information will be tagged with sensitivity levels. Therefore, we have to infer sensitiveness of information based on HTTP requests or responses.

3.3.1.1 Inferring Sensitive Information in https

There are two sources of high-sensitivity information in https-based applications: the service provider and the user. Since the https proxy lies in between the service provider and the client-side application-level software, it can trap any high-sensitivity information originating from the service provider. However, the proxy cannot directly trap or control any high-sensitivity input originating from user. However, the proxy can be programmed to redirect all HTTP responses that request the user to enter high-sensitivity data to the TrustedViewer. To do so, we have to expand the notion of high-sensitivity responses to include even non-sensitive or low-sensitivity responses that lead to high-sensitivity user input.

It is easy for users and service providers to identify high-sensitivity information at the data item level, e.g., password, payment information or a TAN. However, the values for these sensitive data items varies considerably and we cannot expect the proxy to be able to parse every response to determine if a variable sequence of characters or numbers is high-sensitivity data or not. Ideally, we would like to look for static and unique content in HTTP responses to identify responses containing high-sensitivity data. We assume that the user or the service provider has identified *possibly* unique text patterns in HTTP responses that contain high-sensitivity data.

We employ a two stage filtering process on the client-side to accurately identify high-sensitivity responses: first, we use simple pattern matching in the https proxy to trap *all* HTTP responses that contain the text pattern. While this leads to false positives, i.e., a low-sensitivity response is treated as a high-sensitivity response, it is conservative because with proper keywords, we identify all high-sensitivity responses. Next, we employ more complex parsing based on expected HTML page structure in the TrustedViewer (Section 3.3.2) to identify the high-sensitivity responses among the trapped responses and redirect all false positives back to the untrusted software stack.

Note that in the worst case, if the user or the service provider does not provide any keywords, all responses will have to be trapped and parsed. If the parsing fails, the response is redirected to the legacy browser. While this will increase the response time for the user, it will prevent the flow of high-sensitivity information to legacy or untrusted computing bases.

As an example, while shopping on Amazon.com, user's choice on the confirmation page is high-sensitivity data. This high-sensitivity data is encoded in an HTTP request, which originates from the HTTP response that contains the confirm page. The confirm page contains the string `*alt="Place Your Order"*`. So we trap all responses that contain the above string. In the unlikely case that someone has a product with the same title, that particular product page will also be trapped. However, the confirm-page parser that resides in the TrustedViewer will fail to parse the product page and it will be redirected back to the legacy browser.

3.3.1.2 Demultiplexing HTTP Responses

Once high-sensitive HTTP responses have been identified, the https proxy must direct them to the TrustedViewer. To do so, the proxy must also transfer client-side state from the legacy client software to the TrustedViewer. This is easy in the case of https-based applications as client-side state is embedded in a cookie. Since the cookie is

transmitted with every outgoing message, the proxy has to capture the outgoing message and transfer it to the TrustedViewer along with the high-sensitivity response.

Switching from the TrustedViewer to the legacy client software is easier because the TrustedViewer does not modify client-side state. Since the old cookie is still valid, the https proxy does not have to take any additional steps to maintain client-state coherency between the two software stacks.

3.3.1.3 Accepting Configurations from the TrustedViewer

The third task of the https proxy is to accept configuration information from the TrustedViewer. To do so, the https proxy exports a control interface that accepts configuration information from the TrustedViewer. As seen in Section 3.3.1.1, text patterns are the only configuration information needed by the https proxy.

3.3.2. Trusted Viewer

The TrustedViewer consists of three parts: a configuration file, a DOM tree builder that can be used to extract parser and a GUI to interact with users. The configuration file lists all text patterns that are associated with high-sensitivity information and will trigger the activation of the TrustedViewer. The configuration file varies for each service provider. At startup, the TrustedViewer registers these patterns with the https proxy. Upon receiving a HTTP response which contains any of the keywords, the https proxy forwards the HTTP response and the corresponding HTTP request to the TrustedViewer. Remember that the HTTP request contains the cookie that is necessary for all subsequent interactions.

The second part of the TrustedViewer is a simple parser. The parser contains two sets of configuration information: (a) XPaths [32] in the response HTML document pointing to data that must be extracted and (b) sanity check tuples. A sanity check tuple is of the form <XPath, SanityPattern>. This indicates that the parser must check for the

occurrence of the SanityPattern in the given XPath. For example, the confirm page for electronic commerce transactions typically contains standard patterns such as “Shipping Information”, “Order Total” in predetermined XPaths.

The parser builds a DOM tree of the HTML document and extracts content from all specified XPaths. At the same time, the parser also checks if the SanityPatterns specified in the sanity check tuple match the extracted content. On success, the parser returns information in a tabular format to the TrustedViewer. If either the extraction fails or SanityPattern matching fails, the parser sends an error message to the GUI component. For more robust parsing, HTML content extraction tools as described in [79] can be employed. If the page cannot be parsed by the TrustedViewer, the user is given an opportunity to cancel the interaction or forward the page to the legacy software stack.

The third part of the TrustedViewer is a GUI component that displays the high-sensitivity information extracted by the parser. The GUI allows users to enter text input and confirm or cancel an interaction. The GUI also contains an option to forward the page to the browser, if the user feels that the page was not properly parsed or if the user feels that the tradeoff between functionality and security has changed.

Adding a new page to the TrustedViewer is easily accomplished. We have developed a script that operates on templates of HTML files that have to be handled by the TrustedViewer. The user clicks on content that she deems sensitive and the script provides the XPath of the content. At this point, the user can also select text patterns to generate sanity check tuples. The user then provides the XPath values along with the text patterns for the page to the TrustedViewer. The TrustedViewer forwards the text patterns to the https parser and uses the XPath and sanity check tuple to parse the new page.

3.4. Implementation

We implemented the two components of the AppCore on top of Nizza architecture. The resultant system, called BLAC, described in Figure 4. Once again, we

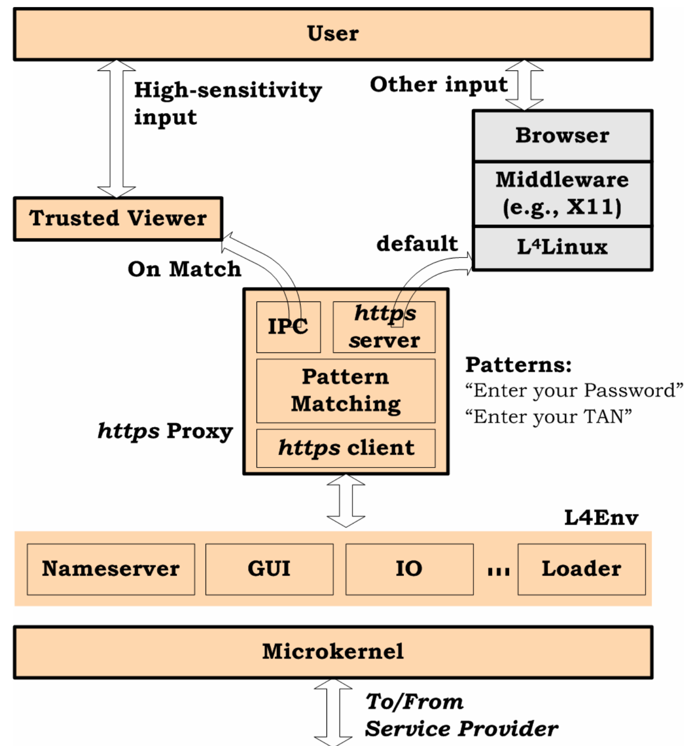

Figure 4. Architecture of the BLAC system. High sensitivity information, as identified by the text patterns, is directed to a small trusted viewer and the rest of the information is directed to a legacy software stack.

rely on the L4 microkernel and its execution environment, L4Env, to provide the necessary functionality. The https proxy and the Trusted Viewer are executed as separate processes directly on top of the microkernel. The legacy browser is executed as an untrusted process on top of a paravirtualized, legacy operating system (L^4Linux). We modified the browser settings, requiring it to communicate with our https proxy to access https URIs.

The https proxy accepts text patterns from the Trusted Viewer to determine the sensitiveness of responses from the web server. In the case of a match, the https proxy directs the response to the Trusted Viewer. All other responses are directed to the browser. In addition the https proxy is also capable of redirecting erroneously classified responses from the Trusted Viewer to the browser.

3.5. Evaluation

3.5.1. Security Properties of BLAC

BLAC improves the security properties of client-side software stack in three ways: First, BLAC switches software stacks depending on the sensitiveness of information being handled. In current systems, the same software stack is used to handle different categories of information flows. Therefore, an attacker can exploit vulnerabilities in handling of non-sensitive information to either access sensitive information (e.g., cross-site scripting attacks [42][92]) or to compromise the software stack (e.g., exploit arbitrary code execution vulnerabilities in browsers to install malicious extensions [34]). Since BLAC separates the handling of different categories of information flow, these attacks will no longer succeed in compromising the flow of sensitive information.

Secondly, BLAC treats the browser and its execution environment (L^4Linux and X server) as untrusted components. By preventing the flow of sensitive information to these components, BLAC ensures that vulnerabilities in these components cannot affect the confidentiality and integrity of sensitive information. However, the current BLAC architecture and implementation does not address availability issues. This leaves open the possibility of Denial of Service attacks, e.g., by crashing the network stack, the attacker can prevent the https proxy from communicating with the service provider. We are exploring the use of replication to improve availability properties of BLAC.

Thirdly, BLAC uses a small and simple application and software stack to handle high-sensitivity information. We compare the size and complexity of BLAC with a comparable Linux-based software stack. The Linux-based software stack is configured to provide the same functionality as provided by BLAC. Table 4 compares the software complexity metrics of the two approaches. At the application layer, we can see that the size and complexity of BLAC is two orders of magnitude smaller than that of the

Table 4: Complexity Comparison of BLAC and a Linux-based software stack. The Linux kernel includes networking, file system, IO, memory management support. L4Env includes servers for bootstrapping the system and resource managers for memory and devices, IO manager, window manager and ABI support.

Component	BLAC			Linux		
	Composition	LOC[a]	MCC	Composition	LOC[a]	MCC
OS	**L4**	14,000	2,300	**Linux Kernel**	383,000	65,000
Middleware	**L4Env**	86,300	11,300	**X Server**	1,015,000	140,300
https Proxy	**https Proxy**	13,600	1,900	**-**	-	-
Application	**Trusted Viewer**	5,000	290	**Mozilla Firefox**	978,000	151,000
Total		**118,900**	**15,790**		**2,376,000**	**345,300**

[a] We used SLOCCount [113] to measure lines of code.

browser. Comparing the complexity of the full system, we see that BLAC is an order of magnitude smaller than a comparable Linux-based system.

This comparison might seem unfair, especially given that the Mozilla Firefox browser possesses much more functionality than the TrustedViewer. However, one should note that BLAC employs the TrustedViewer to operate on high-sensitivity data. The browser, with its varied functionality, is still available to the user to operate on low-sensitivity or non-sensitive data.

3.5.2. Performance

Since the https proxy introduces a layer of indirection between the user and the service provider, we expect performance degradation. This section attempts to quantify the overheads introduced by the https proxy. We used two identical machines, each with a 2.26 GHz, Pentium-4 processor and 512 MB of RAM. The machines were connected using a 100 Mbps switch. The standard deviation was less than 5% in most experiments and less than 10% in all our experiments. We used https traces from four online sites in our experiments: Deutsche Bank, Dresdner Bank, Chase Bank and Amazon.com. Table 5 presents a profile of the four data sets. Pages represent the number of complete pages that were in the trace. Each page is the result of a user-initiated HTTPS request. Since a page

Table 5. Dataset for Measuring Page Access Times. Traces were generated using the WebScarab proxy [16] and the Internet Explorer web browser. Caching was enabled in the browser and the cache was cleaned before generating the trace.

Dataset	Pages	Fragments	Size (KB)	Max. Frags/Page
Amazon	6	52	569	40
Deutsche Bank	14	67	441	20
Dresdner Bank	18	107	931	17
Chase Bank	11	119	1433	59

consists of many fragments, each of which has to be requested separately by the client application, we also list the number of fragments in the trace and the maximum number of fragments per page. In all traces, the login request resulted in the maximum number of fragments requested.

In our experiments, the server side consists of a simple program that reads the traces, accepts HTTP requests over an SSL channel and transmits the appropriate HTTP response over the same SSL channel. We used three client-side configurations. One is a Linux client that connects to the server using SSL and generates HTTP requests. The client reads the complete response and discards it. The second configuration represents a legacy software stack configuration in BLAC. We use the same client as before;

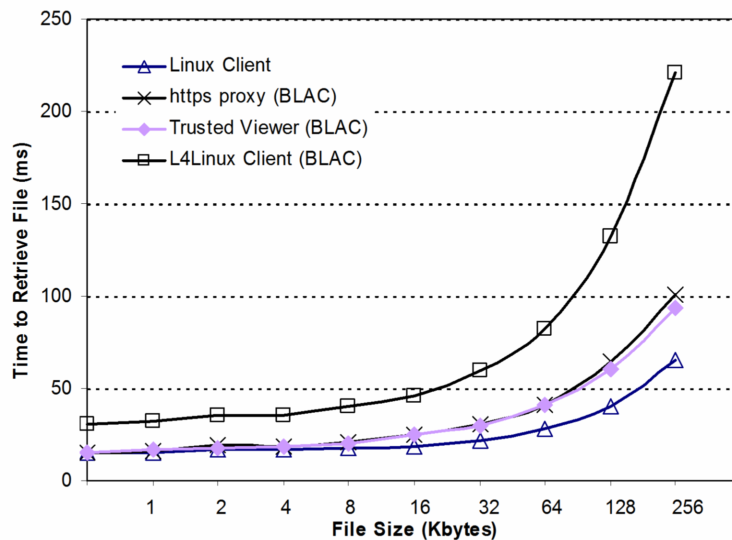

Figure 5. Comparison of File Access Times.

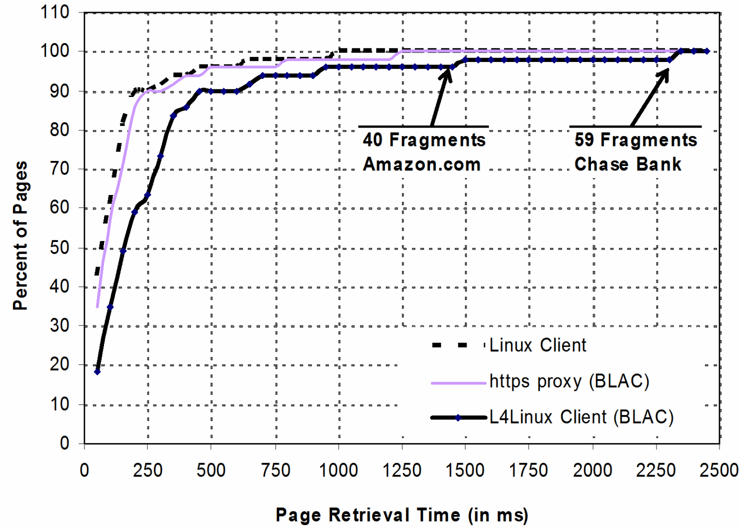

Figure 6. CDF of HTML Page Access Time based on HTML pages from our traces. As expected, pages with fewer fragments (not shown on graph) had lower page retrieval times.

however, it runs on L^4Linux and connects to the https proxy. For this configuration, we also measured the performance of the proxy, when it contacts the service provider. This allows us to identify the overheads introduced by the proxy. The third client configuration consists of a TrustedViewer using IPC to contact the https proxy and request files from the service provider.

First, we measure the retrieval times for individual files. Figure 5 compares the performance of the various client configurations. Accessing a page in the L^4Linux client through the https proxy is about 2 to 2.5 times slower than the original Linux implementation. Regression analysis of data showed that overhead is represented by the equation

ovhd = 0.535 * file_size + 17.9; and $r^2 = 0.998$.

where ovhd is the overhead in milliseconds and file_size is the size of the retrieved file in kilobytes. In absolute terms, the overhead per file is of the order of few tens of milliseconds.

Since an HTML page is typically composed of multiple fragments, we also measure "page" access times for the dataset mentioned in Table 1. Figure 6 plots the CDF

of page access times for the Linux and the L4Linux client. While the L4Linux client is slower than the Linux client, we see that the page retrieval time is less than 0.5 seconds in 90% of the cases. The pathological cases occur when a page has a large number of fragments, as illustrated in the figure. Even in the worst case, the page retrieval time is around 2.25 seconds. A survey by Jupiter Research and Akamai showed that about 75% of shoppers are willing to wait up to four seconds for a page to load [12]. So, even in the worst case, we are within the threshold for a majority of users.

We would like to note that the performance of our proxy can be improved in many ways. One source of improvement is to implement the HTTP/1.1 protocol in the https proxy that allows multiple requests to be sent over a single SSL connection. This amortizes the overhead of the CONNECT message and SSL connection establishment phase (14.5 ms in our current implementation). Techniques such as VMM-bypass-IO [82] can also be employed to reduce the overhead performing IO in virtual machine based systems.

3.5.3. Complications with Real-World Web Servers

Once again, we studied the four web sites mentioned in Table 2. High-sensitivity interactions accounted for two pages in all bank sites: login and confirm page. In Amazon.com, three pages were deemed highly sensitive: login page, confirm page and payment page as the user could enter a new payment method in the payment page. The payment page is a good example of a page where the user might want to transfer a page from the TrustedViewer to the legacy browser, in the case that she is not entering any high-sensitivity information. Compared to the total number of pages in the session, we can see that the TrustedViewer is sparingly used. We successfully tested our complete prototype using Deutsche Bank's demonstration account [8].

In our analysis of these sites, we came across some cases which cannot be handled by our current implementation. First, if the browser chooses to enable

compression, then we are not able to handle HTTP requests and responses. This problem can be solved by adding a compression library to the https proxy or by rejecting compressed requests from the browser.

The second problem is more critical as there are some web service provides who present the login page over an http URL. The login request is still transferred using the https URL. Since our proxy is an https proxy, we cannot currently capture such pages. Once again there are two solutions: one is to use the https URL instead of the http URL while retrieving the login page. This is not a problem as most sites allow http URL content to be served over https URLs. Alternatively, we could add an http proxy to the https proxy. While this will lead to increased overheads, we will be able to capture all requests for high-sensitivity data.

Another problem faced by the current version of BLAC is that SSL and HTTP do not provide support for remote attestation. This opens up the possibility of a malicious, untrusted browser generating an HTTP request that mimics a request generated by the TrustedViewer. For example, in electronic commerce sites such as Amazon.com, the user's choice on Confim/Cancel is high-sensitivity data and this data is encoded in the HTTP request. As there is no secret information involved in generating the request such as a TAN, the browser too can generate similar HTTP requests and modify the state of the user's account. We expect this problem to be alleviated with the use of remote attestation.

3.5.4. Revisiting Design Goals

In Section 2.2.1, we mentioned five design goals of an effective solution. Of the five goals, we evaluated security properties, complexity reductions and performance overheads in the previous subsections. This section evaluates the design and implementation of BLAC with respect to the other two requirements.

Reuse of Legacy Software. BLAC reuses unmodified (excluding proxy settings), legacy software to handle non-sensitive and low-sensitivity information flows. Hence users can employ legacy client application-level software for all non-sensitive and low-sensitivity interactions. We also saw that in online banking and e-commerce sites, the user has to interact sparingly with the Trusted Viewer.

Reuse of Interfaces. The HTTP protocol and data formats of each message (text, images, video, etc...) form the functional and data interface in https-based applications. The https proxy in BLAC does not require any modifications to either SSL or the HTTP protocol. Similarly, the https proxy infers sensitiveness from existing responses and does not require any modifications to the data format. Hence, BLAC interoperates smoothly with legacy clients and service providers.

3.6. Discussion

3.6.1. Applicability to other VMM-based approaches

While we implemented BLAC on a microkernel platform, it can also be implemented on other platforms, including commodity operating systems. In any platform, the first requirement is that the https proxy, the Trusted Viewer and the legacy browser need to be executed in different protection domains to isolate the flow of sensitive information. Second, the platform must support communication primitives so that the https proxy can exchange data with the Trusted Viewer and the legacy browser.

Environments such as Xen [35], Asbestos [54], Linux and Microsoft Windows satisfy both above requirements. A Xen-based implementation, for example, would have each of the above mentioned components executing in a separate VM. They can communicate with each other using network stacks. In a Linux-based implementation, the components can execute as different users and they communicate using the network stack or more efficient IPC such as shared memory.

3.6.2. Server-Side Support

One of the key requirements for AppCores (and BLAC) was support for legacy service providers. This complicates our design and implementation of BLAC, e.g., we have to indirectly infer the sensitiveness of information in the responses. With server-side support, we can further simplify the Trusted Components (https proxy and the Trusted Viewer) in BLAC.

First, the service provider could label information in its responses with sensitivity levels. In HTTP responses, this means adding an additional field to the response header: Sensitivity= [*Non* | *Low* | *High*]. This step would reduce the need for parsing of responses in the https proxy. However, to support user driven configurability, we would still have to rely on indirect methods for inferring sensitiveness of responses.

Secondly, if the service provider tags some responses as High-sensitivity, then the service provider could use a simple representation format (e.g., tabular format for high-sensitivity pages in BLAC) or provide template files for extracting relevant data from the responses. This would simplify the Trusted Viewer.

CHAPTER 4

A SERVER-SIDE APPCORE FOR WEB SERVICE PLATFORMS

Service-Oriented Computing (more recently also referred to as "service computing") is designed to support rapid creation of new, value-added applications and business processes that can span diverse organizations and computing platforms. Concretely, Paypal's Web Services, eBay Developer Program and Amazon Web Services are illustrative examples of web services being used in mission-critical, security-sensitive, and truly large scale applications. Despite the widespread deployment of web services, however, significant research challenges remain. This dissertation is concerned with the protection of security-sensitive information in service computing.

Web Service Platforms (WSPs) such as Apache Axis2, Microsoft .NET and IBM WebSphere provide essential functionality such as SOAP messaging and support for publishing and discovering web services. Additionally, WSPs provide desirable functionality such as support for web service composition, atomicity and message reliability. Support for such large and varied functionality has increased the size and complexity of current WSPs; for example, the open source Axis2 WSP and its extensions account for over 110,000 lines of code (LOC).

However, as we demonstrate later (Sections 4.1 and 4.2), the design and implementation of current WSPs requires users and service providers to trust large and complex components. The configurability and extensibility of many WSPs poses additional challenges to testing and analysis. This has resulted in WSPs with multiple security vulnerabilities [18][20]. We apply the AppCore approach to WSPs to split existing WSPs into two parts: a small, trusted T-WSP that handles security-sensitive information and a legacy, untrusted U-WSP that handles non-sensitive and protected (encrypted or signed) sensitive information (Section 4.3). We call the resulting

architecture ISO-WSP [107]. However, since WSPs interact with application-level code, we also have to modify application-level code. We introduce the notion of a Secure Function Interface (SFI) to split existing applications into trusted and untrusted parts, thereby allowing existing applications to be ported to ISO-WSP with minimal modifications (Section 4.5).

4.1. Design of Web Service Platforms

W3C's web services architecture specification [30] specifies the basic framework for WSPs. WSPs such as Apache Axis2 and Microsoft .NET implement this framework. The framework consists of three entities: a service provider, a client and a web service discovery agency. A WSP is used to mediate interactions between the three entities and this requires support for three basic classes of functionality: exchanging messages, describing web services and publishing and discovering web service descriptions. Since this dissertation addresses the security of information exchanges between the service provider and the client, we focus on the first class of functionality: support for message exchanges.

Figure 7 presents one view of the web services stack, as illustrated in W3C's specification [31]. All WSPs must implement a transport protocol, typically HTTP, and provide support for a message packaging mechanism, typically SOAP. The WSPs might also choose to support additional message packaging and transport mechanisms such as MIME over SMTP. In addition to the basic features, the WSP must also support functionality such as routing, transaction support, message reliability, security and quality of service. W3C has also standardized many types of additional functionality in the form of WS-* extensions such as WS-Addressing, WS-Security amongst others. Table 6 lists some of the WS-* extensions and briefly describes the functionality of each type of extension. WSPs may also possess an extension architecture to seamlessly integrate new

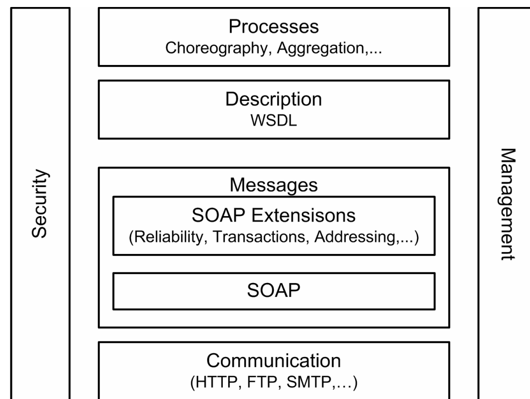

Figure 7. Web Services Stack. From W3C's Web Services Architecture specification [9].

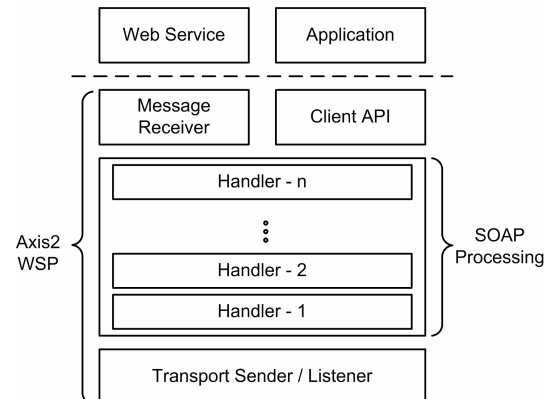

Figure 8. SOAP Processing Model in Apache Axis2.

and upcoming WS-* specifications or to provide support for custom extensions for logging or load-balancing.

The Apache Axis2 WSP [89] is one implementation of the web services framework. We use the Axis2 WSP in our analysis as not only is it widely used, it is also available under an open source license, enabling us to gain a clearer understanding of its workings. Axis2 provides support for developing, deploying, managing and invoking web services. Additionally, Axis2 also provides support for utilizing many WS-* extensions such as WS-Security and WS-ReliableMessaging. As we are primarily interested in the processing of web service requests and responses, we focus on the SOAP processing model of Axis2 [3]. Figure 8 illustrates the SOAP processing chain for a web service request or response in the Axis2 WSP. Axis2 handles all message interactions using two basic message handling sequences: *In Pipe* and *Out Pipe* for incoming and outgoing messages respectively. As the basic structure of both sequences is similar, we use the *In Pipe* as an illustrative example.

An incoming web service request is first received by the Transport Listener. The Transport Listener creates a message context and starts the In Pipe, which consists of a series of handlers. First, the transport handlers are used to verify transport protocol headers and populate the message context with data from the message. Next, handlers are

Table 6. List of some WS-* extensions and the functionality provided by each of them [13]

Extensions	Functionality
WS-Addressing	Provides a uniform naming scheme to address end-points.
WS-Security, WS-Trust, WS-SecureConversation, ...	Securing messages and establishing trust across different domains.
WS-AtomicTransaction, WS-Coordination, ...	Transaction Support for web services.
WS-ReliableMessaging	Support for reliable delivery of messages over non-reliable communication channels.
WSDL, WS-Policy, ...	Support for description and exchange of metadata between clients and service providers.
WS-Eventing	Event-driven programming support for web services

used to process addressing headers and determine the target service. After this, user specified handlers are executed to perform tasks such as security processing, transaction support or reliable messaging support. Finally, the message is deserialized into language level objects and given to the web service implementation, which executes the business logic.

Axis2 uses handlers to implement and support various types of optional functionalities. For example, handlers are used to support many of the WS-* extensions like WS-Security and WS-Addressing. Additionally, Axis2 allows for custom handlers to perform web service-specific, non-standard tasks such as admission control, load balancing or logging.

4.2. Security Problems in Web Service Platforms

The WS-Security specification [15] was designed to protect the confidentiality and integrity of information flow in web services. However, WS-Security-based protection can be bypassed by exploiting vulnerabilities in endpoint software. On the server side, attackers can compromise information flow by exploiting vulnerabilities in server software: operating system, web server, WSPs [18][20], or the business logic and

```
// All Handlers can retrieve the message, service context
param = ctx0.getAxisConfiguration().getParameter("OutflowSecurity");
if (param !=null){
  ome = param.getParameterElement();
  itor = ome.getFirstElement().getChildElements();
  while (itor.hasNext()){
    attr = (OMElement) itor.next();
    // Look for Encryption Key and Set it to a weak key?
    if ("encryptionUser".equals(attr.getLocalName())){
      attr.setText("weak_key_identifier");
    }
}
```

Figure 9. Indirect Access in the Axis2 WSP. All handlers have access to service context, which contains, amongst others, parameters for the encryption key. A malicious handler can replace the encryption key as shown in the highlighted line.

its support software (e.g., database software). Similarly, on the client side, attackers can leverage vulnerabilities in client-side WSPs or client applications like the browser. In this section, we focus on securing WSPs. Securing client applications was discussed previously (Chapter 3).

From a security perspective, WSP implementations have two significant issues. First, at the component level, WSPs violate the principle of least privilege (PoLP). PoLP states that components should execute with the least set of privileges necessary to finish the job. WSPs contain many components that do not need access to sensitive data, e.g., WS-* extensions such as WS-Addressing. However, all these components execute in the same address space and same protection domain. Hence they can either directly access security-sensitive data or modify WS-Security processing by modifying security processing parameters and compromise security-sensitive data.

Concretely, in the Axis2 WSP, all handlers execute in the same protection domain as the rest of the WSP and the application-level code. Therefore, handlers that do not require access to sensitive data get access directly (reading message contents) or indirectly. Figure 9 provides an example of indirect access in the Axis2 WSP, where a misbehaving handler, even one that operates on encrypted data can compromise information flow by changing WS-Security processing parameters.

Secondly, a WSP is a complex piece of software. As seen earlier (Section 4.1), WSPs are expected to perform a large number of tasks. Unsurprisingly, they contain large code bases. Additionally, since WSPs have to be configurable and extensible, they also typically possess configuration files, an extension-architecture and multiple extensions. Concretely, the Axis2 WSP alone contains about 23.5 KLOC. Together with the implementations of the multiple WS-* specifications, the WSP contains over 110 KLOC. Additionally, programmers can write custom handlers to carry out other types of processing like load balancing or admission control. These components add to the complexity of the system. Given the large code base and the multitude of ways in which these components can interact with each other, it is not practical to test the components of the WSP, especially as a complete system, and eliminate all vulnerabilities. Moreover, the configurable nature of WSPs means that extensions can be added, enabled and disabled at runtime, further complicating the analysis and testing of WSPs.

4.3. Components of the AppCore

First, we identify and extract components from the legacy WSP that require access to sensitive information (i.e., trusted components). Second, we have to design a new trusted component, called the Message Splicer to control the flow of sensitive information. These components are composed together to form the AppCore, called the T-WSP (Trusted WSP). The rest of the legacy WSP does not have to be trusted and is called the U-WSP (Untrusted WSP).

4.3.1. Components taken from Legacy WSP

In our analysis of legacy WSP components, we assume that all security-sensitive information is protected using WS-Security [15]. While the SSL and TLS protocols can be used to protect information, they are not always suited for web services because of the

unique needs of web services, e.g., support for fine-grained encryption or signatures such as signing SOAP headers. Hence, we do not consider them in our analysis.

Identifying the components of the T-WSP requires understanding of the various components of the WSP and their corresponding functions. We relied on W3C's architecture specification [30], specifications for the WS-* extensions such as WS-ReliableMessaging, WS-Addressing [13], and the Axis2 web services middleware architecture guide [3]. Based on our analysis, we found that the security-related extensions such as WS-Security, WS-Trust and WS-SecureConversation are the components of the WSP that have to be trusted. These components have to be trusted for one of two reasons:

- **They require access to sensitive contents of a message (Direct Access)**. WS-Security implementations fall under this category as they need access to sensitive contents either to protect them (encrypt or sign) or to verify the protection on them (decrypt or verify signatures).

- **Or, they control behavior of components with direct access (Indirect Access).** This includes other security specifications such as WS-Trust and WS-SecureConversation. The implementation of these specifications have to be trusted, even though they do not require access to sensitive data, as these components control WS-Security processing. A malicious implementation can subvert WS-Security processing, e.g., employ a weak encryption key, to gain access to or leak sensitive information.

The rest of the components of the WSP, including the WS-* extensions, are agnostic to message contents and can be treated as untrusted components. For example, the transport layer protocol implementation or SOAP implementation is agnostic to message contents as long as the contents confirm to the SOAP message format. However, there is a special class of untrusted components that requires access to sensitive data such as signature or encryption keys, albeit indirectly. WS-* extensions such as WS-Addressing recommend the use of WS-Security to protect the integrity and in some cases,

the confidentiality of SOAP message headers. These extensions require the use of WS-Security and associated signature or encryption keys. Even though they are indirectly dependent on sensitive data, they do not require access to sensitive data, especially the contents of the message body. While these extensions are crucial to maintain the reliability and availability properties of web services, they do not affect the confidentiality and integrity of end-user data. Therefore, we do not treat them as Trusted Components.

4.3.2. Message Splicer

Isolating security-relevant processing in the T-WSP does not prevent the flow of sensitive information to the U-WSP. We need a completely new component, analogous to the https proxy in the AppCore for https-based applications, to control the flow of sensitive information. We call this component the *Message Splicer*[1]. The function of the Message Splicer is to replace sensitive data with dummy, non-sensitive data or vice versa. For an incoming message, the Message Splicer replaces sensitive data items with dummy data items and a token that uniquely identifies an instance of a sensitive data item and transfers the message to the U-WSP. The token acts as a capability for the sensitive data item. The Message Splicer passes the sensitive information directly to the trusted part of the application. For an outgoing message, the Message Splicer accepts sensitive data from the trusted part of the application and non-sensitive data from the U-WSP, composes them in a single message, and passes it on for security processing. Sensitive data items are protected (encrypted or signed) before being transferred to the untrusted WSP for further processing and transmission. Section 4.7.1.1 discusses the utility of having a Message Splicer.

4.4. Architecture of ISO-WSP

[1] Message Splicer is derived from *Gene Splicing*. Gene splicing is the process of creating recombinant DNA by cutting and joining DNA sequences from multiple DNA fragments.

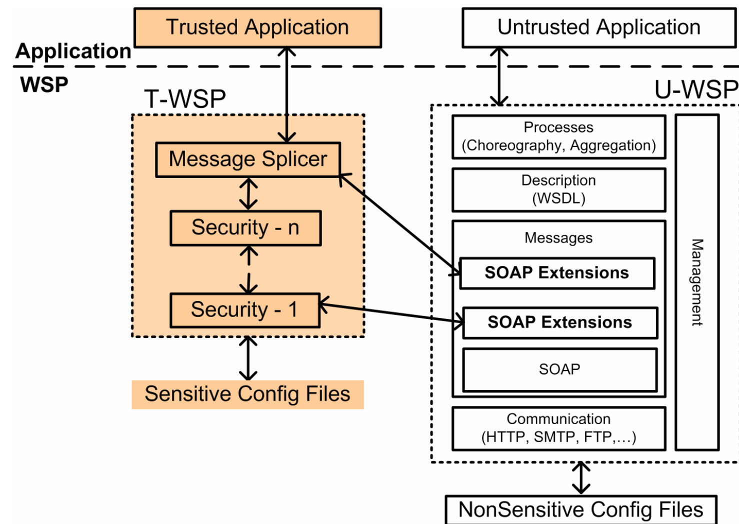

Figure 10. Architecture of ISO-WSP. Shaded boxes represent Trusted Components. Trusted Components execute in a separate protection domain.

We call the combination of the T-WSP and the U-WSP as ISO-WSP. Figure 10 provides an overview of the architecture of ISO-WSP. As explained in the previous subsection, implementation of security specifications form the T-WSP, and the rest of the components form the U-WSP. In each case, the components are grouped together and executed as independent applications. In addition to the separation of components of the WSP, the configuration files and the application-level code too have to be classified and separated into two categories. Since the behavior of the security specifications can be controlled by configuration files, the corresponding configuration files also have to be trusted and secured from access by untrusted components.

We enforce separation between the T-WSP and the U-WSP by executing them in separate protection domains, with the U-WSP running with lower privileges. This prevents U-WSP from modifying the binaries or configuration files of the T-WSP. This separation also prevents U-WSP from accessing the secret keys used for encryption or decryption. The Message Splicer, discussed in Section 4.3.2, prevents the flow of sensitive information to the U-WSP or to the untrusted parts of the application. Section 4.5 discusses our approach of protecting the flow of sensitive information in the application-level code in detail.

A legacy WSP can be converted to an ISO-WSP with a small number of modifications. After constructing a T-WSP, we have to modify the legacy WSP to invoke the T-WSP via remote invocation mechanisms instead of using local calls. This involves identifying parameters that are exchanged between the T-WSP and U-WSP, message and results of security processing, and adding the necessary serializing and deserializing code.

One of the main features of the ISO-WSP is that ISO-WSP exports the same external interface as legacy WSPs, thereby allowing ISO-WSPs to work with legacy clients. The U-WSP provides support for the interface by implementing transport-layer protocols such as HTTP over TCP/IP.

Another key feature of ISO-WSP is that the U-WSP can still execute WS-Security libraries in its protection domain. However, to reiterate, the U-WSP cannot access the secret keys of the T-WSP. Therefore, any data encrypted with these keys is inaccessible to the U-WSP. The U-WSP can use this feature along with weaker keys or keys with weaker security guarantees to provide limited security guarantees. For example, one can envision a service provider maintaining two sets of secret keys: one to protect the confidentiality and integrity of sensitive data items from any attacks on the U-WSP and another to protect all types of data from snooping attacks on the network. Depending on the security requirements of the data items or client preferences, the service provider can choose the set of keys to employ.

4.5. Application Level Support for the AppCore

From the application developer’s point of view, ISO-WSP represents a departure from existing WSPs. The developer has to consciously split the application program into a trusted and an untrusted part. This is a potential drawback for the ISO-WSP architecture as the developer will have to expend additional effort while designing a new application. Additionally, legacy code can no longer be executed on ISO-WSP without appropriate modifications. In this section, we will show that the application developer only needs to

carry out some simple steps to design a new application or port a legacy application to ISO-WSP. We will also see that these steps fit in naturally with security-aware design of web service applications.

Modifications to the application level software can be classified into two types: modifications to data structures and modification to code. We capture these modifications by asking the application developer to define an interface, called the Secure Functional Interface, to security-sensitive objects that can be used by untrusted components. We first describe this interface and then show how modifications to data structures and code can be easily effected with the help of this interface. We use Java as the application-level programming language to illustrate the modifications to data and code. However, our approach is equally applicable to other object-oriented languages.

4.5.1. Secure Functional Interface: A Restricted Interface for Remote Access

To understand the need for defining a restricted interface for remote access of sensitive objects, consider a payment processing service as described in Figure 11-A. Of the data items, it is reasonable to expect that the customer would like to enforce strong confidentiality properties on the credit card information (Figure 11-B). In particular, the customer does not want the credit card information to be inappropriately stored or transferred. Object-oriented languages such as C++ and Java encapsulate data items, providing limited protection against inappropriate access. Even in such languages, sensitive objects contain functions that allow for retrieval of sensitive information, e.g., *getter* functions that enable objects to be easily plugged into existing web service platforms (*getCcNum* in Figure 11-B). These methods can be invoked by untrusted objects to gain access to sensitive information. Therefore, we cannot export the complete list of methods supported by the sensitive object to the untrusted object.

We require the application developer to identify sensitive data items and define a restricted interface, called a Secure Functional Interface (SFI), for access by untrusted

components. The SFI for a sensitive data item lists all operations on the data item that can be performed by untrusted components. The SFI approach assumes that remote access function implementations are well-designed, i.e., a remote access function does not return sensitive data to the caller or compromise the integrity of sensitive data. Figure 11-C presents the SFI for the CreditCard object, which no longer contains the *getter* functions, thereby denying access to sensitive information to the untrusted part of the application.

Defining SFI is not an undue burden as current language-level features such as the *protected* and *private* keywords in function signatures already make the developers aware of security implications of the function. Defining a SFI is similar to the use of those keywords; however, address-space separation between the trusted data items and untrusted components provides stronger security guarantees about information exchanges between them.

4.5.2. Modifications to Data Structures

Based on our definition of SFI, one would assume that application-level data structures would be unmodified in ISO-WSP. However, as described in Section 4.4, we replace sensitive data items with dummy items that contain tokens. Therefore we have to modify the corresponding application-level data structures to include a token field. On the untrusted side, we also have to add another field that holds the stub for invoking the remote procedure on the trusted side. Adding these fields can be automated: add the fields to a class inherited from the class mentioned in the SFI file (Figure 11-E). We refer to this class as an *annotated class*.

A) Web Service Port

```
PPResults ProcessPayment (OrderInfo
ord, CustomerInfo cinf, CreditCard
cc);
```

B) Class Definitions

```
public class CreditCard{
  private String ccNum, Name, zip;
  private int expiryMon, expiryYr;

  /* Getters,Setters */
  public String getCcNum(){...}
  public void setCcNum(String num){.}
  /* Other getters, setters*/...

  /*Charge Card and return a Txn ID*/
  public String chargeCard(float
                amount) {...}
  /* Validate Card*/
  public boolean validate(){...}
  /* Additional Functions */ ...
}
```

Modifications Specified by Developer

C) SFI Definition

```
/* Classname and Namespace*/
class:=edu.gatech.cc.pp.CreditCard
ns:=http://pp.com/CreditCard
/* Interface */
interface CCsfi{
  String chargeCard(float amount);
  boolean validate();
}
```

D) Generated Interface

```
public interface CCsfi{
  public String chargeCard(String
              sfiID, float amount);
  public boolean validate(String
                          sfiID);
  public void cleanup(String sfiID);
}
```

E) Generated Code on Untrusted Side

```
public class CreditCardUnt
          extends CreditCard{
  private String sfiID;
  private CCsfi stub = null;

  /* Getters and Setters for sfiID*/
  public String getSfiID(){
                 return sfiID; }
  public void setSfiID(String id)
                           { ... }
  /* Initialize Stub */
  private void initStub(){ ... }

  /* override the methods defined
     in interface */
  public String chargeCard(float
                           amnt){
    if(sfiID != null){
      initStub();
      return stub.chargeCard(sfiID,
                             amnt);
    }else{
      /* Non Sensitive Object
         Call superclass method*/
      return super.chargeCard(amnt);
    }
  }
}
```

F) Generated Code on Trusted Side

```
public class RemoteCCSfiServer
                 implments CCsfi{
  /*Store ref to CC*/
  static Hashtable ht;
  /*Add Entry to hashtable*/
  public static void add(String id,
             CreditCardTrust cc){
    ht.add(id, cc);
  }
  /* Init Remote Interface */
  public static void initRI() {...}

  /* Handle remote Calls */
  public String chargeCard(String
          sfiID, float amount){
     CreditCardTrust cc =
    (CreditCardTrust)ht.get(sfiID);
     if(cc != null)
       return cc.chargeCard(amount);
     return "INVALID";
  }
}
public class CreditCardTrust extends
                        CreditCard{
  private String sfiID;

  public void setSfiID(String id){
    RemoteCCSfiServer.add(id, this);
    sfiID = id;
  }
  /* Other functions */...
}
```

Figure 11. Partial Code Listing for a Payment Processing Service. Code generation process uses XSLT templates. Boxed area indicates modifications specified by developer. Highlighted areas indicate code reuse.

4.5.3. Modifications to Code

First, by modifying the local application-level data structures, we are invalidating the schema for the web service interface (WSDL specification). Therefore, we have to modify the local application-level code to use the serializer/deserializer for the annotated class instead of the one for the legacy class. Crucially, the remote application-level code (a client in the case the service provider uses an ISO-WSP or vice versa) does not have to be modified as the Message Splicer in the local ISO-WSP adds these new fields to an incoming XML message and strips the fields from an outgoing message. Once again, we can use the list of sensitive classes from SFI specifications to configure the Message Splicer to strip or add token information to messages.

Secondly, when an untrusted component accesses functions specified in the SFI, it has to be converted into a remote call. This requires two modifications on the untrusted side: (a) parameters to the remote call have to be serializable. Most Java classes can be serialized by adding "*implements serializable*" to the class definition. The default serialization methods along with automatically generated getters and setters are sufficient for serializing and deserializing most classes. Otherwise, the developer will have to write custom serializers and deserializers. And (b), the untrusted classes have to be modified to call the remote interface in case the sensitive data resides in the trusted part of the application. This is easily accomplished by modifying the relevant method to pass the arguments along with the token to the trusted part of the application. On the other hand, if the data resides in the untrusted part of the application, then the local method is invoked.

At this point one should note that the untrusted part can modify itself to make a local call instead of a remote call. However, this would not affect the confidentiality or integrity of sensitive data, as the sensitive data is stored in the trusted part of the application.

On the trusted side, we have to write a remote interface server that handles requests from the untrusted part. The server maintains a collection of security-sensitive

objects, indexed by their unique identifiers. Upon receiving a request, the server uses the unique identifier to retrieve the appropriate instance of the sensitive object. It then makes a local call with the arguments provided by untrusted and returns the result.

One potential weakness in the system is that the trusted part now performs operations using data provided by the untrusted part. This may violate the integrity flow in the system, as data is flowing from a lower integrity level (untrusted part) to a higher integrity level (trusted part). We assume that the trusted code validates all inputs from the untrusted part before proceeding with the operation, e.g., by comparing against a local copy.

4.6. Implementation

We implemented an ISO-WSP prototype based on the Apache Axis2 platform. The T-WSP consisted of a WS-Security implementation (WSS4J [2]) and a Message Splicer. Accordingly, the configuration files for WS-Security also formed a part of the T-WSP. The rest of the Axis2 WSP, along with the other WS-* extensions formed the U-WSP. We modified Axis2 to make remote calls to perform WS-Security related processing. Since we used Java, all remote calls were Java RMI calls. This required around 100 new and modified lines of code (LOC). We also had to modify the WS-Security implementation to serialize and deserialize the parameters. Since the WS-Security configuration files were now a part of the T-WSP, we had to add code to read the configuration files. In all, we had to modify or add around 700 LOC. Thus, by adding or modifying around 800 LOC, we were able to convert the existing Axis2 WSP into an ISO-WSP prototype. We execute the T-WSP as a superuser process and the U-WSP as an unprivileged user. The file system permissions of the configuration files of the T-WSP are set such that the U-WSP is unable to read or modify them.

Our implementation of the Message Splicer accepts information about sensitive classes in two ways. First, it allows developers to specify XML files that contain

serialized versions of an instance of a sensitive data item with dummy values, e.g., a credit card data item with an invalid card number. The Message Splicer uses these "dummy" values when replacing sensitive objects in incoming messages. It also inserts unique tokens when replacing sensitive objects. Secondly, the Message Splicer accepts sensitive data items in the form of Document Object Model (DOM) fragments from the trusted part of the application. Each fragment possesses unique tokens that are used by the Message Splicer when replacing dummy data items with actual content in outgoing messages.

For performance analysis, we implemented a simple payment processing service (partial code listing for the service is presented in Figure 11) that accepts order information and payment information in the form of a credit card object and returns a confirmation string containing a transaction identifier and the amount charged to the card. We denoted the credit card information as security-sensitive information. To protect this information, the client specifies that the whole request message be encrypted using WS-Security with the public key of the service provider as the encryption key. In our legacy service implementation, the business logic accepts a DOM fragment containing the input parameters and deserializes it into Java objects. Next, it calls the charge card function to generate a confirmation number. The results are converted into a DOM fragment which is then passed on to the WSP for further processing and transmission.

In the implementation on top of ISO-WSP, the developer first specifies an SFI as illustrated in Figure 11-C. We then use our code generator to generate the necessary class files (D, E and F listings in Figure 11). We also require the developer to modify legacy code and we discuss the number and type of modifications in Section 4.7.3.

For an incoming request, both the trusted part and the untrusted part of the application receive DOM fragments, which are deserialized into Java classes. The trusted part registers itself with a Credit Card server that is executing on the T-WSP. The untrusted application calls the charge card function using the SFI. Hence, the call is

redirected to the Credit Card server on the trusted side (Figure 11-F), which in turn invokes the charge card function of the appropriate instance of the credit card class. Once the untrusted part gets the confirmation number, it invokes the cleanup method to free the credit card object in the trusted part of the application. Finally, the results are converted into a DOM fragment and passed on to the U-WSP.

4.7. Evaluation

We evaluate ISO-WSP along the design goals mentioned in Section 2.2.1. We first discuss the security properties of ISO-WSP and show that the flow of sensitive information is limited to trusted parts of the WSP and application. We also show that applying the AppCore approach to WSPs reduces software complexity by a factor of 5. Next we discuss the performance implications of ISO-WSP and show that it results in a manageable overhead of few milliseconds per request. We then evaluate the cost of porting legacy applications to the ISO-WSP.

4.7.1. Security Properties of ISO-WSP

4.7.1.1 Information Flow Security in ISO-WSP

Before we discuss information flow security in ISO-WSP, we list the basic assumptions and features of ISO-WSP. First, we assume that all sensitive information is protected using WS-Security. Furthermore, we assume that sensitive information protected using WS-Security cannot be compromised without first compromising the security extensions. Secondly, WS-Security processing is carried out in a separate protection domain, with the U-WSP being executed as a lower privilege process. Therefore, untrusted components cannot indirectly access sensitive data, e.g., they cannot change the parameters of security processing. We only have to prevent direct access of sensitive information by the U-WSP.

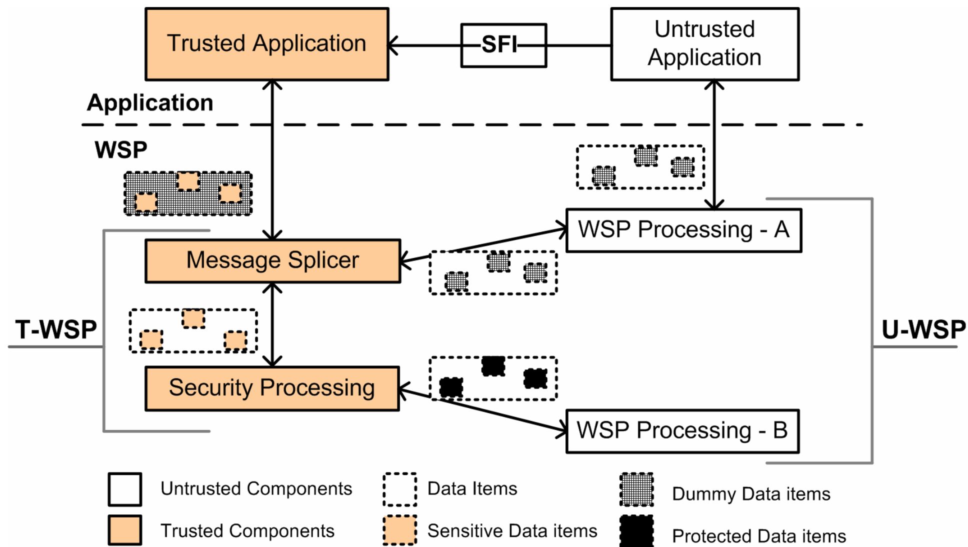

Figure 12. Securing Information Flow in ISO-WSP. The T-WSP replaces security-sensitive data items with protected data items or dummy data items before passing them on to the U-WSP. Secure Functional Interface (SFI) is discussed in Section 4.5.1.

Figure 12 presents the flow of information in ISO-WSPs. The Message Splicer is the key component that filters the flow of sensitive information by splitting and merging flows as needed. To analyze how this process of splitting and merging secures flow of sensitive information in contrast to legacy WSPs, we divide the untrusted WSP into two categories:

- Components that operate below WS-Security (*WSP Processing–B* in Figure 12): Since these components worked on encrypted or signed data, they did not have *direct* access to sensitive data in legacy WSPs. However, they could indirectly compromise flows by modifying security processing parameters, e.g., overwrite configuration files or security processing parameters. In ISO-WSP, they execute with lower privilege and in a separate protection domain and hence, cannot manipulate security processing parameters.
- Components that operate above WS-Security (*WSP Processing–A* in Figure 12): These components had both *direct* and *indirect* access to sensitive data in legacy WSPs. In ISO-WSP, we replace sensitive data with dummy data items before transferring the message back to these components. Hence, these components are deprived of direct

access to sensitive data. Previously discussed arguments against indirect access by untrusted components are equally applicable to these components.

4.7.1.2 Software Complexity Reductions

Our second motivating factor for constructing an ISO-WSP was the increased complexity of WSPs. Table 7 compares the software complexity of various WSP components and compares them against the T-WSP. We measured two properties: Source Lines of Code and McCabe's Cyclomatic complexity [85]. Empirical studies have shown that both measures of software complexity correlate with number of bugs in code [88]. We see that the T-WSP is a factor of 5 smaller and simpler than the current implementation of the Axis2 WSP, making the T-WSP more amenable to exhaustive testing. The small size of the T-WSP also makes it easier to apply static analysis techniques for monitoring the flow of information, as described in [96].

Extensibility of WSPs is a crucial factor in testing and analysis. Since extensions can change the behavior of WSPs and since they can be added or configured at run time, they complicate the testing process. However, extensions provide useful functionality such as support for addressing, transactions and reliability. By extracting a T-WSP and retaining the functionality of the legacy U-WSP, ISO-WSP attempts to gain the best of both worlds. By limiting extensibility and configurability in the T-WSP, ISO-WSP ensures that the testing process is simplified for T-WSP. And by retaining functionality in the U-WSP, ISO-WSP ensures that developers will still be able to use and configure extensions according to their needs.

Table 7. Comparison of Source Lines of Code (SLOC) and McCabe's Cyclomatic Complexity (MCC) of the T-WSP and the Axis2 WSP along with its extensions. All numbers were generated using the JavaNCSS tool [11].

Module	Axis2	Extensions	WS-Security	WSP-Total	T-WSP
SLOC	23,580	70,350	16,900	110,830	19,360
MCC	7,930	24,100	5,180	39,210	6,050

Limitations. We would like to note that the ISO-WSP is vulnerable to Denial of Service attacks, wherein the U-WSP either corrupts messages or does not invoke the T-WSP for security processing. While this affects the availability of the service, it does not compromise the confidentiality or integrity of security-sensitive information.

ISO-WSP can be enhanced with Trusted Computing hardware [25][24] and application level support for Trusted Computing, e.g, Integrity Measurement Architecture [97], to provide additional, desirable security properties such as integrity of software stack at load time and remote attestation.

4.7.2. Performance of ISO-WSP

Our experimental setup consisted of two machines, each with Pentium-4 3.0 GHz processors with 1 GB RAM, running Linux kernel 2.6.15. The machines were connected via a 100 MBps switch. We used Axis2 version 1.1 and WSS4J version 1.5.1 running on top of Apache Tomcat 4.1.31.

There are three sources of overhead in the ISO-WSP. First, since the security-sensitive parts of the WSP and application are separated from the rest of the application, there is the added cost of remote calls. Related to this is the cost of serializing and deserializing data items for remote calls. Finally, there is the cost of performing message splicing operations.

To estimate the remote call overhead (Java RMI in our case), we use a simple echo application with the client and server running on the same machine. Figure 13 presents the results of our experiments. We see that round trip times increased linearly with message sizes, at the rate of about 0.5 ms per kilobyte. For message sizes less than 8 KB, round trip time was less than 1 millisecond which is negligible when compared to typical web service response times which are of the order of hundreds of milliseconds [77].

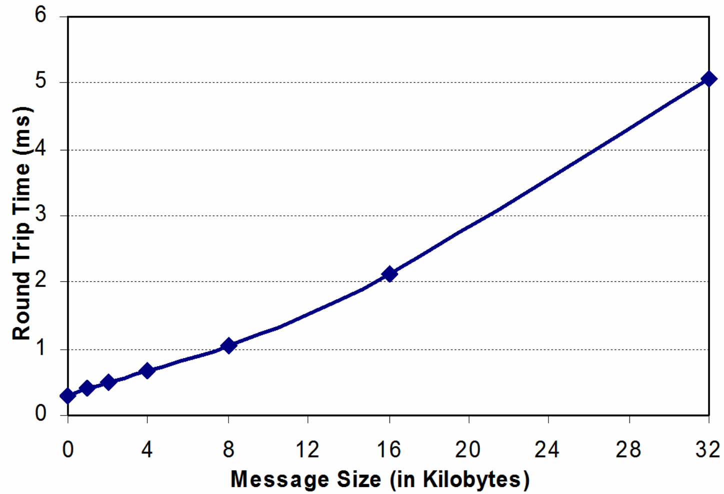

Figure 13. Remote Call Overhead. Java RMI Round Trip Time vs. Message Size

In the ISO-WSP architecture, we have to transfer SOAP messages between the U-WSP and the T-WSP. This involves converting SOAP messages in DOM format to byte array format and vice versa. To estimate these costs, we used an XML data set from the *XMLBench Document Model Benchmark* [33]. We evaluated the cost of serializing and deserializing the DOM representation of the data set for two XML parsers, the AXIOM pull parser used in the Axis2 WSP (and the U-WSP) and the default Xerces parser configuration used in the T-WSP. At this point, we want to reiterate that we only want an estimate of the costs of serializing and deserializing SOAP messages. We are not attempting to rigorously analyze the performance of the XML parser. Figure 14 compares the performance of the various parsers on the data ser. We found that combined serializing and deserializing costs for small to medium sized XML files (~10 kilobytes) to be less than 4 ms. However, the combined costs for the largest XML file in the dataset (36 KB) was about 14 ms (omitted from the figure for clarity).

We do not perform similar microbenchmarks for estimating message splicing costs and application-level serializing and deserializing costs because they are closely dependent on the application-level data structures. Rather, we measure these costs as a part of a concrete web service implementation.

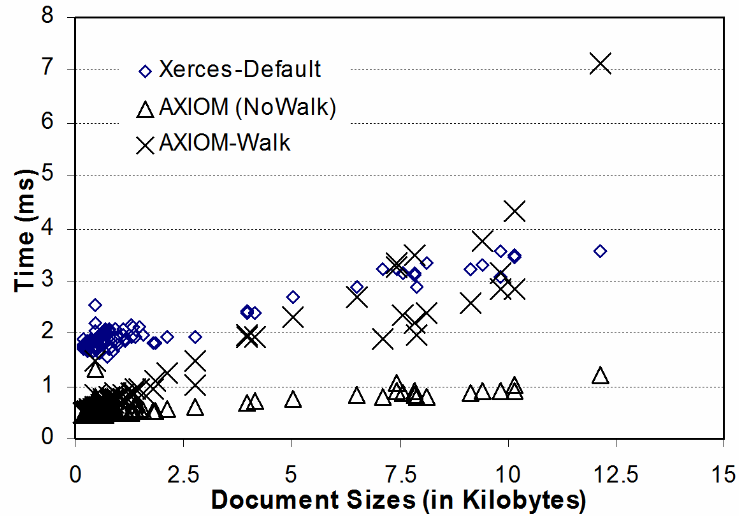

Figure 14. Performance Comparison of Parsers

We use the payment processing web service described earlier (Section 4.5) to evaluate the end-to-end overhead imposed by the ISO-WSP architecture. We found that using ISO-WSP increased the average response time from 40.3 ms to 47.9 ms. To accurately characterize the overheads imposed by ISO-WSP, we also measured the time spent in each stage of the ISO-WSP. To do so, we instrumented the U-WSP and T-WSP to record the time (at microsecond granularity) at each stage of processing. From the results (Figure 15), we see that the overhead of the additional stages is about 8.8 ms. However, the ISO-WSP implementation as a whole is only 5.2 ms slower than the Axis2 WSP. We found that this is because the Axis2 WSP uses the pull-based AXIOM parser whereas the T-WSP in our implementation uses the default Xerces parser configuration. Since WS-Security processing typically involves parsing the whole message (for full message signatures or encryption), the performance of WS-Security with pull-based parsers suffers. However, as expected, the AXIOM parser is faster when serializing and deserializing SOAP messages.

The rest of the costs (~ 2.4 ms) are distributed in the application-level code. These include the cost for two RMI calls from the untrusted part to the trusted part (~0.8 ms): one for charging the credit card and another for cleaning up the state in the trusted part.

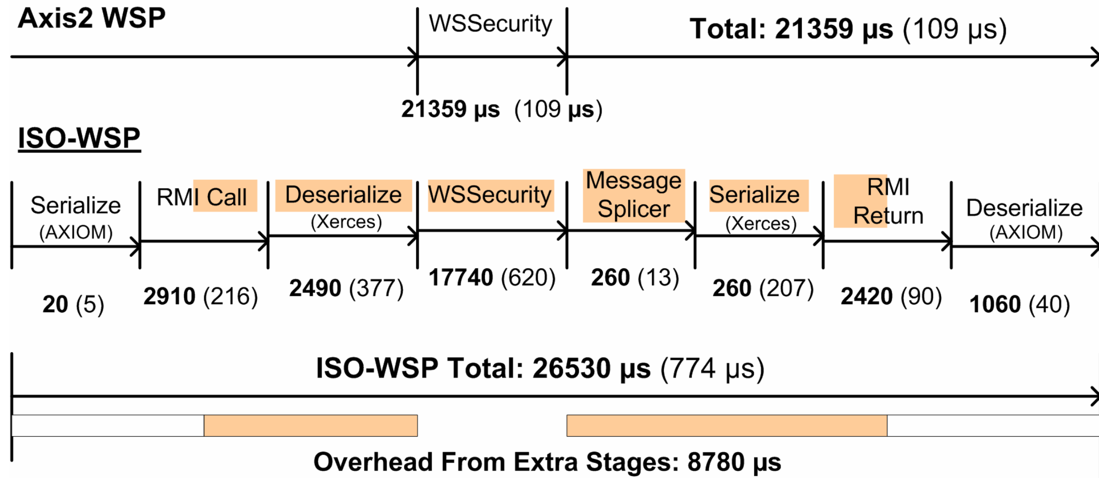

Figure 15. Comparison of security processing costs for an incoming message. All numbers in microseconds. Numbers in parentheses indicate the 95% confidence interval.

The second major cost arises because the SOAP message now has to be additionally deserialized on the trusted side (~1.5 ms).

Based on the microbenchmarks and the results for the payment processing web service, we can see that ISO-WSP introduces overheads of the order of a few milliseconds per request. While this might seem excessive, typical web service invocation time ranges from 0.5 seconds to a few seconds [77]. More importantly, one should note that ISO-WSP is invoked only during the exchange of sensitive information. When exchanging non-sensitive information, ISO-WSP still uses the legacy WSP, thereby maintaining the performance of the legacy WSP.

Moreover, as mentioned in Section 4.3, ISO-WSP is designed such that WS-Security processing can still be performed in the U-WSP. Ideally, the service provider would provide stronger security guarantees when the T-WSP is involved (e.g., protection from potential compromise of large U-WSP). By careful design of web service interfaces, exchange of sensitive information can be separated from performance critical operations. For example, a real-time stock quote service can export two interfaces: an authentication interface that accepts an account password and returns a session identifier and a querying interface that accepts the session identifier and returns stock quotes. And instead of

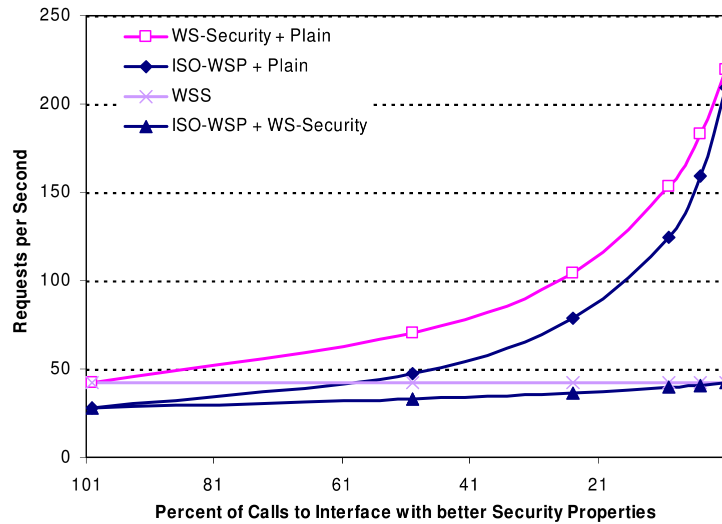

Figure 16. Comparison Web Service Throughput with a mix of differing Security Configurations. Interpretation of configuration: WS-Security + Plain at 1, implies that 1% of calls were made to the WS-Security-enabled implementation and 99% of calls were made to the Plain text implementation.

providing a single public key, the service provider exports two public keys: one for the authentication interface (AI key) and another for the querying interface (QI key). Since the authentication interface involves exchange of account password, this stage is processed using the AI key and the T-WSP. Once account password has been validated, information exchanges for retrieving quotes can be carried out using the QI key and the U-WSP, thereby limiting the performance impact of ISO-WSP.

To illustrate the effectiveness of such an approach, we measured the performance of our ISO-WSP implementation by varying the ratio of requests that have enhanced security requirements. We used three configurations of the payment processing service in our experiments: one that relies on plain text communication, one that uses WS-Security and one that uses T-WSP and a trusted credit card processor, as described at the start of Section 4.6. The maximum throughput of the web service implementation in each case was 252, 43, and 28 requests per second respectively. However to gauge the effect of using T-WSP only for a subset of the web server interface, we queried the different configurations in pairs, varying the percent of calls to the configuration with better

security properties. Figure X summarizes our results. When a portion of the interface uses plain text communication, we see that ISO-WSP underperforms the traditional secure configuration of WS-Security by around 25%, which drops down to 4% when just 1% of the calls require increased security guarantees. However, if all interfaces require the use of WS-Security, we see that securing a portion of the interface with T-WSP has low overheads (7% overhead when 10% of calls are secured with T-WSP).

4.7.3. Cost of Porting Application-level Code

To port an existing web service implementation to ISO-WSP, a developer has to modify or add code to existing code in four places: First the developer has to define SFIs for sensitive objects. SFI size is dependent on the number of functions that are exported by the sensitive object. On a related note, the developer has to add information about dummy objects to the configuration file, e.g., reference to a file containing a dummy object.

Second, the developer has to change the serializer and deserializer in the trusted and untrusted parts of the application to account for modified data items (Section 4.5.2). The number of modifications depends on the data binding framework used by the application, e.g., using Castor [24] as the data binding framework necessitates modifying just 2 LOC: one to modify the settings of the serializer and one to modify the type of object being serialized.

Third, the developer has to add code for cleaning up trusted objects. This involves adding just one LOC. And finally, the developer has to modify the trusted part of the application to interact with the T-WSP. Currently we use hand written code to bind the application to the T-WSP and hence this is a relatively major part of the porting process.

Table 8. Number of Lines of Code added or modified when porting legacy applications to ISO-WSP.

Service	Sensitive Functions	SFI	Untrusted Portion	Trusted Portion	Total (% Modified)
Payment Processor	2	5	3	32	40 (13% [a])
RUBiS [18]	6	9	12	28	49 (<1%)

[a] Payment processor service does not have the actual payment processing code. Hence, the percentage value is not truly representative.

Table 8 presents the number of lines added or modified when porting two applications, the aforementioned Payment Processor service and our port of the RUBiS benchmark [18]. We can see that we have to add or modify few tens of lines of code, which amounts to a small fraction of the application. Moreover, a majority of these modifications are required to the interface the T-WSP with the application level code. Given the standard nature of the code, we feel that code generation techniques, similar to the WSDL2Java code generators of Axis2 [4], can be employed to further reduce the number of modifications required.

4.7.4. Revisiting Design Goals

Once again, of the five design goals, we evaluated security properties, complexity reductions and performance overheads in the previous subsections. This section evaluates the design and implementation of ISO-WSP with respect to the other two requirements.

Reuse of Legacy Software. ISO-WSP reuses legacy WSP for handling non-sensitive or protected sensitive information. For handling sensitive information, ISO-WSP requires that the legacy WSP be modified to call the T-WSP. By doing so, ISO-WSP not only reuses legacy software, but it also retains most of the functionality of legacy WSPs.

Reuse of Interfaces. ISO-WSP relies on the legacy U-WSP to interact with the external entities. Additionally, ISO-WSP contains a Message Splicer which ensures that any changes to the data format (e.g., adding tokens) are not visible to external entities.

Therefore ISO-WSP ensures that the web service interface is retained. However the interface between the middleware and the application level code is modified in two ways: data format changes and instead of one, there are two application-level components per web service request or response. We showed with the use of SFI that these two changes can be incorporated into existing application level code with minimal modifications.

4.8. Discussion

4.8.1. Applicability to Other WSPs

We used Axis2 to present a proof-of-concept implementation of ISO-WSP. However, our approach is applicable to other WSPs such as Microsoft .NET. The procedure for adding a T-WSP to a legacy WSP is simple: first, the legacy WSP has to be modified to transmit and receive XML messages during processing of a SOAP message. Since WSPs are designed to handle XML messages, we can easily add such functionality or borrow functionality for other parts of the WSP. Secondly, we need to add modules that serialize and deserialize parameters passed to, or received from the T-WSP. This stage can be potentially time consuming depending on the programming language employed. Since Java provided serializers and deserializes are sufficient for most classes, we did not have trouble developing a T-WSP for Axis2.

4.8.2. Cost of Splitting Applications

The payment processing service example involved simple information flow between sensitive and non-sensitive objects – *cc.verify(...)* and *cc.chargeCard(...)* were the only functions that had to be invoked. This reduces cost of interaction between trusted and untrusted parts of the application. Secondly, having a clearly separation between sensitive and non-sensitive objects simplifies the process of splitting applications into trusted and untrusted part.

However, applications such as a Health Level 7 hub (HL7 hub) [10] contain more complex interactions between sensitive and non-sensitive information. These applications require greater effort to split, more LOC modified due to larger number of sensitive objects. Additionally, the resulting split is expensive as there will be more remote calls between the untrusted and trusted parts. Alternatively, one can treat the complete application as trusted and execute it in on top of the U-WSP. In such as case, the application has to be modified to interact with the T-WSP – few LOC. There is only one remote call from the untrusted to the trusted part of the application to trigger the execution of the application. The alternate approach, while minimizing the cost of porting and executing the application on the ISO-WSP, results in a larger trusted component. We feel that this tradeoff is dependent on the application-level code complexity. If the application-level code is amenable to exhaustive testing or formal analysis [96] (of the order of few thousand lines of code), then it might be easier to treat the complete application as trusted. Otherwise, we have to split the application to improve its security properties and incur the added costs of splitting.

CHAPTER 5

END-TO-END SECURITY IN THE PRESENCE OF MISBEHAVING INTERMEDIATE SERVICES

Service Oriented Computing standards such as SOAP (Simple Object Access Protocol) and WSDL (Web Services Description Language) have enabled the development of a new class of services and applications called Web service compositions (also referred to as *composition* for brevity). Web service compositions are distributed applications that aggregate content from multiple sources and add value through composition, annotation and refinement (e.g., electronic prescription system in Figure 17). However, these distributed applications also exacerbate the general trade-off between the growing functionality from many components and the weakening of end-to-end system properties such as performance, reliability, security, privacy, and trust, which are dependent on the weakest link of the many components.

We first discuss the flow of information in web service compositions. Using an electronic prescription system, we present the need for an end-to-end security framework. Next, we present WS-FESec [105], a fine-grained, end-to-end framework aimed at preserving integrity and confidentiality in composite web services. We then evaluate the security properties and the performance implications of WS-FESec. We show that WS-FESec is flexible enough to support the lattice model of secure information flow while incurring manageable performance overheads of few tens of milliseconds per encryption or signature.

5.1. Information Flow in Web Service Compositions

In a simple web service invocation, there is a single flow between the service provider and the client. Once again, we assume that this flow is composed of two types of

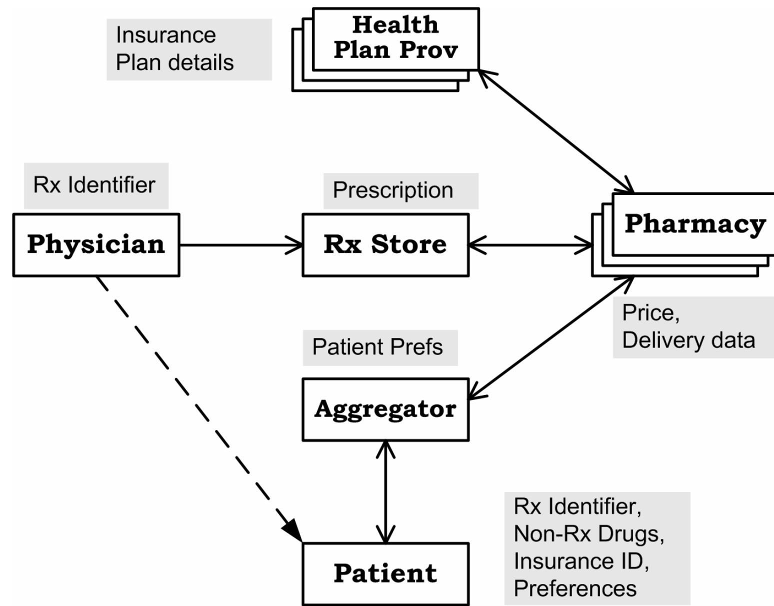

Figure 17. Electronic Prescription Application. Patient contacts aggregator to identify the closest, cheapest or fastest delivery pharmacies. Shaded boxes indicate data items stored by each entity. Broken line indicates personal interaction.

data items: security-sensitive and security-insensitive data items. In the case of simple flows, protocols such as SSL and TLS can be used to protect security-sensitive data items. On the contrary, in web service compositions, information flows via multiple intermediate services before reaching the final consumer. We use an electronic prescription system to illustrate the problem of securing information flow in web service compositions.

5.1.1. Electronic Prescription Application

Electronic prescription systems [45][9] are an example of relatively straightforward e-commerce applications with significant security requirements. As the starting point, the physician writes an electronic prescription, stores it in a trusted prescription store, and provides the patient with a prescription identifier. The patient then provides this identifier to a pharmacy, which retrieves the prescription and fills it. The pharmacy also contacts the health plan providers to determine the deductible and excess costs not covered by insurance and charges the patient accordingly.

Table 9. Structure of an Electronic Prescription [9]. Superscripts indicate classes of information. Pt: Patient, Phy: Physician, Pharm: Pharmacy, Ins: Insurance, Drug: Drug.

Field	Description
Demographics^{Pt}	Patient information, including Optional Insurance information
Clinical decision support rules base^{Pt}	Information to pharmacies to make a case-by-case decision on selecting drugs
Other Information^{Pt}	More Patient details like allergies
Prescriber^{Phy}	Clinician Information
Pharmacy^{Pharm}	Information for a selected Pharmacy.
Prescription^{Drug}	List of Drugs
Drug Dictionary^{Drug}	More information about drugs
Drug Reference^{Drug}	Optional. Information for Patient or pharmacist, e.g., patient ed. materials
Formulary Info.^{Ins}	Detailed Insurance Information
Other functions	Optional. Patient Compliance History, etc

We add an aggregator service, one that attempts to provide a patient with better choice of suppliers, to an electronic prescription application. The modified application is described in Figure 17. Sites such as New York State Attorney General's Office Prescription Drug Website (www.nyagrx.org) allow a patient to compare prices of individual drugs at various pharmacies. An enhanced aggregator service can be designed to compare costs of complete prescriptions and additional non-prescription drugs that a patient might need. Additionally, it could also allow a patient to select pharmacies based on drug availability, delivery mechanism and proximity. In short, the aggregator service provides the patient with an easy way to compare pharmacies with minimal increase in costs, in terms of both time and money.

5.1.2. Security Requirements in an Electronic Prescription Application

The following information is exchanged between the physician, patient, and the component services:

- **ERx:** Electronic prescription (described in Table 9).

- **NRx:** Non-prescription drug list.
- **FinInfo:** Patient's financial information such as bank account number or credit card number.
- **Addr:** Pickup or delivery address of patient.
- **PharmReply:** A pharmacy's reply to pricing queries, or purchase orders. It contains pricing information and delivery options.

The security requirements are as follows:

Conf-1. Availability of complete prescription should be limited to patient, physician, pharmacy that fills the prescription and the patient's insurance company.

Conf-2. Patient's financial information is available only to the pharmacy that fills the prescription.

Int-1. Only the physician is allowed to modify a prescription. The pharmacist filling the prescription would also like source verification of the prescription to detect fraudulent or inappropriate prescriptions. Therefore, this also doubles as an NRO requirement.

Int-2. The cost of filling a prescription, as specified by a pharmacy service (in PharmReply), cannot not be modified by any other service.

The above described security requirements restrict access to data. However, the electronic prescription application also has usability requirements that allow the aggregator service and the various pharmacies to provide a value added service. The requirements are:

Usab-1. For price comparison, pharmacies are allowed to look at the list of drugs in the prescription, *Drug* class in Table 9. Care should be taken to avoid compromising *Conf-1*: all pharmacies should not have access to complete prescription information as that would identify the patient.

Usab-2. The aggregator service must be allowed to look at the delivery address (coarse grained information, e.g., zip codes) to identify nearby pharmacies.

Usab-3. The aggregator service must be allowed to look at the total cost of the prescription to filter pharmacies based on price.

Usab-4. The aggregator service can be allowed to look at non-prescription drugs to allow the user to interactively select appropriate type and dosage.

Usab-5. The aggregator service may be allowed to modify the total cost of non-prescription drugs, and therefore, the cost of the order, to account for changes in patient choice. This feature enables the aggregator service to respond immediately to changes in patient choice, e.g., increase the number of units of a non-prescription drug.

Open Environment. Since the pharmacy services reside on the Internet, the patient typically does not know of all online pharmacies. Also, the aggregator services are known only to the patient and are not *fully* trusted by either the patient or the prescription store.

5.2. Need for Fine-Grained, End-to-End Security Framework

From the electronic prescription application, we see that web service compositions can operate in an open environment. They require mechanisms to enforce *fine-grained confidentiality and integrity policies*, e.g., a policy that distinguishes between different parts of a prescription, non-prescription drugs, shipping address and financial information.

Web service compositions also require mechanisms to enforce *end-to-end confidentiality policies,* e.g., a policy specified by either the patient or the pharmacy that selectively allows the aggregator service read access to portions of message exchanged between them. The unencrypted portions of messages between the pharmacies and the patient require an *end-to-end integrity policy* to prevent the aggregator service from maliciously modifying them. Additionally, the pharmacy requires NRO on the complete prescription.

Since the WS-Security specification supports fine-grained encryption and signatures, it can be used to enforce fine-grained security policies. However, WS-Security requires the developers to specify low level policies. We would like a higher level abstraction where the developer groups data items in a message based on confidentiality and integrity requirements, which are then transformed into WS-Security policies.

The WS-Security specification does not handle open environments well. First, WS-Security needs a shared secret between the source and the recipient to distribute the encryption keys. In an open environment, establishing a shared secret between a source and *each and every* potential recipient is not always feasible, e.g., the prescription store does not know of all pharmacies at message generation time.

Secondly, for every encryption key, WS-Security uses a single key reference (e.g., a WS-SecureConversation context [28] or encryption key encrypted with the public key of the recipient). Therefore, if multiple recipients have to be able to decrypt the message, all of them have to understand the same key reference. This in turn implies that all of them possess the same shared secret. Once again, this is not feasible in an open environment as services execute at different security levels, e.g., from the patient's perspective, atmost one pharmacy executes with a higher security clearance and is allowed to view patient information in the prescription. Therefore, we need a framework that can support end-to-end security properties in open environments.

5.3. Overview of WS-FESec

WS-FESec employs a fine-grained approach to maintain end-to-end confidentiality and integrity in composite web services. WS-FESec relies on fine-grained, field-level digital signatures to allow for value-adding modifications (e.g., composition, annotation and pruning) by intermediate web services. Digital signatures satisfy the NRO property and also allow recipients to verify the source of data items in a message.

Similarly, WS-FESec relies on field-level encryption to allow for fine-grained access to confidential data. WS-FESec is end-to-end because digital signatures can be verified by any recipient and does not depend on the correct behavior of intermediate services. Similarly, by limiting the encryption-key distribution to authorized services, WS-FESec also enforces end-to-end confidentiality of data items. WS-FESec currently does not currently address issues of availability or denial of service attacks.

Let WS_S be a data-generating source web service. Let WS_R represent the recipients of this data. WS_R consists of other web services, web service applications or end-users. WS_S wants to protect the confidentiality and integrity of the generated data. We assume that a web service message consists of distinct fields. These fields are then grouped according to the confidentiality and integrity policy for the web service. Each policy specifies groups of fields that have same confidentiality requirements (*ConfG*) or integrity requirements (*IntG*). Each group of data items is then encrypted or signed separately before transmission. Section 5.4 describes the composition of *IntG*s and discusses the signature mechanisms. Section 5.5 describes the construction of ConfGs and the encryption and key distribution mechanisms. Section 5.6 discusses the interaction between confidentiality and integrity requirements. In each of these sections, we also discuss how the WS-Security specification or implementation can be augmented to support WS-FESec. We use the WSS4J implementation [2] in this dissertation.

5.4. Addressing Integrity Requirements

In this section, we assume that WS_S does not have a confidentiality policy. This is common for services that deal with publicly available information, e.g., a delayed stock quote service. We assume an open environment, i.e., there is no restriction on who can be a member of WS_R. Based on our discussion in Section 5.1.1, we need a framework that is flexible enough to permit value adding modifications like pruning and annotation by

intermediate services, while at the same time minimizing the damage to the integrity of the original data items.

We achieve integrity protection in three steps: First, we require the message originator(e.g., the web service developer) to specify Integrity Groups on the outgoing message. Next, we sign these groups individually before transmitting the message. Finally, we allow the recipient to specify integrity policies that can work with invalid or missing signatures in messages.

5.4.1. Composition of Integrity Groups

The idea of Integrity Groups (*IntG*) is to identify portions of the message that are independent from the rest of the message. This process is carried out by the designer or the developer of a web service. By doing so, we split a message into multiple groups, each of which can be processed separately, without loss of integrity, by the intermediate web services or by the final recipient.

For example, consider an auction listings web site that supports a querying interface. A query can return multiple results that can be filtered by an intermediate web service. If each result is treated as a separate *IntG* and signed independently, then intermediate web services can drop some results without affecting the integrity of the other results.

5.4.2. Signing IntGs

WS-FESec allows the service developer and user to think in terms of integrity groups instead of writing detailed policies. WS-FESec converts these integrity groups in to configuration parameters for a WS-Security implementation (WSS4J in our case). WS-Security allows for messages to be signed with either digital signatures or Keyed message authentication code (HMAC). Digital signatures additionally satisfy the NRO requirement. However, they are computationally expensive. HMAC is cheaper than

digital signatures, but requires a key distribution infrastructure and cannot be used to satisfy NRO requirement. Key distribution for HMAC signatures is discussed in greater detail in Section 3.2.2.

5.5. Addressing Confidentiality Requirements

Our framework utilizes fine-grain encryption to satisfy confidentiality requirements in web service compositions. As data items are now encrypted with a secret key, key distribution now becomes important. Encrypting data items also forces us to rethink the way confidentiality groups are composed. We first address the composition of confidentiality groups and then discuss key distribution mechanisms.

5.5.1. Composition of Confidentiality Groups.

Confidentiality group (*ConfG*) is a set of fields grouped by together confidentiality requirements. *ConfG* differs from *IntG* in two ways: first, *ConfG* does not indicate independence of data items from the rest of the message. Rather, it indicates that a group of items is at a different security level from the rest of the message. Secondly, *IntG* allows a field to appear in multiple groups. On the other hand, fields can appear in at most one *ConfG* as each *ConfG* contains *values* of fields. Therefore, if a field appears in two ConfGs, it can lead to two different values for the same field.

Therefore, each *ConfG* is now encrypted with a separate encryption key (henceforth referred to as *ConfG* Keys). The outgoing message contains all the encrypted data items. Confidentiality requirements are satisfied by limiting the availability of the decryption keys amongst the recipients.

5.5.2. Key Distribution

Key distribution, for both HMAC and encryption, is predicated on two main factors. First, it depends on the existence of a secret context (or a shared secret) between

the source and the recipient. Secondly, a source's knowledge of the potential recipients also affects key distribution.

5.5.3. Establishing Secret Context between WS_S and WS_R

Secret contexts are used to wrap or refer to *ConfG*s Keys. Since the secret context is shared between the source and the recipient, it does not have to be transmitted over the network. *Public keys, WS-SecureConversation* contexts [28] and registration time secrets (e.g., passwords) are examples of secret contexts.

5.5.4. Source Web Service's (WS_S) knowledge of Recipients (WS_R)

Since the encryption keys have to be kept a secret between the source web service and each of the recipients, the source web service's knowledge of recipients becomes an important factor in the design of a confidentiality framework. Key management is easier and more efficient if WS_S knows of all members in WS_R. In such a scenario, we just use the secret context (or a key derived from the secret context) to encrypt the appropriate *ConfG* keys and transmit the encrypted keys with the message.

In the more general case, where the recipients might not be known at message generation time, we transmit the encrypted message, but require the recipient to call back the source web service to receive the key. When the recipient calls back, the WS_S might require the recipient to authenticate itself before wrapping the keys with the secret context and transmitting them.

5.5.5. Encrypting ConfGs

We cannot directly use WS-Security to encrypt messages or distribute keys because of key distribution problems mentioned earlier (Section 5.2). The WS-Security specification has to be enhanced in two ways to support WS-FESec. First, WS-Security needs an *on-demand* key distribution mechanism. We do so by extending the *TokenType*

- */fesec:CallbackReference/@URI*
  This lists the URI that the recipient has to call and authenticate itself to receive the key.
- */fesec:CallbackReference/@fesec:AuthMechanism*
  This contains the authentication techniques supported by the URI. Allowed values include UsernameToken or BinarySecurityToken (refers to X.509 certificates or WS-SecureConversation contexts).
- */fesec:CallbackReference/@fesec:MsgID*
  This is a unique identifier for the message, that identifies the decryption key.

Figure 18. Details for the CallbackReference Type

attribute of *SecurityTokenReference* element of the WS-Security specification [15] with a new type, *CallbackReference*. The CallbackReference type is illustrated in Figure 18.

Secondly, WS-Security uses a single KeyInfo structure per secret key (for encryption or HMAC). This assumes that all services that need access to the secret key share the same secret. The assumption does not hold in web service compositions, where recipients operate at different security levels and therefore might not share a secret. We augment the WS-Security specification to explicitly allow for multiple KeyInfo structures per secret key. This implies that a secret key can be wrapped in multiple ways in a single message, thereby allowing recipient services to independently access the protected contents.

It is straightforward to use this modified WS-Security specification to encrypt messages given the ConfGs and the set of authorized recipients.

5.6. Interaction between Integrity and Confidentiality Requirements

In cases where WS_S would like to enforce an integrity policy in addition to a confidentiality policy, WS-FESec applies the confidentiality policy before the integrity policy. This order of application allows recipients at lower security levels to verify signatures that are computed over higher security level data, e.g., the aggregator service in the electronic prescription application can verify the authenticity of the prescription without having to access the actual prescription.

However, this limits the granularity of signatures to that of the encrypted elements. WS-Security allows for complex data structures to be encrypted as a block, replacing the entire data structure with a single encrypted field. Therefore, the granularity of signatures is limited by the confidentiality policy. However, this problem can be easily bypassed, if necessary, by enumerating the elements in the data structure instead of specifying the top level data structure in ConfGs.

5.7. Evaluation

5.7.1. Security Properties of WS-FESec

Web Service Compositions and Lattice Model: Security levels and clearances in web service compositions can be modeled as a lattice [52]. In a web service composition, data producing web services assign security levels to objects. Security clearances are authorization tokens provided by the data producing web services or by trusted third parties. If the component web services in a composition agree on a uniform security classification, then security levels of the data items form a lattice. But a uniform classification is not always possible in an open environment. Although there are many candidate lattices, they all share the lowest classification level: *Public*. To simplify our lattice model, we introduce a *High* classification that dominates all other security classifications. Using these two classification levels, we combine individual lattices to generate a single lattice. This simple model does not make any additional assumptions about prior knowledge of other web services at data generation time and is therefore suited for the open environment of web services.

WS-FESec is designed to support the lattice model of multi-level security. Each security level is represented by a *ConfG* as described in Section 5.5.1. Each *ConfG* is then encrypted as specified by WS-FESec (Section 5.5.5). Security clearances are handled using authentication and authorization mechanisms. Secret contexts, as described in

Section 5.5.3 give some examples of authentication and authorization mechanisms. Only the encrypted messages are distributed to the intermediate services. Web services with security clearances either query the data producing web service to retrieve the key or use their part of the secret context to retrieve the keys from the message. A web service gets keys for all *ConfG*'s that its security clearance level dominates. And since a single web service might have multiple clearance levels, one for each data producing web service, it might have to query multiple services to realize its clearance level. Effectively, this mechanism allows us to limit the access of intermediate services at a data item granularity.

End-to-End Confidentiality for the Electronic Prescription Application. We now briefly go over how WS-FESec satisfies the security and usability requirements mentioned in Section 5.1.2. The data items exchanged in the electronic prescription application can be categorized using a single, linearly ordered lattice (in increasing order): *Public*, *Semi-Private* and *Private*. The *Public* class includes information on non-prescription drugs, coarse-grained patient address, pharmacy address and delivery options. The *Semi-Private* class includes prescription drug information and anonymized insurance information. The *Private* class contains patient and physician information and financial data.

Each class of data is signed individually to satisfy fine-grained integrity requirements (*Int-1, Int-2*). Public data is not encrypted whereas the Semi-Private and Private data are encrypted with separate encryption keys. Access to data by the various services is controlled by limiting the distribution of the encryption keys. By transmitting Public data in plain-text format, we allow the aggregator service to look at pricing and delivery information to provide value added services (*Usab-2* to *Usab-5*).

Based on the confidentiality requirements (*Conf-1, Conf-2*), initially, only the physician (and by extension, the trusted store) and the patient (and the patient application) require access to Private and Semi-Private data. This is achieved by the

physician providing the encryption keys during personal interaction with the patient. To satisfy the usability requirement for the pharmacies (*Usab-1*), the trusted store provides the key for semi-private information to all authorized pharmacies. We assume the existence of an authentication infrastructure as described in [45].

Finally, when the patient decides to fill the prescription, the patient application provides the encryption key for Private data (encrypted with the public key of the pharmacy) to the appropriate pharmacy. During the whole process, the aggregator service and all pharmacies, except one, do not have access to confidential patient information. However, at the same time, they have access to enough information to provide a useful, pricing service to the patient.

5.7.2. **Performance**

We use a prototype implementation on top WSS4J to understand the performance implications of using WS-FESec. We implemented a simple Stock Quote service, that exports two interfaces: a delayed quote service that transmits stock quotes in plaintext and a real-time interface that encrypts stock price and transmits the rest of the contents (company name, time last updated, volume traded, etc…) in plain text. Each interface accepts an array of symbols and returns an array of stock quotes. The whole setup was then executed on two 3-GHz Pentium-4 machines with 1 gigabyte of RAM each, connected via a gigabit network. The machines run Linux kernel 2.4.27 and Apache Axis 1.4 along with WSS4j 1.5.1 on Jakarta Tomcat 4.1.31.

We ran three sets of experiments, evaluating the performance of fine-grained signature, fine-grained encryption and fine-grained encryption with signatures. For fine-grained signatures, we signed each quote in the response message separately. For fine-grained encryption, we encrypted each stock price individually. For fine-grained encryption with signatures, we first encrypted each price of each stock before signing the resultant quote. In each set of experiments we compare the performance of WS-FESec

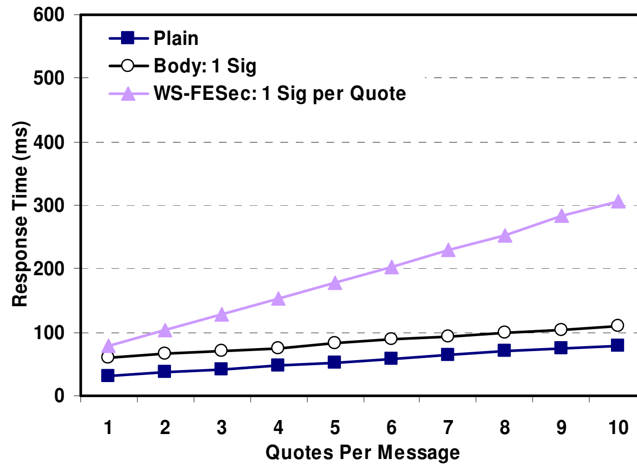

Figure 19. Performance with Multiple Signature per Message

against coarse-grained techniques, i.e., signing or encrypting the message body as a whole.

Figure 19 presents the results for multiple signatures per message. We see that signature overhead is about 20 ms per *IntG,* i.e., per signature. We see such a high cost because of the use of expensive public key cryptography for signatures. Figure 20 presents the results for messages that contain multiple, separately encrypted portions. We see that encryption overhead is about 22 ms per encryption key. Symmetric key encryption is as expensive as signatures because we use RSA to wrap the encryption key. We use two levels of encryption: a single channel mode where all stock prices are encrypted with the same key and a multi-channel mode where each stock price in the message is encrypted with a different key, thereby simulating multiple levels of security in the same message. Though the single-channel mode and full body encryption both use a single key, single channel mode is more expensive (about 5 ms per quote) because WS-Security has to parse the XML document to locate elements for encryption. Combining signatures and encryption gave us similar results, with the overhead amounting to about 45 ms per quote in the multi-channel mode which is expected because encryption and signatures are orthogonal in our implementation.

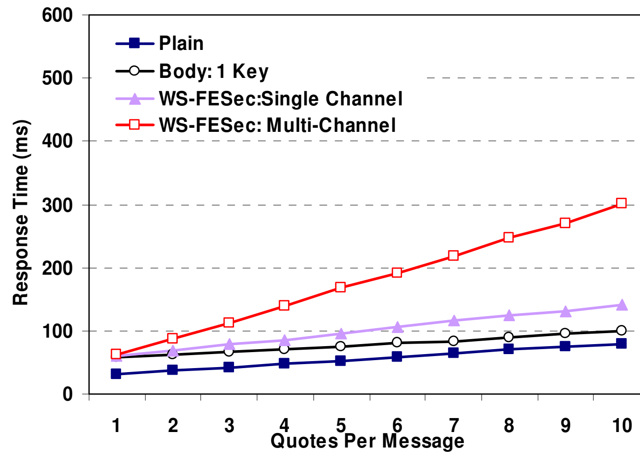

Figure 20. Multiple Encryptions per message. Single Channel refers to encrypting all stock prices in a response message with the same key. In Multi-Channel, every stock price in the message is encrypted with a separate key, creating multiple security levels in the same message. Encryption keys are wrapped with RSA.

While a cost of few tens of milliseconds per signature or encryption might seem expensive at first, WS-FESec allows for services to be combined in novel ways in an open environment without compromising the security of information flow. For example, fine-grained encryption of prescriptions allows for an aggregator service as described in Section 5.1.1. We also expect encryption costs to decrease significantly with the use of WS-SecureConversation [28] as it can function without expensive public key operations.

5.7.3. Interaction with Composition Languages

Composition languages such as BPEL4WS **[5]** assume that there is an underlying security mechanism such as SSL or WS-Security that maintains the integrity and confidentiality of data. Since WS-FESec is layered on top of WS-Security, it can be employed as a security mechanism. However, WS-FESec introduces two new wrinkles that BPEL4WS developers have to take into account: First, the end-to-end confidentiality policy might prevent some data items in a message from being available to the intermediate service, e.g., the aggregator service in the electronic-prescription application does not have access to prescription. Hence business process decisions in intermediate services must only depend on data items made available to them in plain-text format.

This is not a limitation as the confidentiality policy explicitly forbids the intermediate web service from looking at the encrypted data items.

Secondly, WS-FESec depends on WS-Security headers being forwarded to verify signatures or locate encryption keys. Hence, BPEL4WS developers might also have to propagate WS-Security headers in addition to portions of the message body. BPEL4WS tools like Oracle's BPEL Process Manager already supports this feature, wherein the programmer can manipulate WS-Security headers defined in the WSDL document just like message body elements.

5.7.4. Limitations of WS-FESec

WS-FESec use fine-grained cryptography to satisfy integrity and confidentiality requirements in web services. However, there are two situations where fine-grained cryptography may be insufficient to preserve integrity or confidentiality: First, WS-FESec relies on some key data fields remaining untouched by intermediate web services. But there are some services that may overwrite all important fields of a message, e.g., text to speech services.

Second, WS-FESec assumes that individual web services are not malicious and that they have not been compromised. Therefore, it assumes that any web service that has access to confidential data does not distribute the keys or the confidential data to unauthorized entities. Although this assumption is routine for static composition, the situation is more complex and interesting for dynamic composition.

We feel that the problems mentioned above can be mitigated with appropriate hardware support. Trusted Computing principles (and its web service counterparts such as WS-Attestation [116]) such as authenticated booting and remote attestation allow a program to gauge the correctness of a remotely executing service and the rest of the software stack. This property can be used to check services that completely transform data or improve resilience to information leak attacks.

CHAPTER 6

RELATED WORK

Attackers can target flow of sensitive information at three places: on the network, at the intermediate nodes and at the end-point nodes. Attacks on the network are generally countered with cryptographic techniques. With appropriate authentication and key distribution techniques, encryption and signatures can protect web services from information leak or information corruption attacks. SSL [61], TLS [53], and *https* [94] are widely used to protect information flows that involve the browser. The WS-* frameworks aim to provide security properties such as authentication, confidentiality and integrity for web services. WS-Security [15] is a framework for providing quality of protection to SOAP messages. WS-Security provides mechanisms for ensuring message integrity, confidentiality and authenticity. Web Services Trust Language (WS-Trust) [29] is an extension to WS-Security that provides means to establish trust relationships amongst differing trust domains. Other frameworks such as WS-SecureConversation and WS-SecurityPolicy aim to build on top of WS-Security and WS-Trust to provide other features such as establishment of a secret context or specify policy assertions.

These protocols form the basis for protection of information flow on the network and AppCores and WS-FESec leverage these protocols to provide stronger security guarantees: e.g., AppCore reduces the code base that has access to plain-text sensitive data and WS-FESec provides end-to-end security guarantees in web service compositions. We now discuss related work in the fields of protecting information from vulnerable end point software and misbehaving intermediate nodes.

6.1. Protection against Vulnerable Software

6.1.1. Malicious Software

Trusted computing [25][24] provides mechanisms like authenticated booting and remote attestation to protect against malicious or unauthorized software. The Integrity Measurement Architecture [97] extends these mechanisms to include other key components of a system's runtime like dynamically loadable libraries and scripts. The WS-Attestation framework [116] leverages trusted computing primitives to protect against unknown or malicious software in web services. In conjunction with the above described WS-* security frameworks, these techniques protect against a large class of software modification attacks.

However, software vulnerabilities in the TCB can be exploited by attackers to gain access to sensitive information. Vulnerabilities in the operating system can be exploited to modify the memory of running programs. Vulnerabilities in middleware (e.g., libraries, web service platform) and support software (X server, databases) can be either used to directly gain access to sensitive data or to modify software with access to sensitive data, e.g., by tweaking configuration files. Vulnerabilities in the web service itself can be used to bypass access control mechanisms and gain unauthorized access to sensitive data. This problem is tackled by restricting the information flow between applications and minimizing the number of vulnerabilities in applications in the TCB.

6.1.2. Reducing Vulnerabilities in Software

Static analysis and verification techniques can detect bugs and vulnerabilities [56][57][59][112]. However, they work best on relatively small code base due to scalability problems.

CCured [87] is a type system that attempts to retrofit security into legacy code. CCured extends the type system of C, which allows for static type checking of pointer

usage. In cases where static type checking is insufficient, CCured instruments the program for dynamic type checking. This approach prevents a large number of pointer-based attacks such as buffer overflows. Castro et al. [44] use static analysis to instrument loads and stores in programs to maintain the integrity of data flow. While these techniques detect vulnerabilities or protect against a major class of vulnerabilities (data corruption attacks), their scalability, especially to software as large and complex as a browser, is still an open question. More recently, Ganapathy et al. [63] have attempted to automate the process of retrofitting authorization policy enforcement into existing software. Their work involves program analysis of existing software to identify code fragments requiring authorization policies. One example is the availability of events in X Server window manager: the argument being that remote windows should not be allowed access to keyboard or mouse events pertaining to local windows.

There is also work on defenses against specific types of vulnerabilities, e.g., Lhee and Chapin [80] provide an overview of defenses against buffer overflow and format string vulnerabilities including tools that rewrite source code or binaries to eliminate or trap buffer overflows, SQLRand [40] and AMNESIA [65] are defenses against SQL injection attacks. Address space randomization [39], instruction set randomization [36] and N-Variant systems [50] are some techniques used to combat arbitrary code execution vulnerabilities.

The above described techniques are orthogonal to the AppCore approach. Since the resulting trusted components are small and simple, static analysis techniques are more effective. Additionally, since AppCores are invoked only for security-sensitive processing, we expect the costs of run-time protection to be lesser when applied to AppCore than to complete legacy applications.

6.1.3. Refactoring Software

As mentioned previously (Section 2.6.2), there is considerable work on splitting applications and system services to minimize the code that has access to sensitive data, e.g., Privilege separation [41][76][90]. Privilege separation works well when the security-sensitive resources are well-defined (e.g. ports < 1024, RSA private key). This is not the case in browsers or web service platforms, where sensitive and non-sensitive data use the same resources and have nearly similar information flow paths. Automatic privilege separation also does not scale well, especially for complex applications like the browser.

Substantial research has gone into refactoring and customizing middleware to modularize and simplify them. Zhang et al. [118][119] and Eichberg et al. [55] use Aspect Oriented Programming techniques to modularize middleware and then, customize it according to the needs of applications. OpenCOM [49] and CompOSE/Q [111], amongst others, are examples of reflective middleware that allow for customization of middleware to suit the needs of application. ISO-WSP is an attempt at refactoring middleware with security as the driving factor. Techniques described in [118][119] are applicable to the construction of ISO-WSP.

The Proxos approach [109] splits the interface between the operating system and applications into security-sensitive and security-insensitive parts and uses a smaller and simpler operating system to implement the security-sensitive parts of the interface. The design goals of the Proxos approach are similar to that of AppCore. However, a large number of vulnerabilities exist in current application software and middleware. The AppCore approach addresses this issue by additionally simplifying application software and middleware.

6.1.4. Information Flow Restrictions

By restricting the flow of information between applications, we reduce the impact of security vulnerabilities. On a system-wide level, information flow analysis can be used

to control communication between processes. Mandatory access control systems such as Asbestos [54], capability-based system such as EROS [102] and Authority Based Access Control systems such as Polaris [108] aim to control the flow of information or minimize the damage due to compromise of software handling sensitive information. The AppCore approach is orthogonal to these techniques. In fact, by splitting the flow of information in a single application amongst multiple client-side application-level software, AppCore allows for more rigorous enforcement of access control. For example, browsers have to be allowed to connect to the network, opening up avenues for information leaks. On the other hand, the TrustedViewer in BLAC is not expected to communicate over the network; it only needs to access the https proxy and a limited number of system services such as the display manager. Hence, the access rights of a TrustedViewer can be constrained to a greater extent than those of a full-fledged browser.

XFI [58] takes a novel approach to protect software from untrusted extensions by using an untrusted rewriter to instrument the extension with control flow guards and memory access guards. The guards are then verified with a small trusted verifier and the extension is now deemed safe for execution. Such techniques can be applied to protect software such as the browser from malicious extensions. However, the original software program can still be exploited if it contains vulnerabilities.

Tokens used in ISO-WSP by untrusted applications to operate on security-sensitive data can be viewed as capabilities [114]. The ISO-WSP architecture can be considered as retrofitting a capability-based architecture in to WSPs. Protected Data Paths (PDP) [78] uses a similar approach to hide sensitive information from untrusted application level programs. PDP consists of kernel-level modules that traps I/O calls retrieving sensitive data and replaces sensitive data with tokens, thereby preventing application level programs from directly accessing them. ISO-WSP not only adds tokens, but it also sends back dummy data items with similar structure to the sensitive data item, thereby minimizing the changes to existing applications.

Information flow analysis can also be used at a finer granularity to identify incorrect flows within a single program. Language-based information flow analysis [86][96] relies on programmer annotations, called labels, that denote the integrity or confidentiality level of a data item. Static analysis is then used to analyze the flow of sensitive information in a single program to identify flows that violate the confidentiality or integrity policies of the data items. The advantage of such analysis is that it rules out most forms of information leaks. However, we have to rely on the programmer to identify and annotate all sensitive variables accurately. Static analysis may also not scale to large and complex systems like browsers and web service platforms.

These information flow based techniques will be more effective if we have small and simple trusted components. At the same time, smaller components present a smaller profile to the attacker, reducing the chances for a successful attack.

6.1.5. Browser Defenses

Given the large number of vulnerabilities in the browsers, there is a lot of research addressing vulnerabilities in the browser. Research efforts range from interface personalization to thwart spoofing attacks [110], password hashing using browser extensions to limit the damage done by leaking of passwords [95], rewriting scripts in HTML pages to prevent them from exploiting known vulnerabilities [93], and trust indicator extensions for browsers to prevent interface spoofing and phishing attacks [120]. All these systems assume that the browser can be trusted with high-sensitivity information. Given the large number of vulnerabilities (average of 2 per month [21][22]) and the types of vulnerabilities (arbitrary code execution, security-system bypass) in current browsers, attackers can exploit them to bypass the protection afforded by many of these systems.

The Tahoma architecture [51], on the other hand, assumes that the browser cannot be trusted. Tahoma proposes executing browser instances in separate VMs, similar to

VMWare's Browser Appliance [27]. Additionally, Tahoma requires service providers to define a manifest, which is used to control behavior of the browser. Tahoma does not address the issue of a multitude of browser extensions (e.g., weather forecast extensions) that periodically talk to a site outside the service provider's manifest. FlowGuard also treats the browser as untrusted but it prevents the flow of sensitive information to the browser, instead of controlling the behavior of a browser instance that has access to sensitive information. Therefore, it does not have to curtail the browser's functionality.

6.2. Protection from Misbehaving Intermediate Nodes

In one-on-one communication, intermediate nodes can be treated as a part of the network and the security protocols discussed above will protect the flow of sensitive information. However, web service compositions allow for intermediate nodes to look at or modify portions of the message. This issue is addressed by the WS-Security framework which allows for fine-grained signatures and encryption of message contents. This property has been leveraged by distributed Security Authorities [101] and WS-FESec (Chapter 5) to enforce end-to-end security in web service compositions.

Reputation based schemes have been used to assess trustworthiness of web services and web service compositions [81][83][115]. Quality of service information too has been used to select services involved in a composition [84]. WS-Trustworthy [117] uses a multi-layered approach to determine trustworthiness of service implementations. Unlike reputation-based systems, WS-FESec is narrowly targeted to satisfy end-to-end confidentiality and integrity requirements. WS-FESec also guarantees that intermediate services, however trustworthy, do not have unauthorized access to data. This prevents attacks wherein a service provider builds up trust for some duration before it turns malicious. We are exploring ways of integrating trusted computing principles with WS-FESec.

Bengetsson and Westerdahl [38] use digital signatures to securely build composite web service in a cooperative environment. However, they employ message-level signatures which are invalidated at the smallest modification.

WS-Broker [43] proposes a broker-based architecture to match the client's security requirements with those of component services to generate a secure composition. Charfi and Mezini [47] use aspects to generate security conscious web service compositions. WS-FESec provides an enforcement mechanism for fine-grained, end-to-end integrity and confidentiality policies. Therefore, we can reuse techniques described in Charfi and Mezini to carry out WS-FESec processing.

Chafle et al. [46] use decentralized orchestration to provide confidentiality in composite web services. A composite web service is broken into separate portions to satisfy data flow constraints, confidentiality being one of them. WS-FESec too enforces confidentiality policies, but it reuses existing channels of communication, is therefore more flexible. WS-FESec opens a new channel only if the recipient and the source service do not share a secret context.

CHAPTER 7

CONCLUSION

7.1. Summary

Improving end-to-end security properties in web-based applications requires addressing end-point software vulnerabilities. Additionally, web service compositions require extensions to existing security protocols to maintain end-to-end security properties even in the face of misbehaving intermediate service providers. This dissertation presented three contributions towards addressing the above mentioned problems.

First, we presented the AppCore approach which splits existing software into two a small and simple trusted part that handles sensitive information and a legacy, untrusted part that handles non-sensitive information. Not only would the AppCore approach avoid many common and well-known vulnerabilities in the legacy software that compromised sensitive information, we also postulated that it would greatly reduce the size and complexity of the trusted code, thereby making exhaustive testing or formal analysis more feasible.

Secondly, we applied the AppCore approach to two real-world applications: one for client-side used in https-based applications and another for web service platforms (WSPs) that are used on both the client and server side in Service Oriented Computing. The resultant AppCores were smaller and simpler (over 10X simpler for https-based applications and 5X simpler for WSPs) and resulted in manageable overheads (few hundred milliseconds for https-based applications and few milliseconds for WSPs). We also showed that legacy browsers and WSPs can be adapted to interoperate with the corresponding AppCores with minimal modifications. Since the resultant AppCores

reused legacy interfaces, we also saw that they interact smoothly with remote entities executing legacy code.

Thirdly, to protect communication between clients and service providers, we presented WS-FESec, an open, fine-grained, end-to-end framework for preserving integrity and confidentiality in web service compositions. WS-FESec leverages WS-Security for fine-grained encryption and enhances WS-Security to support end-to-end security in open environment. We showed that WS-FESec is flexible enough to support the lattice model of information flow. Since information flow in web service compositions can be modeled as a lattice, WS-FESec can be used to secure information flow in web service compositions. WS-FESec can also be easily integrated with composition languages like BPEL4WS. In our performance evaluation of WS-FESec, we showed that the gains in end-to-end confidentiality and integrity compare favorably with few tens of milliseconds of computation overhead.

Current end-point software does not distinguish between the flow of sensitive and non-sensitive information, except for the use of a security protocol library. By splitting end-point software into a trusted and untrusted part, the AppCore approach forces end-point software to be aware of the sensitiveness of information. In contrast to previous approaches that limited themselves to automated data flow analysis to split software into trusted and untrusted parts, the AppCore approach used manual analysis of the flow of sensitive information to further simplify the trusted part. This resulted in significant reduction in the complexity of the trusted part compared to purely automated approaches. Since the untrusted part is executed in a separate protection domain, common and well-known vulnerabilities in legacy software such as browsers and WSPs can no longer be used to compromise the trusted part, and by extension, any sensitive information flowing through the trusted part. By reusing (slightly) modified legacy code to operate on non-sensitive data items and by retaining the user, programming and remote interfaces, the AppCore approach also allows for incremental deployment.

WS-FESec provides mechanisms to preserve end-to-end security in open environments; specifically it accounts for scenarios where intermediate services operate at different security levels and where message recipients not know message generation time. These scenarios are common on the Internet, where there are thousands of service providers with varying security and trust properties. WS-FESec can be employed in such scenarios to enforce end-to-end confidentiality and integrity policies in many useful value-added web service compositions such as an electronic prescription system.

7.2. Future Work

This dissertation provides techniques to improve end-to-end security properties of information flow in web-based applications. However, there are several interesting avenues that can be explored to improve the effectiveness of our solutions.

We saw in Chapters 2-4 that AppCore construction involves manual effort in identifying sensitive information, identifying components of software that operate on given sensitive information and simplifying the identified components. This is a time-consuming process, and the problem is exacerbated by the large code bases of existing applications. Partially automating the AppCore construction process can provide great benefits. A starting point was suggested in PrivTrans [41], where programmer provided hints are propagated through the rest of code via data flow analysis to identify all sensitive components. We also need tools that simplify the identified components. Techniques from program specialization [91][100] can be adapted to tackle this problem.

The AppCore approach is complementary to many existing techniques targeted at reducing vulnerabilities in software (Section 6.1.2). Since AppCores are smaller and simpler than legacy code, we expect static analysis techniques to be more effective and run-time protection mechanisms to be cheaper when applied to AppCores. An interesting area of work is applying these techniques to AppCores and gauging the effectiveness of the system as a whole. On the same note, our current implementations of AppCores are

not integrated with Trusted Computing-enabled (TC-enabled) hardware. Integrating AppCores with TC-enabled hardware will provide interesting and useful security properties, e.g., remote attestation.

Our current implementation of ISO-WSP uses a static split of application-level code, through the definition of SFIs. However, the security and performance requirements of clients (in case the server runs on ISO-WSP) can vary and they would like a mechanism to specify a custom SFI that suits their security-performance tradeoffs. To do so, we have to address whether existing applications can be split without *any* developer input. Binary rewriting or byte code rewriting [1] techniques present a promising approach to tackle this problem. Alternatively, developers can write code that is aware of ISO-WSP, e.g., use label to identify variables and functions that *can* be split. With this input, source code translation techniques can be used to add a layer of indirection while accessing these variables and functions. This layer of indirection can be used to control the access of variables or behavior of functions in a manner similar to meta-objects [75] and meta-interfaces [73].

Exploring code generation techniques to convert WS-FESec configurations into code performing cryptographic operations is another avenue for further research. Currently, developers use static configuration files to control the behavior of cryptographic libraries. However, this approach is inflexible with WS-FESec where messages might have completely varying signature or encryption patterns. We need a mechanism to transform higher level input into fault-tolerant configuration code for cryptographic libraries, e.g., *IntG* and *ConfG* information for a web service is translated into a configuration that allows applications to proceed even in the absence of a few non-critical signatures, albeit with a lower trust level in the message.

On a related note, the current WSDL specification, which is used by service providers to describe the interface of a service, does not allow service providers to publish the security requirements or the security properties of web service

implementations. Adding security information to WSDLs, especially information about advanced security properties such as SFIs, requires analyzing use cases and coming up with a standardized and flexible way to specify the security requirements and properties of web service implementations.

REFERENCES

[1] Apache. The ByteCode Engineering Library. http://bcel.sourceforge.net/ Retrieved: 15-May-07

[2] Apache WSS4J. http://ws.apache.org/wss4j/ Retrieved: 30-Sept-06

[3] Axis2 Architecture Guide. http://ws.apache.org/axis2/1_0/ Axis2ArchitectureGuide.html Retrieved: 30-Sept-06

[4] Axis User's Guide. http://ws.apache.org/axis/java/user-guide.html Retrieved: 30-Sept-06

[5] Business Process Execution Language for Web Services version 1.1. ftp://www6. software.ibm.com/software/devel-oper/library/ws-bpel.pdf Retrieved: 15-May-07

[6] comScore. Study Reveals Trends in Online Bill Payment, High-Yield Savings, Customer Loyalty and Satisfaction. http://www.comscore.com/press/ release.asp?press=801 Retrieved: 15-May-07

[7] CyberSource. 7th Annual Online Fraud Report. http://www.cybersource.com/resources/collateral/Resource_Center/whitepapers_and_reports/CYBS_2006_Fraud_Report.pdf Retrieved: 15-May-07

[8] Deutsche Bank Online Banking und Brokerage. Demo Account. https://secure.db24.de/pbc/demokonto/login/1;start.jsp. Retrieved 15-May-07

[9] eHealth Initiative, Executive Summary--Electronic Prescribing: Toward Maximum Value and Rapid Adoption. Washington: eHealth Initiative, 2004. http://www.ehealthinitiative.org/ initiatives/erx/document. aspx?Category=249& Document=269. Retrieved 3-Oct-06

[10] Health Level 7. http://www.hl7.org/ Retrieved: 15-May-07

[11] JavaNCSS. http://www.kclee.de/clemens/java/javancss/ Retrieved: 15-May-07

[12] Jupiter Research. Retail Web Site Performance: Consumer Reaction to a Poor Online Shopping Experience. http://www.akamai.com/4seconds Retrieved 5-May-07

[13] Microsoft. Web Services Specifications. http://msdn2.microsoft.com/en-us/webservices /aa740689.aspx Retrieved: 15-May-07

[14] Mozilla Foundation. Mozilla Module Owners. http://www.mozilla.org/owners.html Retrieved: 5-Jan-06

[15] OASIS Web Services Security (WSS) TC. http://www.oasis-open.org/committees/wss/ Retrieved: 30-Sept-06

[16] OWASP Web Scarab Project. http://www.owasp.org/index.php/Category: OWASP_WebScarab_Project Retrieved: 15-May-07

[17] PeerSec Networks. MatrixSSL - Open Source Embedded SSL. http://www.matrixssl.org/ Retrieved: 15-May-07

[18] RUBiS. http://rubis.objectweb.org/index.html Retrieved: 06-Jun-07

[19] Secunia. IBM WebSphere Application Server 5.x – Vulnerability Report. http://secunia.com/product/2614/? task=advisories Retrieved: 30-Sept-06

[20] Secunia. Microsoft .NET Framework 1.x – Vulnerability Report. http://secunia.com/product/667/?task=advisories Retrieved: 30-Sept-06

[21] Secunia. Vulnerability Report – Microsoft Internet Explorer 6. http://secunia.com/product/11/ Retrieved: 15-May-07

[22] Secunia. Vulnerability Report – Mozilla Firefox 1.x. http://secunia.com/product/4227/ Retrieved: 30-Sept-06

[23] SurfControl. SurfControl Identifies Dangerous New 'Secured Phishing' Attack. http://www.prnewswire.co.uk/cgi/news/release?id=154902. Retrieved: 10-Oct-2006

[24] The Castor Project. http://www.castor.org/ Retrieved: 15-May-07

[25] Trusted Computing Group. https://www.trustedcomputinggroup.org/home Retrieved: 30-Sept-06

[26] U.S. Census Bureau. Quarterly Retail E-Commerce Sales. http://www.census.gov/mrts/www/ecomm.html Retrieved: 15-May-07

[27] VMware, Inc. Browser appliance virtual machine. http://www.vmware.com/vmtn/vm/browserapp.html Retrieved: 15-May-07

[28] Web Services Secure Conversation Language. http://www.w3.org/TR/wscl10/ Retrieved: 30-Sept-06

[29] Web Services Trust Language. ftp://www6.software.ibm.com/software/developer/library/ws-trust.pdf Retrieved: 30-Sept-06

[30] W3C. Web Services Architecture. http://www.w3.org/TR/ws-arch/ Retrieved: 30-Sept-06

[31] W3C. Web Services Architecture: W3C Working Draft 14 November 2002. http://www.w3.org/TR/2002/WD-ws-arch-20021114/ Retrieved: 30-Sept-06

[32] W3C. XML Path Language. http://www.w3.org/TR/xpath Retrieved: 5-May-07

[33] XMLBench Document Model Benchmark. http://www.sosnoski.com/opensrc/xmlbench/ Retrieved: 15-May-07

[34] J. Bambenek, SANS Institute. BHO scanning tool and New Scam Targets Bank Customers. http://isc.sans.org/diary.php?date=2004-06-29. Retrieved: 2-Nov-06

[35] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. 2003. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA, October 19 - 22, 2003). SOSP '03. ACM Press, New York, NY, 164-177.

[36] E. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, and D. Zovi. Intrusion Detection: Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *CCS* 2003.

[37] V. Basili and D. Hutchens. An Empirical Study of a Complexity Family. In *IEEE Transactions on Software Engineering*, Volume 9, No. 6, November 1983, pp. 664-672.

[38] A. Bengtsson, L. Westerdahl, Secure Choreography of Cooperating Web Services, In *Proc. Third European Conference on Web Services*, pp. 152-159, 2005.

[39] S. Bhatkar, R. Sekar and D. C. DuVarney, Efficient Techniques for Comprehensive Protection from Memory Error Exploits, In *14th USENIX Security Symposium*, Baltimore, MD, August 2005.

[40] Boyd, S., and Keromytis, A., SQLrand: Preventing SQL injection attacks. In *Proc. of the 2nd Applied Cryptography and Network Security (ACNS) Conference*. Volume 3089 of LNCS, Springer-Verlag (2004) pp. 292--304.

[41] D. Brumley, D. X. Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proc. USENIX Security Symposium,* San Diego, USA. Aug 9-13, 2004.

[42] J. Burns. Cross Site Reference Forgery: An introduction to a common web application weakness. http://isecpartners.com/documents/XSRF_Paper.pdf Retrieved: 15-May-07

[43] Carminati, B. Ferrari, E. Hung, P.C.K, Web Service Composition: A Security Perspective, In *Proc. WIRI 2005*, pp. 248-253, 2005.

[44] M. Castro, M. Costa and T. Harris, Securing Software by Enforcing Data-flow Integrity, In *OSDI 2006*, Seattle, Nov 2006.

[45] D.W. Chadwick and D. Mundy, Policy Based Electronic Transmission of Prescriptions In *Proc. of 4th IEEE Intl. Workshop on Policies for Distributed Systems and Networks*, Lake Como, Italy, pp 197-206, June 2003.

[46] G. Chafle, S. Chandra, V. Mann and M.G. Nanda: Orchestrating Composite Web Services under Data Flow Constraints. In *Proc. ICWS 2005*. pp. 211-218, July 2005.

[47] A. Charfi and M. Mezini, Using aspects for security engineering of Web service compositions, In *Proc. ICWS 2005*. pp. 59- 66, July 2005.

[48] P. M. Chen and B.D. Noble, When Virtual is Better Than Real, In *Eighth Workshop on Hot Topics in Operating Systems,* May 2001, Elmau, Germany.

[49] M. Clarke, G.S. Blair, G. Coulson and N. Parlavantzas, An Efficient Component Model for the Construction of Adaptive Middleware, In *Proc. Middleware 2001*, pp. 160-178, 2001.

[50] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, N-Variant Systems: A Secretless Framework for Security through Diversity, In *15th USENIX Security Symposium*, Vancouver, BC, August 2006.

[51] R.S. Cox, S.D. Gribble, H.M. Levy, and J.G. Hansen, A Safety-Oriented Platform for Web Applications, In *IEEE Symposium on Security & Privacy*, pp. 350-364, 2006.

[52] D. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5), pp 236--242, 1976.

[53] T. Dierks and E. Rescorla, The Transport Layer Security (TLS) Protocol. Version 1.1, *RFC 4346*, April 2006.

[54] Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., and Morris, R. 2005. Labels and event processes in the asbestos operating system. In *Proc. of the 20th ACM SOSP*, Brighton, United Kingdom, October 23 - 26, 2005.

[55] M. Eichberg and M. Mezini, Alice: Modularization of Middleware Using Aspect-Oriented Programming, In *Proc. Software Engineering and Middleware*, pp. 47-63. 2004.

[56] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system specific programmer-written compiler extensions. In *4th USENIX OSDI*. San Diego, Oct. 2000.

[57] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *18th SOSP*. Banff, Canada, Oct. 2001.

[58] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, XFI: Software Guards for System Address Spaces, In *OSDI 2006*, Seattle, Nov 2006.

[59] Evans, D. and Larochelle, D. 2002. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Softw.* 19, 1 (Jan. 2002), pp. 42-51.

[60] A. O. Freier, P. Karlton and P.C. Kocher, The SSL Protocol Version 3.0, *Transport Layer Security Working Group, Internet-Draft*. http://wp.netscape.com/eng/ssl3/draft302.txt

[61] Gaffney, J., Program Control Complexity and Productivity. In *Proceedings of the IEEE Workshop on Quantitative Software Models,* pg 179, October, 1979.

[62] V. Ganapathy, A. Balakrishnan, M.M. Swift and S. Jha, Microdrivers: A New Architecture for Device Drivers, In *Proc. of HotOS XI*, pp 85—90, San Diego, California; May 7-9, 2007

[63] V. Ganapathy, T. Jaeger, and S. Jha, Retrofitting Legacy Code for Authorization Policy Enforcement, In Proc. of the S&P 2006, pp 214--229, Oakland, USA, May 21-24, 2006.

[64] T. Garfinkel. B. Pfaff, J. Chow, M. Rosenblum and D. Boneh, Terra: A virtual machine-based platform for trusted computing. In *Proc. of the 19th SOSP*, October 2003.

[65] Halfond, W. G. and Orso, A., Preventing SQL injection attacks using AMNESIA, In *Proc. of the 28th international Conference on Software Engineering*. ICSE '06. ACM Press, New York, NY, 795-798.

[66] H. Härtig. Security architectures revisited. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[67] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J.Wolter. The performance of μ-kernel-based systems. In *Proc. 16th ACM Symposium on Operating System Principles*, pp 66–77, Oct. 1997.

[68] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert and M. Peter. The Nizza Secure-System Architecture. In *IEEE CollaborateCom 2005.* San Jose, USA. Dec 2005.

[69] C. Helmuth, A. Warg, and N. Feske. Mikro-SINA—Hands-on Experiences with the Nizza Security Architecture. In *Proceedings of the D.A.C.H Security 2005*, Darmstadt, Germany, March 2005.

[70] B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. Integration of SELinux and security-typed languages. In Proc. of the 2007 Security-Enhanced Linux Workshop, March 2007.

[71] L.J. Hoffman, K. Lawson-Jenkins and J. Blum, Trust beyond security: an expanded trust model – In *Comm. of the ACM*, Volume 49, Number 7 (2006), pp. 94-101.

[72] Hohmuth, M., M. Peter, H. Härtig, and J. Shapiro. Reducing TCB size by using untrusted components – small kernels versus virtual machine monitors, in *Proc. of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, 2004.

[73] M. Howard, J. Pincus, and J.M. Wing, Measuring Relative Attack Surfaces," Chapter 8, in *Computer Security in the 21st Century*, D.T. Lee, S.P. Shieh, and J.D. Tygar, editors, Springer, March 2005, pp. 109-137.

[74] Kiczales, G., Beyond the Black Box: Open Implementation. *IEEE Softw.* 13, 1 (Jan. 1996), 8-11, 1996.

[75] Kiczales, G., des Rivières, J., and Bobrow, D. G., *The Art of Metaobject Protocol*. MIT Press, 1991.

[76] D. Kilpatrick, Privman: A Library for Partitioning Applications. In *USENIX Annual Technical Conference, FREENIX Track 2003*, pp 273-284. San Antonio USA, July 2003.

[77] Kim, S. M. and Rosu, M. C., A survey of public web services. In *Proc. of 13th WWW Conf. on Alternate Track Papers & Posters*, pp. 312-313, May 2004.

[78] J. Kong, K. Schwan and P. Widener, Protected Data Paths: Delivering Sensitive Data via Untrusted Proxies, In *Proc. 2006 Intl. Conf. on Privacy, Security and Trust*, Ontario, Oct. 2006.

[79] A.H.F. Laender, B.A. Ribeiro-Neto, A.S. da Silva and J.S. Teixeira, A brief survey of web data extraction tools, In *SIGMOD Rec.*, 32(2), pp 84-93, 2002.

[80] Lhee, K. and Chapin, S. J. 2003. Buffer overflow and format string overflow vulnerabilities. *Softw. Pract. Exper.* 33, 5 (Apr. 2003), 423-460.

[81] K.-J. Lin, H. Lu, T. Yu, and C.-E. Tai , A reputation and trust management broker framework for web applications. In *IEEE EEE 2005*.

[82] J. Liu, W. Huang, B. Abali and D. K. Panda. High Performance VMM-Bypass I/O in Virtual Machines. In *USENIX ATC* 2006, Boston, MA, May 2006.

[83] E.M. Maximilien and M.P. Singh, Reputation and endorsement for web services. *SIGecom Exch.* 3, 1 (Dec. 2001), 24-31.

[84] E.M. Maximilien and M.P. Singh, Toward autonomic web services trust and selection, In *2nd ICSOC*, 2004.

[85] T.J. McCabe, A Complexity Measure, *IEEE Transactions on Software Engineering*, SE-2 No. 4, pp. 308-320, Dec. 1976.

[86] A.C. Myers and B. Liskov. Protecting privacy using the decentralized label model. In *ACM Transactions on Computer Systems*, 9(4):410–442, October 2000.

[87] Necula, G. C., McPeak, S., and Weimer, W. 2002. CCured: type-safe retrofitting of legacy code. In *Proc. POPL '02*. ACM Press, New York, NY, pp. 128-139, 2002.

[88] N. Nagappan, T. Ball and A. Zeller, Mining Metrics to Predict Component Failures, In *ICSE 2006*, Shanghai, Nov. 2006

[89] S. Perera, C. Herath, J. Ekanayake, E. Chinthaka, A. Ranabahu, D. Jayasinghe, S. Weerawarana, and G. Daniels, Axis2, Middleware for Next Generation Web Services, In *Proc. ICWS 2006*, pp. 833-840, Sept. 2006.

[90] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, Washington D.C, Aug. 2003.

[91] Pu, C., Autrey, T., Black, A., Consel, C., Cowan, C., Inouye, J., Kethana, L., Walpole, J., and Zhang, K., Optimistic incremental specialization: streamlining a commercial operating system, In *Proc. 15th ACM SOSP,* pp. 314-321, 1995.

[92] J. Rafail, Cross-Site Scripting Vulnerabilities. http://www.cert.org/archive/pdf/cross_site_scripting.pdf Retrieved 05-May-07

[93] C. Reis, J.Dunagan, H.J. Wang, O. Dubrovsky, and S. Esmeir, BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML, In *OSDI 2006*, Seattle, WA

[94] E. Rescorla, HTTP Over TLS, *RFC 2818*, May 2000.

[95] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. Mitchell, Stronger Password Authentication Using Browser Extensions. In *14th Usenix Security*, Baltimore, 2005.

[96] A. Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. In *Journal on Selected Areas in Communications*, 21(1):5-19, January 2003.

[97] R. Sailer, X. Zhang, T. Jaeger, and L. V. Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of Thirteenth USENIX Security Symposium*, pp 223--238, August 2004.

[98] J.H. Saltzer and M.D. Schroeder, The Protection of Information in Computer Systems, In *Proc. of the IEEE*, Vol.63, No.9, Sept. 1975, pp.1278-1308.

[99] B. Schneier. Software Complexity and Security. *Crypto-Gram Newsletter.* March 2000. http://www.schneier.com /crypto-gram-0003.html

[100] Schultz, U. P., Lawall, J. L., and Consel, C. 2003. Automatic program specialization for Java. *ACM Trans. Program. Lang. Syst.* 25, 4 (Jul. 2003), 452-499.

[101] I. Sedukhin, End to End Security for Web Services and Services Oriented Architectures. *CA White paper*, Mar. 2003

[102] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A Fast Capability System. In *Proc. 17th ACM SOSP*. Charleston, SC, USA. Dec. 1999.

[103] V. Y. Shen, T. Yu, S. M. Thebaut, and L. R. Paulsen, Identifying Error-prone Software --- An Empirical Study, In *IEEE TOSE*, Vol. SE-11, pp. 317--323, April 1985.

[104] L. Singaravelu, B.Kauer, A. Boettcher, H. Härtig, C. Pu, G. Jung, C. Weinhold, Enforcing Configurable Trust in Client-side Software Stacks by Splitting Information Flow, *GIT-CERCS-TR-07-11*, May 2007.

[105] L. Singaravelu and C. Pu, Fine-Grain, End-to-End Security for Web Service Compositions, In *SCC 2007*, Salt Lake City, July 2007.

[106] L. Singaravelu, C. Pu, H. Härtig, C. Helmuth, Reducing TCB Complexity for Security-Sensitive Applications:Three Case Studies, In *Proc. First Eurosys*, Leuven, Belgium, April 2006.

[107] L. Singaravelu, J. Wei, and C. Pu, A Secure Middleware Architecture for Web Service Platforms, *GIT-CERCS-TR-07-14*, May 2007.

[108] Stiegler, M., Karp, A. H., Yee, K., Close, T., and Miller, M. S., Polaris: virus-safe computing for Windows XP, In *CACM*, 49(9), pp 83-88, 2006

[109] R. Ta-Min, L. Litty and D. Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proc. of the 7th USENIX OSDI*, pp. 279-292. Nov. 2006.

[110] J. D. Tygar and A. Whitten. WWW electronic commerce and Java Trojan horses. In *Proc. of the 2nd USENIX Workshop on Electronic Commerce*, Nov. 1996, pp. 243-250.

[111] Venkatasubramanian, N., Deshpande, M., Mohapatra, S., Gutierrez-Nolasco, S., and Wickramasuriya, J., Design and Implementation of a Composable Reflective Middleware Framework. In *ICDCS 2001*. April, 2001.

[112] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, 2000.

[113] D. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/ Retrieved 15-May-07

[114] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F., HYDRA: the kernel of a multiprocessor operating system. *CACM* 17(6), Jun. 1974, pp. 337-345.

[115] S.J.H. Yang, J.S.F. Hsieh, B.C.W. Lan and J-Y. Chung, Composition and Evaluation of Trustworthy Web Services, In *Proc. of the IEEE EEE05 Intl. Workshop on Business services networks*, pp. 5-14, 2005.

[116] S. Yoshihama, T. Ebringer, M. Nakamura, S. Munetoh and H. Maruyama, WS-Attestation: Efficient and Fine-Grained Remote Attestation on Web Services, In *Proc.* ICWS'05, pp. 743-750, 2005.

[117] Zhang, J.; Zhang, L.-J.; Chung, J.-Y., WS-trustworthy: a framework for Web services centered trustworthy computing, In *SCC 2004*.

[118] C. Zhang, H.-A. Jacobsen, Refactoring Middleware with Aspects. In *IEEE Trans. on Parallel and Distributed Systems*. 14(11), p. 1058-1073, 2003.

[119] C. Zhang, H.-A. Jacobsen, Resolving Feature Convolution with Horizontal Decomposition in Middleware. In *Proc. OOPSLA 2004*. p. 188-205. Vancouver, BC, 2004.

[120] Y. Zhang, S. Egelman, L. F. Cranor, and J. Hong, Phinding Phish: Evaluating Anti-Phishing Tools. In *14th NDSS*, San Diego, CA, 2007.