A Secure Information Flow Architecture for Web Services

Lenin Singaravelu, Jinpeng Wei, Calton Pu
College of Computing, Georgia Institute of Technology
lenin@acm.org, {weijp, calton}@cc.gatech.edu

Abstract

Current web service platforms (WSPs) often perform all web services-related processing, including security-sensitive information handling, in the same protection domain. Consequently, the entire WSP may have access to security-sensitive information such as credit card numbers, forcing us to trust a large and complex piece of software. To address this problem, we propose ISO-WSP, a new information flow architecture that decomposes current WSPs into two parts executing in separate protection domains: (1) a small trusted T-WSP to handle security-sensitive data, and (2) a large, legacy untrusted U-WSP that provides the normal WSP functionality, but uses the T-WSP for security-sensitive data handling. By restricting security-sensitive data access to T-WSP, ISO-WSP reduces the software complexity of trusted code, thereby improving the testability of ISO-WSP. Using a prototype implementation based on the Apache Axis2 WSP, we show that ISO-WSP reduces software complexity of trusted components by a factor of five, while incurring a modest performance overhead of few milliseconds per request.

1. Introduction

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 paper 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).

Current WSPs often perform all types of processing, including security-sensitive information handling, in the same protection domain (e.g., a single process). Therefore, all components of the WSP may have direct or indirect access to sensitive data, violating the Principle of Least Privilege [24]. Additionally, the large size and complexity of WSPs hinders their testability, resulting in systems with multiple security vulnerabilities [6], [7]. Therefore, attackers can compromise the flow of sensitive information by exploiting vulnerabilities in the large WSPs despite the use of security protocols such as WS-Security or SSL.

We propose ISO-WSP, an information flow architecture for web services to address the problem of protecting security-sensitive information flow against potential vulnerabilities inherent in large and complex software packages such as WSPs. Applying the App-Core approach [25], ISO-WSP decomposes current WSPs into two parts: (1) a small, functionally-limited, trusted T-WSP and (2) a large, functionally-rich, untrusted U-WSP. The T-WSP consists of components of a WSP that require access to security-sensitive information and the U-WSP contains the rest of the legacy WSP. The T-WSP and U-WSP are executed in separate protection domains, with the U-WSP invoking the T-WSP when it has to operate on security-sensitive data. By restricting security-sensitive data access to the small T-WSP, ISO-WSP eliminates the need to trust the U-WSP. Since the T-WSP is expected to be considerably smaller than the U-WSP, we improve the testability of the ISO-WSP.

We demonstrate the feasibility of our approach by implementing and evaluating an ISO-WSP based on the Axis2 WSP. We show that ISO-WSP results in a five-fold reduction in the size and complexity of the software that has access to sensitive data, while imposing a moderate overhead of few milliseconds per request.

The organization of the rest of the paper is as follows: Section 2 motivates the paper by first presenting the design of WSPs in detail and then discussing the security problems in current WSPs. Section 3 discusses the design of ISO-WSP. Section 4 presents a prototype implementation of ISO-WSP based on the Axis2 WSP and evaluates the resulting system. Section 5 discusses some of the open issues. Section 6 discusses the related work and Section 7 concludes the paper.

2. Motivation

Before we discuss the design of web service platforms (WSPs) and security problems associated with them, we define *security-sensitive* (or) *sensitive* data item as any data item that the end-user or the business logic imposes *confidentiality* or *integrity* requirements, e.g., payment information, patient health information. Sensitive data items also include data items that may not be transmitted over the network, e.g., private keys used by the WSP.

2.1 Design of Web Service Platforms

W3C's web services architecture specification [9] specifies the basic framework for WSPs. WSPs such as Apache Axis2 and Microsoft .NET implement this framework. According to specification, 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 support for uniform naming WS-Addressing, WS-Security, WS-ReliableMessaging, and WS-Eventing (over 35 extensions are listed in [4]). WSPs may also possess an extension architecture to seamlessly integrate new and upcoming WS-* specifications or to provide support for custom extensions for logging or load-balancing.

The Apache Axis2 WSP [21] 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. Figure 1 briefly illustrates the SOAP processing chain in the Axis2 WSP. Axis2 provides support for developing, deploying, managing and invoking web services. In addition to implementing the basic W3C standards such as SOAP over HTTP and a framework for executing business logic, Axis2 also supports an extension architecture. This extension architecture allows web

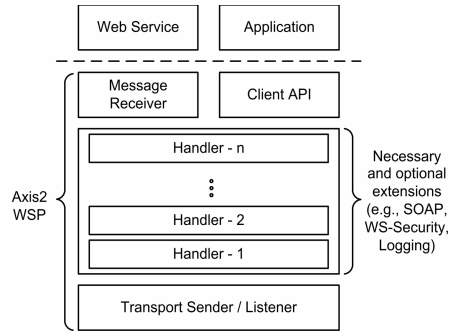

Figure 1. **SOAP Processing Model in Apache Axis2 [2].**

service developers to plug in added functionality using well-known pre-existing libraries (e.g., WSS4J for WS-Security) or customized code for logging or load balancing. Furthermore, Axis2 allows web service developers to change the sequence of processing by adding, removing or reordering the handlers.

2.2 Security Problems in WSPs

WS-Security specification [5] can be used 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 [6], [7], or the business logic and support software (e.g., databases). Similarly, on the client side, attackers can leverage vulnerabilities in client-side WSPs or client applications like the browser. We focus on securing WSPs in this paper. Securing operating systems and applications such as the browser is outside the scope of this paper.

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., transport protocol implementation and 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, we might be forced to carry out WS-Security processing before WS-Addressing processing so as to verify the signatures in the WS-Addressing headers. This allows the WS-Addressing handler access to sensitive message con-

```
//Handlers can access Service/Msg Context
param=ctx0.getAxisConfiguration()
       .getParameter("OutflowSecurity");
ome = param.getParameterElement();
iter = ome.getGirstElement().
                       getChildElements();

while (iter.hasNext()){
  attr = (OMElement) itor.next();
  //Look for Encryption Key
  if ("encryptionUser".equals
                (attr.getLocalName())){
    //replace with weak key
    attr.setText("weak_key_identifier");
  }
}
```

Figure 2. 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.

tents even though it does not need access. Even if we assume that WS-Security processing is carried out as late as possible, malicious or buggy handlers can still compromise security-sensitive data or WS-Security processing because they execute in the same protection domain. Figure 2 illustrates one such scenario, where a malicious handler uses shared data structures to subvert WS-Security processing.

Secondly, a WSP is a complex piece of software. As seen earlier (§ 2.1), 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 feasible to exhaustively 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.

3. ISO-WSP

Based on the security problems discussed in Section 2.2, we see that an effective solution should

R1. Prevent WSP components that do not need access to sensitive data from accessing them.

R2. Reduce the size and complexity of WSP components that have access to sensitive data.

R3. Preserve functional compatibility with existing WSPs, and support existing business logic code with few or no modifications. Due to space constraints, we will not be discussing this requirement in detail in this paper.

Our solution, called ISO-WSP, attempts to address the security problems while meeting the requirements listed above. First, we split the functionality (or code base) of existing WSPs into a small trusted T-WSP and a legacy, untrusted U-WSP. Next, we introduce a new trusted component called a Message Splicer, which limits the flow of plain-text sensitive information to the T-WSP. Taken together, these steps address requirements R1 and R2 (see Section 4.2 for details). Additionally, by reusing the U-WSP to operate on non-sensitive or protected data, ISO-WSP partially satisfies requirement R3.

3.1 Splitting Legacy WSPs

Certain components of a legacy WSP require access to plain-text security-sensitive data. Since these components have to be allowed to read or modify sensitive data, they have to be trusted by both the end-user and the business logic. We call such components *Trusted Components* and these components taken together are referred to as the T-WSP. The rest of the components do not have to be trusted and are henceforth referred to as untrusted components (and collectively referred to as the U-WSP).

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 [9], specifications for the WS-* extensions such as WS-ReliableMessaging, WS-Addressing [4], and the Axis2 web services architecture guide [2]. Based on our analysis, we found that a component has to be trusted for one of two reasons:

- **The component requires 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 or to verify the protection on them.
- **The component controls the behavior of other components with direct access (Indirect Access).** This includes other security specifications such as WS-SecureConversation. These components have to be trusted because they control the behavior of the first type of components, e.g., WS-SecureConversation is used to determine the keys used by WS-Security to encrypt or sign sensitive data items.

The rest of the components of the WSP, including the WS-* extensions, are treated as untrusted components, either because they are agnostic to message contents, or because they only indirectly depend on sensitive data. An example of the first case is the transport layer protocol implementation or SOAP implementation. Examples of the second case are WS-* extensions such as WS-Addressing which relies on the encryption keys employed by WS-Security to protect the integrity and in some cases, the confidentiality of SOAP message headers.

3.2 Message Splicer

In addition to limiting security-relevant *processing* to trusted components, we have to limit the flow of sensitive *information* to trusted components. To achieve this, we introduce a trusted component called the *Message Splicer*. The function of the Message Splicer is to intercept plain-text messages and replace sensitive data with non-sensitive data or vice versa depending on the direction of the message. For an incoming message with sensitive data, the Message Splicer replaces sensitive data items with dummy data items and a token that uniquely identifies an instance of a sensitive data item. The resulting message is passed on to the untrusted portion for further processing. The actual security-sensitive data along with the original message is passed on to the trusted portion for further processing. For an outgoing message, the steps occur in the reverse order.

We replace security-sensitive data items with dummy data items to minimize modifications to the message format, and therefore to application level code. An astute reader will point out that by adding the unique token, we are in fact modifying the message format and hence will have to modify application level code. We note that this modification is very structured and in the case of application-level code written in Java for Axis2, we have been able to incorporate most of these modifications with automated code generation. Due to space limitations, we are unable to discuss this in detail.

3.3 Architecture of ISO-WSP

Figure 3 provides an overview of the architecture of ISO-WSP. As explained previously (§ 3.1), 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.

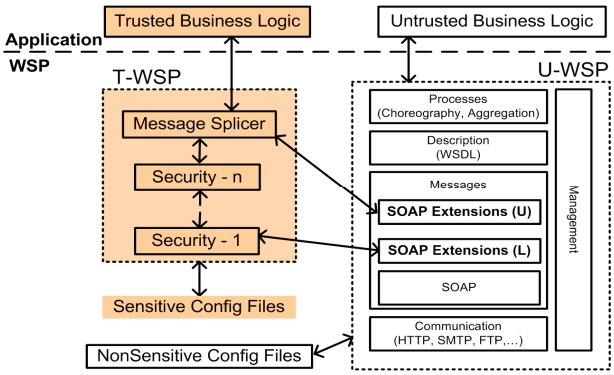

Figure 3. Architecture of ISO-WSP. Shaded boxes represent Trusted Components. Trusted Components execute in a separate protection domain.

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 3.2, addresses how we secure the flow of sensitive information in the ISO-WSP.

Since application-level code also has access to sensitive data, we also have to split application-level code into trusted and untrusted portions. As briefly mentioned in Section 3.2, we have developed code-generators to automate this process as much as possible. More detail is presented in a technical report [26].

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 architecture is that external entities do not perceive any changes to the functional interface of the WSP. ISO-WSP exports the same external interface as legacy WSPs. The U-WSP provides support for the interface by implementing transport-layer protocols such as HTTP over TCP/IP.

4. Implementation and Evaluation

4.1 Implementation

We implemented an ISO-WSP prototype based on the Apache Axis2 platform. The T-WSP consisted of a WS-Security implementation (WSS4J [1]) 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. We also had to modify the WS-Security implementation to support RMI invocation. 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, 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 process. 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. Since we use the RMI protocol to communicate between the T-WSP and U-WSP, each of them can reside on separate virtual or even physical machines, thereby providing stronger isolation properties.

Our implementation of the Message Splicer accepts information about sensitive data items 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.

Payment Processing Service: We implemented a simple payment processing service 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 gets the complete message with both security-sensitive and security-insensitive data. It then performs book-keeping operations such as logging, calls the charge card function to generate a confirmation number, performs some more book-keeping operations before returning the data to the WSP.

In the implementation on top of ISO-WSP, the business logic is split into two parts: one that handles credit card information (trusted) and another that handles order information (untrusted). The Message Splicer of the ISO-WSP replaces security-sensitive information in an incoming message with dummy data items and a unique token before passing it on to the U-WSP and the untrusted business logic. Security-sensitive contents are passed on to the trusted business logic. The untrusted business logic performs the necessary book-keeping operation before invoking T-WSP – with the unique token embedded by the Message Splicer as an argument – to operate on the card. The return value is passed on to the U-WSP.

4.2 Security Properties of ISO-WSP

4.2.1 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, in ISO-WSP, 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.

The Message Splicer is the key component that prevents direct access of sensitive information by the U-WSP by splitting and merging flow of information 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 (*Transport Protocols, SOAP and SOAP Extensions (L)* in Figure 3): 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 (*SOAP Extensions (U) and untrusted business logic* in Figure 3): 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.

4.2.1 Software Complexity Reductions in ISO-WSP. Our second motivating factor for constructing an ISO-WSP was the increased complexity of WSPs. Since access to sensitive information is limited to the T-WSP, we only have to trust the T-WSP to protect the flow of sensitive information. Table 1 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 [18]. Empirical studies have shown that both measures of software complexity correlate with number of bugs in code [20]. 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 [22].

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, including an extension architecture, ISO-WSP attempts to gain the best of both worlds.

4.3 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 RMI overhead, we use a simple echo application with the client and server running on the same machine. We found that round trip times increase 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.

Table 1. 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. Extensions include implementations of WS-Coordination, WS-ResourceFramework, WS-Addressing, amongst others. All numbers were generated using the JavaNCSS tool [3].

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

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* [10]. We found that combined serializing and deserializing costs are of the order of few milliseconds. For small to medium sized XML files (~10 kilobytes), the combined cost is less than 4 ms. For the largest XML file in the dataset (36 KB), the combined cost was about 14 ms.

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.

We use the payment processing web service described earlier (§ 4.1) 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. Of this 7.6 ms overhead, 5.2 ms can be attributed to the additional processing steps in the ISO-WSP architecture: serializing and deserializing SOAP messages and RMI costs. The rest of the overhead (~2.4 ms) was incurred 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. 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. While this might seem excessive, typical web service invocation time ranges from 0.5 seconds to a few seconds [16]. 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.

5. Open Issues

While the ISO-WSP architecture improves security properties of the web service, there are a few remaining key challenges that need to be addressed.

5.1 Modifications to Application Level Code

We saw in Sections 3.2 and 3.3 that ISO-WSP changes the format of the message and the interaction patterns between the WSP and the application-level code. This required modifications to application level code and can impose an undue burden on the web service developer and hinder the adoption of ISO-WSP. We argue that these modifications are very structured: e.g., security sensitive data items now contain a single additional data object. We have been able to use code-generation techniques and leverage object inheritance facilities present in the Java programming language to limit the number of modifications to application level code to the order of few tens of lines of code [26].

We rely on programmer annotations to split application-level code. For applications with complex data flows, we can also leverage considerable research on splitting application level code based on a few simple programmer annotations using information flow analysis tools such as JIF/JFlow [19].

5.2 Denial of Service Attacks

The current ISO-WSP design does not address Denial of Service (DoS) attacks. For example, the U-WSP can carry out DoS attacks by either corrupt messages or choosing not to invoke the T-WSP for security processing. One should note that existing WSPs are also susceptible to similar types of attacks. Moreover, these attacks do not compromise the confidentiality or integrity of security-sensitive information.

ISO-WSP can be enhanced with Trusted Computing hardware [8] and application level support for Trusted Computing, e.g, Integrity Measurement Architecture [23], to provide additional, desirable security properties such as integrity of software stack at load time and remote attestation.

6. Related Work

Previous efforts have addressed refactoring existing systems software [11],[15] and application-level software [25] to reduce the size and complexity of software that has access to sensitive data. Our work can be considered as an application of such techniques to web services middleware. In contrast to previous approaches, by replacing sensitive data items with dummy data items in the middleware, we also enable existing application-level software to run with minimal modifications.

There is also considerable research into refactoring and customizing middleware to modularize and simplify them. Zhang et al. [32], [1] and Eichberg et al. [13] use Aspect Oriented Programming techniques to modularize middleware and then, customize it according to the needs of applications. OpenCOM [12] and CompOSE/Q [28], 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 [32], [1] are applicable to the construction of ISO-WSP.

Trusted Computing initiatives [8] attempt to reassure remote entities about the integrity of the software stack handling sensitive data. The Integrity Measurement architecture [23] aims to extend integrity measurement to include configuration files. WS-Attestation [31] and Trusted Web Services [27] attempt to incorporate Trusted Computing principles into the web services stack. However, an attacker can still compromise the integrity of the software stack by exploiting run time vulnerabilities or by loading malicious extensions at runtime. By reducing the size and complexity of the trusted part of the system, ISO-WSP enables the use of exhaustive testing or static analysis techniques, thereby reducing the number of run-time vulnerabilities.

The use of static analysis techniques [14], [29] to identify bugs in code is gaining popularity. For strongly-typed languages such as Java, one can go even further and apply language-based information flow protection techniques [22]. However, the applicability of such techniques to a large code-base is a subject of open research and we expect such techniques to be more amenable to smaller code bases, such as that of the T-WSP.

Lastly, tokens used in ISO-WSP by untrusted applications to operate on security-sensitive data can be viewed as capabilities [30]. The ISO-WSP architecture can be considered as retrofitting a capability-based architecture in to WSPs. Protected Data Paths (PDP) [17] 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.

7. Conclusion

In this paper, we presented ISO-WSP, a secure information flow architecture to counter the problem of large and complex WSPs. ISO-WSP consisted of two parts: a small, trusted T-WSP that required access to security-sensitive information and a U-WSP that retained the features of legacy WSPs. By limiting the flow of sensitive information to the T-WSP, we ensured that the U-WSP does not have to be trusted, thereby improving the testability of the ISO-WSP.

We demonstrated the feasibility of our approach by building an ISO-WSP based on the Apache Axis2 WSP. By splitting and limiting the flow of sensitive information, we showed that ISO-WSP architecture reduces the complexity of the trusted portion of a WSP by a factor of 5, while imposing manageable overhead of a few milliseconds per request. We also briefly discussed techniques to minimize the cost of porting legacy applications to run on the ISO-WSP.

8. References

[1] Apache WSS4J. http://ws.apache.org/wss4j/

[2] Axis2 Architecture Guide. http://ws.apache.org/axis2/1_0/Axis2ArchitectureGuide.html

[3] JavaNCSS. http://www.kclee.de/clemens/java/javancss/

[4] Microsoft. Web Services Specifications. http://msdn2.microsoft.com/en-us/webservices/aa740689.aspx

[5] OASIS Web Services Security (WSS) TC. http://www.oasis-open.org/committees/wss/

[6] Secunia. IBM WebSphere Application Server 5.x – Vulnerability Report. http://secunia.com/product/2614/

[7] Secunia. Microsoft .NET Framework 1.x – Vulnerability Report. http://secunia.com/product/667/?task=advisories

[8] Trusted Computing Group. https://www.trustedcomputinggroup.org/home

[9] W3C. Web Services Architecture. http://www.w3.org/TR/ws-arch

[10] XMLBench Document Model Benchmark. http://www.sosnoski.com/opensrc/xmlbench/

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

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

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

[14] 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 ACM SOSP*, Oct. 2001.

[15] 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.

[16] 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.

[17] 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.

[18] T.J. McCabe, A Complexity Measure, *IEEE Transactions on Software Engineering*, SE-2 No. 4, 1976.

[19] A. Myers. Jflow: Practical mostly-static information flow control. In *26th POPL,* San Antonio, Jan 1999.

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

[21] S. Perera et al. Axis2, Middleware for Next Generation Web Services, In *Proc. ICWS 2006*, pp. 833-840, Sept. 2006.

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

[23] R. Sailer, X. Zhang, T. Jaeger, and L. V. Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proc. of 13th USENIX Security*, 2004.

[24] 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.

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

[26] L. Singaravelu, J. Wei and C. Pu, A Secure Middleware Architecture for Web Services, *CERCS Technical Report, Georgia Tech*. GIT-CERCS-07-14, 2007.

[27] Z. Song, et al., Trusted Web Service, In *2nd Workshop on Advances in Trusted Computing*, Tokyo, Japan, 2006.

[28] N. Venkatasubramanian, et al., Design and Implementation of a Composable Reflective Middleware Framework. In *ICDCS 2001*. April, 2001.

[29] D. Wagner et al., A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. of ISOC NDSS*, 2000.

[30] 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.

[31] S. Yoshihama et al., WS-Attestation: Efficient and Fine-Grained Remote Attestation on Web Services, In *Proc. ICWS'05*, pp. 743-750, 2005.

[32] C. Zhang, H.-A. Jacobsen. Refactoring Middleware with Aspects. In *IEEE TPDS*. 14(11), p. 1058-1073, 2003.

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