DEVELOPING AND DEPLOYING SECURE WEB APPLICATIONS

A whitepaper from Watchfire

TABLE OF CONTENTS

200 West Street | Waltham, Massachusetts | USA 02451 | T 1-781-810-1450 | F 1-781-890-2087
1 Hines Road, Suite 200 | Kanata, Ontario | Canada K2K 3C7 | T 1-613-599-3888 | F 1-613-599-4661
211 Piccadilly |London, United Kingdom | W1J 9HF | T +44 (0) 20 7917 2962 | F +44 (0) 20 7917 2963

www.watchfire.com

1.0 OVERVIEW: THE LIFECYCLE OF A WEB APPLICATION

Web applications are complex entities. Technically speaking, an application is "a program designed to perform a specific function directly for the user or for another application program". Web applications include code that resides on the Web servers, application servers, databases, and backend systems of an organization. In short, they are any application that will be accessed in some way, shape, or form through the Web.

Securing a Web application is difficult, not only because of the cross-departmental coordination involved, but because most security tools are not designed to address the Web application as a whole, including how the different pieces of the application interact with each other. The potential for a security breach exists in each layer of a Web application. Traditional security solutions, such as access control or intrusion detection/prevention systems, are specialized to protect different layers of the Internet infrastructure, and are usually not designed to handle HTTP and HTML attacks. While these tools are useful for their specific functions, they do not address all of the issues that Web applications present. Most concerning, using these tools can give administrators a false sense of confidence if they don't know the other vulnerabilities exist.

The security of each layer and the interaction of the layers are not perfect. This opens the door to attackers trying to manipulate the application logic to achieve their needs. Sending "illegal" input through the normal interaction with the application can leverage vulnerabilities in any of the layers. Manipulating the protocols and languages used for interacting with the site (usually HTTP & HTML) to values that are not expected by the application can easily be done by changing fixed values, achieving buffer overflows, and subverting the flow of the application.

One needs to look at the application life cycle to fully understand the level of risk inherent in today's ebusinesses. The web application development process is a multi-stage process consisting of design, coding, testing and deployment. A secure web application can only be created by securing all of these stages. Ignoring the security aspects in any of the stages will compromise the security of the whole application and will be extremely hard to correct at later stages. Moreover, creation of the application is not a process that is performed once, but rather a cycle in which the application is initially created and then goes into continuous cycles of changes, improvements and updates. This cycle causes security flaws initiated during early versions of the product to persist through all the later versions as well.

The application typically goes through four stages:

1. The *design stage* where the business logic is defined including detailed design of the specific components of the application and the interaction with other applications.
2. *Coding* the application to meet the design and specifications resulting in a working application, which is then passed on to a testing phase.
3. The *testing or quality assurance* phase verifies the integrity and correct behavior of the application vs. the specifications so it can be deployed in a production environment.
4. The *deployment* stage creates the environment the application will function within when put into production.

These stages are not necessarily separate phases and some of the activity occurs in parallel. For example, testing of certain areas of the application may occur while other areas are still in development. Problems found within any stage might force the application back into a previous stage. A security flaw found during the testing stage, for example, might lead to re-coding or even re-designing part of the application. In addition, the life cycle of the application is often accelerated due to market requirements. This results in a compressed cycle, which leads to a large number of security vulnerabilities due mainly to human error.

The bottom line is, given this complex equation the speed-to-market pressures and the current lack of skilled security professionals available in the marketplace, there is a significant possibility of human error leading to vulnerable applications. Web application security must be addressed at every step in the application lifecycle.

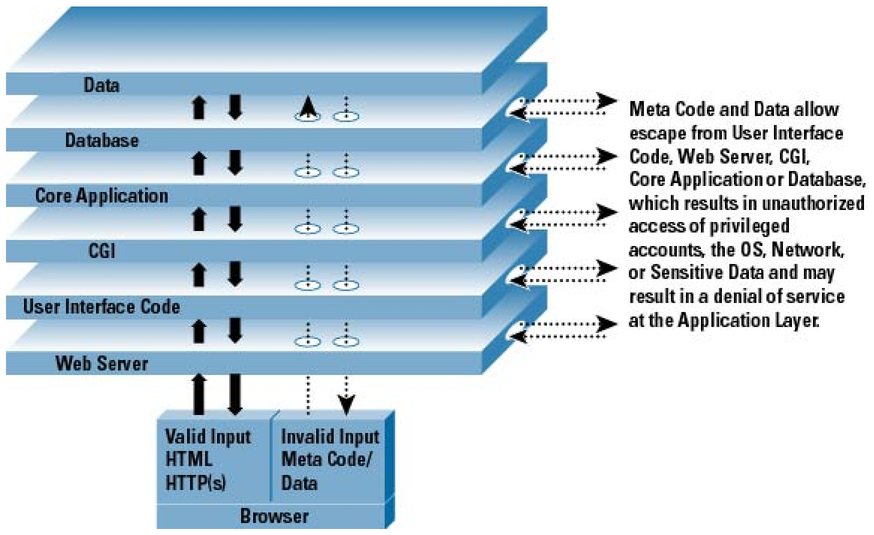
Data
Database
Core Application
CGI
User Interface Code
Web Server
Meta Code and Data allow escape from User Interface Code, Web Server, CGI, Core Application or Database, which results in unauthorized access of privileged accounts, the OS, Network, or Sensitive Data and may result in a denial of service at the Application Layer.
Valid Input HTML HTTP(s)
Invalid Input Meta Code/ Data
Browser

Figure 1: The layers of the web application

2.0 GUIDELINES TO DEVELOPING SECURE APPLICATIONS

With the explosion of web-enabled applications, a new reality has emerged. While you want users to come to your site and interact with the application, to protect your organization you need to be paranoid and build your site assuming the user will try to manipulate the application. Only information that is derived from the enterprise (i.e. the server) can be trusted and everything else (i.e. the client) is always potentially dangerous.

Therefore, the **first guideline** for developing a secure web application is: *Never trust any information that comes from the client, and never assume anything about it.* All security decisions must be made with the underlying assumption that anything that can theoretically be manipulated by the user will be manipulated in reality. Never assume that just because a user supposedly uses a specific tool, that this tool will put any constraints on his actions. For example, the fact that the browser does not show the hidden fields in the HTML of the page does not mean they cannot be seen by looking at the traffic, and manipulated when sent back to the server.

The **second guideline** is**:** *It is always easier to secure simple logic than complex logic.* This actually complements the first guideline as it relates to the usage of logic on the client. The use of client-side logic (such as scripts written in JavaScript or VBScript) might enhance the user experience with the application, but it has serious affects on the site's security. Using client-side logic as a security mechanism by checking information on the client is easily subverted since the mechanism can be bypassed. For example, having JavaScript verification of constraints on form parameters can easily be bypassed by directly creating the request outside the browser. In addition, any changes to the application logic or flow by changing links or parameter values, creates a complex program with multiple sources of flow instead of a single unified point or a simple flow structure.

2.1 THE HIERARCHICAL VIEW

The guidelines for securing a site's applications can be organized in a hierarchical structure in order to address security at each level. The first level, the single transaction, is the smallest piece of logic in a web application. The next level is the complete session and is made up of multiple transactions. The uppermost level is the complete application including a large number of different sessions. Examination of the security requirements for each level is necessary.

2.1.1THE FIRST LEVEL: THE SINGLE TRANSACTION

The basic building block of a web application is the single transaction; this is the single way of accessing an object. Securing the single transaction will give the developer safe building blocks, which can be used to create more complex entities.

Encoding of the transaction

The first step to providing a secure transaction is bringing it to a standard format that can be understood with no ambiguities or multiple representations. Encoding transformation brings the transaction into a canonical format, and is crucial before any additional security measures are performed on the request. The same transaction passed using different encodings may have different lengths and reflect different patterns, making security measures ineffective. The application must establish a *single encoding scheme* for each request (preferably common to all requests within an application) during the design stage, which is non-ambiguous and has a single representation. The decoding function should be the first to be invoked during transaction processing.

Parameter values within the transaction

A transaction is built from an object name and parameter values for the object to operate upon. It is crucial to protect the parameters enforcing their validity and fit with the application logic. An attacker of the web application may change the value of any parameters sent to the server. As a result, nothing can be assumed about this input. *All input must be checked for maximum number of characters on the server side before any use of the parameters.* Setting the maximal side on the HTML page or verifying the input on the client via a scripting language may be useful from a functionality standpoint but cannot serve as a security measure. The client can easily remove the client-side tests by changing the page on his browser or by creating the request outside of the browser. Moreover, even in parameters where the input is not free text, such as a pull-down menu or hidden field, no assumptions should be made on the parameter length since they can assume any value. The validation check for all of the parameters should always be done following the canonization of the encoding of the parameters to avoid changes to the parameter value that may result due to changes in the encoding. Most fields should contain only characters that are used in the specified language and not any meta-characters such as <>"& etc. that can be used to encode special attacks. Filtering such characters out of the parameters to avoid the attacks should enforce this point.

It is important to realize during the design stage which parameters might receive special characters and make them the exception. It is important that for these parameters, only the specific characters are allowed and potentially dangerous sequences are eliminated. It is preferable to start from the "positive" characters such as identifying the legal characters (i.e. A-Z and 0-9) and add as needed, instead of removing the illegal characters. This will ensure getting a minimal subset of needed characters and ensure unwanted characters are being filtered.

Parameters should typically have as many constraints as possible attached to them. Always attempt to *avoid free format input* wherever possible. Instead, define the most limiting structure. The definition of the correct input should be defined in the design stage and should be enforced by adding the specific attributes during the coding stage. Such constraints include the maximal size and the valid characters for the field using the CHARSET HTML attribute. In addition, whenever possible try to choose specific values from a list rather then a free choice of input. For example, instead of allowing users to type the two characters of the state with the USA, provide a list of possible states.

Obscurity

Obscurity does not provide security by itself. However, it is typically better to keep things hidden instead of exposing them to easy access and manipulation. *Obscurity may be easily improved by using HTTP POST method instead of the GET method.* This causes the parameters to be passed in the body of the request instead of being visibly seen and potentially changed within the URL itself. This eliminates many of the naïve URL changes by users trying to poke around the application. By itself, this will not stop the advanced hackers that typically have tools for seeing and manipulating the POST parameters. Improving obscurity may also be performed using a closely related technique: the removal of all parameters from links. Links that contain parameters should either be encrypted or signed for improved security or at least passed as hidden parameters for improved obscurity (see hidden parameter guidelines separately).

Typically, the best way of maintaining the transaction information is by keeping it on the backend and away from the client. However, in some cases the simplest and sometimes only possibility is by passing the information through parameters that are usually passed through the client as hidden parameters. As before, the parameter names and values should be encrypted to avoid revealing their value or at least signed so they cannot be manipulated.

Another important factor for concealing information is the removal of meta-information. Meta-information that exists within the web pages, such as comments sent to the client, can provide valuable clues to the hacker and aid in exposing more vulnerabilities. Therefore, all comments should be stripped from web pages in the production environment. Any client-side code or parts of the HTML that are commented out should be removed as well.

Dynamic Page Creation

Dynamic sites are based on the ability to provide users with a tailored experience, created for that specific profile and session. As part of this process, aspects of the user input should be taken into account to create the dynamic pages sent back to the user.

However, this should be done with care: *Never use values received from the client to directly create dynamic pages.* Implanting users' input directly within pages can lead to severe consequences. For example, a parameter received from the user that contains his/her name might be used to create a page containing a "welcome user" phrase. However, the client might submit a JavaScript code within this parameter that, when implanted in the page, will cause cross-site scripting and lead to a virtual hijacking of the client's communication. This implanting of the script leads to it running in the user's browser and copying or altering data that is passed between the client and the site. *The input should always be verified to be "script free" by removing any dangerous characters.* Care should be taken to verify that any encoding is transformed into canonical structure before removal of the characters.

Usage of HTTP headers

HTTP headers raise security issues. Similar to all other information from the client, these fields are easily manipulated. Therefore, while they might be used to provide certain functionality, they should never be used to provide security. A good example is the REFERER header, which reflects the page leading to the transaction. It can be manipulated to contain any desired URL. Whenever a header needs to be used it must be signed and preferably encrypted, such as in the case with cookies (see more details in the whole session security section). Although such headers should never be used for security, they can be used to make hacking attempts more complicated. For example, by using the REFERER to disqualify some requests from outside the site, a hacker will need to work harder and make manual changes to overcome the obstacle.

Standards

All web protocols and standards are well defined within specifications; however, most web servers and web applications accept a large number of deviations from the standard. Using standard HTML and HTTP will help avoid the possibility of misbehavior by any of the components interpreting the information. Whenever a non-standard protocol is used, it may lead to ambiguity and ill-defined responses. Since it is important at each stage to understand the expected transaction, it is best to avoid potential ambiguities.

The following list contains links to specifications of the most common standards and protocols:

HTTP/1.0 - http://www.ietf.org/rfc/rfc1945.txt
HTTP/1.1 - http://www.ietf.org/rfc/rfc2616.txt
HTML 3.2 - http://www.w3.org/TR/REC-html32
HTML 4.0 - http://www.w3.org/TR/html4
HTML 4.01 - http://www.w3.org/TR/html401

2.1.2 THE SECOND LEVEL: SESSION SECURITY

Multiple requests are organized together in *sessions*, which are tied to a logical entity representing a single user who is using the application. The preliminary condition to securing the session is to secure all its components i.e. the single transactions. Without this preliminary condition, no security efforts for securing the session will succeed.

Web-based protocols and standards such as HTTP and HTML are context-free. Therefore, a context mechanism needs to be created using the application level. A session is typically initiated using an authentication process where the user is identified, usually via a simple user name/password mechanism. Following successful authentication, the user is given a token that identifies them to the application and provides the context in which all their interactions with the server are evaluated.

Authentication

The session creation stage is especially prone to security attacks. Once a session is created, all further actions are associated with this "legal" session without the need for further authentication. There are several rules for securing this process. The most basic is to always pass the authentication information (such as user name/password) over a secure medium, such as SSL. This verifies that the information will not fall into the wrong hands due to tapping. In addition, a strong password scheme is needed to prevent enumerations over the possible passwords due to a short password or obvious ones, which could be found in a dictionary.

In addition, no default users and passwords should exist in the application such as demo and administrative users. Such default users tend to remain accessible after deployment, making them easy prey for attackers.

The authentication process should assume that it might be attacked by enumeration, and therefore use defensive measures to address this case. The authentication mechanism should contain a delay to slow down enumeration. In addition, a threshold should be given for multiple failed authentications for a specific account, both as a measure of consecutive attempts and regarding the number of failed attempts over a longer period of time. Finally, a secondary authentication mechanism, separate from the first, is strongly recommended to perform crucial transactions. For example, having a first set of credentials for entering the online bank, and a second internal set of credentials for transferring money out of the account.

Session Maintenance

Following the authentication process, a session is created and must be associated with the user authenticating into the system. It is crucial to create a secure session identifier since comparison to this identifier actually bypasses the whole authentication mechanism. The session identifier must be cryptographically strong, signed and time stamped. It is better to use well-known algorithms and avoid "inventing" new algorithms since they are prone to design mistakes. The identifier itself has historically been passed as a cookie sent to the client and submitted back to the server with every request. However, due to privacy concerns some applications have moved to URL mangling which adds the identifier as either a parameter to the URL or even as part of the path part of the URL. Note, as the identifier in the URL, it will become part of the REFERER header when the user accesses a different site and will therefore be exposed to other sites. All private areas, following the login procedure of the site, should be associated with the session identifier. Switching to a weaker session ID or counting on IP/SSL information to maintain the user's identity causes a vulnerability for the whole session.

In addition, using a session identifier for all the public areas is a good obstacle for automatic and manual attack methods used by hackers. It will help slow them down or cause them to leave the site. Using this method, a session ID is attached to the user whenever he accesses the application even prior to the authentication. Following the authentication, a second identifier may be used or the public identifier may be associated with the login credentials via the backend. In any case, all session IDs must be secure (i.e. signed and encrypted via a strong crypto mechanism). *The session identifier should never be changed by the client.* Any code running on the client cannot guarantee a secure ID change without opening security vulnerabilities.

Session Termination

Session termination is an important aspect of session maintenance. Unattended sessions left open over time provide a huge security hole for potential breaches. Non-terminated sessions enable attacks on the specific session and potential identity theft of the user owning it. All attempts should be made to shorten the vulnerability period of the session. The session should be terminated under any of the following conditions:

- **Inactive session** – a session that has not been active over a reasonable time, which varies between sites and applications, typically 15-30 minutes.
- **Long session** – a session, which despite being active has reached the maximal allowed time and must be re-authenticated or created. The time can vary according to the application type, typically several hours.
- **Logoff / Logout –** a session should always enable the user to logoff/logout of the session as the most secure option.
- **Security error –** any security error in the application should immediately result in termination of the session.

Flow Maintenance

By default, web based protocols have no flow, and the transactions have no inherent order and sequencing. However, they often do explicitly demand some flow such as forcing an authentication point before accessing private data. More importantly, many web applications contain a large number of implicit constraints on the flow of the application. For example, applying for an account might require filling multiple forms sequentially without the ability to jump to the next form before filling in the prior form. Enforcing this flow is crucial to prevent hackers from getting to parts of the application that they should not reach. *During the design stage, it is important to define all of the possible states of the site that describe its flow and assign every single web page to one of the possible states.* The state (or flow) should be maintained by the server and might be sent to the user as a cookie or a field. It is important that the state indicator should be encrypted and signed for the specific session to avoid its hijacking and usage during attacks on the site. All subsequent web pages must have a different state with a direct or indirect flow from the initial state.

2.1.3THE THIRD LEVEL: APPLICATION SECURITY

Application Organization

The organization of the application plays a major role in the overall security of the site. *During the application design stage, the areas accessed without session information must be determined.* These areas may be accessed by tools that include search engines and other non-session dependent crawlers. It is recommended that these areas be separated into individual directories or servers to avoid mixing the public data with private areas of the application. The simpler the application structure, the greater the chance for achieving good security. Removing cross dependencies within the applications is an important step. All linkages between the applications can increase the security complexity. Therefore, the number of cross-links between the applications should be limited and clearly mapped with their security considerations understood.

Entry Points

Providing security for the application is impossible unless it is clear which entry points exist to the application. These are the points where users can access the site from the outside. Once defined, the entry points can be secured and the flow from them determined. The first stage, therefore, is the definition of entry points, which fall into four main categories:

- User access - these are the root points of the different applications. Any increase in the number of entry points causes a similar increase in the needed effort for securing the application during all of the development stages: design, coding and testing.
- Search and Index agent access- such agents do not maintain a session, and therefore must consider any page they access as an entry point. This is a crucial design consideration to avoid publicly exposing the indexing of pages. The pages that should be indexed must be defined during the design stage and should be considered as public and session-less. It is important to differentiate these areas of the application preferably by assigning them to different areas of the site hierarchy. Another measure, which might prevent a mistaken indexing of a confidential section, is the usage of the robots.txt file,

which can limit access of the search agents or robots.

- Bookmark access- during the design stage, it is important to define the behavior of the application when it receives a bookmark. Bookmarks, which are public and session-less, should be accessible as entry points. Bookmarks into session-based pages are typically those following authentication and should redirect to a legal entry point instead of to the requested object.

- Secure entry points - typically given to partners allowing them to access private areas of the site. These areas should not be treated as simple entry points. They should always be identified early in the design stage and be limited in their number to minimize the threat that they will be used to bypass authentication mechanisms. Such links to non-public areas should always be signed to provide identification of the partner. The signature will typically be within the path of the URL or as one of the parameters. Such entry points should also be passed over SSL to prevent their interception and re-use by an attacker.

There should be as few entry points as possible to maintain a reasonable review process of pathways to the application on an ongoing basis. All other entry points should be disabled.

Encryption

Encryption is a key aspect in providing security to the web application. Encryption may be performed on the information stream using SSL or on specific parts of the transaction. The complete transaction should be encrypted using SSL for *all private areas of the application.* These transactions, which follow the authentication process, are unique to the user and should be kept from interception or change. However, designing the application should not create dependencies on the encryption itself, which could enable the addition of an SSL proxy between the clients and the website. The links should be relative and not contain the **https://** prefix**.** It is also important never to use IP addresses in the URLs but rather to use host names to allow flexibility and the addition of secure applications (or better yet, to keep the links relative).

Applications, which include proxies and SSL accelerators, might change the IP addresses of the server. It is important to remember that SSL acts as a stream encryption mechanism thereby protecting the transport layer and is not specific to the application. Therefore, in addition to the encryption of the data stream, it is important to encrypt specific fields in the HTML forms and mangle links (URLs) within the pages. This is necessary to obscure the content and structure of the application and avoid its probing. All the fields and links used by the server should be signed, and preferably encrypted, or compared with values stored in the backend to avoid their manipulation.

Caching

By allowing proxy servers to cache information, a site actually transfers part of its logic to other servers outside the organization (be that proxy servers or user browsers). External caching servers are important for the performance of the information retrieved from the application. They provide an important method of speeding the delivery of multimedia information to the user. However, their use might cause a security hazard. As a guideline*, never put any content pages (i.e. non multimedia) on external unprotected cache servers.* These pages, which are part of the application, will expose the application to changes in the pages, cause loss of control over the application flow and add complexity in the security design if put on the external server.

Submitting private information with the "no-cache" indication can also protect the privacy of the application's user. This will prevent records from remaining on the user's computer, cached for future use, after the web page is served. Removal of the page by setting this indicator not only adds to the obscurity, but also protects the user's private data from being copied or manipulated.

2.1.4 WEB APPLICATION ENVIRONMENT

The web application is situated within a complex environment. This environment can add a large number of vulnerability threats to the application. Only by deploying the application securely within this web environment, is it possible to have a secure web application.

Production site

The server running the production version of the application must always be separate from the rest of the internal servers (typically in a DMZ, see following section). This server should not run any other software that might disrupt the web application and should never be used to develop the application code. This avoids temporary, old, saved files, etc., which are saved both automatically and manually as part of the development and maintenance cycles. The production server should maintain a sterile environment cleaned thoroughly before every new release. The application should then be copied into the sterile production environment from an *internal* computer. This will ensure that only the minimal needed parts of the application actually reside in the production environment.

The production site should never be administrated from outside the organization in order to prevent an abuse of the management application. It is best not to use remote administration even from within the organization. Administration should optimally be executed locally on the production computer.

DMZ

The DMZ is a crucial component of the periphery and network defense for the organization. In addition to providing the network level protection, it is also a vital part in creating a safe environment for the web application. The DMZ serves as a way of separating external-facing machines from the internal machines, thereby separating the web applications from each other. It is also an important structure for enabling the use of the same applications for dealing with internal vs. external users of the system. Using the DMZ, the application's front end lies within the DMZ while the backend resides in the internal part of the network. By maintaining such a structure it is easier to build differential access for external users accessing through the DMZ and internal users accessing the application through the intranet or other access methods.

A more secure configuration, from the application standpoint, can be achieved by separating the DMZ into two parts. The first part will contain the public areas of the application and the multimedia files on separate servers. The second part will contain the private areas of the application which will also be the only part accessing the back-end system. In such a configuration, a compromise to the public part of the site is still prevented from spreading to the more critical parts of the application.

2.1.5 3RD PARTY TOOLS

The web application typically contains a mix of 3rd party software coming from large vendors to freeware including software for the web servers, application servers, ecommerce packages, etc. Although they are not part of the internal development cycle of the organization they still affect the overall security of the web application.

Secure configuration

The installation of 3rd-party tools tends to lead to less-than-secure configuration by default. All effort should be made to achieve the most secure configuration of any of the products used as part of the web application. In many cases, the vendors will have a guideline for creating a maximum-security installation. Such guidelines should be followed as closely as possible. In addition, it is crucial to verify that all the *default accounts are removed from the system and that any default passwords are changed.* This prevents attackers from simply going into any of the publicly available lists or users and passwords for any of the infrastructure products. Most 3rd party tools will come with demos and sample applications associated with them. These sample applications often prove to be the Achilles heel of the web application, since attackers of the application may abuse them. *All demo and sample applications not needed must be removed from the production server and preferably, never installed in the first place.* Moreover, the most secure way to install3rd party software is to install it on separate servers from the rest of the application (whenever possible), thereby minimizing the potential hazards that it might cause.

Patch process

The majority of the vulnerabilities in 3rd-party tools affect a large number of organizations. Some, like Microsoft's IIS and Apache, are a part of millions of web applications. Therefore, when a vulnerability is found in one of these products, a patch is released to prevent an attack. Patch latency, or the length of exposure to the vulnerability, is the most critical issue during this process. This is the time between the initial discovery of the vulnerability, until the actual deployment of the patch in the production environment that prevents the vulnerability from being used. The patch latency period contains many stages (see figure 3) and minimizing the length of time affected by this process should be a priority. There are many websites and mailing lists available that make released vulnerabilities and patches available publicly. The most comprehensive is the BUGTRAQ mailing list maintained by SecurityFocus[1].

Once a patch has been discovered, the following process should be performed as soon as possible. Install the patch on all the product installations that might be affected. Following the initial installation of the patch, any additional copies of the product in the company (added after the patch is released) must have the patch installed as part of the initial installation process. It is important to remember that even in the best case and under extremely fast cycles of patch installations, the site will still be vulnerable for at least the period between the hacker discovery of the vulnerability and the patch rollout by the application vendor.

[1] http://www.securityfocus.com/

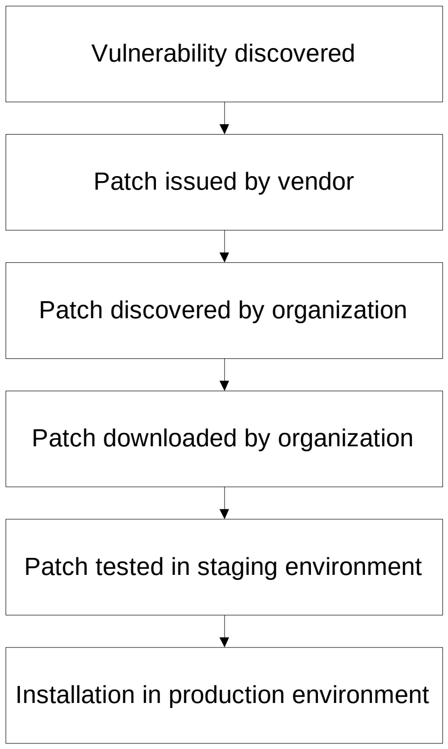
Vulnerability discovered
Patch issued by vendor
Patch discovered by organization
Patch downloaded by organization
Patch tested in staging environment
Installation in production environment

Figure 2: The patch latency stages

3.0 SECURITY CHECKLIST SUMMARY

As a summary of the guidelines highlighted in this paper:

3.1 GENERAL GUIDELINES

- Never trust any information that comes from the client.
- Never assume anything about the information from the client.
- Don't use client side logic as part of the application logic.

3.2 TRANSACTION SECURITY

- Always translate incoming requests into a standard encoding scheme.
- Verify the maximum number of characters for all fields.
- Verify that the input does not contain dangerous characters.
- Avoid free format input; put as many constraints on it as possible.
- Never use values received from the client to directly create dynamic pages.
- Obscure your transaction by using POST method instead of GET.
- Parameters passed through the client must be encrypted and signed.
- Remove all meta-information from the information sent to the client.
- Use only standard protocols in the application.

3.3 SESSION SECURITY

- Authentication information must always be encrypted.
- Strong password schemes must be used to prevent enumeration.
- Require secondary authentication for critical parts of the application.
- Never leave default accounts on the application.
- Use defensive measures to counter authentication attacks.
- All private areas of the application should be associated with a session identifier.
- The client or client code should never change session identifiers.
- Session must be terminated under the following conditions: inactive session, long session, logoff, and security error.
- Always define all the possible states of the application during design stage.
- All web pages should belong to one of the states.

3.4 APPLICATION SECURITY

- Areas without any session information must be identified during design stage.
- Application entry points must be established during design stage,
- Minimize the number of cross-dependencies between applications.
- Encrypt the stream all private areas of the application.
- Do not make links dependent on the encryption to allow use of proxy.
- Encrypt and sign critical information (in addition to the stream encryption).
- Always use the "no-cache" indicator for private information.

3.5 APPLICATION ENVIRONMENT

- Never develop on the production server- always copy to it from an internal server.
- Keep only the minimal part of the application on the production server.
- Never put any content pages on external cache server.
- Separate your application to make use of the DMZ.

3.6 3RD PARTY TOOLS

- Use the vendors' guidelines for maximum-security installation.
- Remove all default accounts and change any default passwords.
- Remove all demo and sample parts of the application.
- Follow an updated list of patches for all the tools.
- Minimize the length of the patch latency.

4.0 CONCLUSION

Developing a secure web application is a complex task demanding constant attention throughout the application's life. Any mistake during any of the stages ranging from the original design through to the final deployment will cause severe security vulnerabilities. Today's Web developers are facing new challenges never encountered by their predecessors. They are developing for users access to the application, but must anticipate malicious intent of manipulating it. To have a truly secure application, attention must be given to security in all life phases of the application lifecycle– from design, through coding, and onto deployment and maintenance. Security must be thought of at all levels within the application – the single transaction, the session, the application, and the whole server. This effort must be methodic, continuous and built into the development-deployment-maintenance process of the organization. This probably will force most organizations to evaluate how their applications are developed today. The only way to beat the hackers at their own game is to start thinking about security from the earliest stage of Web application development, and then fight automation with automation at each stage of the application lifecycle.

ABOUT WATCHFIRE

Watchfire helps organizations manage their online business by providing software solutions to minimize online risk and maximize channel effectiveness. These solutions let ebusiness, marketing, legal, compliance, and IT departments measure, manage, improve and secure their online channels. We analyze online content and applications and generate management intelligence dashboards that report on visitor behavior, quality, data privacy, web application security, and accessibility issues.

Watchfire recently acquired Sanctum, the pioneer in web application security testing and firewall software. Sanctum's web application security testing software, combined with Watchfire's Privacy solutions, help organizations institute comprehensive online risk management programs.

More than 400 enterprise organizations and Government agencies, including AXA Financial, SunTrust Banks Inc., Veteran's Affairs, United States Postal Service, and Dell rely on Watchfire to monitor, measure and manage all aspects of the online business including quality, privacy, web application security, accessibility, user experience and visitor behavior. Watchfire's alliance and technology partners include IBM Global Services, PricewaterhouseCoopers, TRUSTe, Microsoft, Interwoven, Documentum and Mercury Interactive. Founded in 1996, Watchfire is headquartered in Waltham, MA.

GLOSSARY

CSS (Cross-Site Scripting) – injection of a script into a web page causing it to run on the client's browser http://www.cert.org/advisories/CA-2000-02.html.

DMZ (Demilitarized Zone) – a network between the Internet and the internal network that is used to keep external facing content. Its main use is for security reasons that dictate separation of the direct interaction of the Internet with the internal systems.

HTML (Hyper Text Markup Language) – a language for defining the structure and appearance of documents on the web.

HTTP (Hyper Text Transfer Protocol) – a protocol that defines how messages are formatted and transmitted, and what actions web servers and browsers should take in response to various commands within the messages

HTTPS (HTTP over SSL) – a protocol that uses SSL as the communication layer of the HTTP protocol.

SSL (Secure Sockets Layer) – a protocol (originally developed by Netscape) to transfer information securely over the network. This protocol is in use by all major browsers and is the leading encryption mechanism for web-based information.

URL (Uniform Resource Locator) – the standard address of objects on the World Wide Web.

To evaluate Watchfire® AppScan™, please visit
https://www.watchfire.com/securearea/appscanauditdownload.aspx