WHITEPAPER

SECURE WEB APPLICATION FRAMEWORK MANIFESTO

Authors: Tom Aratyn, Sahba Kazerooni, Patrick Szeto, & Rohit Sethi

Version 0.05

http://www.securitycompass.com
labs@securitycompass.com

CONTENTS

1 Mission Statement

The Secure Web Application Framework Manifesto is a document detailing a specific set of security requirements for developers of web application frameworks to adhere to. The manifesto centers around the following beliefs:

- Frameworks that are 'secure by default' will yield a dramatic reduction in the number of common web application security vulnerabilities.
- Application security experts should provide, on a regularly basis, updated guidance to framework developers on how to incorporate mechanisms to avoid newly discovered vulnerabilities.

2 INTRODUCTION

Developers are increasingly relying on scaffolding-based systems like Rails and Django to build applications. The number of web application frameworks, scaffolding or otherwise, is constantly growing and it's becoming increasingly clear that securing these frameworks will be a major boon for the future of secure web applications.

In the words of Jeff Williams, we have plenty of "painkillers" for web application framework developers to follow such as lists of vulnerabilities to avoid. The Security Analysis of Core J2EE Patterns was our first attempt at providing "vitamins" or positive advice to framework developers on what they should do to incorporate security into their design. Recognizing that many developers are gravitating to leveraging web application frameworks, we decided it was time to provide a list of positive features that these frameworks should include.

This "Secure Web Application Framework Manifesto" must, of course, be a living document. At any given point, it should provide a minimum baseline of what a web application framework should include to appeal to security-conscious developers. We contend that if such a web application framework is broadly adopted, it will have far reaching effects into web application security.

Adhering to the manifesto is only a starting point. Developers still can, and surely will, introduce vulnerabilities not covered by the manifesto; especially those pertaining to their core domain such as fine-grained authorization. Secure-by-default frameworks are compliments but not substitutes for developer security awareness.

The Manifesto is not an exhaustive specification. It is designed to provide a minimum standard for frameworks to adhere to in order to facilitate development of secure web applications. Some of these features will come with tradeoffs in performance or usability. Security features should be turned on by default with the option to turn them off explicitly. In some cases, the usability or performance trade-offs may be so great that framework developers will turn the features off by default. Such decisions should be the exception and not the norm.

2.1 INTRINSIC SECURITY WORKING GROUP

We owe a debt of gratitude to Arshan Dabirsiaghi and the entire OWASP Intrinsic Security Working Group. The ISWG aims to measure the security of various frameworks – the inverse of the Secure Web Application Framework Manifesto, which aims to provide the measuring stick itself. Although we initially started this manifesto independent of their work, cross referencing their requirements helped us identify gaps in the Manifesto.

3 REQUIREMENTS

3.1 DATA VALIDATION

3.1.1 PROVIDE TOOLS TO GUARANTEE THAT DATA WILL NOT BE CONFUSED WITH CODE FOR ALL INTERPRETERS

Confounding data with executable code is the cause of the most pervasive application security problems: Cross Site Scripting (XSS), SQL Injection, buffer overflow and several others. The fundamental problem arises when developers can mix user-supplied or user-influenced data, such as HTTP parameters, with static or system-generated code. The resultant data is then executed or otherwise interpreted by a process which can no longer differentiate the code from the data. An obvious example of this principle at work is SQL injection. Pseudo code:

```
bad_query = "select * from accounts where accountid = '" +
user_supplied_value + "'";

DatabaseTool.executeQuery(bad_query);
```

In this example, the database tool has no way to differentiate which parts of the query variable came from the string literal and which parts came from the user supplied value. Most modern programming languages and development frameworks offer a way around this by offering parameterized queries or prepared statements.

Pseudo code:

```
PreparedStatement good_query = "select * from accounts where accountid
= ?";

good_query.setParameter(1, user_supplied_value)

DatabaseTool.executeParameterizedQuery(good_query);
```

DatabaseTool has a number of ways to protect against the vulnerability in the second example. For example, the tool could pre-compile the string literal and pass the parameters to the database separately. The database is then responsible for not misinterpreting any portion of the user supplied data as SQL code. Such an approach renders SQL injection impossible.

Another approach is to encode unsafe data in a context relevant format. For example: to mitigate against Cross Site Scripting, a secure web application framework could automatically HTML, HTML attribute, cascading style sheet, or JavaScript encode nearly all non alpha-numeric characters depending on the context. The encoding functions in the OWASP ESAPI project for Java serve as an excellent reference for this approach. At a minimum, a secure web application framework should endeavor to follow the XSS Prevention Cheatsheet. Ultimately, the strongest form of XSS protection is context aware escaping, demonstrated functionally in Google's template system. No matter which approach the framework uses, the end result must be that developers would have to go out of their way to introduce XSS.

A secure web application framework should provide tools that guarantee separation of data and code for all interpreters, including (but not limited to) all of the following, if applicable:

1. SQL statements
2. HTML
3. HTML attributes
4. JavaScript
5. Cascading StyleSheet (CSS)
6. VBScript
7. URL
8. XML
9. XML Attributes
10. DTDs (if supported)
11. XPath
12. LDAP
13. Flash
14. HTTP response headers
15. Email (SMTP)
16. Log files (optional)

All demo code and documentation should use these safe libraries rather than unsafe String concatenation.

3.1.2 Provide an API to Validate all Input

A secure web application framework should provide an API to validate all forms of user-supplied input, with support for customized regular expressions. The Apache Struts Validator plug-in is a good example of a framework-level resource that allows developers to define validation for form fields. A secure web application framework should provide a similar mechanism for *all* forms of user-supplied input:

- HTTP parameters (names and values, GET and POST)
- HTTP headers
- URLs
- Cookie values
- Data from any other un-trusted/tainted source. For example XML data when XML is used as a transport mechanism such as with Simple Object Access Protocol (SOAP), or data extracted from a database

3.1.3 Provide Tools for Safely Interacting with Functions Developed In Unmanaged Code

A secure web application framework should have special provisions to deal with null and other non-printable characters (e.g. replacement with a safe character) when interacting with functions developed in unmanaged code such as those provided by an operating system. For example, a file open operation is often delegated to the operating system, and supplying the null character ("%00") as part of a file name in a web application will not terminate a string in a managed language but will terminate a string in an unmanaged language. The framework should recognize this as a danger in the file name or path parameters of a file open operation. On the other-hand, since a null character could be a valid character in a file write operation, the framework should allow null characters as part of a file stream.

3.1.4 Specify an Encoding Format for Every Page

Not specifying an encoding format may mean that the browser interprets the encoding format based on page content. This can lead to obscure XSS vectors. A secure web application framework should assign a consistent encoding format such as UTF-8 to all response pages unless there's a specific reason to use a different format.

3.1.5 Do Not Accept Illegal Characters

Overly long and malformed characters in variable length encoding formats such as UTF-8 can be used to bypass filters and may sometimes be translated to the proper format after sanitization by a different component or application. A secure web application framework should only accept properly formed characters for the chosen encoding format.

3.1.6 Provide Optional Hashed Message Authentication Codes (HMAC) for Read-Only Client Parameters

Parameter manipulation on hidden form fields continues to be a devastating class of attacks on web applications. The .Net framework provides a defense in the form of an HMAC for ViewState. The HMAC solution:

- Takes a hash of read-only fields in a form prior to sending them to the client
- Encrypts that hash with a secret key stored on the server
- Adds the hashed and encrypted value as an additional hidden field in the form
- Upon form submission, rehashes and re-encrypts the read only client-supplied parameters and compares the hash with the client-supplied HMAC parameter. Any difference indicates that one or more of the read only parameters were tampered with

This approach picks up changes to the HMAC without forcing the server to maintain client state. Developers can choose whether they want to store data on the server or allow the data to be sent to the client while remaining confident that the data is not susceptible to manipulation by the client.

3.1.7 Automatic Content Security Policy (CSP) Header Generation

Mozilla proposed CSP to help protect against Cross Site Scripting. Although CSP, at the time of this writing, is not fully implemented in most browsers, a secure web application framework should proactively provide this control for when CSP becomes standard.

In a nutshell, CSP allows developers to specify which domains a web application allows to host its scripts. A browser that complies with CSP will only run scripts from the whitelisted domains and will not execute inline or event handling HTML attribute scripts. A secure web application framework should seamlessly generate CSP headers and provide tools to easily extend the list of whitelisted domains when required. By default, the framework should disallow all other domains and disallow support for inline or event handling HTML attribute scripts.

CSP also helps protect against Clickjacking by specifying "which sites may embed contents from my site". By default, the framework should disallow all other sites from being embedded within the application's contents.

3.1.8 Automatic Origin Header Generation

Mozilla also proposed the Origin Header to protect against Cross Site Request Forgery (CSRF). Supporting clients send information about the origin of each request to the server. Wherever possible, a secure web application framework should be able to ensure that the origin of request is from the expected domain.

3.2 Authentication and Authorization

3.2.1 Default Deny Access Control Policy

A secure web application framework should provide configurable methods for providing authentication and access control requirements for specific resources. See Specifying Security Constraints in Java EE. This feature should use a default deny policy, thus denying access to resources if the request does not meet the criteria specified in the configuration. Furthermore, it should not be possible to bypass web authentication and authorization with HTTP verb tampering, as defined in the paper by Arshan Dabirsiaghi.

3.2.2 Provide Access to Authorization-Relevant Data Throughout the Lifetime of a Request

Authorization is a difficult problem in web application security. Compounding the problem is the fact that many authorization decisions need to be made well past the initial presentation layer. Developers should be able to access authorization data at any point while servicing an HTTP request. The OWASP ESAPI project for Java facilitates this by storing Access Control data in a Thread Local – allowing a developer to access the code at any point during the lifetime of that thread.

3.2.3 Provide Indirect Object Reference Functionality

Applications often allow users to access sensitive resources such as user-specific files from the application server. Direct object references use the actual file name (e.g. “file=statement1.pdf”) whereas indirect object references provide an independent identifier that the application later translates into an actual filename (e.g. “file=a”, where ‘a’ later translates to statement1.pdf).The problem with the former method is that attackers can sometimes access files that they shouldn’t (e.g. “file=../config.xml”). An indirect object reference renders such an attack impossible because the application only provides access to a specified set of files (e.g. all files in a particular directory, or a predefind list of individual files). Unfortunately, the complexity of creating an indirect object reference for each file that is to be accessed by the end user means that many developers end up favoring direct object references.

A secure web application framework should provide functionality that creates and translates indirect references for a specific file, a set of files, or all files in a particular directory or directories. See the ESAPI Java AccessReferenceMap and .Net’s Web Resource mechanism for examples of this functionality.

Note that this control applies specifically to resources that require access control. Publicly-accessible static content such as JavaScript or Cascading Style Sheet files that are normally stored on web servers do not necessarily need this protection. On web servers, use operating system or server controls to prevent forcible Path Traversal attacks.

3.2.4 All Stored Passwords Should be Hashed and Salted with Random Bytes

A secure web application framework should provide all the functionality necessary for a developer to implement secure authentication. In particular, authentication should use a secure hashing algorithm salted with a fixed length random byte sequence (see Jasypt). Both the hashing algorithm and salt length should be configurable in case a particular hashing function is defeated in the future. Secure web application frameworks should default to stronger, slower hashing algorithms (e.g. SHA-2) instead of fast algorithms (e.g. MD5 and SHA-1) to mitigate the risk of off-line brute forcing.

3.2.5 Secure Password Reset / Change

Insecure forgotten password mechanisms are one of the easiest ways for attackers to break into an application. In particular, forgotten password questions are often easy to answer and are susceptible to brute forcing. If a secure web application framework provides password reset functionality, it should use the following sequence or one that provides a similar level of security:

- Users visit a password reset page where they submit their User ID or email address. The page should also include an anti-automation technique, such as a CAPTCHA image with accessibility options for visually impaired users, to prevent user enumeration
- Upon form submission, the site displays a page indicating that an email with instructions was sent to the user
- In the interim, the site should send an email with a temporary link to the user. The link should expire after a short period of time (e.g. 1 hour)

- When the user clicks on the link, they should be taken to a page where they respond to pre-determined forgotten passwords
    - The application should enforce account lockout if the user fails to correctly answer a question after a configurable number of tries (e.g. five wrong answers)
- Users should be informed via email that their password has been changed, but the password itself should never be sent via email

For cases where a user wants to change their password after successfully authenticating, ensure that the user enters *both* the old password and new password prior to changing. This is particularly important in web applications due to the risk of Cross Site Request Forgery on password reset pages.

3.2.6 Prevent User Enumeration

An attacker should not be able to derive a list of valid users. When an attacker can abuse application functionality to create a list of valid users then the application is vulnerable to user enumeration. A secure web application framework that provides authentication functionality should mitigate the risk of user enumeration. In particular:

- Do not differentiate between an invalid username or invalid credential (e.g. password) during authentication
- Use anti-automation controls to prevent user enumeration in password reset
- Use anti-automation controls to prevent user enumeration in new user registration
- Prevent users from determining the cause of authentication failures through timing attacks.
    - Use runtime testing on a variety of environments to determine the causes of timing discrepancies in authentication. If necessary, use optimization or sleep functions to ensure that end users cannot reliably determine the cause of an authentication failure

3.2.7 Configurable Password Policy

Application administrators should be able to configure password requirements, including:

- Password complexity requirements
- Password age (minimum and maximum)
- Password history
- Optional dictionary of words that cannot be used as passwords

See NIST standard for default values.

3.2.8 Defense Against Brute Force on the Login Form

Application administers should be able to select from a range of password brute force defense techniques. There are a number of common techniques for blocking brute force attacks. Two of the more popular are account lockout and rate limiting (or delay introduction).

Account Lockout

Application administrators should be able to configure the number of authentication attempts a user should have before being locked out, with the option to disable account lockout. Administrators should also be able to define whether the accounts will be automatically unlocked after a configurable period of time or whether an administrator needs to explicitly unlock an account. The default should be that administrators need to explicitly unlock an account.

Rate Limiting

The framework should allow system administrators to turn on this feature which would introduce random delays in the authentication process.

3.2.9 Pluggable Support for Two-Factor Authentication

Industry experts often contend that password-based authentication alone is insufficient to protect critical applications. A secure web application framework should provide hooks into the authentication process to seamlessly integrate other forms of authentication, such as smart cards or phone-based authentication. See Spring Security authentication for an example of a system that can easily be extended to include two factor authentication.

3.3 Session Management

3.3.1 Use Cryptographically Secure Random Numbers for Session IDs

Create session IDs from Cryptogaphically Strong Random Number Generators such as Java's SecureRandom rather than a pseudo random number generator like the rand() function in C.

3.3.2 Provide Automatic CSRF Protection

Many web application frameworks are responsible for creating or rendering links and form submission pages. A secure web application framework should provide an option to ubiquitously add automatic anti-CSRF tokens to links where possible. Django provides this functionality optionally. Note that this control is addition to the Origin header control to protect against CSRF because not all browsers support the Origin header.

3.3.3 Prevent Session Fixation

Session fixation is one of the easiest types of attacks to prevent, and can be simply added to any framework which handles authentication. Change the Session ID of a user after successful authentication. Spring Security has built-in session fixation defense.

3.3.4 Provide Built-in Support for Cookie Security Flags

The "Secure" and "HttpOnly" cookie flags can help prevent session hijacking and limit the reach of XSS respectively. A secure web application framework should utilize these flags with Session ID cookies by default. The .Net framework provides these functions, although not necessarily by default. Developers should be able to programmatically set these flags for other cookies.

In addition, developer documentation should clarify the security implications of transmitting session cookies in plaintext.

3.3.5 Support Configurable Inactive and Hard / Absolute Session Timeouts

Many web application frameworks already provide support for inactive session timeouts. For example, Java EE developers can configure the session timeout in web.xml. Frameworks should also provide an option to define a hard/absolute session timeout that limits users to a maximum time on the application. This countermeasure protects against the scenario where an attacker can successfully hijack a session and keep it open indefinitely.

3.3.6 Provide a Configuration Option to Tie Session IDs to an IP Address or Subnet

Some application developers opt to correlate a session ID to the client's IP address. After session generation, the application verifies that each request comes from the expected IP address thereby mitigating the risk of session hijacking. The secure web application framework should provide an option to seamlessly deliver this functionality.

In practice, session IP correlation on large networks is difficult if not impossible due to a variety of networking features – in particular, proxy servers such as AOL proxy. A secure web application framework should provide a middle-ground by allowing developers to specify a subnet length and verifying that each request comes from the same subnet. For example, if a developer configures session subnet correlation with a 24 bit subnet, then the application should permit requests from *10.1.1*.3 and *10.1.1*.5 to access the same session but it should not allow requests from *10.1.2*.3 to access the same session.

3.4 XML Specific

3.4.1 Safe Parsers and Validators

Over the years, security researchers have discovered several vulnerabilities in XML libraries – particularly parsers and validators. Where possible, a Secure Web Application Framework should favor XML Schema Definitions (XSD) over Document Type Definitions (DTD). The framework should disallow dangerous functionality be default, namely:

- Disallow external entity resolution by default

- Disallow DTDs defined internally within XML files by default
- Disallow XML Stylesheet Language Transforms (XSLTs) processing instructions within an XML document's prolog by default

XML libraries should also be resilient to commonly known forms of Denial of Service (DoS) attacks, namely:

- Large and/or recursive payloads for DOM-based XML parsers
    - One way to achieve this is to check the size of the XML payload against some configurable maximum size prior to parsing
- Place strict limits on recursive entity definitions to avoid XML Bombs

3.4.2 Disallow Dangerous XSLT Extensions By Default

XML Stylesheet Language Transforms (XSLTs) can be vulnerable to a number of attacks.

Researcher Bradley W. Hill discovered one particularly dangerous form of attack within the WS-Security standard. The XML Digital Signature specification recommends that implementing libraries allow an XML Digital Signature to include arbitrary XSLTs. The default Java and .Net implementations both provide XSLT extensions that ultimately allow command execution. XSLT libraries should disallow dangerous extensions such as Runtime objects by default. Moreover, the framework should take a whitelist approach to which extensions it does allow and ensure that those extensions do not cause security concerns.

3.5 Cryptography

3.5.1 Support for Transparent Database Encryption

Compliance drivers such as the Payment Card Industry Data Security Standard (PCI DSS) are increasing the need for database encryption. Secure Web Application Frameworks should provide configuration options to seamlessly encrypt columns within a database. Hibernate in Java provides seamless, configurable database encryption. Secure Web Application Frameworks and Object Relational Mapping (ORM) software in particular should allow developers to configure column-level encryption.

See "Encrypt Passwords and Keys Stored in Configuration Files" for more information about how to protect the encryption key.

3.5.2 Configurable Encryption Algorithms

Hard-coding specific encryption algorithms and parameters such as key size may leave applications vulnerable to common attacks if a particular algorithm is ever compromised. Secure Web Application Frameworks should allow developers to configure the algorithm and parameters such as key strengths. See Java's security provider architecture.

Favor modes with secure random Initialization Vectors (IVs) rather than modes without IVs such as Electronic Code Book (ECB).

Developers in non-compiled languages such as PHP, Ruby, and Python often favor scripts written in that programming language rather than static configuration files. Frameworks written in such languages should decouple application code from specific encryption algorithms, either by introducing a static configuration file or calling a utility class (e.g. HashingUtility.hash() rather than SHA2.hash()).

3.5.3 Secure TLS/SSL

Attackers have discovered several attacks on TLS/SSL implementations and X509 certificates: downgrade attacks, plaintext injection during renegotiation null prefix attacks, circumventing Online Certificate Status Protocol (OCSP) controls, and several others.

A Secure Web Application Framework should address all of known SSL threats. Ideally, framework developers should reuse libraries that have already accounted for these attacks rather than writing new libraries. Note that some of the attacks, such as null prefix attacks, are actually attacks against the client; however, these attacks also apply to the server during mutual authentication.

Providing an exhaustive set of requirements for TLS/SSL is beyond the scope of this manifesto. Instead, frameworks developers should consult resources that provide practical guidance on TLS/SSL. OWASP maintains the Transport Layer Protection Cheatsheet which provides rules for secure server design. SSL Labs maintains an SSL Server Rating guide that provides guidelines around certificate type, key size, cipher strength, key exchange algorithm, and protocol.

3.6 Configuration Security

3.6.1 Encrypt Passwords and Keys Stored in Configuration Files

Web application frameworks often store plaintext system passwords and keys in configuration files. For example, several frameworks use plaintext configuration files for database connection strings, database encryption keys, Lightweight Directory Access Protocol (LDAP) connection strings, keystore passwords, and other values. Attackers who are able to exploit other vulnerabilities are sometimes able to view the contents of files. As part of a defense-in-depth strategy, Secure Web Application Frameworks should provide native support for encrypted properties in configuration files. Unfortunately, developers will always run into the problem of providing some sort of password or key to decrypt encrypted credentials. While no solution is perfect, there are several password / key storage options that the Secure Web Application Framework can employ:

- Store a private key, unique for each machine, as a binary file that can only be accessed by the application server. While this control succeeds in preventing attackers from viewing plaintext passwords in configuration files, it does not prevent attackers from first accessing the binary key and then the configuration file using the same exploit. This should be the minimum security option. See Weblogic.

- Store the decryption key / password in a file, similar to the preceding option. Use operating system controls to ensure that file is only accessible by a separate launching process – not the application server. The launching process can then pass in the key / password as a command line argument when launching the application server. This way, a user who exploits the application server may not necessarily have access to the decryption key itself.
- Support passphrases from an environment variable and/or web-form. This solution takes more work and possibly manual intervention but greatly decreases the risk of an attacker being able to find plaintext passwords in configuration files. See Jasypt
- Leverage a distributed service, such as the .Net Data Protection API (DPAPI) User Store key storage. This restricts key access to a particular user, so other users on the same machine (including local administrators) cannot access that key

3.7 Misc

3.7.1 Provide Safe File Upload Functionality

File upload is a common requirement in web applications. Secure web application frameworks should facilitate safe file uploads. In particular, the framework should:

- Provide support for library-based third party anti-virus scanning solutions, such as Clam AV
- Disallow saving a file outside of the specified directory (see the Unix chroot command)
- Validate file contents for common file types such as JPEG, GIF, BMP, PDF, DOC, DOCX, etc. For example, use ImageMagik to first validate the contents of an image file.

3.7.2 Provide Security Specific Logs

Provide a security-specific log that is turned on by default. Automatically detect potential attacks using all of the attack points documented in the OWASP AppSensor Project. Ensure consistent use of event IDs (e.g. SE5 for source change of IP during session). Developers should be able to log to the security-specific log as well.

3.7.3 Provide Click-Jacking Protection

In addition to Content Security Policy, a secure web application framework should provide mechanisms to protect against Clickjacking. Although not yet universally supported, the X-FRAME-OPTIONS header is supported by Internet Explorer 8+. Frameworks should automatically set the X-FRAME-OPTIONS value to DENY or SAMEORIGIN depending on if the application requires nested frames or not. As an additional measure of protect, the framework should also use iframe busting code when a particular page never has to have a parent frame.

3.7.4 Overflow Resistant Arithmetic Options

Many programming languages such as Java are vulnerable to Integer and floating point overflow and underflow. Frameworks should provide optional libraries that allow developers to perform basic arithmetic which throw errors / exceptions upon overflow or underflow conditions.

3.7.5 Configurable Anti-Automation Techniques

Automation is a part of many web application attacks which involve brute forcing or enumeration. Frameworks should provide a means for application administrators to make use of anti-automation techniques such as CAPTCHA on certain pages. Which type of anti-automation mechanism and the implementation of that technique should be configurable. An API should then be provided for including it in code.

3.7.6 Limit Size of Payloads

The easiest way to launch a denial of service attack against certain parsers is to send very large payloads. An easy way to defend against excessively large payloads would be to configure the framework to only accept HTTP payloads of a certain size (e.g. < 1 MB).

3.7.7 Generic Error Pages

Web pages can provide detailed information to attackers. Often error pages are verbose by default. Frameworks should provide generic error pages to users by default, while providing an option to turn on development-friendly error pages that yield interesting debugging information (e.g. stack traces).