DRAFT

DO NOT USE THIS UNTIL IT'S DONE!

This is a draft. Do not use this version. The presence of an Axx does not mean it's going to be in the final or the ordering of the final version.

Current status

Taking comments on this draft, and preparing the graphic refresh in time for the official release on OWASP's 20th Anniversary.

Lead Authors

- Andrew van der Stock @vanderaj
- Brian Glas @infosecdad
- Neil Smithline [@]
- Torsten Gigler []

Contributors

- Orange Tsai, Author of A10-2021: Server Side Request Forgery
- Jim Manico and Jakub Maćkowski - OWASP CheatSheets Coordination

How you can help

At this stage, we are asking for

- Data scientists - please peer review our analysis
- Web designers - we need to make a mobile friendly version
- Translators - please review the English text to make sure it's translatable
- ASVS, Testing Guide, and Code Review Guide leadership - please use our data and help us link our documents and standards together

Log issues and pull requests

Please log any corrections or issues:

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

Introduction

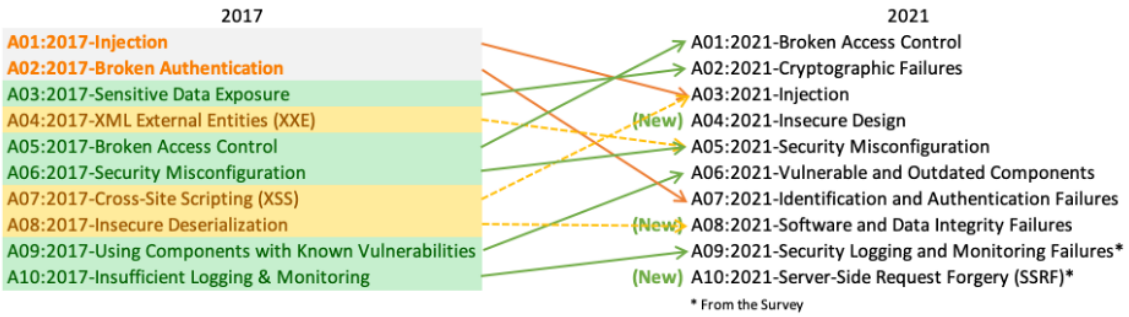
Welcome to the OWASP Top 10 - 2021

Welcome to the latest installment of the OWASP Top 10! The OWASP Top 10 2021 is all-new, with a new graphic design and an available one-page infographic you can print or obtain from our home page.

A huge thank you to everyone that contributed their time and data for this iteration. Without you, this installment would not happen. **THANK YOU!**

What's changed in the Top 10 for 2021

There are three new categories, four categories with naming and scoping changes, and some consolidation in the Top 10 for 2021.



- **A01:2021-Broken Access Control** moves up from the fifth position; 94% of applications were tested for some form of broken access control. The 34 CWEs mapped to Broken Access Control had more occurrences in applications than any other category.
- **A02:2021-Cryptographic Failures** shifts up one position to #2, previously known as Sensitive Data Exposure, which was broad symptom rather than a root cause. The renewed focus here is on failures related to cryptography which often leads to sensitive data exposure or system compromise.
- **A03:2021-Injection** slides down to the third position. 94% of the applications were tested for some form of injection, and the 33 CWEs mapped into this category have the second most occurrences in applications. Cross-site Scripting is now part of this category in this edition.
- **A04:2021-Insecure Design** is a new category for 2021, with a focus on risks related to design flaws. If we genuinely want to "move left" as an industry, it calls for more use of threat modeling, secure design patterns and principles, and reference architectures.

- **A05:2021-Security Misconfiguration** moves up from #6 in the previous edition; 90% of applications were tested for some form of misconfiguration. With more shifts into highly configurable software, it's not surprising to see this category move up. The former category for XML External Entities (XXE) is now part of this category.
- **A06:2021-Vulnerable and Outdated Components** was previously titled Using Components with Known Vulnerabilities and is #2 in the industry survey, but also had enough data to make the Top 10 via data analysis. This category moves up from #9 in 2017 and is a known issue that we struggle to test and assess risk. It is the only category not to have any CVEs mapped to the included CWEs, so a default exploit and impact weights of 5.0 are factored into their scores.
- **A07:2021-Identification and Authentication Failures** was previously Broken Authentication and is sliding down from the second position, and now includes CWEs that are more related to identification failures. This category is still an integral part of the Top 10, but the increased availability of standardized frameworks seems to be helping.
- **A08:2021-Software and Data Integrity Failures** is a new category for 2021, focusing on making assumptions related to software updates, critical data, and CI/CD pipelines without verifying integrity. One of the highest weighted impacts from CVE/CVSS data mapped to the 10 CWEs in this category. Insecure Deserialization from 2017 is now a part of this larger category.
- **A09:2021-Security Logging and Monitoring Failures** was previously Insufficient Logging & Monitoring and is added from the industry survey (#3), moving up from #10 previously. This category is expanded to include more types of failures, is challenging to test for, and isn't well represented in the CVE/CVSS data. However, failures in this category can directly impact visibility, incident alerting, and forensics.
- **A10:2021-Server-Side Request Forgery** is added from the industry survey (#1). The data shows a relatively low incidence rate with above average testing coverage, along with above-average ratings for Exploit and Impact potential. This category represents the scenario where the industry professionals are telling us this is important, even though it's not illustrated in the data at this time.

Methodology

This installment of the Top 10 is more data-driven than ever but not blindly data-driven. We selected eight of the ten categories from contributed data and two categories from an industry survey at a high level. We do this for a fundamental reason, looking at the contributed data is looking into the past. AppSec researchers take time to find new vulnerabilities and new ways to test for them. It takes time to integrate these tests into tools and processes. By the time we can reliably test a weakness at scale, years have likely passed. To balance that view, we use an industry survey to ask people on the front lines what they see as essential weaknesses that the data may not show yet.

There are a few critical changes that we adopted to continue to mature the Top 10.

How the categories are structured

A few categories have changed from the previous installment of the OWASP Top Ten. Here is a high-level summary of the category changes.

Previous data collection efforts were focused on a prescribed subset of approximately 30 CWEs with a field asking for additional findings. We learned that organizations would primarily focus on just those 30 CWEs and rarely add additional CWEs that they saw. In this iteration, we opened it up and just asked for data, with no restriction on CWEs. We asked for the number of applications tested for a given year (starting in 2017), and the number of applications with at least one instance of a CWE found in testing. This format allows us to track how prevalent each CWE is within the population of applications. We ignore frequency for our purposes; while it may be necessary for other situations, it only hides the actual prevalence in the application population. Whether an application has four instances of a CWE or 4,000 instances is not part of the calculation for the Top 10. We went from approximately 30 CWEs to almost 400 CWEs to analyze in the dataset. We plan to do additional data analysis as a supplement in the future. This significant increase in the number of CWEs necessitates changes to how the categories are structured.

We spent several months grouping and categorizing CWEs and could have continued for additional months. We had to stop at some point. There are both root cause and symptom types of CWEs, where root cause types are like "Cryptographic Failure" and "Misconfiguration" contrasted to symptom types like "Sensitive Data Exposure" and "Denial of Service." We decided to focus on the root cause whenever possible as it's more logical for providing identification and remediation guidance. Focusing on the root cause over the symptom isn't a new concept; the Top Ten has been a mix of symptom and root cause. CWEs are also a mix of symptom and root cause; we are simply being more deliberate about it and calling it out. There is an average of 19.6 CWEs per category in this installment, with the lower bounds at 1 CWE for A10:2021-Server-Side Request Forgery (SSRF) to 40 CWEs in A04:2021-Insecure Design. This updated category structure offers additional training benefits as companies can focus on CWEs that make sense for a language/framework.

How the data is used for selecting categories

In 2017, we selected categories by incidence rate to determine likelihood, then ranked them by team discussion based on decades of experience for Exploitability, Detectability (also likelihood), and Technical Impact. For 2021, we want to use data for Exploitability and Impact if possible.

We downloaded OWASP Dependency Check and extracted the CVSS Exploit, and Impact scores grouped by related CWEs. It took a fair bit of research and effort as all the CVEs have CVSSv2 scores, but there are flaws in CVSSv2 that CVSSv3 should address. After a certain point in time, all CVEs are assigned a CVSSv3 score as well. Additionally, the scoring ranges and formulas were updated between CVSSv2 and CVSSv3.

In CVSSv2, both Exploit and Impact could be up to 10.0, but the formula would knock them down to 60% for Exploit and 40% for Impact. In CVSSv3, the theoretical max was limited to 6.0 for Exploit and 4.0 for Impact. With the weighting considered, the Impact scoring shifted higher, almost a point and a half on average in CVSSv3, and exploitability moved nearly half a point lower on average.

There are 125k records of a CVE mapped to a CWE in the NVD data extracted from OWASP Dependency Check, and there are 241 unique CWEs mapped to a CVE. 62k CWE maps have a CVSSv3 score, which is approximately half of the population in the data set.

For the Top Ten, we calculated average exploit and impact scores in the following manner. We grouped all the CVEs with CVSS scores by CWE and weighted both exploit and impact scored by the percentage of the population that had CVSSv3 + the remaining population of CVSSv2 scores to get an overall average. We mapped these averages to the CWEs in the dataset to use as Exploit and Impact scoring for the other half of the risk equation.

Why not just pure statistical data?

The results in the data are primarily limited to what we can test for in an automated fashion. Talk to a seasoned AppSec professional, and they will tell you about stuff they find and trends they see that aren't yet in the data. It takes time for people to develop testing methodologies for certain vulnerability types and then more time for those tests to be automated and run against a large population of applications. Everything we find is looking back in the past and might be missing trends from the last year, which are not present in the data.

Therefore, we only pick eight of ten categories from the data because it's incomplete. The other two categories are from the industry survey. It allows the practitioners on the front lines to vote for what they see as the highest risks that might not be in the data (and may never be expressed in data).

Why incidence rate instead of frequency?

There are three primary sources of data. We identify them as Human-assisted Tooling (HaT), Tool-assisted Human (TaH), and raw Tooling.

Tooling and HaT are high-frequency finding generators. Tools will look for specific vulnerabilities and tirelessly attempt to find every instance of that vulnerability and will generate high finding

counts for some vulnerability types. Look at Cross-Site Scripting, which is typically one of two flavors: it's either a more minor, isolated mistake or a systemic issue. When it's a systemic issue, the finding counts can be in the thousands for an application. This high frequency drowns out most other vulnerabilities found in reports or data. TaH, on the other hand, will find a broader range of vulnerability types but at a much lower frequency due to time constraints. When humans test an application and see something like Cross-Site Scripting, they will typically find three or four instances and stop. They can determine a systemic finding and write it up with a recommendation to fix on an application-wide scale. There is no need (or time) to find every instance.

Suppose we take these two distinct data sets and try to merge them on frequency. In that case, the Tooling and HaT data will drown the more accurate (but broad) TaH data and is a good part of why something like Cross-Site Scripting has been so highly ranked in many lists when the impact is generally low to moderate. It's because of the sheer volume of findings. (Cross-Site Scripting is also reasonably easy to test for, so there are many more tests for it as well). In 2017, we introduced using incidence rate instead to take a fresh look at the data and cleanly merge Tooling and HaT data with TaH data. The incidence rate asks what percentage of the application population had at least one instance of a vulnerability type. We don't care if it was one-off or systemic. That's irrelevant for our purposes; we just need to know how many applications had at least one instance, which helps provide a clearer view of the testing is findings across multiple testing types without drowning the data in high-frequency results.

What is your data collection and analysis process?

We formalized the OWASP Top 10 data collection process at the Open Security Summit in 2017. OWASP Top 10 leaders and the community spent two days working out formalizing a transparent data collection process. The 2021 edition is the second time we have used this methodology. We publish a call for data through social media channels available to us, both project and OWASP. On the OWASP Project page, we list the data elements and structure we are looking for and how to submit them. In the GitHub project, we have example files that serve as templates. We work with organizations as needed to help figure out the structure and mapping to CWEs. We get data from organizations that are testing vendors by trade, bug bounty vendors, and organizations that contribute internal testing data. Once we have the data, we load it together and run a fundamental analysis of what CWEs map to risk categories. There is overlap between some CWEs, and others are very closely related (ex. Cryptographic vulnerabilities). Any decisions related to the raw data submitted are documented and published to be open and transparent with how we normalized the data.

We look at the eight categories with the highest incidence rates for inclusion in the Top 10. We also look at the industry survey results to see which ones may already be present in the data. The top two votes that aren't already present in the data will be selected for the other two places in the Top 10. Once all ten were selected, we applied generalized factors for exploitability and impact; to help rank the Top 10 in order.

Data Factors

There are data factors that are listed for each of the Top 10 Categories, here is what they mean:

- CWEs Mapped: The number of CWEs mapped to a category by the Top 10 team.
- Incidence Rate: Incidence rate is the percentage of applications vulnerable to that CWE from the population tested by that org for that year.
- (Testing) Coverage: The percentage of applications tested by all organizations for a given CWE.
- Weighted Exploit: The Exploit sub-score from CVSSv2 and CVSSv3 scores assigned to CVEs mapped to CWEs, normalized, and placed on a 10pt scale.
- Weighted Impact: The Impact sub-score from CVSSv2 and CVSSv3 scores assigned to CVEs mapped to CWEs, normalized, and placed on a 10pt scale.
- Total Occurrences: Total number of applications found to have the CWEs mapped to a category.
- Total CVEs: Total number of CVEs in the NVD DB that were mapped to the CWEs mapped to a category.

Category Relationships from 2017

There has been a lot of talk about the overlap between the Top Ten risks. By the definition of each (list of CWEs included), there really isn't any overlap. However, conceptually, there can be overlap or interactions based on the higher-level naming. Venn diagrams are many times used to show overlap like this.

The Venn diagram above represents the interactions between the Top Ten 2017 risk categories. While doing so, a couple of essential points became obvious:

1. One could argue that Cross-Site Scripting ultimately belongs within Injection as it's essentially Content Injection. Looking at the 2021 data, it became even more evident that XSS needed to move into Injection.
2. The overlap is only in one direction. We will often classify a vulnerability by the end manifestation or "symptom," not the (potentially deep) root cause. For instance, "Sensitive Data Exposure" may have been the result of a "Security Misconfiguration"; however, you won't see it in the other direction. As a result, arrows are drawn in the interaction zones to indicate which direction it occurs.
3. Sometimes these diagrams are drawn with everything in A06:2021 Using Components with Known Vulnerabilities. While some of these risk categories may be the root cause of third-party vulnerabilities, they are generally managed differently and with different responsibilities. The other types are typically representing first-party risks.

Thank you to our data contributors

The following organizations (along with some anonymous donors) kindly donated data for over 500,000 applications to make this the largest and most comprehensive application security data set. Without you, this would not be possible.

- AppSec Labs
- Cobalt.io
- Contrast Security
- GitLab
- HackerOne
- HCL Technologies
- Micro Focus
- PenTest-Tools
- Probely
- Sqreen
- Veracode
- WhiteHat (NTT)

How to use the OWASP Top 10 as a standard

The OWASP Top 10 is primarily an awareness document. However, this has not stopped organizations using it as a de facto industry AppSec standard since its inception in 2003. If you want to use the OWASP Top 10 as a coding or testing standard, know that it is the bare minimum and just a starting point.

One of the difficulties of using the OWASP Top 10 as a standard is that we document appsec risks, and not necessarily easily testable issues. For example, A04:2021-Insecure Design is beyond the scope of most forms of testing. Another example is testing in place, in use, and effective logging and monitoring can only be done with interviews and requesting a sampling of effective incident responses. A static code analysis tool can look for the absence of logging, but it might be impossible to determine if business logic or access control is logging critical security breaches. Penetration testers may only be able to determine that they have invoked incident response in a test environment, which are rarely monitored in the same way as production.

Here are our recommendations for when it is appropriate to use the OWASP Top 10:

Use Case	**OWASP Top 10 2021**	**OWASP Application Security Verification Standard**
Awareness	Yes	
Training	Entry level	Comprehensive
Design and architecture	Occasionally	Yes
Coding standard	Bare minimum	Yes
Secure Code review	Bare minimum	Yes
Peer review checklist	Bare minimum	Yes
Unit testing	Occasionally	Yes
Integration testing	Occasionally	Yes
Penetration testing	Bare minimum	Yes

Use Case	OWASP Top 10 2021	OWASP Application Security Verification Standard
Tool support	Bare minimum	Yes
Secure Supply Chain	Occasionally	Yes

We would encourage anyone wanting to adopt an application security standard to use the OWASP Application Security Verification Standard (ASVS), as it's designed to be verifiable and tested, and can be used in all parts of a secure development lifecycle.

The ASVS is the only acceptable choice for tool vendors. Tools cannot comprehensively detect, test, or protect against the OWASP Top 10 due to the nature of several of the OWASP Top 10 risks, with reference to A04:2021-Insecure Design. OWASP discourages any claims of full coverage of the OWASP Top 10, because it's simply untrue.

How to start an AppSec Program with the OWASP Top 10

Previously, the OWASP Top 10 was never designed to be the basis for an AppSec program. However, it's essential to start somewhere for many organizations just starting out on their application security journey. The OWASP Top 10 2021 is a good start as a baseline for checklists and so on, but it's not in itself sufficient.

Stage 1. Identify the gaps and goals of your appsec program

Many Application Security (AppSec) programs try to run before they can crawl or walk. These efforts are doomed to failure. We strongly encourage CISOs and AppSec leadership to use OWASP Software Assurance Maturity Model (SAMM) [https://owaspsamm.org] to identify weaknesses and areas for improvement over a 1-3 year period. The first step is to evaluate where you are now, identify the gaps in governance, design, implementation, verification, and operations you need to resolve immediately versus those that can wait, and prioritize implementing or improving the fifteen OWASP SAMM security practices. OWASP SAMM can help you build and measure improvements in your software assurance efforts.

Stage 2. Plan for a paved road secure development lifecycle

Traditionally the preserve of so-called "unicorns," the paved road concept is the easiest way to make the most impact and scale AppSec resources with development team velocity, which only increases every year.

The paved road concept is "the easiest way is also the most secure way" and should involve a culture of deep partnerships between the development team and the security team, preferably such that they are one and the same team. The paved road aims to continuously improve, measure, detect and replace insecure alternatives by having an enterprise-wide library of drop-in secured replacements, with tooling to help see where improvements can be made by adopting the paved road. This allows existing development tools to report on insecure builds and help development teams self-correct away from insecure alternatives.

The paved road might seem a lot to take in, but it should be built incrementally over time. There are other forms of appsec programs out there, notably the Microsoft Agile Secure Development Lifecycle. Not every appsec program methodology suits every business.

Stage 3. Implement the paved road with your development teams

Paved roads are built with the consent and direct involvement of the relevant development and operations teams. The paved road should be aligned strategically with the business and help deliver more secure applications faster. Developing the paved road should be a holistic exercise covering the entire enterprise or application ecosystem, not a per-app band-aid, as in the old days.

Stage 4. Migrate all upcoming and existing applications to the paved road

Add paved road detection tools as you develop them and provide information to development teams to improve the security of their applications by how they can directly adopt elements of the paved road. Once an aspect of the paved road has been adopted, organizations should implement continuous integration checks that inspect existing code and check-ins that use prohibited alternatives and warn or reject the build or check-in. This prevents insecure options from creeping into code over time, preventing technical debt and a defective insecure application. Such warnings should link to the secure alternative, so the development team is given the correct answer immediately. They can refactor and adopt the paved road component quickly.

Stage 5. Test that the paved road has mitigated the issues found in the OWASP Top 10

Paved road components should address a significant issue with the OWASP Top 10, for example, how to automatically detect or fix vulnerable components, or a static code analysis IDE plugin to detect injections or even better a library that is known safe against injection, such as React or Vue. The more of these secure drop-in replacements provided to teams, the better. A vital task of the appsec team is to ensure that the security of these components is continuously evaluated and improved. Once they are improved, some form of communication pathway with consumers of the component should indicate that an upgrade should occur, preferably automatically, but if not, as least highlighted on a dashboard or similar.

Stage 6. Build your program into a mature AppSec program

You must not stop at the OWASP Top 10. It only covers 10 risk categories. We strongly encourage organizations to adopt the Application Security Verification Standard and progressively add paved road components and tests for Level 1, 2, and 3, depending on the developed applications' risk level.

Going beyond

All great AppSec programs go beyond the bare minimum. Everyone must keep going if we're ever going to get on top of appsec vulnerabilities.

- **Conceptual integrity**. Mature AppSec programs must contain some concept of security architecture, whether a formal cloud or enterprise security architecture or threat modeling
- **Automation and scale**. Mature AppSec programs try to automate as much of their deliverables as possible, using scripts to emulate complex penetration testing steps, static code analysis tools directly available to the development teams, assisting dev teams in building appsec unit and integration tests, and more.
- **Culture**. Mature AppSec programs try to build out the insecure design and eliminate the technical debt of existing code by being a part of the development team and not to the side. AppSec teams who see development teams as "us" and "them" are doomed to failure.
- **Continuous improvement**. Mature AppSec programs look to constantly improve. If something is not working, stop doing it. If something is clunky or not scalable, work to improve it. If something is not being used by the development teams and has no or limited impact, do something different. Just because we've done testing like desk checks since the 1970s doesn't mean it's a good idea. Measure, evaluate, and then build or improve.

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
- Cutting edge research
- Standard security controls and libraries
- Complete books on application security testing, secure code development, and secure code review
- Presentations and videos
- Cheat sheets on many common topics
- Chapters meetings
- Events, training, and conferences.
- Google Groups

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

All 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, and 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

A01:2021 – Broken Access Control

Factors

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Max Coverage	Avg Coverage	Avg Weighted Exploit	Avg Weighted Impact
34	55.97%	3.81%	94.55%	47.72%	6.92	5.93

Overview

Moving up from the fifth position, 94% of applications were tested for some form of broken access control. Notable CWEs included are *CWE-200: Exposure of Sensitive Information to an Unauthorized Actor*, *CWE-201: Exposure of Sensitive Information Through Sent Data*, and *CWE-352: Cross-Site Request Forgery*.

Description

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 the user's limits. 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 user's record, permitting 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 JSON Web Token (JWT) access control token, or a cookie or hidden field manipulated to elevate privileges or abusing JWT invalidation.
- CORS misconfiguration allows unauthorized API access.
- Force browsing to authenticated pages as an unauthenticated user or to privileged pages as a standard user. Accessing API with missing access controls for POST, PUT and DELETE.

How to Prevent

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

- Except for public resources, deny by default.
- Implement access control mechanisms once and re-use them throughout the application, including minimizing CORS usage.
- Model access controls should enforce record ownership rather than accepting that the user can create, read, update, or delete any record.
- Unique application business limit requirements should be enforced by domain models.
- Disable web server directory listing and ensure file metadata (e.g., .git) and backup files are not present within web roots.
- Log access control failures, alert admins when appropriate (e.g., repeated failures).
- Rate limit API and controller access to minimize the harm from automated attack tooling.
- JWT tokens should be invalidated on the server after logout.

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 browser's 'acct' parameter to send whatever account number they want. If not correctly verified, the attacker can access any user's account.

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

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

```
https://example.com/app/getappInfo

https://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 Proactive Controls: Enforce Access Controls
- OWASP Application Security Verification Standard: V4 Access Control
- OWASP Testing Guide: Authorization Testing
- OWASP Cheat Sheet: Access Control
- PortSwigger: Exploiting CORS misconfiguration

List of Mapped CWEs

CWE-22 Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

CWE-23 Relative Path Traversal

CWE-35 Path Traversal: '.../...//'

CWE-59 Improper Link Resolution Before File Access ('Link Following')

CWE-200 Exposure of Sensitive Information to an Unauthorized Actor

CWE-201 Exposure of Sensitive Information Through Sent Data

CWE-219 Storage of File with Sensitive Data Under Web Root

CWE-264 Permissions, Privileges, and Access Controls (should no longer be used)

CWE-275 Permission Issues

CWE-276 Incorrect Default Permissions

CWE-284 Improper Access Control

CWE-285 Improper Authorization

CWE-352 Cross-Site Request Forgery (CSRF)

CWE-359 Exposure of Private Personal Information to an Unauthorized Actor

CWE-377 Insecure Temporary File

CWE-402 Transmission of Private Resources into a New Sphere ('Resource Leak')

CWE-425 Direct Request ('Forced Browsing')

CWE-441 Unintended Proxy or Intermediary ('Confused Deputy')

CWE-497 Exposure of Sensitive System Information to an Unauthorized Control Sphere

CWE-538 Insertion of Sensitive Information into Externally-Accessible File or Directory

CWE-540 Inclusion of Sensitive Information in Source Code

CWE-548 Exposure of Information Through Directory Listing

CWE-552 Files or Directories Accessible to External Parties

CWE-566 Authorization Bypass Through User-Controlled SQL Primary Key

CWE-601 URL Redirection to Untrusted Site ('Open Redirect')

CWE-639 Authorization Bypass Through User-Controlled Key

CWE-651 Exposure of WSDL File Containing Sensitive Information

CWE-668 Exposure of Resource to Wrong Sphere

CWE-706 Use of Incorrectly-Resolved Name or Reference

CWE-862 Missing Authorization

CWE-863 Incorrect Authorization

CWE-913 Improper Control of Dynamically-Managed Code Resources

CWE-922 Insecure Storage of Sensitive Information

CWE-1275 Sensitive Cookie with Improper SameSite Attribute

A02:2021 – Cryptographic Failures

Factors

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Max Coverage	Avg Coverage	Avg Weighted Exploit	Avg Weighted Impact
29	46.44%	4.49%	79.33%	34.85%	7.29	6.81

Overview

Shifting up one position to #2, previously known as *Sensitive Data Exposure*, which is more of a broad symptom rather than a root cause, the focus is on failures related to cryptography (or lack thereof). Which often lead to exposure of sensitive data. Notable CWEs included are *CWE-259: Use of Hard-coded Password*, *CWE-327: Broken or Risky Crypto Algorithm*, and *CWE-331 Insufficient Entropy* .

Description

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

- Is any data transmitted in clear text? This concerns protocols such as HTTP, SMTP, and FTP. External internet traffic is hazardous. Verify all internal traffic, e.g., between load balancers, web servers, or back-end systems.
- 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?
- Does the user agent (e.g., app, mail client) not verify if the received server certificate is valid?

See ASVS Crypto (V7), Data Protection (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 an application. Identify which data is sensitive according to privacy laws, regulatory requirements, or business needs.
- Apply controls as per the classification.
- Don't store sensitive data unnecessarily. Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data that is not retained cannot be stolen.
- Make sure to encrypt all sensitive data at rest.
- Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management.
- Encrypt all data in transit with secure protocols such as TLS with perfect forward secrecy (PFS) ciphers, cipher prioritization by the server, and secure parameters. Enforce encryption using directives like HTTP Strict Transport Security (HSTS).
- Disable caching for response that contain sensitive data.
- Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2, scrypt, bcrypt or PBKDF2.
- Verify independently the effectiveness of configuration and 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 a 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 supports weak encryption. An attacker monitors network traffic (e.g., at an insecure wireless network), downgrades connections from HTTPS to HTTP, intercepts requests, 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 they could alter all transported data, e.g., the recipient of a money transfer.

Scenario #3: The password database uses unsalted or simple 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. Hashes generated by simple or fast hash functions may be cracked by GPUs, even if they were salted.

References

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

List of Mapped CWEs

CWE-261 Weak Encoding for Password

CWE-296 Improper Following of a Certificate's Chain of Trust

CWE-310 Cryptographic Issues

CWE-319 Cleartext Transmission of Sensitive Information

CWE-321 Use of Hard-coded Cryptographic Key

CWE-322 Key Exchange without Entity Authentication

CWE-323 Reusing a Nonce, Key Pair in Encryption

CWE-324 Use of a Key Past its Expiration Date

CWE-325 Missing Required Cryptographic Step

CWE-326 Inadequate Encryption Strength

CWE-327 Use of a Broken or Risky Cryptographic Algorithm

CWE-328 Reversible One-Way Hash

CWE-329 Not Using a Random IV with CBC Mode

CWE-330 Use of Insufficiently Random Values

CWE-331 Insufficient Entropy

CWE-335 Incorrect Usage of Seeds in Pseudo-Random Number Generator (PRNG)

CWE-336 Same Seed in Pseudo-Random Number Generator (PRNG)

CWE-337 Predictable Seed in Pseudo-Random Number Generator (PRNG)

CWE-338 Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)

CWE-340 Generation of Predictable Numbers or Identifiers

CWE-347 Improper Verification of Cryptographic Signature

CWE-523 Unprotected Transport of Credentials

CWE-720 OWASP Top Ten 2007 Category A9 - Insecure Communications

CWE-757 Selection of Less-Secure Algorithm During Negotiation ('Algorithm Downgrade')

CWE-759 Use of a One-Way Hash without a Salt

CWE-760 Use of a One-Way Hash with a Predictable Salt

CWE-780 Use of RSA Algorithm without OAEP

CWE-818 Insufficient Transport Layer Protection

CWE-916 Use of Password Hash With Insufficient Computational Effort

A03:2021 – Injection

Factors

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Max Coverage	Avg Coverage	Avg Weighted Exploit	Avg Weighted Impact
33	19.09%	3.37%	94.04%	47.90%	7.25	7.15

Overview

Injection slides down to the third position. 94% of the applications were tested for some form of injection. Notable CWEs included are *CWE-79: Cross-site Scripting*, *CWE-89: SQL Injection*, and *CWE-73: External Control of File Name or Path*.

Description

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures.

Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Source code review is the best method of detecting if applications are vulnerable to injections. Automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs is strongly encouraged. Organizations can include the static source (SAST) and dynamic application test (DAST) tools into the CI/CD pipeline to identify introduced injection flaws before production deployment.

How to Prevent

- Preventing injection requires keeping data separate from commands and queries.
- The preferred option is to use a safe API, which avoids using the interpreter entirely, provides a parameterized interface, or migrates to Object Relational Mapping Tools (ORMs).
- Note: Even 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().
- Use positive or "whitelist" server-side input validation. 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.
- Note: SQL structures 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 their 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 or delete data or even invoke stored procedures.

References

- OWASP Proactive Controls: Secure Database Access
- OWASP ASVS: V5 Input Validation and Encoding
- OWASP Testing Guide: SQL Injection, Command Injection, and ORM Injection
- OWASP Cheat Sheet: Injection Prevention
- OWASP Cheat Sheet: SQL Injection Prevention
- OWASP Cheat Sheet: Injection Prevention in Java
- OWASP Cheat Sheet: Query Parameterization
- OWASP Automated Threats to Web Applications – OAT-014
- PortSwigger: Server-side template injection

List of Mapped CWEs

CWE-20 Improper Input Validation

CWE-74 Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')

CWE-75 Failure to Sanitize Special Elements into a Different Plane (Special Element Injection)

CWE-77 Improper Neutralization of Special Elements used in a Command ('Command Injection')

CWE-78 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

CWE-79 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

CWE-80 Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)

CWE-83 Improper Neutralization of Script in Attributes in a Web Page

CWE-87 Improper Neutralization of Alternate XSS Syntax

CWE-88 Improper Neutralization of Argument Delimiters in a Command ('Argument Injection')

CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

CWE-90 Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')

CWE-91 XML Injection (aka Blind XPath Injection)

CWE-93 Improper Neutralization of CRLF Sequences ('CRLF Injection')

CWE-94 Improper Control of Generation of Code ('Code Injection')

CWE-95 Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')

CWE-96 Improper Neutralization of Directives in Statically Saved Code ('Static Code Injection')

CWE-97 Improper Neutralization of Server-Side Includes (SSI) Within a Web Page

CWE-98 Improper Control of Filename for Include/Require Statement in PHP Program ('PHP Remote File Inclusion')

CWE-99 Improper Control of Resource Identifiers ('Resource Injection')

CWE-100 Deprecated: Was catch-all for input validation issues

CWE-113 Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')

CWE-116 Improper Encoding or Escaping of Output

CWE-138 Improper Neutralization of Special Elements

CWE-184 Incomplete List of Disallowed Inputs

CWE-470 Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')

CWE-471 Modification of Assumed-Immutable Data (MAID)

CWE-564 SQL Injection: Hibernate

CWE-610 Externally Controlled Reference to a Resource in Another Sphere

CWE-643 Improper Neutralization of Data within XPath Expressions ('XPath Injection')

CWE-644 Improper Neutralization of HTTP Headers for Scripting Syntax

CWE-652 Improper Neutralization of Data within XQuery Expressions ('XQuery Injection')

CWE-917 Improper Neutralization of Special Elements used in an Expression Language Statement ('Expression Language Injection')

A04:2021 – Insecure Design

Factors

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Max Coverage	Avg Coverage	Avg Weighted Exploit	Avg Weighted Impact
40	24.19%	3.00%	77.25%	42.51%	6.46	6.78

Overview

A new category for 2021 focuses on risks related to design and architectural flaws, with a call for more use of threat modeling, secure design patterns, and reference architectures. Notable CWEs include *CWE-209: Generation of Error Message Containing Sensitive Information*, *CWE-256: Unprotected Storage of Credentials*, *CWE-501: Trust Boundary Violation*, and *CWE-522: Insufficiently Protected Credentials*.

Description

Insecure design is a broad category representing many different weaknesses, expressed as “missing or ineffective control design.” Missing insecure design is where a control is absent. For example, imagine code that should be encrypting sensitive data, but there is no method. Ineffective insecure design is where a threat could be realized, but insufficient domain (business) logic validation prevents the action. For example, imagine domain logic that is supposed to process pandemic tax relief based upon income brackets but does not validate that all inputs are correctly signed and provides a much more significant relief benefit than should be granted.

Secure design is a culture and methodology that constantly evaluates threats and ensures that code is robustly designed and tested to prevent known attack methods. Secure design requires a secure development lifecycle, some form of secure design pattern or paved road component library or tooling, and threat modeling.

How to Prevent

- Establish and use a secure development lifecycle with AppSec professionals to help evaluate and design security and privacy-related controls

- Establish and use a library of secure design patterns or paved road ready to use components
- Use threat modeling for critical authentication, access control, business logic, and key flows
- Write unit and integration tests to validate that all critical flows are resistant to the threat model

Example Attack Scenarios

Scenario #1: A credential recovery workflow might include "questions and answers," which is prohibited by NIST 800-63b, the OWASP ASVS, and the OWASP Top 10. Questions and answers cannot be trusted as evidence of identity as more than one person can know the answers, which is why they are prohibited. Such code should be removed and replaced with a more secure design.

Scenario #2: A cinema chain allows group booking discounts and has a maximum of fifteen attendees before requiring a deposit. Attackers could threat model this flow and test if they could book six hundred seats and all cinemas at once in a few requests, causing a massive loss of income.

Scenario #3: A retail chain's e-commerce website does not have protection against bots run by scalpers buying high-end video cards to resell auction websites. This creates terrible publicity for the video card makers and retail chain owners and enduring bad blood with enthusiasts who cannot obtain these cards at any price. Careful anti-bot design and domain logic rules, such as purchases made within a few seconds of availability, might identify inauthentic purchases and rejected such transactions.

References

- [OWASP Cheat Sheet: Secure Design Principles] (TBD)
- NIST – Guidelines on Minimum Standards for Developer Verification of > Software > https://www.nist.gov/system/files/documents/2021/07/09/Developer%20Verification%20of%20Software.pdf

List of Mapped CWEs

CWE-73 External Control of File Name or Path

CWE-183 Permissive List of Allowed Inputs

CWE-209 Generation of Error Message Containing Sensitive Information

CWE-213 Exposure of Sensitive Information Due to Incompatible Policies

CWE-235 Improper Handling of Extra Parameters

CWE-256 Unprotected Storage of Credentials

CWE-257 Storing Passwords in a Recoverable Format

CWE-266 Incorrect Privilege Assignment

CWE-269 Improper Privilege Management

CWE-280 Improper Handling of Insufficient Permissions or Privileges

CWE-311 Missing Encryption of Sensitive Data

CWE-312 Cleartext Storage of Sensitive Information

CWE-313 Cleartext Storage in a File or on Disk

CWE-316 Cleartext Storage of Sensitive Information in Memory

CWE-419 Unprotected Primary Channel

CWE-430 Deployment of Wrong Handler

CWE-434 Unrestricted Upload of File with Dangerous Type

CWE-444 Inconsistent Interpretation of HTTP Requests ('HTTP Request Smuggling')

CWE-451 User Interface (UI) Misrepresentation of Critical Information

CWE-472 External Control of Assumed-Immutable Web Parameter

CWE-501 Trust Boundary Violation

CWE-522 Insufficiently Protected Credentials

CWE-525 Use of Web Browser Cache Containing Sensitive Information

CWE-539 Use of Persistent Cookies Containing Sensitive Information

CWE-579 J2EE Bad Practices: Non-serializable Object Stored in Session

CWE-598 Use of GET Request Method With Sensitive Query Strings

CWE-602 Client-Side Enforcement of Server-Side Security

CWE-642 External Control of Critical State Data

CWE-646 Reliance on File Name or Extension of Externally-Supplied File

CWE-650 Trusting HTTP Permission Methods on the Server Side

CWE-653 Insufficient Compartmentalization

CWE-656 Reliance on Security Through Obscurity

CWE-657 Violation of Secure Design Principles

CWE-799 Improper Control of Interaction Frequency

CWE-807 Reliance on Untrusted Inputs in a Security Decision

CWE-840 Business Logic Errors

CWE-841 Improper Enforcement of Behavioral Workflow

CWE-927 Use of Implicit Intent for Sensitive Communication

CWE-1021 Improper Restriction of Rendered UI Layers or Frames

CWE-1173 Improper Use of Validation Framework

A05:2021 – Security Misconfiguration

Factors

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Max Coverage	Avg Coverage	Avg Weighted Exploit	Avg Weighted Impact
20	19.84%	4.51%	89.58%	44.84%	8.12	6.56

Overview

Moving up from #6 in the previous edition, 90% of applications were tested for some form of misconfiguration. With more shifts into highly configurable software, it's not surprising to see this category move up. Notable CWEs included are *CWE-16 Configuration* and *CWE-611 Improper Restriction of XML External Entity Reference*.

Description

The application might be vulnerable if the application is:

- Missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services.
- Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, or privileges).
- Default accounts and their passwords are still enabled and unchanged.
- Error handling reveals stack traces or other overly informative error messages to users.
- For upgraded systems, the latest security features are disabled or not configured securely.
- The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values.
- The server does not send security headers or directives, or they are not set to secure values.
- The software is out of date or vulnerable (see A06:2021-Vulnerable and Outdated Components).

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 makes it fast and easy to deploy another environment that is appropriately 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 set up a new secure environment.
- A minimal platform without any unnecessary features, components, documentation, and samples. Remove or do not install unused features and frameworks.
- A task to review and update the configurations appropriate to all security notes, updates, and patches as part of the patch management process (see A06:2021-Vulnerable and Outdated Components). Review cloud storage permissions (e.g., S3 bucket permissions).
- A segmented application architecture provides effective and secure separation between components or tenants, with segmentation, containerization, or cloud security groups (ACLs).
- Sending security directives to clients, e.g., Security Headers.
- An automated process to verify the effectiveness of the configurations and settings in all environments.

Example Attack Scenarios

Scenario #1: The application server comes with sample applications not removed from the production server. These sample applications have known security flaws attackers use to compromise the server. Suppose one of these applications is the admin console, and default accounts weren't changed. In that case, the attacker logs in with default passwords and takes over.

Scenario #2: Directory listing is not disabled on the server. An attacker discovers they can simply list directories. The attacker finds and downloads the compiled Java classes, which they decompile and reverse engineer to view the code. The attacker then finds a severe access control flaw in the application.

Scenario #3: The application server's configuration allows detailed error messages, e.g., stack traces, to be returned to users. This potentially exposes sensitive information or underlying flaws such as component versions that are known to be vulnerable.

Scenario #4: A cloud service provider has default sharing permissions open to the Internet by other CSP users. This allows sensitive data stored within cloud storage to be accessed.

References

- OWASP Testing Guide: Configuration Management
- OWASP Testing Guide: Testing for Error Codes
- Application Security Verification Standard V19 Configuration
- NIST Guide to General Server Hardening
- CIS Security Configuration Guides/Benchmarks
- Amazon S3 Bucket Discovery and Enumeration

List of Mapped CWEs

CWE-2 Configuration

CWE-11 ASP.NET Misconfiguration: Creating Debug Binary

CWE-13 ASP.NET Misconfiguration: Password in Configuration File

CWE-15 External Control of System or Configuration Setting

CWE-16 Configuration

CWE-260 Password in Configuration File

CWE-315 Cleartext Storage of Sensitive Information in a Cookie

CWE-520 .NET Misconfiguration: Use of Impersonation

CWE-526 Exposure of Sensitive Information Through Environmental Variables

CWE-537 Java Runtime Error Message Containing Sensitive Information

CWE-541 Inclusion of Sensitive Information in an Include File

CWE-547 Use of Hard-coded, Security-relevant Constants

CWE-611 Improper Restriction of XML External Entity Reference

CWE-614 Sensitive Cookie in HTTPS Session Without 'Secure' Attribute

CWE-756 Missing Custom Error Page

CWE-776 Improper Restriction of Recursive Entity References in DTDs ('XML Entity Expansion')

CWE-942 Overly Permissive Cross-domain Whitelist

CWE-1004 Sensitive Cookie Without 'HttpOnly' Flag

CWE-1032 OWASP Top Ten 2017 Category A6 - Security Misconfiguration

CWE-1174 ASP.NET Misconfiguration: Improper Model Validation

A06:2021 – Vulnerable and Outdated Components

Factors

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Max Coverage	Avg Coverage	Avg Weighted Exploit	Avg Weighted Impact
3	27.96%	8.77%	51.78%	22.47%	5.00	5.00

Overview

It was #2 from the industry survey but also had enough data to make the Top 10 via data. Vulnerable Components are a known issue that we struggle to test and assess risk and is the only category to not have any CVEs mapped to the included CWEs, so a default exploits/impact weight of 5.0 is used. Notable CWEs included are *CWE-1104: Use of Unmaintained Third-Party Components* and the two CWEs from Top 10 2013 and 2017.

Description

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 the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries.
- If you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use.
- If you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure to fixed vulnerabilities.
- If software developers do not test the compatibility of updated, upgraded, or patched libraries.

- If you do not secure the components' configurations (see A05:2021-Security Misconfiguration).

How to Prevent

There should be a patch management 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 (e.g., frameworks, libraries) and their dependencies using tools like versions, OWASP Dependency Check, retire.js, etc. Continuously monitor sources like CVE and NVD for vulnerabilities in the components. Use software composition analysis tools to automate the process. Subscribe to email alerts for security vulnerabilities related to components you use.
- Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component (See A08:2021-Software and Data Integrity Failures).
- Monitor for libraries and components that are unmaintained or do not create security patches for older versions. If patching is not possible, consider deploying a virtual patch to monitor, detect, or protect against the discovered issue.

Every organization must ensure 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., a backdoor in a component). Some example exploitable component vulnerabilities discovered are:

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

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 Heartbleed vulnerability patched in April 2014.

References

- OWASP Application Security Verification Standard: V1 Architecture, design and threat modelling
- OWASP Dependency Check (for Java and .NET libraries)
- OWASP Testing Guide - Map Application Architecture (OTG-INFO-010)
- OWASP Virtual Patching Best Practices
- 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
- https://safecode.org/publication/SAFECode_Software_Integrity_Controls0610.pdf

List of Mapped CWEs

CWE-937 OWASP Top 10 2013: Using Components with Known Vulnerabilities

CWE-1035 2017 Top 10 A9: Using Components with Known Vulnerabilities

CWE-1104 Use of Unmaintained Third Party Components

A07:2021 – Identification and Authentication Failures

Factors

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Max Coverage	Avg Coverage	Avg Weighted Exploit	Avg Weighted Impact
22	14.84%	2.55%	79.51%	45.72%	7.40	6.50

Overview

Previously known as *Broken Authentication*, this category slid down from the second position and now includes CWEs related to identification failures. Notable CWEs included are *CWE-297: Improper Validation of Certificate with Host Mismatch*, *CWE-287: Improper Authentication*, and *CWE-384: Session Fixation*.

Description

Confirmation of the user's identity, authentication, and session management is critical to protect against authentication-related attacks. There may be authentication weaknesses if the application:

- Permits automated attacks such as credential stuffing, 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 ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe.
- Uses plain text, encrypted, or weakly hashed passwords (see A3:2017-Sensitive Data Exposure).
- Has missing or ineffective multi-factor authentication.
- Exposes Session IDs in the URL (e.g., URL rewriting).
- Do not rotate Session IDs after successful login.

- Does not correctly invalidate Session IDs. User sessions or authentication tokens (mainly single sign-on (SSO) tokens) aren't properly invalidated during logout or a period of inactivity.

How to Prevent

- Where possible, implement multi-factor authentication to prevent automated credential stuffing, brute force, and stolen credential reuse attacks.
- Do not ship or deploy with any default credentials, particularly for admin users.
- Implement weak password checks, such as testing new or changed passwords against the top 10,000 worst passwords list.
- Align password length, complexity, and rotation policies with NIST 800-63b'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.
- Limit or increasingly delay failed login attempts. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected.
- Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. Session IDs should not be in the URL, be securely stored, and invalidated after logout, idle, and absolute timeouts.

Example Attack Scenarios

Scenario #1: Credential stuffing, the use of lists of known passwords, is a common attack. Suppose an application does not implement automated threat or credential stuffing protection. In that case, 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 encourage 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: Application session timeouts aren't set correctly. A user uses a public computer to access an application. Instead of selecting "logout," the user simply closes the browser tab and walks away. An attacker uses the same browser an hour later, and the user is still authenticated.

References

- OWASP Proactive Controls: Implement Digital Identity

- OWASP Application Security Verification Standard: V2 authentication
- OWASP Application Security Verification Standard: 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: Session Management
- OWASP Automated Threats Handbook
- NIST 800-63b: 5.1.1 Memorized Secrets

List of Mapped CWEs

CWE-255 Credentials Management Errors

CWE-259 Use of Hard-coded Password

CWE-287 Improper Authentication

CWE-288 Authentication Bypass Using an Alternate Path or Channel

CWE-290 Authentication Bypass by Spoofing

CWE-294 Authentication Bypass by Capture-replay

CWE-295 Improper Certificate Validation

CWE-297 Improper Validation of Certificate with Host Mismatch

CWE-300 Channel Accessible by Non-Endpoint

CWE-302 Authentication Bypass by Assumed-Immutable Data

CWE-304 Missing Critical Step in Authentication

CWE-306 Missing Authentication for Critical Function

CWE-307 Improper Restriction of Excessive Authentication Attempts

CWE-346 Origin Validation Error

CWE-384 Session Fixation

CWE-521 Weak Password Requirements

CWE-613 Insufficient Session Expiration

CWE-620 Unverified Password Change

CWE-640 Weak Password Recovery Mechanism for Forgotten Password

CWE-798 Use of Hard-coded Credentials

CWE-940 Improper Verification of Source of a Communication Channel

CWE-1216 Lockout Mechanism Errors

A08:2021 – Software and Data Integrity Failures

Factors

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Max Coverage	Avg Coverage	Avg Weighted Exploit	Avg Weighted Impact
10	16.67%	2.05%	75.04%	45.35%	6.94	7.94

Overview

A new category for 2021 focuses on making assumptions related to software updates, critical data, and CI/CD pipelines without verifying integrity. One of the highest weighted impacts from CVE/CVSS data. Notable CWEs include *CWE-502: Deserialization of Untrusted Data*, *CWE-829: Inclusion of Functionality from Untrusted Control Sphere*, and *CWE-494: Download of Code Without Integrity Check*.

Description

Software and data integrity failures relate to code and infrastructure that does not protect against integrity violations. For example, where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization. Another form of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline can introduce the potential for unauthorized access, malicious code, or system compromise. Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations.

How to Prevent

- Ensure that unsigned or unencrypted serialized data is not sent to untrusted clients without some form of integrity check or digital signature to detect tampering or replay of the serialized data
- Verify the software or data is from the expected source via signing or similar mechanisms

- Ensure libraries and dependencies, such as npm or Maven, are consuming trusted repositories
- Ensure that a software supply chain security tool, such as OWASP Dependency Check or OWASP CycloneDX, is used to verify that components do not contain known vulnerabilities
- Ensure that your CI/CD pipeline has proper configuration and access control to ensure the integrity of the code flowing through the build and deploy processes.

Example Attack Scenarios

Scenario #1 Insecure Deserialization: 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 the 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 Update without signing: Many home routers, set-top boxes, device firmware, and others do not verify updates via signed firmware. Unsigned firmware is a growing target for attackers and is expected to only get worse. This is a major concern as many times there is no mechanism to remediate other than to fix in a future version and wait for previous versions to age out.

Scenario #3 SolarWinds malicious update: Nation-states have been known to attack update mechanisms, with a recent notable attack being the SolarWinds Orion attack. The company that develops the software had secure build and update integrity processes. Still, these were able to be subverted, and for several months, the firm distributed a highly targeted malicious update to more than 18,000 organizations, of which around 100 or so were affected. This is one of the most far-reaching and most significant breaches of this nature in history.

References

- [OWASP Cheat Sheet: Deserialization](https://www.owasp.org/index.php/Deserialization_Cheat_Sheet)
- [OWASP Cheat Sheet: Software Supply Chain Security]()
- [OWASP Cheat Sheet: Secure build and deployment]()
- [SAFECode Software Integrity Controls](https://safecode.org/publication/SAFECode_Software_Integrity_Controls0610.pdf)
- [A 'Worst Nightmare' Cyberattack: The Untold Story Of The SolarWinds Hack](https://www.npr.org/2021/04/16/985439655/a-worst-nightmare-cyberattack-the-untold-story-of-the-solarwinds-hack)
- https://www.manning.com/books/securing-devops

List of Mapped CWEs

CWE-345 Insufficient Verification of Data Authenticity

CWE-353 Missing Support for Integrity Check

CWE-426 Untrusted Search Path

CWE-494 Download of Code Without Integrity Check

CWE-502 Deserialization of Untrusted Data

CWE-565 Reliance on Cookies without Validation and Integrity Checking

CWE-784 Reliance on Cookies without Validation and Integrity Checking in a Security Decision

CWE-829 Inclusion of Functionality from Untrusted Control Sphere

CWE-830 Inclusion of Web Functionality from an Untrusted Source

CWE-915 Improperly Controlled Modification of Dynamically-Determined Object Attributes

A09:2021 – Security Logging and Monitoring Failures

Factors

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Max Coverage	Avg Coverage	Avg Weighted Exploit	Avg Weighted Impact
4	19.23%	6.51%	53.67%	39.97%	6.87	4.99

Overview

Security logging and monitoring came from the industry survey (#3), up slightly from the tenth position in the OWASP Top 10 2017. Logging and monitoring can be challenging to test, often involving interviews or asking if attacks were detected during a penetration test. There isn't much CVE/CVSS data for this category, but detecting and responding to breaches is critical. Still, it can be very impactful for visibility, incident alerting, and forensics. This category expands beyond *CWE-778 Insufficient Logging* to include *CWE-117 Improper Output Neutralization for Logs*, *CWE-223 Omission of Security-relevant Information*, and *CWE-532 Insertion of Sensitive Information into Log File*.

Description

Returning to the OWASP Top 10 2021, this category is to help detect, escalate, and respond to active breaches. Without logging and monitoring, breaches cannot be detected. Insufficient logging, detection, monitoring, and active response occurs any time:

- Auditable events, such as logins, failed logins, and high-value transactions, are not logged.
- Warnings and errors generate no, inadequate, or unclear log messages.
- Logs of applications and APIs are not monitored for suspicious activity.
- Logs are only stored locally.
- Appropriate alerting thresholds and response escalation processes are not in place or effective.
- Penetration testing and scans by DAST tools (such as OWASP ZAP) do not trigger alerts.

- The application cannot detect, escalate, or alert for active attacks in real-time or near real-time.

You are vulnerable to information leakage by making logging and alerting events visible to a user or an attacker (see A01:2021 – Broken Access Control).

How to Prevent

Developers should implement some or all the following controls, depending on the risk of the application:

- Ensure all login, access control, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts and held for enough time to allow delayed forensic analysis.
- Ensure that logs are generated in a format that log management solutions can easily consume.
- Ensure log data is encoded correctly to prevent injections or attacks on the logging or monitoring systems.
- Ensure high-value transactions have an audit trail with integrity controls to prevent tampering or deletion, such as append-only database tables or similar.
- DevSecOps teams should establish effective monitoring and alerting such that suspicious activities are detected and responded to quickly.
- Establish or adopt an incident response and recovery plan, such as NIST 800-61r2 or later.

There are commercial and open-source application protection frameworks such as the OWASP ModSecurity Core Rule Set, and open-source log correlation software, such as the ELK stack, that feature custom dashboards and alerting.

Example Attack Scenarios

Scenario #1: A childrens' health plan provider's website operator couldn't detect a breach due to a lack of monitoring and logging. An external party informed the health plan provider that an attacker had accessed and modified thousands of sensitive health records of more than 3.5 million children. A post-incident review found that the website developers had not addressed significant vulnerabilities. As there was no logging or monitoring of the system, the data breach could have been in progress since 2013, a period of more than seven years.

Scenario #2: A major Indian airline had a data breach involving more than ten years' worth of personal data of millions of passengers, including passport and credit card data. The data breach occurred at a third-party cloud hosting provider, who notified the airline of the breach after some time.

Scenario #3: A major European airline suffered a GDPR reportable breach. The breach was reportedly caused by payment application security vulnerabilities exploited by attackers, who harvested more than 400,000 customer payment records. The airline was fined 20 million pounds as a result by the privacy regulator.

References

- OWASP Proactive Controls: Implement Logging and Monitoring
- OWASP Application Security Verification Standard: V8 Logging and Monitoring
- OWASP Testing Guide: Testing for Detailed Error Code
- OWASP Cheat Sheet: Logging
- Data Integrity: Recovering from Ransomware and Other Destructive Events
- Data Integrity: Identifying and Protecting Assets Against Ransomware and Other Destructive Events
- Data Integrity: Detecting and Responding to Ransomware and Other Destructive Events

List of Mapped CWEs

CWE-117 Improper Output Neutralization for Logs

CWE-223 Omission of Security-relevant Information

CWE-532 Insertion of Sensitive Information into Log File

CWE-778 Insufficient Logging

A10:2021 – Server-Side Request Forgery (SSRF)

Factors

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Max Coverage	Avg Coverage	Avg Weighted Exploit	Avg Weighted Impact
1	2.72%	2.72%	67.72%	67.72%	8.28	6.72

Overview

This category is added from the industry survey (#1). The data shows a relatively low incidence rate with above average testing coverage and above-average Exploit and Impact potential ratings. As new entries are likely to be a single or small cluster of CWEs for attention and awareness, the hope is that they are subject to focus and can be rolled into a larger category in a future edition.

Description

SSRF flaws occur whenever a web application is fetching a remote resource without validating the user-supplied URL. It allows an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall, VPN, or another type of network ACL.

As modern web applications provide end-users with convenient features, fetching a URL becomes a common scenario. As a result, the incidence of SSRF is increasing. Also, the severity of SSRF is becoming higher due to cloud services and the complexity of architectures.

How to Prevent

Developers can prevent SSRF by implementing some or all the following defense in depth controls:

From Network layer

- Segment remote resource access functionality in separate networks to reduce the impact of SSRF
- Enforce “deny by default” firewall policies or network access control rules to block all but essential intranet traffic

From Application layer:

- Sanitize and validate all client-supplied input data
- Enforce the URL schema, port, and destination with a positive allow list
- Do not send raw responses to clients
- Disable HTTP redirections
- Be aware of the URL consistency to avoid attacks such as DNS rebinding and “time of check, time of use” (TOCTOU) race conditions

Do not mitigate SSRF via the use of a deny list or regular expression. Attackers have payload lists, tools, and skills to bypass deny lists.

Example Attack Scenarios

Attackers can use SSRF to attack systems protected behind web application firewalls, firewalls, or network ACLs, using scenarios such as:

Scenario #1: Port scan internal servers. If the network architecture is unsegmented, attackers can map out internal networks and determine if ports are open or closed on internal servers from connection results or elapsed time to connect or reject SSRF payload connections.

Scenario #2: Sensitive data exposure. Attackers can access local files such as or internal services to gain sensitive information.

Scenario #3: Access metadata storage of cloud services. Most cloud providers have metadata storage such as http://169.254.169.254/. An attacker can read the metadata to gain sensitive information.

Scenario #4: Compromise internal services – The attacker can abuse internal services to conduct further attacks such as Remote Code Execution (RCE) or Denial of Service (DoS).

References

- OWASP - Server-Side Request Forgery Prevention Cheat Sheet

- PortSwigger - Server-side request forgery (SSRF)
- Acunetix - What is Server-Side Request Forgery (SSRF)?
- SSRF bible
- A New Era of SSRF - Exploiting URL Parser in Trending Programming Languages!

List of Mapped CWEs

CWE-918 Server-Side Request Forgery (SSRF)

A11:2021 – Next Steps

By design, the OWASP Top 10 is innately limited to the ten most significant risks. Every OWASP Top 10 has “on the cusp” risks considered at length for inclusion, but in the end, they didn’t make it. No matter how we tried to interpret or twist the data, the other risks were more prevalent and impactful.

Organizations working towards a mature appsec program or security consultancies or tool vendors wishing to expand coverage for their offerings, the following four issues are well worth the effort to identify and remediate.

Code Quality issues

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Max Coverage	Avg Coverage	Avg Weighted Exploit	Avg Weighted Impact
38	49.46%	2.22%	60.85%	23.42%		

- **Description.** Code quality issues include known security defects or patterns, reusing variables for multiple purposes, exposure of sensitive information in debugging output, off-by-one errors, time of check/time of use (TOCTOU) race conditions, unsigned or signed conversion errors, use after free, and more. The hallmark of this section is that they can usually be identified with stringent compiler flags, static code analysis tools, and linter IDE plugins. Modern languages by design eliminated many of these issues, such as Rust’s memory ownership and borrowing concept, Rust’s threading design, and Go’s strict typing and bounds checking.
- **How to prevent**. Enable and use your editor and language’s static code analysis options. Consider using a static code analysis tool. Consider if it might be possible to use or migrate to a language or framework that eliminates bug classes, such as Rust or Go.
- **Example attack scenarios**. An attacker might obtain or update sensitive information by exploiting a race condition using a statically shared variable across multiple threads.
- **References**. TBA

Denial of Service

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Max Coverage	Avg Coverage	Avg Weighted Exploit	Avg Weighted Impact
8	17.54%	4.89%	79.58%	33.26%		

- **Description**. Denial of service is always possible given sufficient resources. However, design and coding practices have a significant bearing on the magnitude of the denial of service. Suppose anyone with the link can access a large file, or a computationally expensive transaction occurs on every page. In that case, denial of service requires less effort to conduct.
- **How to prevent**. Performance test code for CPU, I/O, and memory usage, re-architect, optimize, or cache expensive operations. Consider access controls for larger objects to ensure that only authorized individuals can access huge files or objects or serve them by an edge caching network.
- **Example attack scenarios**. An attacker might determine that an operation takes 5-10 seconds to complete. When running four concurrent threads, the server seems to stop responding. The attacker uses 1000 threads and takes the entire system offline.
- **References**. TBA

Memory Management Errors

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Max Coverage	Avg Coverage	Avg Weighted Exploit	Avg Weighted Impact
14	7.03%	1.16%	56.06%	31.74%		

- **Description**. Web applications tend to be written in managed memory languages, such as Java, .NET, or node.js (JavaScript or TypeScript). However, these languages are written in systems languages that have memory management issues, such as buffer or heap overflows, use after free, integer overflows, and more. There have been many sandbox escapes over the years that prove that just because the web application language is nominally memory “safe,” the foundations are not.
- **How to prevent**. Many modern APIs are now written in memory-safe languages such as Rust or Go. In the case of Rust, memory safety is a crucial feature of the language. For

existing code, the use of strict compiler flags, strong typing, static code analysis, and fuzz testing can be beneficial in identifying memory leaks, memory, and array overruns, and more.

- **Example attack scenarios**. Buffer and heap overflows have been a mainstay of
- **References**. TBA

Security Control Failures

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Max Coverage	Avg Coverage	Avg Weighted Exploit	Avg Weighted Impact
2	11.35%	9.64%	76.60%	45.23%		

- **Description**.
- **How to prevent**.
- **Example attack scenarios**.
- **References**. TBA