Application Security Verification Standard 4.0

Final

March 2019

Table of Contents

Frontispiece

About the Standard

The Application Security Verification Standard is a list of application security requirements or tests that can be used by architects, developers, testers, security professionals, tool vendors, and consumers to define, build, test and verify secure applications.

Copyright and License

Version 4.0.1, March 2019

Project Leads

- Andrew van der Stock
- Daniel Cuthbert
- Jim Manico
- Josh C Grossman
- Mark Burnett

Contributors and Reviewers

- Osama Elnaggar
- Erlend Oftedal
- Serg Belkommen
- David Johansson
- Tonimir Kisasondi
- Ron Perris
- Jason Axley
- Abhay Bhargav
- Benedikt Bauer
- Elar Lang
- ScriptingXSS
- Philippe De Ryck
- Grog's Axle
- Marco Schnüriger
- Jacob Salassi
- Glenn ten Cate
- Anthony Weems
- bschach
- javixeneize
- Dan Cornell
- hello7s
- Lewis Ardern
- Jim Newman
- Stuart Gunter
- Geoff Baskwill
- Talargoni
- Ståle Pettersen
- Kelby Ludwig
- Jason Morrow
- Rogan Dawes

The Application Security Verification Standard is built upon the shoulders of those involved from ASVS 1.0 in 2008 to 3.0 in 2016. Much of the structure and verification items that are still in the ASVS today were originally written by Mike Boberski, Jeff Williams and Dave Wichers, but there are many more contributors. Thank you to all those previously involved. For a comprehensive list of all those who have contributed to earlier versions, please consult each prior version.

If a credit is missing from the 4.0 credit list above, please contact vanderaj@owasp.org or log a ticket at GitHub to be recognized in future 4.x updates.

Preface

Welcome to the Application Security Verification Standard (ASVS) version 4.0. The ASVS is a community-driven effort to establish a framework of security requirements and controls that focus on defining the functional and non-functional security controls required when designing, developing and testing modern web applications and web services.

ASVS v4.0 is the culmination of community effort and industry feedback over the last decade. We have attempted to make it easier to adopt the ASVS for a variety of different use cases throughout any secure software development lifecycle.

We expect that there will most likely never be 100% agreement on the contents of any web application standard, including the ASVS. Risk analysis is always subjective to some extent, which creates a challenge when attempting to generalize in a one-size-fits-all standard. However, we hope that the latest updates made in this version are a step in the right direction, and enhance the concepts introduced in this critical industry standard.

What's new in 4.0

The most significant change in this version is the adoption of the NIST 800-63-3 Digital Identity Guidelines, introducing modern, evidence based, and advanced authentication controls. Although we expect some pushback on aligning with an advanced authentication standard, we feel that it is essential for standards to be aligned, mainly when another well-regarded application security standard is evidence-based.

Information security standards should try to minimize the number of unique requirements, so that complying organizations do not have to decide on competing or incompatible controls. The OWASP Top 10 2017 and now the OWASP Application Security Verification Standard have now aligned with NIST 800-63 for authentication and session management. We encourage other standards-setting bodies to work with us, NIST, and others to come to a generally accepted set of application security controls to maximize security and minimize compliance costs.

ASVS 4.0 has been wholly renumbered from start to finish. The new numbering scheme allowed us to close up gaps from long-vanished chapters, and to allow us to segment longer chapters to minimize the number of controls that a developer or team have to comply. For example, if an application does not use JWT, the entire section on JWT in session management is not applicable.

New in 4.0 is a comprehensive mapping to the Common Weakness Enumeration (CWE), one of the most commonly desired feature requests we've had over the last decade. CWE mapping allows tool manufacturers and those using vulnerability management software to match up results from other tools and previous ASVS versions to 4.0 and later. To make room for the CWE entry, we've had to retire the "Since" column, which as we completely renumbered, makes less sense than in previous versions of the ASVS. Not every item in the ASVS has an associated CWE, and as CWE has a great deal of duplication, we've attempted to use the most commonly used rather than necessarily the closest match. Verification controls are not always mappable to equivalent weaknesses. We welcome ongoing discussion with the CWE community and information security field more generally on closing this gap.

We have worked to comprehensively meet and exceed the requirements for addressing the OWASP Top 10 2017 and the OWASP Proactive Controls 2018. As the OWASP Top 10 2018 is the bare minimum to avoid negligence, we have deliberately made all but specific logging Top 10 requirements Level 1 controls, making it easier for OWASP Top 10 adopters to step up to an actual security standard.

We set out to ensure that the ASVS 4.0 Level 1 is a comprehensive superset of PCI DSS 3.2.1 Sections 6.5, for application design, coding, testing, secure code reviews, and penetration tests. This necessitated covering buffer overflow and unsafe memory operations in V5, and unsafe memory-related compilation flags in V14, in addition to existing industry-leading application and web service verification requirements.

We have completed the shift of the ASVS from monolithic server-side only controls, to providing security controls for all modern applications and APIs. In the days of functional programming, server-less API, mobile, cloud,

containers, CI/CD and DevSecOps, federation and more, we cannot continue to ignore modern application architecture. Modern applications are designed very differently to those built when the original ASVS was released in 2009. The ASVS must always look far into the future so that we provide sound advice for our primary audience - developers. We have clarified or dropped any requirement that assumes that applications are executed on systems owned by a single organization.

Due to the size of the ASVS 4.0, as well as our desire to become the baseline ASVS for all other ASVS efforts, we have retired the mobile section, in favor of the Mobile Application Security Verification Standard (MASVS). The Internet of Things appendix will appear in a future IoT ASVS care of the OWASP Internet of Things Project. We have included an early preview of the IoT ASVS in Appendix C. We thank both the OWASP Mobile Team and OWASP IoT Project Team for their support of the ASVS, and look forward to working with them in the future to provide complementary standards.

Lastly, we have de-duped and retired less impactful controls. Over time, the ASVS started being a comprehensive set of controls, but not all controls are equal at producing secure software. This effort to eliminate low impact items could go further. In a future edition of the ASVS, the Common Weakness Scoring System (CWSS) will help prioritize further those controls which are truly important and those that should be retired.

As of version 4.0, the ASVS will focus solely on being the leading web apps and service standard, covering traditional and modern application architecture, and agile security practices and DevSecOps culture.

Using the ASVS

ASVS has two main goals:

- to help organizations develop and maintain secure applications.
- to allow security service vendors, security tools vendors, and consumers to align their requirements and offerings.

Application Security Verification Levels

The Application Security Verification Standard defines three security verification levels, with each level increasing in depth.

- ASVS Level 1 is for low assurance levels, and is completely penetration testable
- ASVS Level 2 is for applications that contain sensitive data, which requires protection and is the recommended level for most apps
- ASVS Level 3 is for the most critical applications - applications that perform high value transactions, contain sensitive medical data, or any application that requires the highest level of trust.

Each ASVS level contains a list of security requirements. Each of these requirements can also be mapped to security-specific features and capabilities that must be built into software by developers.

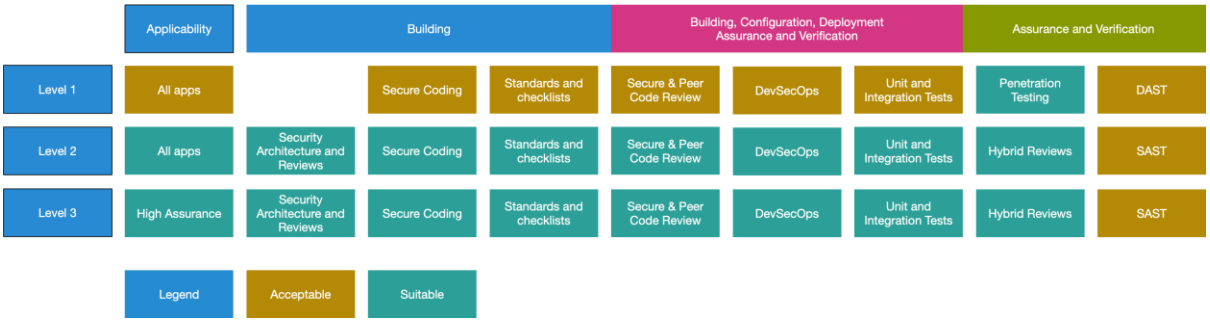

Figure 1 - OWASP Application Security Verification Standard 4.0 Levels

Level 1 is the only level that is completely penetration testable using humans. All others require access to documentation, source code, configuration, and the people involved in the development process. However, even if L1 allows "black box" (no documentation and no source) testing to occur, it is not effective assurance and must stop. Malicious attackers have a great deal of time, most penetration tests are over within a couple of weeks. Defenders need to build in security controls, protect, find and resolve all weaknesses, and detect and respond to malicious actors in a reasonable time. Malicious actors have essentially infinite time and only require a single porous defense, a single weakness, or missing detection to succeed. Black box testing, often performed at the end of development, quickly, or not at all, is completely unable to cope with that asymmetry.

Over the last 30+ years, black box testing has proven over and over again to miss critical security issues that led directly to ever more massive breaches. We strongly encourage the use of a wide range of security assurance and verification, including replacing penetration tests with source code led (hybrid) penetration tests at Level 1, with full access to developers and documentation throughout the development process. Financial regulators do not tolerate external financial audits with no access to the books, sample transactions, or the people performing the controls. Industry and governments must demand the same standard of transparency in the software engineering field.

We strongly encourage the use of security tools, but within the development process itself, such as DAST and SAST tools being used continuously by the build pipeline to find easy to find security issues that should never be present.

Automated tools and online scans are unable to complete more than half of the ASVS without human assistance. If comprehensive test automation for each build is required, then a combination of custom unit and integration tests, along with build initiated online scans are used. Business logic flaws and access control testing is only possible using human assistance. These should be turned into unit and integration tests.

How to use this standard

One of the best ways to use the Application Security Verification Standard is to use it as a blueprint to create a Secure Coding Checklist specific to your application, platform or organization. Tailoring the ASVS to your use cases will increase the focus on the security requirements that are most important to your projects and environments.

Level 1 - First steps, automated, or whole of portfolio view

An application achieves ASVS Level 1 if it adequately defends against application security vulnerabilities that are easy to discover, and included in the OWASP Top 10 and other similar checklists.

Level 1 is the bare minimum that all applications should strive for. It is also useful as a first step in a multi-phase effort or when applications do not store or handle sensitive data and therefore do not need the more rigorous controls of Level 2 or 3. Level 1 controls can be checked either automatically by tools or simply manually without access to source code. We consider Level 1 the minimum required for all applications.

Threats to the application will most likely be from attackers who are using simple and low effort techniques to identify easy-to-find and easy-to-exploit vulnerabilities. This is in contrast to a determined attacker who will spend focused energy to specifically target the application. If data processed by your application has high value, you would rarely want to stop at a Level 1 review.

Level 2 - Most applications

An application achieves ASVS Level 2 (or Standard) if it adequately defends against most of the risks associated with software today.

Level 2 ensures that security controls are in place, effective, and used within the application. Level 2 is typically appropriate for applications that handle significant business-to-business transactions, including those that process healthcare information, implement business-critical or sensitive functions, or process other sensitive assets, or industries where integrity is a critical facet to protect their business, such as the game industry to thwart cheaters and game hacks.

Threats to Level 2 applications will typically be skilled and motivated attackers focusing on specific targets using tools and techniques that are highly practiced and effective at discovering and exploiting weaknesses within applications.

Level 3 - High value, high assurance, or high safety

ASVS Level 3 is the highest level of verification within the ASVS. This level is typically reserved for applications that require significant levels of security verification, such as those that may be found within areas of military, health and safety, critical infrastructure, etc.

Organizations may require ASVS Level 3 for applications that perform critical functions, where failure could significantly impact the organization's operations, and even its survivability. Example guidance on the application of ASVS Level 3 is provided below. An application achieves ASVS Level 3 (or Advanced) if it adequately defends against advanced application security vulnerabilities and also demonstrates principles of good security design.

An application at ASVS Level 3 requires more in depth analysis or architecture, coding, and testing than all the other levels. A secure application is modularized in a meaningful way (to facilitate resiliency, scalability, and most of all, layers of security), and each module (separated by network connection and/or physical instance) takes care of its own security responsibilities (defense in depth), that need to be properly documented. Responsibilities include controls for ensuring confidentiality (e.g. encryption), integrity (e.g. transactions, input validation), availability (e.g. handling load gracefully), authentication (including between systems), non-repudiation, authorization, and auditing (logging).

Applying ASVS in Practice

Different threats have different motivations. Some industries have unique information and technology assets and domain specific regulatory compliance requirements.

Organizations are strongly encouraged to look deeply at their unique risk characteristics based on the nature of their business, and based upon that risk and business requirements determine the appropriate ASVS level.

Assessment and Certification

OWASP's Stance on ASVS Certifications and Trust Marks

OWASP, as a vendor-neutral not-for-profit organization, does not currently certify any vendors, verifiers or software.

All such assurance assertions, trust marks, or certifications are not officially vetted, registered, or certified by OWASP, so an organization relying upon such a view needs to be cautious of the trust placed in any third party or trust mark claiming ASVS certification.

This should not inhibit organizations from offering such assurance services, as long as they do not claim official OWASP certification.

Guidance for Certifying Organizations

The Application Security Verification Standard can be used as an open book verification of the application, including open and unfettered access to key resources such as architects and developers, project documentation, source code, authenticated access to test systems (including access to one or more accounts in each role), particularly for L2 and L3 verifications.

Historically, penetration testing and secure code reviews have included issues “by exception” - that is only failed tests appear in the final report. A certifying organization must include in any report the scope of the verification (particularly if a key component is out of scope, such as SSO authentication), a summary of verification findings, including passed and failed tests, with clear indications of how to resolve the failed tests.

Certain verification requirements may not be applicable to the application under test. For example, if you provide a stateless service layer API without a client implementation to your customers, many of the requirements in V3 Session Management are not directly applicable. In such cases, a certifying organization may still claim full ASVS compliance, but must clearly indicate in any report a reason for non-applicability of such excluded verification requirements.

Keeping detailed work papers, screenshots or movies, scripts to reliably and repeatedly exploit an issue, and electronic records of testing, such as intercepting proxy logs and associated notes such as a cleanup list, is considered standard industry practice and can be really useful as proofs of the findings for the most doubtful developers. It is not sufficient to simply run a tool and report on the failures; this does not (at all) provide sufficient evidence that all issues at a certifying level have been tested and tested thoroughly. In case of dispute, there should be sufficient assurance evidence to demonstrate each and every verified requirement has indeed been tested.

Testing Method

Certifying organizations are free to choose the appropriate testing method(s), but should indicate them in a report.

Depending on the application under test and the verification requirement, different testing methods may be used to gain similar confidence in the results. For example, validating the effectiveness of an application's input verification mechanisms may either be analysed with a manual penetration test or by means of source code analyses.

The Role of Automated Security Testing Tools

The use of automated penetration testing tools is encouraged to provide as much coverage as possible.

It is not possible to fully complete ASVS verification using automated penetration testing tools alone. Whilst a large majority of requirements in L1 can be performed using automated tests, the overall majority of requirements are not amenable to automated penetration testing.

Please note that the lines between automated and manual testing have blurred as the application security industry matures. Automated tools are often manually tuned by experts and manual testers often leverage a wide variety of automated tools.

The Role of Penetration Testing

In version 4.0, we decided to make L1 completely penetration testable without access to source code, documentation, or developers. Two logging items, which are required to comply with OWASP Top 10 2017 A10, will require interviews, screenshots or other evidence collection, just as they do in the OWASP Top 10 2017. However, testing without access to necessary information is not an ideal method of security verification, as it misses out on the possibility of reviewing the source, identifying threats and missing controls, and performing a far more thorough test in a shorter timeframe.

Where possible, access to developers, documentation, code, and access to a test application with non-production data, is required when performing a L2 or L3 Assessment. Penetration testing done at these levels requires this level of access, which we call "hybrid reviews" or "hybrid penetration tests".

Other uses for the ASVS

Aside from being used to assess the security of an application, we have identified a number of other potential uses for the ASVS.

As Detailed Security Architecture Guidance

One of the more common uses for the Application Security Verification Standard is as a resource for security architects. The Sherwood Applied Business Security Architecture (SABSA) is missing a great deal of information that is necessary to complete a thorough application security architecture review. ASVS can be used to fill in those gaps

by allowing security architects to choose better controls for common problems, such as data protection patterns and input validation strategies.

As a Replacement for Off-the-shelf Secure Coding Checklists

Many organizations can benefit from adopting the ASVS, by choosing one of the three levels, or by forking ASVS and changing what is required for each application risk level in a domain specific way. We encourage this type of forking as long as traceability is maintained, so that if an app has passed requirement 4.1, this means the same thing for forked copies as the standard as it evolves.

As a Guide for Automated Unit and Integration Tests

The ASVS is designed to be highly testable, with the sole exception of architectural and malicious code requirements. By building unit and integration tests that test for specific and relevant fuzz and abuse cases, the application becomes nearly self-verifying with each and every build. For example, additional tests can be crafted for the test suite for a login controller, testing the username parameter for common default usernames, account enumeration, brute forcing, LDAP and SQL injection, and XSS. Similarly, a test on the password parameter should include common passwords, password length, null byte injection, removing the parameter, XSS, and more.

For Secure Development Training

ASVS can also be used to define characteristics of secure software. Many “secure coding” courses are simply ethical hacking courses with a light smear of coding tips. This may not necessarily help developers to write more secure code. Instead, secure development courses can use the ASVS with a strong focus on the proactive controls found in the ASVS, rather than the Top 10 negative things not to do.

As a Driver for Agile Application Security

ASVS can be used in an agile development process as a framework to define specific tasks that need to be implemented by the team to have a secure product. One approach might be: Starting with Level 1, verify the specific application or system according to ASVS requirements for the specified level, find what controls are missing and raise specific tickets/tasks in the backlog. This helps with prioritization of specific tasks (or grooming), and makes security visible in the agile process. This can also be used to prioritize auditing and reviewing tasks in the organization, where a specific ASVS requirement can be a driver for review, refactor or auditing for a specific team member and visible as "debt" in the backlog that needs to be eventually done.

As a Framework for Guiding the Procurement of Secure Software

ASVS is a great framework to help with secure software procurement or procurement of custom development services. The buyer can simply set a requirement that the software they wish to procure must be developed at ASVS level X, and request that the seller proves that the software satisfies ASVS level X. This works well when combined with the OWASP Secure Software Contract Annex

V1: Architecture, Design and Threat Modeling Requirements

Control Objective

Security architecture has almost become a lost art in many organizations. The days of the enterprise architect have passed in the age of DevSecOps. The application security field must catch up and adopt agile security principles while re-introducing leading security architecture principles to software practitioners. Architecture is not an implementation, but a way of thinking about a problem that has potentially many different answers, and no one single "correct" answer. All too often, security is seen as inflexible and demanding that developers fix code in a particular way, when the developers may know a much better way to solve the problem. There is no single, simple solution for architecture, and to pretend otherwise is a disservice to the software engineering field.

A specific implementation of a web application is likely to be revised continuously throughout its lifetime, but the overall architecture will likely rarely change but evolve slowly. Security architecture is identical - we need authentication today, we will require authentication tomorrow, and we will need it five years from now. If we make sound decisions today, we can save a lot of effort, time, and money if we select and re-use architecturally compliant solutions. For example, a decade ago, multifactor authentication was rarely implemented.

If developers had invested in a single, secure identity provider model, such as SAML federated identity, the identity provider could be updated to incorporate new requirements such as NIST 800-63 compliance, while not changing the interfaces of the original application. If many applications shared the same security architecture and thus that same component, they all benefit from this upgrade at once. However, SAML will not always remain as the best or most suitable authentication solution - it might need to be swapped out for other solutions as requirements change. Changes like this are either complicated, so costly as to necessitate a complete re-write, or outright impossible without security architecture.

In this chapter, the ASVS covers off the primary aspects of any sound security architecture: availability, confidentiality, processing integrity, non-repudiation, and privacy. Each of these security principles must be built in and be innate to all applications. It is critical to "shift left", starting with developer enablement with secure coding checklists, mentoring and training, coding and testing, building, deployment, configuration, and operations, and finishing with follow up independent testing to assure that all of the security controls are present and functional. The last step used to be everything we did as an industry, but that is no longer sufficient when developers push code into production tens or hundreds of times a day. Application security professionals must keep up with agile techniques, which means adopting developer tools, learning to code, and working with developers rather than criticizing the project months after everyone else has moved on.

V1.1 Secure Software Development Lifecycle Requirements

#	Description	L1	L2	L3	CWE
1.1.1	Verify the use of a secure software development lifecycle that addresses security in all stages of development. (C1)		✓	✓	
1.1.2	Verify the use of threat modeling for every design change or sprint planning to identify threats, plan for countermeasures, facilitate appropriate risk responses, and guide security testing.		✓	✓	1053
1.1.3	Verify that all user stories and features contain functional security constraints, such as "As a user, I should be able to view and edit my profile. I should not be able to view or edit anyone else's profile"		✓	✓	1110
1.1.4	Verify documentation and justification of all the application's trust boundaries, components, and significant data flows.		✓	✓	1059

#	Description	L1	L2	L3	CWE
1.1.5	Verify definition and security analysis of the application's high-level architecture and all connected remote services. (C1)		✓	✓	1059
1.1.6	Verify implementation of centralized, simple (economy of design), vetted, secure, and reusable security controls to avoid duplicate, missing, ineffective, or insecure controls. (C10)		✓	✓	637
1.1.7	Verify availability of a secure coding checklist, security requirements, guideline, or policy to all developers and testers.		✓	✓	637

V1.2 Authentication Architectural Requirements

When designing authentication, it doesn't matter if you have strong hardware enabled multi-factor authentication if an attacker can reset an account by calling a call center and answering commonly known questions. When proofing identity, all authentication pathways must have the same strength.

#	Description	L1	L2	L3	CWE
1.2.1	Verify the use of unique or special low-privilege operating system accounts for all application components, services, and servers. (C3)		✓	✓	250
1.2.2	Verify that communications between application components, including APIs, middleware and data layers, are authenticated. Components should have the least necessary privileges needed. (C3)		✓	✓	306
1.2.3	Verify that the application uses a single vetted authentication mechanism that is known to be secure, can be extended to include strong authentication, and has sufficient logging and monitoring to detect account abuse or breaches.		✓	✓	306
1.2.4	Verify that all authentication pathways and identity management APIs implement consistent authentication security control strength, such that there are no weaker alternatives per the risk of the application.		✓	✓	306

V1.3 Session Management Architectural Requirements

This is a placeholder for future architectural requirements.

V1.4 Access Control Architectural Requirements

#	Description	L1	L2	L3	CWE
1.4.1	Verify that trusted enforcement points such as at access control gateways, servers, and serverless functions enforce access controls. Never enforce access controls on the client.		✓	✓	602
1.4.2	Verify that the chosen access control solution is flexible enough to meet the application's needs.		✓	✓	284
1.4.3	Verify enforcement of the principle of least privilege in functions, data files, URLs, controllers, services, and other resources. This implies protection against spoofing and elevation of privilege.		✓	✓	272

#	Description	L1	L2	L3	CWE
1.4.4	Verify the application uses a single and well-vetted access control mechanism for accessing protected data and resources. All requests must pass through this single mechanism to avoid copy and paste or insecure alternative paths. (C7)		✓	✓	284
1.4.5	Verify that attribute or feature-based access control is used whereby the code checks the user's authorization for a feature/data item rather than just their role. Permissions should still be allocated using roles. (C7)		✓	✓	275

V1.5 Input and Output Architectural Requirements

In 4.0, we have moved away from the term "server-side" as a loaded trust boundary term. The trust boundary is still concerning - making decisions on untrusted browsers or client devices is bypassable. However, in mainstream architectural deployments today, the trust enforcement point has dramatically changed. Therefore, where the term "trusted service layer" is used in the ASVS, we mean any trusted enforcement point, regardless of location, such as a microservice, serverless API, server-side, a trusted API on a client device that has secure boot, partner or external APIs, and so on.

#	Description	L1	L2	L3	CWE
1.5.1	Verify that input and output requirements clearly define how to handle and process data based on type, content, and applicable laws, regulations, and other policy compliance.		✓	✓	1029
1.5.2	Verify that serialization is not used when communicating with untrusted clients. If this is not possible, ensure that adequate integrity controls (and possibly encryption if sensitive data is sent) are enforced to prevent deserialization attacks including object injection.		✓	✓	502
1.5.3	Verify that input validation is enforced on a trusted service layer. (C5)		✓	✓	602
1.5.4	Verify that output encoding occurs close to or by the interpreter for which it is intended. (C4)		✓	✓	116

V1.6 Cryptographic Architectural Requirements

Applications need to be designed with strong cryptographic architecture to protect data assets as per their classification. Encrypting everything is wasteful, not encrypting anything is legally negligent. A balance must be struck, usually during architectural or high level design, design sprints or architectural spikes. Designing cryptography as you go or retrofitting it will inevitably cost much more to implement securely than simply building it in from the start.

Architectural requirements are intrinsic to the entire code base, and thus difficult to unit or integrate test. Architectural requirements require consideration in coding standards, throughout the coding phase, and should be reviewed during security architecture, peer or code reviews, or retrospectives.

#	Description	L1	L2	L3	CWE
1.6.1	Verify that there is an explicit policy for management of cryptographic keys and that a cryptographic key lifecycle follows a key management standard such as NIST SP 800-57.		✓	✓	320
1.6.2	Verify that consumers of cryptographic services protect key material and other secrets by using key vaults or API based alternatives.		✓	✓	320

#	Description	L1	L2	L3	CWE
1.6.3	Verify that all keys and passwords are replaceable and are part of a well-defined process to re-encrypt sensitive data.		✓	✓	320
1.6.4	Verify that symmetric keys, passwords, or API secrets generated by or shared with clients are used only in protecting low risk secrets, such as encrypting local storage, or temporary ephemeral uses such as parameter obfuscation. Sharing secrets with clients is clear-text equivalent and architecturally should be treated as such.		✓	✓	320

V1.7 Errors, Logging and Auditing Architectural Requirements

#	Description	L1	L2	L3	CWE
1.7.1	Verify that a common logging format and approach is used across the system. (C9)		✓	✓	1009
1.7.2	Verify that logs are securely transmitted to a preferably remote system for analysis, detection, alerting, and escalation. (C9)		✓	✓	

V1.8 Data Protection and Privacy Architectural Requirements

#	Description	L1	L2	L3	CWE
1.8.1	Verify that all sensitive data is identified and classified into protection levels.		✓	✓	
1.8.2	Verify that all protection levels have an associated set of protection requirements, such as encryption requirements, integrity requirements, retention, privacy and other confidentiality requirements, and that these are applied in the architecture.		✓	✓	

V1.9 Communications Architectural Requirements

#	Description	L1	L2	L3	CWE
1.9.1	Verify the application encrypts communications between components, particularly when these components are in different containers, systems, sites, or cloud providers. (C3)		✓	✓	319
1.9.2	Verify that application components verify the authenticity of each side in a communication link to prevent person-in-the-middle attacks. For example, application components should validate TLS certificates and chains.		✓	✓	295

V1.10 Malicious Software Architectural Requirements

#	Description	L1	L2	L3	CWE
1.10.1	Verify that a source code control system is in use, with procedures to ensure that check-ins are accompanied by issues or change tickets. The source code control system should have access control and identifiable users to allow traceability of any changes.		✓	✓	284

V1.11 Business Logic Architectural Requirements

#	Description	L1	L2	L3	CWE
1.11.1	Verify the definition and documentation of all application components in terms of the business or security functions they provide.		✓	✓	1059
1.11.2	Verify that all high-value business logic flows, including authentication, session management and access control, do not share unsynchronized state.		✓	✓	362
1.11.3	Verify that all high-value business logic flows, including authentication, session management and access control are thread safe and resistant to time-of-check and time-of-use race conditions.			✓	367

V1.12 Secure File Upload Architectural Requirements

#	Description	L1	L2	L3	CWE
1.12.1	Verify that user-uploaded files are stored outside of the web root.		✓	✓	552
1.12.2	Verify that user-uploaded files - if required to be displayed or downloaded from the application - are served by either octet stream downloads, or from an unrelated domain, such as a cloud file storage bucket. Implement a suitable content security policy to reduce the risk from XSS vectors or other attacks from the uploaded file.		✓	✓	646

V1.13 API Architectural Requirements

This is a placeholder for future architectural requirements.

V1.14 Configuration Architectural Requirements

#	Description	L1	L2	L3	CWE
1.14.1	Verify the segregation of components of differing trust levels through well-defined security controls, firewall rules, API gateways, reverse proxies, cloud-based security groups, or similar mechanisms.		✓	✓	923
1.14.2	Verify that if deploying binaries to untrusted devices makes use of binary signatures, trusted connections, and verified endpoints.		✓	✓	494
1.14.3	Verify that the build pipeline warns of out-of-date or insecure components and takes appropriate actions.		✓	✓	1104
1.14.4	Verify that the build pipeline contains a build step to automatically build and verify the secure deployment of the application, particularly if the application infrastructure is software defined, such as cloud environment build scripts.		✓	✓	
1.14.5	Verify that application deployments adequately sandbox, containerize and/or isolate at the network level to delay and deter attackers from attacking other applications, especially when they are performing sensitive or dangerous actions such as deserialization. (C5)		✓	✓	265

1.14.6	Verify the application does not use unsupported, insecure, or deprecated client-side technologies such as NSAPI plugins, Flash, Shockwave, ActiveX, Silverlight, NACL, or client-side Java applets.	✓	✓	477

References

For more information, see also:

- OWASP Threat Modeling Cheat Sheet
- OWASP Attack Surface Analysis Cheat Sheet
- OWASP Threat modeling
- OWASP Secure SDLC Cheat Sheet
- Microsoft SDL
- NIST SP 800-57

V2: Authentication Verification Requirements

Control Objective

Authentication is the act of establishing, or confirming, someone (or something) as authentic and that claims made by a person or about a device are correct, resistant to impersonation, and prevent recovery or interception of passwords.

When the ASVS was first released, username + password was the most common form of authentication outside of high security systems. Multi-factor authentication (MFA) was commonly accepted in security circles but rarely required elsewhere. As the number of password breaches increased, the idea that usernames are somehow confidential and passwords unknown, rendered many security controls untenable. For example, NIST 800-63 considers usernames and knowledge based authentication (KBA) as public information, SMS and email notifications as "restricted" authenticator types , and passwords as pre-breached. This reality renders knowledge based authenticators, SMS and email recovery, password history, complexity, and rotation controls useless. These controls always have been less than helpful, often forcing users to come up with weak passwords every few months, but with the release of over 5 billion username and password breaches, it's time to move on.

Of all the sections in the ASVS, the authentication and session management chapters have changed the most. Adoption of effective, evidence-based leading practice will be challenging for many, and that's perfectly okay. We have to start the transition to a post-password future now.

NIST 800-63 - Modern, evidence-based authentication standard

NIST 800-63b is a modern, evidence-based standard, and represents the best advice available, regardless of applicability. The standard is helpful for all organizations all over the world but is particularly relevant to US agencies and those dealing with US agencies.

NIST 800-63 terminology can be a little confusing at first, especially if you're only used to username + password authentication. Advancements in modern authentication are necessary, so we have to introduce terminology that will become commonplace in the future, but we do understand the difficulty in understanding until the industry settles on these new terms. We have provided a glossary at the end of this chapter to assist. We have rephrased many requirements to satisfy the intent of the requirement, rather than the letter of the requirement. For example, the ASVS uses the term "password" when NIST uses "memorized secret" throughout this standard.

ASVS V2 Authentication, V3 Session Management, and to a lesser extent, V4 Access Controls have been adapted to be a compliant subset of selected NIST 800-63b controls, focused around common threats and commonly exploited authentication weaknesses. Where full NIST 800-63 compliance is required, please consult NIST 800-63.

Selecting an appropriate NIST AAL Level

The Application Security Verification Standard has tried to map ASVS L1 to NIST AAL1 requirements, L2 to AAL2, and L3 to AAL3. However, the approach of ASVS Level 1 as "essential" controls may not necessarily be the correct AAL level to verify an application or API. For example, if the application is a Level 3 application or has regulatory requirements to be AAL3, Level 3 should be chosen in Sections V2 and V3 Session Management. The choice of NIST compliant authentication assertion level (AAL) should be performed as per NIST 800-63b guidelines as set out in *Selecting AAL* in NIST 800-63b Section 6.2.

Legend

Applications can always exceed the current level's requirements, especially if modern authentication is on an application's roadmap. Previously, the ASVS has required mandatory MFA. NIST does not require mandatory MFA. Therefore, we have used an optional designation in this chapter to indicate where the ASVS encourages but does not require a control. The following keys are used throughout this standard:

Mark	Description

Not required

o Recommended, but not required

✓ Required

V2.1 Password Security Requirements

Passwords, called "Memorized Secrets" by NIST 800-63, include passwords, PINs, unlock patterns, pick the correct kitten or another image element, and passphrases. They are generally considered "something you know", and often used as single factor authenticators. There are significant challenges to the continued use of single-factor authentication, including billions of valid usernames and passwords disclosed on the Internet, default or weak passwords, rainbow tables and ordered dictionaries of the most common passwords.

Applications should strongly encourage users to enrol in multi-factor authentication, and should allow users to re-use tokens they already possess, such as FIDO or U2F tokens, or link to a credential service provider that provides multi-factor authentication.

Credential service providers (CSPs) provide federated identity for users. Users will often have more than one identity with multiple CSPs, such as an enterprise identity using Azure AD, Okta, Ping Identity or Google, or consumer identity using Facebook, Twitter, Google, or WeChat, to name a just few common alternatives. This list is not an endorsement of these companies or services, but simply an encouragement for developers to consider the reality that many users have many established identities. Organizations should consider integrating with existing user identities, as per the risk profile of the CSP's strength of identity proofing. For example, it is unlikely a government organization would accept a social media identity as a login for sensitive systems, as it is easy to create fake or throw away identities, whereas a mobile game company may well need to integrate with major social media platforms to grow their active player base.

#	Description	L1	L2	L3	CWE	NIST §
2.1.1	Verify that user set passwords are at least 12 characters in length. (C6)	✓	✓	✓	521	5.1.1.2
2.1.2	Verify that passwords 64 characters or longer are permitted. (C6)	✓	✓	✓	521	5.1.1.2
2.1.3	Verify that passwords can contain spaces and truncation is not performed. Consecutive multiple spaces MAY optionally be coalesced. (C6)	✓	✓	✓	521	5.1.1.2
2.1.4	Verify that Unicode characters are permitted in passwords. A single Unicode code point is considered a character, so 12 emoji or 64 kanji characters should be valid and permitted.	✓	✓	✓	521	5.1.1.2
2.1.5	Verify users can change their password.	✓	✓	✓	620	5.1.1.2
2.1.6	Verify that password change functionality requires the user's current and new password.	✓	✓	✓	620	5.1.1.2
2.1.7	Verify that passwords submitted during account registration, login, and password change are checked against a set of breached passwords either locally (such as the top 1,000 or 10,000 most common passwords which match the system's password policy) or using an external API. If using an API a zero knowledge proof or other mechanism should be used to ensure that the plain text password is not sent or used in verifying the breach status of the password. If the password is	✓	✓	✓	521	5.1.1.2

#	Description	L1	L2	L3	CWE	NIST §
	breached, the application must require the user to set a new non-breached password. (C6)					
2.1.8	Verify that a password strength meter is provided to help users set a stronger password.	✓	✓	✓	521	5.1.1.2
2.1.9	Verify that there are no password composition rules limiting the type of characters permitted. There should be no requirement for upper or lower case or numbers or special characters. (C6)	✓	✓	✓	521	5.1.1.2
2.1.10	Verify that there are no periodic credential rotation or password history requirements.	✓	✓	✓	263	5.1.1.2
2.1.11	Verify that "paste" functionality, browser password helpers, and external password managers are permitted.	✓	✓	✓	521	5.1.1.2
2.1.12	Verify that the user can choose to either temporarily view the entire masked password, or temporarily view the last typed character of the password on platforms that do not have this as native functionality.	✓	✓	✓	521	5.1.1.2

Note: The goal of allowing the user to view their password or see the last character temporarily is to improve the usability of credential entry, particularly around the use of longer passwords, passphrases, and password managers. Another reason for including the requirement is to deter or prevent test reports unnecessarily requiring organizations to override native platform password field behavior to remove this modern user-friendly security experience.

V2.2 General Authenticator Requirements

Authenticator agility is essential to future-proof applications. Refactor application verifiers to allow additional authenticators as per user preferences, as well as allowing retiring deprecated or unsafe authenticators in an orderly fashion.

NIST considers email and SMS as "restricted" authenticator types, and they are likely to be removed from NIST 800-63 and thus the ASVS at some point the future. Applications should plan a roadmap that does not require the use of email or SMS.

#	Description	L1	L2	L3	CWE	NIST §
2.2.1	Verify that anti-automation controls are effective at mitigating breached credential testing, brute force, and account lockout attacks. Such controls include blocking the most common breached passwords, soft lockouts, rate limiting, CAPTCHA, ever increasing delays between attempts, IP address restrictions, or risk-based restrictions such as location, first login on a device, recent attempts to unlock the account, or similar. Verify that no more than 100 failed attempts per hour is possible on a single account.	✓	✓	✓	307	5.2.2 / 5.1.1.2 / 5.1.4.2 / 5.1.5.2
2.2.2	Verify that the use of weak authenticators (such as SMS and email) is limited to secondary verification and transaction approval and not as a replacement for more secure authentication methods. Verify that stronger methods are offered before weak methods, users are aware of the risks, or that proper measures are in place to limit the risks of account compromise.	✓	✓	✓	304	5.2.10

#	Description	L1	L2	L3	CWE	NIST §
2.2.3	Verify that secure notifications are sent to users after updates to authentication details, such as credential resets, email or address changes, logging in from unknown or risky locations. The use of push notifications - rather than SMS or email - is preferred, but in the absence of push notifications, SMS or email is acceptable as long as no sensitive information is disclosed in the notification.	✓	✓	✓	620	
2.2.4	Verify impersonation resistance against phishing, such as the use of multi-factor authentication, cryptographic devices with intent (such as connected keys with a push to authenticate), or at higher AAL levels, client-side certificates.			✓	308	5.2.5
2.2.5	Verify that where a credential service provider (CSP) and the application verifying authentication are separated, mutually authenticated TLS is in place between the two endpoints.			✓	319	5.2.6
2.2.6	Verify replay resistance through the mandated use of OTP devices, cryptographic authenticators, or lookup codes.			✓	308	5.2.8
2.2.7	Verify intent to authenticate by requiring the entry of an OTP token or user-initiated action such as a button press on a FIDO hardware key.			✓	308	5.2.9

V2.3 Authenticator Lifecycle Requirements

Authenticators are passwords, soft tokens, hardware tokens, and biometric devices. The lifecycle of authenticators is critical to the security of an application - if anyone can self-register an account with no evidence of identity, there can be little trust in the identity assertion. For social media sites like Reddit, that's perfectly okay. For banking systems, a greater focus on the registration and issuance of credentials and devices is critical to the security of the application.

Note: Passwords are not to have a maximum lifetime or be subject to password rotation. Passwords should be checked for being breached, not regularly replaced.

#	Description	L1	L2	L3	CWE	NIST §
2.3.1	Verify system generated initial passwords or activation codes SHOULD be securely randomly generated, SHOULD be at least 6 characters long, and MAY contain letters and numbers, and expire after a short period of time. These initial secrets must not be permitted to become the long term password.	✓	✓	✓	330	5.1.1.2 / A.3
2.3.2	Verify that enrollment and use of subscriber-provided authentication devices are supported, such as a U2F or FIDO tokens.		✓	✓	308	6.1.3
2.3.3	Verify that renewal instructions are sent with sufficient time to renew time bound authenticators.		✓	✓	287	6.1.4

V2.4 Credential Storage Requirements

Architects and developers should adhere to this section when building or refactoring code. This section can only be fully verified using source code review or through secure unit or integration tests. Penetration testing cannot identify any of these issues.

The list of approved one-way key derivation functions is detailed in NIST 800-63 B section 5.1.1.2, and in BSI Kryptographische Verfahren: Empfehlungen und Schlussellängen (2018). The latest national or regional algorithm and key length standards can be chosen in place of these choices.

This section cannot be penetration tested, so controls are not marked as L1. However, this section is of vital importance to the security of credentials if they are stolen, so if forking the ASVS for an architecture or coding guideline or source code review checklist, please place these controls back to L1 in your private version.

#	Description	L1	L2	L3	CWE	NIST §
2.4.1	Verify that passwords are stored in a form that is resistant to offline attacks. Passwords SHALL be salted and hashed using an approved one-way key derivation or password hashing function. Key derivation and password hashing functions take a password, a salt, and a cost factor as inputs when generating a password hash. (C6)		✓	✓	916	5.1.1.2
2.4.2	Verify that the salt is at least 32 bits in length and be chosen arbitrarily to minimize salt value collisions among stored hashes. For each credential, a unique salt value and the resulting hash SHALL be stored. (C6)		✓	✓	916	5.1.1.2
2.4.3	Verify that if PBKDF2 is used, the iteration count SHOULD be as large as verification server performance will allow, typically at least 100,000 iterations. (C6)		✓	✓	916	5.1.1.2
2.4.4	Verify that if bcrypt is used, the work factor SHOULD be as large as verification server performance will allow, typically at least 13. (C6)		✓	✓	916	5.1.1.2
2.4.5	Verify that an additional iteration of a key derivation function is performed, using a salt value that is secret and known only to the verifier. Generate the salt value using an approved random bit generator [SP 800-90Ar1] and provide at least the minimum security strength specified in the latest revision of SP 800-131A. The secret salt value SHALL be stored separately from the hashed passwords (e.g., in a specialized device like a hardware security module).		✓	✓	916	5.1.1.2

Where US standards are mentioned, a regional or local standard can be used in place of or in addition to the US standard as required.

V2.5 Credential Recovery Requirements

#	Description	L1	L2	L3	CWE	NIST §
2.5.1	Verify that a system generated initial activation or recovery secret is not sent in clear text to the user. (C6)	✓	✓	✓	640	5.1.1.2
2.5.2	Verify password hints or knowledge-based authentication (so-called "secret questions") are not present.	✓	✓	✓	640	5.1.1.2
2.5.3	Verify password credential recovery does not reveal the current password in any way. (C6)	✓	✓	✓	640	5.1.1.2
2.5.4	Verify shared or default accounts are not present (e.g. "root", "admin", or "sa").	✓	✓	✓	16	5.1.1.2 / A.3

#	Description	L1	L2	L3	CWE	NIST §
2.5.5	Verify that if an authentication factor is changed or replaced, that the user is notified of this event.	✓	✓	✓	304	6.1.2.3
2.5.6	Verify forgotten password, and other recovery paths use a secure recovery mechanism, such as TOTP or other soft token, mobile push, or another offline recovery mechanism. (C6)	✓	✓	✓	640	5.1.1.2
2.5.7	Verify that if OTP or multi-factor authentication factors are lost, that evidence of identity proofing is performed at the same level as during enrollment.		✓	✓	308	6.1.2.3

V2.6 Look-up Secret Verifier Requirements

Look up secrets are pre-generated lists of secret codes, similar to Transaction Authorization Numbers (TAN), social media recovery codes, or a grid containing a set of random values. These are distributed securely to users. These lookup codes are used once, and once all used, the lookup secret list is discarded. This type of authenticator is considered "something you have".

#	Description	L1	L2	L3	CWE	NIST §
2.6.1	Verify that lookup secrets can be used only once.		✓	✓	308	5.1.2.2
2.6.2	Verify that lookup secrets have sufficient randomness (112 bits of entropy), or if less than 112 bits of entropy, salted with a unique and random 32-bit salt and hashed with an approved one-way hash.		✓	✓	330	5.1.2.2
2.6.3	Verify that lookup secrets are resistant to offline attacks, such as predictable values.		✓	✓	310	5.1.2.2

V2.7 Out of Band Verifier Requirements

In the past, a common out of band verifier would have been an email or SMS containing a password reset link. Attackers use this weak mechanism to reset accounts they don't yet control, such as taking over a person's email account and re-using any discovered reset links. There are better ways to handle out of band verification.

Secure out of band authenticators are physical devices that can communicate with the verifier over a secure secondary channel. Examples include push notifications to mobile devices. This type of authenticator is considered "something you have". When a user wishes to authenticate, the verifying application sends a message to the out of band authenticator via a connection to the authenticator directly or indirectly through a third party service. The message contains an authentication code (typically a random six digit number or a modal approval dialog). The verifying application waits to receive the authentication code through the primary channel and compares the hash of the received value to the hash of the original authentication code. If they match, the out of band verifier can assume that the user has authenticated.

The ASVS assumes that only a few developers will be developing new out of band authenticators, such as push notifications, and thus the following ASVS controls apply to verifiers, such as authentication API, applications, and single sign-on implementations. If developing a new out of band authenticator, please refer to NIST 800-63B § 5.1.3.1.

Unsafe out of band authenticators such as e-mail and VOIP are not permitted. PSTN and SMS authentication are currently "restricted" by NIST and should be deprecated in favor of push notifications or similar. If you need to use telephone or SMS out of band authentication, please see § 5.1.3.3.

#	Description	L1	L2	L3	CWE	NIST §
2.7.1	Verify that clear text out of band (NIST "restricted") authenticators, such as SMS or PSTN, are not offered by default, and stronger alternatives such as push notifications are offered first.	✓	✓	✓	287	5.1.3.2
2.7.2	Verify that the out of band verifier expires out of band authentication requests, codes, or tokens after 10 minutes.	✓	✓	✓	287	5.1.3.2
2.7.3	Verify that the out of band verifier authentication requests, codes, or tokens are only usable once, and only for the original authentication request.	✓	✓	✓	287	5.1.3.2
2.7.4	Verify that the out of band authenticator and verifier communicates over a secure independent channel.	✓	✓	✓	523	5.1.3.2
2.7.5	Verify that the out of band verifier retains only a hashed version of the authentication code.		✓	✓	256	5.1.3.2
2.7.6	Verify that the initial authentication code is generated by a secure random number generator, containing at least 20 bits of entropy (typically a six digital random number is sufficient).		✓	✓	310	5.1.3.2

V2.8 Single or Multi Factor One Time Verifier Requirements

Single factor one time passwords (OTPs) are physical or soft tokens that display a continually changing pseudo-random one time challenge. These devices make phishing (impersonation) difficult, but not impossible. This type of authenticator is considered "something you have". Multi-factor tokens are similar to single factor OTPs, but require a valid PIN code, biometric unlocking, USB insertion or NFC pairing or some additional value (such as transaction signing calculators) to be entered to create the final OTP.

#	Description	L1	L2	L3	CWE	NIST §
2.8.1	Verify that time-based OTPs have a defined lifetime before expiring.	✓	✓	✓	613	5.1.4.2 / 5.1.5.2
2.8.2	Verify that symmetric keys used to verify submitted OTPs are highly protected, such as by using a hardware security module or secure operating system based key storage.		✓	✓	320	5.1.4.2 / 5.1.5.2
2.8.3	Verify that approved cryptographic algorithms are used in the generation, seeding, and verification.		✓	✓	326	5.1.4.2 / 5.1.5.2
2.8.4	Verify that time-based OTP can be used only once within the validity period.		✓	✓	287	5.1.4.2 / 5.1.5.2
2.8.5	Verify that if a time-based multi factor OTP token is re-used during the validity period, it is logged and rejected with secure notifications being sent to the holder of the device.		✓	✓	287	5.1.5.2
2.8.6	Verify physical single factor OTP generator can be revoked in case of theft or other loss. Ensure that revocation is immediately effective across logged in sessions, regardless of location.		✓	✓	613	5.2.1

2.8.7	Verify that biometric authenticators are limited to use only as secondary factors in conjunction with either something you have and something you know.	o	✓		308	5.2.3

V2.9 Cryptographic Software and Devices Verifier Requirements

Cryptographic security keys are smart cards or FIDO keys, where the user has to plug in or pair the cryptographic device to the computer to complete authentication. Verifiers send a challenge nonce to the cryptographic devices or software, and the device or software calculates a response based upon a securely stored cryptographic key.

The requirements for single factor cryptographic devices and software, and multi-factor cryptographic devices and software are the same, as verification of the cryptographic authenticator proves possession of the authentication factor.

#	Description	L1	L2	L3	CWE	NIST §
2.9.1	Verify that cryptographic keys used in verification are stored securely and protected against disclosure, such as using a TPM or HSM, or an OS service that can use this secure storage.		✓	✓	320	5.1.7.2
2.9.2	Verify that the challenge nonce is at least 64 bits in length, and statistically unique or unique over the lifetime of the cryptographic device.		✓	✓	330	5.1.7.2
2.9.3	Verify that approved cryptographic algorithms are used in the generation, seeding, and verification.		✓	✓	327	5.1.7.2

V2.10 Service Authentication Requirements

This section is not penetration testable, so does not have any L1 requirements. However, if used in an architecture, coding or secure code review, please assume that software (just as Java Key Store) is the minimum requirement at L1. Clear text storage of secrets is not acceptable under any circumstances.

#	Description	L1	L2	L3	CWE	NIST §
2.10.1	Verify that integration secrets do not rely on unchanging passwords, such as API keys or shared privileged accounts.		OS assisted	HSM	287	5.1.1.1
2.10.2	Verify that if passwords are required, the credentials are not a default account.		OS assisted	HSM	255	5.1.1.1
2.10.3	Verify that passwords are stored with sufficient protection to prevent offline recovery attacks, including local system access.		OS assisted	HSM	522	5.1.1.1
2.10.4	Verify passwords, integrations with databases and third-party systems, seeds and internal secrets, and API keys are managed securely and not included in the source code or stored within source code repositories. Such storage SHOULD resist offline attacks. The use of a secure software key store (L1), hardware trusted platform module (TPM), or a hardware security module (L3) is recommended for password storage.		OS assisted	HSM	798	

Additional US Agency Requirements

US Agencies have mandatory requirements concerning NIST 800-63. The Application Security Verification Standard has always been about the 80% of controls that apply to nearly 100% of apps, and not the last 20% of advanced

controls or those that have limited applicability. As such, the ASVS is a strict subset of NIST 800-63, especially for IAL1/2 and AAL1/2 classifications, but is not sufficiently comprehensive, particularly concerning IAL3/AAL3 classifications.

We strongly urge US government agencies to review and implement NIST 800-63 in its entirety.

Glossary of terms

Term	Meaning
CSP	Credential Service Provider also called an Identity Provider
Authenticator	Code that authenticates a password, token, MFA, federated assertion, and so on.
Verifier	"An entity that verifies the claimant's identity by verifying the claimant's possession and control of one or two authenticators using an authentication protocol. To do this, the verifier may also need to validate credentials that link the authenticator(s) to the subscriber's identifier and check their status"
OTP	One-time password
SFA	Single factor authenticators, such as something you know (memorized secrets, passwords, passphrases, PINs), something you are (biometrics, fingerprint, face scans), or something you have (OTP tokens, a cryptographic device such as a smart card),
MFA	Multi factor authenticator, which includes two or more single factors

References

For more information, see also:

- NIST 800-63 - Digital Identity Guidelines
- NIST 800-63 A - Enrollment and Identity Proofing
- NIST 800-63 B - Authentication and Lifecycle Management
- NIST 800-63 C - Federation and Assertions
- NIST 800-63 FAQ
- OWASP Testing Guide 4.0: Testing for Authentication
- OWASP Cheat Sheet - Password storage
- OWASP Cheat Sheet - Forgot password
- OWASP Cheat Sheet - Choosing and using security questions

V3: Session Management Verification Requirements

Control Objective

One of the core components of any web-based application or stateful API is the mechanism by which it controls and maintains the state for a user or device interacting with it. Session management changes a stateless protocol to stateful, which is critical for differentiating different users or devices.

Ensure that a verified application satisfies the following high-level session management requirements:

- Sessions are unique to each individual and cannot be guessed or shared.
- Sessions are invalidated when no longer required and timed out during periods of inactivity.

As previously noted, these requirements have been adapted to be a compliant subset of selected NIST 800-63b controls, focused around common threats and commonly exploited authentication weaknesses. Previous verification requirements have been retired, de-duped, or in most cases adapted to be strongly aligned with the intent of mandatory NIST 800-63b requirements.

Security Verification Requirements

V3.1 Fundamental Session Management Requirements

#	Description	L1	L2	L3	CWE	NIST §
3.1.1	Verify the application never reveals session tokens in URL parameters or error messages.	✓	✓	✓	598	

V3.2 Session Binding Requirements

#	Description	L1	L2	L3	CWE	NIST §
3.2.1	Verify the application generates a new session token on user authentication. (C6)	✓	✓	✓	384	7.1
3.2.2	Verify that session tokens possess at least 64 bits of entropy. (C6)	✓	✓	✓	331	7.1
3.2.3	Verify the application only stores session tokens in the browser using secure methods such as appropriately secured cookies (see section 3.4) or HTML 5 session storage.	✓	✓	✓	539	7.1
3.2.4	Verify that session token are generated using approved cryptographic algorithms. (C6)		✓	✓	331	7.1

TLS or another secure transport channel is mandatory for session management. This is covered off in the Communications Security chapter.

V3.3 Session Logout and Timeout Requirements

Session timeouts have been aligned with NIST 800-63, which permits much longer session timeouts than traditionally permitted by security standards. Organizations should review the table below, and if a longer time out is desirable based around the application's risk, the NIST value should be the upper bounds of session idle timeouts.

L1 in this context is IAL1/AAL1, L2 is IAL2/AAL3, L3 is IAL3/AAL3. For IAL2/AAL2 and IAL3/AAL3, the shorter idle timeout is, the lower bound of idle times for being logged out or re-authenticated to resume the session.

#	Description	L1	L2	L3	CWE	NIST §
3.3.1	Verify that logout and expiration invalidate the session token, such that the back button or a downstream relying party does not resume an authenticated session, including across relying parties. (C6)	✓	✓	✓	613	7.1
3.3.2	If authenticators permit users to remain logged in, verify that re-authentication occurs periodically both when actively used or after an idle period. (C6)	30 days	12 hours or 30 minutes of inactivity, 2FA optional	12 hours or 15 minutes of inactivity, with 2FA	613	7.2
3.3.3	Verify that the application terminates all other active sessions after a successful password change, and that this is effective across the application, federated login (if present), and any relying parties.		✓	✓	613	
3.3.4	Verify that users are able to view and log out of any or all currently active sessions and devices.		✓	✓	613	7.1

V3.4 Cookie-based Session Management

#	Description	L1	L2	L3	CWE	NIST §
3.4.1	Verify that cookie-based session tokens have the 'Secure' attribute set. (C6)	✓	✓	✓	614	7.1.1
3.4.2	Verify that cookie-based session tokens have the 'HttpOnly' attribute set. (C6)	✓	✓	✓	1004	7.1.1
3.4.3	Verify that cookie-based session tokens utilize the 'SameSite' attribute to limit exposure to cross-site request forgery attacks. (C6)	✓	✓	✓	16	7.1.1
3.4.4	Verify that cookie-based session tokens use "__Host-" prefix (see references) to provide session cookie confidentiality.	✓	✓	✓	16	7.1.1
3.4.5	Verify that if the application is published under a domain name with other applications that set or use session cookies that might override or disclose the session cookies, set the path attribute in cookie-based session tokens using the most precise path possible. (C6)	✓	✓	✓	16	7.1.1

V3.5 Token-based Session Management

Token-based session management includes JWT, OAuth, SAML, and API keys. Of these, API keys are known to be weak and should not be used in new code.

#	Description	L1	L2	L3	CWE	NIST §
3.5.1	Verify the application does not treat OAuth and refresh tokens — on their own — as the presence of the subscriber and allows users to terminate trust relationships with linked applications.		✓	✓	290	7.1.2
3.5.2	Verify the application uses session tokens rather than static API secrets and keys, except with legacy implementations.		✓	✓	798	
3.5.3	Verify that stateless session tokens use digital signatures, encryption, and other countermeasures to protect against tampering, enveloping, replay, null cipher, and key substitution attacks.		✓	✓	345	

V3.6 Re-authentication from a Federation or Assertion

This section relates to those writing relying party (RP) or credential service provider (CSP) code. If relying on code implementing these features, ensure that these issues are handled correctly.

#	Description	L1	L2	L3	CWE	NIST §
3.6.1	Verify that relying parties specify the maximum authentication time to CSPs and that CSPs re-authenticate the subscriber if they haven't used a session within that period.			✓	613	7.2.1
3.6.2	Verify that CSPs inform relying parties of the last authentication event, to allow RPs to determine if they need to re-authenticate the user.			✓	613	7.2.1

V3.7 Defenses Against Session Management Exploits

There are a small number of session management attacks, some related to the user experience (UX) of sessions. Previously, based on ISO 27002 requirements, the ASVS has required blocking multiple simultaneous sessions. Blocking simultaneous sessions is no longer appropriate, not only as modern users have many devices or the app is an API without a browser session, but in most of these implementations, the last authenticator wins, which is often the attacker. This section provides leading guidance on deterring, delaying and detecting session management attacks using code.

Description of the half-open Attack

In early 2018, several financial institutions were compromised using what the attackers called "half-open attacks". This term has stuck in the industry. The attackers struck multiple institutions with different proprietary code bases, and indeed it seems different code bases within the same institutions. The half-open attack is exploiting a design pattern flaw commonly found in many existing authentication, session management and access control systems.

Attackers start a half-open attack by attempting to lock, reset, or recover a credential. A popular session management design pattern re-uses user profile session objects/models between unauthenticated, half-authenticated (password resets, forgot username), and fully authenticated code. This design pattern populates a valid session object or token containing the victim's profile, including password hashes and roles. If access control checks in controllers or routers does not correctly verify that the user is fully logged in, the attacker will be able to act as the user. Attacks could include changing the user's password to a known value, update the email address to

perform a valid password reset, disable multi-factor authentication or enroll a new MFA device, reveal or change API keys, and so on.

#	Description	L1	L2	L3	CWE	NIST §
3.7.1	Verify the application ensures a valid login session or requires re-authentication or secondary verification before allowing any sensitive transactions or account modifications.	✓	✓	✓	778	

References

For more information, see also:

- OWASP Testing Guide 4.0: Session Management Testing
- OWASP Session Management Cheat Sheet
- Set-Cookie __Host- prefix details

V4: Access Control Verification Requirements

Control Objective

Authorization is the concept of allowing access to resources only to those permitted to use them. Ensure that a verified application satisfies the following high level requirements:

- Persons accessing resources hold valid credentials to do so.
- Users are associated with a well-defined set of roles and privileges.
- Role and permission metadata is protected from replay or tampering.

Security Verification Requirements

V4.1 General Access Control Design

#	Description	L1	L2	L3	CWE
4.1.1	Verify that the application enforces access control rules on a trusted service layer, especially if client-side access control is present and could be bypassed.	✓	✓	✓	602
4.1.2	Verify that all user and data attributes and policy information used by access controls cannot be manipulated by end users unless specifically authorized.	✓	✓	✓	639
4.1.3	Verify that the principle of least privilege exists - users should only be able to access functions, data files, URLs, controllers, services, and other resources, for which they possess specific authorization. This implies protection against spoofing and elevation of privilege. (C7)	✓	✓	✓	285
4.1.4	Verify that the principle of deny by default exists whereby new users/roles start with minimal or no permissions and users/roles do not receive access to new features until access is explicitly assigned. (C7)	✓	✓	✓	276
4.1.5	Verify that access controls fail securely including when an exception occurs. (C10)	✓	✓	✓	285

V4.2 Operation Level Access Control

#	Description	L1	L2	L3	CWE
4.2.1	Verify that sensitive data and APIs are protected against direct object attacks targeting creation, reading, updating and deletion of records, such as creating or updating someone else's record, viewing everyone's records, or deleting all records.	✓	✓	✓	639
4.2.2	Verify that the application or framework enforces a strong anti-CSRF mechanism to protect authenticated functionality, and effective anti-automation or anti-CSRF protects unauthenticated functionality.	✓	✓	✓	352

V4.3 Other Access Control Considerations

#	Description	L1	L2	L3	CWE
4.3.1	Verify administrative interfaces use appropriate multi-factor authentication to prevent unauthorized use.	✓	✓	✓	419

#	Description	L1	L2	L3	CWE
4.3.2	Verify that directory browsing is disabled unless deliberately desired. Additionally, applications should not allow discovery or disclosure of file or directory metadata, such as Thumbs.db, .DS_Store, .git or .svn folders.	✓	✓	✓	548
4.3.3	Verify the application has additional authorization (such as step up or adaptive authentication) for lower value systems, and / or segregation of duties for high value applications to enforce anti-fraud controls as per the risk of application and past fraud.		✓	✓	732

References

For more information, see also:

- OWASP Testing Guide 4.0: Authorization
- OWASP Cheat Sheet: Access Control
- OWASP CSRF Cheat Sheet
- OWASP REST Cheat Sheet

V5: Validation, Sanitization and Encoding Verification Requirements

Control Objective

The most common web application security weakness is the failure to properly validate input coming from the client or the environment before directly using it without any output encoding. This weakness leads to almost all of the significant vulnerabilities in web applications, such as Cross-Site Scripting (XSS), SQL injection, interpreter injection, locale/Unicode attacks, file system attacks, and buffer overflows.

Ensure that a verified application satisfies the following high-level requirements:

- Input validation and output encoding architecture have an agreed pipeline to prevent injection attacks.
- Input data is strongly typed, validated, range or length checked, or at worst, sanitized or filtered.
- Output data is encoded or escaped as per the context of the data as close to the interpreter as possible.

With modern web application architecture, output encoding is more important than ever. It is difficult to provide robust input validation in certain scenarios, so the use of safer API such as parameterized queries, auto-escaping templating frameworks, or carefully chosen output encoding is critical to the security of the application.

V5.1 Input Validation Requirements

Properly implemented input validation controls, using positive whitelisting and strong data typing, can eliminate more than 90% of all injection attacks. Length and range checks can reduce this further. Building in secure input validation is required during application architecture, design sprints, coding, and unit and integration testing. Although many of these items cannot be found in penetration tests, the results of not implementing them are usually found in V5.3 - Output encoding and Injection Prevention Requirements. Developers and secure code reviewers are recommended to treat this section as if L1 is required for all items to prevent injections.

#	Description	L1	L2	L3	CWE
5.1.1	Verify that the application has defenses against HTTP parameter pollution attacks, particularly if the application framework makes no distinction about the source of request parameters (GET, POST, cookies, headers, or environment variables).	✓	✓	✓	235
5.1.2	Verify that frameworks protect against mass parameter assignment attacks, or that the application has countermeasures to protect against unsafe parameter assignment, such as marking fields private or similar. (C5)	✓	✓	✓	915
5.1.3	Verify that all input (HTML form fields, REST requests, URL parameters, HTTP headers, cookies, batch files, RSS feeds, etc) is validated using positive validation (whitelisting). (C5)	✓	✓	✓	20
5.1.4	Verify that structured data is strongly typed and validated against a defined schema including allowed characters, length and pattern (e.g. credit card numbers or telephone, or validating that two related fields are reasonable, such as checking that suburb and zip/postcode match). (C5)	✓	✓	✓	20
5.1.5	Verify that URL redirects and forwards only allow whitelisted destinations, or show a warning when redirecting to potentially untrusted content.	✓	✓	✓	601

V5.2 Sanitization and Sandboxing Requirements

#	Description	L1	L2	L3	CWE
5.2.1	Verify that all untrusted HTML input from WYSIWYG editors or similar is properly sanitized with an HTML sanitizer library or framework feature. (C5)	✓	✓	✓	116
5.2.2	Verify that unstructured data is sanitized to enforce safety measures such as allowed characters and length.	✓	✓	✓	138
5.2.3	Verify that the application sanitizes user input before passing to mail systems to protect against SMTP or IMAP injection.	✓	✓	✓	147
5.2.4	Verify that the application avoids the use of eval() or other dynamic code execution features. Where there is no alternative, any user input being included must be sanitized or sandboxed before being executed.	✓	✓	✓	95
5.2.5	Verify that the application protects against template injection attacks by ensuring that any user input being included is sanitized or sandboxed.	✓	✓	✓	94
5.2.6	Verify that the application protects against SSRF attacks, by validating or sanitizing untrusted data or HTTP file metadata, such as filenames and URL input fields, use whitelisting of protocols, domains, paths and ports.	✓	✓	✓	918
5.2.7	Verify that the application sanitizes, disables, or sandboxes user-supplied SVG scriptable content, especially as they relate to XSS resulting from inline scripts, and foreignObject.	✓	✓	✓	159
5.2.8	Verify that the application sanitizes, disables, or sandboxes user-supplied scriptable or expression template language content, such as Markdown, CSS or XSL stylesheets, BBCode, or similar.	✓	✓	✓	94

V5.3 Output encoding and Injection Prevention Requirements

Output encoding close or adjacent to the interpreter in use is critical to the security of any application. Typically, output encoding is not persisted, but used to render the output safe in the appropriate output context for immediate use. Failing to output encode will result in an insecure, injectable, and unsafe application.

#	Description	L1	L2	L3	CWE
5.3.1	Verify that output encoding is relevant for the interpreter and context required. For example, use encoders specifically for HTML values, HTML attributes, JavaScript, URL Parameters, HTTP headers, SMTP, and others as the context requires, especially from untrusted inputs (e.g. names with Unicode or apostrophes, such as ねこ or O'Hara). (C4)	✓	✓	✓	116
5.3.2	Verify that output encoding preserves the user's chosen character set and locale, such that any Unicode character point is valid and safely handled. (C4)	✓	✓	✓	176
5.3.3	Verify that context-aware, preferably automated - or at worst, manual - output escaping protects against reflected, stored, and DOM based XSS. (C4)	✓	✓	✓	79
5.3.4	Verify that data selection or database queries (e.g. SQL, HQL, ORM, NoSQL) use parameterized queries, ORMs, entity frameworks, or are otherwise protected from database injection attacks. (C3)	✓	✓	✓	89

#	Description	L1	L2	L3	CWE
5.3.5	Verify that where parameterized or safer mechanisms are not present, context-specific output encoding is used to protect against injection attacks, such as the use of SQL escaping to protect against SQL injection. (C3, C4)	✓	✓	✓	89
5.3.6	Verify that the application projects against JavaScript or JSON injection attacks, including for eval attacks, remote JavaScript includes, CSP bypasses, DOM XSS, and JavaScript expression evaluation. (C4)	✓	✓	✓	830
5.3.7	Verify that the application protects against LDAP Injection vulnerabilities, or that specific security controls to prevent LDAP Injection have been implemented. (C4)	✓	✓	✓	943
5.3.8	Verify that the application protects against OS command injection and that operating system calls use parameterized OS queries or use contextual command line output encoding. (C4)	✓	✓	✓	78
5.3.9	Verify that the application protects against Local File Inclusion (LFI) or Remote File Inclusion (RFI) attacks.	✓	✓	✓	829
5.3.10	Verify that the application protects against XPath injection or XML injection attacks. (C4)	✓	✓	✓	643

Note: Using parameterized queries or escaping SQL is not always sufficient; table and column names, ORDER BY and so on, cannot be escaped. The inclusion of escaped user-supplied data in these fields results in failed queries or SQL injection.

Note: The SVG format explicitly allows ECMA script in almost all contexts, so it may not be possible to block all SVG XSS vectors completely. If SVG upload is required, we strongly recommend either serving these uploaded files as text/plain or using a separate user supplied content domain to prevent successful XSS from taking over the application.

V5.4 Memory, String, and Unmanaged Code Requirements

The following requirements will only apply when the application uses a systems language or unmanaged code.

#	Description	L1	L2	L3	CWE
5.4.1	Verify that the application uses memory-safe string, safer memory copy and pointer arithmetic to detect or prevent stack, buffer, or heap overflows.		✓	✓	120
5.4.2	Verify that format strings do not take potentially hostile input, and are constant.		✓	✓	134
5.4.3	Verify that sign, range, and input validation techniques are used to prevent integer overflows.		✓	✓	190

V5.5 Deserialization Prevention Requirements

#	Description	L1	L2	L3	CWE
5.5.1	Verify that serialized objects use integrity checks or are encrypted to prevent hostile object creation or data tampering. (C5)	✓	✓	✓	502
5.5.2	Verify that the application correctly restricts XML parsers to only use the most restrictive configuration possible and to ensure that unsafe features such as resolving external entities are disabled to prevent XXE.	✓	✓	✓	611

#	Description	L1	L2	L3	CWE
5.5.3	Verify that deserialization of untrusted data is avoided or is protected in both custom code and third-party libraries (such as JSON, XML and YAML parsers).	✓	✓	✓	502
5.5.4	Verify that when parsing JSON in browsers or JavaScript-based backends, JSON.parse is used to parse the JSON document. Do not use eval() to parse JSON.	✓	✓	✓	95

References

For more information, see also:

- OWASP Testing Guide 4.0: Input Validation Testing
- OWASP Cheat Sheet: Input Validation
- OWASP Testing Guide 4.0: Testing for HTTP Parameter Pollution
- OWASP LDAP Injection Cheat Sheet
- OWASP Testing Guide 4.0: Client Side Testing
- OWASP Cross Site Scripting Prevention Cheat Sheet
- OWASP DOM Based Cross Site Scripting Prevention Cheat Sheet
- OWASP Java Encoding Project
- OWASP Mass Assignment Prevention Cheat Sheet
- DOMPurify - Client-side HTML Sanitization Library
- XML External Entity (XXE) Prevention Cheat Sheet)

For more information on auto-escaping, please see:

- Reducing XSS by way of Automatic Context-Aware Escaping in Template Systems
- AngularJS Strict Contextual Escaping
- AngularJS ngBind
- Angular Sanitization
- Angular Template Security
- ReactJS Escaping
- Improperly Controlled Modification of Dynamically-Determined Object Attributes

For more information on deserialization, please see:

- OWASP Deserialization Cheat Sheet
- OWASP Deserialization of Untrusted Data Guide

V6: Stored Cryptography Verification Requirements

Control Objective

Ensure that a verified application satisfies the following high level requirements:

- All cryptographic modules fail in a secure manner and that errors are handled correctly.
- A suitable random number generator is used.
- Access to keys is securely managed.

V6.1 Data Classification

The most important asset is the data processed, stored or transmitted by an application. Always perform a privacy impact assessment to classify the data protection needs of any stored data correctly.

#	Description	L1	L2	L3	CWE
6.1.1	Verify that regulated private data is stored encrypted while at rest, such as personally identifiable information (PII), sensitive personal information, or data assessed likely to be subject to EU's GDPR.		✓	✓	311
6.1.2	Verify that regulated health data is stored encrypted while at rest, such as medical records, medical device details, or de-anonymized research records.		✓	✓	311
6.1.3	Verify that regulated financial data is stored encrypted while at rest, such as financial accounts, defaults or credit history, tax records, pay history, beneficiaries, or de-anonymized market or research records.		✓	✓	311

V6.2 Algorithms

Recent advances in cryptography mean that previously safe algorithms and key lengths are no longer safe or sufficient to protect data. Therefore, it should be possible to change algorithms.

Although this section is not easily penetration tested, developers should consider this entire section as mandatory even though L1 is missing from most of the items.

#	Description	L1	L2	L3	CWE
6.2.1	Verify that all cryptographic modules fail securely, and errors are handled in a way that does not enable Padding Oracle attacks.	✓	✓	✓	310
6.2.2	Verify that industry proven or government approved cryptographic algorithms, modes, and libraries are used, instead of custom coded cryptography. (C8)		✓	✓	327
6.2.3	Verify that encryption initialization vector, cipher configuration, and block modes are configured securely using the latest advice.		✓	✓	326
6.2.4	Verify that random number, encryption or hashing algorithms, key lengths, rounds, ciphers or modes, can be reconfigured, upgraded, or swapped at any time, to protect against cryptographic breaks. (C8)		✓	✓	326
6.2.5	Verify that known insecure block modes (i.e. ECB, etc.), padding modes (i.e. PKCS#1 v1.5, etc.), ciphers with small block sizes (i.e. Triple-DES, Blowfish, etc.), and weak hashing algorithms (i.e. MD5, SHA1, etc.) are not used unless required for backwards compatibility.		✓	✓	326

#	Description	L1	L2	L3	CWE
6.2.6	Verify that nonces, initialization vectors, and other single use numbers must not be used more than once with a given encryption key. The method of generation must be appropriate for the algorithm being used.		✓	✓	326
6.2.7	Verify that encrypted data is authenticated via signatures, authenticated cipher modes, or HMAC to ensure that ciphertext is not altered by an unauthorized party.			✓	326
6.2.8	Verify that all cryptographic operations are constant-time, with no 'short-circuit' operations in comparisons, calculations, or returns, to avoid leaking information.			✓	385

V6.3 Random Values

True pseudo-random number generation (PRNG) is incredibly difficult to get right. Generally, good sources of entropy within a system will be quickly depleted if over-used, but sources with less randomness can lead to predictable keys and secrets.

#	Description	L1	L2	L3	CWE
6.3.1	Verify that all random numbers, random file names, random GUIDs, and random strings are generated using the cryptographic module's approved cryptographically secure random number generator when these random values are intended to be not guessable by an attacker.		✓	✓	338
6.3.2	Verify that random GUIDs are created using the GUID v4 algorithm, and a cryptographically-secure pseudo-random number generator (CSPRNG). GUIDs created using other pseudo-random number generators may be predictable.		✓	✓	338
6.3.3	Verify that random numbers are created with proper entropy even when the application is under heavy load, or that the application degrades gracefully in such circumstances.			✓	338

V6.4 Secret Management

Although this section is not easily penetration tested, developers should consider this entire section as mandatory even though L1 is missing from most of the items.

#	Description	L1	L2	L3	CWE
6.4.1	Verify that a secrets management solution such as a key vault is used to securely create, store, control access to and destroy secrets. (C8)		✓	✓	798
6.4.2	Verify that key material is not exposed to the application but instead uses an isolated security module like a vault for cryptographic operations. (C8)		✓	✓	320

References

For more information, see also:

- OWASP Testing Guide 4.0: Testing for weak Cryptography
- OWASP Cheat Sheet: Cryptographic Storage
- FIPS 140-2

V7: Error Handling and Logging Verification Requirements

Control Objective

The primary objective of error handling and logging is to provide useful information for the user, administrators, and incident response teams. The objective is not to create massive amounts of logs, but high quality logs, with more signal than discarded noise.

High quality logs will often contain sensitive data, and must be protected as per local data privacy laws or directives. This should include:

- Not collecting or logging sensitive information unless specifically required.
- Ensuring all logged information is handled securely and protected as per its data classification.
- Ensuring that logs are not stored forever, but have an absolute lifetime that is as short as possible.

If logs contain private or sensitive data, the definition of which varies from country to country, the logs become some of the most sensitive information held by the application and thus very attractive to attackers in their own right.

It is also important to ensure that the application fails securely and that errors do not disclose unnecessary information.

V7.1 Log Content Requirements

Logging sensitive information is dangerous - the logs become classified themselves, which means they need to be encrypted, become subject to retention policies, and must be disclosed in security audits. Ensure only necessary information is kept in logs, and certainly no payment, credentials (including session tokens), sensitive or personally identifiable information.

V7.1 covers OWASP Top 10 2017:A10. As 2017:A10 and this section are not penetration testable, it's important for:

- Developers to ensure full compliance with this section, as if all items were marked as L1
- Penetration testers to validate full compliance of all items in V7.1 via interview, screenshots, or assertion

#	Description	L1	L2	L3	CWE
7.1.1	Verify that the application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form. (C9, C10)	✓	✓	✓	532
7.1.2	Verify that the application does not log other sensitive data as defined under local privacy laws or relevant security policy. (C9)	✓	✓	✓	532
7.1.3	Verify that the application logs security relevant events including successful and failed authentication events, access control failures, deserialization failures and input validation failures. (C5, C7)		✓	✓	778
7.1.4	Verify that each log event includes necessary information that would allow for a detailed investigation of the timeline when an event happens. (C9)		✓	✓	778

V7.2 Log Processing Requirements

Timely logging is critical for audit events, triage, and escalation. Ensure that the application's logs are clear and can be easily monitored and analyzed either locally or log shipped to a remote monitoring system.

V7.2 covers OWASP Top 10 2017:A10. As 2017:A10 and this section are not penetration testable, it's important for:

- Developers to ensure full compliance with this section, as if all items were marked as L1

- Penetration testers to validate full compliance of all items in V7.2 via interview, screenshots, or assertion

#	Description	L1	L2	L3	CWE
7.2.1	Verify that all authentication decisions are logged, without storing sensitive session identifiers or passwords. This should include requests with relevant metadata needed for security investigations.		✓	✓	778
7.2.2	Verify that all access control decisions can be logged and all failed decisions are logged. This should include requests with relevant metadata needed for security investigations.		✓	✓	285

V7.3 Log Protection Requirements

Logs that can be trivially modified or deleted are useless for investigations and prosecutions. Disclosure of logs can expose inner details about the application or the data it contains. Care must be taken when protecting logs from unauthorized disclosure, modification or deletion.

#	Description	L1	L2	L3	CWE
7.3.1	Verify that the application appropriately encodes user-supplied data to prevent log injection. (C9)		✓	✓	117
7.3.2	Verify that all events are protected from injection when viewed in log viewing software. (C9)		✓	✓	117
7.3.3	Verify that security logs are protected from unauthorized access and modification. (C9)		✓	✓	200
7.3.4	Verify that time sources are synchronized to the correct time and time zone. Strongly consider logging only in UTC if systems are global to assist with post-incident forensic analysis. (C9)		✓	✓	

Note: Log encoding (7.3.1) is difficult to test and review using automated dynamic tools and penetration tests, but architects, developers, and source code reviewers should consider it an L1 requirement.

V7.4 Error Handling

The purpose of error handling is to allow the application to provide security relevant events for monitoring, triage and escalation. The purpose is not to create logs. When logging security related events, ensure that there is a purpose to the log, and that it can be distinguished by SIEM or analysis software.

#	Description	L1	L2	L3	CWE
7.4.1	Verify that a generic message is shown when an unexpected or security sensitive error occurs, potentially with a unique ID which support personnel can use to investigate. (C10)	✓	✓	✓	210
7.4.2	Verify that exception handling (or a functional equivalent) is used across the codebase to account for expected and unexpected error conditions. (C10)		✓	✓	544
7.4.3	Verify that a "last resort" error handler is defined which will catch all unhandled exceptions. (C10)		✓	✓	460

Note: Certain languages, such as Swift and Go - and through common design practice - many functional languages, do not support exceptions or last resort event handlers. In this case, architects and developers should use a

pattern, language, or framework friendly way to ensure that applications can securely handle exceptional, unexpected, or security-related events.

References

For more information, see also:

- OWASP Testing Guide 4.0 content: Testing for Error Handling

V8: Data Protection Verification Requirements

Control Objective

There are three key elements to sound data protection: Confidentiality, Integrity and Availability (CIA). This standard assumes that data protection is enforced on a trusted system, such as a server, which has been hardened and has sufficient protections.

Applications have to assume that all user devices are compromised in some way. Where an application transmits or stores sensitive information on insecure devices, such as shared computers, phones and tablets, the application is responsible for ensuring data stored on these devices is encrypted and cannot be easily illicitly obtained, altered or disclosed.

Ensure that a verified application satisfies the following high level data protection requirements:

- Confidentiality: Data should be protected from unauthorized observation or disclosure both in transit and when stored.
- Integrity: Data should be protected from being maliciously created, altered or deleted by unauthorized attackers.
- Availability: Data should be available to authorized users as required.

V8.1 General Data Protection

#	Description	L1	L2	L3	CWE
8.1.1	Verify the application protects sensitive data from being cached in server components such as load balancers and application caches.		✓	✓	524
8.1.2	Verify that all cached or temporary copies of sensitive data stored on the server are protected from unauthorized access or purged/invalidated after the authorized user accesses the sensitive data.		✓	✓	524
8.1.3	Verify the application minimizes the number of parameters in a request, such as hidden fields, Ajax variables, cookies and header values.		✓	✓	233
8.1.4	Verify the application can detect and alert on abnormal numbers of requests, such as by IP, user, total per hour or day, or whatever makes sense for the application.		✓	✓	770
8.1.5	Verify that regular backups of important data are performed and that test restoration of data is performed.			✓	19
8.1.6	Verify that backups are stored securely to prevent data from being stolen or corrupted.			✓	19

V8.2 Client-side Data Protection

#	Description	L1	L2	L3	CWE
8.2.1	Verify the application sets sufficient anti-caching headers so that sensitive data is not cached in modern browsers.	✓	✓	✓	525
8.2.2	Verify that data stored in client side storage (such as HTML5 local storage, session storage, IndexedDB, regular cookies or Flash cookies) does not contain sensitive data or PII.	✓	✓	✓	922

#	Description	L1	L2	L3	CWE
8.2.3	Verify that authenticated data is cleared from client storage, such as the browser DOM, after the client or session is terminated.	✓	✓	✓	922

V8.3 Sensitive Private Data

This section helps protect sensitive data from being created, read, updated, or deleted without authorization, particularly in bulk quantities.

Compliance with this section implies compliance with V4 Access Control, and in particular V4.2. For example, to protect against unauthorized updates or disclosure of sensitive personal information requires adherence to V4.2.1. Please comply with this section and V4 for full coverage.

Note: Privacy regulations and laws, such as the Australian Privacy Principles APP-11 or GDPR, directly affect how applications must approach the implementation of storage, use, and transmission of sensitive personal information. This ranges from severe penalties to simple advice. Please consult your local laws and regulations, and consult a qualified privacy specialist or lawyer as required.

#	Description	L1	L2	L3	CWE
8.3.1	Verify that sensitive data is sent to the server in the HTTP message body or headers, and that query string parameters from any HTTP verb do not contain sensitive data.	✓	✓	✓	319
8.3.2	Verify that users have a method to remove or export their data on demand.	✓	✓	✓	212
8.3.3	Verify that users are provided clear language regarding collection and use of supplied personal information and that users have provided opt-in consent for the use of that data before it is used in any way.	✓	✓	✓	285
8.3.4	Verify that all sensitive data created and processed by the application has been identified, and ensure that a policy is in place on how to deal with sensitive data. (C8)	✓	✓	✓	200
8.3.5	Verify accessing sensitive data is audited (without logging the sensitive data itself), if the data is collected under relevant data protection directives or where logging of access is required.		✓	✓	532
8.3.6	Verify that sensitive information contained in memory is overwritten as soon as it is no longer required to mitigate memory dumping attacks, using zeroes or random data.		✓	✓	226
8.3.7	Verify that sensitive or private information that is required to be encrypted, is encrypted using approved algorithms that provide both confidentiality and integrity. (C8)		✓	✓	327
8.3.8	Verify that sensitive personal information is subject to data retention classification, such that old or out of date data is deleted automatically, on a schedule, or as the situation requires.		✓	✓	285

When considering data protection, a primary consideration should be around bulk extraction or modification or excessive usage. For example, many social media systems only allow users to add 100 new friends per day, but which system these requests came from is not important. A banking platform might wish to block more than 5 transactions per hour transferring more than 1000 euro of funds to external institutions. Each system's

requirements are likely to be very different, so deciding on "abnormal" must consider the threat model and business risk. Important criteria are the ability to detect, deter, or preferably block such abnormal bulk actions.

References

For more information, see also:

- Consider using Security Headers website to check security and anti-caching headers
- OWASP Secure Headers project
- OWASP Privacy Risks Project
- OWASP User Privacy Protection Cheat Sheet
- European Union General Data Protection Regulation (GDPR) overview
- European Union Data Protection Supervisor - Internet Privacy Engineering Network

V9: Communications Verification Requirements

Control Objective

Ensure that a verified application satisfies the following high level requirements:

- TLS or strong encryption is always used, regardless of the sensitivity of the data being transmitted
- The most recent, leading configuration advice is used to enable and order preferred algorithms and ciphers
- Weak or soon to be deprecated algorithms and ciphers are ordered as a last resort
- Deprecated or known insecure algorithms and ciphers are disabled.

Leading industry advice on secure TLS configuration changes frequently, often due to catastrophic breaks in existing algorithms and ciphers. Always use the most recent versions of TLS configuration review tools (such as SSLyze or other TLS scanners) to configure the preferred order and algorithm selection. Configuration should be periodically checked to ensure that secure communications configuration is always present and effective.

V9.1 Communications Security Requirements

All client communications should only take place over encrypted communication paths. In particular, the use of TLS 1.2 or later is essentially all but required by modern browsers and search engines. Configuration should be regularly reviewed using online tools to ensure that the latest leading practices are in place.

#	Description	L1	L2	L3	CWE
9.1.1	Verify that secured TLS is used for all client connectivity, and does not fall back to insecure or unencrypted protocols. (C8)	✓	✓	✓	319
9.1.2	Verify using online or up to date TLS testing tools that only strong algorithms, ciphers, and protocols are enabled, with the strongest algorithms and ciphers set as preferred.	✓	✓	✓	326
9.1.3	Verify that old versions of SSL and TLS protocols, algorithms, ciphers, and configuration are disabled, such as SSLv2, SSLv3, or TLS 1.0 and TLS 1.1. The latest version of TLS should be the preferred cipher suite.	✓	✓	✓	326

V9.2 Server Communications Security Requirements

Server communications are more than just HTTP. Secure connections to and from other systems, such as monitoring systems, management tools, remote access and ssh, middleware, database, mainframes, partner or external source systems — must be in place. All of these must be encrypted to prevent "hard on the outside, trivially easy to intercept on the inside".

#	Description	L1	L2	L3	CWE
9.2.1	Verify that connections to and from the server use trusted TLS certificates. Where internally generated or self-signed certificates are used, the server must be configured to only trust specific internal CAs and specific self-signed certificates. All others should be rejected.		✓	✓	295
9.2.2	Verify that encrypted communications such as TLS is used for all inbound and outbound connections, including for management ports, monitoring, authentication, API, or web service calls, database, cloud, serverless, mainframe, external, and partner connections. The server must not fall back to insecure or unencrypted protocols.		✓	✓	319

#	Description	L1	L2	L3	CWE
9.2.3	Verify that all encrypted connections to external systems that involve sensitive information or functions are authenticated.		✓	✓	287
9.2.4	Verify that proper certification revocation, such as Online Certificate Status Protocol (OCSP) Stapling, is enabled and configured.		✓	✓	299
9.2.5	Verify that backend TLS connection failures are logged.			✓	544

References

For more information, see also:

- OWASP – TLS Cheat Sheet
- Notes on “Approved modes of TLS”. In the past, the ASVS referred to the US standard FIPS 140-2, but as a global standard, applying US standards can be difficult, contradictory, or confusing to apply. A better method of achieving compliance with 9.1.3 would be to review guides such as Mozilla's Server Side TLS or generate known good configurations, and use known TLS evaluation tools, such as sslyze, various vulnerability scanners or trusted TLS online assessment services to obtain a desired level of security. In general, we see non-compliance for this section being the use of outdated or insecure ciphers and algorithms, the lack of perfect forward secrecy, outdated or insecure SSL protocols, weak preferred ciphers, and so on.

V10: Malicious Code Verification Requirements

Control Objective

Ensure that code satisfies the following high level requirements:

- Malicious activity is handled securely and properly to not affect the rest of the application.
- Does not have time bombs or other time-based attacks.
- Does not "phone home" to malicious or unauthorized destinations.
- Does not have back doors, Easter eggs, salami attacks, rootkits, or unauthorized code that can be controlled by an attacker.

Finding malicious code is proof of the negative, which is impossible to completely validate. Best efforts should be undertaken to ensure that the code has no inherent malicious code or unwanted functionality.

V10.1 Code Integrity Controls

The best defense against malicious code is "trust, but verify". Introducing unauthorized or malicious code into code is often a criminal offence in many jurisdictions. Policies and procedures should make sanctions regarding malicious code clear.

Lead developers should regularly review code check-ins, particularly those that might access time, I/O, or network functions.

#	Description	L1	L2	L3	CWE
10.1.1	Verify that a code analysis tool is in use that can detect potentially malicious code, such as time functions, unsafe file operations and network connections.			✓	749

V10.2 Malicious Code Search

Malicious code is extremely rare and is difficult to detect. Manual line by line code review can assist looking for logic bombs, but even the most experienced code reviewer will struggle to find malicious code even if they know it exists.

Complying with this section is not possible without complete access to source code, including third-party libraries.

#	Description	L1	L2	L3	CWE
10.2.1	Verify that the application source code and third party libraries do not contain unauthorized phone home or data collection capabilities. Where such functionality exists, obtain the user's permission for it to operate before collecting any data.		✓	✓	359
10.2.2	Verify that the application does not ask for unnecessary or excessive permissions to privacy related features or sensors, such as contacts, cameras, microphones, or location.		✓	✓	272
10.2.3	Verify that the application source code and third party libraries do not contain back doors, such as hard-coded or additional undocumented accounts or keys, code obfuscation, undocumented binary blobs, rootkits, or anti-debugging, insecure debugging features, or otherwise out of date, insecure, or hidden functionality that could be used maliciously if discovered.			✓	507

#	Description	L1	L2	L3	CWE
10.2.4	Verify that the application source code and third party libraries does not contain time bombs by searching for date and time related functions.			✓	511
10.2.5	Verify that the application source code and third party libraries does not contain malicious code, such as salami attacks, logic bypasses, or logic bombs.			✓	511
10.2.6	Verify that the application source code and third party libraries do not contain Easter eggs or any other potentially unwanted functionality.			✓	507

V10.3 Deployed Application Integrity Controls

Once an application is deployed, malicious code can still be inserted. Applications need to protect themselves against common attacks, such as executing unsigned code from untrusted sources and sub-domain takeovers.

Complying with this section is likely to be operational and continuous.

#	Description	L1	L2	L3	CWE
10.3.1	Verify that if the application has a client or server auto-update feature, updates should be obtained over secure channels and digitally signed. The update code must validate the digital signature of the update before installing or executing the update.	✓	✓	✓	16
10.3.2	Verify that the application employs integrity protections, such as code signing or sub-resource integrity. The application must not load or execute code from untrusted sources, such as loading includes, modules, plugins, code, or libraries from untrusted sources or the Internet.	✓	✓	✓	353
10.3.3	Verify that the application has protection from sub-domain takeovers if the application relies upon DNS entries or DNS sub-domains, such as expired domain names, out of date DNS pointers or CNAMEs, expired projects at public source code repos, or transient cloud APIs, serverless functions, or storage buckets (*autogen-bucket-id*.cloud.example.com) or similar. Protections can include ensuring that DNS names used by applications are regularly checked for expiry or change.	✓	✓	✓	350

References

- Hostile Sub-Domain Takeover, Detectify Labs
- Hijacking of abandoned subdomains part 2, Detectify Labs

V11: Business Logic Verification Requirements

Control Objective

Ensure that a verified application satisfies the following high level requirements:

- The business logic flow is sequential, processed in order, and cannot be bypassed.
- Business logic includes limits to detect and prevent automated attacks, such as continuous small funds transfers, or adding a million friends one at a time, and so on.
- High value business logic flows have considered abuse cases and malicious actors, and have protections against spoofing, tampering, repudiation, information disclosure, and elevation of privilege attacks.

V11.1 Business Logic Security Requirements

Business logic security is so individual to every application that no one checklist will ever apply. Business logic security must be designed in to protect against likely external threats - it cannot be added using web application firewalls or secure communications. We recommend the use of threat modelling during design sprints, for example using the OWASP Cornucopia or similar tools.

#	Description	L1	L2	L3	CWE
11.1.1	Verify the application will only process business logic flows for the same user in sequential step order and without skipping steps.	✓	✓	✓	841
11.1.2	Verify the application will only process business logic flows with all steps being processed in realistic human time, i.e. transactions are not submitted too quickly.	✓	✓	✓	779
11.1.3	Verify the application has appropriate limits for specific business actions or transactions which are correctly enforced on a per user basis.	✓	✓	✓	770
11.1.4	Verify the application has sufficient anti-automation controls to detect and protect against data exfiltration, excessive business logic requests, excessive file uploads or denial of service attacks.	✓	✓	✓	770
11.1.5	Verify the application has business logic limits or validation to protect against likely business risks or threats, identified using threat modelling or similar methodologies.	✓	✓	✓	841
11.1.6	Verify the application does not suffer from "time of check to time of use" (TOCTOU) issues or other race conditions for sensitive operations.		✓	✓	367
11.1.7	Verify the application monitors for unusual events or activity from a business logic perspective. For example, attempts to perform actions out of order or actions which a normal user would never attempt. (C9)		✓	✓	754
11.1.8	Verify the application has configurable alerting when automated attacks or unusual activity is detected.		✓	✓	390

References

For more information, see also:

- OWASP Testing Guide 4.0: Business Logic Testing
- OWASP Cheat Sheet
- Anti-automation can be achieved in many ways, including the use of OWASP AppSensor and OWASP Automated Threats to Web Applications
- OWASP AppSensor can also help with Attack Detection and Response.
- OWASP Cornucopia

V12: File and Resources Verification Requirements

Control Objective

Ensure that a verified application satisfies the following high level requirements:

- Untrusted file data should be handled accordingly and in a secure manner.
- Untrusted file data obtained from untrusted sources are stored outside the web root and with limited permissions.

V12.1 File Upload Requirements

Although zip bombs are eminently testable using penetration testing techniques, they are considered L2 and above to encourage design and development consideration with careful manual testing, and to avoid automated or unskilled manual penetration testing of a denial of service condition.

#	Description	L1	L2	L3	CWE
12.1.1	Verify that the application will not accept large files that could fill up storage or cause a denial of service attack.	✓	✓	✓	400
12.1.2	Verify that compressed files are checked for "zip bombs" - small input files that will decompress into huge files thus exhausting file storage limits.		✓	✓	409
12.1.3	Verify that a file size quota and maximum number of files per user is enforced to ensure that a single user cannot fill up the storage with too many files, or excessively large files.		✓	✓	770

V12.2 File Integrity Requirements

#	Description	L1	L2	L3	CWE
12.2.1	Verify that files obtained from untrusted sources are validated to be of expected type based on the file's content.		✓	✓	434

V12.3 File execution Requirements

#	Description	L1	L2	L3	CWE
12.3.1	Verify that user-submitted filename metadata is not used directly with system or framework file and URL API to protect against path traversal.	✓	✓	✓	22
12.3.2	Verify that user-submitted filename metadata is validated or ignored to prevent the disclosure, creation, updating or removal of local files (LFI).	✓	✓	✓	73
12.3.3	Verify that user-submitted filename metadata is validated or ignored to prevent the disclosure or execution of remote files (RFI), which may also lead to SSRF.	✓	✓	✓	98
12.3.4	Verify that the application protects against reflective file download (RFD) by validating or ignoring user-submitted filenames in a JSON, JSONP, or URL parameter, the response Content-Type header should be set to text/plain, and the Content-Disposition header should have a fixed filename.	✓	✓	✓	641
12.3.5	Verify that untrusted file metadata is not used directly with system API or libraries, to protect against OS command injection.	✓	✓	✓	78

#	Description	L1	L2	L3	CWE
12.3.6	Verify that the application does not include and execute functionality from untrusted sources, such as unverified content distribution networks, JavaScript libraries, node npm libraries, or server-side DLLs.		✓	✓	829

V12.4 File Storage Requirements

#	Description	L1	L2	L3	CWE
12.4.1	Verify that files obtained from untrusted sources are stored outside the web root, with limited permissions, preferably with strong validation.	✓	✓	✓	922
12.4.2	Verify that files obtained from untrusted sources are scanned by antivirus scanners to prevent upload of known malicious content.	✓	✓	✓	509

V12.5 File Download Requirements

#	Description	L1	L2	L3	CWE
12.5.1	Verify that the web tier is configured to serve only files with specific file extensions to prevent unintentional information and source code leakage. For example, backup files (e.g. .bak), temporary working files (e.g. .swp), compressed files (.zip, .tar.gz, etc) and other extensions commonly used by editors should be blocked unless required.	✓	✓	✓	552
12.5.2	Verify that direct requests to uploaded files will never be executed as HTML/JavaScript content.	✓	✓	✓	434

V12.6 SSRF Protection Requirements

#	Description	L1	L2	L3	CWE
12.6.1	Verify that the web or application server is configured with a whitelist of resources or systems to which the server can send requests or load data/files from.	✓	✓	✓	918

References

For more information, see also:

- File Extension Handling for Sensitive Information
- Reflective file download by Oren Hafif
- OWASP Third Party JavaScript Management Cheat Sheet

V13: API and Web Service Verification Requirements

Control Objective

Ensure that a verified application that uses trusted service layer APIs (commonly using JSON or XML or GraphQL) has:

- Adequate authentication, session management and authorization of all web services.
- Input validation of all parameters that transit from a lower to higher trust level.
- Effective security controls for all API types, including cloud and Serverless API

Please read this chapter in combination with all other chapters at this same level; we no longer duplicate authentication or API session management concerns.

V13.1 Generic Web Service Security Verification Requirements

#	Description	L1	L2	L3	CWE
13.1.1	Verify that all application components use the same encodings and parsers to avoid parsing attacks that exploit different URI or file parsing behavior that could be used in SSRF and RFI attacks.	✓	✓	✓	116
13.1.2	Verify that access to administration and management functions is limited to authorized administrators.	✓	✓	✓	419
13.1.3	Verify API URLs do not expose sensitive information, such as the API key, session tokens etc.	✓	✓	✓	598
13.1.4	Verify that authorization decisions are made at both the URI, enforced by programmatic or declarative security at the controller or router, and at the resource level, enforced by model-based permissions.		✓	✓	285
13.1.5	Verify that requests containing unexpected or missing content types are rejected with appropriate headers (HTTP response status 406 Unacceptable or 415 Unsupported Media Type).		✓	✓	434

V13.2 RESTful Web Service Verification Requirements

JSON schema validation is in a draft stage of standardization (see references). When considering using JSON schema validation, which is best practice for SOAP web services, consider using these additional data validation strategies in combination with JSON schema validation:

- Parsing validation of the JSON object, such as if there are missing or extra elements.
- Validation of the JSON object values using standard input validation methods, such as data type, data format, length, etc.
- and formal JSON schema validation.

Once the JSON schema validation standard is formalized, ASVS will update its advice in this area. Carefully monitor any JSON schema validation libraries in use, as they will need to be updated regularly until the standard is formalized and bugs are ironed out of reference implementations.

#	Description	L1	L2	L3	CWE
13.2.1	Verify that enabled RESTful HTTP methods are a valid choice for the user or action, such as preventing normal users using DELETE or PUT on protected API or resources.	✓	✓	✓	650
13.2.2	Verify that JSON schema validation is in place and verified before accepting input.	✓	✓	✓	20
13.2.3	Verify that RESTful web services that utilize cookies are protected from Cross-Site Request Forgery via the use of at least one or more of the following: triple or double submit cookie pattern (see references), CSRF nonces, or ORIGIN request header checks.	✓	✓	✓	352
13.2.4	Verify that REST services have anti-automation controls to protect against excessive calls, especially if the API is unauthenticated.		✓	✓	779
13.2.5	Verify that REST services explicitly check the incoming Content-Type to be the expected one, such as application/xml or application/JSON.		✓	✓	436
13.2.6	Verify that the message headers and payload are trustworthy and not modified in transit. Requiring strong encryption for transport (TLS only) may be sufficient in many cases as it provides both confidentiality and integrity protection. Per-message digital signatures can provide additional assurance on top of the transport protections for high-security applications but bring with them additional complexity and risks to weigh against the benefits.		✓	✓	345

V13.3 SOAP Web Service Verification Requirements

#	Description	L1	L2	L3	CWE
13.3.1	Verify that XSD schema validation takes place to ensure a properly formed XML document, followed by validation of each input field before any processing of that data takes place.	✓	✓	✓	20
13.3.2	Verify that the message payload is signed using WS-Security to ensure reliable transport between client and service.		✓	✓	345

Note: Due to issues with XXE attacks against DTDs, DTD validation should not be used, and framework DTD evaluation disabled as per the requirements set out in V14 Configuration.

V13.4 GraphQL and other Web Service Data Layer Security Requirements

#	Description	L1	L2	L3	CWE
13.4.1	Verify that query whitelisting or a combination of depth limiting and amount limiting should be used to prevent GraphQL or data layer expression denial of service (DoS) as a result of expensive, nested queries. For more advanced scenarios, query cost analysis should be used.		✓	✓	770
13.4.2	Verify that GraphQL or other data layer authorization logic should be implemented at the business logic layer instead of the GraphQL layer.		✓	✓	285

References

For more information, see also:

- OWASP Serverless Top 10
- OWASP Serverless Project
- OWASP Testing Guide 4.0: Configuration and Deployment Management Testing
- OWASP Cross-Site Request Forgery cheat sheet
- OWASP XML External Entity Prevention Cheat Sheet - General Guidance* JSON Web Tokens (and Signing)
- REST Security Cheat Sheet
- JSON Schema
- XML DTD Entity Attacks
- Orange Tsai - A new era of SSRF Exploiting URL Parser In Trending Programming Languages

V14: Configuration Verification Requirements

Control Objective

Ensure that a verified application has:

- A secure, repeatable, automatable build environment.
- Hardened third party library, dependency and configuration management such that out of date or insecure components are not included by the application.
- A secure-by-default configuration, such that administrators and users have to weaken the default security posture.

Configuration of the application out of the box should be safe to be on the Internet, which means a safe out of the box configuration.

V14.1 Build

Build pipelines are the basis for repeatable security - every time something insecure is discovered, it can be resolved in the source code, build or deployment scripts, and tested automatically. We are strongly encouraging the use of build pipelines with automatic security and dependency checks that warn or break the build to prevent known security issues being deployed into production. Manual steps performed irregularly directly leads to avoidable security mistakes.

As the industry moves to a DevSecOps model, it is important to ensure the continued availability and integrity of deployment and configuration to achieve a "known good" state. In the past, if a system was hacked, it would take days to months to prove that no further intrusions had taken place. Today, with the advent of software defined infrastructure, rapid A/B deployments with zero downtime, and automated containerized builds, it is possible to automatically and continuously build, harden, and deploy a "known good" replacement for any compromised system.

If traditional models are still in place, then manual steps must be taken to harden and back up that configuration to allow the compromised systems to be quickly replaced with high integrity, uncompromised systems in a timely fashion.

Compliance with this section requires an automated build system, and access to build and deployment scripts.

#	Description	L1	L2	L3	CWE
14.1.1	Verify that the application build and deployment processes are performed in a secure and repeatable way, such as CI / CD automation, automated configuration management, and automated deployment scripts.		✓	✓	
14.1.2	Verify that compiler flags are configured to enable all available buffer overflow protections and warnings, including stack randomization, data execution prevention, and to break the build if an unsafe pointer, memory, format string, integer, or string operations are found.		✓	✓	120
14.1.3	Verify that server configuration is hardened as per the recommendations of the application server and frameworks in use.		✓	✓	16
14.1.4	Verify that the application, configuration, and all dependencies can be re-deployed using automated deployment scripts, built from a documented and tested runbook in a reasonable time, or restored from backups in a timely fashion.		✓	✓	

#	Description	L1	L2	L3	CWE
14.1.5	Verify that authorized administrators can verify the integrity of all security-relevant configurations to detect tampering.			✓	

V14.2 Dependency

Dependency management is critical to the safe operation of any application of any type. Failure to keep up to date with outdated or insecure dependencies is the root cause of the largest and most expensive attacks to date.

Note: At Level 1, 14.2.1 compliance relates to observations or detections of client-side and other libraries and components, rather than the more accurate build-time static code analysis or dependency analysis. These more accurate techniques could be discoverable by interview as required.

#	Description	L1	L2	L3	CWE
14.2.1	Verify that all components are up to date, preferably using a dependency checker during build or compile time. (C2)	✓	✓	✓	1026
14.2.2	Verify that all unneeded features, documentation, samples, configurations are removed, such as sample applications, platform documentation, and default or example users.	✓	✓	✓	1002
14.2.3	Verify that if application assets, such as JavaScript libraries, CSS stylesheets or web fonts, are hosted externally on a content delivery network (CDN) or external provider, Subresource Integrity (SRI) is used to validate the integrity of the asset.	✓	✓	✓	714
14.2.4	Verify that third party components come from pre-defined, trusted and continually maintained repositories. (C2)		✓	✓	829
14.2.5	Verify that an inventory catalog is maintained of all third party libraries in use. (C2)		✓	✓	
14.2.6	Verify that the attack surface is reduced by sandboxing or encapsulating third party libraries to expose only the required behaviour into the application. (C2)		✓	✓	265

V14.3 Unintended Security Disclosure Requirements

Configurations for production should be hardened to protect against common attacks, such as debug consoles, raise the bar for cross-site scripting (XSS) and remote file inclusion (RFI) attacks, and to eliminate trivial information discovery "vulnerabilities" that are the unwelcome hallmark of many penetration testing reports. Many of these issues are rarely rated as a significant risk, but they are chained together with other vulnerabilities. If these issues are not present by default, it raises the bar before most attacks can succeed.

#	Description	L1	L2	L3	CWE
14.3.1	Verify that web or application server and framework error messages are configured to deliver user actionable, customized responses to eliminate any unintended security disclosures.	✓	✓	✓	209
14.3.2	Verify that web or application server and application framework debug modes are disabled in production to eliminate debug features, developer consoles, and unintended security disclosures.	✓	✓	✓	497
14.3.3	Verify that the HTTP headers or any part of the HTTP response do not expose detailed version information of system components.	✓	✓	✓	200

V14.4 HTTP Security Headers Requirements

#	Description	L1	L2	L3	CWE
14.4.1	Verify that every HTTP response contains a content type header specifying a safe character set (e.g., UTF-8, ISO 8859-1).	✓	✓	✓	173
14.4.2	Verify that all API responses contain Content-Disposition: attachment; filename="api.json" (or other appropriate filename for the content type).	✓	✓	✓	116
14.4.3	Verify that a content security policy (CSPv2) is in place that helps mitigate impact for XSS attacks like HTML, DOM, JSON, and JavaScript injection vulnerabilities.	✓	✓	✓	1021
14.4.4	Verify that all responses contain X-Content-Type-Options: nosniff.	✓	✓	✓	116
14.4.5	Verify that HTTP Strict Transport Security headers are included on all responses and for all subdomains, such as Strict-Transport-Security: max-age=15724800; includeSubdomains.	✓	✓	✓	523
14.4.6	Verify that a suitable "Referrer-Policy" header is included, such as "no-referrer" or "same-origin".	✓	✓	✓	116
14.4.7	Verify that a suitable X-Frame-Options or Content-Security-Policy: frame-ancestors header is in use for sites where content should not be embedded in a third-party site.	✓	✓	✓	346

V14.5 Validate HTTP Request Header Requirements

#	Description	L1	L2	L3	CWE
14.5.1	Verify that the application server only accepts the HTTP methods in use by the application or API, including pre-flight OPTIONS.	✓	✓	✓	749
14.5.2	Verify that the supplied Origin header is not used for authentication or access control decisions, as the Origin header can easily be changed by an attacker.	✓	✓	✓	346
14.5.3	Verify that the cross-domain resource sharing (CORS) Access-Control-Allow-Origin header uses a strict white-list of trusted domains to match against and does not support the "null" origin.	✓	✓	✓	346
14.5.4	Verify that HTTP headers added by a trusted proxy or SSO devices, such as a bearer token, are authenticated by the application.		✓	✓	306

References

For more information, see also:

- OWASP Testing Guide 4.0: Testing for HTTP Verb Tampering
- Adding Content-Disposition to API responses helps prevent many attacks based on misunderstanding on the MIME type between client and server, and the "filename" option specifically helps prevent Reflected File Download attacks.
- Content Security Policy Cheat Sheet
- Exploiting CORS misconfiguration for BitCoins and Bounties
- OWASP Testing Guide 4.0: Configuration and Deployment Management Testing

- Sandboxing third party components

Appendix A: Glossary

- **2FA** – Two-factor authentication(2FA) adds a second level of authentication to an account log-in.
- **Address Space Layout Randomization (ASLR)** – A technique to make exploiting memory corruption bugs more difficult.
- **Application Security** – Application-level security focuses on the analysis of components that comprise the application layer of the Open Systems Interconnection Reference Model (OSI Model), rather than focusing on for example the underlying operating system or connected networks.
- **Application Security Verification** – The technical assessment of an application against the OWASP ASVS.
- **Application Security Verification Report** – A report that documents the overall results and supporting analysis produced by the verifier for a particular application.
- **Authentication** – The verification of the claimed identity of an application user.
- **Automated Verification** – The use of automated tools (either dynamic analysis tools, static analysis tools, or both) that use vulnerability signatures to find problems.
- **Black box testing** – It is a method of software testing that examines the functionality of an application without peering into its internal structures or workings.
- **Component** – a self-contained unit of code, with associated disk and network interfaces that communicates with other components.
- **Cross-Site Scripting** (XSS) – A security vulnerability typically found in web applications allowing the injection of client-side scripts into content.
- **Cryptographic module** – Hardware, software, and/or firmware that implements cryptographic algorithms and/or generates cryptographic keys.
- **CWE** - Common Weakness Enumeration (CWE) is a community-developed list of common software security weaknesses. It serves as a common language, a measuring stick for software security tools, and as a baseline for weakness identification, mitigation, and prevention efforts.
- **DAST** – Dynamic application security testing (DAST) technologies are designed to detect conditions indicative of a security vulnerability in an application in its running state.
- **Design Verification** – The technical assessment of the security architecture of an application.
- **Dynamic Verification** – The use of automated tools that use vulnerability signatures to find problems during the execution of an application.
- **Globally Unique Identifier** (GUID) – a unique reference number used as an identifier in software.
- **Hyper Text Transfer Protocol** (HTTPS) – An application protocol for distributed, collaborative, hypermedia information systems. It is the foundation of data communication for the World Wide Web.
- **Hardcoded keys** – Cryptographic keys which are stored on the filesystem, be it in code, comments or files.
- **Input Validation** – The canonicalization and validation of untrusted user input.
- **Malicious Code** – Code introduced into an application during its development unbeknownst to the application owner, which circumvents the application's intended security policy. Not the same as malware such as a virus or worm!
- **Malware** – Executable code that is introduced into an application during runtime without the knowledge of the application user or administrator.

- **Open Web Application Security Project** (OWASP) – The Open Web Application Security Project (OWASP) is a worldwide free and open community focused on improving the security of application software. Our mission is to make application security "visible," so that people and organizations can make informed decisions about application security risks. See: https://www.owasp.org/
- **Personally Identifiable Information** (PII) - is information that can be used on its own or with other information to identify, contact, or locate a single person, or to identify an individual in context.
- **PIE** – Position-independent executable (PIE) is a body of machine code that, being placed somewhere in the primary memory, executes properly regardless of its absolute address.
- **PKI** – Public Key Infrastructure (PKI) is an arrangement that binds public keys with respective identities of entities. The binding is established through a process of registration and issuance of certificates at and by a certificate authority (CA).
- **SAST** – Static application security testing (SAST) is a set of technologies designed to analyze application source code, byte code and binaries for coding and design conditions that are indicative of security vulnerabilities. SAST solutions analyze an application from the “inside out” in a nonrunning state.
- **SDLC** – Software development lifecycle.
- **Security Architecture** – An abstraction of an application's design that identifies and describes where and how security controls are used, and also identifies and describes the location and sensitivity of both user and application data.
- **Security Configuration** – The runtime configuration of an application that affects how security controls are used.
- **Security Control** – A function or component that performs a security check (e.g. an access control check) or when called results in a security effect (e.g. generating an audit record).
- **SQL Injection (SQLi)** – A code injection technique used to attack data driven applications, in which malicious SQL statements are inserted into an entry point.
- **SSO Authentication** – Single Sign On (SSO) occurs when a user logs into one application and is then automatically logged in to other applications without having to re-authenticate. For example, when you login to Google, when accessing other Google services such as Youtube, Google Docs, and Gmail you will be automatically logged in.
- **Threat Modeling** - A technique consisting of developing increasingly refined security architectures to identify threat agents, security zones, security controls, and important technical and business assets.
- **Transport Layer Security** – Cryptographic protocols that provide communication security over a network connection
- **URI/URL/URL fragments** – A Uniform Resource Identifier is a string of characters used to identify a name or a web resource. A Uniform Resource Locator is often used as a reference to a resource.
- **Verifier** – The person or team that is reviewing an application against the OWASP ASVS requirements.
- **Whitelist** – A list of permitted data or operations, for example a list of characters that are allowed to perform input validation.
- **X.509 Certificate** – An X.509 certificate is a digital certificate that uses the widely accepted international X.509 public key infrastructure (PKI) standard to verify that a public key belongs to the user, computer or service identity contained within the certificate.

Appendix B: References

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

OWASP Core Projects

1. OWASP Top 10 Project: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
2. OWASP Testing Guide: https://www.owasp.org/index.php/OWASP_Testing_Project
3. OWASP Proactive Controls: https://www.owasp.org/index.php/OWASP_Proactive_Controls
4. OWASP Security Knowledge Framework: https://www.owasp.org/index.php/OWASP_Security_Knowledge_Framework
5. OWASP Software Assurance Maturity Model (SAMM): https://www.owasp.org/index.php/OWASP_SAMM_Project

Mobile Security Related Projects

1. OWASP Mobile Security Project: https://www.owasp.org/index.php/OWASP_Mobile_Security_Project
2. OWASP Mobile Top 10 Risks: https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks
3. OWASP Mobile Security Testing Guide: https://www.owasp.org/index.php/OWASP_Mobile_Security_Testing_Guide

OWASP Internet of Things related projects

1. OWASP Internet of Things Project: https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project

OWASP Serverless projects

1. OWASP Serverless Project: https://www.owasp.org/index.php/OWASP_Serverless_Top_10_Project

Others

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

1. SecLists Github: https://github.com/danielmiessler/SecLists
2. MITRE Common Weakness Enumeration: https://cwe.mitre.org/
3. PCI Security Standards Council: https://www.pcisecuritystandards.org
4. PCI Data Security Standard (DSS) v3.2.1 Requirements and Security Assessment Procedures: https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-2-1.pdf
5. PCI Software Security Framework - Secure Software Requirements and Assessment Procedures: https://www.pcisecuritystandards.org/documents/PCI-Secure-Software-Standard-v1_0.pdf
6. PCI Secure Software Lifecycle (Secure SLC) Requirements and Assessment Procedures: https://www.pcisecuritystandards.org/documents/PCI-Secure-SLC-Standard-v1_0.pdf

Appendix C: Internet of Things Verification Requirements

This section was originally in the main branch, but with the work that the OWASP IoT team has done, it doesn't make sense to maintain two different standards on the subject. For the 4.0 release, we are moving this to the Appendix, and urge all who require this, to rather use the main OWASP IoT project

Control Objective

Embedded/IoT devices should:

- Have the same level of security controls within the device as found in the server, by enforcing security controls in a trusted environment.
- Sensitive data stored on the device should be done so in a secure manner using hardware backed storage such as secure elements.
- All sensitive data transmitted from the device should utilize transport layer security.

Security Verification Requirements

#	Description	L1	L2	L3	Since
C.1	Verify that application layer debugging interfaces such USB, UART, and other serial variants are disabled or protected by a complex password.	✓	✓	✓	4.0
C.2	Verify that cryptographic keys and certificates are unique to each individual device.	✓	✓	✓	4.0
C.3	Verify that memory protection controls such as ASLR and DEP are enabled by the embedded/IoT operating system, if applicable.	✓	✓	✓	4.0
C.4	Verify that on-chip debugging interfaces such as JTAG or SWD are disabled or that available protection mechanism is enabled and configured appropriately.	✓	✓	✓	4.0
C.5	Verify that trusted execution is implemented and enabled, if available on the device SoC or CPU.	✓	✓	✓	4.0
C.6	Verify that sensitive data, private keys and certificates are stored securely in a Secure Element, TPM, TEE (Trusted Execution Environment), or protected using strong cryptography.	✓	✓	✓	4.0
C.7	Verify that the firmware apps protect data-in-transit using transport layer security.	✓	✓	✓	4.0
C.8	Verify that the firmware apps validate the digital signature of server connections.	✓	✓	✓	4.0
C.9	Verify that wireless communications are mutually authenticated.	✓	✓	✓	4.0
C.10	Verify that wireless communications are sent over an encrypted channel.	✓	✓	✓	4.0
C.11	Verify that any use of banned C functions are replaced with the appropriate safe equivalent functions.	✓	✓	✓	4.0
C.12	Verify that each firmware maintains a software bill of materials cataloging third-party components, versioning, and published vulnerabilities.	✓	✓	✓	4.0
C.13	Verify all code including third-party binaries, libraries, frameworks are reviewed for hardcoded credentials (backdoors).	✓	✓	✓	4.0

C.14	Verify that the application and firmware components are not susceptible to OS Command Injection by invoking shell command wrappers, scripts, or that security controls prevent OS Command Injection.	✓	✓	✓	4.0
C.15	Verify that the firmware apps pin the digital signature to a trusted server(s).		✓	✓	4.0
C.16	Verify the presence of tamper resistance and/or tamper detection features.		✓	✓	4.0
C.17	Verify that any available Intellectual Property protection technologies provided by the chip manufacturer are enabled.		✓	✓	4.0
C.18	Verify security controls are in place to hinder firmware reverse engineering (e.g., removal of verbose debugging symbols).		✓	✓	4.0
C.19	Verify the device validates the boot image signature before loading.		✓	✓	4.0
C.20	Verify that the firmware update process is not vulnerable to time-of-check vs time-of-use attacks.		✓	✓	4.0
C.21	Verify the device uses code signing and validates firmware upgrade files before installing.		✓	✓	4.0
C.22	Verify that the device cannot be downgraded to old versions (anti-rollback) of valid firmware.		✓	✓	4.0
C.23	Verify usage of cryptographically secure pseudo-random number generator on embedded device (e.g., using chip-provided random number generators).		✓	✓	4.0
C.24	Verify that firmware can perform automatic firmware updates upon a predefined schedule.		✓	✓	4.0
C.25	Verify that the device wipes firmware and sensitive data upon detection of tampering or receipt of invalid message.			✓	4.0
C.26	Verify that only micro controllers that support disabling debugging interfaces (e.g. JTAG, SWD) are used.			✓	4.0
C.27	Verify that only micro controllers that provide substantial protection from de-capping and side channel attacks are used.			✓	4.0
C.28	Verify that sensitive traces are not exposed to outer layers of the printed circuit board.			✓	4.0
C.29	Verify that inter-chip communication is encrypted (e.g. Main board to daughter board communication).			✓	4.0
C.30	Verify the device uses code signing and validates code before execution.			✓	4.0
C.31	Verify that sensitive information maintained in memory is overwritten with zeros as soon as it is no longer required.			✓	4.0
C.32	Verify that the firmware apps utilize kernel containers for isolation between apps.			✓	4.0
C.33	Verify that secure compiler flags such as -fPIE, -fstack-protector-all, -Wl,-z,noexecstack, -Wl,-z,noexecheap are configured for firmware builds.			✓	4.0

C.34	Verify that micro controllers are configured with code protection (if applicable).	✓	4.0

References

For more information, see also:

- OWASP Internet of Things Top 10
- OWASP Embedded Application Security Project
- OWASP Internet of Things Project
- Trudy TCP Proxy Tool