Application Security Verification Standard

Version 5.0.0

May 2025

Contents

Frontispiece **7**
About the Standard . . . 7
Copyright and License . . . 7
Project Leads . . . 7
Working Group . . . 7
Other Major Contributors . . . 7
Other Contributors and Reviewers . . . 8

Preface **8**
Introduction . . . 8
Principles behind version 5.0 . . . 8
Looking ahead . . . 9

What is the ASVS? **9**
Scope of the ASVS . . . 9
Application . . . 10
Security . . . 10
Verification . . . 10
Standard . . . 10
Requirement . . . 11
Documented security decisions . . . 11
Application Security Verification Levels . . . 12
Level evaluation . . . 13
Level 1 . . . 13
Level 2 . . . 13
Level 3 . . . 14
Which level to achieve . . . 14
How to use the ASVS . . . 14
The structure of the ASVS . . . 14
Release strategy . . . 14
Flexibility with the ASVS . . . 15
How to Reference ASVS Requirements . . . 15
Forking the ASVS . . . 16
Use cases for the ASVS . . . 16
As Detailed Security Architecture Guidance . . . 16
As a Specialized Secure Coding Reference . . . 16
As a Guide for Automated Unit and Integration Tests . . . 17
For Secure Development Training . . . 17
As a Framework for Guiding the Procurement of Secure Software . . . 17
Applying ASVS in Practice . . . 17

Assessment and Certification **18**
OWASP's Stance on ASVS Certifications and Trust Marks . . . 18
How to Verify ASVS Compliance . . . 18
Verification reporting . . . 18
Scope of Verification . . . 18
Verification Mechanisms . . . 19

Changes Compared to v4.x **20**
Introduction . . . 20
Requirement Philosophy . . . 20
Scope and Focus . . . 20
Emphasis on Security Goals Over Mechanisms . . . 20
Documented Security Decisions . . . 20
Structural Changes and New Chapters . . . 21
Removal of Direct Mappings to Other Standards . . . 21
Reduced Coupling with NIST Digital Identity Guidelines . . . 22
Moving Away from Common Weakness Enumeration (CWE) . . . 22
Rethinking Level Definitions . . . 22
Easier Entry Level . . . 22
The Fallacy of Testability . . . 22
Not Just About Risk . . . 23

V1 Encoding and Sanitization **23**
Control Objective . . . 23
V1.1 Encoding and Sanitization Architecture . . . 23
V1.2 Injection Prevention . . . 24
V1.3 Sanitization . . . 25
V1.4 Memory, String, and Unmanaged Code . . . 27
V1.5 Safe Deserialization . . . 27
References . . . 28

V2 Validation and Business Logic **28**
Control Objective . . . 28
V2.1 Validation and Business Logic Documentation . . . 29
V2.2 Input Validation . . . 29
V2.3 Business Logic Security . . . 30
V2.4 Anti-automation . . . 31
References . . . 31

V3 Web Frontend Security **32**
Control Objective . . . 32
V3.1 Web Frontend Security Documentation . . . 32

V3.2 Unintended Content Interpretation . . . 32
V3.3 Cookie Setup . . . 33
V3.4 Browser Security Mechanism Headers . . . 33
V3.5 Browser Origin Separation . . . 35
V3.6 External Resource Integrity . . . 36
V3.7 Other Browser Security Considerations . . . 37
References . . . 37

V4 API and Web Service **38**
Control Objective . . . 38
V4.1 Generic Web Service Security . . . 38
V4.2 HTTP Message Structure Validation . . . 39
V4.3 GraphQL . . . 40
V4.4 WebSocket . . . 40
References . . . 41

V5 File Handling **41**
Control Objective . . . 41
V5.1 File Handling Documentation . . . 41
V5.2 File Upload and Content . . . 42
V5.3 File Storage . . . 42
V5.4 File Download . . . 43
References . . . 43

V6 Authentication **44**
Control Objective . . . 44
V6.1 Authentication Documentation . . . 44
V6.2 Password Security . . . 45
V6.3 General Authentication Security . . . 46
V6.4 Authentication Factor Lifecycle and Recovery . . . 47
V6.5 General Multi-factor authentication requirements . . . 48
V6.6 Out-of-Band authentication mechanisms . . . 49
V6.7 Cryptographic authentication mechanism . . . 50
V6.8 Authentication with an Identity Provider . . . 51
References . . . 52

V7 Session Management **52**
Control Objective . . . 52
V7.1 Session Management Documentation . . . 53
V7.2 Fundamental Session Management Security . . . 53
V7.3 Session Timeout . . . 54
V7.4 Session Termination . . . 54

V7.5 Defenses Against Session Abuse . . . 55
V7.6 Federated Re-authentication . . . 56
References . . . 56

V8 Authorization **56**
Control Objective . . . 56
V8.1 Authorization Documentation . . . 57
V8.2 General Authorization Design . . . 57
V8.3 Operation Level Authorization . . . 58
V8.4 Other Authorization Considerations . . . 58
References . . . 59

V9 Self-contained Tokens **59**
Control Objective . . . 59
V9.1 Token source and integrity . . . 59
V9.2 Token content . . . 60
References . . . 61

V10 OAuth and OIDC **61**
Control Objective . . . 61
V10.1 Generic OAuth and OIDC Security . . . 63
V10.2 OAuth Client . . . 63
V10.3 OAuth Resource Server . . . 64
V10.4 OAuth Authorization Server . . . 65
V10.5 OIDC Client . . . 67
V10.6 OpenID Provider . . . 68
V10.7 Consent Management . . . 68
References . . . 69

V11 Cryptography **70**
Control Objective . . . 70
V11.1 Cryptographic Inventory and Documentation . . . 70
V11.2 Secure Cryptography Implementation . . . 71
V11.3 Encryption Algorithms . . . 72
V11.4 Hashing and Hash-based Functions . . . 73
V11.5 Random Values . . . 73
V11.6 Public Key Cryptography . . . 74
V11.7 In-Use Data Cryptography . . . 74
References . . . 75

V12 Secure Communication **75**
Control Objective . . . 75
V12.1 General TLS Security Guidance . . . 75

V12.2 HTTPS Communication with External Facing Services . . . 76
V12.3 General Service to Service Communication Security . . . 76
References . . . 77

V13 Configuration **77**
Control Objective . . . 77
V13.1 Configuration Documentation . . . 78
V13.2 Backend Communication Configuration . . . 78
V13.3 Secret Management . . . 79
V13.4 Unintended Information Leakage . . . 80
References . . . 81

V14 Data Protection **81**
Control Objective . . . 81
V14.1 Data Protection Documentation . . . 81
V14.2 General Data Protection . . . 82
V14.3 Client-side Data Protection . . . 83
References . . . 84

V15 Secure Coding and Architecture **84**
Control Objective . . . 84
V15.1 Secure Coding and Architecture Documentation . . . 84
V15.2 Security Architecture and Dependencies . . . 85
V15.3 Defensive Coding . . . 86
V15.4 Safe Concurrency . . . 87
References . . . 88

V16 Security Logging and Error Handling **88**
Control Objective . . . 88
V16.1 Security Logging Documentation . . . 89
V16.2 General Logging . . . 89
V16.3 Security Events . . . 90
V16.4 Log Protection . . . 91
V16.5 Error Handling . . . 91
References . . . 92

V17 WebRTC **92**
Control Objective . . . 92
V17.1 TURN Server . . . 93
V17.2 Media . . . 93
V17.3 Signaling . . . 95
References . . . 95

Appendix A: Glossary **96**

Appendix B: References **102**
OWASP Core Projects . . . 102
OWASP Cheat Sheet Series project . . . 102
Mobile Security Related Projects . . . 102
OWASP Internet of Things related projects . . . 102
OWASP Serverless projects . . . 102
Others . . . 102

Appendix C: Cryptography Standards **103**
Cryptographic Inventory and Documentation . . . 103
Equivalent Strengths of Cryptographic Parameters . . . 104
Random Values . . . 104
Cipher Algorithms . . . 105
AES Cipher Modes . . . 106
Key Wrapping . . . 107
Authenticated Encryption . . . 107
Hash Functions . . . 108
Hash Functions for General Use Cases . . . 108
Hash Functions for Password Storage . . . 111
Key Derivation Functions (KDFs) . . . 112
General Key Derivation Functions . . . 112
Password-based Key Derivation Functions . . . 112
Key Exchange Mechanisms . . . 113
KEX Schemes . . . 113
Diffie-Hellman groups . . . 113
Message Authentication Codes (MAC) . . . 114
Digital Signatures . . . 115
Post-Quantum Encryption Standards . . . 116

Appendix D: Recommendations **116**
Introduction . . . 116
Recommended, in-scope mechanisms . . . 116
Software Security principles . . . 117
Software Security processes . . . 117

Appendix E - Contributors **118**

Frontispiece

About the Standard

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

Copyright and License

Version 5.0.0, May 2025

Figure 1: license

Copyright © 2008-2025 The OWASP Foundation.

This document is released under the Creative Commons Attribution-ShareAlike 4.0 International License.

For any reuse or distribution, you must clearly communicate the license terms of this work to others.

Project Leads

Elar Lang	Josh C Grossman
Jim Manico	Daniel Cuthbert

Working Group

Tobias Ahnoff	Ralph Andalis	Ryan Armstrong	Gabriel Corona
Meghan Jacquot	Shanni Prutchi	Iman Sharafaldin	Eden Yardeni

Other Major Contributors

Sjoerd Langkemper	Isaac Lewis

Mark Carney	Sandro Gauci

Other Contributors and Reviewers

We have included a list of the other contributors in Appendix E.

If a credit is missing from the 5.x credit list, please log a ticket at GitHub to be recognized in future 5.x updates.

The Application Security Verification Standard builds on the work of those involved in ASVS 1.0 (2008) through 4.0 (2019). Much of the structure and many of the verification items that remain in ASVS today were originally written by Andrew van der Stock, Mike Boberski, Jeff Williams, and Dave Wichers, among numerous other contributors. Thank you to everyone who has contributed in the past. For a comprehensive list of earlier contributors, please consult each prior version.

Preface

Welcome to the Application Security Verification Standard (ASVS) Version 5.0.

Introduction

Originally launched in 2008 through a global community collaboration, the ASVS defines a comprehensive set of security requirements for designing, developing, and testing modern web applications and services.

Following the release of ASVS 4.0 in 2019 and its minor update (v4.0.3) in 2021, Version 5.0 represents a significant milestone—modernized to reflect the latest advances in software security.

ASVS 5.0 is the result of extensive contributions from project leaders, working group members, and the wider OWASP community to update and improve this important standard.

Principles behind version 5.0

This major revision has been developed with several key principles in mind:

- Refined Scope and Focus: This version of the standard has been designed to align more directly with the foundational pillars in its name: Application, Security, Verification, and Standard. Requirements have been rewritten to emphasize the prevention of security flaws rather than mandating specific technical implementations. Requirement texts are intended to be self-explanatory, explaining why they exist.

- Support for Documented Security Decisions: ASVS 5.0 introduces requirements for documenting key security decisions. This enhances traceability and supports context-sensitive implementations, allowing organizations to tailor their security posture to their specific needs and risks.
- Updated Levels: While ASVS retains its three-tier model, the level definitions have evolved to make the ASVS easier to adopt. Level 1 is designed as the initial step to adopting the ASVS, providing the first layer of defense. Level 2 represents a comprehensive view of standard security practices, and Level 3 addresses advanced, high-assurance requirements.
- Restructured and Expanded Content: ASVS 5.0 includes approximately 350 requirements across 17 chapters. Chapters have been reorganized for clarity and usability. A two-way mapping between v4.0 and v5.0 is provided to facilitate migration.

Looking ahead

Just as securing an application is never truly finished, neither is the ASVS. Although Version 5.0 is a major release, development continues. This release allows the wider community to benefit from the improvements and additions which have been accumulated but also lays the groundwork for future enhancements. This could include community-driven efforts to create implementation and verification guidance built on top of the core requirement set.

ASVS 5.0 is designed to serve as a reliable foundation for secure software development. The community is invited to adopt, contribute, and build upon this standard to collectively advance the state of application security.

What is the ASVS?

The Application Security Verification Standard (ASVS) defines security requirements for web applications and services, and it is a valuable resource for anyone aiming to design, develop, and maintain secure applications or evaluate their security.

This chapter outlines the essential aspects of using the ASVS, including its scope, the structure of its priority-based levels, and the primary use cases for the standard.

Scope of the ASVS

The scope of the ASVS is defined by its name: Application, Security, Verification, and Standard. It establishes which requirements are included or excluded, with the overarching goal of identifying the security principles that must be achieved. The scope also considers documentation requirements, which serve as the foundation for implementation requirements.

There is no such thing as scope for attackers. Therefore, ASVS requirements should be evaluated alongside guidance for other aspects of the application lifecycle, including CI/CD processes, hosting, and operational activities.

Application

ASVS defines an "application"as the software product being developed, into which security controls must be integrated. ASVS does not prescribe development lifecycle activities or dictate how the application should be built via a CI/CD pipeline; instead, it specifies the security outcomes that must be achieved within the product itself.

Components that serve, modify, or validate HTTP traffic, such as Web Application Firewalls (WAFs), load balancers, or proxies, may be considered part of the application for those specific purposes, as some security controls depend directly on them or can be implemented through them. These components should be considered for requirements related to cached responses, rate limiting, or restricting incoming and outgoing connections based on source and destination.

Conversely, ASVS generally excludes requirements that are not directly relevant to the application or where configuration is outside the application's responsibility. For example, DNS issues are typically managed by a separate team or function.

Similarly, while the application is responsible for how it consumes input and produces output, if an external process interacts with the application or its data, it is considered out of scope for ASVS. For instance, backing up the application or its data is usually the responsibility of an external process and is not controlled by the application or its developers.

Security

Every requirement must have a demonstrable impact on security. The absence of a requirement must result in a less secure application, and implementing the requirement must reduce either the likelihood or the impact of a security risk.

All other considerations, such as functional aspects, code style, or policy requirements, are out of scope.

Verification

The requirement must be verifiable, and the verification must result in a "fail"or "pass"decision.

Standard

The ASVS is designed to be a collection of security requirements to be implemented to comply with the standard. This means that requirements are limited to defining the security goal to achieve that. Other related information can be built on top of ASVS or linked via mappings.

Specifically, OWASP has many projects, and the ASVS deliberately avoids overlapping with the content in other projects. For example, developers may have a question, “how do I implement a particular requirement in my particular technology or environment,”and this should be covered by the Cheat Sheet Series project. Verifiers may have a question “how do I test this requirement in this environment,”and this should be covered by the Web Security Testing Guide project.

Whilst the ASVS is not just intended for security experts to use, it does expect the reader to have technical knowledge to understand the content or the ability to research particular concepts.

Requirement

The word requirement is used specifically in the ASVS as it describes what must be achieved to satisfy it. The ASVS only contains requirements (must) and does not contain recommendations (should) as the main condition.

In other words, recommendations, whether they are just one of many possible options to solve a problem or code style considerations, do not satisfy the definition to be a requirement.

ASVS requirements are intended to address specific security principles without being too implementation or technology-specific, at the same time, being self-explanatory as to why they exist. This also means that requirements are not built around a particular verification method or implementation.

Documented security decisions

In software security, planning security design and the mechanisms to be used early on will lead to a more consistent and reliable implementation in the finished product or feature.

Additionally, for certain requirements, implementation will be complicated and very specific to an application’s needs. Common examples include permissions, input validation, and protective controls around different levels of sensitive data.

To account for this, rather than sweeping statements like “all data must be encrypted”or trying to cover every possible use case in a requirement, documentation requirements were included which mandate that the application developer’s approach and configuration to these sorts of controls must be documented. This can then be reviewed for appropriateness and then the actual implementation can be compared to the documentation to assess whether the implementation matches expectations.

These requirements are intended to document the decisions which the organization developing the application has taken regarding how to implement certain security requirements.

Documentation requirements are always in the first section of a chapter (although not every chapter has them) and always have a related implementation requirement where the decisions that are documented should actually be put into place. The point here is that verifying that the documentation is in place and that the actual implementation are two separate activities.

There are two key drivers for including these requirements. The first driver is that a security requirement will often involve enforcing rules e.g., what kind of file types are allowed to be uploaded, what business controls should be enforced, what are the allowed characters for a particular field. These rules will differ for every application, and therefore, the ASVS cannot prescriptively define what they should be, nor will a cheat sheet or more detailed response help in this case. Similarly, without these decisions being documented, it will not be possible to perform verification of the requirements that implement these decisions.

The second driver is that for certain requirements, it is important to provide an application development with flexibility regarding how to address particular security challenges. For example, in previous ASVS versions, session timeout rules were very prescriptive. Practically speaking, many applications, especially those that are consumer-facing, have much more relaxed rules and prefer to implement other mitigation controls instead. Documentation requirements, therefore, explicitly allow for flexibility around this.

Clearly, it is not expected that individual developers will be making and documenting these decisions but rather the organization as a whole will be taking those decisions and making sure that they are communicated to developers who then make sure to follow them.

Providing developers with specifications and designs for new features and functionality is a standard part of software development. Similarly, developers are expected to use common components and user interface mechanisms rather than just making their own decisions each time. As such, extending this to security should not be seen as surprising or controversial.

There is also flexibility around how to achieve this. Security decisions might be documented in a literal document, which developers are expected to refer to. Alternatively, security decisions could be documented and implemented in a common code library that all developers are mandated to use. In both cases, the desired result is achieved.

Application Security Verification Levels

The ASVS defines three security verification levels, with each level increasing in depth and complexity. The general aim is for organizations to start with the first level to address the most critical security concerns, and then move up to the higher levels according to the organization and application needs. Levels may be presented as L1, L2, and L3 in the document and in requirement texts.

Each ASVS level indicates the security requirements that are required to achieve from that level, with the higher remaining level requirements as recommendations.

In order to avoid duplicate requirements or requirements that are no longer relevant at higher levels, some requirements apply to a particular level but have more stringent conditions for higher levels.

Level evaluation

Levels are defined by priority-based evaluation of each requirement based on experience implementing and testing security requirements. The main focus is on comparing risk reduction with the effort to implement the requirement. Another key factor is to keep a low barrier to entry.

Risk reduction considers the extent to which the requirement reduces the level of security risk within the application, taking into account the classic Confidentiality, Integrity, and Availability impact factors as well as considering whether this is a primary layer of defense or whether it would be considered defense in depth.

The rigorous discussions around both the criteria and the leveling decisions have resulted in an allocation which should hold true for the vast majority of cases, whilst accepting that it may not be a 100% fit for every situation. This means that in certain cases, organizations may wish to prioritize requirements from a higher level earlier on based on their own specific risk considerations.

The types of requirements in each level could be characterized as follows.

Level 1

This level contains the minimum requirements to consider when securing an application and represents a critical starting point. This level contains around 20% of the ASVS requirements. The goal for this level is to have as few requirements as possible, to decrease the barrier to entry.

These requirements are generally critical or basic, first-layer of defense requirements for preventing common attacks that do not require other vulnerabilities or preconditions to be exploitable.

In addition to the first layer of defense requirements, some requirements have less of an impact at higher levels, such as requirements related to passwords. Those are more important for Level 1, as from higher levels, the multi-factor authentication requirements become relevant.

Level 1 is not necessarily penetration testable by an external tester without internal access to documentation or code (such as “black box”testing), although the lower number of requirements should make it easier to verify.

Level 2

Most applications should be striving to achieve this level of security. Around 50% of the requirements in the ASVS are L2 meaning that an application needs to implement around 70% of the requirements in the ASVS (all of the L1 and L2 requirements) in order to comply with L2.

These requirements generally relate to either less common attacks or more complicated protections against common attacks. They may still be a first layer of defense, or they may require certain preconditions for the attack to be successful.

Level 3

This level should be the goal for applications looking to demonstrate the highest levels of security and provides the final ~30% of requirements to comply with.

Requirements in this section are generally either defense-in-depth mechanisms or other useful but hard-to-implement controls.

Which level to achieve

The priority-based levels are intended to provide a reflection of the application security maturity of the organization and the application. Rather than the ASVS prescriptively stating what level an application should be at, an organization should analyze its risks and decide what level it believes it should be at, depending on the sensitivity of the application and of course, the expectations of the application's users.

For example, an early-stage startup that is only collecting limited sensitive data may decide to focus on Level 1 for its initial security goals, but a bank may have difficulty justifying anything less than Level 3 to its customers for its online banking application.

How to use the ASVS

The structure of the ASVS

The ASVS is made up of a total of around 350 requirements which are divided into 17 chapters, each of which is further divided into sections.

The aim of the chapter and section division is to simplify choosing or filtering out chapters and sections based on the what is relevant for the application. For example, for a machine-to-machine API, the requirements in chapter V3 related to web frontends will not be relevant. If there is no use of OAuth or WebRTC, then those chapters can be ignored as well.

Release strategy

ASVS releases follow the pattern "Major.Minor.Patch"and the numbers provide information on what has changed within the release. In a major release, the first number will change, in a minor release, the second number will change, and in a patch release, the third number will change.

- Major release - Full reorganization, almost everything may have changed, including requirement numbers. Reevaluation for compliance will be necessary (for example, 4.0.3 -> 5.0.0).
- Minor release - Requirements may be added or removed, but overall numbering will stay the same. Reevaluation for compliance will be necessary, but should be easier (for example, 5.0.0 -> 5.1.0).

- Patch release - Requirements may be removed (for example, if they are duplicates or outdated) or made less stringent, but an application that complied with the previous release will comply with the patch release as well (for example, 5.0.0 -> 5.0.1).

The above specifically relates to the requirements in the ASVS. Changes to surrounding text and other content such as the appendices will not be considered to be a breaking change.

Flexibility with the ASVS

Several of the points described above, such as documentation requirements and the levels mechanism, provide the ability to use the ASVS in a more flexible and organization-specific way.

Additionally, organizations are strongly encouraged to create an organization- or domain-specific fork that adjusts requirements based on the specific characteristics and risk levels of their applications. However, it is important to maintain traceability so that passing requirement 4.1.1 means the same across all versions.

Ideally, each organization should create its own tailored ASVS, omitting irrelevant sections (e.g., GraphQL, WebSockets, SOAP, if unused). An organization-specific ASVS version or supplement is also a good place to provide organization-specific implementation guidance, detailing libraries or resources to use when complying with requirements.

How to Reference ASVS Requirements

Each requirement has an identifier in the format `<chapter>.<section>.<requirement>`, where each element is a number. For example, `1.11.3`.

- The `<chapter>` value corresponds to the chapter from which the requirement comes; for example, all `1.#.#` requirements are from the 'Encoding and Sanitization'chapter.
- The `<section>` value corresponds to the section within that chapter where the requirement appears, for example: all `1.2.#` requirements are in the 'Injection Prevention'section of the 'Encoding and Sanitization'chapter.
- The `<requirement>` value identifies the specific requirement within the chapter and section, for example, `1.2.5` which as of version 5.0.0 of this standard is:

> 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.

Since the identifiers may change between versions of the standard, it is preferable for other documents, reports, or tools to use the following format: `v<version>-<chapter>.<section>.<requirement>`, where: 'version'is the ASVS version tag. For example: `v5.0.0-1.2.5` would be understood to mean specifically the 5th requirement in the 'Injection Prevention'section of the 'Encoding and Sanitization'chapter from version 5.0.0. (This could be summarized as `v<version>-<requirement_identifier>`.)

Note: The `v` preceding the version number in the format should always be lowercase.

If identifiers are used without including the `v<version>` element then they should be assumed to refer to the latest Application Security Verification Standard content. As the standard grows and changes this becomes problematic, which is why writers or developers should include the version element.

ASVS requirement lists are made available in CSV, JSON, and other formats which may be useful for reference or programmatic use.

Forking the ASVS

Organizations can benefit from adopting ASVS by choosing one of the three levels or by creating a domain-specific fork that adjusts requirements per application risk level. This type of fork is encouraged, provided that it maintains traceability so that passing requirement 4.1.1 means the same across all versions.

Ideally, each organization should create its own tailored ASVS, omitting irrelevant sections (e.g., GraphQL, Websockets, SOAP, if unused). Forking should start with ASVS Level 1 as a baseline, advancing to Levels 2 or 3 based on the application's risk.

Use cases for the ASVS

The ASVS can be used to assess the security of an application and this is explored in more depth in the next chapter. However, several other potential uses for the ASVS (or a forked version) have been identified.

As Detailed Security Architecture Guidance

One of the more common uses for the Application Security Verification Standard is as a resource for security architects. There are limited resources available for how to build a secure application archiecture, especially with modern applications. 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. The architecture and documentation requirements will be particularly useful for this.

As a Specialized Secure Coding Reference

The ASVS can be used as a basis for preparing a secure coding reference during application development, helping developers to make sure that they keep security in mind when they build software. Whilst the ASVS can be the base, prganizations should prepare their own specific guidance which is clear and unified and ideally be prepared based on guidance from security engineers or security

architects. As an extension to this, organizations are encouraged wherever possible to prepare approved security mechanisms and libraries that can be referenced in the guidance and used by developers.

As a Guide for Automated Unit and Integration Tests

The ASVS is designed to be highly testable. Some verifications will be technical where as other requirements (such as the architectural and documentation requirements) may require documentation or architecture review. By building unit and integration tests that test and fuzz for specific and relevant abuse cases related to the requirements that are verifiable by technical means, it should be easier to check that these controls are operating correctly on each 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 the 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 positive mechanisms found in the ASVS, rather than the Top 10 negative things not to do. The ASVS structure also provides a logical structure for walking through the different topics when securing an application.

As a Framework for Guiding the Procurement of Secure Software

The 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.

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 nonprofit, does not certify any vendors, verifiers, or software. Any assurance, trust mark, or certification claiming ASVS compliance is not officially endorsed by OWASP, so organizations should be cautious of third-party claims of ASVS certification.

Organizations may offer assurance services, provided they do not claim official OWASP certification.

How to Verify ASVS Compliance

The ASVS is deliberately not presciptive about exactly how to verify compliance at the level of a testing guide. However, it is important to highlight some key points.

Verification reporting

Traditional penetration testing reports issues "by exception,"only listing failures. However, an ASVS certification report should include scope, a summary of all requirements checked, the requirements where execptions were noted, and guidance on resolving issues. Some requirements may be non-applicable (e.g., session management in stateless APIs), and this must be noted in the report.

Scope of Verification

An organization developing an application will generally not implement all requirements, as some may be irrelevant or less significant based on the functionality of the application. The verifier should make the scope of the verification clear including which Level the organization is attempting to achieve and which requirements were included. This should be from the perspective of what was included rather than what was not included. They should also provide an opinion on the rationale of excluding the requirements which haven't been implemented.

This should allow the consumer of a verification report to understand the context of the verification and make an informed decision about the level of trust they can place in the application.

Certifying organizations can choose their testing methods but should disclose them in the report and this should ideally be repeatable. Different methods, like manual penetration tests or source code analysis, may be used to verify aspects such as input validation, depending on the application and requirements.

Verification Mechanisms

There are a number of different techniques which may be needed to verify specific ASVS requirements. Aside from penetration testing (using valid credentials to get full application coverage), verifying ASVS requirements may require access to documentation, source code, configuration, and the people involved in the development process. Especially for verifying L2 and L3 requirements. It is standard practice to provide robust evidence of findings with detailed documentation, which may include work papers, screenshots, scripts, and testing logs. Merely running an automated tool without thorough testing is insufficient for certification, as each requirement must be verifiably tested.

The use of automation to verify ASVS requirements is a topic that is constantly of interest. It is therefore important to clarify some points related to automated and black box testing.

The Role of Automated Security Testing Tools When automated security testing tools such as Dynamic and Static Application Security Testing tools (DAST and SAST) are correctly implemented in the build pipeline, they may be able to identify some security issues that should never exist. However, without careful configuration and tuning they will not provide the required coverage and the level of noise will prevent real security issues from being identified and mitigated.

Whilst this may provide coverage of some of the more basic and straightforward technical requirements such as those relating to output encoding or sanitiation, it is critical to note that these tools will be unable entirely to verify many of the more complicated ASVS requirements or those that relate to business logic and access control.

For less straightforward requirements, it is likely that automation can still be utilized but application specific verifications will need to be written to achieve this. These may be similar to unit and integration tests that the organization may already be using. It may therefore be possible to use this existing test automation infrastructure to write these ASVS specific tests. Whilst doing this will require short term investment, the long term benefits being able to continually verify these ASVS requirements will be significant.

In summary, testable using automation != running an off the shelf tool.

The Role of Penetration Testing Whilst L1 in version 4.0 was optimized for “black box”(no documentation and no source) testing to occur, even then the standard was clear that it is not an effective assurance activity and should be actively discouraged.

Testing without access to necessary additional information is an inefficient and ineffective mechanism for 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.

It is strongly encouraged to perform documentation or source code-led (hybrid) penetration testing, which have full access to the application developers and the application’s documentation, rather than traditional penetration tests. This will certainly be necessary in order to verify many of the ASVS requirements.

Changes Compared to v4.x

Introduction

Users familiar with version 4.x of the standard may find it helpful to review the key changes introduced in version 5.0, including updates in content, scope, and underlying philosophy.

Of the 286 requirements in version 4.0.3, only 11 remain unchanged, while 15 have undergone minor grammatical adjustments without altering their meaning. In total 109 requirements (38%) are no longer separate requirements in version 5.0 with 50 simply being deleted, 28 removed as duplicates and 31 merged into other requirements. The rest have been revised in some way. Even requirements that were not substantively modified have different identifiers due to reordering or restructuring.

To facilitate adoption of version 5.0, mapping documents are provided to help users trace how requirements from version 4.x correspond to those in version 5.0. These mappings are not tied to release versioning and may be updated or clarified as needed.

Requirement Philosophy

Scope and Focus

Version 4.x included requirements that did not align with the intended scope of the standard; these have been removed. Requirements that did not meet the scope criteria for 5.0 or were not verifiable have also been excluded.

Emphasis on Security Goals Over Mechanisms

In version 4.x, many requirements focused on specific mechanisms rather than the underlying security objectives. In version 5.0, requirements are centered on security goals, referencing particular mechanisms only when they are the sole practical solution, or providing them as examples or supplementary guidance.

This approach recognizes that multiple methods may exist to achieve a given security objective, and avoids unnecessary prescriptiveness that could limit organizational flexibility.

Additionally, requirements addressing the same security concern have been consolidated where appropriate.

Documented Security Decisions

While the concept of documented security decisions may appear new in version 5.0, it is an evolution of earlier requirements related to policy application and threat modeling in version 4.0. Previously, some requirements implicitly demanded analysis to inform the implementation of security controls, such as determining permitted network connections.

To ensure that necessary information is available for implementation and verification, these expectations are now explicitly defined as documentation requirements, making them clear, actionable, and verifiable.

Structural Changes and New Chapters

Several chapters in version 5.0 introduce entirely new content:

- OAuth and OIDC –Given the widespread adoption of these protocols for access delegation and single sign-on, dedicated requirements have been added to address the diverse scenarios developers may encounter. This area may eventually evolve into a standalone standard, similar to the treatment of Mobile and IoT requirements in previous versions.
- WebRTC –As this technology gains popularity, its unique security considerations and challenges are now addressed in a dedicated section.

Efforts have also been made to ensure that chapters and sections are organized around coherent sets of related requirements.

This restructuring has led to the creation of additional chapters:

- Self-contained Tokens –Formerly grouped under session management, self-contained tokens are now recognized as a distinct mechanism and a foundational element for stateless communication (such as in OAuth and OIDC). Due to their unique security implications, they are addressed in a dedicated chapter, with some new requirements introduced in version 5.x.
- Web Frontend Security –With the increasing complexity of browser-based applications and the rise of API-only architectures, frontend security requirements have been separated into their own chapter.
- Secure Coding and Architecture –New requirements addressing general security practices that did not fit within existing chapters have been grouped here.

Other organizational changes in version 5.0 were made to clarify intent. For example, input validation requirements were moved alongside business logic, reflecting their role in enforcing business rules, rather than being grouped with sanitization and encoding.

The former V1 Architecture chapter has been removed. Its initial section contained requirements that were out of scope, while subsequent sections have been redistributed to relevant chapters, with requirements deduplicated and clarified as necessary.

Removal of Direct Mappings to Other Standards

Direct mappings to other standards have been removed from the main body of the standard. The aim is to prepare a mapping with the OWASP Common Requirement Enumeration (CRE) project, which in turn will link ASVS to a range of OWASP projects and external standards.

Direct mappings to CWE and NIST are no longer maintained, as explained below.

Reduced Coupling with NIST Digital Identity Guidelines

The NIST Digital Identity Guidelines (SP 800-63) have long served as a reference for authentication and authorization controls. In version 4.x, certain chapters were closely aligned with NIST's structure and terminology.

While these guidelines remain an important reference, strict alignment introduced challenges, including less widely recognized terminology, duplication of similar requirements, and incomplete mappings. Version 5.0 moves away from this approach to improve clarity and relevance.

Moving Away from Common Weakness Enumeration (CWE)

The Common Weakness Enumeration (CWE) provides a useful taxonomy of software security weaknesses. However, challenges such as category-only CWEs, difficulties in mapping requirements to a single CWE, and the presence of imprecise mappings in version 4.x have led to the decision to discontinue direct CWE mappings in version 5.0.

Rethinking Level Definitions

Version 4.x described the levels as L1 ("Minimum"), L2 ("Standard"), and L3 ("Advanced"), with the implication that all applications handling sensitive data should meet at least L2.

Version 5.0 addresses several issues with this approach which are described in the following paragraphs.

As a practical matter, whereas version 4.x used tick marks for level indicators, 5.x uses a simple number on all formats of the standard including markdown, PDF, DOCX, CSV, JSON and XML. For backwards compatibility, legacy versions of the CSV, JSON and XML outputs which still use tick marks are also generated.

Easier Entry Level

Feedback indicated that the large number of Level 1 requirements (~120), combined with its designation as the "minimum"level that is not good enough for most applications, discouraged adoption. Version 5.0 aims to lower this barrier by defining Level 1 primarily around first-layer defense requirements, resulting in clearer and fewer requirements at that level. To demonstrate this numerically, in v4.0.3 there were 128 L1 requirements out of a total of 278 requirements, representing 46%. In 5.0.0 there are 70 L1 requirements out of a total of 345 requirements, representing 20%.

The Fallacy of Testability

A key factor in selecting controls for Level 1 in version 4.x was their suitability for assessment through "black box"external penetration testing. However, this approach was not fully aligned with the intent

of Level 1 as the minimum set of security controls. Some users argued that Level 1 was insufficient for securing applications, while others found it too difficult to test.

Relying on testability as a criterion is both relative and, at times, misleading. The fact that a requirement is testable does not guarantee that it can be tested in an automated or straightforward manner. Moreover, the most easily testable requirements are not always those with the greatest security impact or the simplest to implement.

As such, in version 5.0, the level decisions were made primarily based on risk reduction and also keeping in mind the effort to implement.

Not Just About Risk

The use of prescriptive, risk-based levels that mandate a specific level for certain applications has proven to be overly rigid. In practice, the prioritization and implementation of security controls depend on multiple factors, including both risk reduction and the effort required for implementation.

Therefore, organizations are encouraged to achieve the level that they feel like they should be achieving based on their maturity and the message they want to send to their users.

V1 Encoding and Sanitization

Control Objective

This chapter addresses the most common web application security weaknesses related to the unsafe processing of untrusted data. Such weaknesses can result in various technical vulnerabilities, where untrusted data is interpreted according to the syntax rules of the relevant interpreter.

For modern web applications, it is always best to use safer APIs, such as parameterized queries, auto-escaping, or templating frameworks. Otherwise, carefully performed output encoding, escaping, or sanitization becomes critical to the application's security.

Input validation serves as a defense-in-depth mechanism to protect against unexpected or dangerous content. However, since its primary purpose is to ensure that incoming content matches functional and business expectations, requirements related to this can be found in the "Validation and Business Logic"chapter.

V1.1 Encoding and Sanitization Architecture

In the sections below, syntax-specific or interpreter-specific requirements for safely processing unsafe content to avoid security vulnerabilities are provided. The requirements in this section cover the order in which this processing should occur and where it should take place. They also aim to

ensure that whenever data is stored, it remains in its original state and is not stored in an encoded or escaped form (e.g., HTML encoding), to prevent double encoding issues.

#	Description	Level
1.1.1	Verify that input is decoded or unescaped into a canonical form only once, it is only decoded when encoded data in that form is expected, and that this is done before processing the input further, for example it is not performed after input validation or sanitization.	2
1.1.2	Verify that the application performs output encoding and escaping either as a final step before being used by the interpreter for which it is intended or by the interpreter itself.	2

V1.2 Injection Prevention

Output encoding or escaping, performed close to or adjacent to a potentially dangerous context, is critical to the security of any application. Typically, output encoding and escaping are not persisted, but are instead used to render output safe for immediate use in the appropriate interpreter. Attempting to perform this too early may result in malformed content or render the encoding or escaping ineffective.

In many cases, software libraries include safe or safer functions that perform this automatically, although it is necessary to ensure that they are correct for the current context.

#	Description	Level
1.2.1	Verify that output encoding for an HTTP response, HTML document, or XML document is relevant for the context required, such as encoding the relevant characters for HTML elements, HTML attributes, HTML comments, CSS, or HTTP header fields, to avoid changing the message or document structure.	1
1.2.2	Verify that when dynamically building URLs, untrusted data is encoded according to its context (e.g., URL encoding or base64url encoding for query or path parameters). Ensure that only safe URL protocols are permitted (e.g., disallow javascript: or data:).	1
1.2.3	Verify that output encoding or escaping is used when dynamically building JavaScript content (including JSON), to avoid changing the message or document structure (to avoid JavaScript and JSON injection).	1

#	Description	Level
1.2.4	Verify that data selection or database queries (e.g., SQL, HQL, NoSQL, Cypher) use parameterized queries, ORMs, entity frameworks, or are otherwise protected from SQL Injection and other database injection attacks. This is also relevant when writing stored procedures.	1
1.2.5	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.	1
1.2.6	Verify that the application protects against LDAP injection vulnerabilities, or that specific security controls to prevent LDAP injection have been implemented.	2
1.2.7	Verify that the application is protected against XPath injection attacks by using query parameterization or precompiled queries.	2
1.2.8	Verify that LaTeX processors are configured securely (such as not using the "–shell-escape"flag) and an allowlist of commands is used to prevent LaTeX injection attacks.	2
1.2.9	Verify that the application escapes special characters in regular expressions (typically using a backslash) to prevent them from being misinterpreted as metacharacters.	2
1.2.10	Verify that the application is protected against CSV and Formula Injection. The application must follow the escaping rules defined in RFC 4180 sections 2.6 and 2.7 when exporting CSV content. Additionally, when exporting to CSV or other spreadsheet formats (such as XLS, XLSX, or ODF), special characters (including '=', '+', '-', '@', '\t'(tab), and '\0'(null character)) must be escaped with a single quote if they appear as the first character in a field value.	3

Note: Using parameterized queries or escaping SQL is not always sufficient. Query parts such as table and column names (including "ORDER BY"column names) cannot be escaped. Including escaped user-supplied data in these fields results in failed queries or SQL injection.

V1.3 Sanitization

The ideal protection against using untrusted content in an unsafe context is to use context-specific encoding or escaping, which maintains the same semantic meaning of the unsafe content but renders it safe for use in that particular context, as discussed in more detail in the previous section.

Where this is not possible, sanitization becomes necessary, removing potentially dangerous characters or content. In some cases, this may change the semantic meaning of the input, but for security

reasons, there may be no alternative.

#	Description	Level
1.3.1	Verify that all untrusted HTML input from WYSIWYG editors or similar is sanitized using a well-known and secure HTML sanitization library or framework feature.	1
1.3.2	Verify that the application avoids the use of eval() or other dynamic code execution features such as Spring Expression Language (SpEL). Where there is no alternative, any user input being included must be sanitized before being executed.	1
1.3.3	Verify that data being passed to a potentially dangerous context is sanitized beforehand to enforce safety measures, such as only allowing characters which are safe for this context and trimming input which is too long.	2
1.3.4	Verify that user-supplied Scalable Vector Graphics (SVG) scriptable content is validated or sanitized to contain only tags and attributes (such as draw graphics) that are safe for the application, e.g., do not contain scripts and foreignObject.	2
1.3.5	Verify that the application sanitizes or disables user-supplied scriptable or expression template language content, such as Markdown, CSS or XSL stylesheets, BBCode, or similar.	2
1.3.6	Verify that the application protects against Server-side Request Forgery (SSRF) attacks, by validating untrusted data against an allowlist of protocols, domains, paths and ports and sanitizing potentially dangerous characters before using the data to call another service.	2
1.3.7	Verify that the application protects against template injection attacks by not allowing templates to be built based on untrusted input. Where there is no alternative, any untrusted input being included dynamically during template creation must be sanitized or strictly validated.	2
1.3.8	Verify that the application appropriately sanitizes untrusted input before use in Java Naming and Directory Interface (JNDI) queries and that JNDI is configured securely to prevent JNDI injection attacks.	2
1.3.9	Verify that the application sanitizes content before it is sent to memcache to prevent injection attacks.	2
1.3.10	Verify that format strings which might resolve in an unexpected or malicious way when used are sanitized before being processed.	2
1.3.11	Verify that the application sanitizes user input before passing to mail systems to protect against SMTP or IMAP injection.	2

#	Description	Level
1.3.12	Verify that regular expressions are free from elements causing exponential backtracking, and ensure untrusted input is sanitized to mitigate ReDoS or Runaway Regex attacks.	3

V1.4 Memory, String, and Unmanaged Code

The following requirements address risks associated with unsafe memory use, which generally apply when the application uses a systems language or unmanaged code.

In some cases, it may be possible to achieve this by setting compiler flags that enable buffer overflow protections and warnings, including stack randomization and data execution prevention, and that break the build if unsafe pointer, memory, format string, integer, or string operations are found.

#	Description	Level
1.4.1	Verify that the application uses memory-safe string, safer memory copy and pointer arithmetic to detect or prevent stack, buffer, or heap overflows.	2
1.4.2	Verify that sign, range, and input validation techniques are used to prevent integer overflows.	2
1.4.3	Verify that dynamically allocated memory and resources are released, and that references or pointers to freed memory are removed or set to null to prevent dangling pointers and use-after-free vulnerabilities.	2

V1.5 Safe Deserialization

The conversion of data from a stored or transmitted representation into actual application objects (deserialization) has historically been the cause of various code injection vulnerabilities. It is important to perform this process carefully and safely to avoid these types of issues.

In particular, certain methods of deserialization have been identified by programming language or framework documentation as insecure and cannot be made safe with untrusted data. For each mechanism in use, careful due diligence should be performed.

#	Description	Level
1.5.1	Verify that the application configures XML parsers to use a restrictive configuration and that unsafe features such as resolving external entities are disabled to prevent XML eXternal Entity (XXE) attacks.	1

#	Description	Level
1.5.2	Verify that deserialization of untrusted data enforces safe input handling, such as using an allowlist of object types or restricting client-defined object types, to prevent deserialization attacks. Deserialization mechanisms that are explicitly defined as insecure must not be used with untrusted input.	2
1.5.3	Verify that different parsers used in the application for the same data type (e.g., JSON parsers, XML parsers, URL parsers), perform parsing in a consistent way and use the same character encoding mechanism to avoid issues such as JSON Interoperability vulnerabilities or different URI or file parsing behavior being exploited in Remote File Inclusion (RFI) or Server-side Request Forgery (SSRF) attacks.	3

References

For more information, see also:

- OWASP LDAP Injection Prevention Cheat Sheet
- OWASP Cross Site Scripting Prevention Cheat Sheet
- OWASP DOM Based Cross Site Scripting Prevention Cheat Sheet
- OWASP XML External Entity Prevention Cheat Sheet
- OWASP Web Security Testing Guide: Client-Side Testing
- OWASP Java Encoding Project
- DOMPurify - Client-side HTML Sanitization Library
- RFC4180 - Common Format and MIME Type for Comma-Separated Values (CSV) Files

For more information, specifically on deserialization or parsing issues, please see:

- OWASP Deserialization Cheat Sheet
- An Exploration of JSON Interoperability Vulnerabilities
- Orange Tsai - A New Era of SSRF Exploiting URL Parser In Trending Programming Languages

V2 Validation and Business Logic

Control Objective

This chapter aims to ensure that a verified application meets the following high-level goals:

- Input received by the application matches business or functional expectations.
- The business logic flow is sequential, processed in order, and cannot be bypassed.

- Business logic includes limits and controls to detect and prevent automated attacks, such as continuous small funds transfers or adding a million friends one at a time.
- High-value business logic flows have considered abuse cases and malicious actors, and have protections against spoofing, tampering, information disclosure, and elevation of privilege attacks.

V2.1 Validation and Business Logic Documentation

Validation and business logic documentation should clearly define business logic limits, validation rules, and contextual consistency of combined data items, so it is clear what needs to be implemented in the application.

#	Description	Level
2.1.1	Verify that the application's documentation defines input validation rules for how to check the validity of data items against an expected structure. This could be common data formats such as credit card numbers, email addresses, telephone numbers, or it could be an internal data format.	1
2.1.2	Verify that the application's documentation defines how to validate the logical and contextual consistency of combined data items, such as checking that suburb and ZIP code match.	2
2.1.3	Verify that expectations for business logic limits and validations are documented, including both per-user and globally across the application.	2

V2.2 Input Validation

Effective input validation controls enforce business or functional expectations around the type of data the application expects to receive. This ensures good data quality and reduces the attack surface. However, it does not remove or replace the need to use correct encoding, parameterization, or sanitization when using the data in another component or for presenting it for output.

In this context, "input"could come from a wide variety of sources, including HTML form fields, REST requests, URL parameters, HTTP header fields, cookies, files on disk, databases, and external APIs.

A business logic control might check that a particular input is a number less than 100. A functional expectation might check that a number is below a certain threshold, as that number controls how many times a particular loop will take place, and a high number could lead to excessive processing and a potential denial of service condition.

While schema validation is not explicitly mandated, it may be the most effective mechanism for full validation coverage of HTTP APIs or other interfaces that use JSON or XML.

Please note the following points on Schema Validation:

- The “published version”of the JSON Schema validation specification is considered production-ready, but not strictly speaking “stable.”When using JSON Schema validation, ensure there are no gaps with the guidance in the requirements below.
- Any JSON Schema validation libraries in use should also be monitored and updated if necessary once the standard is formalized.
- DTD validation should not be used, and framework DTD evaluation should be disabled, to avoid issues with XXE attacks against DTDs.

#	Description	Level
2.2.1	Verify that input is validated to enforce business or functional expectations for that input. This should either use positive validation against an allow list of values, patterns, and ranges, or be based on comparing the input to an expected structure and logical limits according to predefined rules. For L1, this can focus on input which is used to make specific business or security decisions. For L2 and up, this should apply to all input.	1
2.2.2	Verify that the application is designed to enforce input validation at a trusted service layer. While client-side validation improves usability and should be encouraged, it must not be relied upon as a security control.	1
2.2.3	Verify that the application ensures that combinations of related data items are reasonable according to the pre-defined rules.	2

V2.3 Business Logic Security

This section considers key requirements to ensure that the application enforces business logic processes in the correct way and is not vulnerable to attacks that exploit the logic and flow of the application.

#	Description	Level
2.3.1	Verify that the application will only process business logic flows for the same user in the expected sequential step order and without skipping steps.	1
2.3.2	Verify that business logic limits are implemented per the application’s documentation to avoid business logic flaws being exploited.	2
2.3.3	Verify that transactions are being used at the business logic level such that either a business logic operation succeeds in its entirety or it is rolled back to the previous correct state.	2

#	Description	Level
2.3.4	Verify that business logic level locking mechanisms are used to ensure that limited quantity resources (such as theater seats or delivery slots) cannot be double-booked by manipulating the application's logic.	2
2.3.5	Verify that high-value business logic flows require multi-user approval to prevent unauthorized or accidental actions. This could include but is not limited to large monetary transfers, contract approvals, access to classified information, or safety overrides in manufacturing.	3

V2.4 Anti-automation

This section includes anti-automation controls to ensure that human-like interactions are required and excessive automated requests are prevented.

#	Description	Level
2.4.1	Verify that anti-automation controls are in place to protect against excessive calls to application functions that could lead to data exfiltration, garbage-data creation, quota exhaustion, rate-limit breaches, denial-of-service, or overuse of costly resources.	2
2.4.2	Verify that business logic flows require realistic human timing, preventing excessively rapid transaction submissions.	3

References

For more information, see also:

- OWASP Web Security Testing Guide: Input Validation Testing
- OWASP Web Security Testing Guide: Business Logic Testing
- Anti-automation can be achieved in many ways, including the use of the OWASP Automated Threats to Web Applications
- OWASP Input Validation Cheat Sheet
- JSON Schema

V3 Web Frontend Security

Control Objective

This category focuses on requirements designed to protect against attacks executed via a web frontend. These requirements do not apply to machine-to-machine solutions.

V3.1 Web Frontend Security Documentation

This section outlines the browser security features that should be specified in the application's documentation.

#	Description	Level
3.1.1	Verify that application documentation states the expected security features that browsers using the application must support (such as HTTPS, HTTP Strict Transport Security (HSTS), Content Security Policy (CSP), and other relevant HTTP security mechanisms). It must also define how the application must behave when some of these features are not available (such as warning the user or blocking access).	3

V3.2 Unintended Content Interpretation

Rendering content or functionality in an incorrect context can result in malicious content being executed or displayed.

#	Description	Level
3.2.1	Verify that security controls are in place to prevent browsers from rendering content or functionality in HTTP responses in an incorrect context (e.g., when an API, a user-uploaded file or other resource is requested directly). Possible controls could include: not serving the content unless HTTP request header fields (such as Sec-Fetch-*) indicate it is the correct context, using the sandbox directive of the Content-Security-Policy header field or using the attachment disposition type in the Content-Disposition header field.	1
3.2.2	Verify that content intended to be displayed as text, rather than rendered as HTML, is handled using safe rendering functions (such as createTextNode or textContent) to prevent unintended execution of content such as HTML or JavaScript.	1

#	Description	Level
3.2.3	Verify that the application avoids DOM clobbering when using client-side JavaScript by employing explicit variable declarations, performing strict type checking, avoiding storing global variables on the document object, and implementing namespace isolation.	3

V3.3 Cookie Setup

This section outlines requirements for securely configuring sensitive cookies to provide a higher level of assurance that they were created by the application itself and to prevent their contents from leaking or being inappropriately modified.

#	Description	Level
3.3.1	Verify that cookies have the 'Secure'attribute set, and if the '__Host-'prefix is not used for the cookie name, the '__Secure-'prefix must be used for the cookie name.	1
3.3.2	Verify that each cookie's 'SameSite'attribute value is set according to the purpose of the cookie, to limit exposure to user interface redress attacks and browser-based request forgery attacks, commonly known as cross-site request forgery (CSRF).	2
3.3.3	Verify that cookies have the '__Host-'prefix for the cookie name unless they are explicitly designed to be shared with other hosts.	2
3.3.4	Verify that if the value of a cookie is not meant to be accessible to client-side scripts (such as a session token), the cookie must have the 'HttpOnly' attribute set and the same value (e. g. session token) must only be transferred to the client via the 'Set-Cookie'header field.	2
3.3.5	Verify that when the application writes a cookie, the cookie name and value length combined are not over 4096 bytes. Overly large cookies will not be stored by the browser and therefore not sent with requests, preventing the user from using application functionality which relies on that cookie.	3

V3.4 Browser Security Mechanism Headers

This section describes which security headers should be set on HTTP responses to enable browser security features and restrictions when handling responses from the application.

#	Description	Level
3.4.1	Verify that a Strict-Transport-Security header field is included on all responses to enforce an HTTP Strict Transport Security (HSTS) policy. A maximum age of at least 1 year must be defined, and for L2 and up, the policy must apply to all subdomains as well.	1
3.4.2	Verify that the Cross-Origin Resource Sharing (CORS) Access-Control-Allow-Origin header field is a fixed value by the application, or if the Origin HTTP request header field value is used, it is validated against an allowlist of trusted origins. When 'Access-Control-Allow-Origin: *'needs to be used, verify that the response does not include any sensitive information.	1
3.4.3	Verify that HTTP responses include a Content-Security-Policy response header field which defines directives to ensure the browser only loads and executes trusted content or resources, in order to limit execution of malicious JavaScript. As a minimum, a global policy must be used which includes the directives object-src 'none'and base-uri 'none'and defines either an allowlist or uses nonces or hashes. For an L3 application, a per-response policy with nonces or hashes must be defined.	2
3.4.4	Verify that all HTTP responses contain an 'X-Content-Type-Options: nosniff' header field. This instructs browsers not to use content sniffing and MIME type guessing for the given response, and to require the response's Content-Type header field value to match the destination resource. For example, the response to a request for a style is only accepted if the response's Content-Type is 'text/css'. This also enables the use of the Cross-Origin Read Blocking (CORB) functionality by the browser.	2
3.4.5	Verify that the application sets a referrer policy to prevent leakage of technically sensitive data to third-party services via the 'Referer'HTTP request header field. This can be done using the Referrer-Policy HTTP response header field or via HTML element attributes. Sensitive data could include path and query data in the URL, and for internal non-public applications also the hostname.	2
3.4.6	Verify that the web application uses the frame-ancestors directive of the Content-Security-Policy header field for every HTTP response to ensure that it cannot be embedded by default and that embedding of specific resources is allowed only when necessary. Note that the X-Frame-Options header field, although supported by browsers, is obsolete and may not be relied upon.	2
3.4.7	Verify that the Content-Security-Policy header field specifies a location to report violations.	3

#	Description	Level
3.4.8	Verify that all HTTP responses that initiate a document rendering (such as responses with Content-Type text/html), include the Cross-Origin-Opener-Policy header field with the same-origin directive or the same-origin-allow-popups directive as required. This prevents attacks that abuse shared access to Window objects, such as tabnabbing and frame counting.	3

V3.5 Browser Origin Separation

When accepting a request to sensitive functionality on the server side, the application needs to ensure the request is initiated by the application itself or by a trusted party and has not been forged by an attacker.

Sensitive functionality in this context could include accepting form posts for authenticated and non-authenticated users (such as an authentication request), state-changing operations, or resource-demanding functionality (such as data export).

The key protections here are browser security policies like Same Origin Policy for JavaScript and also SameSite logic for cookies. Another common protection is the CORS preflight mechanism. This mechanism will be critical for endpoints designed to be called from a different origin, but it can also be a useful request forgery prevention mechanism for endpoints which are not designed to be called from a different origin.

#	Description	Level
3.5.1	Verify that, if the application does not rely on the CORS preflight mechanism to prevent disallowed cross-origin requests to use sensitive functionality, these requests are validated to ensure they originate from the application itself. This may be done by using and validating anti-forgery tokens or requiring extra HTTP header fields that are not CORS-safelisted request-header fields. This is to defend against browser-based request forgery attacks, commonly known as cross-site request forgery (CSRF).	1
3.5.2	Verify that, if the application relies on the CORS preflight mechanism to prevent disallowed cross-origin use of sensitive functionality, it is not possible to call the functionality with a request which does not trigger a CORS-preflight request. This may require checking the values of the 'Origin' and 'Content-Type'request header fields or using an extra header field that is not a CORS-safelisted header-field.	1

#	Description	Level
3.5.3	Verify that HTTP requests to sensitive functionality use appropriate HTTP methods such as POST, PUT, PATCH, or DELETE, and not methods defined by the HTTP specification as “safe”such as HEAD, OPTIONS, or GET. Alternatively, strict validation of the Sec-Fetch-* request header fields can be used to ensure that the request did not originate from an inappropriate cross-origin call, a navigation request, or a resource load (such as an image source) where this is not expected.	1
3.5.4	Verify that separate applications are hosted on different hostnames to leverage the restrictions provided by same-origin policy, including how documents or scripts loaded by one origin can interact with resources from another origin and hostname-based restrictions on cookies.	2
3.5.5	Verify that messages received by the postMessage interface are discarded if the origin of the message is not trusted, or if the syntax of the message is invalid.	2
3.5.6	Verify that JSONP functionality is not enabled anywhere across the application to avoid Cross-Site Script Inclusion (XSSI) attacks.	3
3.5.7	Verify that data requiring authorization is not included in script resource responses, like JavaScript files, to prevent Cross-Site Script Inclusion (XSSI) attacks.	3
3.5.8	Verify that authenticated resources (such as images, videos, scripts, and other documents) can be loaded or embedded on behalf of the user only when intended. This can be accomplished by strict validation of the Sec-Fetch-* HTTP request header fields to ensure that the request did not originate from an inappropriate cross-origin call, or by setting a restrictive Cross-Origin-Resource-Policy HTTP response header field to instruct the browser to block returned content.	3

V3.6 External Resource Integrity

This section provides guidance for the safe hosting of content on third-party sites.

#	Description	Level
3.6.1	Verify that client-side assets, such as JavaScript libraries, CSS, or web fonts, are only hosted externally (e.g., on a Content Delivery Network) if the resource is static and versioned and Subresource Integrity (SRI) is used to validate the integrity of the asset. If this is not possible, there should be a documented security decision to justify this for each resource.	3

V3.7 Other Browser Security Considerations

This section includes various other security controls and modern browser security features required for client-side browser security.

#	Description	Level
3.7.1	Verify that the application only uses client-side technologies which are still supported and considered secure. Examples of technologies which do not meet this requirement include NSAPI plugins, Flash, Shockwave, ActiveX, Silverlight, NACL, or client-side Java applets.	2
3.7.2	Verify that the application will only automatically redirect the user to a different hostname or domain (which is not controlled by the application) where the destination appears on an allowlist.	2
3.7.3	Verify that the application shows a notification when the user is being redirected to a URL outside of the application's control, with an option to cancel the navigation.	3
3.7.4	Verify that the application's top-level domain (e.g., site.tld) is added to the public preload list for HTTP Strict Transport Security (HSTS). This ensures that the use of TLS for the application is built directly into the main browsers, rather than relying only on the Strict-Transport-Security response header field.	3
3.7.5	Verify that the application behaves as documented (such as warning the user or blocking access) if the browser used to access the application does not support the expected security features.	3

References

For more information, see also:

- Set-Cookie __Host- prefix details

- OWASP Content Security Policy Cheat Sheet
- OWASP Secure Headers Project
- OWASP Cross-Site Request Forgery Prevention Cheat Sheet
- HSTS Browser Preload List submission form
- OWASP DOM Clobbering Prevention Cheat Sheet

V4 API and Web Service

Control Objective

Several considerations apply specifically to applications that expose APIs for use by web browsers or other consumers (commonly using JSON, XML, or GraphQL). This chapter covers the relevant security configurations and mechanisms that should be applied.

Note that authentication, session management, and input validation concerns from other chapters also apply to APIs, so this chapter cannot be taken out of context or tested in isolation.

V4.1 Generic Web Service Security

This section addresses general web service security considerations and, consequently, basic web service hygiene practices.

#	Description	Level
4.1.1	Verify that every HTTP response with a message body contains a Content-Type header field that matches the actual content of the response, including the charset parameter to specify safe character encoding (e.g., UTF-8, ISO-8859-1) according to IANA Media Types, such as “text/”, “/+xml” and “/xml”.	1
4.1.2	Verify that only user-facing endpoints (intended for manual web-browser access) automatically redirect from HTTP to HTTPS, while other services or endpoints do not implement transparent redirects. This is to avoid a situation where a client is erroneously sending unencrypted HTTP requests, but since the requests are being automatically redirected to HTTPS, the leakage of sensitive data goes undiscovered.	2
4.1.3	Verify that any HTTP header field used by the application and set by an intermediary layer, such as a load balancer, a web proxy, or a backend-for-frontend service, cannot be overridden by the end-user. Example headers might include X-Real-IP, X-Forwarded-*, or X-User-ID.	2

#	Description	Level
4.1.4	Verify that only HTTP methods that are explicitly supported by the application or its API (including OPTIONS during preflight requests) can be used and that unused methods are blocked.	3
4.1.5	Verify that per-message digital signatures are used to provide additional assurance on top of transport protections for requests or transactions which are highly sensitive or which traverse a number of systems.	3

V4.2 HTTP Message Structure Validation

This section explains how the structure and header fields of an HTTP message should be validated to prevent attacks such as request smuggling, response splitting, header injection, and denial of service via overly long HTTP messages.

These requirements are relevant for general HTTP message processing and generation, but are especially important when converting HTTP messages between different HTTP versions.

#	Description	Level
4.2.1	Verify that all application components (including load balancers, firewalls, and application servers) determine boundaries of incoming HTTP messages using the appropriate mechanism for the HTTP version to prevent HTTP request smuggling. In HTTP/1.x, if a Transfer-Encoding header field is present, the Content-Length header must be ignored per RFC 2616. When using HTTP/2 or HTTP/3, if a Content-Length header field is present, the receiver must ensure that it is consistent with the length of the DATA frames.	2
4.2.2	Verify that when generating HTTP messages, the Content-Length header field does not conflict with the length of the content as determined by the framing of the HTTP protocol, in order to prevent request smuggling attacks.	3
4.2.3	Verify that the application does not send nor accept HTTP/2 or HTTP/3 messages with connection-specific header fields such as Transfer-Encoding to prevent response splitting and header injection attacks.	3
4.2.4	Verify that the application only accepts HTTP/2 and HTTP/3 requests where the header fields and values do not contain any CR (`\r`), LF (`\n`), or CRLF (`\r\n`) sequences, to prevent header injection attacks.	3

#	Description	Level
4.2.5	Verify that, if the application (backend or frontend) builds and sends requests, it uses validation, sanitization, or other mechanisms to avoid creating URIs (such as for API calls) or HTTP request header fields (such as Authorization or Cookie), which are too long to be accepted by the receiving component. This could cause a denial of service, such as when sending an overly long request (e.g., a long cookie header field), which results in the server always responding with an error status.	3

V4.3 GraphQL

GraphQL is becoming more common as a way of creating data-rich clients that are not tightly coupled to a variety of backend services. This section covers security considerations for GraphQL.

#	Description	Level
4.3.1	Verify that a query allowlist, depth limiting, amount limiting, or query cost analysis is used to prevent GraphQL or data layer expression Denial of Service (DoS) as a result of expensive, nested queries.	2
4.3.2	Verify that GraphQL introspection queries are disabled in the production environment unless the GraphQL API is meant to be used by other parties.	2

V4.4 WebSocket

WebSocket is a communications protocol that provides a simultaneous two-way communication channel over a single TCP connection. It was standardized by the IETF as RFC 6455 in 2011 and is distinct from HTTP, even though it is designed to work over HTTP ports 443 and 80.

This section provides key security requirements to prevent attacks related to communication security and session management that specifically exploit this real-time communication channel.

#	Description	Level
4.4.1	Verify that WebSocket over TLS (WSS) is used for all WebSocket connections.	1
4.4.2	Verify that, during the initial HTTP WebSocket handshake, the Origin header field is checked against a list of origins allowed for the application.	2

#	Description	Level
4.4.3	Verify that, if the application's standard session management cannot be used, dedicated tokens are being used for this, which comply with the relevant Session Management security requirements.	2
4.4.4	Verify that dedicated WebSocket session management tokens are initially obtained or validated through the previously authenticated HTTPS session when transitioning an existing HTTPS session to a WebSocket channel.	2

References

For more information, see also:

- OWASP REST Security Cheat Sheet
- Resources on GraphQL Authorization from graphql.org and Apollo.
- OWASP Web Security Testing Guide: GraphQL Testing
- OWASP Web Security Testing Guide: Testing WebSockets

V5 File Handling

Control Objective

The use of files can present a variety of risks to the application, including denial of service, unauthorized access, and storage exhaustion. This chapter includes requirements to address these risks.

V5.1 File Handling Documentation

This section includes a requirement to document the expected characteristics of files accepted by the application, as a necessary precondition for developing and verifying relevant security checks.

#	Description	Level
5.1.1	Verify that the documentation defines the permitted file types, expected file extensions, and maximum size (including unpacked size) for each upload feature. Additionally, ensure that the documentation specifies how files are made safe for end-users to download and process, such as how the application behaves when a malicious file is detected.	2

V5.2 File Upload and Content

File upload functionality is a primary source of untrusted files. This section outlines the requirements for ensuring that the presence, volume, or content of these files cannot harm the application.

#	Description	Level
5.2.1	Verify that the application will only accept files of a size which it can process without causing a loss of performance or a denial of service attack.	1
5.2.2	Verify that when the application accepts a file, either on its own or within an archive such as a zip file, it checks if the file extension matches an expected file extension and validates that the contents correspond to the type represented by the extension. This includes, but is not limited to, checking the initial 'magic bytes', performing image re-writing, and using specialized libraries for file content validation. For L1, this can focus just on files which are used to make specific business or security decisions. For L2 and up, this must apply to all files being accepted.	1
5.2.3	Verify that the application checks compressed files (e.g., zip, gz, docx, odt) against maximum allowed uncompressed size and against maximum number of files before uncompressing the file.	2
5.2.4	Verify that a file size quota and maximum number of files per user are enforced to ensure that a single user cannot fill up the storage with too many files, or excessively large files.	3
5.2.5	Verify that the application does not allow uploading compressed files containing symlinks unless this is specifically required (in which case it will be necessary to enforce an allowlist of the files that can be symlinked to).	3
5.2.6	Verify that the application rejects uploaded images with a pixel size larger than the maximum allowed, to prevent pixel flood attacks.	3

V5.3 File Storage

This section includes requirements to prevent files from being inappropriately executed after upload, to detect dangerous content, and to avoid untrusted data being used to control where files are being stored.

#	Description	Level
5.3.1	Verify that files uploaded or generated by untrusted input and stored in a public folder, are not executed as server-side program code when accessed directly with an HTTP request.	1
5.3.2	Verify that when the application creates file paths for file operations, instead of user-submitted filenames, it uses internally generated or trusted data, or if user-submitted filenames or file metadata must be used, strict validation and sanitization must be applied. This is to protect against path traversal, local or remote file inclusion (LFI, RFI), and server-side request forgery (SSRF) attacks.	1
5.3.3	Verify that server-side file processing, such as file decompression, ignores user-provided path information to prevent vulnerabilities such as zip slip.	3

V5.4 File Download

This section contains requirements to mitigate risks when serving files to be downloaded, including path traversal and injection attacks. This also includes making sure they don't contain dangerous content.

#	Description	Level
5.4.1	Verify that the application validates or ignores user-submitted filenames, including in a JSON, JSONP, or URL parameter and specifies a filename in the Content-Disposition header field in the response.	2
5.4.2	Verify that file names served (e.g., in HTTP response header fields or email attachments) are encoded or sanitized (e.g., following RFC 6266) to preserve document structure and prevent injection attacks.	2
5.4.3	Verify that files obtained from untrusted sources are scanned by antivirus scanners to prevent serving of known malicious content.	2

References

For more information, see also:

- OWASP File Upload Cheat Sheet
- Example of using symlinks for arbitrary file read
- Explanation of "Magic Bytes"from Wikipedia

V6 Authentication

Control Objective

Authentication is the process of establishing or confirming the authenticity of an individual or device. It involves verifying claims made by a person or about a device, ensuring resistance to impersonation, and preventing the recovery or interception of passwords.

NIST SP 800-63 is a modern, evidence-based standard that is valuable for organizations worldwide, but is particularly relevant to US agencies and those interacting with US agencies.

While many of the requirements in this chapter are based on the second section of the standard (known as NIST SP 800-63B “Digital Identity Guidelines - Authentication and Lifecycle Management”), the chapter focuses on common threats and frequently exploited authentication weaknesses. It does not attempt to comprehensively cover every point in the standard. For cases where full NIST SP 800-63 compliance is necessary, please refer to NIST SP 800-63.

Additionally, NIST SP 800-63 terminology may sometimes differ, and this chapter often uses more commonly understood terminology to improve clarity.

A common feature of more advanced applications is the ability to adapt authentication stages required based on various risk factors. This feature is covered in the “Authorization”chapter, since these mechanisms also need to be considered for authorization decisions.

V6.1 Authentication Documentation

This section contains requirements detailing the authentication documentation that should be maintained for an application. This is crucial for implementing and assessing how the relevant authentication controls should be configured.

#	Description	Level
6.1.1	Verify that application documentation defines how controls such as rate limiting, anti-automation, and adaptive response, are used to defend against attacks such as credential stuffing and password brute force. The documentation must make clear how these controls are configured and prevent malicious account lockout.	1
6.1.2	Verify that a list of context-specific words is documented in order to prevent their use in passwords. The list could include permutations of organization names, product names, system identifiers, project codenames, department or role names, and similar.	2

#	Description	Level
6.1.3	Verify that, if the application includes multiple authentication pathways, these are all documented together with the security controls and authentication strength which must be consistently enforced across them.	2

V6.2 Password Security

Passwords, called “Memorized Secrets”by NIST SP 800-63, include passwords, passphrases, PINs, unlock patterns, and picking the correct kitten or another image element. They are generally considered “something you know”and are often used as a single-factor authentication mechanism.

As such, this section contains requirements for making sure that passwords are created and handled securely. Most of the requirements are L1 as they are most important at that level. From L2 onwards, multi-factor authentication mechanisms are required, where passwords may be one of those factors.

The requirements in this section mostly relate to § 5.1.1.2 of NIST’s Guidance.

#	Description	Level
6.2.1	Verify that user set passwords are at least 8 characters in length although a minimum of 15 characters is strongly recommended.	1
6.2.2	Verify that users can change their password.	1
6.2.3	Verify that password change functionality requires the user’s current and new password.	1
6.2.4	Verify that passwords submitted during account registration or password change are checked against an available set of, at least, the top 3000 passwords which match the application’s password policy, e.g. minimum length.	1
6.2.5	Verify that passwords of any composition can be used, without rules limiting the type of characters permitted. There must be no requirement for a minimum number of upper or lower case characters, numbers, or special characters.	1
6.2.6	Verify that password input fields use type=password to mask the entry. Applications may allow the user to temporarily view the entire masked password, or the last typed character of the password.	1
6.2.7	Verify that “paste”functionality, browser password helpers, and external password managers are permitted.	1

#	Description	Level
6.2.8	Verify that the application verifies the user's password exactly as received from the user, without any modifications such as truncation or case transformation.	1
6.2.9	Verify that passwords of at least 64 characters are permitted.	2
6.2.10	Verify that a user's password stays valid until it is discovered to be compromised or the user rotates it. The application must not require periodic credential rotation.	2
6.2.11	Verify that the documented list of context specific words is used to prevent easy to guess passwords being created.	2
6.2.12	Verify that passwords submitted during account registration or password changes are checked against a set of breached passwords.	2

V6.3 General Authentication Security

This section contains general requirements for the security of authentication mechanisms as well as setting out the different expectations for levels. L2 applications must force the use of multi-factor authentication (MFA). L3 applications must use hardware-based authentication, performed in an attested and trusted execution environment (TEE). This could include device-bound passkeys, eIDAS Level of Assurance (LoA) High enforced authenticators, authenticators with NIST Authenticator Assurance Level 3 (AAL3) assurance, or an equivalent mechanism.

While this is a relatively aggressive stance on MFA, it is critical to raise the bar around this to protect users, and any attempt to relax these requirements should be accompanied by a clear plan on how the risks around authentication will be mitigated, taking into account NIST's guidance and research on the topic.

Note that at the time of release, NIST SP 800-63 considers email as not acceptable as an authentication mechanism (archived copy).

The requirements in this section relate to a variety of sections of NIST's Guidance, including: § 4.2.1, § 4.3.1, § 5.2.2, and § 6.1.2.

#	Description	Level
6.3.1	Verify that controls to prevent attacks such as credential stuffing and password brute force are implemented according to the application's security documentation.	1
6.3.2	Verify that default user accounts (e.g., "root", "admin", or "sa") are not present in the application or are disabled.	1

#	Description	Level
6.3.3	Verify that either a multi-factor authentication mechanism or a combination of single-factor authentication mechanisms, must be used in order to access the application. For L3, one of the factors must be a hardware-based authentication mechanism which provides compromise and impersonation resistance against phishing attacks while verifying the intent to authenticate by requiring a user-initiated action (such as a button press on a FIDO hardware key or a mobile phone). Relaxing any of the considerations in this requirement requires a fully documented rationale and a comprehensive set of mitigating controls.	2
6.3.4	Verify that, if the application includes multiple authentication pathways, there are no undocumented pathways and that security controls and authentication strength are enforced consistently.	2
6.3.5	Verify that users are notified of suspicious authentication attempts (successful or unsuccessful). This may include authentication attempts from an unusual location or client, partially successful authentication (only one of multiple factors), an authentication attempt after a long period of inactivity or a successful authentication after several unsuccessful attempts.	3
6.3.6	Verify that email is not used as either a single-factor or multi-factor authentication mechanism.	3
6.3.7	Verify that users are notified after updates to authentication details, such as credential resets or modification of the username or email address.	3
6.3.8	Verify that valid users cannot be deduced from failed authentication challenges, such as by basing on error messages, HTTP response codes, or different response times. Registration and forgot password functionality must also have this protection.	3

V6.4 Authentication Factor Lifecycle and Recovery

Authentication factors may include passwords, soft tokens, hardware tokens, and biometric devices. Securely handling the lifecycle of these mechanisms is critical to the security of an application, and this section includes requirements related to this.

The requirements in this section mostly relate to § 5.1.1.2 or § 6.1.2.3 of NIST's Guidance.

#	Description	Level
6.4.1	Verify that system generated initial passwords or activation codes are securely randomly generated, follow the existing password policy, and expire after a short period of time or after they are initially used. These initial secrets must not be permitted to become the long term password.	1
6.4.2	Verify that password hints or knowledge-based authentication (so-called "secret questions") are not present.	1
6.4.3	Verify that a secure process for resetting a forgotten password is implemented, that does not bypass any enabled multi-factor authentication mechanisms.	2
6.4.4	Verify that if a multi-factor authentication factor is lost, evidence of identity proofing is performed at the same level as during enrollment.	2
6.4.5	Verify that renewal instructions for authentication mechanisms which expire are sent with enough time to be carried out before the old authentication mechanism expires, configuring automated reminders if necessary.	3
6.4.6	Verify that administrative users can initiate the password reset process for the user, but that this does not allow them to change or choose the user's password. This prevents a situation where they know the user's password.	3

V6.5 General Multi-factor authentication requirements

This section provides general guidance that will be relevant to various different multi-factor authentication methods.

The mechanisms include:

- Lookup Secrets
- Time based One-time Passwords (TOTPs)
- Out-of-Band mechanisms

Lookup 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. This type of authentication mechanism is considered "something you have"because the codes are deliberately not memorable so will need to be stored somewhere.

Time based One-time Passwords (TOTPs) are physical or soft tokens that display a continually changing pseudo-random one-time challenge. This type of authentication mechanism is considered "something you have". Multi-factor TOTPs are similar to single-factor TOTPs, 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 One-time Password (OTP).

Details on out-of-band mechanisms will be provided in the next section.

The requirements in these sections mostly relate to § 5.1.2, § 5.1.3, § 5.1.4.2, § 5.1.5.2, § 5.2.1, and § 5.2.3 of NIST's Guidance.

#	Description	Level
6.5.1	Verify that lookup secrets, out-of-band authentication requests or codes, and time-based one-time passwords (TOTPs) are only successfully usable once.	2
6.5.2	Verify that, when being stored in the application's backend, lookup secrets with less than 112 bits of entropy (19 random alphanumeric characters or 34 random digits) are hashed with an approved password storage hashing algorithm that incorporates a 32-bit random salt. A standard hash function can be used if the secret has 112 bits of entropy or more.	2
6.5.3	Verify that lookup secrets, out-of-band authentication code, and time-based one-time password seeds, are generated using a Cryptographically Secure Pseudorandom Number Generator (CSPRNG) to avoid predictable values.	2
6.5.4	Verify that lookup secrets and out-of-band authentication codes have a minimum of 20 bits of entropy (typically 4 random alphanumeric characters or 6 random digits is sufficient).	2
6.5.5	Verify that out-of-band authentication requests, codes, or tokens, as well as time-based one-time passwords (TOTPs) have a defined lifetime. Out of band requests must have a maximum lifetime of 10 minutes and for TOTP a maximum lifetime of 30 seconds.	2
6.5.6	Verify that any authentication factor (including physical devices) can be revoked in case of theft or other loss.	3
6.5.7	Verify that biometric authentication mechanisms are only used as secondary factors together with either something you have or something you know.	3
6.5.8	Verify that time-based one-time passwords (TOTPs) are checked based on a time source from a trusted service and not from an untrusted or client provided time.	3

V6.6 Out-of-Band authentication mechanisms

This usually involves the authentication server communicating with a physical device over a secure secondary channel. For example, sending push notifications to mobile devices. This type of authentication mechanism is considered "something you have".

Unsafe out-of-band authentication mechanisms such as e-mail and VOIP are not permitted. PSTN and SMS authentication are currently considered to be "restricted"authentication mechanisms by

NIST and should be deprecated in favor of Time based One-time Passwords (TOTPs), a cryptographic mechanism, or similar. NIST SP 800-63B § 5.1.3.3 recommends addressing the risks of device swap, SIM change, number porting, or other abnormal behavior, if telephone or SMS out-of-band authentication absolutely has to be supported. While this ASVS section does not mandate this as a requirement, not taking these precautions for a sensitive L2 app or an L3 app should be seen as a significant red flag.

Note that NIST has also recently provided guidance which discourages the use of push notifications. While this ASVS section does not do so, it is important to be aware of the risks of “push bombing”.

#	Description	Level
6.6.1	Verify that authentication mechanisms using the Public Switched Telephone Network (PSTN) to deliver One-time Passwords (OTPs) via phone or SMS are offered only when the phone number has previously been validated, alternate stronger methods (such as Time based One-time Passwords) are also offered, and the service provides information on their security risks to users. For L3 applications, phone and SMS must not be available as options.	2
6.6.2	Verify that out-of-band authentication requests, codes, or tokens are bound to the original authentication request for which they were generated and are not usable for a previous or subsequent one.	2
6.6.3	Verify that a code based out-of-band authentication mechanism is protected against brute force attacks by using rate limiting. Consider also using a code with at least 64 bits of entropy.	2
6.6.4	Verify that, where push notifications are used for multi-factor authentication, rate limiting is used to prevent push bombing attacks. Number matching may also mitigate this risk.	3

V6.7 Cryptographic authentication mechanism

Cryptographic authentication mechanisms include smart cards or FIDO keys, where the user has to plug in or pair the cryptographic device to the computer to complete authentication. The authentication server will send a challenge nonce to the cryptographic device or software, and the device or software calculates a response based upon a securely stored cryptographic key. The requirements in this section provide implementation-specific guidance for these mechanisms, with guidance on cryptographic algorithms being covered in the “Cryptography”chapter.

Where shared or secret keys are used for cryptographic authentication, these should be stored using the same mechanisms as other system secrets, as documented in the “Secret Management”section in the “Configuration”chapter.

The requirements in this section mostly relate to § 5.1.7.2 of NIST’s Guidance.

#	Description	Level
6.7.1	Verify that the certificates used to verify cryptographic authentication assertions are stored in a way protects them from modification.	3
6.7.2	Verify that the challenge nonce is at least 64 bits in length, and statistically unique or unique over the lifetime of the cryptographic device.	3

V6.8 Authentication with an Identity Provider

Identity Providers (IdPs) provide federated identity for users. Users will often have more than one identity with multiple IdPs, such as an enterprise identity using Azure AD, Okta, Ping Identity, or Google, or consumer identity using Facebook, Twitter, Google, or WeChat, to name just a 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 IdP'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.

Secure use of external identity providers requires careful configuration and verification to prevent identity spoofing or forged assertions. This section provides requirements to address these risks.

#	Description	Level
6.8.1	Verify that, if the application supports multiple identity providers (IdPs), the user's identity cannot be spoofed via another supported identity provider (eg. by using the same user identifier). The standard mitigation would be for the application to register and identify the user using a combination of the IdP ID (serving as a namespace) and the user's ID in the IdP.	2
6.8.2	Verify that the presence and integrity of digital signatures on authentication assertions (for example on JWTs or SAML assertions) are always validated, rejecting any assertions that are unsigned or have invalid signatures.	2
6.8.3	Verify that SAML assertions are uniquely processed and used only once within the validity period to prevent replay attacks.	2

#	Description	Level
6.8.4	Verify that, if an application uses a separate Identity Provider (IdP) and expects specific authentication strength, methods, or recentness for specific functions, the application verifies this using the information returned by the IdP. For example, if OIDC is used, this might be achieved by validating ID Token claims such as 'acr', 'amr', and 'auth_time'(if present). If the IdP does not provide this information, the application must have a documented fallback approach that assumes that the minimum strength authentication mechanism was used (for example, single-factor authentication using username and password).	2

References

For more information, see also:

- NIST SP 800-63 - Digital Identity Guidelines
- NIST SP 800-63B - Authentication and Lifecycle Management
- NIST SP 800-63 FAQ
- OWASP Web Security Testing Guide: Testing for Authentication
- OWASP Password Storage Cheat Sheet
- OWASP Forgot Password Cheat Sheet
- OWASP Choosing and Using Security Questions Cheat Sheet
- CISA Guidance on "Number Matching"
- Details on the FIDO Alliance

V7 Session Management

Control Objective

Session management mechanisms allow applications to correlate user and device interactions over time, even when using stateless communication protocols (such as HTTP). Modern applications may use multiple session tokens with distinct characteristics and purposes. A secure session management system is one that prevents attackers from obtaining, utilizing, or otherwise abusing a victim's session. Applications maintaining sessions must ensure that the following high-level session management requirements are met:

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

Many of the requirements in this chapter relate to selected NIST SP 800-63 Digital Identity Guidelines controls, focusing on common threats and commonly exploited authentication weaknesses.

Note that requirements for specific implementation details of certain session management mechanisms can be found elsewhere:

- HTTP Cookies are a common mechanism for securing session tokens. Specific security requirements for cookies can be found in the “Web Frontend Security”chapter.
- Self-contained tokens are frequently used as a way of maintaining sessions. Specific security requirements can be found in the “Self-contained Tokens”chapter.

V7.1 Session Management Documentation

There is no single pattern that suits all applications. Therefore, it is not feasible to define universal boundaries and limits that suit all cases. A risk analysis with documented security decisions related to session handling must be conducted as a prerequisite to implementation and testing. This ensures that the session management system is tailored to the specific requirements of the application.

Regardless of whether a stateful or “stateless”session mechanism is chosen, the analysis must be complete and documented to demonstrate that the selected solution is capable of satisfying all relevant security requirements. Interaction with any Single Sign-on (SSO) mechanisms in use should also be considered.

#	Description	Level
7.1.1	Verify that the user’s session inactivity timeout and absolute maximum session lifetime are documented, are appropriate in combination with other controls, and that the documentation includes justification for any deviations from NIST SP 800-63B re-authentication requirements.	2
7.1.2	Verify that the documentation defines how many concurrent (parallel) sessions are allowed for one account as well as the intended behaviors and actions to be taken when the maximum number of active sessions is reached.	2
7.1.3	Verify that all systems that create and manage user sessions as part of a federated identity management ecosystem (such as SSO systems) are documented along with controls to coordinate session lifetimes, termination, and any other conditions that require re-authentication.	2

V7.2 Fundamental Session Management Security

This section satisfies the essential requirements of secure sessions by verifying that session tokens are securely generated and validated.

#	Description	Level
7.2.1	Verify that the application performs all session token verification using a trusted, backend service.	1
7.2.2	Verify that the application uses either self-contained or reference tokens that are dynamically generated for session management, i.e. not using static API secrets and keys.	1
7.2.3	Verify that if reference tokens are used to represent user sessions, they are unique and generated using a cryptographically secure pseudo-random number generator (CSPRNG) and possess at least 128 bits of entropy.	1
7.2.4	Verify that the application generates a new session token on user authentication, including re-authentication, and terminates the current session token.	1

V7.3 Session Timeout

Session timeout mechanisms serve to minimize the window of opportunity for session hijacking and other forms of session abuse. Timeouts must satisfy documented security decisions.

#	Description	Level
7.3.1	Verify that there is an inactivity timeout such that re-authentication is enforced according to risk analysis and documented security decisions.	2
7.3.2	Verify that there is an absolute maximum session lifetime such that re-authentication is enforced according to risk analysis and documented security decisions.	2

V7.4 Session Termination

Session termination may be handled either by the application itself or by the SSO provider if the SSO provider is handling session management instead of the application. It may be necessary to decide whether the SSO provider is in scope when considering the requirements in this section as some may be controlled by the provider.

Session termination should result in requiring re-authentication and be effective across the application, federated login (if present), and any relying parties.

For stateful session mechanisms, termination typically involves invalidating the session on the backend. In the case of self-contained tokens, additional measures are required to revoke or block these tokens, as they may otherwise remain valid until expiration.

#	Description	Level
7.4.1	Verify that when session termination is triggered (such as logout or expiration), the application disallows any further use of the session. For reference tokens or stateful sessions, this means invalidating the session data at the application backend. Applications using self-contained tokens will need a solution such as maintaining a list of terminated tokens, disallowing tokens produced before a per-user date and time or rotating a per-user signing key.	1
7.4.2	Verify that the application terminates all active sessions when a user account is disabled or deleted (such as an employee leaving the company).	1
7.4.3	Verify that the application gives the option to terminate all other active sessions after a successful change or removal of any authentication factor (including password change via reset or recovery and, if present, an MFA settings update).	2
7.4.4	Verify that all pages that require authentication have easy and visible access to logout functionality.	2
7.4.5	Verify that application administrators are able to terminate active sessions for an individual user or for all users.	2

V7.5 Defenses Against Session Abuse

This section provides requirements to mitigate the risk posed by active sessions that are either hijacked or abused through vectors that rely on the existence and capabilities of active user sessions. For example, using malicious content execution to force an authenticated victim browser to perform an action using the victim's session.

Note that the level-specific guidance in the "Authentication"chapter should be taken into account when considering requirements in this section.

#	Description	Level
7.5.1	Verify that the application requires full re-authentication before allowing modifications to sensitive account attributes which may affect authentication such as email address, phone number, MFA configuration, or other information used in account recovery.	2
7.5.2	Verify that users are able to view and (having authenticated again with at least one factor) terminate any or all currently active sessions.	2

#	Description	Level
7.5.3	Verify that the application requires further authentication with at least one factor or secondary verification before performing highly sensitive transactions or operations.	3

V7.6 Federated Re-authentication

This section relates to those writing Relying Party (RP) or Identity Provider (IdP) code. These requirements are derived from the NIST SP 800-63C for Federation & Assertions.

#	Description	Level
7.6.1	Verify that session lifetime and termination between Relying Parties (RPs) and Identity Providers (IdPs) behave as documented, requiring re-authentication as necessary such as when the maximum time between IdP authentication events is reached.	2
7.6.2	Verify that creation of a session requires either the user's consent or an explicit action, preventing the creation of new application sessions without user interaction.	2

References

For more information, see also:

- OWASP Web Security Testing Guide: Session Management Testing
- OWASP Session Management Cheat Sheet

V8 Authorization

Control Objective

Authorization ensures that access is granted only to permitted consumers (users, servers, and other clients). To enforce the Principle of Least Privilege (POLP), verified applications must meet the following high-level requirements:

- Document authorization rules, including decision-making factors and environmental contexts.
- Consumers should have access only to resources permitted by their defined entitlements.

V8.1 Authorization Documentation

Comprehensive authorization documentation is essential to ensure that security decisions are consistently applied, auditable, and aligned with organizational policies. This reduces the risk of unauthorized access by making security requirements clear and actionable for developers, administrators, and testers.

#	Description	Level
8.1.1	Verify that authorization documentation defines rules for restricting function-level and data-specific access based on consumer permissions and resource attributes.	1
8.1.2	Verify that authorization documentation defines rules for field-level access restrictions (both read and write) based on consumer permissions and resource attributes. Note that these rules might depend on other attribute values of the relevant data object, such as state or status.	2
8.1.3	Verify that the application's documentation defines the environmental and contextual attributes (including but not limited to, time of day, user location, IP address, or device) that are used in the application to make security decisions, including those pertaining to authentication and authorization.	3
8.1.4	Verify that authentication and authorization documentation defines how environmental and contextual factors are used in decision-making, in addition to function-level, data-specific, and field-level authorization. This should include the attributes evaluated, thresholds for risk, and actions taken (e.g., allow, challenge, deny, step-up authentication).	3

V8.2 General Authorization Design

Implementing granular authorization controls at the function, data, and field levels ensures that consumers can access only what has been explicitly granted to them.

#	Description	Level
8.2.1	Verify that the application ensures that function-level access is restricted to consumers with explicit permissions.	1
8.2.2	Verify that the application ensures that data-specific access is restricted to consumers with explicit permissions to specific data items to mitigate insecure direct object reference (IDOR) and broken object level authorization (BOLA).	1

#	Description	Level
8.2.3	Verify that the application ensures that field-level access is restricted to consumers with explicit permissions to specific fields to mitigate broken object property level authorization (BOPLA).	2
8.2.4	Verify that adaptive security controls based on a consumer's environmental and contextual attributes (such as time of day, location, IP address, or device) are implemented for authentication and authorization decisions, as defined in the application's documentation. These controls must be applied when the consumer tries to start a new session and also during an existing session.	3

V8.3 Operation Level Authorization

The immediate application of authorization changes in the appropriate tier of an application's architecture is crucial to preventing unauthorized actions, especially in dynamic environments.

#	Description	Level
8.3.1	Verify that the application enforces authorization rules at a trusted service layer and doesn't rely on controls that an untrusted consumer could manipulate, such as client-side JavaScript.	1
8.3.2	Verify that changes to values on which authorization decisions are made are applied immediately. Where changes cannot be applied immediately, (such as when relying on data in self-contained tokens), there must be mitigating controls to alert when a consumer performs an action when they are no longer authorized to do so and revert the change. Note that this alternative would not mitigate information leakage.	3
8.3.3	Verify that access to an object is based on the originating subject's (e.g. consumer's) permissions, not on the permissions of any intermediary or service acting on their behalf. For example, if a consumer calls a web service using a self-contained token for authentication, and the service then requests data from a different service, the second service will use the consumer's token, rather than a machine-to-machine token from the first service, to make permission decisions.	3

V8.4 Other Authorization Considerations

Additional considerations for authorization, particularly for administrative interfaces and multi-tenant environments, help prevent unauthorized access.

#	Description	Level
8.4.1	Verify that multi-tenant applications use cross-tenant controls to ensure consumer operations will never affect tenants with which they do not have permissions to interact.	2
8.4.2	Verify that access to administrative interfaces incorporates multiple layers of security, including continuous consumer identity verification, device security posture assessment, and contextual risk analysis, ensuring that network location or trusted endpoints are not the sole factors for authorization even though they may reduce the likelihood of unauthorized access.	3

References

For more information, see also:

- OWASP Web Security Testing Guide: Authorization
- OWASP Authorization Cheat Sheet

V9 Self-contained Tokens

Control Objective

The concept of a self-contained token is mentioned in the original RFC 6749 OAuth 2.0 from 2012. It refers to a token containing data or claims on which a receiving service will rely to make security decisions. This should be differentiated from a simple token containing only an identifier, which a receiving service uses to look up data locally. The most common examples of self-contained tokens are JSON Web Tokens (JWTs) and SAML assertions.

The use of self-contained tokens has become very widespread, even outside of OAuth and OIDC. At the same time, the security of this mechanism relies on the ability to validate the integrity of the token and to ensure that the token is valid for a particular context. There are many pitfalls with this process, and this chapter provides specific details of the mechanisms that applications should have in place to prevent them.

V9.1 Token source and integrity

This section includes requirements to ensure that the token has been produced by a trusted party and has not been tampered with.

#	Description	Level
9.1.1	Verify that self-contained tokens are validated using their digital signature or MAC to protect against tampering before accepting the token's contents.	1
9.1.2	Verify that only algorithms on an allowlist can be used to create and verify self-contained tokens, for a given context. The allowlist must include the permitted algorithms, ideally only either symmetric or asymmetric algorithms, and must not include the 'None'algorithm. If both symmetric and asymmetric must be supported, additional controls will be needed to prevent key confusion.	1
9.1.3	Verify that key material that is used to validate self-contained tokens is from trusted pre-configured sources for the token issuer, preventing attackers from specifying untrusted sources and keys. For JWTs and other JWS structures, headers such as 'jku', 'x5u', and 'jwk'must be validated against an allowlist of trusted sources.	1

V9.2 Token content

Before making security decisions based on the content of a self-contained token, it is necessary to validate that the token has been presented within its validity period and that it is intended for use by the receiving service and for the purpose for which it was presented. This helps avoid insecure cross-usage between different services or with different token types from the same issuer.

Specific requirements for OAuth and OIDC are covered in the dedicated chapter.

#	Description	Level
9.2.1	Verify that, if a validity time span is present in the token data, the token and its content are accepted only if the verification time is within this validity time span. For example, for JWTs, the claims 'nbf'and 'exp'must be verified.	1
9.2.2	Verify that the service receiving a token validates the token to be the correct type and is meant for the intended purpose before accepting the token's contents. For example, only access tokens can be accepted for authorization decisions and only ID Tokens can be used for proving user authentication.	2
9.2.3	Verify that the service only accepts tokens which are intended for use with that service (audience). For JWTs, this can be achieved by validating the 'aud' claim against an allowlist defined in the service.	2

#	Description	Level
9.2.4	Verify that, if a token issuer uses the same private key for issuing tokens to different audiences, the issued tokens contain an audience restriction that uniquely identifies the intended audiences. This will prevent a token from being reused with an unintended audience. If the audience identifier is dynamically provisioned, the token issuer must validate these audiences in order to make sure that they do not result in audience impersonation.	2

References

For more information, see also:

- OWASP JSON Web Token Cheat Sheet for Java Cheat Sheet (but has useful general guidance)

V10 OAuth and OIDC

Control Objective

OAuth2 (referred to as OAuth in this chapter) is an industry-standard framework for delegated authorization. For example, using OAuth, a client application can obtain access to APIs (server resources) on a user's behalf, provided the user has authorized the client application to do so.

By itself, OAuth is not designed for user authentication. The OpenID Connect (OIDC) framework extends OAuth by adding a user identity layer on top of OAuth. OIDC provides support for features including standardized user information, Single Sign-On (SSO), and session management. As OIDC is an extension of OAuth, the OAuth requirements in this chapter also apply to OIDC.

The following roles are defined in OAuth:

- The OAuth client is the application that attempts to obtain access to server resources (e.g., by calling an API using the issued access token). The OAuth client is often a server-side application.
  - A confidential client is a client capable of maintaining the confidentiality of the credentials it uses to authenticate itself with the authorization server.
  - A public client is not capable of maintaining the confidentiality of credentials for authenticating with the authorization server. Therefore, instead of authenticating itself (e.g., using 'client_id'and 'client_secret'parameters), it only identifies itself (using a 'client_id' parameter).
- The OAuth resource server (RS) is the server API exposing resources to OAuth clients.
- The OAuth authorization server (AS) is a server application that issues access tokens to OAuth clients. These access tokens allow OAuth clients to access RS resources, either on behalf of an

end-user or on the OAuth client's own behalf. The AS is often a separate application, but (if appropriate) it may be integrated into a suitable RS.
- The resource owner (RO) is the end-user who authorizes OAuth clients to obtain limited access to resources hosted on the resource server on their behalf. The resource owner consents to this delegated authorization by interacting with the authorization server.

The following roles are defined in OIDC:

- The relying party (RP) is the client application requesting end-user authentication through the OpenID Provider. It assumes the role of an OAuth client.
- The OpenID Provider (OP) is an OAuth AS that is capable of authenticating the end-user and provides OIDC claims to an RP. The OP may be the identity provider (IdP), but in federated scenarios, the OP and the identity provider (where the end-user authenticates) may be different server applications.

OAuth and OIDC were initially designed for third-party applications. Today, they are often used by first-party applications as well. However, when used in first-party scenarios, such as authentication and session management, the protocol adds some complexity, which may introduce new security challenges.

OAuth and OIDC can be used for many types of applications, but the focus for ASVS and the requirements in this chapter is on web applications and APIs.

Since OAuth and OIDC can be considered logic on top of web technologies, general requirements from other chapters always apply, and this chapter cannot be taken out of context.

This chapter addresses best current practices for OAuth2 and OIDC aligned with specifications found at https://oauth.net/2/ and https://openid.net/developers/specs/. Even if RFCs are considered mature, they are updated frequently. Thus, it is important to align with the latest versions when applying the requirements in this chapter. See the references section for more details.

Given the complexity of the area, it is vitally important for a secure OAuth or OIDC solution to use well-known industry-standard authorization servers and apply the recommended security configuration.

Terminology used in this chapter aligns with OAuth RFCs and OIDC specifications, but note that OIDC terminology is only used for OIDC-specific requirements; otherwise, OAuth terminology is used.

In the context of OAuth and OIDC, the term "token"in this chapter refers to:

- Access tokens, which shall only be consumed by the RS and can either be reference tokens that are validated using introspection or self-contained tokens that are validated using some key material.
- Refresh tokens, which shall only be consumed by the authorization server that issued the token.
- OIDC ID Tokens, which shall only be consumed by the client that triggered the authorization flow.

The risk levels for some of the requirements in this chapter depend on whether the client is a confidential client or regarded as a public client. Since using strong client authentication mitigates many attack vectors, a few requirements might be relaxed when using a confidential client for L1 applications.

V10.1 Generic OAuth and OIDC Security

This section covers generic architectural requirements that apply to all applications using OAuth or OIDC.

#	Description	Level
10.1.1	Verify that tokens are only sent to components that strictly need them. For example, when using a backend-for-frontend pattern for browser-based JavaScript applications, access and refresh tokens shall only be accessible for the backend.	2
10.1.2	Verify that the client only accepts values from the authorization server (such as the authorization code or ID Token) if these values result from an authorization flow that was initiated by the same user agent session and transaction. This requires that client-generated secrets, such as the proof key for code exchange (PKCE) 'code_verifier', 'state'or OIDC 'nonce', are not guessable, are specific to the transaction, and are securely bound to both the client and the user agent session in which the transaction was started.	2

V10.2 OAuth Client

These requirements detail the responsibilities for OAuth client applications. The client can be, for example, a web server backend (often acting as a Backend For Frontend, BFF), a backend service integration, or a frontend Single Page Application (SPA, aka browser-based application).

In general, backend clients are regarded as confidential clients and frontend clients are regarded as public clients. However, native applications running on the end-user device can be regarded as confidential when using OAuth dynamic client registration.

#	Description	Level
10.2.1	Verify that, if the code flow is used, the OAuth client has protection against browser-based request forgery attacks, commonly known as cross-site request forgery (CSRF), which trigger token requests, either by using proof key for code exchange (PKCE) functionality or checking the 'state'parameter that was sent in the authorization request.	2

#	Description	Level
10.2.2	Verify that, if the OAuth client can interact with more than one authorization server, it has a defense against mix-up attacks. For example, it could require that the authorization server return the 'iss'parameter value and validate it in the authorization response and the token response.	2
10.2.3	Verify that the OAuth client only requests the required scopes (or other authorization parameters) in requests to the authorization server.	3

V10.3 OAuth Resource Server

In the context of ASVS and this chapter, the resource server is an API. To provide secure access, the resource server must:

- Validate the access token, according to the token format and relevant protocol specifications, e.g., JWT-validation or OAuth token introspection.
- If valid, enforce authorization decisions based on the information from the access token and permissions which have been granted. For example, the resource server needs to verify that the client (acting on behalf of RO) is authorized to access the requested resource.

Therefore, the requirements listed here are OAuth or OIDC specific and should be performed after token validation and before performing authorization based on information from the token.

#	Description	Level
10.3.1	Verify that the resource server only accepts access tokens that are intended for use with that service (audience). The audience may be included in a structured access token (such as the 'aud'claim in JWT), or it can be checked using the token introspection endpoint.	2
10.3.2	Verify that the resource server enforces authorization decisions based on claims from the access token that define delegated authorization. If claims such as 'sub', 'scope', and 'authorization_details'are present, they must be part of the decision.	2
10.3.3	Verify that if an access control decision requires identifying a unique user from an access token (JWT or related token introspection response), the resource server identifies the user from claims that cannot be reassigned to other users. Typically, it means using a combination of 'iss'and 'sub'claims.	2

#	Description	Level
10.3.4	Verify that, if the resource server requires specific authentication strength, methods, or recentness, it verifies that the presented access token satisfies these constraints. For example, if present, using the OIDC ‘acr’, ‘amr’and ‘auth_time’claims respectively.	2
10.3.5	Verify that the resource server prevents the use of stolen access tokens or replay of access tokens (from unauthorized parties) by requiring sender-constrained access tokens, either Mutual TLS for OAuth 2 or OAuth 2 Demonstration of Proof of Possession (DPoP).	3

V10.4 OAuth Authorization Server

These requirements detail the responsibilities for OAuth authorization servers, including OpenID Providers.

For client authentication, the ‘self_signed_tls_client_auth’method is allowed with the prerequisites required by section 2.2 of RFC 8705.

#	Description	Level
10.4.1	Verify that the authorization server validates redirect URIs based on a client-specific allowlist of pre-registered URIs using exact string comparison.	1
10.4.2	Verify that, if the authorization server returns the authorization code in the authorization response, it can be used only once for a token request. For the second valid request with an authorization code that has already been used to issue an access token, the authorization server must reject a token request and revoke any issued tokens related to the authorization code.	1
10.4.3	Verify that the authorization code is short-lived. The maximum lifetime can be up to 10 minutes for L1 and L2 applications and up to 1 minute for L3 applications.	1
10.4.4	Verify that for a given client, the authorization server only allows the usage of grants that this client needs to use. Note that the grants ‘token’(Implicit flow) and ‘password’(Resource Owner Password Credentials flow) must no longer be used.	1

#	Description	Level
10.4.5	Verify that the authorization server mitigates refresh token replay attacks for public clients, preferably using sender-constrained refresh tokens, i.e., Demonstrating Proof of Possession (DPoP) or Certificate-Bound Access Tokens using mutual TLS (mTLS). For L1 and L2 applications, refresh token rotation may be used. If refresh token rotation is used, the authorization server must invalidate the refresh token after usage, and revoke all refresh tokens for that authorization if an already used and invalidated refresh token is provided.	1
10.4.6	Verify that, if the code grant is used, the authorization server mitigates authorization code interception attacks by requiring proof key for code exchange (PKCE). For authorization requests, the authorization server must require a valid 'code_challenge'value and must not accept a 'code_challenge_method'value of 'plain'. For a token request, it must require validation of the 'code_verifier'parameter.	2
10.4.7	Verify that if the authorization server supports unauthenticated dynamic client registration, it mitigates the risk of malicious client applications. It must validate client metadata such as any registered URIs, ensure the user's consent, and warn the user before processing an authorization request with an untrusted client application.	2
10.4.8	Verify that refresh tokens have an absolute expiration, including if sliding refresh token expiration is applied.	2
10.4.9	Verify that refresh tokens and reference access tokens can be revoked by an authorized user using the authorization server user interface, to mitigate the risk of malicious clients or stolen tokens.	2
10.4.10	Verify that confidential client is authenticated for client-to-authorized server backchannel requests such as token requests, pushed authorization requests (PAR), and token revocation requests.	2
10.4.11	Verify that the authorization server configuration only assigns the required scopes to the OAuth client.	2
10.4.12	Verify that for a given client, the authorization server only allows the 'response_mode'value that this client needs to use. For example, by having the authorization server validate this value against the expected values or by using pushed authorization request (PAR) or JWT-secured Authorization Request (JAR).	3
10.4.13	Verify that grant type 'code'is always used together with pushed authorization requests (PAR).	3

#	Description	Level
10.4.14	Verify that the authorization server issues only sender-constrained (Proof-of-Possession) access tokens, either with certificate-bound access tokens using mutual TLS (mTLS) or DPoP-bound access tokens (Demonstration of Proof of Possession).	3
10.4.15	Verify that, for a server-side client (which is not executed on the end-user device), the authorization server ensures that the 'authorization_details' parameter value is from the client backend and that the user has not tampered with it. For example, by requiring the usage of pushed authorization request (PAR) or JWT-secured Authorization Request (JAR).	3
10.4.16	Verify that the client is confidential and the authorization server requires the use of strong client authentication methods (based on public-key cryptography and resistant to replay attacks), such as mutual TLS ('tls_client_auth', 'self_signed_tls_client_auth') or private key JWT ('private_key_jwt').	3

V10.5 OIDC Client

As the OIDC relying party acts as an OAuth client, the requirements from the section "OAuth Client" apply as well.

Note that the "Authentication with an Identity Provider"section in the "Authentication"chapter also contains relevant general requirements.

#	Description	Level
10.5.1	Verify that the client (as the relying party) mitigates ID Token replay attacks. For example, by ensuring that the 'nonce'claim in the ID Token matches the 'nonce'value sent in the authentication request to the OpenID Provider (in OAuth2 refereed to as the authorization request sent to the authorization server).	2
10.5.2	Verify that the client uniquely identifies the user from ID Token claims, usually the 'sub'claim, which cannot be reassigned to other users (for the scope of an identity provider).	2
10.5.3	Verify that the client rejects attempts by a malicious authorization server to impersonate another authorization server through authorization server metadata. The client must reject authorization server metadata if the issuer URL in the authorization server metadata does not exactly match the pre-configured issuer URL expected by the client.	2

#	Description	Level
10.5.4	Verify that the client validates that the ID Token is intended to be used for that client (audience) by checking that the ‘aud’claim from the token is equal to the ‘client_id’value for the client.	2
10.5.5	Verify that, when using OIDC back-channel logout, the relying party mitigates denial of service through forced logout and cross-JWT confusion in the logout flow. The client must verify that the logout token is correctly typed with a value of ‘logout+jwt’, contains the ‘event’claim with the correct member name, and does not contain a ‘nonce’claim. Note that it is also recommended to have a short expiration (e.g., 2 minutes).	2

V10.6 OpenID Provider

As OpenID Providers act as OAuth authorization servers, the requirements from the section “OAuth Authorization Server”apply as well.

Note that if using the ID Token flow (not the code flow), no access tokens are issued, and many of the requirements for OAuth AS are not applicable.

#	Description	Level
10.6.1	Verify that the OpenID Provider only allows values ‘code’, ‘ciba’, ‘id_token’, or ‘id_token code’for response mode. Note that ‘code’is preferred over ‘id_token code’(the OIDC Hybrid flow), and ‘token’(any Implicit flow) must not be used.	2
10.6.2	Verify that the OpenID Provider mitigates denial of service through forced logout. By obtaining explicit confirmation from the end-user or, if present, validating parameters in the logout request (initiated by the relying party), such as the ‘id_token_hint’.	2

V10.7 Consent Management

These requirements cover the verification of the user’s consent by the authorization server. Without proper user consent verification, a malicious actor may obtain permissions on the user’s behalf through spoofing or social-engineering.

#	Description	Level
10.7.1	Verify that the authorization server ensures that the user consents to each authorization request. If the identity of the client cannot be assured, the authorization server must always explicitly prompt the user for consent.	2
10.7.2	Verify that when the authorization server prompts for user consent, it presents sufficient and clear information about what is being consented to. When applicable, this should include the nature of the requested authorizations (typically based on scope, resource server, Rich Authorization Requests (RAR) authorization details), the identity of the authorized application, and the lifetime of these authorizations.	2
10.7.3	Verify that the user can review, modify, and revoke consents which the user has granted through the authorization server.	2

References

For more information on OAuth, please see:

- oauth.net
- OWASP OAuth 2.0 Protocol Cheat Sheet

For OAuth-related requirements in ASVS following published and in draft status RFC-s are used:

- RFC6749 The OAuth 2.0 Authorization Framework
- RFC6750 The OAuth 2.0 Authorization Framework: Bearer Token Usage
- RFC6819 OAuth 2.0 Threat Model and Security Considerations
- RFC7636 Proof Key for Code Exchange by OAuth Public Clients
- RFC7591 OAuth 2.0 Dynamic Client Registration Protocol
- RFC8628 OAuth 2.0 Device Authorization Grant
- RFC8707 Resource Indicators for OAuth 2.0
- RFC9068 JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens
- RFC9126 OAuth 2.0 Pushed Authorization Requests
- RFC9207 OAuth 2.0 Authorization Server Issuer Identification
- RFC9396 OAuth 2.0 Rich Authorization Requests
- RFC9449 OAuth 2.0 Demonstrating Proof of Possession (DPoP)
- RFC9700 Best Current Practice for OAuth 2.0 Security
- draft OAuth 2.0 for Browser-Based Applications
- draft The OAuth 2.1 Authorization Framework

For more information on OpenID Connect, please see:

- OpenID Connect Core 1.0

- FAPI 2.0 Security Profile

V11 Cryptography

Control Objective

The objective of this chapter is to define best practices for the general use of cryptography, as well as to instill a fundamental understanding of cryptographic principles and inspire a shift toward more resilient and modern approaches. It encourages the following:

- Implementing robust cryptographic systems that fail securely, adapt to evolving threats, and are future-proof.
- Utilizing cryptographic mechanisms that are both secure and aligned with industry best practices.
- Maintaining a secure cryptographic key management system with appropriate access controls and auditing.
- Regularly evaluating the cryptographic landscape to assess new risks and adapt algorithms accordingly.
- Discovering and managing cryptographic use cases throughout the application's lifecycle to ensure that all cryptographic assets are accounted for and secured.

In addition to outlining general principles and best practices, this document also provides more in-depth technical information about the requirements in Appendix C - Cryptography Standards. This includes algorithms and modes that are considered "approved"for the purposes of the requirements in this chapter.

Requirements that use cryptography to solve a separate problem, such as secrets management or communications security, will be in different parts of the standard.

V11.1 Cryptographic Inventory and Documentation

Applications need to be designed with strong cryptographic architecture to protect data assets according to 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 "on the fly"or retrofitting it will inevitably cost much more to implement securely than simply building it in from the start.

It is important to ensure that all cryptographic assets are regularly discovered, inventoried, and assessed. Please see the appendix for more information on how this can be done.

The need to future-proof cryptographic systems against the eventual rise of quantum computing is also critical. Post-Quantum Cryptography (PQC) refers to cryptographic algorithms designed to

remain secure against attacks by quantum computers, which are expected to break widely used algorithms such as RSA and elliptic curve cryptography (ECC).

Please see the appendix for current guidance on vetted PQC primitives and standards.

#	Description	Level
11.1.1	Verify that there is a documented policy for management of cryptographic keys and a cryptographic key lifecycle that follows a key management standard such as NIST SP 800-57. This should include ensuring that keys are not overshared (for example, with more than two entities for shared secrets and more than one entity for private keys).	2
11.1.2	Verify that a cryptographic inventory is performed, maintained, regularly updated, and includes all cryptographic keys, algorithms, and certificates used by the application. It must also document where keys can and cannot be used in the system, and the types of data that can and cannot be protected using the keys.	2
11.1.3	Verify that cryptographic discovery mechanisms are employed to identify all instances of cryptography in the system, including encryption, hashing, and signing operations.	3
11.1.4	Verify that a cryptographic inventory is maintained. This must include a documented plan that outlines the migration path to new cryptographic standards, such as post-quantum cryptography, in order to react to future threats.	3

V11.2 Secure Cryptography Implementation

This section defines the requirements for the selection, implementation, and ongoing management of core cryptographic algorithms for an application. The objective is to ensure that only robust, industry-accepted cryptographic primitives are deployed, in alignment with current standards (e.g., NIST, ISO/IEC) and best practices. Organizations must ensure that each cryptographic component is selected based on peer-reviewed evidence and practical security testing.

#	Description	Level
11.2.1	Verify that industry-validated implementations (including libraries and hardware-accelerated implementations) are used for cryptographic operations.	2

#	Description	Level
11.2.2	Verify that the application is designed with crypto agility such that random number, authenticated encryption, MAC, or hashing algorithms, key lengths, rounds, ciphers and modes can be reconfigured, upgraded, or swapped at any time, to protect against cryptographic breaks. Similarly, it must also be possible to replace keys and passwords and re-encrypt data. This will allow for seamless upgrades to post-quantum cryptography (PQC), once high-assurance implementations of approved PQC schemes or standards are widely available.	2
11.2.3	Verify that all cryptographic primitives utilize a minimum of 128-bits of security based on the algorithm, key size, and configuration. For example, a 256-bit ECC key provides roughly 128 bits of security where RSA requires a 3072-bit key to achieve 128 bits of security.	2
11.2.4	Verify that all cryptographic operations are constant-time, with no ‘short-circuit’operations in comparisons, calculations, or returns, to avoid leaking information.	3
11.2.5	Verify that all cryptographic modules fail securely, and errors are handled in a way that does not enable vulnerabilities, such as Padding Oracle attacks.	3

V11.3 Encryption Algorithms

Authenticated encryption algorithms built on AES and CHACHA20 form the backbone of modern cryptographic practice.

#	Description	Level
11.3.1	Verify that insecure block modes (e.g., ECB) and weak padding schemes (e.g., PKCS#1 v1.5) are not used.	1
11.3.2	Verify that only approved ciphers and modes such as AES with GCM are used.	1
11.3.3	Verify that encrypted data is protected against unauthorized modification preferably by using an approved authenticated encryption method or by combining an approved encryption method with an approved MAC algorithm.	2
11.3.4	Verify that nonces, initialization vectors, and other single-use numbers are not used for more than one encryption key and data-element pair. The method of generation must be appropriate for the algorithm being used.	3

#	Description	Level
11.3.5	Verify that any combination of an encryption algorithm and a MAC algorithm is operating in encrypt-then-MAC mode.	3

V11.4 Hashing and Hash-based Functions

Cryptographic hashes are used in a wide variety of cryptographic protocols, such as digital signatures, HMAC, key derivation functions (KDF), random bit generation, and password storage. The security of the cryptographic system is only as strong as the underlying hash functions used. This section outlines the requirements for using secure hash functions in cryptographic operations.

For password storage, as well as the cryptography appendix, the OWASP Password Storage Cheat Sheet will also provide useful context and guidance.

#	Description	Level
11.4.1	Verify that only approved hash functions are used for general cryptographic use cases, including digital signatures, HMAC, KDF, and random bit generation. Disallowed hash functions, such as MD5, must not be used for any cryptographic purpose.	1
11.4.2	Verify that passwords are stored using an approved, computationally intensive, key derivation function (also known as a “password hashing function”), with parameter settings configured based on current guidance. The settings should balance security and performance to make brute-force attacks sufficiently challenging for the required level of security.	2
11.4.3	Verify that hash functions used in digital signatures, as part of data authentication or data integrity are collision resistant and have appropriate bit-lengths. If collision resistance is required, the output length must be at least 256 bits. If only resistance to second pre-image attacks is required, the output length must be at least 128 bits.	2
11.4.4	Verify that the application uses approved key derivation functions with key stretching parameters when deriving secret keys from passwords. The parameters in use must balance security and performance to prevent brute-force attacks from compromising the resulting cryptographic key.	2

V11.5 Random Values

Cryptographically secure Pseudo-random Number Generation (CSPRNG) 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	Level
11.5.1	Verify that all random numbers and strings which are intended to be non-guessable must be generated using a cryptographically secure pseudo-random number generator (CSPRNG) and have at least 128 bits of entropy. Note that UUIDs do not respect this condition.	2
11.5.2	Verify that the random number generation mechanism in use is designed to work securely, even under heavy demand.	3

V11.6 Public Key Cryptography

Public Key Cryptography will be used where it is not possible or not desirable to share a secret key between multiple parties.

As part of this, there exists a need for approved key exchange mechanisms, such as Diffie-Hellman and Elliptic Curve Diffie-Hellman (ECDH) to ensure that the cryptosystem remains secure against modern threats. The “Secure Communication”chapter provides requirements for TLS so the requirements in this section are intended for situations where Public Key Cryptography is being used in use cases other than TLS.

#	Description	Level
11.6.1	Verify that only approved cryptographic algorithms and modes of operation are used for key generation and seeding, and digital signature generation and verification. Key generation algorithms must not generate insecure keys vulnerable to known attacks, for example, RSA keys which are vulnerable to Fermat factorization.	2
11.6.2	Verify that approved cryptographic algorithms are used for key exchange (such as Diffie-Hellman) with a focus on ensuring that key exchange mechanisms use secure parameters. This will prevent attacks on the key establishment process which could lead to adversary-in-the-middle attacks or cryptographic breaks.	3

V11.7 In-Use Data Cryptography

Protecting data while it is being processed is paramount. Techniques such as full memory encryption, encryption of data in transit, and ensuring data is encrypted as quickly as possible after use is recommended.

#	Description	Level
11.7.1	Verify that full memory encryption is in use that protects sensitive data while it is in use, preventing access by unauthorized users or processes.	3
11.7.2	Verify that data minimization ensures the minimal amount of data is exposed during processing, and ensure that data is encrypted immediately after use or as soon as feasible.	3

References

For more information, see also:

- OWASP Web Security Testing Guide: Testing for Weak Cryptography
- OWASP Cryptographic Storage Cheat Sheet
- FIPS 140-3
- NIST SP 800-57

V12 Secure Communication

Control Objective

This chapter includes requirements related to the specific mechanisms that should be in place to protect data in transit, both between an end-user client and a backend service, as well as between internal and backend services.

The general concepts promoted by this chapter include:

- Ensuring that communications are encrypted externally, and ideally internally as well.
- Configuring encryption mechanisms using the latest guidance, including preferred algorithms and ciphers.
- Ensuring that communications are not being intercepted by unauthorized parties through the use of signed certificates.

In addition to outlining general principles and best practices, the ASVS also provides more in-depth technical information about cryptographic strength in Appendix C - Cryptography Standards.

V12.1 General TLS Security Guidance

This section provides initial guidance on how to secure TLS communications. Up-to-date tools should be used to review TLS configuration on an ongoing basis.

While the use of wildcard TLS certificates is not inherently insecure, a compromise of a certificate that is deployed across all owned environments (e.g., production, staging, development, and test) may lead to a compromise of the security posture of the applications using it. Proper protection, management, and the use of separate TLS certificates in different environments should be employed if possible.

#	Description	Level
12.1.1	Verify that only the latest recommended versions of the TLS protocol are enabled, such as TLS 1.2 and TLS 1.3. The latest version of the TLS protocol must be the preferred option.	1
12.1.2	Verify that only recommended cipher suites are enabled, with the strongest cipher suites set as preferred. L3 applications must only support cipher suites which provide forward secrecy.	2
12.1.3	Verify that the application validates that mTLS client certificates are trusted before using the certificate identity for authentication or authorization.	2
12.1.4	Verify that proper certification revocation, such as Online Certificate Status Protocol (OCSP) Stapling, is enabled and configured.	3
12.1.5	Verify that Encrypted Client Hello (ECH) is enabled in the application's TLS settings to prevent exposure of sensitive metadata, such as the Server Name Indication (SNI), during TLS handshake processes.	3

V12.2 HTTPS Communication with External Facing Services

Ensure all HTTP traffic to external-facing services which the application exposes is sent encrypted, with publicly trusted certificates.

#	Description	Level
12.2.1	Verify that TLS is used for all connectivity between a client and external facing, HTTP-based services, and does not fall back to insecure or unencrypted communications.	1
12.2.2	Verify that external facing services use publicly trusted TLS certificates.	1

V12.3 General Service to Service Communication Security

Server communications (both internal and external) involve more than just HTTP. Connections to and from other systems must also be secure, ideally using TLS.

#	Description	Level
12.3.1	Verify that an encrypted protocol such as TLS is used for all inbound and outbound connections to and from the application, including monitoring systems, management tools, remote access and SSH, middleware, databases, mainframes, partner systems, or external APIs. The server must not fall back to insecure or unencrypted protocols.	2
12.3.2	Verify that TLS clients validate certificates received before communicating with a TLS server.	2
12.3.3	Verify that TLS or another appropriate transport encryption mechanism used for all connectivity between internal, HTTP-based services within the application, and does not fall back to insecure or unencrypted communications.	2
12.3.4	Verify that TLS connections between internal services use trusted certificates. Where internally generated or self-signed certificates are used, the consuming service must be configured to only trust specific internal CAs and specific self-signed certificates.	2
12.3.5	Verify that services communicating internally within a system (intra-service communications) use strong authentication to ensure that each endpoint is verified. Strong authentication methods, such as TLS client authentication, must be employed to ensure identity, using public-key infrastructure and mechanisms that are resistant to replay attacks. For microservice architectures, consider using a service mesh to simplify certificate management and enhance security.	3

References

For more information, see also:

- OWASP - Transport Layer Security Cheat Sheet
- Mozilla's Server Side TLS configuration guide
- Mozilla's tool to generate known good TLS configurations.
- O-Saft - OWASP Project to validate TLS configuration

V13 Configuration

Control Objective

The application's default configuration must be secure for use on the Internet.

This chapter provides guidance on the various configurations necessary to achieve this, including those applied during development, build, and deployment.

Topics covered include preventing data leakage, securely managing communication between components, and protecting secrets.

V13.1 Configuration Documentation

This section outlines documentation requirements for how the application communicates with internal and external services, as well as techniques to prevent loss of availability due to service inaccessibility. It also addresses documentation related to secrets.

#	Description	Level
13.1.1	Verify that all communication needs for the application are documented. This must include external services which the application relies upon and cases where an end user might be able to provide an external location to which the application will then connect.	2
13.1.2	Verify that for each service the application uses, the documentation defines the maximum number of concurrent connections (e.g., connection pool limits) and how the application behaves when that limit is reached, including any fallback or recovery mechanisms, to prevent denial of service conditions.	3
13.1.3	Verify that the application documentation defines resource-management strategies for every external system or service it uses (e.g., databases, file handles, threads, HTTP connections). This should include resource-release procedures, timeout settings, failure handling, and where retry logic is implemented, specifying retry limits, delays, and back-off algorithms. For synchronous HTTP request–response operations it should mandate short timeouts and either disable retries or strictly limit retries to prevent cascading delays and resource exhaustion.	3
13.1.4	Verify that the application's documentation defines the secrets that are critical for the security of the application and a schedule for rotating them, based on the organization's threat model and business requirements.	3

V13.2 Backend Communication Configuration

Applications interact with multiple services, including APIs, databases, or other components. These may be considered internal to the application but not included in the application's standard access control mechanisms, or they may be entirely external. In either case, it is necessary to configure the application to interact securely with these components and, if required, protect that configuration.

Note: The “Secure Communication”chapter provides guidance for encryption in transit.

#	Description	Level
13.2.1	Verify that communications between backend application components that don’t support the application’s standard user session mechanism, including APIs, middleware, and data layers, are authenticated. Authentication must use individual service accounts, short-term tokens, or certificate-based authentication and not unchanging credentials such as passwords, API keys, or shared accounts with privileged access.	2
13.2.2	Verify that communications between backend application components, including local or operating system services, APIs, middleware, and data layers, are performed with accounts assigned the least necessary privileges.	2
13.2.3	Verify that if a credential has to be used for service authentication, the credential being used by the consumer is not a default credential (e.g., root/root or admin/admin).	2
13.2.4	Verify that an allowlist is used to define the external resources or systems with which the application is permitted to communicate (e.g., for outbound requests, data loads, or file access). This allowlist can be implemented at the application layer, web server, firewall, or a combination of different layers.	2
13.2.5	Verify that the web or application server is configured with an allowlist of resources or systems to which the server can send requests or load data or files from.	2
13.2.6	Verify that where the application connects to separate services, it follows the documented configuration for each connection, such as maximum parallel connections, behavior when maximum allowed connections is reached, connection timeouts, and retry strategies.	3

V13.3 Secret Management

Secret management is an essential configuration task to ensure the protection of data used in the application. Specific requirements for cryptography can be found in the “Cryptography”chapter, but this section focuses on the management and handling aspects of secrets.

#	Description	Level
13.3.1	Verify that a secrets management solution, such as a key vault, is used to securely create, store, control access to, and destroy backend secrets. These could include passwords, key material, integrations with databases and third-party systems, keys and seeds for time-based tokens, other internal secrets, and API keys. Secrets must not be included in application source code or included in build artifacts. For an L3 application, this must involve a hardware-backed solution such as an HSM.	2
13.3.2	Verify that access to secret assets adheres to the principle of least privilege.	2
13.3.3	Verify that all cryptographic operations are performed using an isolated security module (such as a vault or hardware security module) to securely manage and protect key material from exposure outside of the security module.	3
13.3.4	Verify that secrets are configured to expire and be rotated based on the application's documentation.	3

V13.4 Unintended Information Leakage

Production configurations should be hardened to avoid disclosing unnecessary data. Many of these issues are rarely rated as significant risks but are often chained with other vulnerabilities. If these issues are not present by default, it raises the bar for attacking an application.

For example, hiding the version of server-side components does not eliminate the need to patch all components, and disabling folder listing does not remove the need to use authorization controls or keep files away from the public folder, but it raises the bar.

#	Description	Level
13.4.1	Verify that the application is deployed either without any source control metadata, including the .git or .svn folders, or in a way that these folders are inaccessible both externally and to the application itself.	1
13.4.2	Verify that debug modes are disabled for all components in production environments to prevent exposure of debugging features and information leakage.	2
13.4.3	Verify that web servers do not expose directory listings to clients unless explicitly intended.	2
13.4.4	Verify that using the HTTP TRACE method is not supported in production environments, to avoid potential information leakage.	2

#	Description	Level
13.4.5	Verify that documentation (such as for internal APIs) and monitoring endpoints are not exposed unless explicitly intended.	2
13.4.6	Verify that the application does not expose detailed version information of backend components.	3
13.4.7	Verify that the web tier is configured to only serve files with specific file extensions to prevent unintentional information, configuration, and source code leakage.	3

References

For more information, see also:

- OWASP Web Security Testing Guide: Configuration and Deployment Management Testing

V14 Data Protection

Control Objective

Applications cannot account for all usage patterns and user behaviors, and should therefore implement controls to limit unauthorized access to sensitive data on client devices.

This chapter includes requirements related to defining what data needs to be protected, how it should be protected, and specific mechanisms to implement or pitfalls to avoid.

Another consideration for data protection is bulk extraction, modification, or excessive usage. Each system's requirements are likely to be very different, so determining what is "abnormal"must consider the threat model and business risk. From an ASVS perspective, detecting these issues is handled in the "Security Logging and Error Handling"chapter, and setting limits is handled in the "Validation and Business Logic"chapter.

V14.1 Data Protection Documentation

A key prerequisite for being able to protect data is to categorize what data should be considered sensitive. There are likely to be several different levels of sensitivity, and for each level, the controls required to protect data at that level will be different.

There are various privacy regulations and laws that affect how applications must approach the storage, use, and transmission of sensitive personal information. This section no longer tries to duplicate

these types of data protection or privacy legislation, but rather focuses on key technical considerations for protecting sensitive data. Please consult local laws and regulations, and consult a qualified privacy specialist or lawyer as required.

#	Description	Level
14.1.1	Verify that all sensitive data created and processed by the application has been identified and classified into protection levels. This includes data that is only encoded and therefore easily decoded, such as Base64 strings or the plaintext payload inside a JWT. Protection levels need to take into account any data protection and privacy regulations and standards which the application is required to comply with.	2
14.1.2	Verify that all sensitive data protection levels have a documented set of protection requirements. This must include (but not be limited to) requirements related to general encryption, integrity verification, retention, how the data is to be logged, access controls around sensitive data in logs, database-level encryption, privacy and privacy-enhancing technologies to be used, and other confidentiality requirements.	2

V14.2 General Data Protection

This section contains various practical requirements related to the protection of data. Most are specific to particular issues such as unintended data leakage, but there is also a general requirement to implement protection controls based on the protection level required for each data item.

#	Description	Level
14.2.1	Verify that sensitive data is only sent to the server in the HTTP message body or header fields, and that the URL and query string do not contain sensitive information, such as an API key or session token.	1
14.2.2	Verify that the application prevents sensitive data from being cached in server components, such as load balancers and application caches, or ensures that the data is securely purged after use.	2
14.2.3	Verify that defined sensitive data is not sent to untrusted parties (e.g., user trackers) to prevent unwanted collection of data outside of the application's control.	2

#	Description	Level
14.2.4	Verify that controls around sensitive data related to encryption, integrity verification, retention, how the data is to be logged, access controls around sensitive data in logs, privacy and privacy-enhancing technologies, are implemented as defined in the documentation for the specific data's protection level.	2
14.2.5	Verify that caching mechanisms are configured to only cache responses which have the expected content type for that resource and do not contain sensitive, dynamic content. The web server should return a 404 or 302 response when a non-existent file is accessed rather than returning a different, valid file. This should prevent Web Cache Deception attacks.	3
14.2.6	Verify that the application only returns the minimum required sensitive data for the application's functionality. For example, only returning some of the digits of a credit card number and not the full number. If the complete data is required, it should be masked in the user interface unless the user specifically views it.	3
14.2.7	Verify that sensitive information is subject to data retention classification, ensuring that outdated or unnecessary data is deleted automatically, on a defined schedule, or as the situation requires.	3
14.2.8	Verify that sensitive information is removed from the metadata of user-submitted files unless storage is consented to by the user.	3

V14.3 Client-side Data Protection

This section contains requirements preventing data from leaking in specific ways at the client or user agent side of an application.

#	Description	Level
14.3.1	Verify that authenticated data is cleared from client storage, such as the browser DOM, after the client or session is terminated. The 'Clear-Site-Data' HTTP response header field may be able to help with this but the client-side should also be able to clear up if the server connection is not available when the session is terminated.	1
14.3.2	Verify that the application sets sufficient anti-caching HTTP response header fields (i.e., Cache-Control: no-store) so that sensitive data is not cached in browsers.	2

#	Description	Level
14.3.3	Verify that data stored in browser storage (such as localStorage, sessionStorage, IndexedDB, or cookies) does not contain sensitive data, with the exception of session tokens.	2

References

For more information, see also:

- Consider using the Security Headers website to check security and anti-caching header fields
- Documentation about anti-caching headers by Mozilla
- OWASP Secure Headers project
- OWASP Privacy Risks Project
- OWASP User Privacy Protection Cheat Sheet
- Australian Privacy Principle 11 - Security of personal information
- European Union General Data Protection Regulation (GDPR) overview
- European Union Data Protection Supervisor - Internet Privacy Engineering Network
- Information on the “Clear-Site-Data”header
- White paper on Web Cache Deception

V15 Secure Coding and Architecture

Control Objective

Many ASVS requirements either relate to a particular area of security, such as authentication or authorization, or pertain to a particular type of application functionality, such as logging or file handling.

This chapter provides general security requirements to consider when designing and developing applications. These requirements focus not only on clean architecture and code quality but also on specific architecture and coding practices necessary for application security.

V15.1 Secure Coding and Architecture Documentation

Many requirements for establishing a secure and defensible architecture depend on clear documentation of decisions made regarding the implementation of specific security controls and the components used within the application.

This section outlines the documentation requirements, including identifying components considered to contain “dangerous functionality”or to be “risky components.”

A component with “dangerous functionality”may be an internally developed or third-party component that performs operations such as deserialization of untrusted data, raw file or binary data parsing, dynamic code execution, or direct memory manipulation. Vulnerabilities in these types of operations pose a high risk of compromising the application and potentially exposing its underlying infrastructure.

A “risky component”is a 3rd party library (i.e., not internally developed) with missing or poorly implemented security controls around its development processes or functionality. Examples include components that are poorly maintained, unsupported, at the end-of-life stage, or have a history of significant vulnerabilities.

This section also emphasizes the importance of defining appropriate timeframes for addressing vulnerabilities in third-party components.

#	Description	Level
15.1.1	Verify that application documentation defines risk based remediation time frames for 3rd party component versions with vulnerabilities and for updating libraries in general, to minimize the risk from these components.	1
15.1.2	Verify that an inventory catalog, such as software bill of materials (SBOM), is maintained of all third-party libraries in use, including verifying that components come from pre-defined, trusted, and continually maintained repositories.	2
15.1.3	Verify that the application documentation identifies functionality which is time-consuming or resource-demanding. This must include how to prevent a loss of availability due to overusing this functionality and how to avoid a situation where building a response takes longer than the consumer’s timeout. Potential defenses may include asynchronous processing, using queues, and limiting parallel processes per user and per application.	2
15.1.4	Verify that application documentation highlights third-party libraries which are considered to be “risky components”.	3
15.1.5	Verify that application documentation highlights parts of the application where “dangerous functionality”is being used.	3

V15.2 Security Architecture and Dependencies

This section includes requirements for handling risky, outdated, or insecure dependencies and components through dependency management.

It also includes using architectural-level techniques such as sandboxing, encapsulation, containerization, and network isolation to reduce the impact of using “dangerous operations”or “risky compo-

nents"(as defined in the previous section) and prevent loss of availability due to overusing resource-demanding functionality.

#	Description	Level
15.2.1	Verify that the application only contains components which have not breached the documented update and remediation time frames.	1
15.2.2	Verify that the application has implemented defenses against loss of availability due to functionality which is time-consuming or resource-demanding, based on the documented security decisions and strategies for this.	2
15.2.3	Verify that the production environment only includes functionality that is required for the application to function, and does not expose extraneous functionality such as test code, sample snippets, and development functionality.	2
15.2.4	Verify that third-party components and all of their transitive dependencies are included from the expected repository, whether internally owned or an external source, and that there is no risk of a dependency confusion attack.	3
15.2.5	Verify that the application implements additional protections around parts of the application which are documented as containing "dangerous functionality"or using third-party libraries considered to be "risky components". This could include techniques such as sandboxing, encapsulation, containerization or network level isolation to delay and deter attackers who compromise one part of an application from pivoting elsewhere in the application.	3

V15.3 Defensive Coding

This section covers vulnerability types, including type juggling, prototype pollution, and others, which result from using insecure coding patterns in a particular language. Some may not be relevant to all languages, whereas others will have language-specific fixes or may relate to how a particular language or framework handles a feature such as HTTP parameters. It also considers the risk of not cryptographically validating application updates.

It also considers the risks associated with using objects to represent data items and accepting and returning these via external APIs. In this case, the application must ensure that data fields that should not be writable are not modified by user input (mass assignment) and that the API is selective about what data fields get returned. Where field access depends on a user's permissions, this should be considered in the context of the field-level access control requirement in the Authorization chapter.

#	Description	Level
15.3.1	Verify that the application only returns the required subset of fields from a data object. For example, it should not return an entire data object, as some individual fields should not be accessible to users.	1
15.3.2	Verify that where the application backend makes calls to external URLs, it is configured to not follow redirects unless it is intended functionality.	2
15.3.3	Verify that the application has countermeasures to protect against mass assignment attacks by limiting allowed fields per controller and action, e.g., it is not possible to insert or update a field value when it was not intended to be part of that action.	2
15.3.4	Verify that all proxying and middleware components transfer the user's original IP address correctly using trusted data fields that cannot be manipulated by the end user, and the application and web server use this correct value for logging and security decisions such as rate limiting, taking into account that even the original IP address may not be reliable due to dynamic IPs, VPNs, or corporate firewalls.	2
15.3.5	Verify that the application explicitly ensures that variables are of the correct type and performs strict equality and comparator operations. This is to avoid type juggling or type confusion vulnerabilities caused by the application code making an assumption about a variable type.	2
15.3.6	Verify that JavaScript code is written in a way that prevents prototype pollution, for example, by using Set() or Map() instead of object literals.	2
15.3.7	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 (query string, body parameters, cookies, or header fields).	2

V15.4 Safe Concurrency

Concurrency issues such as race conditions, time-of-check to time-of-use (TOCTOU) vulnerabilities, deadlocks, livelocks, thread starvation, and improper synchronization can lead to unpredictable behavior and security risks. This section includes various techniques and strategies to help mitigate these risks.

#	Description	Level
15.4.1	Verify that shared objects in multi-threaded code (such as caches, files, or in-memory objects accessed by multiple threads) are accessed safely by using thread-safe types and synchronization mechanisms like locks or semaphores to avoid race conditions and data corruption.	3
15.4.2	Verify that checks on a resource's state, such as its existence or permissions, and the actions that depend on them are performed as a single atomic operation to prevent time-of-check to time-of-use (TOCTOU) race conditions. For example, checking if a file exists before opening it, or verifying a user's access before granting it.	3
15.4.3	Verify that locks are used consistently to avoid threads getting stuck, whether by waiting on each other or retrying endlessly, and that locking logic stays within the code responsible for managing the resource to ensure locks cannot be inadvertently or maliciously modified by external classes or code.	3
15.4.4	Verify that resource allocation policies prevent thread starvation by ensuring fair access to resources, such as by leveraging thread pools, allowing lower-priority threads to proceed within a reasonable timeframe.	3

References

For more information, see also:

- OWASP Prototype Pollution Prevention Cheat Sheet
- OWASP Mass Assignment Prevention Cheat Sheet
- OWASP CycloneDX Bill of Materials Specification
- OWASP Web Security Testing Guide: Testing for HTTP Parameter Pollution

V16 Security Logging and Error Handling

Control Objective

Security logs are distinct from error or performance logs and are used to record security-relevant events such as authentication decisions, access control decisions, and attempts to bypass security controls, such as input validation or business logic validation. Their purpose is to support detection, response, and investigation by providing high-signal, structured data for analysis tools like SIEMs.

Logs should not include sensitive personal data unless legally required, and any logged data must be protected as a high-value asset. Logging must not compromise privacy or system security. Applications must also fail securely, avoiding unnecessary disclosure or disruption.

For detailed implementation guidance, refer to the OWASP Cheat Sheets in the references section.

V16.1 Security Logging Documentation

This section ensures a clear and complete inventory of logging across the application stack. This is essential for effective security monitoring, incident response, and compliance.

#	Description	Level
16.1.1	Verify that an inventory exists documenting the logging performed at each layer of the application's technology stack, what events are being logged, log formats, where that logging is stored, how it is used, how access to it is controlled, and for how long logs are kept.	2

V16.2 General Logging

This section provides requirements to ensure that security logs are consistently structured and contain the expected metadata. The goal is to make logs machine-readable and analyzable across distributed systems and tools.

Naturally, security events often involve sensitive data. If such data is logged without consideration, the logs themselves become classified and therefore subject to encryption requirements, stricter retention policies, and potential disclosure during audits.

Therefore, it is critical to log only what is necessary and to treat log data with the same care as other sensitive assets.

The requirements below establish foundational requirements for logging metadata, synchronization, format, and control.

#	Description	Level
16.2.1	Verify that each log entry includes necessary metadata (such as when, where, who, what) that would allow for a detailed investigation of the timeline when an event happens.	2
16.2.2	Verify that time sources for all logging components are synchronized, and that timestamps in security event metadata use UTC or include an explicit time zone offset. UTC is recommended to ensure consistency across distributed systems and to prevent confusion during daylight saving time transitions.	2
16.2.3	Verify that the application only stores or broadcasts logs to the files and services that are documented in the log inventory.	2

#	Description	Level
16.2.4	Verify that logs can be read and correlated by the log processor that is in use, preferably by using a common logging format.	2
16.2.5	Verify that when logging sensitive data, the application enforces logging based on the data's protection level. For example, it may not be allowed to log certain data, such as credentials or payment details. Other data, such as session tokens, may only be logged by being hashed or masked, either in full or partially.	2

V16.3 Security Events

This section defines requirements for logging security-relevant events within the application. Capturing these events is critical for detecting suspicious behavior, supporting investigations, and fulfilling compliance obligations.

This section outlines the types of events that should be logged but does not attempt to provide exhaustive detail. Each application has unique risk factors and operational context.

Note that while ASVS includes logging of security events in scope, alerting and correlation (e.g., SIEM rules or monitoring infrastructure) are considered out of scope and are handled by operational and monitoring systems.

#	Description	Level
16.3.1	Verify that all authentication operations are logged, including successful and unsuccessful attempts. Additional metadata, such as the type of authentication or factors used, should also be collected.	2
16.3.2	Verify that failed authorization attempts are logged. For L3, this must include logging all authorization decisions, including logging when sensitive data is accessed (without logging the sensitive data itself).	2
16.3.3	Verify that the application logs the security events that are defined in the documentation and also logs attempts to bypass the security controls, such as input validation, business logic, and anti-automation.	2
16.3.4	Verify that the application logs unexpected errors and security control failures such as backend TLS failures.	2

V16.4 Log Protection

Logs are valuable forensic artifacts and must be protected. If logs can be easily modified or deleted, they lose their integrity and become unreliable for incident investigations or legal proceedings. Logs may expose internal application behavior or sensitive metadata, making them an attractive target for attackers.

This section defines requirements to ensure that logs are protected from unauthorized access, tampering, and disclosure, and that they are safely transmitted and stored in secure, isolated systems.

#	Description	Level
16.4.1	Verify that all logging components appropriately encode data to prevent log injection.	2
16.4.2	Verify that logs are protected from unauthorized access and cannot be modified.	2
16.4.3	Verify that logs are securely transmitted to a logically separate system for analysis, detection, alerting, and escalation. The aim is to ensure that if the application is breached, the logs are not compromised.	2

V16.5 Error Handling

This section defines requirements to ensure that applications fail gracefully and securely without disclosing sensitive internal details.

#	Description	Level
16.5.1	Verify that a generic message is returned to the consumer when an unexpected or security-sensitive error occurs, ensuring no exposure of sensitive internal system data such as stack traces, queries, secret keys, and tokens.	2
16.5.2	Verify that the application continues to operate securely when external resource access fails, for example, by using patterns such as circuit breakers or graceful degradation.	2
16.5.3	Verify that the application fails gracefully and securely, including when an exception occurs, preventing fail-open conditions such as processing a transaction despite errors resulting from validation logic.	2

#	Description	Level
16.5.4	Verify that a “last resort”error handler is defined which will catch all unhandled exceptions. This is both to avoid losing error details that must go to log files and to ensure that an error does not take down the entire application process, leading to a loss of availability.	3

Note: Certain languages, (including Swift, 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 Web Security Testing Guide: Testing for Error Handling
- OWASP Authentication Cheat Sheet section about error messages
- OWASP Logging Cheat Sheet
- OWASP Application Logging Vocabulary Cheat Sheet

V17 WebRTC

Control Objective

Web Real-Time Communication (WebRTC) enables real-time voice, video, and data exchange in modern applications. As adoption increases, securing WebRTC infrastructure becomes critical. This section provides security requirements for stakeholders who develop, host, or integrate WebRTC systems.

The WebRTC market can be broadly categorized into three segments:

1. Product Developers: Proprietary and open-source vendors that create and supply WebRTC products and solutions. Their focus is on developing robust and secure WebRTC technologies that can be used by others.
2. Communication Platforms as a Service (CPaaS): Providers that offer APIs, SDKs, and the necessary infrastructure or platforms to enable WebRTC functionalities. CPaaS providers may use products from the first category or develop their own WebRTC software to offer these services.
3. Service Providers: Organizations that leverage products from product developers or CPaaS providers, or develop their own WebRTC solutions. They create and implement applications

for online conferencing, healthcare, e-learning, and other domains where real-time communication is crucial.

The security requirements outlined here are primarily focused on Product Developers, CPaaS, and Service Providers who:

- Utilize open-source solutions to build their WebRTC applications.
- Use commercial WebRTC products as part of their infrastructure.
- Use internally developed WebRTC solutions or integrate various components into a cohesive service offering.

It is important to note that these security requirements do not apply to developers who exclusively use SDKs and APIs provided by CPaaS vendors. For such developers, the CPaaS providers are typically responsible for most of the underlying security concerns within their platforms, and a generic security standard like ASVS may not fully address their needs.

V17.1 TURN Server

This section defines security requirements for systems that operate their own TURN (Traversal Using Relays around NAT) servers. TURN servers assist in relaying media in restrictive network environments but can pose risks if misconfigured. These controls focus on secure address filtering and protection against resource exhaustion.

#	Description	Level
17.1.1	Verify that the Traversal Using Relays around NAT (TURN) service only allows access to IP addresses that are not reserved for special purposes (e.g., internal networks, broadcast, loopback). Note that this applies to both IPv4 and IPv6 addresses.	2
17.1.2	Verify that the Traversal Using Relays around NAT (TURN) service is not susceptible to resource exhaustion when legitimate users attempt to open a large number of ports on the TURN server.	3

V17.2 Media

These requirements only apply to systems that host their own WebRTC media servers, such as Selective Forwarding Units (SFUs), Multipoint Control Units (MCUs), recording servers, or gateway servers. Media servers handle and distribute media streams, making their security critical to protect communication between peers. Safeguarding media streams is paramount in WebRTC applications to prevent eavesdropping, tampering, and denial-of-service attacks that could compromise user privacy and communication quality.

In particular, it is necessary to implement protections against flood attacks such as rate limiting, validating timestamps, using synchronized clocks to match real-time intervals, and managing buffers to prevent overflow and maintain proper timing. If packets for a particular media session arrive too quickly, excess packets should be dropped. It is also important to protect the system from malformed packets by implementing input validation, safely handling integer overflows, preventing buffer overflows, and employing other robust error-handling techniques.

Systems that rely solely on peer-to-peer media communication between web browsers, without the involvement of intermediate media servers, are excluded from these specific media-related security requirements.

This section refers to the use of Datagram Transport Layer Security (DTLS) in the context of WebRTC. A requirement related to having a documented policy for the management of cryptographic keys can be found in the “Cryptography”chapter. Information on approved cryptographic methods can be found either in the Cryptography Appendix of the ASVS or in documents such as NIST SP 800-52 Rev. 2 or BSI TR-02102-2 (Version 2025-01).

#	Description	Level
17.2.1	Verify that the key for the Datagram Transport Layer Security (DTLS) certificate is managed and protected based on the documented policy for management of cryptographic keys.	2
17.2.2	Verify that the media server is configured to use and support approved Datagram Transport Layer Security (DTLS) cipher suites and a secure protection profile for the DTLS Extension for establishing keys for the Secure Real-time Transport Protocol (DTLS-SRTP).	2
17.2.3	Verify that Secure Real-time Transport Protocol (SRTP) authentication is checked at the media server to prevent Real-time Transport Protocol (RTP) injection attacks from leading to either a Denial of Service condition or audio or video media insertion into media streams.	2
17.2.4	Verify that the media server is able to continue processing incoming media traffic when encountering malformed Secure Real-time Transport Protocol (SRTP) packets.	2
17.2.5	Verify that the media server is able to continue processing incoming media traffic during a flood of Secure Real-time Transport Protocol (SRTP) packets from legitimate users.	3
17.2.6	Verify that the media server is not susceptible to the “ClientHello”Race Condition vulnerability in Datagram Transport Layer Security (DTLS) by checking if the media server is publicly known to be vulnerable or by performing the race condition test.	3

#	Description	Level
17.2.7	Verify that any audio or video recording mechanisms associated with the media server are able to continue processing incoming media traffic during a flood of Secure Real-time Transport Protocol (SRTP) packets from legitimate users.	3
17.2.8	Verify that the Datagram Transport Layer Security (DTLS) certificate is checked against the Session Description Protocol (SDP) fingerprint attribute, terminating the media stream if the check fails, to ensure the authenticity of the media stream.	3

V17.3 Signaling

This section defines requirements for systems that operate their own WebRTC signaling servers. Signaling coordinates peer-to-peer communication and must be resilient against attacks that could disrupt session establishment or control.

To ensure secure signaling, systems must handle malformed inputs gracefully and remain available under load.

#	Description	Level
17.3.1	Verify that the signaling server is able to continue processing legitimate incoming signaling messages during a flood attack. This should be achieved by implementing rate limiting at the signaling level.	2
17.3.2	Verify that the signaling server is able to continue processing legitimate signaling messages when encountering malformed signaling message that could cause a denial of service condition. This could include implementing input validation, safely handling integer overflows, preventing buffer overflows, and employing other robust error-handling techniques.	2

References

For more information, see also:

- The WebRTC DTLS ClientHello DoS is best documented at Enable Security's blog post aimed at security professionals and the associated white paper aimed at WebRTC developers
- RFC 3550 - RTP: A Transport Protocol for Real-Time Applications
- RFC 3711 - The Secure Real-time Transport Protocol (SRTP)
- RFC 5764 - Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP))

- RFC 8825 - Overview: Real-Time Protocols for Browser-Based Applications
- RFC 8826 - Security Considerations for WebRTC
- RFC 8827 - WebRTC Security Architecture
- DTLS-SRTP Protection Profiles

Appendix A: Glossary

- **Absolute Maximum Session Lifetime** –Also referred to as "Overall Timeout"by NIST, this is the maximal amount of time a session can remain active following authentication regardless of user interaction. This is a component of session expiration.
- **Allowlist** –A list of permitted data or operations, for example, a list of characters that are allowed to perform input validation.
- **Anti-forgery token** –A mechanism by which one or more tokens are passed in a request and validated by the application server to ensure that the request has come from an expected endpoint.
- **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** –A method of software testing that examines the functionality of an application without peering into its internal structures or workings.
- **Common Weakness Enumeration** (CWE) –A community-developed list of common software security weaknesses. It serves as a common language, a measuring stick for software security tools, and a baseline for weakness identification, mitigation, and prevention efforts.
- **Component** –A self-contained unit of code, with associated disk and network interfaces that communicates with other components.
- **Credential Service Provider** (CSP) –Also called an Identity Provider (IdP). A source of user data which may be used as an authentication source by other applications.
- **Cross-Site Script Inclusion** (XSSI) - A variant of Cross-Site Scripting (XSS) attack in which a web application retrieves malicious code from an external resource and includes that code as part of its own content.
- **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.
- **Cryptographically secure pseudo-random number generator** (CSPRNG) - A pseudorandom number generator with properties that make it suitable for use in cryptography, also referred to as a cryptographic random number generator (CRNG).
- **Datagram Transport Layer Security** (DTLS) –A cryptographic protocol which provides communication security over a network connection. It is based on the TLS protocol but adapted for protecting datagram-oriented protocols (usually over UDP). Defined in RFC 9147 for DTLS 1.3.
- **Datagram Transport Layer Security Extension to Establish Keys for the Secure Real-time Transport Protocol** (DTLS-SRTP) –A mechanism for using a DTLS handshake for establishing key material for a SRTP session. Defined in RFC 5764.
- **Design Verification** –The technical assessment of the security architecture of an application.
- **Dynamic Application Security Testing** (DAST) –Technologies are designed to detect conditions indicative of a security vulnerability in an application in its running state.
- **Dynamic Verification** –The use of automated tools that use vulnerability signatures to find problems during the execution of an application.
- **Fast IDentity Online** (FIDO) –A set of authentication standards that allow a variety of different authentication methods to be used including biometrics, Trusted Platform Modules (TPMs), USB security tokens, etc.
- **Hardware Security Module** (HSM) –Hardware component that stores cryptographic keys and other secrets in a protected manner.
- **Hibernate Query Language** (HQL) –A query language that is similar in appearance to SQL used by the Hibernate ORM library.
- **HTTP Strict Transport Security** (HSTS) –An policy which instructs the browser to only connect to the domain returning the header via TLS and when a valid certificate is presented. It is activated using the Strict-Transport-Security response header field.
- **HyperText Transfer Protocol** (HTTP) –An application protocol for distributed, collaborative, hypermedia information systems. It is the foundation of data communication for the World Wide Web.
- **HyperText Transfer Protocol over SSL/TLS** (HTTPS) –A method of securing HTTP communication by encrypting it using Transport Layer Security (TLS).
- **Identity Provider** (IdP) –Also called a Credential Service Provider (CSP) in NIST references. An entity that provides an authentication source for other applications.
- **Inactivity Timeout** –This is the length of time a session can remain active in the absence of user interaction with the application. This is a component of session expiration.
- **Input Validation** –The canonicalization and validation of untrusted user input.
- **JSON Web Token** (JWT) –RFC 7519 defines a standard for a JSON data object made up of a header section which explains how to validate the object, a body section containing a set of claims, and a signature section which contains a digital signature which can be used to validate the contents of the body section. It is a type of self-contained token.
- **Local File Inclusion** (LFI) - An attack that exploits vulnerable file inclusion procedures in an

application, leading to the inclusion of local files already present on the server.

- **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.
- **Message authentication code** (MAC) - A cryptographic checksum on data, computed by a MAC generation algorithm, that is used to provide assurance on its integrity and authenticity.
- **Multi-factor authentication** (MFA) –Authentication which includes two or more of the single factors.
- **Mutual TLS** (mTLS) –See TLS client authentication.
- **Object-relational Mapping** (ORM) –A system used to allow a relational/table-based database to be referenced and queried within an application program using an application-compatible object model.
- **One-time Password** (OTP) –A password that is uniquely generated to be used on a single occasion.
- **Open Worldwide Application Security Project** (OWASP) –The Open Worldwide 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/.
- **Password-Based Key Derivation Function 2** (PBKDF2) –A special one-way algorithm used to create a strong cryptographic key from an input text (such as a password) and an additional random salt value and can therefore be used to make it harder to crack a password offline if the resulting value is stored instead of the original password.
- **Public Key Infrastructure** (PKI) –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).
- **Public Switched Telephone Network** (PSTN) –The traditional telephone network that includes both fixed-line telephones and mobile telephones.
- **Real-time Transport Protocol** (RTP) and **Real-time Transport Control Protocol** (RTCP) –Two protocols used in association for transporting multimedia streams. Used by the WebRTC stack. Defined in RFC 3550.
- **Reference Token** –A type of token that acts as a pointer or identifier to state or metadata stored on a server, sometimes referred to as random tokens or opaque tokens. Unlike self-contained tokens, which embed some of their relevant data within the token itself, reference tokens contain no intrinsic information, instead relying on the server for context. The reference token will either be or contain a session identifier.
- **Relying Party** (RP) –Generally an application which is relying on a user having authenticated against a separate authentication provider. The application relies on some sort of token or set of signed assertions provided by that authentication provider to trust that the user is who they

say they are.

- **Remote File Inclusion** (RFI) - An attack that exploits vulnerable inclusion procedures in the application, resulting in the inclusion of remote files.
- **Scalable Vector Graphics** (SVG) –An XML-based markup language for describing two-dimensional based vector graphics.
- **Secure Real-time Transport Protocol** (SRTP) and **Secure Real-time Transport Control Protocol** (SRTCP) –A profile of the RTP and RTCP protocols providing support for message encryption, authentication and integrity protection. Defined in RFC 3711.
- **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 Assertion Markup Language** (SAML) –An open standard for single sign-on authentication based on passing signed assertions (usually XML objects) between the identity provider and the relying party.
- **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 authorization check) or when called results in a security effect (e.g., generating an audit record).
- **Security information and event management** (SIEM) - A system for threat detection, compliance and security incident management through the collection and analysis of security-related data from various sources within an organization's IT infrastructure.
- **Self-Contained Token** –A token that encapsulates one or more attributes that do not rely on server-side state or other external storage. These tokens ensure the authenticity and integrity of their contained attributes, enabling secure, "stateless"information exchange across systems. Self-contained tokens are generally secured using cryptographic techniques, such as digital signatures or message authentication codes (MACs), to ensure the authenticity, integrity, and in some cases the confidentiality of its data. Common examples include SAML Assertions and JWTs.
- **Server-side Request Forgery** (SSRF) –An attack that abuses functionality on the server to read or update internal resources. The attacker supplies or modifies a URL, which the code running on the server will read or submit data to.
- **Session Description Protocol** (SDP) –A message format for setting up multimedia session (used for example in WebRTC). Defined in RFC 4566.
- **Session Identifier** or **Session ID** –A key which identifies a stateful session stored at the back end. Will be transfered to and from the client either as or inside a "Reference Token".
- **Session Token** –A "catch-all"phrase used in this standard to refer to the token or value used in either stateless session mechanisms (which use a self-contained token) or stateful session mechanisms (which use a reference token).
- **Session Traversal Utilities for NAT** (STUN) –A protocol used to assist NAT traversal in order to establish peer-to-peer communications. Defined in RFC 3489.
- **Single-factor authenticator** –A mechanism to check that a user is authenticated. It should ei-

ther be 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).

- **Single Sign-on Authentication** (SSO) –This occurs when a user logs into one application and is then automatically logged into other applications without having to re-authenticate. For example, when logging into Google, the user will be automatically logged into other Google services such as YouTube, Google Docs, and Gmail.
- **Software bill of materials** (SBOM) - A structured, comprehensive list of all components, modules, libraries, frameworks and other resources required to build or assemble a software application.
- **Software Composition Analysis** (SCA) –A set of technologies designed to analyze application composition, dependencies, libraries and packages for security vulnerabilities of specific component versions in use. This is not to be confused with source-code analysis which is now commonly referred to as SAST.
- **Software development lifecycle** (SDLC) –The step-by-step process by which software is developed going from the initial requirements to deployment and maintenance.
- **SQL Injection** (SQLi) –A code injection technique used to attack data-driven applications, in which malicious SQL statements are inserted into an entry point.
- **Stateful Session Mechanism** –In a stateful session mechanism, the application retains session state at the backend which typically corresponds to a session token, generated using a cryptographically secure pseudo-random number generator (CSPRNG), which is issued to the end user.
- **Stateless Session Mechanism** –A stateless session mechanism will use a self-contained token which is passed to clients, and contains session information that is not necessarily stored within the service which then receives and validates the token. In reality, a service will need to have access to some session information (such as a JWT revocation list) in order to be able to enforce required security controls.
- **Static application security testing** (SAST) –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 non-running state.
- **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.
- **Time-of-check to time-of-use** (TOCTOU) –A situation where an application checks the state of a resource before using that resource, but the resource’s state can be changed between the check and the use. This can invalidate the results of the check and cause a situation where the application performs invalid actions due to this state mismatch.
- **Time based One-time Passwords** (TOTPs) - A method of generating an OTP where the current time acts as part of the algorithm to generate the password.
- **TLS client authentication**, also called **Mutual TLS** (mTLS) –In a standard TLS connection, a

client can use the certificate provided by the server to validate the server's identity. Where TLS client authentication is used, the client also uses its own private key and certificate to allow the server to also validate the client's identity.

- **Transport Layer Security** (TLS) –Cryptographic protocols that provide communication security over a network connection.
- **Traversal Using Relays around NAT** (TURN) –An extension of the STUN protocol using a TURN server as a relay when direct peer-to-peer connections cannot be established. Defined in RFC 8656.
- **Trusted execution environment** (TEE) - An isolated processing environment in which applications can be securely executed irrespective of the rest of the system.
- **Trusted Platform Module** (TPM) –A type of HSM that is usually attached to a larger hardware component such as a motherboard and acts as the "root of trust"for that system.
- **Trusted Service Layer** –Any trusted control enforcement point, 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. Trusted means that there is no concern that an untrusted user will be able to bypass or skip the layer or controls implemented at that layer.
- **Uniform Resource Identifier** (URI)- A unique string of characters that identifies a resource, such as webpage, mail address, places.
- **Uniform Resource Locator** (URL) –A string that specifies the location of resource on the Internet.
- **Universally Unique Identifier** (UUID) –A unique reference number used as an identifier in software.
- **Verifier** –The person or team that is reviewing an application against the OWASP ASVS requirements.
- **Web Real-Time Communication** (WebRTC) –A protocol stack and associated web API used for the transport of multimedia streams in web applications, usually in the context of teleconferencing. Based on SRTP, SRTCP, DTLS, SDP and STUN/TURN.
- **WebSocket over TLS** (WSS) –A practice of securing WebSocket communication by layering WebSocket over TLS protocol.
- **What You See Is What You Get** (WYSIWYG) –A type of rich content editor that shows how the content will actually look when rendered rather than showing the coding used to govern the rendering.
- **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.
- **XML eXternal Entity** (XXE) –A type of XML entity that can access local or remote content via a declared system identifier. This may lead to various injection attacks.

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://owasp.org/www-project-top-ten/
2. OWASP Web Security Testing Guide: https://owasp.org/www-project-web-security-testing-guide/
3. OWASP Proactive Controls: https://owasp.org/www-project-proactive-controls/
4. OWASP Software Assurance Maturity Model (SAMM): https://owasp.org/www-project-samm/
5. OWASP Secure Headers Project: https://owasp.org/www-project-secure-headers/

OWASP Cheat Sheet Series project

This project has several cheat sheets that will be relevant to different topics in the ASVS.

There is a mapping to the ASVS which can be found here: https://cheatsheetseries.owasp.org/IndexASVS.html

Mobile Security Related Projects

1. OWASP Mobile Security Project: https://owasp.org/www-project-mobile-security/
2. OWASP Mobile Top 10 Risks: https://owasp.org/www-project-mobile-top-10/
3. OWASP Mobile Security Testing Guide and Mobile Application Security Verification Standard: https://owasp.org/www-project-mobile-security-testing-guide/

OWASP Internet of Things related projects

1. OWASP Internet of Things Project: https://owasp.org/www-project-internet-of-things/

OWASP Serverless projects

1. OWASP Serverless Project: https://owasp.org/www-project-serverless-top-10/

Others

Similarly, the following websites 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
7. OWASP ASVS 4.0 Testing Guide https://github.com/BlazingWind/OWASP-ASVS-4.0-testing-guide

Appendix C: Cryptography Standards

The “Cryptography”chapter goes beyond simply defining best practices. It aims to enhance understanding of cryptography principles and encourage the adoption of more resilient, modern security methods. This appendix provides detailed technical information regarding each requirement, complementing the overarching standards outlined in the “Cryptography”chapter.

This appendix defines the level of approval for different cryptographic mechanisms:

- Approved (A) mechanisms can be used in applications.
- Legacy mechanisms (L) should not be used in applications but might still be used for compatibility with existing legacy applications or code onyly. While the usage of such these mechanisms is currently not considered to be a vulnerability in itself, they should be replaced by more secure and future-proof mechanisms as soon as possible.
- Disallowed mechanisms (D) must not be used because they are currently considered broken or do not provide sufficient security.

This list may be overridden in the context of a given application for various reasons including:

- new evolutions in the field of cryptography;
- compliance with regulation.

Cryptographic Inventory and Documentation

This section provides additional information for V11.1 Cryptographic Inventory and Documentation.

It is important to ensure that all cryptographic assets, such as algorithms, keys, and certificates, are regularly discovered, inventoried, and assessed. For Level 3, this should include the use of static and dynamic scanning to discover the use of cryptography in an application. Tools such as SAST and DAST may help with this but it is possible that dedicated tools would be needed to get more comprehensive coverage. Freeware examples of tools include:

- CryptoMon - Network Cryptography Monitor - using eBPF, written in python

- Cryptobom Forge Tool: Generating Comprehensive CBOMs from CodeQL Outputs

Equivalent Strengths of Cryptographic Parameters

The relative security strengths for various cryptographic systems are in this table (from NIST SP 800-57 Part 1, p.71):

Security Strength	Symmetric Key Algorithms	Finite Field	Integer Factorisation	Elliptic Curve
<= 80	2TDEA	L = 1024 N = 160	k = 1024	f = 160-223
112	3TDEA	L = 2048 N = 224	k = 2048	f = 224-255
128	AES-128	L = 3072 N = 256	k = 3072	f = 256-383
192	AES-192	L = 7680 N = 384	k = 7680	f = 384-511
256	AES-256	L = 15360 N = 512	k = 15360	f = 512+

Example of applications:

- Finite Field Cryptography: DSA, FFDH, MQV
- Integer Factorisation Cryptography: RSA
- Elliptic Curve Cryptography: ECDSA, EdDSA, ECDH, MQV

Note: that this section assumes that no quantum computer exists; if such a computer would exist, the estimates for the last 3 columns would be no longer valid.

Random Values

This section provides additional information for V11.5 Random Values.

Name	Version/Reference	Notes	Status
`/dev/random`	Linux 4.8+ (Oct 2016), also found in iOS, Android, and other Linux-based POSIX operating systems. Based on RFC7539	Utilizing ChaCha20 stream. Found in iOS `SecRandomCopyBytes` and Android `SecureRandom` with the correct settings provided to each.	A

Name	Version/Reference	Notes	Status
`/dev/urandom`	Linux kernel's special file for providing random data	Provides high-quality, entropy sources from hardware randomness	A
`AES-CTR-DRBG`	NIST SP800-90A	As used in common implementations, such as Windows CNG API `BCryptGenRandom` set by `BCRYPT_RNG_ALGORITHM`.	A
`HMAC-DRBG`	NIST SP800-90A		A
`Hash-DRBG`	NIST SP800-90A		A
`getentropy()`	OpenBSD, available in Linux glibc 2.25+ and macOS 10.12+	Provides secure random bytes directly from the kernel's entropy source with a straightforward and minimal API. It's more modern and avoids pitfalls associated with older APIs.	A

The underlying hash function used with HMAC-DRBG or Hash-DRBG must be approved for this usage.

Cipher Algorithms

This section provides additional information for V11.3 Encryption Algorithms.

Approved cipher algorithms are listed in order of preference.

Symmetric Key Algorithms	Reference	Status
AES-256	FIPS 197	A
Salsa20	Salsa 20 specification	A
XChaCha20	XChaCha20 Draft	A
XSalsa20	Extending the Salsa20 nonce	A

Symmetric Key Algorithms	Reference	Status
ChaCha20	RFC 8439	A
AES-192	FIPS 197	A
AES-128	FIPS 197	L
2TDEA		D
TDEA (3DES/3DEA)		D
IDEA		D
RC4		D
Blowfish		D
ARC4		D
DES		D

AES Cipher Modes

Block ciphers, such as AES, can be used with different modes of operations. Many modes of operations, such as Electronic codebook (ECB), are insecure and must not be used. The Galois/Counter Mode (GCM) and Counter with cipher block chaining message authentication code (CCM) modes of operations provide authenticated encryption and should be used in modern applications.

Approved modes are listed in order of preference.

Mode	Authenticated	Reference	Status	Restriction
GCM	Yes	NIST SP 800-38D	A	
CCM	Yes	NIST SP 800-38C	A	
CBC	No	NIST SP 800-38A	L	
CCM-8	Yes		D	
ECB	No		D	
CFB	No		D	
OFB	No		D	
CTR	No		D	

Notes:

- All encrypted messages must be authenticated. For ANY use of CBC mode there MUST be an associated hashing MAC algorithm to validate the message. In general, this MUST be applied in

the Encrypt-Then-Hash method (but TLS 1.2 uses Hash-Then-Encrypt instead). If this cannot be guaranteed, then CBC MUST NOT be used. The only application where encryption without a MAC algorithm is allowed is disk encryption.
- If CBC is used, it shall be guaranteed that the verification of the padding is performed in constant time.
- When using CCM-8, the MAC tag only has 64 bits of security. This does not conform to requirement 6.2.9 which requires at least 128 bits of security.
- Disk encryption is considered out of scope for the ASVS. Therefore this appendix does not list any approved method for disk encryption. For this usage, encryption without authentication is usually accepted and the XTS, XEX and LRW modes are typically used.

Key Wrapping

Cryptographic key wrap (and corresponding key unwrap) is a method of protecting an existing key by encapsulating (i.e., wrapping) it by employing an additional encryption mechanism so that the original key is not obviously exposed, e.g., during a transfer. This additional key used to protect the original key is referred to as the wrap key.

This operation may be performed when it is desirable to protect keys in places deemed untrustworthy, or to send sensitive keys over untrusted networks or within applications. However, serious consideration should be given to understanding the nature (e.g., the identity and the purpose) of the original key prior to committing to a wrap/unwrap procedure as this may have repercussions for both source and target systems/applications in terms of security and especially compliance which may include audit trails of a key's function (e.g., signing) as well as appropriate key storage.

Specifically, AES-256 MUST be used for key wrapping, following NIST SP 800-38F and considering forward-looking provisions against the quantum threat. Cipher modes using AES are the following, in order of preference:

Key Wrapping	Reference	Status
KW	NIST SP 800-38F	A
KWP	NIST SP 800-38F	A

AES-192 and AES-128 MAY be used if the use case demands it, but its motivation MUST be documented in the entity's cryptography inventory.

Authenticated Encryption

With the exception of disk encryption, encrypted data must be protected against unauthorized modification using some form of authenticated encryption (AE) scheme, usually using an authenticated encryption with associated data (AEAD) scheme.

The application should preferably use an approved AEAD scheme. It might alternatively combine an approved cipher scheme and an approved MAC algorithm with a Encrypt-then-MAC construct.

MAC-then-encrypt is still allowed for compatibility with legacy applications. It is used in TLS v1.2 with old ciphers suites.

AEAD mechanism	Reference	Status
AES-GCM	SP 800-38D	A
AES-CCM	SP 800-38C	A
ChaCha-Poly1305	RFC 7539	A
AEGIS-256	AEGIS: A Fast Authenticated Encryption Algorithm (v1.1)	A
AEGIS-128	AEGIS: A Fast Authenticated Encryption Algorithm (v1.1)	A
AEGIS-128L	AEGIS: A Fast Authenticated Encryption Algorithm (v1.1)	A
Encrypt-then-MAC		A
MAC-then-encrypt		L

Hash Functions

This section provides additional information for V11.4 Hashing and Hash-based Functions.

Hash Functions for General Use Cases

The following table lists hash functions approved in general cryptographic use cases such as digital signatures:

- Approved hash functions provide strong collision resistance and are suitable for high-security applications.
- Some of these algorithms offer strong resistance to attacks when used with proper cryptographic key management, and so are additionally approved for HMAC, KDF, and RBG functions.

- Hash function with less than 254 bit of output have insufficient collision resistancea and must not be used for digital signature or other applications requiring collision resistance. For other usages, they might be used for compatibility and verification ONLY with legacy systems but must not be used in new designs.

Hash function	Reference	Status	Restrictions
SHA3-512	FIPS 202	A	
SHA-512	FIPS 180-4	A	
SHA3-384	FIPS 202	A	
SHA-384	FIPS 180-4	A	
SHA3-256	FIPS 202	A	
SHA-512/256	FIPS 180-4	A	
SHA-256	FIPS 180-4	A	
SHAKE256	FIPS 202	A	
BLAKE2s	BLAKE2: simpler, smaller, fast as MD5	A	
BLAKE2b	BLAKE2: simpler, smaller, fast as MD5	A	
BLAKE3	BLAKE3 one function, fast everywhere	A	
SHA-224	FIPS 180-4	L	Not suitable for HMAC, KDF, RBG, digital signatures

Hash function	Reference	Status	Restrictions
SHA-512/224	FIPS 180-4	L	Not suitable for HMAC, KDF, RBG, digital signatures
SHA3-224	FIPS 202	L	Not suitable for HMAC, KDF, RBG, digital signatures

Hash function	Reference	Status	Restrictions
SHA-1	RFC 3174 & RFC 6194	L	Not suitable for HMAC, KDF, RBG, digital signatures
CRC (any length)		D	
MD4	RFC 1320	D	
MD5	RFC 1321	D	

Hash Functions for Password Storage

For secure password hashing, dedicated hash functions must be used. These slow-hashing algorithms mitigate brute-force and dictionary attacks by increasing the computational difficulty of password cracking.

KDF	Reference	Required Parameters	Status
argon2id	RFC 9106	t = 1: m ≥ 47104 (46 MiB), p = 1t = 2: m ≥ 19456 (19 MiB), p = 1t ≥ 3: m ≥ 12288 (12 MiB), p = 1	A
scrypt	RFC 7914	p = 1: N ≥ 2^17 (128 MiB), r = 8p = 2: N ≥ 2^16 (64 MiB), r = 8p ≥ 3: N ≥ 2^15 (32 MiB), r = 8	A
bcrypt	A Future-Adaptable Password Scheme	cost ≥ 10	A
PBKDF2-HMAC-SHA-512	NIST SP 800-132, FIPS 180-4	iterations ≥ 210,000	A

KDF	Reference	Required Parameters	Status
PBKDF2-HMAC-SHA-256	NIST SP 800-132, FIPS 180-4	iterations ≥ 600,000	A
PBKDF2-HMAC-SHA-1	NIST SP 800-132, FIPS 180-4	iterations ≥ 1,300,000	L

Approved password-based key derivations functions can be used for password storage.

Key Derivation Functions (KDFs)

General Key Derivation Functions

KDF	Reference	Status
HKDF	RFC 5869	A
TLS 1.2 PRF	RFC 5248	L
MD5-based KDFs	RFC 1321	D
SHA-1-based KDFs	RFC 3174 & RFC 6194	D

Password-based Key Derivation Functions

KDF	Reference	Required Parameters	Status
argon2id	RFC 9106	t = 1: m ≥ 47104 (46 MiB), p = 1t = 2: m ≥ 19456 (19 MiB), p = 1t ≥ 3: m ≥ 12288 (12 MiB), p = 1	A
scrypt	RFC 7914	p = 1: N ≥ 2^17 (128 MiB), r = 8p = 2: N ≥ 2^16 (64 MiB), r = 8p ≥ 3: N ≥ 2^15 (32 MiB), r = 8	A
PBKDF2-HMAC-SHA-512	NIST SP 800-132, FIPS 180-4	iterations ≥ 210,000	A

KDF	Reference	Required Parameters	Status
PBKDF2-HMAC-SHA-256	NIST SP 800-132, FIPS 180-4	iterations ≥ 600,000	A
PBKDF2-HMAC-SHA-1	NIST SP 800-132, FIPS 180-4	iterations ≥ 1,300,000	L

Key Exchange Mechanisms

This section provides additional information for V11.6 Public Key Cryptography.

KEX Schemes

A security strength of 112 bits or above MUST be ensured for all Key Exchange schemes, and their implementation MUST follow the parameter choices in the following table.

Scheme	Domain Parameters	Forward Secrecy	Status
Finite Field Diffie-Hellman (FFDH)	L >= 3072 & N >= 256	Yes	A
Elliptic Curve Diffie-Hellman (ECDH)	f >= 256-383	Yes	A
Encrypted key transport with RSA-PKCS#1 v1.5		No	D

Where the following parameters are:

- k is the key size for RSA keys.
- L is the size of the public key and N is the size of the private key for finite field cryptography.
- f is the range of key sizes for ECC.

Any new implementation MUST NOT use any scheme that is NOT compliant with NIST SP 800-56A & B and NIST SP 800-77. Specifically, IKEv1 MUST NOT be used in production.

Diffie-Hellman groups

The following groups are approved for implementations of Diffie-Hellman key exchange. Security strengths are documented in NIST SP 800-56A, Appendix D, and NIST SP 800-57 Part 1 Rev.5.

Group	Status
P-224, secp224r1	A
P-256, secp256r1	A
P-384, secp384r1	A
P-521, secp521r1	A
K-233, sect233k1	A
K-283, sect283k1	A
K-409, sect409k1	A
K-571, sect571k1	A
B-233, sect233r1	A
B-283, sect283r1	A
B-409, sect409r1	A
B-571, sect571r1	A
Curve448	A
Curve25519	A
MODP-2048	A
MODP-3072	A
MODP-4096	A
MODP-6144	A
MODP-8192	A
ffdhe2048	A
ffdhe3072	A
ffdhe4096	A
ffdhe6144	A
ffdhe8192	A

Message Authentication Codes (MAC)

Message Authentication Codes (MACs) are cryptographic constructs used to verify the integrity and authenticity of a message. A MAC takes a message and a secret key as inputs and produces a fixed-size tag (the MAC value). MACs are widely used in secure communication protocols (e.g., TLS/SSL) to ensure that messages exchanged between parties are authentic and intact.

MAC Algorithm	Reference	Status	Restrictions
HMAC-SHA-256	RFC 2104 & FIPS 198-1	A	
HMAC-SHA-384	RFC 2104 & FIPS 198-1	A	
HMAC-SHA-512	RFC 2104 & FIPS 198-1	A	
KMAC128	NIST SP 800-185	A	
KMAC256	NIST SP 800-185	A	
BLAKE3 (keyed_hash mode)	BLAKE3 one function, fast everywhere	A	
AES-CMAC	RFC 4493 & NIST SP 800-38B	A	
AES-GMAC	NIST SP 800-38D	A	
Poly1305-AES	The Poly1305-AES message-authentication code	A	
HMAC-SHA-1	RFC 2104 & FIPS 198-1	L	
HMAC-MD5	RFC 1321	D	

Digital Signatures

Signature schemes MUST use approved key sizes and parameters per NIST SP 800-57 Part 1.

Signature Algorithm	Reference	Status
EdDSA (Ed25519, Ed448)	RFC 8032	A
XEdDSA (Curve25519, Curve448)	XEdDSA	A
ECDSA (P-256, P-384, P-521)	FIPS 186-4	A
RSA-RSSA-PSS	RFC 8017	A

Signature Algorithm	Reference	Status
RSA-SSA-PKCS#1 v1.5	RFC 8017	D
DSA (any key size)	FIPS 186-4	D

Post-Quantum Encryption Standards

PQC implementations must be in line with FIPS-203/204/205 as there is minimal hardened code nor implementation reference yet. https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards

The proposed mlkem768x25519 post-quantum hybrid TLS key agreement method is supported by major browsers such as Firefox release 132 and Chrome release 131. It may be used in cryptographic testing environments or when available within industry- or government-approved libraries.

Appendix D: Recommendations

Introduction

Whilst preparing version 5.0 of the Application Security Verification Standard (ASVS), it became clear that there were a number of existing and newly suggested items that shouldn't be included as requirements in 5.0. This may have been because they were not in scope for ASVS as per the definition for 5.0 or alternatively it was felt that while they were a good idea, they could not be made mandatory.

Not wanting to lose all these items entirely, some have been captured in this appendix.

Recommended, in-scope mechanisms

The following items are in-scope for ASVS. They should not be made mandatory but it is strongly recommended to consider them as part of a secure application.

- A password strength meter should provided to help users set a stronger password.
- Create a publicly available security.txt file at the root or .well-known directory of the application that clearly defines a link or e-mail address for people to contact owners about security issues.
- Client-side input validation should be enforced in addition to validation at a trusted service layer as this provides a good opportunity to discover when someone has bypassed client-side controls in an attempt to attack the application.
- Prevent accidentally accessible and sensitive pages from appearing in search engines using a robots.txt file, the X-Robots-Tag response header or a robots html meta tag.
- When using GraphQL, implement authorization logic at the business logic layer instead of the GraphQL or resolver layer to avoid having to handle authorization on every separate interface.

References:

- More information on security.txt including a link to the RFC

Software Security principles

The following items were previously in ASVS but are not really requirements. Rather they are principles to consider when implementing security controls that when followed will lead to more robust controls. These include:

- Security controls should be centralized, simple (economy of design), verifiably secure, and reusable. This should avoid duplicate, missing, or ineffective controls.
- Wherever possible, use previously written and well-vetted security control implementations rather than relying on implementing controls from scratch.
- Ideally, a single access control mechanism should be used to access protected data and resources. All requests should pass through this single mechanism to avoid copy and paste or insecure alternative paths.
- Attribute or feature-based access control is a recommended pattern whereby the code checks the user's authorization for a feature or data item rather than just their role. Permissions should still be allocated using roles.

Software Security processes

There are a number of security processes which were removed from ASVS 5.0 but are still a good idea. The OWASP SAMM project may be a good source for how to effectively implement these processes. The items which were previously in ASVS include:

- Verify the use of a secure software development lifecycle that addresses security in all stages of development.
- 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.
- 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"
- Verify availability of a secure coding checklist, security requirements, guideline, or policy to all developers and testers.
- Verify that an ongoing process exists to ensure that the application source code is free from backdoors, malicious code (e.g., salami attacks, logic bombs, time bombs), and undocumented or hidden features (e.g., Easter eggs, insecure debugging tools). Complying with this section is not possible without complete access to source code, including third-party libraries, and is therefore probably only suitable for applications requiring the very highest levels of security.

- Verify that mechanisms are in place to detect and respond to configuration drift in deployed environments. This may include using immutable infrastructure, automated redeployment from a secure baseline, or drift detection tools that compare current state against approved configurations.
- Verify that configuration hardening is performed on all third-party products, libraries, frameworks, and services as per their individual recommendations.

References:

- OWASP Threat Modeling Cheat Sheet
- OWASP Threat modeling
- OWASP Software Assurance Maturity Model Project
- Microsoft SDL

Appendix E - Contributors

We gratefully acknowledge the contributions of the following people who have commented or opened pull requests since the the release of ASVS 4.0.0.

If you are aware of any mistakes or would like your name to appear differently, please let us know.

Johan Sydseter (sydseter)	luis servin (lfservin)	Oleksii Dovydkov (oleksiidov)	IZUKA Masahiro (maizuka)
James Sulinski (jsulinski)	Eli Saad (ThunderSon)	kkshitish9	Andrew van der Stock (vanderaj)
Rick M (kingthorin)	Bankde Eakasit (Bankde)	Michael Gargiullo (mgargiullo)	Raphael Dunant (Racater)
Cesar Kohl (cesarkohl)	inaz0	Joerg Bruenner (JoergBruenner)	David Deatherage (securitydave)
John Carroll (yosignals)	Jim Fenton (jimfenton)	Matteo Pace (M4tteoP)	Sebastien gioria (SPoint42)
Steven van der Baan (vdbaan)	Jeremy Bonghwan Choi (jeremychoi)	craig-shony	Riccardo Sirigu (ricsirigu)
Tomasz Wrobel (tw2as)	Alena Dubeshko (belalena)	Rafael Green (RafaelGreen1)	mjang-cobalt
clallier94	Kevin W. Wall (kwwall)	Jordan Sherman (jsherm-fwdsec / deleterepo)	Ingo Rauner (ingo-rauner)

Dirk Wetter (drwetter)	Moshe Zioni (moshe-apiiro)	Patrick Dwyer (coderpatros)	David Clarke (davidclarke-au)
Takaharu Ogasa (takaharuogasa)	Arkadii Yakovets (arkid15r)	Motoyasu Saburi (motoyasu-saburi)	leirn
wet-certitude	timhemel	RL Thornton (thornshadow99)	Thomas Bandt (aspnetde)
Roel Storms (roelstorms)	Jeroen Willemsen (commjoen)	anonymous-31	Kamran Saifullah (deFr0ggy)
Steve Springett (stevespringett)	Spyros (northdpole)	Hans Herrera (hansphp)	Marx314
CarlosAllendes	Yonah Russ (yruss972)	Sander Maijers (sanmai-NL)	Luboš Bretschneider (bretik)
Eva Sarafianou (esarafianou)	ataseren	Steve Thomas (Sc00bz)	Dominique RIGHETTO (righettod)
Steven van der Baan (svdb-ncc)	Michael Vacarella (Aif4thah)	Tonimir Kisasondi (tkisason)	Stefan Streichsbier (streichsbaer)
hi-unc1e	sb3k (starbuck3000)	mario-platt	Devdatta Akhawe (devd)
Michael Gissing (scolytus)	Jet Anderson (thatsjet)	Dave Wichers (davewichers)	Jonny Schnittger (JonnySchnittger)
Silvia Väli (silviavali)	jackgates73	1songb1rd	Timur - (timurozkul)
Gareth Heyes (hackvertor)	appills	suvikaartinen	chaals (chaals)
DanielPharos (AtlasHackert)	will Farrell (willfarrell)	Alina Vasiljeva (avasiljeva)	Paul McCann (ismisepaul)
Sage (SajjadPourali)	rbsec	Benedikt Bauer (mastacheata)	James Jardine (jamesjardine)
Mark Burnett (m8urnett)	dschwarz91	Cyber-AppSec (Cyber-AppSec)	Tib3rius
BitnessWise (bitnesswise)	damienbod (damienbod)	Jared Meit (jmeit-fwdsec)	Stefan Seelmann (sseelmann)
Brendan O'Connor (ussjoin)	Andrei Titov (andrettv)	Hans-Petter Fjeld (atluxity)	markehack
Neil Madden (NeilMadden)	Michael Geramb (mgeramb)	Osama Elnaggar (ossie-git)	mackowski

Ravi Balla (raviballa)	Hazana (hazanasec)	David Means (dmeans82)	Alexander Stein (tohch4)
BaeSenseii (baesenseii)	Vincent De Schutter (VincentDS)	S Bani (sbani)	Mitsuaki Akiyama (mak1yama)
Christopher Loessl (hashier)	victorxm	Michal Rada (michalradacz)	Veeresh Devireddy (drveresh)
MaknaSEO	darkzero2022	Liam (LiamDobbelaere)	Frank Denis (jedisct1)
Otto Sulin (ottosulin)	carllaw6885	Anders Johan Holmefjord (aholmis)	Richard Fritsch (rfricz)
mesutgungor	Scott Helme (ScottHelme)	Carlo Reggiani (carloreggiani)	Suyash Srivastava (suyash5053)
Mark Potter (markonweb)	Arjan Lamers (alamers)	Gøran Breivik (gobrtg)	flo-blg
Guillaume Déflache (guillaume-d)	Toufik Airane (toufik-airane)	Keith Hoodlet (securingdev)	Sinner (SoftwareSinner)
iloving	Jeroen Beckers (TheDauntless)	Joubin Jabbari (joubin)	yu fujioka (fujiokayu)
execjosh (execjosh)	Alicja Kario (tomato42)	Sidney Ribeiro (srjsoftware)	Gabriel Marquet (Gby56)
Drew Schulz (drschulz)	bedirhan	muralito	Ronnie Flathers (ropnop)
Philippe De Ryck (philippederyck)	Malte (mal33)	MazeOfThoughts	Andreas Falk (andifalk)
Javi (javixeneize)	Daniel Hahn (averell23)	borislav-c	Robin Wood (digininja)
miro2ns	Jan Dockx (jandockx)	vipinsaini434	priyanshukumar397
Nat Sakimura (sakimura)	Benjamin Häublein (BenjaminHae)	unknown-user-from	Ali Ramazan TAŞDELEN (alitasdln)
Pedro Escaleira (oEscal)	Josh (josh-hemphill)	Tim Würtele (SECtim)	AviD (avidouglen)
SheHacksPurple (shehackspurple)	fcerullo-cycubix	Hector Eryx Paredes Camacho (heryxpc)	Irene Michlin (irene221b)
Jonah Y-M (TG-Techie)	Dhiraj Bahroos (bahroos)	Jef Meijvis (jefmeijvis)	IzmaDoesItbeta

Abdessamad TEMMAR (TmmmmmR)	sectroyer	Soh Satoh (sohsatoh)	regoravalaz
james-t (james-bitherder)	Aram Hovsepyan (aramhovsepyan)	JaimeGomezGarciaSan	ValdiGit01
iwatachan (ishowta)	Vinod Anandan (VinodAnandan)	Kevin Kien (KevinKien)	paul-williamson-swoop
endergzr	Radhwan Alshamamri (Rado0z)	Grant Ongers (rewtd)	Cure53 (cure53)
AliR2Linux	Ads Dawson (Gang-GreenTemperTatum)	William Reyor (BillReyor)	gabe (gcrow)
mascotter	luissaiz	Suren Manukyan (vx-sec)	Piotr Gliźniewicz (pgliźniewicz)
Tadeusz Wachowski (tadeuszwachowski)	Nasir aka Nate (andesec)	settantasette	Lars Haulin (LarsH)
Terence Eden (edent)	JasmineScholz	Arun Sivadasan (teavanist)	Yusuf GÜR (yusuffgur)
Troy Marshall (troymarshall)	Tanner Prynn (tprynn)	Nick K. (nickific)	raoul361
Azeem Ilyas (TheAxZim)	Evo Stamatov (avioli)	Tim Potter (timpotter87)	Gavin Ray (GavinRay97)
monis (demideus)	Marcin Hoppe (MarcinHoppe)	Grambulf (ramshazar)	Jordan Pike (computersarebad)
Jason Rogers (jason-invision)	Ben Hall (benbhall)	JamesPoppyCock (jamesly123)	WhiteHackLabs (whitehacklabs)
Alex Gaynor (alex)	Filip van Laenen (filipvanlaenen)	jeurgen	GraoMelo
Andreas Kurtz (ay-kay)	Tom Tervoort (TomTervoort)	old man (deveras)	Marco Schnüriger (marcortw)
stiiin	infoseclearn (teaminfoseclearn)	hljupkij	Noe (nmarher)
Lyz (lyz-code)	Martin Riedel (mrtnrdl)	KIM Jaesuck (tcaesvk)	Barbara Schachner (bschach)
René Reuter (AresSec)	carhackpils	Tyler (tyler2cr)	Hugo (hasousa)
Wouter Bloeyaert (Someniak)	Mark de Rijk (markderijkinfosec)	Ramin (picohub)	Philip D. Turner (philipdturner)

Will Chatham (willc)