afine

Desktop Application Security Verification Standard

DASVS v1.0

Table of Contents

Cover Page

About

If you build, test, or secure desktop applications, you've probably noticed there's no good security standard for what you do. Web apps have OWASP ASVS. Mobile apps have MASVS. Desktop apps haven't had anything comparable.

DASVS gives you a practical framework for building and verifying secure desktop software. Rather than abstract theory or vague "best practices," it provides precise, testable security requirements - whether you're writing code, performing a security assessment, or evaluating vendor software.

Desktop apps are everywhere in enterprises, handling everything from customer data to financial records. They also face unique security challenges, including deep OS integration, broad file system access, and hardware interactions, which web and mobile security guides often overlook. This standard fills that gap with requirements specifically built for desktop threats.

Copyright and License

Creators

AFINE Team

Contributors

N/A

Introduction

Why We Built DASVS

Desktop applications have often been overlooked in application security standards. We have excellent frameworks for web applications and mobile apps, but desktop software - which runs on millions of enterprise workstations and handles everything from customer data to financial records - has lacked comprehensive security guidance.

Desktop applications present their own unique security headaches. They integrate more deeply with operating systems than web apps do. They can access local resources in ways mobile apps can't. They often run with elevated privileges and interact directly with hardware. These aren't just minor differences - they fundamentally change what you need to worry about from a security perspective.

We created DASVS to address this gap. The goal is straightforward: provide organizations with a framework they can use to build, test, and verify secure desktop applications, regardless of the operating system they are targeting. By standardizing what we mean by "secure desktop application", we aim to raise the overall security bar and provide developers and security professionals with a common vocabulary.

How Desktop Apps Are Different

Desktop applications differ fundamentally from web and mobile applications in several key aspects that affect their security profile:

Desktop applications aren't just web or mobile apps that happen to run on a computer - they are architecturally different in ways that matter for security.

Take web applications. Most of the actual application logic runs server-side, behind firewalls and security controls. The browser acts as a relatively dumb client. This creates natural defense layers.

Mobile applications run entirely on the device, which sounds similar to desktop apps. But mobile platforms impose strict sandboxing and security boundaries. Yes, attackers can reverse engineer mobile apps, but the platform itself restricts what those apps can do with system resources.

Desktop applications have neither advantage. They execute locally with broad access to system resources and minimal isolation from the OS. This creates a larger attack surface that demands careful security design.

The security boundaries tell a similar story. Browsers sandbox web applications away from local resources. Mobile operating systems enforce capability restrictions. Desktop applications often have free rein to access files, registry entries, system services - whatever they need. You are not getting automatic platform-level protection, so you need robust application-level controls to compensate.

Data storage highlights another key difference. Web applications store limited local data, mostly cookies and local storage, with size restrictions. Mobile apps utilize structured storage with platform-provided encryption options, although developers often do not use them properly. Desktop applications can create, modify, and delete files throughout the entire file system, frequently with administrative privileges. The potential for data exposure is substantially higher.

Even updates work differently. Web applications update transparently when users visit them. Mobile apps are updated through curated app stores that provide at least some level of security screening. Desktop applications utilize various update mechanisms, many of which require administrative access, which can become malware distribution channels if not adequately secured.

These aren't academic distinctions. They are why desktop applications need their own security verification approach.

Cross-Platform Support

DASVS provides guidance that works across all major desktop operating systems:

- Windows
- macOS
- Linux

Each platform has its quirks and specific security features, but the fundamental security principles remain consistent.

What Makes Desktop Security Hard

Several challenges are unique to desktop applications, or at least more pronounced than in web or mobile contexts.

Operating System Integration

Desktop applications typically integrate deeply with the operating system - accessing system services, calling native libraries, and manipulating system resources. When developers fail to secure these interactions properly, they create vulnerabilities that are more difficult to exploit

in more isolated environments. Operating system hardening becomes particularly important because desktop applications are more susceptible to threats that target OS-level vulnerabilities.

Privilege Escalation

Many desktop applications need elevated privileges to install or perform certain functions. Without careful security design, this creates opportunities for privilege escalation attacks. Managing least privilege is challenging when your application legitimately needs to interact with protected system resources. Unlike web applications, which typically run in a controlled server environment, desktop applications often run with the user's privileges - or worse, with administrative rights.

Offline Access and Local Operations

Desktop applications typically work offline and process data locally. This is convenient for users but problematic for security. You can't rely on network-level defenses, such as web application firewalls. Server-side validation isn't available to catch malicious input. Everything needs to happen locally, which means your local security controls must be robust enough to handle threats that web applications may never encounter.

Local Storage Risks

When desktop applications store data locally, they often do so with less structured protection than you would find in enterprise server environments. This creates risks around data confidentiality, integrity, and availability, especially on shared computers or systems that are compromised. The security of stored data becomes your responsibility in ways it might not be for web applications, where most sensitive data lives on protected servers.

Hardware Interactions

Desktop applications frequently interact with hardware - USB devices, biometric sensors, smart cards, printers, and external drives. Each of these interactions opens potential attack vectors that web application security frameworks rarely address. A compromised USB device or malicious peripheral can become an attack vector against your application.

Legacy Baggage

Desktop applications often depend on older libraries, frameworks, or operating system components that may contain known vulnerabilities or lack modern security features. Users might be running your application on outdated operating systems that no longer receive security patches. This legacy dependency problem requires special attention to vulnerability management that extends beyond simply securing your own code.

Installation Risks

The installation process itself deserves security attention. Desktop applications can be installed through various methods, including official installers, package managers, direct downloads, and enterprise deployment tools. Each method represents a potential avenue for malware distribution or system compromise. Securing the application isn't enough if the installation process itself is vulnerable. This standard explicitly addresses installation security because it is too important to ignore.

What DASVS Does

DASVS helps in several ways. It provides a benchmark for measuring desktop application security. Developers and security teams get concrete guidance on which controls to implement and when. Security testers get a comprehensive verification checklist. And when dealing with vendors or writing security requirements, you can reference DASVS directly rather than starting from scratch.

The standard applies to desktop applications, regardless of what you are building. Consumer software, enterprise apps, simple utilities, mission-critical systems - DASVS scales to match your risk level.

How We Approach Desktop Security

DASVS has three main parts that work together.

The verification requirements are the heart of it - specific security controls you can actually test. Each one is clear enough that there's no debate about whether you've met it or not. Either the control works or it doesn't.

Then there are best practices that explain how to implement these controls effectively. It is one thing to know you need input validation. It is another thing to understand how to do it right so it actually stops attacks instead of just looking good on paper.

We also include platform-specific recommendations, as implementation details are crucial. The standard stays technology-neutral overall, but let's be honest - securing a Windows application sometimes means doing things differently than you would on Linux or macOS.

The entire concept is based on a few key principles. You can measure whether requirements are met. They apply regardless of what programming language or framework you are using. They are specific enough to actually implement without having to guess what we meant. Together, they address the security issues that genuinely matter for desktop applications.

Who This is For

Different people will use DASVS in various ways.

Security auditors and penetration testers get a systematic framework for evaluating desktop application security. Instead of treating every engagement as if you are starting from scratch, you have a baseline to work from. That lets you spend your time finding the subtle vulnerabilities that scanners can't catch.

Developers receive clear guidance on what security actually entails at each stage of development. Everyone says "build security in from the start," but what does that mean? These requirements show you. Fixing security problems early is way cheaper than trying to patch them into a finished product.

Security architects can utilize these patterns and controls during the design phase, when security decisions have the most impact. Designing security properly beats trying to fix a broken architecture later.

QA teams can expand testing beyond just "does it work" to include "is it secure?". Security testing shouldn't be someone else's problem. It belongs in QA alongside your performance tests and functional tests.

Compliance teams can map these requirements to the relevant regulatory framework, such as GDPR, HIPAA, PCI DSS, or industry-specific rules. DASVS provides security controls that support compliance without requiring you to build everything from scratch.

IT leaders and executives gain a structured approach to establishing security baselines, making informed risk decisions, and setting clear expectations for desktop application security throughout the organization.

Framework, Not Certification

DASVS is a verification framework. It is not a certification program, and we are not handing out official approvals or stamps of legitimacy. If you require formal certification for compliance or business purposes, consider established programs such as ISO 27001. DASVS tells you what to test for and how to verify security controls work - think of it as a testing guide, not a certification body.

This Is Just the Beginning

You are reading the first edition of DASVS. Like any good security standard, it needs to evolve as threats change and technology advances. We expect future versions to incorporate lessons learned from real-world usage, address emerging threats, and adapt to the maturation of

desktop application security practices. If you identify gaps, have suggestions, or discover more effective ways to address these security concerns, we would like to hear about them. Community feedback is how security standards improve.

Using This Standard

Practical Implementation of DASVS

Organizations can use DASVS in several ways.

Security teams can use it as an assessment checklist to identify vulnerabilities systematically. Instead of poking around randomly, you've got a structured approach. You are less likely to miss critical issues.

Developers can integrate DASVS requirements into the entire development cycle, from design through testing. Security goes into design reviews, coding guidelines, and test plans. Finding problems early is cheap. Finding them in production is not.

For third-party assessments, DASVS gives penetration testers and consultants a clear scope. Different testers work differently, but you'll know they are checking what matters for desktop application risks.

As a benchmark, you can track security improvements. Last quarter, you met 60% of Level 2 requirements; this quarter 75%. That's progress you can show.

For procurement, DASVS provides contract language that outlines vendor security expectations. It provides acceptance criteria before deployment. Teams can self-assess against recognized controls. Security testing fits into existing QA work. It also serves as training material for building security awareness.

Application Security Verification Levels

DASVS defines three verification levels with increasing security rigor. Think of these as security postures matched to different risk profiles.

Level 1: Baseline Security

Level 1 covers common desktop vulnerabilities - basic input validation, simple authentication, and essential data protection. Every desktop application should meet Level 1 requirements at a minimum. These are the security controls that protect against widespread, well-known attacks.

This level addresses threats like those in the OWASP Top 10 Desktop Application Security Risks. If you are already practicing basic secure coding, Level 1 shouldn't require massive additional effort. Most Level 1 requirements can be verified through black-box testing, which makes them accessible even if you do not have source code access.

Level 2: Enhanced Security

Level 2 is appropriate when you are handling sensitive data or performing business-critical operations. At this level, you are implementing more sophisticated authentication, granular access controls, comprehensive logging, and thorough input validation.

Meeting Level 2 means integrating security activities throughout your development process. You are conducting threat modeling during the design phase. You are conducting security-focused code reviews. You are performing more thorough testing than just running an automated scanner. It is not just about the code you write - it is about how you build and test software.

Level 3: High-Risk Security

Level 3 is for when a security breach would be catastrophic. Think financial systems, healthcare apps, defense software, and critical infrastructure. At this level, you assume that skilled attackers will specifically target your application.

Level 3 means stringent crypto requirements, minimizing the attack surface wherever possible, implementing layered defenses throughout the app, and maintaining thorough security monitoring. You'll need detailed documentation and security experts directly involved in the development process. These controls assume someone competent is actively trying to break in.

Select your level based on what you are protecting and the applicable regulations. If you are unsure which level you need, choose the higher one. Maintaining strong security is easier than cleaning up after a breach has occurred.

Referencing DASVS Requirements

DASVS uses a straightforward identifier system for requirements:

- Basic format: <**chapter**>.<**section**>.<**requirement**>
  Example: **4.2.3** refers to requirement 3 in Section 2 of Chapter 4.
- Expanded format for version clarity: **v**<**version**><**chapter**>.<**section**>.<**requirement**>
  Example: **v1.0-4.2.3** specifies exactly which version you are referencing

This numbering system facilitates easy reference to specific requirements in documentation, bug reports, or security discussions. When you tell a developer "we need to address V4.2.3," they know exactly what you are talking about.

Tailoring DASVS to Your Needs

Do not treat DASVS like a rigid checklist. Adjust it to fit your situation.

Start with risk assessment. What data are you protecting? What regulations do you need to follow? What threats actually worry you? Based on those answers, you may modify the requirements, add your own controls, or focus more on specific areas.

That's fine - actually, we encourage it. Just document what you changed. If you modify a requirement, write down why and what you are doing to cover that security gap. Documentation keeps you honest when someone audits you later.

You'll probably add items that aren't in DASVS. This standard gives you a baseline. Your environment might have specific threats or needs that require additional measures.

How to Verify - Methodology

Comprehensive security verification uses multiple testing approaches. No single technique catches everything.

Dynamic Application Security Testing (DAST):

- Runs your application and interacts with it in real-time. You are finding runtime problems - authentication bypass, injection attacks, and memory corruption. Content that only appears when the code is executed. Think of it as testing from an attacker's perspective, because that's essentially what you are doing.

Static Application Security Testing (SAST):

- Examines code or binaries without executing anything. Good for spotting logic flaws, insecure data flows, and dangerous coding patterns that won't be obvious when you are just clicking through the application. The catch is that it can't tell you whether a problem is actually exploitable. It'll flag potential issues, but you still need to figure out if they matter in practice.

Interactive Application Security Testing (IAST):

- Combines both approaches. It instruments the application during runtime, which provides real-world behavior along with the detailed context you would get from static analysis. You end up with more accurate results and fewer false alarms.

Manual reviews are where experience counts. Automated tools often miss subtle problems - vulnerabilities hidden in business logic and flaws embedded in architectural decisions from the outset. Someone who knows what they are doing will spot these. Threat modeling gets you thinking like an attacker before you write code. Architecture reviews identify structural issues, while fixing them remains relatively inexpensive.

Here's why you need all of them: SAST might flag a possible SQL injection. DAST confirms it is exploitable. A manual review tells you why the vulnerability exists and how to actually fix it, not just slap a patch on it. One technique alone leaves gaps. Together, they give you reasonable coverage.

Documentation Requirements

Reasonable security verification needs good documentation. What you need depends on your verification level.

- **Level 1** applications require the basics, including functional specifications, high-level architecture diagrams, and user documentation. That a tester understands what the application does and how users interact with it is enough.

- **Level 2** is where documentation gets serious. You need detailed data flows that show where sensitive information actually travels through the application. Threat models that map out what could go wrong and how it could happen. Comprehensive design docs. Security guides for those who'll administer this system. At Level 2, a reviewer needs to understand more than just what your application does - they need to see how it makes security decisions and why.

- **Level 3** demands complete transparency. Full source code with security annotations. Formal security architecture documentation. Component inventories that track every library and dependency, including third-party risks. At this level, reviewers need to understand both the application and the developers who built it.

Whatever level you are at, certain things should always be documented clearly. Why did you make specific security decisions? What assumptions are you relying on? How does authentication work? What's your authorization model? Which crypto algorithms are you using, and how are you managing keys?

Documentation isn't just about passing verification. It facilitates ongoing assessments, enables knowledge transfer when team members change, and establishes a foundation for continuous security improvement. Plus, when something breaks six months from now, you'll be grateful you wrote this stuff down.

Guidelines

This is where DASVS gets specific. The guidelines section contains the actual security controls you need to implement and verify. Everything is organized into domains that cover various aspects of desktop application security, including architecture, authentication, data protection, and more.

How the Guidelines Work

Each domain begins with a control objective that explains what you are trying to accomplish and why it matters. Then come the individual requirements, all structured the same way:

- **Unique Identifier:** Each requirement has a unique ID in the format: V<**domain**>.<**section**>.<**requirement**> (like **V1.2.3**). This lets you reference specific requirements precisely in bug reports or security discussions.
- **Description:** Clear explanation of what the security control does, written in plain language you can actually understand.
- **Verification Level:** Shows whether the requirement applies to **L1**, **L2**, or **L3** applications based on your risk profile.
- **CWE Reference:** Mapping to relevant Common Weakness Enumeration identifiers

We designed these requirements to be technology-neutral - they work whether you are building in C++, Java, Python, or something else entirely. They are testable through various methods, so you are not locked into one verification approach. And they are specific enough that you can actually implement them without having to guess what we meant.

Matching Requirements to Risk

The three verification levels correspond to different risk profiles, as we described earlier.

- **Level 1** covers baseline security. Every desktop application should meet these requirements. They protect against common, well-documented attacks.
- **Level 2** is for applications handling sensitive business data. You are implementing stronger controls and being more thorough about security throughout the development process.

- **Level 3** assumes you are a target. These requirements are for mission-critical applications where a breach would be catastrophic.

Pick your level based on what you are protecting, what regulations you need to follow, and what happens if someone breaks in. When in doubt, go higher. Better to over-engineer security than explain to your CEO why customer data leaked.

List of all domains

The guidelines cover twelve security domains:

- **V1: Architecture and Design** - Get the foundation right from the start
- **V2: Authentication** - Make sure users are who they say they are
- **V3: Access Control** - Control what authenticated users can actually do
- **V4: Data Protection** - Keep sensitive information safe
- **V5: Communication** - Secure network traffic
- **V6: Validation and Output Encoding** - Handle untrusted input without getting burned
- **V7: File Operations** - Manage file system access securely
- **V8: Hardware Integration** - Deal with USB devices and other peripherals
- **V9: Logging and Monitoring** - Know when something's going wrong
- **V10: Installation and Updates** - Secure the distribution process
- **V11: Self-Protection** - Make your application harder to tamper with
- **V12: UI Security and User Privacy** - Protect the interface layer

Each domain addresses specific threats relevant to desktop applications. Together, they address the security concerns that truly matter when building desktop software.

V1: Architecture and Design

Control Objective

Security needs to be integrated into your architecture from the start, not added later. Desktop applications have unique threats - privilege boundaries that need defining, data that needs protecting, and update mechanisms that can become attack vectors. Get the architecture right before you write code, because fixing structural security problems, after the fact is expensive and painful. Think about layered defenses, how privileges escalate, where sensitive data flows, and how updates get distributed.

Security Verification Requirements

V1.1: Secure Development Lifecycle

#	Description	L1	L2	L3	CWE
1.1.1	Verify that a secure software development lifecycle is used throughout the development process, with security considerations at each phase from requirements to maintenance.		✓	✓	1053
1.1.2	Verify that threat modeling is performed during design, addressing potential threats and implementing appropriate countermeasures.		✓	✓	1053
1.1.3	Verify that security requirements are defined and validated alongside functional requirements.		✓	✓	1110
1.1.4	Verify that the desktop application's security architecture is documented, including trust boundaries, data flows, and security controls.		✓	✓	1059
1.1.5	Verify the implementation of a secure-by-default configuration, with security features enabled and unnecessary functionality disabled.	✓	✓	✓	1059

V1.2: Privilege and Trust Boundaries

#	Description	L1	L2	L3	CWE
1.2.1	Verify that the application follows the principle of least privilege, requesting only the minimum necessary operating system permissions to function.	✓	✓	✓	250
1.2.2	Verify that privilege elevation occurs only when necessary, with appropriate user notification and consent.	✓	✓	✓	250
1.2.3	Verify that the application identifies and enforces a separation between high and low-privilege operations, interfaces, and data.		✓	✓	272
1.2.4	Verify that security-sensitive operations require additional authentication or verification beyond the initial application authentication.		✓	✓	272
1.2.5	Verify that applications implement appropriate sandboxing and security mechanisms to restrict system resource access, following platform-specific best practices.	✓	✓	✓	272

V2: Authentication

Control Objective

Desktop applications often need to authenticate users without relying on network services. Maybe the user is offline. Maybe your app needs to work when the network is down. Either way, you need authentication that works locally.

This domain encompasses secure authentication for desktop environments, including credential storage that prevents password leaks, multi-factor authentication, biometric verification, keychain integration on platforms that support it, and handling authentication in the absence of a network connection. The goal is straightforward: protect user credentials from being compromised, whether the threat is a local attacker, malware, or someone who has stolen the laptop.

Security Verification Requirements

V2.1: Local Authentication Mechanisms

#	Description	L1	L2	L3	CWE
2.1.1	Verify that local authentication credentials are never stored in cleartext. Instead, use secure platform-approved storage mechanisms.	✓	✓	✓	256
2.1.2	Verify that password-based authentication uses secure password policies, including minimum length, complexity, and breach detection.	✓	✓	✓	521
2.1.3	Verify that authentication failures are handled securely, with appropriate timing, logging, and protection against brute force attacks.	✓	✓	✓	307
2.1.4	Verify that biometric authentication, if used, is implemented as a second factor only, with primary secrets not replaceable by biometric factors.		✓	✓	308
2.1.5	Verify that multi-factor authentication options are available and enforced for high-value operations.		✓	✓	308
2.1.6	Verify that applications utilize secure, platform-appropriate credential storage mechanisms to protect sensitive data.	✓	✓	✓	256

V2.2: Offline Authentication

#	Description	L1	L2	L3	CWE
2.2.1	Verify that offline authentication mechanisms maintain the same security posture as online mechanisms.	✓	✓	✓	306
2.2.2	Verify that offline account unlocking or recovery mechanisms are at least as strong as the primary authentication mechanism.	✓	✓	✓	640
2.2.3	Verify that any cached authentication data is protected with encryption and integrity checking to prevent offline attacks.		✓	✓	922
2.2.4	Verify that time-limited offline access is enforced when appropriate, requiring online re-authentication after the defined period.		✓	✓	613

V2.3: Integration with Platform Authentication

#	Description	L1	L2	L3	CWE
2.3.1	Verify that the application correctly implements the authentication workflow and security model when using OS-provided authentication services.	✓	✓	✓	287
2.3.2	Verify that integrated authentication mechanisms (e.g., Single Sign-On) maintain security context in a manner that cannot be bypassed.		✓	✓	287
2.3.3	Verify that applications correctly implement secure authentication mechanisms, including biometric or system-integrated authentication, with proper fallback options.	✓	✓	✓	287

V3: Access Control

Control Objective

Desktop applications run with serious system privileges. They are reading files from anywhere on the disk, switching between user accounts, and communicating directly with hardware. This creates obvious problems if you do not control what can access what.

You need authorization checks before your application touches system resources, executes data operations, or runs privileged functions. The goal here is straightforward - to prevent unauthorized access to sensitive data and functionality.

Here's where most developers get it wrong: they lock down the UI and think they are done. Great, users can't click the admin button. But what if someone calls that admin function directly? What if another process invokes it? Your access control cannot be limited to UI entry points. It needs to be in your business logic, in your data layer, anywhere that authorization actually matters. Otherwise, you are just hoping attackers won't notice the back door you left open.

Security Verification Requirements

V3.1: Resource Access Control

#	Description	L1	L2	L3	CWE
3.1.1	Verify that the application enforces access control rules on a trusted enforcement layer that cannot be bypassed.	✓	✓	✓	602
3.1.2	Verify that access to sensitive files, configuration, and resources is restricted based on the principle of least privilege.	✓	✓	✓	732
3.1.3	Verify that the application restricts access to local system resources (file system, registry, etc.) based on user roles and permissions.	✓	✓	✓	732
3.1.4	Verify that the application properly validates user permissions before accessing or modifying local resources.	✓	✓	✓	285
3.1.5	Verify that access control failures are logged, monitored, and responded to.		✓	✓	778
3.1.6	Verify that applications properly enforce file system permissions and access controls.	✓	✓	✓	732

V3.2: Operation Authorization

#	Description	L1	L2	L3	CWE
3.2.1	Verify that sensitive operations require additional verification or elevated permissions with proper user notification.	✓	✓	✓	862
3.2.2	Verify that the application implements role-based access control for different user types or permission levels.		✓	✓	862

3.2.3	Verify that administrative or high-privilege functions are clearly separated from normal user functions.	✓	✓	✓	272
3.2.4	Verify that the elevation of privilege workflows are secure and cannot be bypassed.		✓	✓	269
3.2.5	Verify that access control checks fail securely (closed by default).	✓	✓	✓	285

V3.3: Multi-User Environment Controls

#	Description	L1	L2	L3	CWE
3.3.1	Verify that the application properly isolates data and settings between different user accounts on the same system.	✓	✓	✓	668
3.3.2	Verify that the application does not leak sensitive information between local user sessions or accounts.	✓	✓	✓	522
3.3.3	Verify that shared resources used by multiple users implement proper access controls.		✓	✓	732
3.3.4	Verify that the application properly handles user context switching, ensuring complete session isolation between users.		✓	✓	287

V4: Data Protection

Control Objective

Desktop apps store sensitive data directly on the local file system, including configuration files, user credentials, cached content, and application secrets. Web apps typically store most of this information on secure servers. Desktop apps do not have that luxury. Everything sits locally, where an attacker with physical access can potentially grab it.

This domain covers encryption, secure storage, and proper data handling on the local system. Data needs protection whether it is in a file, loaded in memory, or being written to disk. Someone shouldn't be able to walk up to the machine, copy your application's data folder, and read everything in plain text.

Security Verification Requirements

V4.1: Local Data Encryption

#	Description	L1	L2	L3	CWE
4.1.1	Verify that sensitive data is encrypted when stored locally using platform-approved encryption APIs.	✓	✓	✓	311
4.1.2	Verify that encryption keys are securely managed and protected, not hardcoded or stored insecurely.	✓	✓	✓	798
4.1.3	Verify that industry-standard cryptographic algorithms and libraries are used for encryption.	✓	✓	✓	327
4.1.4	Verify that encryption keys used for data protection have appropriate key rotation and management processes.		✓	✓	320
4.1.5	Verify that the application implements secure wiping or destruction of sensitive data when it is no longer needed.		✓	✓	226
4.1.6	Verify that applications use appropriate encryption and data protection mechanisms to secure sensitive information.	✓	✓	✓	311

V4.2: Configuration and Settings Protection

#	Description	L1	L2	L3	CWE
4.2.1	Verify that application configuration files containing sensitive data are properly protected with appropriate permissions and encryption.	✓	✓	✓	732
4.2.2	Verify that application settings and preferences are stored securely with tamper protection.		✓	✓	642
4.2.3	Verify that the security configuration cannot be downgraded without appropriate authorization.		✓	✓	276
4.2.4	Verify that security-relevant configuration changes are logged.		✓	✓	778
4.2.5	Verify that configuration backups or exports properly protect sensitive information.		✓	✓	311

V4.3: Memory Protection

#	Description	L1	L2	L3	CWE
4.3.1	Verify that sensitive data is cleared from memory when no longer needed.		✓	✓	226
4.3.2	Verify that the memory containing sensitive data is properly protected against unauthorized access.		✓	✓	922
4.3.3	Verify that the application minimizes the exposure of sensitive data in memory dumps.		✓	✓	226
4.3.4	Verify that memory allocated for sensitive operations is protected from swapping to disk.			✓	591
4.3.5	Verify that the application uses secure memory allocation and management techniques for sensitive data processing.			✓	119

V5: Communication

Control Objective

Desktop applications communicate with remote servers, other local applications, and system services - sometimes all three. Each communication channel is a potential attack vector if not adequately secured.

This domain ensures secure communication across network connections and inter-process communication. The goal is to protect the confidentiality and integrity of data in transit, prevent interception, and ensure you are actually talking to the endpoint you think you are talking to. It doesn't matter if the data is crossing the internet or just jumping between processes on the same machine - it needs protection.

Security Verification Requirements

V5.1: Network Communication

#	Description	L1	L2	L3	CWE
5.1.1	Verify that all network communication uses encrypted protocols (TLS) with proper certificate validation.	✓	✓	✓	319
5.1.2	Verify that the application correctly validates TLS certificates and does not trust invalid or self-signed certificates.	✓	✓	✓	295
5.1.3	Verify the application uses certificate pinning for critical connections to prevent man-in-the-middle attacks.		✓	✓	295
5.1.4	Verify that the application handles certificate validation errors correctly, notifying users and failing securely.	✓	✓	✓	297
5.1.5	Verify that the application uses strong TLS configurations with modern cipher suites and proper key lengths.	✓	✓	✓	326
5.1.6	Verify that the application can operate securely in hostile network environments.		✓	✓	350

V5.2: Inter-Process Communication

#	Description	L1	L2	L3	CWE
5.2.1	Verify that inter-process communication (IPC) mechanisms implement proper access controls to prevent unauthorized access.	✓	✓	✓	923
5.2.2	Verify that IPC channels are authenticated and protected from tampering or interception.		✓	✓	923
5.2.3	Verify that the application validates all data received through IPC channels before processing.	✓	✓	✓	74
5.2.4	Verify that IPC mechanisms fail securely when communication errors occur.		✓	✓	636
5.2.5	Verify that applications securely implement interprocess communication with appropriate access controls and validation.	✓	✓	✓	923

V5.3: API Security

#	Description	L1	L2	L3	CWE
5.3.1	Verify that all API calls to remote services use proper authentication with securely managed credentials.	✓	✓	✓	287
5.3.2	Verify that API keys and secrets are protected at rest and not exposed in the application's code or resources.	✓	✓	✓	798
5.3.3	Verify that API requests and responses are validated for proper structure and content.	✓	✓	✓	20
5.3.4	Verify that the application handles API errors gracefully without exposing sensitive information.	✓	✓	✓	209
5.3.5	Verify that API communication uses transport encryption with proper certificate validation.	✓	✓	✓	319

V6: Input Validation and Output Encoding

Control Objective

Desktop applications accept input from files, inter-process communication, network sources, and user interfaces - anywhere data can come from. Trusting input without validation leads to injection attacks and compromised systems.

This domain covers proper input validation and output encoding to prevent injection attacks and other input-based vulnerabilities. The objective is to handle input in a way that prevents malicious data from compromising the application or underlying system. All input requires validation. Files may contain malicious payloads, IPC messages may be attack vectors, and user input cannot be trusted.

Security Verification Requirements

V6.1: Input Validation

#	Description	L1	L2	L3	CWE
6.1.1	Verify that the application validates all inputs from untrusted sources, including files, IPC, network sources, and user interfaces.	✓	✓	✓	20
6.1.2	Verify that input validation is enforced on a trusted service layer that cannot be bypassed.	✓	✓	✓	602
6.1.3	Verify that the application uses positive validation (allow listing) where possible, rather than rejecting known bad input (delisting).	✓	✓	✓	20
6.1.4	Verify that structured data is validated against a defined schema, including type, length, format, and range.	✓	✓	✓	20
6.1.5	Verify that user-supplied file names or paths are validated and sanitized to prevent path traversal attacks.	✓	✓	✓	22
6.1.6	Verify that the application correctly validates and handles data from external sources (like files or networks) before processing.	✓	✓	✓	74

V6.2: Command and Code Injection Prevention

#	Description	L1	L2	L3	CWE
6.2.1	Verify that the application does not use dangerous APIs like eval() or execute user-controlled code.	✓	✓	✓	95
6.2.2	Verify that OS command execution never occurs with user-supplied data without proper sanitization and parameterization.	✓	✓	✓	78
6.2.3	Verify that the application properly neutralizes command injection metacharacters from user-supplied input.	✓	✓	✓	77
6.2.4	Verify that contexts where data is executed are strictly separated from contexts where data is shown to users.		✓	✓	74
6.2.5	Verify that user input is never interpreted as code or markup without explicit validation and sanitization.	✓	✓	✓	94

6.2.6	Verify that the application correctly handles command execution and API calls to prevent code injection vulnerabilities.	✓	✓	✓	95

V6.3: Output Encoding and Content Security

#	Description	L1	L2	L3	CWE
6.3.1	Verify that output encoding is applied close to or by the interpreter for which it is intended.	✓	✓	✓	116
6.3.2	Verify that the data displayed to users has appropriate output encoding for the context.	✓	✓	✓	116
6.3.3	Verify that HTML content in desktop applications is properly sanitized to prevent XSS attacks.	✓	✓	✓	79
6.3.4	Verify the application implements appropriate controls for embedded web content using mechanisms like a Content Security Policy.		✓	✓	79
6.3.5	Verify that embedded browser components or web views implement proper security restrictions and isolation.	✓	✓	✓	610

V7: File Operations

Control Objective

File operations are where desktop applications differ most from web apps. Instead of reading and writing to a controlled database, desktop apps interact directly with the file system, often with broad permissions.

This domain ensures secure file handling - uploads, downloads, parsing, and execution. The objective is to prevent unauthorized file system access, stop malicious file execution, and avoid data leakage through insecure file operations. Applications must verify file contents rather than relying on extensions or MIME types.

Security Verification Requirements

V7.1: File Access and Permissions

#	Description	L1	L2	L3	CWE
7.1.1	Verify that the application only accesses files and directories within its intended permission scope.	✓	✓	✓	22
7.1.2	Verify that file operations are performed with the minimum necessary privileges.	✓	✓	✓	732
7.1.3	Verify that files created by the application have proper permissions that restrict access to authorized users only.	✓	✓	✓	732
7.1.4	Verify that user-supplied file paths are validated and normalized to prevent path traversal attacks.	✓	✓	✓	22
7.1.5	Verify that the application gracefully handles failures in file operations without exposing sensitive information.	✓	✓	✓	209
7.1.6	Verify that applications enforce proper file access controls and security mechanisms to prevent unauthorized data access.	✓	✓	✓	732

V7.2: File Upload and Processing

#	Description	L1	L2	L3	CWE
7.2.1	Verify that the application validates uploaded files for type, size, and content before processing.	✓	✓	✓	434
7.2.2	Verify that uploaded files are stored in a location that prevents executable content from being directly accessed by users.	✓	✓	✓	434
7.2.3	Verify that temporary files created during file processing are correctly secured and removed after use.		✓	✓	459
7.2.4	Verify that the application handles large files appropriately without exhausting system resources.		✓	✓	400

V7.3: File Content and Parsing

#	Description	L1	L2	L3	CWE
7.3.1	Verify that the application uses secure parsing for file formats that may contain malicious content (PDF, XML, ZIP, etc.).	✓	✓	✓	74
7.3.2	Verify that XML parsing is configured to prevent XXE (XML External Entity) attacks.	✓	✓	✓	611
7.3.3	Verify that deserialization of untrusted data is avoided or properly secured.	✓	✓	✓	502
7.3.4	Verify that file parsers are kept updated to address known vulnerabilities.	✓	✓	✓	937
7.3.5	Verify that parsers operate with the minimum necessary privileges and in a sandboxed environment when possible.		✓	✓	265

V8: Hardware Integration and External Device Security

Control Objective

Desktop applications interact with various hardware devices - USB drives, printers, cameras, biometric readers, and smart cards. Each device connection creates potential attack vectors. USB-based attacks have compromised air-gapped systems in real-world incidents.

This domain focuses on secure hardware integration: device trust, data exchange security, and peripheral access control. Malicious devices can compromise applications, and attackers can gain unauthorized access to sensitive hardware. USB devices shouldn't be trusted based on how they identify themselves.

Security Verification Requirements

V8.1: Device Access and Authentication

#	Description	L1	L2	L3	CWE
8.1.1	Verify that the application implements proper access controls for hardware resources and peripheral devices.	✓	✓	✓	732
8.1.2	Verify that device connections and disconnections are properly handled, maintaining a secure state.	✓	✓	✓	636
8.1.3	Verify that the application validates and authenticates devices before establishing trusted communication.		✓	✓	287
8.1.4	Verify that device drivers or libraries used for hardware integration are kept updated and securely configured.	✓	✓	✓	937
8.1.5	Verify that the application minimizes the attack surface exposed to potentially malicious devices.		✓	✓	653
8.1.6	Verify that the application securely requests, validates, and enforces permissions for hardware access according to platform-specific security best practices.	✓	✓	✓	272

V8.2: Data Exchange with External Devices

#	Description	L1	L2	L3	CWE
8.2.1	Verify that data exchanged with external devices is validated for integrity and authenticity.		✓	✓	354
8.2.2	Verify that sensitive data transmitted to external devices is encrypted when appropriate.	✓	✓	✓	311
8.2.3	Verify that the application does not expose sensitive information to unauthorized devices.	✓	✓	✓	200
8.2.4	Verify that the application implements safeguards against buffer overflows and other memory corruption vulnerabilities in device communication.		✓	✓	120

8.2.5	Verify that the application properly handles unexpected or malformed data from external devices.	✓	✓	✓	20

V8.3: Hardware Security Features

#	Description	L1	L2	L3	CWE
8.3.1	Verify that the application leverages hardware security features when available (TPM, secure enclaves, etc.).		✓	✓	653
8.3.2	Verify that cryptographic operations leverage hardware-backed key storage when available.		✓	✓	320
8.3.3	Verify that biometric authentication, when used, is implemented securely with appropriate liveness detection and fallback mechanisms.		✓	✓	308
8.3.4	Verify that the application validates the security posture of the hardware environment before executing sensitive operations.			✓	653
8.3.5	Verify that the application implements defense in depth and does not solely rely on hardware security features.		✓	✓	653

V9: Logging and Monitoring

Control Objective

Logs are needed for incident detection, forensic analysis, and compliance. However, desktop logging is challenging - balancing security visibility with privacy concerns and local resource constraints. Excessive logging fills disk space or exposes sensitive user data. Insufficient logging blinds incident response.

This domain covers secure logging practices for desktop applications. The objective is to record security-relevant events properly, protect those logs from tampering, and make them accessible for analysis without compromising user privacy or impacting system performance. Logs should capture what happened, when it happened, and who did it - but exclude passwords, personal data, or other sensitive information.

Security Verification Requirements

V9.1: Security Event Logging

#	Description	L1	L2	L3	CWE
9.1.1	Verify that the application logs security-relevant events, including authentication attempts, privilege changes, and access control failures.	✓	✓	✓	778
9.1.2	Verify that log entries include sufficient information for security analysis without capturing sensitive data or PII.	✓	✓	✓	532
9.1.3	Verify that log entries use a consistent time source and include timestamps in a standardized format.		✓	✓	778
9.1.4	Verify that logs are written using well-structured formats that can be easily parsed.		✓	✓	778
9.1.5	Verify that security logs capture both successful and failed security events.		✓	✓	778
9.1.6	Verify that the application uses appropriate system logging facilities with proper log levels and security best practices.	✓	✓	✓	778

V9.2: Log Protection and Management

#	Description	L1	L2	L3	CWE
9.2.1	Verify that access to log files is restricted to authorized personnel only.		✓	✓	732
9.2.2	Verify that application logs are protected from unauthorized modification and deletion.		✓	✓	778
9.2.3	Verify that the application implements appropriate log rotation or size limiting to prevent disk space exhaustion.	✓	✓	✓	770
9.2.4	Verify that logs containing sensitive information are encrypted when stored.		✓	✓	311

9.2.5	Verify that error-handling routines do not expose sensitive information in logs or to users.	✓	✓	✓	209

V9.3: Monitoring and Alerting

#	Description	L1	L2	L3	CWE
9.3.1	Verify that the application can integrate with enterprise monitoring systems or security information and event management (SIEM) tools.		✓	✓	778
9.3.2	Verify that the application can generate alerts for critical security events requiring immediate attention.		✓	✓	778
9.3.3	Verify that the application monitors and alerts on suspicious activity patterns, such as repeated authentication failures.		✓	✓	778
9.3.4	Verify that logging and monitoring functions fail securely and do not impact application functionality.		✓	✓	778
9.3.5	Verify that the application includes self-monitoring capabilities for critical security controls.			✓	778

V10: Installation and Update Security

Control Objective

Installation and updates are critical attack windows. If someone can compromise your installer or update mechanism, they can distribute malware to every user of your application. This is why supply chain attacks targeting update systems are so popular with attackers.

This domain ensures applications are installed, updated, and removed securely with integrity and authenticity verification at each stage. The objective is to prevent unauthorized modifications during deployment and updates while maintaining a secure and auditable installation environment. Your update mechanism needs to verify it is actually pulling updates from you, not from an attacker doing a man-in-the-middle attack.

Security Verification Requirements

V10.1: Secure Installation

#	Description	L1	L2	L3	CWE
10.1.1	Verify that installation packages are digitally signed and verified before execution.	✓	✓	✓	353
10.1.2	Verify that the installation process runs with the minimum necessary privileges, elevating only when required.	✓	✓	✓	250
10.1.3	Verify that installation routines perform integrity checks on critical files and components.		✓	✓	354
10.1.4	Verify that the installation processes validate that the target system meets minimum security requirements.		✓	✓	1391
10.1.5	Verify that installation logs capture sufficient information for security auditing and troubleshooting.		✓	✓	778
10.1.6	Verify the application follows secure installation and distribution practices, including proper code signing and platform-specific security requirements.	✓	✓	✓	353
10.1.8	Verify that installation routines properly handle errors and do not leave the system insecure if they fail.	✓	✓	✓	636
10.1.9	Verify that applications implement secure code-signing and distribution mechanisms appropriate to their platform to ensure integrity and trust.	✓	✓	✓	353

V10.2: Update Security

#	Description	L1	L2	L3	CWE
10.2.1	Verify that updates are downloaded over secure channels with proper TLS configuration.	✓	✓	✓	319
10.2.2	Verify that update packages are cryptographically signed and verified before installation.	✓	✓	✓	353

10.2.3	Verify that the update process validates the integrity of all updated files.	✓	✓	✓	354
10.2.4	Verify that the update mechanism cannot be used as an attack vector to install malicious code.		✓	✓	494
10.2.5	Verify that update processes maintain secure configurations and do not reset security settings.		✓	✓	16
10.2.6	Verify that users are informed about security-relevant changes in updates.	✓	✓	✓	656
10.2.7	Verify that update processes run with appropriate (ideally unprivileged) permissions, with privilege elevation only when necessary for installation.		✓	✓	250

V10.3: Uninstallation and Cleanup

#	Description	L1	L2	L3	CWE
10.3.1	Verify that the application can be completely and cleanly uninstalled, removing all components and data.	✓	✓	✓	459
10.3.2	Verify that uninstallation routines securely remove or destroy sensitive data and credentials.		✓	✓	312
10.3.3	Verify that uninstallation does not leave the system in an insecure state.	✓	✓	✓	459
10.3.4	Verify that administrator privileges are only requested when necessary during uninstallation.	✓	✓	✓	250
10.3.5	Verify that shared components are handled properly during uninstallation to prevent breaking other applications.		✓	✓	459

V11: Self-Protection and Integrity

Control Objective

Desktop applications face threats that web apps rarely see - attackers with direct access to the binary, memory, and execution environment. They can attach debuggers, modify memory at runtime, patch the executable, and reverse engineer the code. This is the reality of desktop software.

This domain encompasses self-protection mechanisms for detecting and responding to tampering, debugging, and runtime attacks. The objective is to maintain application integrity throughout execution and to resist reverse engineering and modification attempts. While determined attackers may eventually succeed, these controls raise the bar and provide detection capabilities

Security Verification Requirements

V11.1: Binary and Runtime Protection

#	Description	L1	L2	L3	CWE
11.1.1	Verify that the application implements runtime integrity checks for critical code and data.		✓	✓	693
11.1.2	Verify that the application detects and responds to debugging attempts when handling sensitive operations.		✓	✓	693
11.1.3	Verify that the application implements controls to resist reverse engineering, appropriate to the identified risk.		✓	✓	494
11.1.4	Verify that the application detects and responds to tampering with its executable code, configuration, or data files.	✓	✓	✓	693
11.1.5	Verify that the application appropriately obfuscates sensitive logic or data to increase the difficulty of reverse engineering.			✓	693
11.1.6	Verify that applications implement appropriate memory protection features such as ASLR, NX, and CFG.	✓	✓	✓	693
11.1.7	Verify that applications implement appropriate hardening flags during compilation and linking.	✓	✓	✓	693
11.1.8	Verify that applications implement code signing and hardening measures.		✓	✓	353

V11.2: Anti-Tampering Controls

#	Description	L1	L2	L3	CWE
11.2.1	Verify that the application implements checksums or digital signatures to verify the integrity of critical components.		✓	✓	354

11.2.2	Verify that integrity verification mechanisms are applied to application dependencies and libraries.		✓	✓	829
11.2.3	Verify that the application implements environment validation to detect virtualized or debugging environments when processing sensitive data.			✓	693
11.2.4	Verify that the application responds to integrity violations in a manner that does not expose the exact protection mechanism.		✓	✓	693
11.2.5	Verify that anti-tampering controls are tested and effective against known analysis and bypass techniques.			✓	693

V11.3: Secret Protection

#	Description	L1	L2	L3	CWE
11.3.1	Verify that sensitive constants, credentials, or cryptographic keys are not hardcoded in the application binary.	✓	✓	✓	798
11.3.2	Verify that cryptographic keys and secrets are protected in memory against local attacks.		✓	✓	316
11.3.3	Verify that the application utilizes platform-provided secure key storage when available.	✓	✓	✓	316
11.3.4	Verify that secrets are not stored in insecure locations like the registry, preferences files, or standard directories.	✓	✓	✓	922
11.3.5	Verify that the application implements secret splitting or other advanced techniques to protect high-value secrets.			✓	798

V12: UI Security and User Privacy

Control Objective

UI implementations commonly leak sensitive data. Password fields that display text instead of masking it. Sensitive information appearing in window titles captured by screen-sharing software. Clipboard operations that expose secrets to other applications.

This domain covers secure UI implementation, preventing data leakage through the interface. It also addresses UI-based attacks, such as clickjacking. Sensitive data should only be displayed when necessary. Consider exposure risks from screen sharing and shoulder surfing.

Security Verification Requirements

V12.1: Input Handling in User Interface

#	Description	L1	L2	L3	CWE
12.1.1	Verify that UI components validate and sanitize all user input before processing.	✓	✓	✓	20
12.1.2	Verify that the application prevents UI-based injection attacks, including script injection in text fields.	✓	✓	✓	74
12.1.3	Verify that the UI framework is kept updated to address known security vulnerabilities.	✓	✓	✓	937
12.1.4	Verify that the application properly handles unexpected or malformed input in UI components.	✓	✓	✓	20
12.1.5	Verify that UI automation and accessibility features cannot be exploited for unauthorized actions.		✓	✓	610
12.1.6	Verify that applications properly validate data input and storage in user interface frameworks.	✓	✓	✓	20

V12.2: Protection of Sensitive Data in UI

#	Description	L1	L2	L3	CWE
12.2.1	Verify that sensitive information, such as passwords or PINs, is masked in UI components when displayed.	✓	✓	✓	319
12.2.2	Verify that screenshots or screen recordings are prevented or obscured when displaying sensitive information.		✓	✓	200
12.2.3	Verify that copy-paste operations are disabled or controlled for fields containing sensitive data.		✓	✓	200
12.2.4	Verify that the clipboard contents containing sensitive data are cleared after a defined period.		✓	✓	200

12.2.5	Verify that the application UI does not expose sensitive information in window titles, dialog box messages, or error notifications.	✓	✓	✓	209

V12.3: UI-Based Privacy Controls

#	Description	L1	L2	L3	CWE
12.3.1	Verify that the application implements privacy indicators when accessing sensitive hardware like cameras or microphones.	✓	✓	✓	200
12.3.2	Verify that the application provides clear user notifications and consent options before collecting personal data.	✓	✓	✓	200
12.3.3	Verify that privacy-relevant settings are persistent and cannot be silently changed.	✓	✓	✓	285
12.3.4	Verify that the application allows users to export and delete their personal data.		✓	✓	200
12.3.5	Verify that privacy controls are consistent with platform expectations and guidelines.	✓	✓	✓	200

References

- OWASP Application Security Verification Standard (ASVS), OWASP Foundation
- OWASP Mobile Application Security Verification Standard (MASVS), OWASP Foundation
- NIST Cybersecurity Framework (CSF), National Institute of Standards and Technology
- CIS Controls & Benchmarks, Center for Internet Security
- MITRE ATT&CK® Framework, The MITRE Corporation
- afine.com