



CASVS

APPLICATION SECURITY
VERIFICATION STANDARD

What's New in ASVS v5

Eden Yardeni - SecAppDev 2026

Agenda

Intros (of ASVS and of myself)

01

3 Key Refactors

02

A Curated Selection of New Requirements

03

Demo: Code Review

04

Conclusion / Q&A

05



Introductions

OF ASVS AND OF MYSELF

Eden Yardeni

- Worked as a full-time software engineer for 12 years
 - 5 of those on AppSec teams, one of which I founded
- Joined the OWASP ASVS working group in October 2024
- Also started the OWASP chapter for Cleveland, Ohio in 2024
 - But then moved to London shortly thereafter (the Cleveland chapter is in great hands though)
- I've been **using** the ASVS since 2017—we will get to that!



The Application Security Verification Standard

- A curated set of **positive** requirements that govern **properties** of web application and API code
- Can be used as:
 - a checklist for secure coding, including writing the code and code reviews
 - a reference for penetration testing
 - secure architecture guidance
 - the basis for unit / integration tests
 - topics for developer training
 - a baseline for procurement decisions

The Application Security Verification Standard

- One of OWASP's flagship projects
- Developed in the open by contributors like you and me <3
- Designed to be:
 - Actionable
 - Forkable
 - Flexible



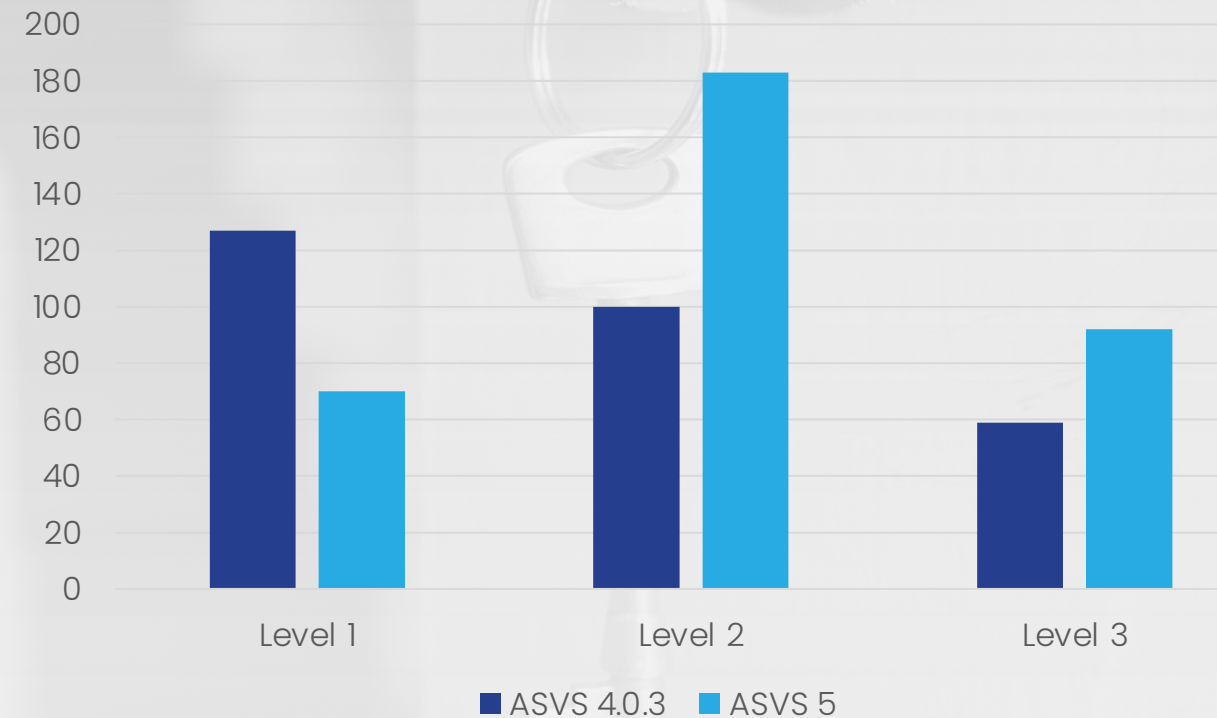
Three Key Refactors

STRUCTURAL CHANGES

Refactor #1 – Easier Level Structure

- Rebalanced levels to make getting started easier
- Levels are now relative / risk based and no longer represent “minimum” / “standard” / “advanced” baselines
- Recognizing that pentests should cover code too

Requirement Distribution Across Levels

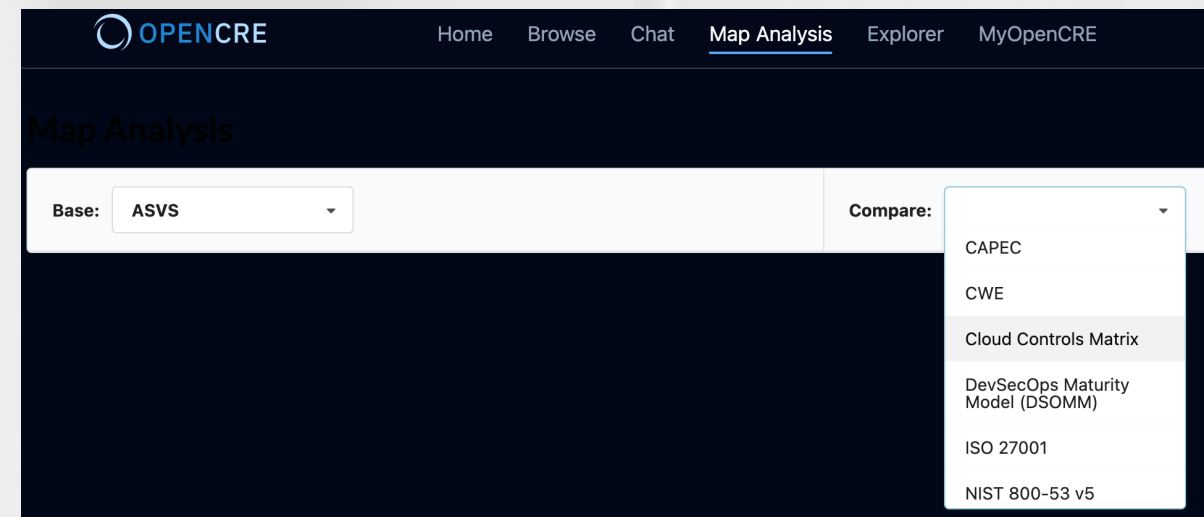


Refactor #2 – Properties over Process

- Architectural requirements were dropped to focus on properties of the **application**
- Is an application specifically less **secure** without the requirement being satisfied?
- Refocusing on **verifiable** goals that we believe are musts vs. nice-to-haves
- Written like an enforceable **standard**

Refactor #3 – Clean Mappings

- Dropped direct mappings to CWE, NIST in favor of OpenCRE
- Ongoing work to create OpenCRE mappings
- CWE as an abstract taxonomy did not always have clean mappings
- Did I mention OpenCRE yet? Go check that out...
- NIST is US-specific and mapping to non-US standards is becoming increasingly common





New chapters in 5.0

A CURATED SELECTION, BECAUSE THERE ARE A LOT

Encoding and Sanitization (V1)

- V1.2.8: LaTeX Processors (Lv. 2)
- V1.2.10: CSV and Formula Injection (Lv. 3)
- V1.3.8: JNDI query protection (Lv. 2)
- V1.3.9: Memcache injection (Lv. 2)
- V1.3.12: Runaway Regex / ReDoS mitigation (Lv. 3)
- V.1.4.3: Use-after-free protection (Lv. 2)



Web Frontend Security (V3)

- V3.2.3: DOM Clobbering (Lv. 3)
- V3.3.5: Cookie Length Limitations (Lv. 3)
- V3.5: Browser Origin Separation
- New header requirements including Cross-Origin-Opener-Policy, Cross-Origin-Resource-Policy, Sec-Fetch-*

Self-contained Tokens (V9)

- V9.1.2: Disallow 'None' algorithm for JWTs (Lv. 1)
- V9.2.1: Enforce expiration time for stateless tokens (Lv.1)



Demoing big new sections in 5.0

APPLYING REQUIREMENTS TO CODE REVIEW

Updated Requirements – OAuth (formerly V3.5)

V3.5 Token-based Session Management

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

#	Description	L1	L2	L3	CWE	NIST §
3.5.1	Verify the application allows users to revoke OAuth tokens that form trust relationships with linked applications.		✓	✓	290	7.1.2
3.5.2	Verify the application uses session tokens rather than static API secrets and keys, except with legacy implementations.		✓	✓	798	
3.5.3	Verify that stateless session tokens use digital signatures, encryption, and other countermeasures to protect against tampering, enveloping, replay, null cipher, and key substitution attacks.		✓	✓	345	

Updated Requirements – OAuth (now v10)

- V10.1: Generic OAuth and OIDC Security
- V10.2: OAuth Client
- V10.3: OAuth Resource Server
- V10.4: OAuth Authorization Server
- V10.5: OIDC Client
- V10.6: OpenID Provider
- V10.7: Consent Management


```

@Bean
JwtDecoder jwtDecoder() {
    NimbusJwtDecoder decoder = NimbusJwtDecoder.withJwkSetUri("https://login.example.com/.well-known/jwks.json").build();

    JwtTimestampValidator time = new JwtTimestampValidator();
    time.setClockSkew(java.time.Duration.ofSeconds(60));

    OAuth2TokenValidator<Jwt> issuer = jwt -> {
        String iss = jwt.getIssuer() != null ? jwt.getIssuer().toString() : "";
        boolean ok = iss.equals("https://login.example.com/");
        return ok
            ? OAuth2TokenValidatorResult.success()
            : OAuth2TokenValidatorResult.failure(new OAuth2Error("invalid_token", "Untrusted issuer", null));
    };

    OAuth2TokenValidator<Jwt> alg = jwt -> {
        String algorithm = jwt.getHeaders().getAlgorithm() != null ? jwt.getHeaders().getAlgorithm().getName() : "";
        boolean ok = algorithm.equals("RS256");
        return ok
            ? OAuth2TokenValidatorResult.success()
            : OAuth2TokenValidatorResult.failure(new OAuth2Error("invalid_token", "Unexpected signing algorithm", null));
    };

    decoder.setJwtValidator(new DelegatingOAuth2TokenValidator<>(time, issuer, alg));
    return decoder;
}

```

V10.3: OAuth Resource Server

#	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

```

@Bean
JwtDecoder jwtDecoder() {

    ...
    ...
    ...

    OAuth2TokenValidator<Jwt> audience = jwt -> {
        List<String> aud = jwt.getAudience();
        boolean ok = aud != null && aud.contains("api://inventory");
        return ok
            ? OAuth2TokenValidatorResult.success()
            : OAuth2TokenValidatorResult.failure(new OAuth2Error("invalid_token", "Missing required
              audience", null));
    };

    decoder.setJwtValidator(new DelegatingOAuth2TokenValidator<>(time, issuer, alg));
    return decoder;
}

```

Updated Requirements – Race conditions (formerly V11.1.6)

11.1.6	Verify that the application does not suffer from "Time Of Check to Time Of Use" (TOCTOU) issues or other race conditions for sensitive operations.	✓	✓	367
---------------	--	---	---	-----

Updated Requirements – Race conditions (now V15.4)

#	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

A sample OTP service in Go

Stateful one-time passwords. The server generates the code, stores it for a short validity window, and verifies the submission. Picture an SMS or email second factor.

WHAT IT DOES

- Issue a six-digit code, deliver it, verify the submission.
- Stateful: the server picks and stores the code, unlike TOTP.
- Three parts: a challenge store, a rate limiter, a validator.
- Service is a singleton instance that restricts users to five OTP challenge attempts.

```
type Service struct {  
    store ChallengeStore // active codes  
    limiter RateLimiter   // per-user caps  
    clock func() time.Time  
}  
  
func (s *Service) Issue(  
    user string) (Code, error)  
  
func (s *Service) Verify(  
    user, guess string) error
```

Goroutines and channels make concurrency cheap to write, but do not make it automatically safe, as we'll see.

Threat model: defend or document

Before any code, sort every risk into one of two buckets.

ACCEPTED

Documented risks that the business accepts the tradeoff of allowing these risks in production.

SMS bombing

Denial of Service exhausts SMS budget, spams the user

SIM swapping

Spoofing attacker controls the delivery channel

SS7 interception

Information Disclosure code exposed in transit

Email account compromise

Spoofing attacker receives the codes

Service restart wipes state

Tampering counter resets out of band

Multi-replica divergence

Tampering guesses spread across replicas

OWNED

Addressed in this service as it ships.

Online brute force

Spoofing guessing the second factor

Weak code generation

Spoofing predictable codes, no observation needed

Timing attack on comparison

Information Disclosure latency leaks correct prefixes

Replay of consumed codes

Spoofing · EoP an already-used code reused

Cross-user confusion

Spoofing · EoP one user's code accepted for another

Our limiter implementation

```
type Limiter struct {
    attempts map[string]*store.Entry
    ttl       time.Duration
    now       func() time.Time
}

func (l *Limiter) RecordFailure(_ context.Context, key string, limit
int,
) error {
    e := store.Ensure(l.attempts, key, l.ttl, l.now())
    if e.Count >= limit {
        return ratelimit.ErrLimitExceeded
    }
    e.Count++
    return nil
}
```

47 of 50

admitted against a limit of five

go test -race

8 data races

every one inside RecordFailure

Read, check, act, store: four operations that have to be atomic.

15.4.1 · Shared objects

Make the data structure safe

```
type Limiter struct {
    attempts sync.Map // map[string]store.Entry
    ttl      time.Duration
    now      func() time.Time
}

func (l *Limiter) RecordFailure(
    _ context.Context, key string, limit int,
) error {
    now := l.now()
    e, ok := l.load(key, now) // load returns a copy
    if !ok {
        e = store.Entry{ExpiresAt: now.Add(l.ttl)}
    }
    if e.Count >= limit {
        return ratelimit.ErrLimitExceeded
    }
    e.Count++ // increment the copy
    l.attempts.Store(key, e) // write the copy back
    return nil
}
```

Verify that shared objects in multi-threaded code are accessed safely using thread-safe types and synchronization mechanisms like locks or semaphores, to avoid race conditions and data corruption.

Go's race detector doesn't find anything.

The race detector stays silent and the test passes. `sync.Map` suits the workload: disjoint keys, one per user.

15.4.2 · Atomic compound ops

```
type Limiter struct {
    mu      sync.Mutex
    attempts map[string]*store.Entry
    ttl      time.Duration
    now      func() time.Time
}

func (l *Limiter) RecordFailure(
    _ context.Context, key string, limit int,
) error {
    l.mu.Lock()
    defer l.mu.Unlock()

    e := store.Ensure(l.attempts, key, l.ttl, l.now())
    if e.Count >= limit {           // check and act
        return ratelimit.ErrLimitExceeded
    }
    e.Count++                      // both under the lock
    return nil
}
```

Verify that checks on a resource's state 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.



Conclusion

CALLS TO ACTION

Other new sections worth noting

- V4.4: WebSockets
- V17: WebRTC
- Appendix C: Cryptography
- Appendix D: Recommendations

Want a full list of what's new?

- <https://asvs.dev/Mappings/>
- <https://asvs.dev/v5.0.0/For-Users-Of-4.0/>

Related OWASP Projects You Should Also Care About

Awareness documents

- OWASP Top 10
- Top 10 Proactive Controls

Requirements for other domains

- MASVS
- ISVS
- SPVS
- AISVS

Testing guides

- WSTG
- MSTG

Other guidance documents

- Cheat Sheet Series
- SAMM

Mappings

- OpenCRE

OWASP Project Wayfinder

- <https://owasp.org/projects/>

It's the LLMs' world and we're all just living in it

- Rigor is even more important for vibe coders!
- Even your favorite bot du jour needs standards to follow
- The faster we build, the more we need guardrails
- Keep an eye out for projects aiming to integrate ASVS with agents, MCP servers etc.



Thanks :)
Any Questions?

Eden Yardeni

✉ eden.yardeni@owasp.org

✉ eden@cronchie.dev



@cronchie

ASVS Project



@OWASP_ASVS



<https://github.com/OWASP/ASVS>



<https://owasp.slack.com>, #project-asvs



<https://owasp.org/asvs>
<https://asvs.dev>

