HTTP
Internet-Draft
Intended status: Standards Track
Expires: 27 October 2026

K. Rampalli
Glyphzero Labs Inc.

25 April 2026

SURADAR: Context-Bound Per-Request Authentication for Machine-to-Machine APIs

draft-rampalli-suradar-00

Abstract

This document defines SURADAR (Subsurface Undertow RADAR), an HTTP authentication scheme in which each request is authenticated by a one-time HMAC tag derived from a shared seed, the current time band, and the full request context (method, path, organisation, scope, and body). Unlike bearer-token schemes, a captured SURADAR token is cryptographically bound to exactly one request and cannot be reused, replayed, or re-scoped. The protocol requires zero per-request handshakes, produces 48-byte tokens, and relies solely on HMAC-SHA-256 and SHA-256 -- no asymmetric cryptography.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 27 October 2026.

Copyright Notice

Table of Contents

1. Introduction

Existing HTTP authentication mechanisms -- Bearer tokens [RFC6750], JSON Web Tokens [RFC7519], OAuth 2.0 [RFC6749], and HTTP Message Signatures [RFC9421] -- authenticate the caller but not the request. A valid token for one endpoint is equally valid for any other endpoint within its scope, for any request body, until the token expires. This creates a class of attacks where a stolen or intercepted token can be reused for unintended purposes.

SURADAR addresses this by deriving a unique one-time key for each request from:

* A shared secret seed (established at enrollment)

* The current time band (30-second window)

* A context fingerprint: SHA-256(method || path || orgID || scope)

* A random client nonce

* The request body

The resulting token is valid for exactly one HTTP request, at one endpoint, with one scope, within one time window, with exactly the body that was sent.

1.1. Goals

Zero per-request handshakes Token generation is entirely local; the client never contacts the server before sending the authenticated request.

Context binding The HTTP method, path, organisation identifier, and scope are cryptographically embedded in the key derivation, not merely carried as metadata.

Body integrity The full request body is covered by the HMAC, preventing body-swap attacks.

Minimal blast radius A captured token is valid for exactly one request; it cannot be replayed, re-scoped, or used against a different endpoint.

Symmetric simplicity The protocol uses only HMAC-SHA-256 and SHA-256. No asymmetric cryptography is required.

1.2. Non-Goals

Key distribution Seed enrollment is specified at the minimum level required for interoperability (Section 4). Full key lifecycle management is out of scope.

Authorization SURADAR authenticates requests; authorization policy is left to the application.

Transport security TLS MUST still be used. SURADAR provides authentication and integrity within the application layer, not confidentiality.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2.1. Notation

Symbol	Definition
S	Shared seed (32 bytes)
RSK	Root Server Key (32 bytes)
T	Time band index: floor(unix_seconds / TBandSeconds)
TBandSeconds	Time band width (default: 30)
TBandSkew	Adjacent bands accepted (default: 1)
ctx	Context fingerprint: SHA-256(method || 0x00 || path || 0x00 || orgID || 0x00 || scope)
K1	Partial key: HMAC-SHA-256(S, T_bytes || ctx)
R	Client nonce: 16 random bytes
K	Full one-time key: HMAC-SHA-256(K1, R)
sig	Request authenticator: HMAC-SHA-256(K, body)
token	Wire encoding: base64url(R || sig) -- 64 chars, 48 bytes
||	Byte concatenation
0x00	Null byte separator

Table 1: SURADAR Notation

2.2. Algorithms

SURADAR uses HMAC-SHA-256 [RFC2104] [RFC6234] for key derivation and request authentication, and SHA-256 [FIPS180-4] for context fingerprinting.

3. Protocol Overview

3.1. Enrollment Phase (Once Per Client)

The enrollment phase establishes a shared seed S between client and server. It occurs once per client over an authenticated channel.

```
Client                                        Server
  |--- [authenticated channel] ------------>|
  |    clientID, orgID, pubKey               |
  |    enrollNonce = rand(16)                |
  |    S = HMAC-SHA-256(RSK, clientID || enrollNonce)
  |    encrypted_S = XOR(S, SHA-256(pubKey))|
  |<-- enrollNonce, encrypted_S ------------|
  |    S = XOR(encrypted_S, SHA-256(privKey))
  |    Store S in secure keychain            |
  Server stores: (clientID, orgID, enrollNonce)
  Server NEVER stores S
```

3.2. Per-Request Phase (Zero Roundtrips)

Each HTTP request is independently authenticated with no additional roundtrips.

```
Client                                        Server
  | T = floor(now / 30)                      |
  | ctx = SHA-256(method||0x00||path||0x00||orgID||0x00||scope)
  | K1 = HMAC-SHA-256(S, T_bytes || ctx)     |
  | R = rand(16)                             |
  | K = HMAC-SHA-256(K1, R)                  |
  | sig = HMAC-SHA-256(K, body)              |
  | zero(K1, K)                              |
  | token = base64url(R || sig)              |
  |--- HTTP request with headers: --------->|
  |    X-SURADAR-Auth: <token>               |
  |    X-SURADAR-Client: <clientID>          |
  |    X-SURADAR-TBand: <T>                  |
  |    S' = HMAC-SHA-256(RSK, clientID || enrollNonce)
  |    ctx' = SHA-256(method||0x00||path||0x00||orgID||0x00||scope)
  |    K1' = HMAC-SHA-256(S', T_bytes || ctx')
  |    K' = HMAC-SHA-256(K1', R_from_token)  |
  |    verify: sig == HMAC-SHA-256(K', body)|
  |    replay_check(T, ctx, R)               |
  |    zero(K1', K')                         |
  |<-- 200 OK or 401 Unauthorized ---------|
```

4. Enrollment

4.1. Prerequisites

The client MUST authenticate via an existing mechanism (OAuth 2.0, mutual TLS, or a one-time enrollment token) before requesting seed enrollment. The enrollment channel MUST be protected by TLS.

4.2. Seed Derivation

The server generates a 16-byte enrollment nonce using a cryptographically secure pseudorandom number generator (CSPRNG) [RFC4086]:

```
enrollNonce = CSPRNG(16)
S = HMAC-SHA-256(RSK, clientID || enrollNonce)
```

The RSK MUST be stored in a hardware security module (HSM) or equivalent secure storage. The server MUST NOT store S. The server stores the tuple (clientID, orgID, enrollNonce).

4.3. Seed Delivery

The server delivers the seed to the client encrypted under the client’s public key:

```
encrypted_S = XOR(S, SHA-256(clientPubKey))
```

The client recovers S:

```
S = XOR(encrypted_S, SHA-256(privKey))
```

Alternative delivery mechanisms (e.g., direct TLS-protected transport, envelope encryption) are permitted provided confidentiality and integrity of S are maintained.

4.4. Seed Storage (Client)

The client MUST store S in one of the following:

* Operating system keychain (e.g., macOS Keychain, Windows Credential Manager)
* Hardware security element or TPM
* Secrets manager (e.g., HashiCorp Vault, AWS Secrets Manager)

The client MUST NOT store S in plaintext on disk or in environment variables.

4.5. RSK Management

The RSK MUST be stored in an HSM or secrets manager. RSK rotation is supported: the server MUST accept tokens derived from the previous RSK for a grace period equal to NonceTTL (default: 90 seconds) after rotation. During this period, both old and new RSK values are tried during verification.

5. Token Generation (Client)

5.1. Compute Time Band

The client computes the time band index:

T = floor(unix_seconds / TBandSeconds)

T is encoded as an 8-byte big-endian unsigned integer (T_bytes).

5.2. Compute Context Fingerprint

The context fingerprint binds the token to the specific request:

ctx = SHA-256(method || 0x00 || path || 0x00 || orgID || 0x00 || scope)

Null byte (0x00) separators prevent field boundary ambiguity. For example, without separators, method="GETX" path="Y" and method="GET" path="XY" would produce the same hash input.

5.3. Derive Partial Key

K1 = HMAC-SHA-256(S, T_bytes || ctx)

K1 MUST be zeroed from memory immediately after K is derived.

5.4. Generate Client Nonce

The client generates a 16-byte random nonce:

R = CSPRNG(16)

5.5. Derive Full Key

K = HMAC-SHA-256(K1, R)

K MUST be zeroed from memory immediately after the signature is computed.

5.6. Compute Signature

sig = HMAC-SHA-256(K, body)

For requests with no body (e.g., GET), body is the empty byte string. K MUST be zeroed after this step.

5.7. Encode Token

token = base64url_no_pad(R || sig)

R is 16 bytes and sig is 32 bytes, producing 48 bytes total, which encodes to exactly 64 base64url characters (no padding) per Section 5 of [RFC4648].

5.8. Set HTTP Headers

The client sets the following HTTP headers on the request:

Header	Value
X-SURADAR-Auth	token (64 base64url characters)
X-SURADAR-Client	clientID
X-SURADAR-TBand	T (decimal integer)

Table 2: SURADAR Request Headers

6. Token Verification (Server)

6.1. Parse Headers

The server extracts X-SURADAR-Auth, X-SURADAR-Client, and X-SURADAR-TBand from the request. If any header is absent, the server MUST respond with HTTP 401 Unauthorized.

6.2. Validate Time Band

The server computes T_server = floor(now / TBandSeconds) and checks:

|T_client - T_server| <= TBandSkew

If the time band is outside the acceptable skew window, the server MUST reject the request with HTTP 401.

6.3. Re-derive Seed

The server looks up enrollNonce for the given clientID, then re-derives the seed:

S′ = HMAC-SHA-256(RSK, clientID || enrollNonce)

During RSK rotation, the server MUST attempt verification with both the current and previous RSK values.

6.4. Recompute Key Chain

The server recomputes the context fingerprint and key chain from the actual HTTP request (method, path, orgID looked up from the client record, scope from application context):

```
ctx′ = SHA-256(method || 0x00 || path || 0x00 || orgID || 0x00 || scope)
K1′  = HMAC-SHA-256(S′, T_bytes || ctx′)
K′   = HMAC-SHA-256(K1′, R_from_token)
```

The server MUST NOT use any client-supplied values for method, path, or orgID in this computation.

6.5. Verify Signature

```
expected = HMAC-SHA-256(K′, body)
valid    = constant_time_equal(sig_from_token, expected)
```

The comparison MUST use a constant-time equality function to prevent timing side-channel attacks. K′ MUST be zeroed after use.

6.6. Check Replay

Replay checking MUST occur after HMAC verification succeeds. This ordering prevents an attacker from poisoning the nonce store with garbage tuples.

The server checks the tuple (T, ctx, R) against the nonce store. If the tuple is already present, the request is a replay and MUST be rejected with HTTP 401.

The nonce store entry TTL MUST be at least (TBandSkew + 1) * TBandSeconds seconds. With defaults, this is (1 + 1) * 30 = 60 seconds. A default of 90 seconds is RECOMMENDED to account for clock drift.

6.7. Return Principal

Upon successful verification, the server makes the following values available to downstream authorization logic:

* clientID
* orgID
* scope

7. Replay Prevention

7.1. Nonce Store Requirements

The nonce store provides a single operation:

Check(T, ctx, R) -> OK | REPLAY

The store MUST be concurrent-safe. Entries MUST expire after NonceTTL seconds (default: 90).

7.2. Bloom Filter Implementation

A Bloom filter implementation is RECOMMENDED for single-server deployments. The following parameters provide approximately 0.01% false positive rate:

* m = 10,000,000 bits (~1.2 MB)
* k = 7 hash functions

The implementation uses dual-rotating filters: the active filter receives new entries, while the previous filter is kept for the duration of NonceTTL before being cleared and rotated.

Hash functions for the Bloom filter are derived from SHA-256 of the (T, ctx, R) tuple by splitting the 256-bit output into segments.

7.3. Distributed Implementation

For distributed deployments, a Redis SETNX with TTL equal to NonceTTL is RECOMMENDED:

```
key = "suradar:nonce:" || hex(SHA-256(T || ctx || R))
result = SETNX(key, 1)
EXPIRE(key, NonceTTL)
```

If SETNX returns 0 (key already exists), the request is a replay.

8. HTTP Header Registration

This document registers the following HTTP header fields:

X-SURADAR-Auth Contains the SURADAR authentication token: 64 base64url characters encoding the 16-byte client nonce concatenated with the 32-byte HMAC-SHA-256 signature.

X-SURADAR-Client Contains the client identifier used to look up the enrollment record on the server.

X-SURADAR-TBand Contains the time band index as a decimal integer, allowing the server to verify the token was generated within an acceptable time window.

9. Security Considerations

9.1. Threat Model

The attacker is assumed to be able to observe network traffic (despite TLS, e.g., via compromised middlebox), capture tokens, and attempt to reuse them. The attacker cannot obtain the shared seed S or the Root Server Key RSK.

9.2. Stolen Token Analysis

A captured token reveals R (not secret) and sig. The attacker cannot derive K without K1, cannot derive K1 without S, and cannot derive S without RSK. The blast radius of a stolen SURADAR token is exactly one request -- the request for which it was generated.

9.3. Scope Escalation Prevention

The scope is embedded in ctx, which is embedded in the K1 derivation. Changing the scope changes ctx, which changes K1, which changes K, which changes sig. An attacker cannot re-scope a captured token -- this is a cryptographic guarantee, not a policy check.

9.4. Body Integrity

The HMAC signature covers the full request body. Any modification to the body invalidates the signature.

9.5. Nonce Store Poisoning Resistance

The replay check occurs after HMAC verification. An attacker who sends garbage tokens cannot poison the nonce store because those tokens will fail HMAC verification before the nonce is recorded.

9.6. Timing Attacks

Implementations MUST use constant-time comparison for HMAC verification. Implementations MUST zero all key material (K1, K) immediately after use to limit the window for memory-disclosure attacks.

9.7. Key Material Lifetime

Material	Lifetime	Storage
RSK	Long-lived, rotatable	HSM / Vault
S	Client lifetime	OS keychain
K1	~microseconds	Memory only
K	~microseconds	Memory only
R	Single request	Transmitted then discarded

Table 3: Key Material Lifetime and Storage

9.8. Clock Synchronization

Both client and server clocks SHOULD be synchronized via NTP or an equivalent protocol. The TBandSkew parameter (default: 1) allows for minor clock drift by accepting tokens from adjacent time bands.

9.9. Forward Secrecy

K1 is derived from the time band T. Once a time band expires, the corresponding K1 cannot be recomputed without the seed S and the exact time band value. Past tokens are unrecoverable without the R values, which are ephemeral and not stored by the server after replay checking.

9.10. Comparison with HTTP Message Signatures (RFC 9421)

Property	RFC 9421	SURADAR
Key per request	Same key	Unique key
Context in key	No (in signature input)	Yes (in key derivation)
Body coverage	Optional	Always
Token size	Variable (large)	Fixed (48 bytes)
Asymmetric support	Yes	No
Replay prevention	Not specified	Built-in

Table 4: SURADAR vs RFC 9421 Comparison

10. IANA Considerations

10.1. HTTP Authentication Scheme Registration

This document requests registration of the following HTTP authentication scheme in the "HTTP Authentication Scheme Registry" established by [RFC9421]:

Authentication Scheme Name: SURADAR

Reference: This document

Notes: The SURADAR scheme uses per-request HMAC-based authentication. The token is transmitted in the X-SURADAR-Auth header rather than the Authorization header to avoid conflicts with existing authentication infrastructure.

10.2. HTTP Header Field Registration

This document requests registration of the following HTTP header fields in the "Hypertext Transfer Protocol (HTTP) Field Name Registry":

X-SURADAR-Auth Status: permanent

Reference: Section 8 of this document

X-SURADAR-Client Status: permanent

Reference: Section 8 of this document

X-SURADAR-TBand Status: permanent

Reference: Section 8 of this document

10.3. SURADAR Algorithm Registry

IANA is requested to create a new registry titled "SURADAR Algorithm Registry" with the following initial entry, under the registration policy of Specification Required [RFC8126]:

Algorithm Name	HMAC Function	Hash Function	Reference
SURADAR-SHA256	HMAC-SHA-256	SHA-256	This document

Table 5: SURADAR Algorithm Registry Initial Contents

11. References

11.1. Normative References

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[FIPS180-4] National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)", FIPS PUB 180-4, August 2015, <https://csrc.nist.gov/publications/detail/fips/180/4/final>.

11.2. Informative References

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/info/rfc6749>.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/info/rfc6750>.

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <https://www.rfc-editor.org/info/rfc6819>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8179] Bradner, S. and J. Contreras, "Intellectual Property Rights in IETF Technology", RFC 8179, DOI 10.17487/RFC8179, May 2017, <https://www.rfc-editor.org/info/rfc8179>.

[RFC9421] Backman, A., Ed., Richer, J., Ed., and M. Sporny, "HTTP Message Signatures", RFC 9421, DOI 10.17487/RFC9421, February 2024, <https://www.rfc-editor.org/info/rfc9421>.

[RFC6238] M'Raihi, D., Machani, S., Pei, M., and J. Rydell, "TOTP: Time-Based One-Time Password Algorithm", RFC 6238, DOI 10.17487/RFC6238, May 2011, <https://www.rfc-editor.org/info/rfc6238>.

Appendix A. Test Vectors

A.1. Context Fingerprint

Input parameters:

```
method   = "GET"
path     = "/api/v1/findings"
orgID    = "acme-corp"
scope    = "api:read"
```

Concatenated input (hex):

```
474554                          # "GET"
00                              # separator
2f6170692f76312f66696e64696e6773 # "/api/v1/findings"
00                              # separator
61636d652d636f7270              # "acme-corp"
00                              # separator
6170693a72656164                # "api:read"
```

Context fingerprint:

```
ctx = SHA-256(above) =
  e3b7a0... (implementors: compute from the above byte sequence)
```

A.2. Full Token Generation

Input parameters:

```
seed      = 0x0102030405060708090a0b0c0d0e0f10
            111213141516171819 1a1b1c1d1e1f20   (32 bytes)
clientID  = "ci-runner-01"
orgID     = "acme-corp"
method    = "GET"
path      = "/api/v1/findings"
scope     = "api:read"
body      = ""  (empty)
unix_ts   = 1709769600
TBandSeconds = 30
R         = 0xdeadbeefdeadbeefdeadbeefdeadbeef  (16 bytes, fixed for test)
```

Derivation steps:

```
T          = floor(1709769600 / 30) = 56992320
T_bytes    = 0x0000000003660000  (8-byte big-endian)
ctx        = SHA-256("GET" || 0x00 || "/api/v1/findings" || 0x00
             || "acme-corp" || 0x00 || "api:read")
K1         = HMAC-SHA-256(seed, T_bytes || ctx)
K          = HMAC-SHA-256(K1, R)
sig        = HMAC-SHA-256(K, "")
token      = base64url_no_pad(R || sig)
```

Implementors MUST verify that their implementation produces identical intermediate values at each step.

A.3. Cross-Language Byte Identity

Implementations in different languages MUST produce byte-identical output for identical inputs. In particular:

* String-to-bytes conversion MUST use UTF-8 encoding.

* The time band T MUST be encoded as an 8-byte big-endian unsigned integer with leading zeros.

* The base64url encoding MUST NOT include padding characters.

Appendix B. Comparison with Existing Schemes

B.1. Attack Resistance Matrix

Attack	Bearer	JWT	JWT+JTI	SURADAR
Token Replay	VULN	VULN	RESIST	RESIST
Scope Escalation	VULN	VULN	VULN	RESIST
Cross-Org Abuse	VULN	VULN	VULN	RESIST
Body Tampering	VULN	VULN	VULN	RESIST
Method Swap	VULN	VULN	VULN	RESIST
Path Swap	VULN	VULN	VULN	RESIST
Stolen Token Blast	VULN	VULN	VULN	RESIST
Score	0/7	0/7	2/7	7/7

Table 6: Attack Resistance: Bearer vs JWT vs JWT+JTI vs SURADAR

B.2. Performance Comparison

Benchmarks measured on Apple M3, Go 1.26.1, single-threaded, 1000 iterations, median values:

Metric	Bearer	JWT	JWT+JTI	SURADAR
Generation	729 ns	1024 ns	1024 ns	1298 ns
Verification	52 ns	1107 ns	1868 ns	1630 ns
Token size	32 B	195 B	236 B	82 B
Security score	0/7	0/7	2/7	7/7

Table 7: Performance: Bearer vs JWT vs JWT+JTI vs SURADAR

Author’s Address

Karthik Rampalli
Glyphzero Labs Inc.
Email: karthik@phantomcorgi.com