// Security Assessment

07.29.2025 - 08.12.2025

Peersyst

NEAR Foundation

HALBORN

Peersyst - NEAR Foundation

Prepared by: HALBORN

Last Updated 09/30/2025

Date of Engagement: July 29th, 2025 - August 12th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
10	0	0	3	2	5

TABLE OF CONTENTS

8.10 Excessive allocations parsing jwt with unbounded split

1. Introduction

Halborn was engaged by the FastAuth team to perform a targeted security assessment of the FastAuth authentication and signing stack on NEAR. The assessment focused on the Rust smart contracts and the browser SDK that collaboratively verify JWT-based identities and orchestrate MPC-backed signatures. The review was conducted against commit `c9bbb3a` .

2. Assessment Summary

A senior blockchain security engineer with expertise in Rust, NEAR smart contracts, cross-contract call patterns, and authentication protocols (JWT/OIDC) undertook the audit on a full-time basis.

The assessment concentrated on authorization boundaries, trust assumptions in guards and routers, integrity of MPC paths, resistance to replay attacks, storage and gas economics, and developer ergonomics.

The architecture relies on:

- A core contract (`fa`) that verifies identities via a guard, then requests an MPC signature over a caller-supplied payload.
- An optional router (`jwt-guard-router`) that maps tenant and issuer names to specific guard contracts.
- Guard contracts (e.g., `auth0-guard`) that validate JWTs and enforce binding between the JWT's `fatxn` claim and the `sign_payload` request.
- A browser SDK that sources JWTs, constructs contract calls, and verifies signatures client-side.

3. Engagement Objectives

- Confirm that new and modified contract and SDK behaviors operate as intended and resist misuse.
- Identify vulnerabilities that could lead to incorrect authorization, tenant or guard impersonation, privilege escalation, denial of service (DoS), state corruption, or fund loss.
- Highlight opportunities for hardening, including stricter access controls, improved input validation, replay protections, storage and gas metering, and safer logging practices.

4. Test Approach And Methodology

Architecture and threat modeling

- Mapped trust boundaries across the client, IdP/JWT, `fa` contract, router, guard, and MPC.
- Modeled attacker capabilities such as untrusted clients, malicious routers or guards, confused-deputy call chains, replay attacks, and routing hijacks.

Manual code review (Rust)

- Conducted line-by-line analysis of contract state, access control mechanisms (`signer` vs `predecessor`), cross-contract calls, callbacks, and panic paths.
- Reviewed SDK request construction, focusing on how user input influences on-chain calls and how signatures are verified client-side.

Targeted dynamic testing

- Executed unit and integration tests using `near-workspaces` to validate guard routing, JWT verification flows, and MPC request formation.
- Crafted adversarial inputs, including malformed or oversized JWTs, prefix/suffix manipulations, and callback spoofing attempts.

Static checks and defensive coding review

- Assessed storage and deposit handling, gas budgeting, input size restrictions, logging practices, and handler visibility (e.g., `#[private]` on callbacks and `init` functions).

This comprehensive approach enabled us to identify both classical smart contract vulnerabilities, such as confused deputy issues, inadequate input validation, and storage mishandling and domain-specific risks related to JWT and IdP trust, router configuration, and MPC path integrity. The Detailed Findings section provides impact assessments and specific mitigations for each identified issue.

5. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

5.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

5.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

5.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

6. SCOPE

REPOSITORY

(a) Repository: fast-auth

(b) Assessed Commit ID: c9bbb3a

(c) Items in scope:

- auth0-guard
- fa
- jwt-guard-router
- mocks

REMEDIATION COMMIT ID:

- ece5e1c

Out-of-Scope: New features/implementations after the remediation commit IDs.

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	3	2	5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING VALIDATION OF STANDARD JWT CLAIMS	MEDIUM	SOLVED - 09/19/2025
CROSS-APPLICATION IMPERSONATION INSIDE A SINGLE AUTH0 TENANT	MEDIUM	SOLVED - 09/19/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
CROSS-CONTRACT PRIVILEGE ESCALATION VIA SIGNER_ACCOUNT_ID OWNERSHIP CHECK	MEDIUM	SOLVED - 09/19/2025
RS256 VERIFIER REJECTS VALID SIGNATURES WITH LEADING ZEROS	LOW	SOLVED - 09/19/2025
UNRESTRICTED KEY ROTATION PERMITS WEAK OR MALFORMED RSA PUBLIC KEYS	LOW	SOLVED - 09/19/2025
UNBOUNDED JSON PARSING ENABLES GAS GRIEFING	INFORMATIONAL	SOLVED - 09/19/2025
UNBOUNDED EXPONENTIATION LOOP IN RSA VERIFICATION	INFORMATIONAL	SOLVED - 09/19/2025
TIMING/GAS SIDE-CHANNEL IN RSA EXPONENTIATION	INFORMATIONAL	SOLVED - 09/19/2025
PATH INJECTION VIA UNTRUSTED SUB USED IN MPC PATH	INFORMATIONAL	SOLVED - 09/19/2025
EXCESSIVE ALLOCATIONS PARSING JWT WITH UNBOUNDED SPLIT	INFORMATIONAL	SOLVED - 09/19/2025

8. FINDINGS & TECH DETAILS

8.1 MISSING VALIDATION OF STANDARD JWT CLAIMS

// MEDIUM

Description

`auth0-guard` verifies the RS256 signature and a custom fatxn claim but ignores all standard security claims that limit a token's lifetime and scope. A token signed by the configured public key is accepted even if it is:

- Expired (exp) or not yet valid (nbf)
- Issued long ago (iat) – replayable forever
- Coming from another issuer (iss)
- Intended for a different audience (aud)

```
fn verify_custom_claims(&self, jwt_payload: Vec<u8>, sign_payload: Vec<u8>) -> (bool, String) {
    let claims: CustomClaims = match serde_json::from_slice(&jwt_payload) {
        Ok(claims) => claims,
        Err(error) => return (false, error.to_string()),
    };

    // Only custom equality check
    if claims.fatxn != sign_payload {
        return (false, "Transaction payload mismatch".to_string());
    }

    // No checks for exp / nbf / iat / iss / aud
    (true, claims.sub)
}
```

Impact

- Replay of login tokens:
  - Any stolen or archived JWT grants indefinite access wherever fa.verify or jwt-guard-router.verify is used for authentication.
- Cross-audience escalation:
  - Tokens issued for a different API or client_id (but same Auth0 tenant/key) are accepted, letting attackers access services they were never entitled to.
- Bypass of scheduled revocation:
  - Administrators rotating keys or shortening session lifetime gain no protection until this contract is upgraded.
- Transaction-signature replay (lower risk):
  - In the sign flow the attacker can only resign the same fatxn payload. The damage is limited to replaying a previously authorised on-chain action.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:H/A:N/I:M/D:N/Y:N (5.9)

Recommendation

It is recommended that a comprehensive claim structure be introduced, covering `exp`, `nbf`, `iat`, `iss`, and `aud` in addition to the existing `sub` and `fatxn` fields:

```
#[derive(Deserialize)]

struct StdClaims {
    sub:   String,
    fatxn: Vec<u8>,
    iss:   String,
    aud:   Vec<String>,
    exp:   u64,
    nbf:   Option<u64>,
    iat:   Option<u64>,
}
```

During verification, the following checks should be performed:

```
let now = env::block_timestamp_ms() / 1_000;          // seconds

assert!(claims.exp >  now,              "token expired");
assert!(claims.nbf.unwrap_or(0) <= now, "token not yet valid");
assert!(claims.iss == EXPECTED_ISSUER,  "invalid issuer");
assert!(claims.aud.contains(&EXPECTED_AUD.to_string()), "invalid audience");
```

• On any failure, the function should return `(false, "token invalid")` instead of panicking.
• It is further advised that unit tests be added for expired tokens, future-dated tokens, wrong audiences, and wrong issuers to guard against regressions.

Remediation Comment

SOLVED: The **FastAuth team** successfully solved this finding. Guard now enforces iss/exp/nbf.

Remediation Hash

https://github.com/Peersyst/fast-auth/commit/ece5e1ca33cfa7c5e93a1453d6183b5f59c92aa4

8.2 CROSS-APPLICATION IMPERSONATION INSIDE A SINGLE AUTH0 TENANT

// MEDIUM

Description

Fast-Auth decides which guard contract to invoke by keeping only the portion of `guard_id` that precedes the first '`#`' (the prefix).

Everything after that character (the suffix, which normally identifies the Auth0 tenant or application) is ignored when selecting the guard, yet the full string is forwarded to the guard contract.

```
pub fn verify(&self, guard_id: String, verify_payload: String, sign_payload: Vec<u8>) -> Promise {
    // ❶ keep only the prefix
    let guard_prefix  = self.get_guard_prefix(guard_id.clone());      // "jwt"
    // ❷ resolve guard by prefix
    let guard_address = self.guards.get(&guard_prefix)
        .expect("Cannot verify: Guard does not exist");
    // ❸ forward the *entire* guard_id (prefix + suffix)
    external_guard::ext(guard_address)
        .verify(guard_id, verify_payload, sign_payload)
        .then(Self::ext(env::current_account_id()).on_verify_callback())
}
```

Exploit Flow

- Tenant domain: https://acme.us.auth0.com/
  - Auth0 signs **all** applications in this tenant with the same RSA key K.
- Fast-Auth registry:

```
  fa.add_guard("jwt", router.near)              // prefix only
  router.add_guard("acme.us.auth0.com", guard)   // tenant suffix
```

- Attacker logs in to Application B (aud = "app_api") and receives a valid JWT or obtain/steal another users AppB JWT.
- Attacker calls Fast-Auth:

```
  guard_id        = "jwt#acme.us.auth0.com"
  verify_payload  = <JWT for app_api>
  sign_payload    = <fatxn bytes from that JWT>
```

- Fast-Auth strips to "jwt" → routes to the guard; guard verifies the signature, ignores the aud claim, and returns success.
- Application A (“Dashboard”) now accepts the user, even though the token was minted for Application B.

Impact

- Cross-application impersonation: a user authenticated for one Client-ID gains access to any other application that relies on Fast-Auth.
- Potential privilege escalation if different applications (or suffixes such as jwt#pro) carry higher roles or paid tiers.

Proof of Concept

```
use near_sdk::test_utils::{accounts, VMContextBuilder};
use near_sdk::{testing_env, AccountId};
use std::collections::HashMap;

// Bring FastAuth into scope
use fa::FastAuth;

#[test]
fn resolves_by_prefix_only() {
    // 1. mock context
    let owner: AccountId = accounts(0);
    let mut ctx = VMContextBuilder::new();
    ctx.signer_account_id(owner.clone());
    testing_env!(ctx.build());

    // 2. FastAuth with single prefix mapping
    let mut guards: HashMap<String, AccountId> = HashMap::new();
    guards.insert("jwt".to_string(), "router.near".parse().unwrap());

    let contract = FastAuth {
        guards,
        owner,
        mpc_address: "mpc.near".parse().unwrap(),
        mpc_key_version: 1,
        version: "test".to_string(),
    };

    // 3. Attacker-supplied guard_id carries unapproved suffix
    let prefix = contract.get_guard_prefix("jwt#evil.com".to_string());
    assert_eq!(prefix, "jwt");

    // 4. Prefix resolves successfully even though suffix unregistered
    let resolved = contract.guards.get(&prefix).unwrap();
    assert_eq!(resolved.as_str(), "router.near");
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:H/A:M/I:N/D:N/Y:N (5.9)

Recommendation

To mitigate this issue the following improvements are recommended:

- Guards should be registered and resolved by the **full** identifier, including the tenant suffix ("jwt#acme.us.auth0.com").
- Alternatively, before forwarding, the router must check the supplied suffix and the JWT's aud/iss against an allow-list; requests that do not match must be rejected.

- In addition, auth0-guard should validate standard claims (aud, iss, exp, nbf, iat) so that a token issued for Application B cannot be reused for Application A even if the same key signs both.

Remediation Comment

SOLVED: The **FastAuth team** successfully solved this finding. Router `add_guard` is owner-only and guard enforces issuer.

Remediation Hash

https://github.com/Peersyst/fast-auth/commit/ece5e1ca33cfa7c5e93a1453d6183b5f59c92aa4

8.3 CROSS-CONTRACT PRIVILEGE ESCALATION VIA SIGNER_ACCOUNT_ID OWNERSHIP CHECK

// MEDIUM

Description

Owner-only functions in every Fast-Auth contract (`auth0-guard` , `jwt-guard-router` , `fa`) rely on the top-level transaction signer (`env::signer_account_id()`) instead of the immediate caller (`env::predecessor_account_id()`).

Since the signer value is preserved across all asynchronous calls, any contract that the owner interacts with can forward privileged calls on the owner's behalf.

```
fn only_owner(&self) {
    assert!(env::signer_account_id() == self.owner, "Only the owner can call this function");
}
```

Impact

- Attacker convinces the owner to call any function on a malicious contract (AttackerContract).
- AttackerContract issues a promise to a privileged method, e.g.:

```
Promise::new("auth0.guard.near")
       .function_call("set_public_key", malicious_key, 0, KEY);
```

- The only_owner assertion passes and the attacker gains full administrative control:
  - Replace RSA modulus with a weak / even value → forge signatures or brick verification.
  - Transfer ownership of router or Fast-Auth, remove guards, redirect MPC, etc.

BVSS

AO:S/AC:L/AX:L/R:N/S:C/C:C/A:C/I:C/D:C/Y:C (5.0)

Recommendation

To mitigate this issue, we recommend the the following improvements:

- Replace the checks with `predecessor validation` :

```
assert_eq!(env::predecessor_account_id(), self.owner, "Only owner");
```

- If legitimate cross-contract administration is required, expose a public wrapper that first validates `predecessor_account_id` and then invokes the internal function via `Promise::new(env::current_account_id()).function_call(...)` .

Remediation Comment

SOLVED: The **FastAuth team** successfully solved this finding; the `fa` and `jwt-guard-router` owner checks switched to predecessor-based authorization. Note: `fa` 's pauser role still uses signer (lower impact).

Remediation Hash

https://github.com/Peersyst/fast-auth/commit/ece5e1ca33cfa7c5e93a1453d6183b5f59c92aa4

8.4 RS256 VERIFIER REJECTS VALID SIGNATURES WITH LEADING ZEROS

// LOW

Description

The current RS256 verification rejects signatures whose big-endian integer has fewer significant bits than the modulus (i.e., signatures with leading zero bits/bytes). PKCS#1 v1.5 and JWS allow leading zeros; the only required bound is signature < n.

Code Location

```
// contracts/auth0-guard/src/rsa/rs256.rs (simplified)
if signature >= *pub_key.n.as_ref() || signature.bits_precision() != pub_key.n.bits_precision() {
    return false;
}
```

- The second condition enforces exact bit-precision equality with the modulus. Valid signatures that decode with leading zero bits/bytes fail this check.

Impact

- False negatives: valid JWT signatures can be rejected, causing failed verifications, degraded UX, and intermittent outages.
- Prevalence: “Top bit not set” (signature’s bit-length < modulus bit-length): = 50% of valid signatures.
    - “Leading zero byte” (first byte 0x00): ≈ 1/256 ≈ 0.39%.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

To mitigate this issue we recommend one of the following:

1) (Preferred) left-pad signatures to k bytes before bigint conversion:

- Compute k = pub_key.size() (modulus length in bytes).
- If signature_bytes.len() > k → reject (no panic).
- If signature_bytes.len() < k → left-pad with zeros to length k.
- Keep the bound check signature < n and full PKCS#1 v1.5 padding checks.

```
// PSEUDO CODE ONLY
let k = pub_key.size();
if signature_bytes.len() > k { return false; } // reject oversize
let sig_be = if signature_bytes.len() < k {
    let mut v = vec![0u8; k];
```

```
    v[k - signature_bytes.len()..].copy_from_slice(&signature_bytes);
    v
} else {
    signature_bytes
};
// then parse bigint and continue
```

2) (minimal change) Drop the bit-precision equality check:

- Keep only signature < n:

```
// PSEUDO CODE ONLY
if signature >= *pub_key.n.as_ref() { return false; }
```

Remediation Comment

SOLVED: The **FastAuth team** successfully solved this finding by converting left-pad signatures to k bytes before bigint conversion.

Remediation Hash

https://github.com/Peersyst/fast-auth/commit/ece5e1ca33cfa7c5e93a1453d6183b5f59c92aa4

8.5 UNRESTRICTED KEY ROTATION PERMITS WEAK OR MALFORMED RSA PUBLIC KEYS

// LOW

Description

`auth0-guard` exposes the `set_public_key` method to update the RSA2048 modulus (n) and exponent (e) used for JWT signature verification.

The function accepts the two components as raw byte vectors and stores them verbatim in contract state, performing no validation of size, parity, or cryptographic strength.

```
/// Sets new RSA public key components for signature verification
pub fn set_public_key(&mut self, n: Vec<u8>, e: Vec<u8>) {
    self.only_owner();           // access-control only
    self.n_component = n;        // <- no checks on length or parity
    self.e_component = e;        // <- any exponent accepted
}
```

Why this is a problem:
RSA implementations assume that the modulus is odd and of a fixed, sufficiently large length and the public exponent is chosen from a small set of safe values (typically 0x10001 = 65 537 or, less commonly, 3).

Without these constraints:

1. **Even modulus**: The verification routine calls `Odd::new(n).expect("Odd value required")`. An even modulus triggers this expect, making every future verify call panic, permanently disabling authentication.
2. **Short modulus**: Any modulus shorter than 2 048 bits is left-padded with zeros to match the internal precision. A 512-bit or 768-bit modulus can be factored with commodity hardware; once the private key is recovered, forged JWTs will pass on-chain verification.

Impact

- **Availability:** A single transaction that stores an even modulus results in a contract that panics on every verification attempt (permanent denial-of-service until redeployment).
- **Integrity risk**: Installing a factorable modulus enables creation of valid signatures for arbitrary JWT headers and payloads, breaking authentication for every application that relies on this guard.

BVSS

AO:S/AC:L/AX:M/R:N/S:U/C:C/A:C/I:C/D:N/Y:N (2.0)

Recommendation

To mitigate this issue, it is recommended to implement validation layer before state mutation:

```
// modulus length and parity
assert_eq!(n.len(), 256, "modulus must be 2048 bits");
let n_int = BoxedUint::from_be_slice(&n, PRECISION).unwrap();
assert!(n_int.is_odd().into(), "modulus must be odd");

// exponent whitelist
let allowed_e: &[&[u8]] = &[&[0x01, 0x00, 0x01], &[0x03]];
assert!(allowed_e.iter().any(|v| *v == e.as_slice()), "invalid exponent");
```

Remediation Comment

SOLVED: The **FastAuth team** successfully solved this finding by making `set_public_key` validates 2048-bit modulus, odd modulus, and restricting the exponent to 65537.

Remediation Hash

https://github.com/Peersyst/fast-auth/commit/ece5e1ca33cfa7c5e93a1453d6183b5f59c92aa4

8.6 UNBOUNDED JSON PARSING ENABLES GAS GRIEFING

// INFORMATIONAL

Description

The JWT's fatxn claim is deserialized into an unbounded `Vec<u8>` and compared against the caller-supplied `sign_payload`. Neither the JWT size nor `fatxn` / `sign_payload` lengths are capped. Large arrays cause proportional parsing, allocation, cross-contract serialization, and hashing costs, enabling an attacker to inflate gas usage (and potentially exceed limits), especially if a relayer/app submits the on-chain call on behalf of users.

```
#[derive(Serialize, Deserialize)]
pub struct CustomClaims {
    pub sub: String,
    pub fatxn: Vec<u8>,
}
```

```
fn verify_custom_claims(&self, jwt_payload: Vec<u8>, sign_payload: Vec<u8>) -> (bool, String) {
    let claims: CustomClaims = match serde_json::from_slice(&jwt_payload) {
        Ok(claims) => claims,
        Err(error) => return (false, error.to_string()),
    };
    if claims.fatxn != sign_payload {
        return (false, "Transaction payload mismatch".to_string());
    }
    (true, claims.sub)
}
```

Impact

- Heavy gas consumption on parse/deserialize/hash proportional to input length.
- If the app/relayer is the sender, attacker can burn the app's gas

by supplying huge `fatxn` / `sign_payload`.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (1.5)

Recommendation

To mitigate this issue, it is recommended to enforce strict size caps.

Remediation Comment

SOLVED: The **FastAuth team** successfully solved this finding by enforcing strict size caps.

Remediation Hash

https://github.com/Peersyst/fast-auth/commit/ece5e1ca33cfa7c5e93a1453d6183b5f59c92aa4

8.7 UNBOUNDED EXPONENTIATION LOOP IN RSA VERIFICATION

// INFORMATIONAL

Description

`auth0-guard`'s `RSA-SHA256` verifier performs a square-and-multiply loop that iterates once for every bit in the public exponent `e` (`for i in 0..e.bits()`).

Because `set_public_key` stores the exponent verbatim, a key with an unusually long `e` (hundreds or thousands of bits) would make every verify call do proportionally more modular multiplications, inflating gas cost or, at extreme sizes, exceeding per-transaction limits.

```
// rs256.rs
for i in 0..pub_key.e.bits() {   // runs once per bit of e
    if pub_key.e.bit(i).into() {
        result = result.mul(&base).rem_vartime(modulus);
    }
    base = base.mul(&base).rem_vartime(modulus);
}
```

Impact

- **Gas inefficiency:** A 2048-bit exponent makes verification ≈120 × slower than the common 17-bit 0x10001.
- **Self-inflicted DoS:** If the owner accidentally installs an extreme exponent, legitimate users might hit gas limits when verifying JWTs.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to reject exponents longer than, say, 32 bits or not equal to 0x10001/3.

Remediation Comment

SOLVED: The **FastAuth team** successfully solved this finding by adding bound checks.

Remediation Hash

https://github.com/Peersyst/fast-auth/commit/ece5e1ca33cfa7c5e93a1453d6183b5f59c92aa4

8.8 TIMING/GAS SIDE-CHANNEL IN RSA EXPONENTIATION

// INFORMATIONAL

Description

The RS256 verifier uses square-and-multiply with a data-dependent branch and loop count proportional to the number of bits in the public exponent `e`. This makes runtime (and thus gas on NEAR) depend on:

- The bit length and Hamming weight of `e`.
- How far an input signature progresses through checks before rejection.

```
for i in 0..pub_key.e.bits() {
    if pub_key.e.bit(i).into() {
        result = result.mul(&base).rem_vartime(modulus);
    }
    base = base.mul(&base).rem_vartime(modulus);
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended that a fixed-cost failure path be applied so rejections consume approximately the same gas as success.

Remediation Comment

SOLVED: The **FastAuth team** successfully solved this finding; exponent is restricted to 65537, removing attacker-controlled timing/work variance.

Remediation Hash

https://github.com/Peersyst/fast-auth/commit/ece5e1ca33cfa7c5e93a1453d6183b5f59c92aa4

8.9 PATH INJECTION VIA UNTRUSTED SUB USED IN MPC PATH

// INFORMATIONAL

Description

`fa` constructs the MPC derivation path by concatenating the user-controlled `guard_id` with the guard-returned sub:

```
let request = SignRequest {
    payload: payload_hash,
    path: format!("{}#{}", guard_id.clone(), user),
    key_version: self.mpc_key_version,
};
```

A malicious guard can return any `sub` (including #, very long strings, non-printables), because guards currently just return the JWT sub and do not sanitize:

```
if claims.fatxn != sign_payload { return (false, "Transaction payload mismatch".to_string()); }
(true, claims.sub)
```

The router formats `guard_name#sub` similarly when it proxies verifications:

```
let (valid, sub) = call_result.unwrap();
if valid {
    let result = self.format_path(guard_name, sub);
    (true, result)
} else { (false, "".to_string()) }
```

SDK/verifier derives the expected path from the JWT issuer and `sub` (not from guard output), so mismatches lead to verification failure (DoS), not privilege gain:

```
const token = await this.client.getTokenSilently();
const { sub } = decodeJwt(token);
return `jwt#https://${this.options.domain}/#${sub}`;
```

```
const token = await this.client.getTokenSilently();
const decoded = decodeJwt(token);
return { guardId: `jwt#https://${this.options.domain}/`, verifyPayload: token, signPayload: decoded["f
```

Impact

- Today: DoS and potential confusion. A malicious guard can force fa to request signatures for unusable paths (won't verify against SDK-derived path).
- If any downstream (MPC/verifier) normalizes/splits path by #, crafted sub like victim#extra could collide with or impersonate jwt#tenant#victim.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

To mitigate this issue, the following possible hardenings are recommended:

- **In fa before building path:** Reject sub containing # or control chars; enforce a safe charset and a strict length cap (e.g., 256).
- In guards: Return exactly the JWT sub; validate charset, length; strip/forbid #.
- In jwt-guard-router: Make `add_guard` owner-only or add proof-of-ownership; this reduces exposure to malicious guards.
- In MPC/verifier: Treat path as opaque; do not split/normalize; enforce input constraints server-side too.

Remediation Comment

SOLVED: The **FastAuth team** successfully solved this finding; `fa` now rejects `sub` containing `#` and caps length (≤256).

Remediation Hash

https://github.com/Peersyst/fast-auth/commit/ece5e1ca33cfa7c5e93a1453d6183b5f59c92aa4

8.10 EXCESSIVE ALLOCATIONS PARSING JWT WITH UNBOUNDED SPLIT

// INFORMATIONAL

Description

The function splits an attacker-supplied JWT using `split('.')` and collects all segments into a `Vec<&str>`. A malicious caller can submit an oversized string with many '.' characters, causing:

- Large intermediate allocations (for the vector and substrings).
- This function is on the hot path of `auth0-guard.verify`, reached via `fa.sign/verify` → `router` (optional) → `guard`

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

To mitigate this issue, it is recommended to switch to `splitn(3, '.')` to avoid collecting unbounded segments and enforce a sane maximum JWT size (e.g., 16 KiB) and non-empty segments before processing or base64-decoding.

Remediation Comment

SOLVED: The **FastAuth team** successfully solved this finding; switched to `splitn(3, '.')` and rejected extra dots; and added optional size caps for completeness.

Remediation Hash

https://github.com/Peersyst/fast-auth/commit/ece5e1ca33cfa7c5e93a1453d6183b5f59c92aa4
