Guvenkaya® The Bedrock of Security

Aurora

NEAR Intents Passkeys Pull Request Security Review

Lead Security Engineer: Timur Guvenkaya
Date of Engagement: 6th February 2025 - 10th February 2025
Visit: www.guvenkaya.co

Contents

About Us 01
About Aurora 01
Audit Results 02
.1 Project Scope 02
.2 Out of Scope 03
.3 Timeline 03
Methodology 04
Severity Breakdown 05
.1 Likelihood Ratings 05
.2 Impact 05
.3 Severity Ratings 05
.4 Likelihood Matrix 06
.5 Likelihood/Impact Matrix 06
Findings Summary 07
Findings Details 09
.1 GUV-1 Potential DoS of The Main Functionality Through Malicious Solvers - Medium 09
.2 GUV-2 Anyone Can Fake The Passkey Signature For Their Own Account - Informational 11
.3 GUV-3 Unsafe Callback URL Validation - Informational 12
.4 GUV-4 Insufficient Nonce Entropy Validation - Informational 14
.5 GUV-5 Different Signature Malleability For Each Curve - Informational 15

About Us

Guvenkaya is a security research firm specializing in Rust security, Web3 security of Non-EVM protocols, and Web2 security. With our expertise, we provide both security auditing services and custom security solutions

About Aurora

Aurora is a Virtual Chain built on NEAR. It's, at the same time, the sandbox and the proof of the robustness of the parent protocol. It's a smart contract - probably the most complex that exists - that is also an Ethereum Virtual Machine, providing a turn-key solution for developers to operate their apps on an Ethereum-compatible, high-throughput, scalable and future-safe platform, with low transaction costs.

Audit Results

Guvenkaya conducted a security assessment of the **NEAR Intents Passkeys Pull Request** from 6th February 2025 to 10th February 2025. During this engagement, a total of **5 findings** were reported. 1 of the findings was medium and the remaining were informational severity. All issues were either fixed, acknowledged, or scheduled per the remediation status.

Project Scope

Files	Link
Passkeys Pull Request	https://github.com/near/intents/pull/43

Out of Scope

The audit will include reviewing the code for security vulnerabilities. The audit does not include a review of the tests and dependencies.

Timeline

Start of the audit	Draft report	Final report
6th February 2025	10th February 2025	6th March 2025

Methodology

RESEARCH INTO PROJECT ARCHITECTURE

PREPARING ATTACK VECTORS

SETTING UP AN ENVIRONMENT

MANUAL CODE REVIEW OF THE CODE

ASSESSMENT OF RUST SECURITY ISSUES

ASSESSMENT OF NEAR SECURITY ISSUES

ASSESSMENT OF ARITHMETIC ISSUES

BUSINESS LOGIC VULNERABILITY ASSESSMENT

ONCHAIN TESTING USING NEAR WORKSPACES

BEST PRACTICES AND CODE QUALITY

CHECKING FOR CODE REFACTORING/SIMPLIFICATION POSSIBILITIES

ARCHITECTURE IMPROVEMENT SUGGESTIONS

PREPARING POCS AND/OR TESTS FOR EACH CRITICAL/HIGH/MEDIUM ISSUES

Severity Breakdown

01. Likelihood Ratings

Likely: The vulnerability is easily discoverable and not overly complex to exploit.
Possible: The vulnerability presents some challenges either in discovery or in the complexity of the attack.
Rare: The vulnerability is either very difcult to discover or complex to exploit, or both.
This matrix provides a nuanced view, taking into account both the ease of discovering a vulnerability and the complexity involved in exploiting it.

02. Impact

Severe: The vulnerability is easily discoverable and not overly complex to exploit.
Moderate: The vulnerability presents some challenges either in discovery or in the complexity of the attack.
Negligible: The vulnerability is either very difcult to discover or complex to exploit, or both.

03. Severity Ratings

Critical: Assigned to vulnerabilities with severe impact and a likely likelihood of exploitation.
High: For vulnerabilities with either severe impact but only a possible likelihood, or moderate impact with a likely likelihood.
Medium: Used for vulnerabilities with severe impact but a rare likelihood, moderate impact with a possible likelihood, or negligible impact with a likely likelihood.
Low: For vulnerabilities with moderate impact and rare likelihood, or negligible impact with a possible likelihood.
Informational: The lowest severity rating, typically for vulnerabilities with negligible impact and a rare likelihood of exploitation.

CRITICAL **HIGH** **MEDIUM** Low Informational

Likelihood Matrix:

Attack Complexity \ Discovery Ease	Obvious	Concealed	Hidden
Complex	Possible	Rare	Rare
Moderate	Likely	Possible	Rare
Straightforward	Likely	Possible	Possible

Likelihood/Impact Matrix:

Likelihood \ Impact	Severe	Moderate	Negligible
Likely	**CRITICAL**	**HIGH**	**MEDIUM**
Possible	**HIGH**	**MEDIUM**	Low
Rare	**MEDIUM**	Low	Informational

Findings Summary

01. Remediation Complexity: This measures how difcult it is to fx the vulnerability once it has been identifed.

Simple: Patches or fixes are readily available and easily implemented.

Moderate: Requires some time and resources to remediate, but well within the capabilities of most organizations.

Difficult: Remediation requires significant resources, specialized skills, or substantial changes to systems or architecture.

02. Status: This measures how difcult it is to fx the vulnerability once it has been identifed.

Not Fixed: Indicates that the vulnerability has been identifed but no remedial action has been taken yet. This status is crucial for newly discovered vulnerabilities or those awaiting prioritization.

Fixed: This status is applied when the vulnerability has been successfully remediated. It implies that appropriate measures (like patching, confguration changes, or architectural modifcations) have been implemented to resolve the issue.

Acknowledged: This status is used for vulnerabilities that have been recognized, but for various reasons (such as risk acceptance, cost, or other business decisions), have not been fxed. It indicates that the risk posed by the vulnerability is known and has been consciously accepted.

Scheduled: This status indicates that the vulnerability has been acknowledged and a plan is in place to fix it in the future. It signifies that while remediation hasn't yet occurred, the issue has been prioritized and is part of the planned development roadmap.

Finding	Impact	Likelihood	Severity	Remediation Complexity	Remediation Status
GUV-1: Potential DoS of The Main Functionality Through Malicious Solvers - Medium	Severe	Rare	**MEDIUM**	Moderate	Scheduled
GUV-2: Anyone Can Fake The Passkey Signature For Their Own Account	Negligible	Rare	Informational	Complex	Acknowledged
GUV-3: Unsafe Callback URL Validation	Negligible	Rare	Informational	Simple	Acknowledged
GUV-4: Insufficient Nonce Entropy Validation	Negligible	Rare	Informational	Simple	Acknowledged
GUV-5: Different Signature Malleability For Each Curve	Negligible	Rare	Informational	Simple	Fixed

Findings Details

GUV-1 Potential DoS of The Main Functionality Through Malicious Solvers - Medium

Malicious solvers can cause service degradation or denial of service (DoS) to frontends and any systems relying on solver bus data.

Causes That Make This Attack Possible:

- Frontends typically are optimizing to give the best quote to the user
- Solver bus simulates intents and aggregates quotes submitted via websockets
- A time gap exists between the solver bus's intent simulation and user signature submission
- Quote submission to the solver bus is unrestricted
- The failure of one intent causes all other intents to fail:

core:execute_signed_intents:core/src/engine/mod.rs

```
pub fn execute_signed_intents(
        mut self,
        signed: impl IntoIterator<Item = MultiPayload>,
    ) -> Result<Transfers> {
     for signed in signed {
       self.execute_signed_intent(signed)?;
     }
        self.finalize()
    }
```

- An attacker controls multiple solvers with deposited funds for popular token pairs, consistently submitting the best prices. The solver's available funds aren't verified after intent simulation
- The solver bus returns quotes including the attacker's quote. Frontends likely select the attacker's quote due to its superior pricing
- The user signs the transaction and submits it to the smart contract
- A single malicious intent causes the entire transaction to fail, even if other intents were valid, due to the execution loop's error handling
- By repeating this process for each quote request, the attacker prevents reliable intent execution. This risk is heightened during early adoption when solver numbers are limited

Impact:

- Reputation damage
- Inability for regular users to execute intents (griefing)

Recommendation

Implement a solver reputation system (possibly using a trusted solver list), economic penalties, and progressive timeout/backoff for unreliable solvers. This could be implemented with the addition of API keys.

Remediation - Scheduled

The Aurora team has acknowledged the issue and will fix it soon by introducing API keys and/or reputaion-based filteting for solvers on Solver Bus level.

GUV-2 Anyone Can Fake The Passkey Signature For Their Own Account - Informational

It was observed that, anyone can generate a valid **P256 key**, add their public key through the **add_public_key** function, sign the required data, and successfully pass the smart contract verification. This effectively allows users to trick the smart contract into believing that an object was signed by a passkey when it wasn't. The significance of this issue depends on whether there's a need to differentiate between passkey and hardware key signatures, or if there are special benefits planned for passkey users.

Impact:

Users can fake signatures without utilizing passkeys

Recommendation

To ensure the passkey is genuine and from a verified source, you should:

Offchain - During Registration:

- Obtain the attestation certificate (CA and device) from the blob at https://fidoalliance.org/metadata/ or other sources
- Verify the device's AAGUID, certificate chain, and newly generated public key
- If verification succeeds, call the smart contract to add the public key (consider mapping between AAGUID/credential_id &pubkey). Disable unverified public key additions for P256 signatures

Offchain - During Signing: Sign the intents object without the public_key field and send it to SC

Onchain:

- Verify the flags and attestedData = true, then fetch the stored public key (using credential ID/AAGUID)
- Check the signature against the known public key
- Perform regular verifications (nonce, signer matches public key, etc.)

Remediation - Acknowledged

The Aurora team has acknowledged the issue but it won't be fixed since it is allowed by design.

GUV-3 Unsafe Callback URL Validation - Informational

The **Nep413Payload** structure and **with_callback_url** function in Nep413Payload implementation accepts callback URLs without any validation, potentially allowing malicious or unsafe URLs to be passed.

nep413:nep413/src/lib.rs

```
#[near(serializers = [borsh, json])]
        #[serde(rename_all = "camelCase")]
        #[derive(Debug, Clone)]
        pub struct Nep413Payload {
            pub message: String,

            #[serde_as(as = "Base64")]
            pub nonce: [u8; 32],

            pub recipient: String,

            #[serde(default, skip_serializing_if = "Option::is_none")]
          pub callback_url: Option<String>,
        }

        // ... existing code ...
        #[inline]
        pub fn with_callback_url(mut self, callback_url: String) -> Self {
       self.callback_url = Some(callback_url);
        self
        }
```

Impact:

This allows arbitrary strings to be used as callback URLs, including:

- Dangerous protocols (e.g., file://, data://)
- Malformed URLs
- URLs with null bytes or other control characters

If any off-chain component processes these callback URLs, it could be vulnerable to exploitation.

Recommendation

Implement proper URL validation in the with_callback_url and write a custom deserializer to verify Url on the parsing level in Nep413Payload.

Remediation - Acknowledged

The Aurora team has acknowledged the issue but it won't be fixed since it is allowed by design.

GUV-4 Insufficient Nonce Entropy Validation - Informational

Any signed payload accepts any 32-byte array as a nonce without validating its entropy or randomness.

poc:test_zero_nonce_security

```
#[test]
        fn test_zero_nonce_security() {
            let zero_nonce_payload = Nep413Payload {
                message: payload.message.clone(),
                nonce: [0u8; 32], // Zero nonce accepted
                recipient: payload.recipient.clone(),
                callback_url: None,
            };
        }
```

Impact:

If you're only using the nonce as a once-per-user unique value there are no problems. However, It is still safer and more standard to generate a random nonce of sufficient length (128 bits+) so you avoid accidental collisions and stay in line with typical cryptographic best practices.

Recommendation

Implement proper nonce validation and generation

Remediation - Acknowledged

The Aurora team has acknowledged the issue but it won't be fixed since it is allowed by design.

GUV-5 Different Signature Malleability For Each Curve - Informational

Each cryptographic curve implements signature malleability differently:

- **SeckP**: Not Allowed. Uses *ecrecover* with a **true** flag to enable malleability checks
- **Ed25519**: Allowed. Uses *env::ed25519_verify*
- **P256**: Allowed. Implements passkey verification without signature malleability checking

The behavior of signature malleability should be unified across all curves. Having consistent security properties would simplify threat modeling, debugging, and tracking.

Impact:

While nonces currently protect against replay attacks, disallowing malleability for all curves would be more secure. Future code changes, such as adding functions without nonce validation, could introduce vulnerabilities if malleability remains inconsistent.

Recommendation

Enable signature malleability checks across all curves.

Remediation - Fixed

The Aurora team has fixed the issue by enabling signature malleability checks for all curves in these PRs:

- **Ed25519**: PR-47
- **P256**: PR-44