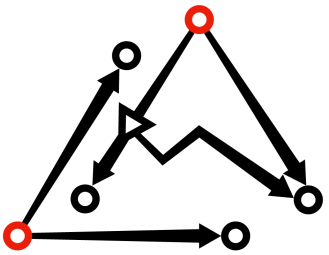

Least Authority

PRIVACY MATTERS

BLS Signatures Crate

Security Audit Report

Anza Technology

Updated Final Audit Report: 22 May 2026

Table of Contents

Overview

Background

Anza Technology has requested Least Authority perform a security audit of the BLS Signatures Crate.

Project Dates

- **March 25 - April 18, 2026:** Initial Code Review *(Completed)*
- **April 13, 2026:** Delivery of Initial Audit Report *(Completed)*
- **April 27, 2026:** Verification Review *(Completed)*
- **April 29, 2026:** Delivery of Final Audit Report *(Completed)*
- **May 22, 2026:** Delivery of Updated Final Audit Report *(Completed)*

Review Team

- George Gkitsas, Security / Cryptography Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the `solana_bls_signatures` and `solana_bls12_381_syscall` crates followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- `Solana-bls_signatures`: https://github.com/anza-xyz/solana-sdk/tree/master/bls-signatures
  - excluding the tests and benches
- `Solana_bls12_381_syscall`: https://github.com/anza-xyz/agave/tree/master/bls12-381-syscall

Specifically, we examined the following Git revisions for our initial review:

- `Solana_bls12_381_syscall`: `3fb50b4b56532ac521330e604ee4b39164aded3b`
- `Solana-bls-signatures`: `5853c4463a7957cd77eaa08266417106c5eb42d9`

For the verification, we examined the following Git revisions:

- `Solana-bls-signatures`: `4c1e9a24bc43a400adc330dc6f379e949a3699ca`
- `Solana_bls12_381_syscall`: `267b781a471ae787e390a67477fca8b08ebd6d9b`

For the review, these repositories were cloned for use during the audit and for reference in this report:

- `Solana-bls-signatures`: https://github.com/LeastAuthority/anza-xyz-solana-sdk

- `Solana_bls12_381_syscall`: https://github.com/LeastAuthority/anza-xyz-agave/tree/master

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- `solana-improvement-documents`:
  - https://github.com/solana-foundation/solana-improvement-documents/blob/main/proposals/0388-bls12-381-syscalls.md
  - https://github.com/solana-foundation/solana-improvement-documents/pull/387
- Website: https://www.anza.xyz

In addition, this audit report references the following documents:

- J. Camenisch, S. Hohenberger, and M. Ø. Pedersen, "Batch Verification of Short Signatures." *Springer Nature Link*, 2012, [CHP12]
- BLS Signatures:
  - [BBG+25]: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature
  - [BGW+22]: https://www.ietf.org/archive/id/draft-irtf-cfrg-bls-signature-05.html
- RFC 9380:
  - [FSS+23]: https://www.rfc-editor.org/rfc/rfc9380.html
- Ethereum Repository: https://github.com/ethereum/bls12-381-tests
- Serialization:
  - https://www.ietf.org/archive/id/draft-irtf-cfrg-pairing-friendly-curves-11.html#name-bls-curves-for-the-128-bit-
  - https://github.com/zkcrypto/bls12_381
- BLS12-381 | README.md: https://github.com/zkcrypto/pairing/blob/34aa52b0f7bef705917252ea63e5a13fa01af551/src/bls12_381/README.md#serialization

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Vulnerabilities within each component and whether the interaction between the components is secure;
- Key management, including secure private key storage and management of encryption and signing keys;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the execution;
- Protection against malicious attacks and other ways to exploit;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Anza's `solana-sdk` is a modular Rust SDK for the Solana blockchain, providing the core libraries, protocol types, and utility crates used by on-chain programs and the Agave validator. In this audit, we reviewed the `bls-signatures` subfolder containing the `solana-bls-signatures` crate, which implements BLS12-381 key generation, signing, verification, proof of possession, and aggregate, but not batched signature verification.

In addition, Anza's Agave is a Rust codebase for the Solana validator and related node infrastructure, including the runtime, networking, and execution components that support the operation of the network. We also reviewed the `bls12-381-syscall` subfolder, containing the `solana-bls12-381-syscall` crate, which implements native BLS12-381 syscall operations such as point arithmetic, pairing, validation, and decompression.

This review was conducted under the assumption that the `blst` and `blstrs` crates function as intended and have already been audited. Accordingly, our team did not audit the functionality of `blst` or `blstrs`.

System Design

Our team examined the design of the `solana-bls-signatures` and `solana-bls12-381-syscall` crates and found the code to be of high quality, the system to be well designed, and security considerations to be clearly reflected throughout the implementation. We did not identify any general weaknesses in the system or development processes, nor did we observe recurring patterns in the issues found.

As part of the audit, we reviewed byte encodings, checked and unchecked conversions, default and zero handling, identity validation, aggregation, decompression, group operations, subgroup checks, multiplication, and pairing behavior. We identified that zeroization is incomplete on multiple transient buffers containing secrets (Issue B), that a full private keypair is written to a caller-controlled path (Issue C), and that hashes in generated proof of possessions are not guaranteed to include the associated public key, resulting in potentially forgeable proofs (Issue A).

For `solana-bls-signatures`, we further mapped the implementation against the specification in [BGW+22] and RFC [FSS+23] and ran the relevant test suites while tracking issues and test coverage gaps encountered during the audit. We also identified factually incorrect documentation regarding the capabilities of the verification algorithms (Issue D).

Dependencies

Our team did not identify any security issues in `solana-bls-signatures` and `solana-bls12-381-syscall`'s use of dependencies.

Code Quality

We performed a manual review of the repositories in scope and found the code to be well organized and closely aligned with development best practices.

Tests

Coverage metrics indicate strong test coverage.

For `bls-signatures`, coverage is as follows:

- Region coverage: 86.97%
- Line coverage: 84.53%
- Function coverage: 72.93%

For `bls12-381-syscall`, coverage is as follows:

- Region coverage: 98.64%
- Line coverage: 99.47%
- Function coverage: 100.00%

However, BLS logic is not tested against test vectors (Suggestion 1).

Documentation and Code Comments

The project documentation clearly outlines the project's purpose and sufficiently describes the system's intended functionality. However, we identified one issue where the documentation stated nonexistent properties, specifically batch verification, of the implemented verification algorithms (Issue D).

In addition, code comments explaining expected behavior could be improved. We recommend expanding code comments (Suggestion 3).

Scope

The scope of this review was sufficient and encompassed all security-critical components under the assumption that `blst` and `blstrs` function as intended.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	SEVERITY	STATUS
Issue A: Proof of Possession Does Not Necessarily Protect Against Rogue Key Attacks	High	Resolved
Issue B: Incomplete Zeroization of Transient Secret Buffers During Derivation and Serialization	Medium	Resolved
Issue C: Insecure Keypair File Write Can Expose Private Keys and Clobber Arbitrary Writable Files	Medium	Resolved
Issue D: Misleading Batch Verification Documentation	Medium	Resolved
Suggestion 1: Test BLS logic Against Test Vectors	Informational	Resolved
Suggestion 2: Clarify EIP-2537 Semantics in G1 and G2 Operation Names	Informational	Resolved
Suggestion 3: Improve Code Comments	Informational	Partially

		Resolved
Suggestion 4: Improve Documentation	Informational	Partially Resolved
Suggestion 5: Remove Default From Serialized BLS Point Wrappers	Informational	Resolved
Suggestion 6: Clarify or Seal the VerifySignature Safety Boundary	Informational	Resolved

Issue A: Proof of Possession Does Not Necessarily Protect Against Rogue Key Attacks

Location

`bls-signatures/src/secret_key.rs#L96`

`bls-signatures/src/pubkey/verify.rs#L24`

Synopsis

When `payload` is present, proof generation in `proof_of_possession` and verification in `verify_proof_of_possession` hash only `payload` but do not necessarily bind the claimed public key into the hash input. This weakens the proof from a proof of possession for a specific key into a signature over a shared message and reintroduces rogue key attacks.

Impact

High.

An attacker could make a rogue public key pass `proof_of_possession` verification and then use it to forge aggregate signatures with honest participants.

Feasibility

Medium.

The attack is practical when a deployment uses the custom-payload proof path with a payload not bound to the public key, and relies on successful proof verification to admit keys into aggregation.

Severity

High.

In affected configurations, the issue defeats the main security property that proof of possession is intended to provide.

Preconditions

The system must use `proof_of_possession(Some(payload))` where `payload` does not contain any binding to the associated public key, verify with the same shared `payload`, and treat a valid proof as sufficient protection against rogue-key attacks in aggregate signature workflows.

Technical Details

The current construction computes `PoP = sk*H(payload)`, where `payload` is not required to be bound to the associated public key, and verifies the proof as `e(pk,H(payload))==e(G,PoP)`.

Because `H(payload)` does not necessarily depend on pk, an attacker can combine a published proof for an honest key with attacker-chosen values to construct a valid proof for a rogue key without knowledge of that rogue key's secret.

To be more precise, given an honest user key `PK_A=sk_A*G` with associated proof `PoP_A=sk_A*H(payload)`, an attacker can choose a scalar `s` to define a rogue key `PK_R=s*G-PK_A` with a forged proof of possession `PoP_R=s*H(payload)-PoP_A`.

Despite the fact that no secret key `sk_R` is known for `PK_R`, the proof `PoP_R` is verified because the verification identity `e(PK_R,H(payload))==e(G, PoP_R)` can be rewritten as `e(s*G,H(payload))-e(PK_A, H(payload))==e(G,s*H(payload))-e(G,PoP_A)` using the definition of `PK_R` and `PoP_R` and the bilinearity of the pairing. Rewriting this as `s*(e(G,H(payload))-e(G,H(payload)))==-(e(G, PoP_A)-e(PK_A, H(payload)))` and using that `PoP_A` is a valid proof of possession for `PK_A`, it follows that both sides cancel to zero.

Once `PK_R` is accepted, the classic BLS multi-signature forgery follows, since `PK_A + PK_R = s*G` and the attacker alone can produce a signature on any message `m` as `sig = s*H(m)`, because that signature verifies under the aggregate key `PK_A+PK_R`, making it appear that the honest user and the rogue key jointly signed `m`.

While no production call site in the repository uses the unsafe `Some(payload)` mode outside of tests, the API permits downstream users to do so.

Remediation

We recommend that proof generation and verification hash the same `public_key`-bound input, for example, by using `H(pubkey_bytes||payload)`.

Status

The Anza team has resolved the issue by binding PoP hashes to the claimed key through hashing `payload || pubkey_bytes` during generation and verification.

Verification

Resolved.

Issue B: Incomplete Zeroization of Transient Secret Buffers During Derivation and Serialization

Location

`bls-signatures/src/secret_key.rs#L32-L46`

`bls-signatures/src/secret_key.rs#L56-L75`

`bls-signatures/src/secret_key.rs#L77-L92`

`bls-signatures/src/secret_key.rs#L131-L134`

`bls-signatures/src/keypair.rs#L122-L149`

Synopsis

The crate already zeroizes the owned `SecretKey` scalar on drop, but several helper paths still keep secret data in stack and heap buffers without explicit zeroization. Those transient copies can remain

recoverable in process memory after the operation completes, increasing exposure to memory disclosure bugs, crash dumps or swap and pagefile leakage.

Impact

High.

This issue could result in the disclosure of secret keys.

Feasibility

Low.

The attacker would need to be local or have additional vulnerabilities against the host that grant read access to the memory or filesystem.

Severity

Medium.

Preconditions

An attacker can inspect process memory through a memory disclosure primitive, core dump, crash dump, swap reading, or similar local vectors. For `derive_from_signer`, the `solana-signer-derive` feature needs to be enabled.

Technical Details

The current code zeroizes the owned scalar in `SecretKey::drop` and redacts `SecretKey` debug output. Helper code creates additional buffers that keep secrets outside that protected owner type and does not zeroize them afterward.

Affected assets:

- The temporary `blst_scalar` allocated in `SecretKey::derive` before conversion into `SecretKey`.
- The signature stored in `SecretKey::derive_from_signer` (in this API, those signature bytes are effective input key material for the derived BLS secret).
- The raw `[u8; BLS_SECRET_KEY_SIZE]` array materialized by `From<&SecretKey>`.
- The raw `[u8; BLS_KEYPAIR_SIZE]` array materialized by `From<&Keypair>`.
- The `Vec<u8>` allocated by `Keypair::read_json` to hold a serialized keypair before parsing.
- The JSON String allocated and returned by `Keypair::write_json`, which creates an additional heap copy of the private keypair.

Mitigation

We suggest minimizing the creation of copies of secret material.

Remediation

We recommend keeping secret material inside zeroizing wrappers for its full lifetime, including temporary serialization and deserialization buffers.

Status

The Anza team has addressed this issue through PR 662 and PR 702, which add zeroization for `blst_scalar`, signer signature bytes, JSON input bytes, and temporary secret-key buffers, as well as `compiler_fence` to prevent the compiler from optimizing away zeroization.

Verification

Resolved.

Issue C: Insecure Keypair File Write Can Expose Private Keys and Clobber Arbitrary Writable Files

Location

bls-signatures/src/keypair.rs#L152-L178

Synopsis

`Keypair::write_json_file` writes a full private keypair to a caller-controlled path using `create(true)` and `truncate(true)` without exclusive creation, symlink rejection, or file validation. This permits two local filesystem attacks: writing secrets into a pre-existing file and following a symlink that redirects truncation and output to another writable target.

Impact

High.

The attacker could extract the private key or use symlink redirection for path traversal attacks against the host.

Feasibility

Low.

The attacker would need to have local filesystem access, either directly or via other host vulnerabilities.

Severity

Medium.

Preconditions

For this issue to occur, the following must hold true:

- The caller must use `Keypair::write_json_file` on a path an attacker can influence or within a directory an attacker can write to.
- The attacker must have local filesystem access on the same host.

Technical Details

`Keypair::write_json_file` creates parent directories automatically and then opens `outfile` with `write(true)`, `truncate(true)`, and `create(true)`.

On Unix, mode `0o600` applies only when a new file is created. If an attacker creates `outfile` as a regular file with permissive mode bits, the function will reopen and truncate that file without tightening its permissions, leaving the serialized secret key readable after the write completes.

The open call also follows symlinks. If an attacker can place a symlink at `outfile`, the function can be redirected to truncate and overwrite a different file that the process is allowed to write to. This is primarily a file-clobbering issue, though it can also disclose the key if the redirected target is attacker-readable.

On non-Unix platforms, the file is created with default inherited ACLs, so confidentiality also depends on the surrounding filesystem policy.

Remediation

We recommend the following steps to remediate this issue:

- Using an atomic write pattern that creates a new temporary file with exclusive creation, applies restrictive permissions explicitly, verifies that the opened handle refers to a regular non-symlink file, and then renames it into place.
- Enforcing permissions after opening, if existing files must be replaced on Unix, rather than relying on `mode(0o600)` during creation.

Status

The Anza team has addressed this issue by changing `Keypair::write_json_file` to remove any preexisting target path and then reopen it with `create_new(true)`, thereby preventing overwriting of an existing file and blocking final-component symlink clobbering when writing secret key material.

Verification

Resolved.

Issue D: Misleading Batch Verification Documentation

Location

anza-xyz-solana-sdk/blob/master/bls-signatures/README.md#solana-bls-signatures

anza-xyz-solana-sdk/blob/master/bls-signatures/src/signature/mod.rs#L12

anza-xyz-solana-sdk/blob/master/bls-signatures/src/signature/mod.rs#L22

anza-xyz-solana-sdk/blob/master/bls-signatures/src/signature/mod.rs#L794

anza-xyz-solana-sdk/blob/master/bls-signatures/src/signature/mod.rs#L832

Synopsis

The public documentation describes the various `verify_distinct` functions as "batch verification," which conventionally implies the properties of Definition 1 from [CHP12]. However, the implementation provides only the weaker screening-style guarantee of Definition 2 in [CHP12]. This creates a documentation-level vulnerability because it can cause integrators to rely on a stronger security property than the API actually provides.

Impact

Low.

Callers may incorrectly use a successful `verify_distinct*` result as evidence that every listed signer submitted a valid individual signature. In systems where that result drives signer attribution, rewards, penalties, or any other acceptance of per-signer attestations, an attacker could cause invalid individual submissions to be treated as valid.

Feasibility

High.

If an application relies on the documented “batch verification” guarantee, an attacker can exploit the mismatch by submitting repeated messages and attacker-controlled keys whose contributions cancel within a message bucket, allowing `verify_distinct*` to accept even though the submitted per-signer signatures are not individually valid. This requires no cryptanalytic break and only the standard use of valid keys and proofs of possession.

Severity

Medium.

Preconditions

A caller must rely on the crate documentation as describing Definition 1-style batch verification and must use `verify_distinct*` in a setting where per-submitted-signature validity matters.

Technical Details

The documentation describes support for “batch verification” of distinct-message tuples, where batch verification is technically defined such that a valid batch signature implies valid individual batched signatures (Definition 1 in [CHP12]). However, `verify_distinct*` internally groups equal message hashes and verifies only the resulting aggregate relation for each bucket. This behavior is consistent with screening, not full batch verification. A caller reading the current documentation can therefore infer a stronger guarantee than the code provides.

Remediation

We recommend updating the documentation to explicitly state that `verify_distinct*` provides `screening` rather than proper `batch_verification`.

Status

The Anza team has updated the public documentation, API documentation, benchmark labels, and Ethereum vector test comments to consistently describe `verify_distinct*` as aggregate screening rather than batch verification. The documentation now explicitly states that a successful `verify_distinct*` result does not imply individual per-signer signature validity and does not provide CHP12-style weighted batch verification guarantees.

The only remaining use of `batch_verify` is for the Ethereum test-vector directory name, and the surrounding comments explain that this is Ethereum’s fixture-suite name rather than terminology used to describe this crate’s `verify_distinct*` behavior.

Verification

Resolved.

Suggestions

Suggestion 1: Test BLS logic Against Test Vectors

Location

`anza-xyz-agave/blob/master/bls12-381-syscall`

Synopsis

There are no tests using test vectors for the BLS logic. The official draft does not currently provide test vectors. Other sources exist, such as the Ethereum repository.

Mitigation

We recommend consulting and implementing tests from the Ethereum repository.

Status

The Anza team has addressed this issue by fetching Ethereum BLS vectors and exercising the Rust implementation against them.

Verification

Resolved.

Suggestion 2: Clarify EIP-2537 Semantics in G1 and G2 Operation Names

Location

`bls12-381-syscall/src/lib.rs#L8`

`bls12-381-syscall/src/addition.rs#L9`

`bls12-381-syscall/src/subtraction.rs#L10`

`EIPS-2537`

Synopsis

The documentation, function names, and return types of `bls12_381_g1/2_addition` and `bls12_381_g1/2_subtraction` suggest that these operations are performed only on points in the prime-order groups G1 and G2. However, the implementation follows EIP-2537 and therefore operates on any valid on-curve points without enforcing subgroup membership. As a result, the naming can be misleading, especially because this deviation from the subgroup interpretation is not stated explicitly.

Mitigation

We recommend adding a short API note and code comment above the G1 and G2 addition and subtraction functions to clarify that the naming follows EIP-2537 and that these operations intentionally do not perform subgroup checks. We also recommend updating the documentation.

Status

The Anza team has addressed this issue by renaming the G1/G2 add/sub APIs and syscall call sites to `_unchecked`, while having validation and encoding paths document subgroup-checked versus unchecked behavior.

Verification

Resolved.

Suggestion 3: Improve Code Comments

Location
Examples (non-exhaustive):

anza-xyz-agave/blob/master/bls12-381-syscall/src/lib.rs#L11

Synopsis
There are insufficient and sometimes misleading code comments explaining the rationale behind certain critical lines of code. This reduces the readability of the code and, as a result, makes reasoning about the security of the system more difficult. Comprehensive in-line documentation, including references to implemented standards and reference papers, helps provide reviewers of the code with a better understanding of the system design and a greater ability to reason about it.

Mitigation
We recommend expanding and improving the code comments to facilitate reasoning about the security properties of the system.

Status
The Anza team has addressed this issue in part by fixing limited documentation errors. However, this change does not address the broader comment-rationale recommendation, and the current decompression comments still appear swapped or misleading at bls12-381-syscall/src/decompression.rs#L19-L20 and bls12-381-syscall/src/decompression.rs#L52-L53.

Verification
Partially Resolved.

Suggestion 4: Improve Documentation

Synopsis
The general documentation provided by the Anza team was minimal. Robust and comprehensive documentation allows a security team to assess the in-scope components and understand the expected behavior of the system being audited. In addition, clear and concise user documentation provides users with a guide to utilize the application according to security best practices.

Mitigation
We recommend that the Anza team improve the audited crate's general documentation by creating a high-level description of the implemented capabilities and the cryptographic guarantees provided. This can include developer documentation, architectural diagrams, as well as implemented standards and reference algorithms.

Status
The Anza team has stated that it maintains a continuous process for improving documentation.

Verification
Partially Resolved.

Suggestion 5: Remove Default From Serialized BLS Point Wrappers

Location

anza-xyz-solana-sdk/bls-signatures/src/pubkey/bytes.rs#L42

anza-xyz-solana-sdk/bls-signatures/src/pubkey/bytes.rs#L72

anza-xyz-solana-sdk/bls-signatures/src/signature/bytes.rs#L39

anza-xyz-solana-sdk/bls-signatures/src/signature/bytes.rs#L69

anza-xyz-solana-sdk/bls-signatures/src/proof_of_possession/bytes.rs#L42

anza-xyz-solana-sdk/bls-signatures/src/proof_of_possession/bytes.rs#L75

Synopsis

Default implementations return all-zero byte arrays for a serialized curve-point wrapper. However, those zero arrays are not canonical encodings of the represented BLS elements, so `::default()` can create values that are not actually valid inhabitants of the type in which they are implemented. This is most evident for public keys, where identity decoding is explicitly rejected, but the same type-level mismatch also applies to signatures and proofs of possession because their valid identity encodings are distinct from zero bytes.

Mitigation

We recommend either redefining the `Default` implementations so that they return canonical encodings of a valid element of the wrapped type or removing `Default` entirely for these wrappers if no natural default exists.

Status

The Anza team has addressed this issue by removing the zero-array `Default` implementations from all affected serialized BLS point wrappers, including `Pubkey`, `Signature`, and `ProofOfPossession` variants.

Verification

Resolved.

Suggestion 6: Clarify or Seal the VerifySignature Safety Boundary

Location

anza-xyz-solana-sdk/blob/master/bls-signatures/src/pubkey/verify.rs#L67

anza-xyz-solana-sdk/blob/master/bls-signatures/src/macros.rs#L559

anza-xyz-solana-sdk/blob/master/bls-signatures/README.md#L120

Synopsis

The crate's intended safety model relies on signature verification being performed through `PopVerified<T>` or aggregate wrappers, but `VerifySignature` remains a public trait that downstream code can implement for raw key types. As a result, the boundary is advisory rather than fully enforced, and the documentation suggests stronger guarantees than the API provides.

Mitigation

We recommend that current users avoid implementing `VerifySignature` for unverified public key types and continue verifying PoP before exposing verification or aggregation flows. If stronger enforcement is desired, the crate can either seal the trait or clarify in the documentation that this constitutes a convention rather than a hard type-system guarantee.

Status

The Anza team has addressed this suggestion by clarifying the `VerifySignature` safety boundary. The documentation now states that `VerifySignature` remains a public extension trait, that the crate-provided implementations are limited to `PopVerified<T>` and aggregate public-key wrappers, and that downstream implementations for unverified key wrappers opt out of the crate's PoP-verified usage convention.

The `README` and API documentation now explicitly describe this as a convention boundary rather than a hard type-system guarantee and advise against implementing `VerifySignature` for unverified key wrappers in code paths that rely on PoP validation, aggregation safety, signer attribution, rewards, or penalties.

Verification

Resolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.