Stacks WSTS

Security Assessment

February 4th, 2025 — Prepared by OtterSec

Tuyết Dương	tuyet@osec.io
Himanshu Sheoran	deuterium@osec.io
Nakul Choudhary	quasar@osec.io
Robert Chen	r@osec.io

Table of Contents

TABLE OF CONTENTS

01 — Executive Summary

Overview

Trust Machines engaged OtterSec to assess the `wsts` program. This assessment was conducted between January 21st and January 31st, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 12 findings throughout this audit engagement.

In particular, we identified several high vulnerabilities, including an absence of verification to ensure that the key IDs provided in the nonce gathering round and the signature share verification phase are consistent, allowing malicious parties to provide fewer key IDs in their signature shares, resulting in invalid signatures (OS-STS-ADV-00).

Furthermore, malicious parties may send fewer private shares than expected, resulting in timeouts and incorrect threshold evaluations, which may allow the protocol to proceed despite missing the required share count (OS-STS-ADV-07), and there is a lack of validation to ensure that the number of keys per participant is greater than or equal to the number of participating parties, enabling parties without associated keys to participate in the protocol, performing actions such as sending nonces (OS-STS-ADV-06).

Additionally, The FIRE algorithm's utilization of signer key IDs may allow zero key ID values during signer initialization and private share distribution, risking exposure of private signing keys associated with such IDs (OS-STS-ADV-08).

We also advised to ensure adherence to the FROST standard, as in the current implementation, the omission of hashing the group public key when computing the binding factor deviates from the FROST standard (OS-STS-SUG-02). Lastly, we provided suggestions to address the possibility of batching private share verifications, allowing an adversary to manipulate the shares such that their sum appears valid, even if the individual shares themselves are invalid (**??**.

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/Trust-Machines/wsts. This audit was performed against 612c023.

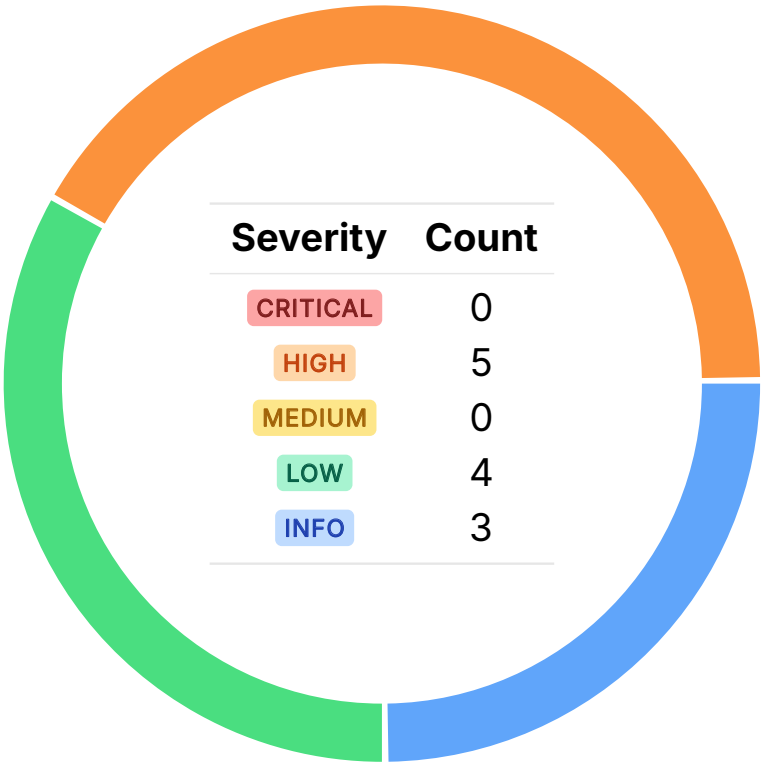
A brief description of the program is as follows:

Name	Description
wsts	A system for creating Weighted Schnorr Threshold Signatures (WileyProofs). It enables a group of signers, each controlling a set of keys, to produce a valid Schnorr signature, provided that at least T (the threshold) signers act honestly.

03 — Findings

Overall, we reported 12 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	5
MEDIUM	0
LOW	4
INFO	3

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-STS-ADV-00	HIGH	RESOLVED	There is no verification to ensure that the `key_ids` provided in the nonce gathering round and in the signature share verification are consistent, allowing malicious parties to provide fewer `key_ids` in their signature shares, resulting in invalid signatures.
OS-STS-ADV-01	HIGH	RESOLVED	The absence of key ID hashing in the `SignatureShareResponse` message allows for potential manipulation of key IDs without detection, compromising the integrity of the signature sharing process.
OS-STS-ADV-02	HIGH	RESOLVED	`gather_nonces` in the FROST implementation lacks validation for the `key_ids` field, allowing malicious parties to provide out-of-range values, resulting in the failure of signature share verification.
OS-STS-ADV-03	HIGH	RESOLVED	A malicious party may submit a public share utilizing another signer's party ID, overwriting the original commitment. This results in a failure of verification in `compute_secret`, falsely marking the legitimate signer as malicious.
OS-STS-ADV-04	HIGH	RESOLVED	A malicious party may manipulate `src_id` to overwrite legitimate private shares and select public and private shares that pass verification in `compute_secret` to replace the actual secret shares with their chosen values, thereby compromising the DKG process.

OS-STS-ADV-05	LOW	RESOLVED ✓	`dkg_ended` fails to properly handle empty `comms` in `dkg_public_shares` and empty shares in `dkg_private_shares`, allowing missing shares to go undetected.
OS-STS-ADV-06	LOW	RESOLVED ✓	There is no check to ensure that $n_k \geq n_p$, allowing parties without associated keys to participate in the protocol and perform actions such as sending nonces.
OS-STS-ADV-07	LOW	RESOLVED ✓	Malicious parties may send fewer private shares than expected, resulting in timeouts and incorrect threshold evaluations that may allow the protocol to proceed despite missing the required share count.
OS-STS-ADV-08	LOW	RESOLVED ✓	The FIRE algorithm's utilization of `signer_key_ids` may allow zero `key_id` values during signer initialization and private share distribution, risking exposure of private signing keys associated with those IDs.

Invalid Signatures Due to Mismatch in Key IDs HIGH OS-STS-ADV-00

Description

`FIRE::gather_sig_shares` does not enforce a strict check to ensure that the key IDs included in the signature share (`sig_shares[i].key_ids`) match those collected during the nonce gathering round. In the nonce gathering round, the coordinator collects `key_ids` from each participant as part of the nonce gathering response. When the coordinator then proceeds to gather the signature shares from parties, it assumes that the structure of the share (`sig_share.key_ids`) matches what was initially sent during the nonce gathering.

>_ wsts/src/state$_{machine/coordinator/fire.rs}$ RUST

```rust
fn gather_sig_shares(
    &mut self,
    packet: &Packet,
    signature_type: SignatureType,
) -> Result<(), Error> {
    if let Message::SignatureShareResponse(sig_share_response) = &packet.msg {
        [...]
        if response_info.sign_wait_signer_ids.contains(&sig_share_response.signer_id)
        {
            response_info.sign_wait_signer_ids.remove(&sig_share_response.signer_id);
            for sig_share in &sig_share_response.signature_shares {
                for key_id in &sig_share.key_ids {
                    response_info.sign_recv_key_ids.insert(*key_id);
                }
            } [...]
        } [...]
    }
    [...]
}
```

The issue arises because the coordinator does not verify if the `key_ids` in the `sig_shares` (signature shares) match those collected during the nonce-gathering phase. In `check_signature_shares` , it utilizes `sig_shares[i].key_ids` from the signature shares, allowing the check to incorrectly pass. Consequently, an invalid signature may be generated without detecting malicious participants, even if the total number of gathered signature shares is below the required threshold.

Remediation

Ensure that the `sig_shares[i].key_ids` correspond directly to the `key_ids` collected during the nonce gathering phase. This explicit comparison ensures that both rounds maintain consistent key sets.

Patch

Fixed in PR#130.

Lack of Inclusion of Key ID in Signature Hash HIGH OS-STS-ADV-01

Description

In the current implementation of `net`, the `key_ids` from the `signature_share` mapping are not included in the hash calculation in the `SignatureShareResponse` message. This opens the door for the `key_ids` to be tampered with without detection by the coordinator, which undermines the integrity of the signing process. If a malicious participant changes the `key_id`, it may result in the coordinator associating a signature share with the wrong key.

>_ *src/net.rs* RUST

```rust
impl Signable for SignatureShareResponse {
    fn hash(&self, hasher: &mut Sha256) {
        hasher.update("SIGNATURE_SHARE_RESPONSE".as_bytes());
        hasher.update(self.dkg_id.to_be_bytes());
        hasher.update(self.sign_id.to_be_bytes());
        hasher.update(self.signer_id.to_be_bytes());

        for signature_share in &self.signature_shares {
            hasher.update(signature_share.id.to_be_bytes());
            hasher.update(signature_share.z_i.to_bytes());
        }
    }
}
```

The integrity of the cryptographic signing process relies on the assumption that each participant is contributing shares corresponding to their own key. Without including `key_id` in the hash, it will not be possible to ensure that signature share messages are authenticated.

Remediation

Ensure that the `key_id` is included in the hash calculation of the `SignatureShareResponse` message.

Patch

Fixed in PR#130.

Out of Range Key IDS in FROST HIGH OS-STS-ADV-02

Description

The FROST implementation lacks validation for the `key_ids` field in the `nonce_response` during the `gather_nonces` phase. The valid range for `key_ids` should be `[1, num_keys + 1]`, where `num_keys` is the total number of participant keys. Without this validation, a malicious party could submit `key_ids` outside this expected range.

>_ *wsts/src/state_machine/coordinator/fire.rs* RUST

```rust
fn gather_nonces([...]) -> Result<(), Error> {
    if let Message::NonceResponse(nonce_response) = &packet.msg {
        if nonce_response.dkg_id != self.current_dkg_id {
            return Err(Error::BadDkgId(nonce_response.dkg_id, self.current_dkg_id));
        }
        if nonce_response.sign_id != self.current_sign_id {
            return Err(Error::BadSignId(
                nonce_response.sign_id,
                self.current_sign_id,
            ));
        }
        if nonce_response.sign_iter_id != self.current_sign_iter_id {
            return Err(Error::BadSignIterId(
                nonce_response.sign_iter_id,
                self.current_sign_iter_id,
            ));
        }
       [...]
    }
    [...]
}
```

Consequently, when the `key_ids` are utilized in `gather_sig_shares` to identify the keys associated with each received signature share and perform verification against the aggregate signature, the verification process will fail.

Remediation

Ensure that all `key_ids` in the received `nonce_response` are within the expected range (`[1, num_keys + 1]`).

Patch

Fixed in PR#130.

Possibility of Overwriting Public Share HIGH OS-STS-ADV-03

Description

`dkg_ended` in the signer module is responsible for handling the finalization of the distributed key generation phase and storing the received public commitments (`public_shares`). `v2::compute_secret` later verifies these commitments against the received private shares. `dkg_ended` checks if `comm` (the public commitment) is valid via `check_public_shares(comm, threshold)` . If the check fails, the signer (`signer_id`) is marked as malicious.

>_ *src/state_machine/signer/mod.rs* RUST

```rust
pub fn dkg_ended<R: RngCore + CryptoRng>(&mut self, rng: &mut R) -> Result<Message, Error> {
    [...]
    for signer_id in &signer_ids_set {
        if let Some(shares) = self.dkg_public_shares.get(signer_id) {
            for (party_id, comm) in shares.comms.iter() {
                if !check_public_shares(comm, threshold) {
                    bad_public_shares.insert(*signer_id);
                } else {
                    self.commitments.insert(*party_id, comm.clone());
                }
            }
        }[...]
    }[...]
}
```

If the check passes, the commitment (`comm`) is stored in `self.commitments` under the key `party_id` , not `signer_id` . A malicious signer (`malicious_id`) may submit a valid public commitment but associate it with the `party_id` of an honest participant (`honest_id`). This overwrites the actual public commitment of `honest_id` in `self.commitments` , replacing it with the maliciously injected one.

>_ *src/v2.rs* RUST

```rust
pub fn compute_secret([...]) -> Result<(), DkgError> {
    [...]
        if let Some(shares) = private_shares.get(key_id) for (sender, s) in shares {
            if let Some(comm) = public_shares.get(sender) {
                if s * G != compute::poly(&compute::id(*key_id), &comm.poly)? {
                    bad_shares.push(*sender);
                }
            } [....]
        }
    [...]
}
```

Thus when `honest_id` utilizes its correct private share `s` for verification in `compute_secret`, `compute::poly(&compute::id(*key_id), &comm.poly)` will evaluate the wrong commitment, causing the check to fail and flagging the actual share owner as malicious.

Remediation

Enforce that only signer IDs are utilized, instead of utilizing `party_id` to store commitments.

Patch

Fixed in PR#130.

Malicious Share Overwrite HIGH OS-STS-ADV-04

Description

This vulnerability arises from the way private shares are stored and verified in the distributed key generation process. A malicious party may craft both public and private shares that pass verification in `compute_secret` but ultimately replace the actual private shares. In `dkg_private_shares` in the signer module, malicious parties may obtain the `src_id` in the private share, and the private shares in `decrypted_shares` will be overwritten.

>_ *src/state_machine/signer/mod.rs* RUST

```rust
pub fn dkg_private_shares<R: RngCore + CryptoRng>([...]) -> Result<Vec<Message>, Error> {
    [...]
    for (src_id, shares) in &dkg_private_shares.shares {
        let mut decrypted_shares = HashMap::new();
        for (dst_key_id, bytes) in shares {
            if key_ids.contains(dst_key_id) {
                match decrypt(&shared_secret, bytes) {
                    Ok(plain) => match Scalar::try_from(&plain[..]) {
                        Ok(s) => {
                            decrypted_shares.insert(*dst_key_id, s);
                        }[...]
                    },[...]
                }
            }
        }[...]
    }
    [...]
}
```

Thus, if a malicious signer strategically chooses public and private shares that pass the public and private shares verification in `compute_secret`, the actual shares will be replaced by those chosen by the malicious party. Consecutively, if a threshold number of signers are malicious and they all inject manipulated shares, the threshold security is compromised.

Remediation

Ensure that once a valid private share is assigned to `dst_key_id`, overwriting it is restricted.

Patch

Fixed in PR#130.

Improper Validation of Empty Shares Vectors LOW OS-STS-ADV-05

Description

In `dkg_ended` in the signer module, while the `dkg_public_shares` and `dkg_private_shares` for a given `signer_id` contain the required data, if the vectors are empty (the vector of `comms` in `dkg_public_shares` or the `shares` vector in `dkg_private_shares`), no missing shares are flagged as missing. Thus, these missing private or public shares will not be added to the respective `missing_public_shares` and `missing_private_shares` vectors.

>_ src/state_machine/signer/mod.rs RUST

```rust
pub fn dkg_ended<R: RngCore + CryptoRng>(&mut self, rng: &mut R) -> Result<Message, Error> {
    [...]
    for signer_id in &signer_ids_set {
        if let Some(shares) = self.dkg_public_shares.get(signer_id) {
            for (party_id, comm) in shares.comms.iter() {
                if !check_public_shares(comm, threshold) {
                    bad_public_shares.insert(*signer_id);
                } else {
                    self.commitments.insert(*party_id, comm.clone());
                }
            }
        } else {
            missing_public_shares.insert(*signer_id);
        }
        if let Some(shares) = self.dkg_private_shares.get(signer_id) {
            for dst_key_id in self.signer.get_key_ids() {
                for (_src_key_id, shares) in &shares.shares {
                    if shares.get(&dst_key_id).is_none() {
                        missing_private_shares.insert(*signer_id);
                    }
                }
            }
        } else {
            missing_private_shares.insert(*signer_id);
        }
    }
    [...]
}
```

If the `comms` vector (which contains public commitments) is empty, the loop (`for (party_id, comm) in shares.comms.iter()`) will not execute. As a result, the code does not check if any public shares are missing. Similarly, if the `shares` vector is empty, missing private shares are not flagged. Thus, the absence of public or private shares is not flagged as a problem. As a result, the DKG process continues without detecting that a signer has not properly contributed the shares.

Remediation

Explicitly check for empty `comms` or `shares` and add the missing private or public shares to `missing_public_shares` and `missing_private_shares` respectively.

Patch

Fixed in PR#130.

Protocol Participation Without Associated Keys LOW OS-STS-ADV-06

Description

The vulnerability arises from a missing validation in the protocol, specifically failing to ensure that the number of keys per participant (n_k) is greater than or equal to the number of participating parties (n_p). If the protocol allows more participants than the available keys ($n_p > n_k$), it effectively enables parties without valid key associations to join and interact with the protocol, compromising the signature generation process.

Remediation

Validate that $n_k \geq n_p$ to ensure cryptographic integrity of the signing process.

Patch

Fixed in PR#130.

Threshold Manipulation via Incorrect Submission of Key IDs LOW OS-STS-ADV-07

Description

The protocol calculates a threshold (`dkg_threshold`) based on the expected total number of private shares derived from the `key_ids` in the configuration. The size of gathered shares (`dkg_size`) is determined via `compute_dkg_private_size`, which aggregates valid private shares from participants. However, malicious participants may manipulate this process by selectively submitting fewer private shares, limiting their submissions to only a subset of the `key_ids` they control. Thus, if sufficient private shares are not gathered within the allowed time frame, the protocol may timeout, fail to meet the threshold, and erroneously proceed to the next phase with an invalid signature.

>_ wsts/src/state_machine/coordinator/fire.rs RUST

```rust
pub fn process_timeout(&mut self) -> Result<(Option<Packet>, Option<OperationResult>), Error> {
    [...]
        if now.duration_since(start) > timeout {
            // check dkg_threshold to determine if we can continue
            let dkg_size = self.compute_dkg_private_size();
            if self.config.dkg_threshold > dkg_size {
                [...]
                let wait =self.dkg_wait_signer_ids.iter().copied().collect();
                return Ok((
                    None,
                Some(OperationResult::DkgError(DkgError::DkgPrivateTimeout(
                        wait,
                    ))),
                ));
            } [...]
        }
    [...]
}
```

Remediation

During `compute_dkg_private_size`, verify that all `key_ids` listed in the previous `DkgPublicDistribute` phase have corresponding private shares in the current `DkgPrivateGather` phase.

Patch

Fixed in PR#130.

Failure to Filter Invalid Key ID LOW OS-STS-ADV-08

Description

The FIRE algorithm retrieves `active_key_ids` from the `config.signer_key_ids` mapping. The `config.signer_key_ids` is expected to provide a mapping from `signer_id` (a unique signer identifier) to the list of key IDs (`u32`) associated with that participant. If the `config.signer_key_ids` for any `signer_id` contains a zero key ID, it will be included in the `active_key_ids` list because the function does not filter these keys. Additionally, the signer also doesn't check if `key_id` is in the range when it is initialized.

>_ *wsts/src/state_machine/signer/mod.rs* RUST

```rust
        if threshold == 0 || threshold > total_keys {
            return Err(Error::InvalidThreshold);
        }

        if dkg_threshold == 0 || dkg_threshold < threshold {
            return Err(Error::InvalidThreshold);
        }

        let signer = SignerType::new(
            signer_id,
            &key_ids,
            total_signers,
            total_keys,
            threshold,
            rng,
        );
```

Remediation

Ensure that all key IDs are in a specific range.

Patch

Fixed in PR#130.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-STS-SUG-00	Batching private share verifications allows an adversary to manipulate the shares such that their sum appears valid, even if the individual shares themselves are invalid.
OS-STS-SUG-01	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.
OS-STS-SUG-02	The omission of hashing the group public key when computing the binding factor deviates from the FROST standard.

Batch Verification Bypass OS-STS-SUG-00

Description

The vulnerability in the batch verification of private shares in `v1::compute_secret` relates to the possibility of an adversary manipulating the private shares to pass the batch verification process even though the shares are invalid. The issue lies in how batch verification of private shares is performed using multi-scalar multiplication. The code combines the sharers as following : $-f_{j1}(i)G + P_{j1}(i) - f_{j2}(i)G + P_{j2}(i) = 0$. Thus, it batches the sharers into a single multi-scalar multiplication: $\sum_j(-f_j(i)G + P_j(i)) = 0$, assuming the shares are valid if their sum is zero.

>_ *src/v1.rs* RUST

```rust
/// Compute this party's share of the group secret key
pub fn compute_secret(
    &mut self,
    private_shares: HashMap<u32, Scalar>,
    public_shares: &HashMap<u32, PolyCommitment>,
) -> Result<(), DkgError> {
    [...]

    // building a vector of scalars and points from public poly evaluations and expectedvalues
        ↪ takes too much memory
    // instead make an object which implements p256k1 MultiMult trait, using theexisting powers
        ↪ of x and shares
    let mut check_shares =
        CheckPrivateShares::new(self.id(), &private_shares, public_shares.clone());

    // if the batch verify fails then check them one by one and find the bad ones
    if Point::multimult_trait(&mut check_shares)? != Point::zero() {
        [...]
        return Err(DkgError::BadPrivateShares(bad_shares));
    }
    [...]
}
```

This method allows an adversary to craft private shares such that $-f_{j1}(i)G + P_{j1}(i) = x$ and $-f_{j2}(i)G + P_{j2}(i) = -x$. Consequently, when performing batching verification of these private shares via summation, the end result will come out to be zero as the shares cancel each other, thereby passing the validation check. Therefore, the batch verification incorrectly accepts the adversarial shares, even though each individual share is incorrect.

Remediation

We suggest multiplying each term in the batch by a random scalar to prevent this attack.

Code Maturity OS-STS-SUG-01

Description

1. Mark parties with missing public shares as malicious to strengthen the robustness of the FIRE algorithm. Currently, the function merely returns a failure message (`DkgFailure::MissingPublicShares`) in `dkg_ended`, without taking punitive action against the non-compliant participants.
2. To ensure the integrity of the `PublicNonce`, add validation checks to make sure that `D` and `E` are not zero.

>_ *wsts/src/common.rs* RUST

```rust
#[derive(Clone, Debug, Eq, PartialEq, Deserialize, Serialize)]
#[allow(non_snake_case)]
/// A commitment to the private nonce
pub struct PublicNonce {
    /// A commitment to the private nonce's first value
    pub D: Point,
    /// A commitment to the private nonce's second value
    pub E: Point,
}
```

Remediation

Implement the above-mentioned suggestions.

Deviation from FROST Standards OS-STS-SUG-02

Description

A potential deviation from the FROST standard (as outlined in this link) exists in the computation of the binding factor (ρ). According to the standard, the group public key should be included in the hash when computing the binding factor. While the security impact of this omission is unclear, aligning with the standard would be advisable.

Remediation

Ensure adherence to the FROST Standard.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.