Optimism Safe Extensions
Competition

June 6, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Optimism is a Collective of companies, communities, and citizens working together to reward public goods and build a sustainable future for Ethereum.

From May 6th to May 10th Cantina hosted a competition based on safe-extensions. The participants identified a total of **143** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 6
- Low Risk: 93
- Gas Optimizations: 0
- Informational: 44

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 Medium Risk

3.1.1 An owner can be censored by another owner with a lower address

Submitted by zigtur, also found by r0bert, Jeiwan, Niroh, Jonatas Martins, ZdravkoHr, sammy, crypticdefense, 0xhuy0512, Rotciv Egaf, trachev, nmirchev8, 0xforge, bronzepickaxe, KupiaSec, elhaj, cyber, 0xleadwizard, 0x73696d616f, Aamirusmani1552, Mahmud and 99Crits

Severity: Medium Risk

Context: LivenessGuard.sol#L110-L117

Description: According to the specs, it is known that:

> 3. When a transaction is executed, the signatures on that transaction are passed to the guard and used to identify the signers. If more than the required number of signatures is provided, they are ignored.

By combining the fact that providing more signatures than the required number AND that the signatures must be ordered ascendingly by the corresponding owner addresses, a malicious owner can target the last owners with high addresses to bypass the liveness increase with signature.

Impact: Medium as the "victim owner" can be considered as down even if he participated in the protocol, leading to removing him from the owners list without expecting it.

Likelihood: High as the "victim owner" don't expect being banned while taking part in the signature process, so he will not call `showLiveness()`.

Moreover, the "attacker owner" only needs an address lower than the victim one.

Proof of concept: Let's say that we have a 10/13 Safe. We define:

- 6 owners have an address lower than the attacker one (from `owner1 to` owner6').
- "Attacker owner" = `0x7777777777777777777777777777777777777777` (also called `owner7`).
- 5 owners have an address greater than the attacker one and lower than the victim one (from `owner8` to `owner12`).
- "Victim owner" = `0xEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE` (also called `owner13`).

An attacker owner always monitor the current state of proposals to define how many owners signed it. He waits for it to be 10/13 and look if the victim owner has signed. Let's say that attacker, `owner11` and `owner12` haven't signed, the signatures look like:

```
[ sig_owner1, sig_owner2, sig_owner3, sig_owner4, sig_owner5, sig_owner6, sig_owner8, sig_owner9, sig_owner10,
↪  sig_victim]
```

If the victim owner has signed, then the attacker signs the proposal too and make it a 11/13 proposal. Then, the signature array look like:

```
[ sig_owner1, sig_owner2, sig_owner3, sig_owner4, sig_owner5, sig_owner6, sig_ATTACKER, sig_owner8,
↪  sig_owner9, sig_owner10, sig_victim]
```

Since the `Safe.checkNSignatures` requires signatures to be ordered ascendingly by the owner address and only checks the required number (here 10), the victim owner signature is always ignored:

```
function checkNSignatures(bytes32 dataHash, bytes memory data, bytes memory signatures, uint256
↪  requiredSignatures) public view {
    // ...

    // There cannot be an owner with address 0.
    address lastOwner = address(0);
    address currentOwner;

    // ...

    for (i = 0; i < requiredSignatures; i++) {

        // ...

        require(currentOwner > lastOwner && owners[currentOwner] != address(0) && currentOwner !=
↪  SENTINEL_OWNERS, "GS026");
        lastOwner = currentOwner;
    }
}
```

Because the `LivenessGuard.checkTransaction` function uses the `SafeSigners.getNSigners` that also ignores the signatures with index greater than the required threshold, the victim owner address will not be retrieved and so its liveness will not be updated.

```
/// @notice Records the most recent time which any owner has signed a transaction.
/// @dev Called by the Safe contract before execution of a transaction.
function checkTransaction(
    address to,
    uint256 value,
    bytes memory data,
    Enum.Operation operation,
    uint256 safeTxGas,
    uint256 baseGas,
    uint256 gasPrice,
    address gasToken,
    address payable refundReceiver,
    bytes memory signatures,
    address msgSender
)
    external
{
    /// ...

    uint256 threshold = SAFE.getThreshold();
    address[] memory signers =
        SafeSigners.getNSigners({ dataHash: txHash, signatures: signatures, requiredSignatures: threshold });

    for (uint256 i = 0; i < signers.length; i++) { // @POC: signers length is 10, not 11. So victim address is
↪  ignored.
        lastLive[signers[i]] = block.timestamp;
        emit OwnerRecorded(signers[i]);
    }
}
```

Recommendation: Consider incrementing the liveness of **all signers**.

However, the signers for which the signature wasn't verified by the Safe must be checked to ensure that they are legit owners (otherwise, any signer would see liveness updated even if not an owner of the safe).

3.1.2 The acceptance of a new `safe.changeThreshold` in the `livenessModule` depends incorrectly on liveness updates of owners

Submitted by Manuel Polzhofer, also found by ladboy233, Aamirusmani1552, ZdravkoHr, Al-Qa-qa and 99Crits

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: In the current design the `LivenessModule` stores a `THRESHOLD_PERCENTAGE` as an immutable variable. The `SAFE` itself stores an absolute `threshold`. The owners of the `SAFE` could change the threshold at any time by calling `SAFE.changeThreshold`.

This newly `threshold` would be incorrectly overwritten in case a `LivenessModule.removeOwners` call happens based on the `THRESHOLD_PERCENTAGE`, which is not correct anymore. The `threshold` would be adjusted based on the `THRESHOLD_PERCENTAGE` and the newly amount of owners but doesn't consider, that the `threshold` could have been changed by the `SAFE` itself.

Example:

- Safe:
  - Owners: 13
  - Threshold: 10
- `LivenessModule`:
  - `THRESHOLD_PERCENTAGE` = 75

If the safe owners now change the required threshold to `5`. The current behavior would be as long as liveliness updates happen the `SAFE.threshold` would stay `5`. If an owner is not live anymore the `LivenessModule` would remove one owner (from 13 to 12) and change the threshold back to 9 (75%).

This behavior doesn't make sense. Either the `threshold` is controlled by the `LivenessModule` or by `SAFE`. Changing it only based on the liveness of the owner shouldn't be the case.

Recommendation: Consider one of the following options:

1. `safe.changeThreshold` not allowed by the `LivenessGuard`. `LivenessModule` controls the `threshold`. It is only adjust in the event of `removeOwners` to keep it align with the stored percentage threshold. The `SAFE` can not change the threshold.
2. `safe.getThreshold` is the source of truth. The `thresholdPercentage` is moved to the `LivenessGard`. The guard detects a `changeThreshold` transaction and updates the `thresholdPercentage` accordingly. The `LivenessModule` gets the threshold from the `LivenessGuard`.
3. Safe can only change the threshold by calling the `LivenessModule`. The `LivenessModule` would get a new method called `changeThreshold` with the `threshold` and the `percentageThreshold` as parameters. Only the `SAFE` is allowed to call this method. The `safe.changeThreshold` with only the `threshold` as parameter would be blocked by the `LivenessGuard`.

3.1.3 Removing owners via `livenessModule` does not update the guard `lastlive` mapping

Submitted by ZdravkoHr, also found by r0bert, Jonatas Martins, J4X98, trachev, elhaj, Topmark and 99Crits

Severity: Medium Risk

Context: LivenessGuard.sol#L146, LivenessModule.sol#L133-L169

Description: `LivenessGuard` has a mapping where the last time a signer was active is saved.

```
mapping(address => uint256) public lastLive;
```

When an owner is removed through a normal safe transaction, this mapping is updated and the old owner's address is removed from it.

```
delete lastLive[ownerBefore];
```

However, when `LivenessModule.removeOwners()` is called, the transaction bypasses the guard and the delete logic won't be executed. This will result in a removed owner still having their `lastLive` value set.

Impact: Medium, breaks the invariant that removed owners must not be present in the `lastLive` mapping.

Likelihood: High, as it happens every time the module removes owners.

Recommendation: The same way there is a `showLiveness()` function, a `removeLiveness()` function may be introducted to the Guard:

```
function removeLiveness(address _account) external {
    require(!SAFE.isOwner(_account), "LivenessGuard: Account is owner");
    delete lastLive[_account];
}
```

It may be then called by the module after removal.

3.1.4 In case of `exectransaction()` reentrancy all owners will be marked as live

Submitted by 0xa5df, also found by KumaCrypto, lukaprini, yixxas, Manuel Polzhofer, J4X98, cyber and 99Crits

Severity: Medium Risk

Context: LivenessGuard.sol#L134

Description: Before a transaction starts (at `checkTransaction()`) the `LivenessGuard` stores the list of current owners at `ownersBefore` `EnumerableMap`, and after the transaction is executed (at `checkAfterExecution()`) it compeares the list of owners from `SAFE.getOwners()` and `ownersBefore`. Any owner present at `SAFE.getOwners()` but not at `ownersBefore` is assumed to be a new owner and marked as alive.

The issue is that in case of a transaction reentrancy (transaction B is executed while transaction A didn't finish yet) `ownersBefore` will be empty when `checkAfterExecution()` is called for the first transaction, the function would assume all owners are new owners and would mark them all as alive.

Consider the following scenario:

- The safe owners sign transaction A to send an NFT to Alice's contract.
- They also sign another transaction B (e.g. send 5K USDC to Bob).
- Alice modifies her contract so that when `onerc721received()` is called it'll execute transaction B.
- Alice executes transaction A (which then execute transaction B).
- As demonstrated above, when `checkAfterExecution()` is called for transaction A `ownersBeofre` is empty and all owners are marked as alive.

Recommendation: Either:

- Don't allow reentrancy (reentrancy lock on `checkTransaction()`).
- Allow reentrancy but keep a separate list of `ownersBefore` for each level of the reentrancy.

3.1.5 Liveness is erroneously reset for all owners when `livenessguard` is upgraded or replaced

Submitted by ethan, also found by ZdravkoHr and 0x73696d616f

Severity: Medium Risk

Context: LivenessGuard.sol#L51

Description: As a means of initializing the `lastLive` mapping, the constructor of `LivenessGuard` iterates through the Safe's owners and sets `lastLive` for each one to `block.timestamp`. However, this only makes sense the first time it is deployed: when the `LivenessGuard` needs to be upgraded or replaced in the future, this initialization will refresh the liveness of potentially inactive owners and undermine the functionality of the `LivenessModule`.

Impact: The entirety of the `LivenessModule` and `LivenessGuard`'s combined functionality is aimed at facilitating the efficient removal of inactive owners. This utility is compromised by the fact that the `LIVENESS_INTERVAL`, an ostensibly `immutable` value, could be increased without limit as a consequence of normal development and operation.

Likelihood: This is guaranteed to occur anytime the `LivenessGuard` is replaced, as long as the constructor remains unchanged.

Proof of Concept: Since a test is not necessary to demonstrate that the `LivenessGuard` resets `lastLive` for every owner of the Safe, this proof of concept will walk through what a higher-impact consequence of this vulnerability might look like in practice. But, for reference, below is the constructor code that resets each `lastLive` value:

```
address[] memory owners = _safe.getOwners();
for (uint256 i = 0; i < owners.length; i++) {
    address owner = owners[i];
    lastLive[owner] = block.timestamp;
    emit OwnerRecorded(owner);
}
```

Now, consider the following scenario:

- Five owners in a 10-of-12 Safe have been inactive for five months.
- The `LIVENESS_INTERVAL` for this Safe is six months.
- The `LivenessModule` will require a shutdown imminently.
- A bug is found in the `LivenessGuard`, necessitating an urgent replacement.
- `lastLive` is reset for all owners, including the five inactive ones, when the new `LivenessGuard` is deployed.
- The `LivenessModule` is forced to wait nearly a year in total to remove these owners, initiate a shutdown, and recover the Safe.
- This process could repeat infinitely. If the `LivenessGuard` needed an update or replacement every five months during a five year period, for example, its true `LIVENESS_INTERVAL` would be an order of magnitude greater than intended.

Recommendation: The `LivenessGuard` could optionally initialize the mapping with existing values:

```
constructor(Safe _safe, address _prevGuard) {
    SAFE = _safe;
    address[] memory owners = _safe.getOwners();
    for (uint256 i = 0; i < owners.length; i++) {
        address owner = owners[i];

        lastLive[owner] = prevGuard == address(0) ?
            block.timestamp :
            LivenessGuard(_prevGuard).lastLive(owner);

        emit OwnerRecorded(owner);
    }
}
```

3.1.6 EIP-1271 non-compliance and denial of service risk for account abstraction wallets in council safe

Submitted by elhaj, also found by Putra Laksmana, J4X98, bronzepickaxe, BoRonGod, deth and nmirchev8

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Owners using smart contract wallets (account abstraction), are facing a blocking issue when trying to sign transactions on the **Council Safe**. This is due to the use of incorrect validation logic for smart contract wallet signatures as defined in EIP1271 in the version of the contract used by the Council Safe.

The problem occurs in the `checkNSignatures` function. The contract calls the `isValidSignature` function with the wrong types of inputs.

```
require(ISignatureValidator(currentOwner).isValidSignature(
  data,
  contractSignature
) == EIP1271_MAGIC_VALUE, 'GS024')
```

The `ISignatureValidator` in the `EIP1271` takes `(bytes32 , bytes)` , while the interface used in this version of safe define it as : `(bytes,bytes)`.

This leads to diffrent function signatures and thus the diffrent (`EIP1271_MAGIC_VALUE`), so `EIP1271_MAGIC_VALUE` expected to be returned when the validation is successful is incorrectly implemented when compared to the standard defined in EIP-1271.

- `safe_magic_value` => `0x20c13b0b`
- `EIP1271_magic_value` => `0x1626ba7e`

This can lead to two major issues:

1. Owners with smart contract wallets (account abstraction) are unable to sign transactions, violating this specified property.
2. More severely, the **Council Safe** could become entirely dysfunctional. If the number of owners with smart contract wallets - `smartWallets owners` - is greater than the difference between the total number of owners - `ownersCount` - and the required number to approve a transaction - `threshold` - no transactions can be executed. This situation could arise if a smart contract wallet is added as a new owner, change of `treshold` etc...

   Moreover,The `FALLBACK_OWNER` is itself a Safe wallet, and if it adopts or upgrades to `version 1.5.0` or later of Safe , it could lead to serious issues since it uses the correct `magic_value`, (CompatibilityFallbackHandler.sol#L57-L68). In the event of a shutdown where the `FALLBACK_OWNER` becomes the sole owner of the Council Safe, With such an upgrade, the `FALLBACK_OWNER` would not be able to sign or execute transactions, resulting in a complete DoS for the Council Safe.

Recommendation: Since the contract is already deployed and can only be upgraded, the recommendation is to Upgrade the Council Safe to the version `Safe` contract that resolves the signature validation logic issue (version 1.5.0 or above) in accordance with the EIP-1271 standard. This upgrade will ensure that owners using smart contract wallets can sign transactions and the Council Safe remains fully functional.