
# Optimism — Safe Extensions

**Audited by Ethan Bennett**

**May 2024**

# Contents

**About Darklinear Solutions** 1

**Introduction** 1

**Findings Summary** 1

**Findings** 2

M-1: Liveness is erroneously reset for all owners when `LivenessGuard` is upgraded or replaced . . . . . 2

## About Darklinear Solutions

Darklinear Solutions provides unrivaled security for blockchain applications, from the bytecode to the browser. With years of experience in smart contract development and traditional software engineering, we find the bugs that others miss. Learn more at darklinear.com.

---

## Introduction

Optimism is a layer two network that leverages optimistic rollups to enable a cheaper and more efficient way to interact with Ethereum. The network's governance operates Safe multisignature wallets, and the code in the scope of this review is designed to make some of those operations more secure and efficient. This review consists of issues discovered during the course of Cantina's competitive audit for the Optimism Safe Extensions in May 2024.

The findings described below are classified according to Cantina's standards.

---

## Findings Summary

**This review ranked #4 of 227 on Cantina.**

In addition to the medium-severity vulnerability described in this version of the report, there were seven confirmed low-severity and informational findings. These findings have not yet been made public by the Optimism team, so they are currently excluded from this report. This will be updated to include them as soon as they become publicly available.

| Classification | Finding |
|---|---|
| Medium risk (M-1) | Liveness is erroneously reset for all owners when `LivenessGuard` is upgraded or replaced |

---

# Findings

## M-1: Liveness is erroneously reset for all owners when `LivenessGuard` is upgraded or replaced

**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 Explanation:** 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 Explanation:** 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);
    }
}
```