Optimism SaferSafes Security Review

Auditors

MiloTruck, Lead Security Researcher

R0bert, Lead Security Researcher

Report prepared by: Lucas Goiriz

November 13, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Optimism is a fast, stable, and scalable L2 blockchain built by Ethereum developers, for Ethereum developers. Built as a minimal extension to existing Ethereum software, Optimism's EVM-equivalent architecture scales your Ethereum apps without surprises. If it works on Ethereum, it works on Optimism at a fraction of the cost.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Optimism SaferSafes according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- 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.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 7 days in total, OP Labs engaged with Spearbit to review the Optimism protocol. The review focused on the following scope:

- `packages/contracts-bedrock/src/safe/LivenessModule2.sol`.
- `packages/contracts-bedrock/src/safe/TimelockGuard.sol`.
- `packages/contracts-bedrock/src/safe/SaferSafes.sol`.

In this period of time a total of **8** issues were found.

Summary

Project Name	OP Labs
Repository	Optimism
Commit	cb54822c
Type of Project	Vaults, Safes
Audit Timeline	Oct 26th to Nov 2nd

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	1	0
Low Risk	2	1	1
Gas Optimizations	0	0	0
Informational	5	4	1
Total	**8**	**6**	**2**

5 Findings

5.1 Medium Risk

5.1.1 Using the same signatures for `scheduleTransaction()` and `execTransaction()` introduces security risks

Severity: Medium Risk

Context: TimelockGuard.sol#L489-L494

Description: In `TimelockGuard`, the signatures used to schedule a transaction via `scheduleTransaction()` are the same as the signatures passed to Safe's `execTransaction()` for execution later on:

```
/// @dev This function validates signatures in the exact same way as the Safe's own
///      execTransaction function, meaning that the same signatures used to schedule a
///      transaction can be used to execute it later. This maintains compatibility with
///      existing signature generation tools. Owners can use any method to sign the a
///      transaction, including signing with a private key, calling the Safe's approveHash
///      function, or EIP1271 contract signatures.
```

> This function validates signatures in the exact same way as the Safe's own `execTransaction` function, meaning that the same signatures used to schedule a transaction can be used to execute it later.

However, this design introduces several security risks:

1. Whenever a Safe uninstalls `TimelockGuard`, all transactions that were previously scheduled immediately become executable, even ones that were cancelled. An attacker can simply call `Safe.execTransaction()` with the signatures that were passed to `scheduleTransaction()`, since they are the same.
2. If `TimelockGuard` is re-installed and `clearTimelockGuard()` is called, anyone can call `scheduleTransaction()` again with the same signatures to re-schedule older transactions. Practically, this only matters for cancelled transactions since transactions which are already executed cannot be executed again.
3. The signature doesn't include `address(this)`. If there are multiple `TimelockGuard` contracts (e.g. an upgraded version in the future), the same signature will work for both contracts.

The only exception to (1) and (2) are transaction hashes with nonces older than the Safe's current nonce.

Recommendation: A possible mitigation would be to sign the hash of (`txHash`, `address(this)`, `_safeConfigNonces[_safe]`) instead.

Optimism: Acknowledged. Documented this in the code in commits 954854c and c576a017. This has also been fixed by bumping the nonce during operations as specified in the following runbooks:

1. TimelockGuard Removal Runbook
2. Cancel Scheduled Transaction Runbook
3. Liveness Incident Runbook

Spearbit: Fix verified.

5.2 Low Risk

5.2.1 FREI-PI invariant can be bypassed

Severity: Low Risk

Context: SaferSafes.sol#L43-L45

Description: `SaferSafes._checkCombinedConfig` is supposed to maintain the FREI-PI invariant `livenessResponsePeriod >= 2 * timelockDelay` whenever both the timelock guard and liveness module are active. The function returns immediately if either extension is disabled:

```
if (!(_isGuardEnabled(_safe) && _safe.isModuleEnabled(address(this)))) {
    return;
}
```

`TimelockGuard.configureTimelockGuard` allows signers to raise the timelock even while the guard is disabled. Because `_checkCombinedConfig` exits early, the new delay is never checked against the existing liveness response period. Later, operators can re-enable the guard via `GuardManager.setGuard(...)`, but that pathway performs no revalidation, leaving any stale configuration in place. A Safe can therefore end up in a state with `livenessResponsePeriod < 2 * timelockDelay`.

Once the guard is re-enabled, every Safe transaction, including `respond()` from `LivenessModule2`, must pass through `TimelockGuard.checkTransaction`, which enforces the oversized delay. Owners must wait longer than the allowed response window, so they cannot execute `respond()` before `changeOwnershipToFallback` succeeds. The fallback owner retains the ability to finalize the challenge and take control even if the Safe owners are online and responsive.

Impact: The FREI-PI invariant (`livenessResponsePeriod >= 2 * timelockDelay`) can be broken. After re-enabling the guard, the inflated timelock still blocks owners from calling `respond()` before the shorter liveness window closes, so the fallback owner can always finalize the takeover.

Recommendation: Close the configuration/enable-order gap so the FREI-PI invariant cannot be bypassed. Either refuse timelock updates unless the guard is currently enabled, or re-run `_checkCombinedConfig` when the guard or module transitions from disabled to enabled, reverting if `livenessResponsePeriod < 2 * timelockDelay`.

Optimism: Fixed in commit b436d76f.

Spearbit: Fix verified.

5.2.2 `success` boolean returned by `execTransactionFromModule()` is not checked

Severity: Low Risk

Context: ModuleManager.sol#L89-L92, LivenessModule2.sol#L353-L358

Description: In Safe's `ModuleManager`, `execTransactionFromModule()` does not revert if the executed call fails. Instead, it returns a `success` boolean which indicates if the underlying call was successful:

```
// Execute transaction without further confirmations.
success = execute(to, value, data, operation, type(uint256).max);
if (success) emit ExecutionFromModuleSuccess(msg.sender);
else emit ExecutionFromModuleFailure(msg.sender);
```

However, in `LivenessModule2`, `changeOwnershipToFallback()` does not check the `success` boolean returned by `execTransactionFromModule()`. For example, when calling `setGuard()`:

```
_safe.execTransactionFromModule({
    to: address(_safe),
    value: 0,
    operation: Enum.Operation.Call,
    data: abi.encodeCall(GuardManager.setGuard, (address(0)))
});
```

This makes it possible for any of the calls to the safe in `changeOwnershipToFallback()` to silently fail.

One way this could occur is if `changeOwnershipToFallback()` is called with sufficient gas such that the underlying call to `setGuard()` runs out-of-gas, but execution in `changeOwnershipToFallback()` still completes. Note that this is unlikely to occur as the `fallbackOwner` is trusted to not exploit his own safe.

Recommendation: Check the `success` boolean returned by `execTransactionFromModule()` to ensure all calls succeeded.

Optimism: Acknowledged. Explained why we prefer not to do this in ec527e6 and 1f62006.

Spearbit: Acknowledged.

5.3 Informational

5.3.1 Fallback owner must reject Safe address

Severity: Informational

Context: LivenessModule2.sol#L176-L179

Description: `LivenessModule2.configureLivenessModule` only validates that the fallback owner is non-zero:

```
if (_config.fallbackOwner == address(0)) {
    revert LivenessModule2_InvalidFallbackOwner();
}
```

If operators accidentally configure the module with the Safe's own address, the takeover path bricks later. When the fallback owner invokes `changeOwnershipToFallback`, the module asks the Safe to execute:

```
OwnerManager.swapOwner(SENTINEL_OWNER, owners[0], _config.fallbackOwner);
```

However, the Safe enforces `require(newOwner != address(this), "GS203")`. Because `_config.fallbackOwner` equals the Safe, the swap reverts, the challenge window expires, and the module never regains liveness. Rejecting the sentinel (`address(0x1)`) is also worth considering; although that address cannot trigger the takeover because no one controls it, guarding against the Safe itself is still valuable to catch obvious misconfigurations early.

Recommendation: Extend the configuration validation to reject the Safe's own address in addition to the zero address. Optionally also screen out the sentinel owner value and addresses that already appear in the owner list to prevent other dead-end configurations.

Optimism: Fixed in commit 0e475c7e.

Spearbit: Fix verified.

5.3.2 Disabling the `LivenessModule2` can leave a stale challenge

Severity: Informational

Context: LivenessModule2.sol#L256-L257

Description: `LivenessModule2` records active challenges in `challengeStartTime[_safe]`. Once the fallback owner calls `openChallenge`, the module validates configuration with `_assertModuleEnabled` and writes `challengeStartTime[_safe] = block.timestamp;`. If Safe owners disable the module after that assignment, subsequent calls to `respond` or `changeOwnershipToFallback` revert because the module is disabled. While the module remains off, the stored start time keeps aging even though nobody can interact. When owners later re-enable the module via a signed Safe transaction that proves they are live the previously stored challenge is still present. If the response window elapsed while the module was disabled, the fallback owner can immediately call `changeOwnershipToFallback`, causing the Safe to lose ownership despite having just demonstrated liveness. The impact is an unintended takeover triggered solely by configuration sequencing rather than by actual inactivity.

Recommendation: Ensure the module clears challenge state whenever it is brought back online. One practical approach is to bundle `enableModule` and `configureLivenessModule` in the same Safe multisend so the configuration overwrites `challengeStartTime`. Alternatively, require operators to call `clearLivenessModule` immediately after disabling the module to keep state consistent.

Optimism: Documented to clear the module when disabling it in the relevant runbook.

Spearbit: Verified.

5.3.3 Relayer Safe submissions are blocked

Severity: Informational

Context: TimelockGuard.sol#L343-L348

Description: `TimelockGuard.checkTransaction` enforces that the caller is a Safe owner by evaluating `if (!callingSafe.isOwner(_msgSender())) { revert TimelockGuard_NotOwner(); }`. Because valid Safe multisig flows often rely on relayers or paymasters to relay fully signed transactions, this guard rejects those submissions purely based on `msg.sender`. Therefore, deployed Safes lose support for sponsored transactions and teams may disable the guard entirely to regain standard UX, eroding the intended protection.

Recommendation: Merely informative. If you still want to keep relayed executions working, gate the owner-only requirement behind a Safe-controlled configuration flag (default true) or add support for a Safe-managed whitelist of trusted relayer addresses so the guard can remain enabled without breaking sponsored transactions.

Optimism: This is a chosen trade-off, documented in commit de3a859e.

Spearbit: Verified.

5.3.4 Minor improvements to code and comments

Severity: Informational

Context: *(See each case below)*

Description/Recommendation:

1. TimelockGuard.sol#L232 - Typo: "overriden" should be "overridden".
2. TimelockGuard.sol#L217 - The natspec for this function is inaccurate, consider changing it to: "Internal helper to check if TimelockGuard is enabled for a Safe".
3. TimelockGuard.sol#L213-L215 - Since `_safe.getOwners()` loops through all owner addresses, it is gas inefficient. Additionally, the call could theoretically run out of gas if there are too many owners configured. Consider fetching the `ownerCount` storage variable in `OwnerManager` with `getStorageAt()` instead.
4. TimelockGuard.sol#L440 - Consider adding a `_timelockDelay != 0` check as input sanitization.

Optimism: Fixed in the following commits:

- 329ed74.
- 054b7a0.
- 6acd71c.
- 0cda697.

Spearbit: Verified.

5.3.5 Edge case in `LivenessModule2.changeOwnershipToFallback()` when `fallbackOwner` is an existing owner

Severity: Informational

Context: OwnerManager.sol#L102-L103, LivenessModule2.sol#L332-L340

Description: In Safe's `OwnerManager`, `swapOwner()` reverts when attempting to replace an old owner with an existing owner due to the following check:

```
// No duplicate owners allowed.
require(owners[newOwner] == address(0), "GS204");
```

As such, in `LivenessModule2`, it is possible for the following call in `changeOwnershipToFallback()` to fail if `owners[0] == fallbackOwner` (i.e. the fallback owner is the last owner in the Safe):

```
// No duplicate owners allowed.
require(owners[newOwner] == address(0), "GS204");
```

This is possible when `fallbackOwner` is an existing owner of the Safe, which occurs if:

- `fallbackOwner` is added as an owner.
- `changeOwnershipToFallback()` was called by `fallbackOwner` previously to take control of the safe.

This is currently not an issue since the `success` value from `execTransactionFromModule()` is not checked. Even if `swapOwner()` reverts, `changeOwnershipToFallback()` doesn't revert and the safe ends up with `fallbackOwner` as the only owner, which is the desired end result anyways. However, note that this would be an issue if the `success` value from `execTransactionFromModule()` was checked. A test to demonstrate:

```
function test_changeOwnershipToFallback_twice() external {
    // Start and execute first challenge
    vm.prank(fallbackOwner);
    livenessModule2.challenge(safeInstance.safe);

    vm.warp(block.timestamp + CHALLENGE_PERIOD + 1);
    vm.prank(fallbackOwner);
    livenessModule2.changeOwnershipToFallback(safeInstance.safe);

    // Start a new challenge (as fallback owner)
    vm.prank(fallbackOwner);
    livenessModule2.challenge(safeInstance.safe);

    // Execute second ownership transfer (this doesn't revert)
    vm.warp(livenessModule2.getChallengePeriodEnd(safeInstance.safe) + 1);
    vm.prank(fallbackOwner);
    livenessModule2.changeOwnershipToFallback(safeInstance.safe);
}
```

Recommendation: If necessary, a possible mitigation would be to check if `owners[0]` is already `fallbackOwner` and skip the call to `swapOwner()` if so.

Optimism: Acknowledged. Since this is not an issue, we prefer not to fix it. Explained in ec527e6

Spearbit: Acknowledged.