OP Stack L1 Contracts

Security Assessment

November 6, 2023

Prepared for:
John Mardlin
OP Labs

Prepared by: **Michael Colburn and Anish Naik**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Brooke Langhorne, Project Manager
brooke.langhorne@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Michael Colburn, Consultant
michael.colburn@trailofbits.com

Anish Naik, Consultant
anish.naik@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
September 29, 2023	Pre-project kickoff call
October 6, 2023	Delivery of storage layout changes summary report draft
October 6, 2023	Status update meeting #1
October 13, 2023	Delivery of storage layout changes summary report
October 16, 2023	Status update meeting #2
October 30, 2023	Status update meeting #3
November 6, 2023	Delivery of report draft
November 6, 2023	Report readout meeting

Executive Summary

Engagement Overview

OP Labs engaged Trail of Bits to review the security of a variety of changes made to its L1 contracts to support the transition of the OP Stack toward a Superchain architecture. The changes include updates to the storage layout of many critical L1 contracts, the introduction of a new contract upgrade mechanism and system for Superchain configuration management, and custom logic for the multi-signature Gnosis Safe to manage inactive signers.

A team of 2 consultants conducted the review from October 2 to November 3, 2023 for a total of 6 engineer-weeks of effort. Our testing efforts focused on ensuring that storage layout changes do not adversely affect the current or future state of any contract and that no external actor can perform an access control bypass, write to critical storage slots, initiate or veto an upgrade, remove owners from the Safe, or put the Safe in an unexpected state. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

We identified a total of six informational severity issues during this security review. TOB-OPT-MCP-1 and TOB-OPT-MCP-2 highlight minor shortcomings in the OP Labs team's PR review process, which can be improved with additional automation. We also found insufficient post-condition checking during the deployment and initialization of L1 contracts (see TOB-OPT-MCP-4). The OP Labs team should ensure that any action that is taken (e.g., a function call in a unit test or contract deployment) has the necessary post-condition checks to ensure that all state changes are tracked and validated. Additionally, we noticed a number of inconsistencies between the provided system requirements and their implementation (TOB-OPT-MCP-5 and appendix D). These inconsistencies may result in confusion on how to react to specific security incidents and, in the worst case, could lead to an incorrect response to a security incident. Similarly, we identified gaps in the security requirements for the Safe in the nominal case (when all the owners are active) as well as in addressing edge cases like the one highlighted in TOB-OPT-MCP-6.

The OP Labs team independently identified one high-severity issue during the first week of the audit (TOB-OPT-MCP-3) that would allow an attacker to reinitialize the L1 ETH/ERC-20 bridge and steal user funds. This issue was introduced as part of a workaround to simultaneously reset and initialize a previously used storage slot. We include it in this report to provide recommendations toward a new solution to support contract upgrades that allows overwriting previously used storage in a safer fashion.

Outside of the high-severity issue, the changes and additions made to the L1 contracts were thoroughly tested, both with unit and fuzz tests, and the documentation provided highlights the OP Labs team's strong understanding of their system's behavior and invariants.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that OP Labs take the following steps prior to carrying out the Superchain upgrade:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Review and update the system specification** to address the gaps and inconsistencies highlighted in TOB-OPT-MCP-5, appendix D, and TOB-OPT-MCP-6.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	0
Low	0
Informational	5
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	1
Data Validation	2
Testing	1
Undefined Behavior	2

Project Goals

The engagement was scoped to provide a security assessment of the OP Stack L1 contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are the storage layout changes made to the L1 contracts correctly tracked, and do they provide the necessary flexibility needed for parallel OP Stack-based chains?
- Do the storage layout changes made to the L1 contracts introduce any undefined behavior?
- Can an attacker use the `DelayedVetoable` contract to initiate an upgrade, veto an upgrade, initiate the delay, or execute unexpected arbitrary calldata?
- Can an attacker write to the `SuperchainConfig` contract or (un)pause the system?
- Do all the L1 contracts correctly interface with the `SuperchainConfig` contract?
- Does the L1 Forge deployment script correctly deploy and initialize the system?
- Does the L1 Forge deployment script correctly assign the right roles and permissions to the relevant actors for each system contract?
- Does the `LivenessGuard` contract correctly track the liveness of each Safe owner?
- Can the `LivenessGuard` contract be put into a state by an attacker that would cause it to consistently revert and block the execution of Safe transactions?
- Does the `LivenessModule` contract correctly interface with the `LivenessGuard` contract to remove inactive owners and assert the necessary post-conditions to prevent the Safe from being put in an unexpected state by an attacker?

Project Targets

The engagement involved a review and testing of the following target.

ethereum-optimism

Repository	https://github.com/ethereum-optimism/optimism/
Version	`cc5cf70dab4cf4a4be0213f7e602071710bfb4b1 (storage layout)`
	`59ad93676ab5b4beb7c92f4a2623a5fa5489280f (superchain)`
	`515de0eb5418a1628a92396cd257a53c35380cb5 (liveness)`
Type	Solidity
Platform	EVM

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Storage layout updates.** The L1 implementation contracts' storage layouts were updated by changing many immutable variables into storage variables. This allows multiple proxy contracts (for the various OP Stack-based chains) to point to the same implementation contract but specify their own values for that chain. We performed a manual review of these storage layout changes and investigated the following:
    - Whether the updated contracts adhered to a consistent pattern for (re)initialization and all changes to the contract storage layout were being correctly tracked for manual validation.
    - Whether the deprecated storage slots would be reusable by existing or new chains using these implementations.
    - Potential mitigations and recommendations for TOB-OPT-MCP-3 that would allow for safe storage slot overwrites during contract upgrades.
- **`SuperchainConfig`.** The `SuperchainConfig` contract holds the configuration and system parameters for all OP Stack-based chains (e.g., upgrade initiator/vetoer). Additionally, the contract also allowed the Guardian to pause the system, preventing L2 withdrawals. We performed a manual review of this contract (and related contracts) and investigated the following:
    - Whether an attacker can bypass the `SuperchainConfig` contract's access controls to manipulate critical system variables, (un)pause the system, or add/remove sequencers.
    - Whether any other L1 contract has write access to the `SuperchainConfig` contract such that they can manipulate critical state variables in an unexpected way.
    - Whether the `SystemConfig` contract correctly interfaces with the `SuperchainConfig` contract.
    - Whether an attacker can initiate a call, veto a call, or instantiate the upgrade delay using the `DelayedVetoable` contract. Similarly, we evaluated whether the initiator can veto a call or whether the vetoer can initiate a call.

    - Whether upgrading the `SuperchainConfig` contract introduced any edge cases that would allow an attacker to manipulate the upgrade of the `SuperchainConfig` contract.
    - Whether any *messages* can be relayed from L2 to L1 while the system is paused.

- **L1 deployment script**. The Forge L1 deployment script is used to deploy, initialize, and configure all L1 contracts during development and testing. We performed a manual review of this script and investigated the following:
    - Whether the deployment reflects the requirements described in milestone 2 of the documentation provided by the OP Labs team. This led to the discovery of TOB-OPT-MCP-5, which describes discrepancies between the deployment requirements and the deployment script.
    - Whether the deployment script has the necessary post-deployment checks after the deployment and initialization of each L1 contract. This led to the discovery of TOB-OPT-MCP-4, which highlights that the deployment of some contracts are missing the necessary checks.
    - The deployment script assigns the correct roles and permissions to the relevant actors for each system contract.
- **`LivenessModule` and `LivenessGuard`.** The `LivenessModule` and `LivenessGuard` contracts are L1 contracts that extend OP Labs' Gnosis Safe's capabilities to remove an owner's access to the Safe in case they become inactive (e.g., lose access to their key). The `LivenessGuard` is invoked before and after the execution of a Safe transaction and updates the "liveness" of each signer of the transaction. The `LivenessModule` can then be used to remove owners that have not shown "liveness" in a given interval. The `LivenessModule` uses the data stored in the `LivenessGuard` to determine the liveness of a given owner. We performed a manual review of these contracts and investigated the following:
    - Whether an attacker can use the `LivenessModule` to remove owners that should not have been removed or cause the Safe to enter a state that would render the `LivenessModule` unusable.
    - Whether the `LivenessModule` contract has all the necessary post-condition checks to prevent the Safe from entering an invalid state.
    - Whether the `LivenessGuard` contract can be put in a state, either maliciously or due to unexpected input, that would cause it to consistently revert and prevent the execution of any Safe transactions.

    - Whether the `LivenessGuard` contract correctly extracts signer addresses from the signature payload (via the `SafeSigners.getNSigners` method) and tracks the addition, removal, and swapping of owners.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **Configuration files for deployment.** It is important to note that which configuration file, the values in the configuration file, or the type of deployment (e.g., local chain or testnet) was not considered during the evaluation of the deployment script. These areas were considered out-of-scope.
- **Deployment of the `LivenessGuard` and `LivenessModule` contracts.** No Forge script was provided implementing the deployment model and order of the liveness contracts.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The changes in scope did not introduce or update any critical arithmetic operations/functions.	**Not Applicable**
Auditing	All critical state-changing operations emit events. Additionally, sufficient documentation was provided highlighting the actions that need to be taken in case of a security incident.	**Strong**
Authentication / Access Controls	All privileged functions have the necessary access controls. Additionally, no permissionless functions allow for access control bypasses or allow for unexpected state changes. The system documentation clearly highlights all system actors and their expected roles within the system.	**Strong**
Complexity Management	All the contracts and functions reviewed during the audit showcase a strong care for simplicity, conciseness, and purpose. All functions are clearly documented, and core functions can be easily tested using unit or automated testing. In isolation, these changes may appear to introduce additional complexity to the system, partially as a result of having to work around existing legacy design decisions. However, moving to a shared implementation in this way will ease the longer term maintenance burden of operating multiple OP Stack chains. The complexity of the larger system was not evaluated during this audit.	**Satisfactory**
Decentralization	The components in scope for this review are not intended to be fully decentralized themselves. They do contain sufficient access controls and safeguards to prevent any entity from making unilateral changes to one	**Not Considered**

	or more OP Stack instances on L1. However, the L2 components, on- and off-chain governance, as well as the actual membership of the multisigs play are significant factors in determining how decentralized the entire system is.	
Documentation	All contracts and functions are thoroughly documented, and a system specification is provided that highlights critical security properties and requirements. We did identify a few discrepancies between the specification and codebase that should be remediated (TOB-OPT-MCP-5 and appendix D).	**Satisfactory**
Low-Level Manipulation	There is minimal use of low-level assembly in the L1 smart contracts. However, the high-severity bug identified by the OP Labs team (TOB-OPT-MCP-3) highlights the risks of using low-level assembly to directly manipulate storage.	**Moderate**
Testing and Verification	All contracts are thoroughly tested with unit tests and automated fuzz testing. However, issues like TOB-OPT-MCP-3 indicate there may be potential gaps in how thoroughly sad paths are covered.	**Satisfactory**
Transaction Ordering	We did not identify any risks related to transaction ordering. A large majority of the code that was reviewed was primarily callable only by privileged actors, which reduces the likelihood of any profitable transaction ordering vectors. Any permissionless functions did not showcase any risks related to transaction ordering either.	**Satisfactory**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Missing storage layout for L1ERC721Bridge and L2ERC721Bridge	Auditing and Logging	**Informational**
2	The ERC721Bridge contract has fewer reserved storage slots than expected	Data Validation	**Informational**
3	clearLegacySlot allows the theft of funds through reinitialization	Data Validation	**High**
4	Insufficient post-deployment checks	Testing	**Informational**
5	Implementation of the Challenger does not satisfy the product requirements	Undefined Behavior	**Informational**
6	LivenessModule may reduce the threshold more than expected when removing an inactive owner	Undefined Behavior	**Informational**

Detailed Findings

1. Missing storage layout for L1ERC721Bridge and L2ERC721Bridge	
Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-OPT-MCP-1
Target: `packages/contracts-bedrock/src/L1/L1ERC721Bridge`, `packages/contracts-bedrock/src/L2/L2ERC721Bridge`	

Description

The `L1ERC721Bridge` and `L2ERC721Bridge` contracts do not have their storage layouts tracked. Changes to either contracts that unexpectedly affect their storage layouts may go unnoticed and lead to unexpected system failures.

Most of OP Labs' L1 and L2 contracts sit behind proxies allowing them to be upgraded with new features over time. Upgradeability comes at the cost of ensuring that storage changes to the implementation contract are performed carefully such that they do not overwrite existing storage slots in the proxy or introduce other undefined behavior.

To prevent such issues from occurring, the OP Labs team uses a script to output the storage layout of each implementation contract. This layout is then reviewed during the PR review process to ensure that no unexpected changes have been made to the storage layout. This is done by the `storage-snapshot.sh` script (figure 1.1). The script uses Foundry to inspect the storage layout for each contract specified in the `contracts` list (highlighted in figure 1.1).

```
#!/usr/bin/env bash

set -e

if ! command -v forge &> /dev/null
then
    echo "forge could not be found. Please install forge by running:"
    echo "curl -L https://foundry.paradigm.xyz | bash"
    exit
fi

contracts=(
  src/L1/L1CrossDomainMessenger.sol:L1CrossDomainMessenger
  src/L1/L1StandardBridge.sol:L1StandardBridge
  src/L1/L2OutputOracle.sol:L2OutputOracle
  src/L1/OptimismPortal.sol:OptimismPortal
  src/L1/SystemConfig.sol:SystemConfig
  src/legacy/DeployerWhitelist.sol:DeployerWhitelist
  src/L2/L1Block.sol:L1Block
  src/legacy/L1BlockNumber.sol:L1BlockNumber
  src/L2/L2CrossDomainMessenger.sol:L2CrossDomainMessenger
  src/L2/L2StandardBridge.sol:L2StandardBridge
  src/L2/L2ToL1MessagePasser.sol:L2ToL1MessagePasser
  src/legacy/LegacyERC20ETH.sol:LegacyERC20ETH
  src/L2/SequencerFeeVault.sol:SequencerFeeVault
  src/L2/BaseFeeVault.sol:BaseFeeVault
  src/L2/L1FeeVault.sol:L1FeeVault
  src/vendor/WETH9.sol:WETH9
  src/universal/ProxyAdmin.sol:ProxyAdmin
  src/universal/Proxy.sol:Proxy
  src/legacy/L1ChugSplashProxy.sol:L1ChugSplashProxy
  src/universal/OptimismMintableERC20.sol:OptimismMintableERC20
  src/universal/OptimismMintableERC20Factory.sol:OptimismMintableERC20Factory
  src/dispute/DisputeGameFactory.sol:DisputeGameFactory
)

dir=$(dirname "$0")

echo "Creating storage layout diagrams.."

echo "=======================" > $dir/../.storage-layout
echo "👁👁 STORAGE LAYOUT snapshot 👁👁" >> $dir/../.storage-layout
echo "=======================" >> $dir/../.storage-layout

for contract in ${contracts[@]}
do
  echo -e "\n=======================" >> $dir/../.storage-layout
  echo "➡ $contract">> $dir/../.storage-layout
  echo -e "=======================\n" >> $dir/../.storage-layout
  forge inspect --pretty $contract storage-layout >> $dir/../.storage-layout
done
echo "Storage layout snapshot stored at $dir/../.storage-layout"
```

Figure 1.1: The `storage-snapshot.sh` script does not output the storage layout of the `L1ERC721Bridge` and `L2ERC721Bridge` contracts (`storage-snapshot.sh#L1-52`)

However, notice that the `contracts` list does not contain the `L1ERC721Bridge` and the `L2ERC721Bridge` contracts. Thus, any storage changes made to these contracts may go unnoticed during the PR review process.

Recommendations

Short term, add `src/L1/L1ERC721Bridge:L1ERC721Bridge` and `src/L2/L2ERC721Bridge:L2ERC721Bridge` to the list of contracts that will be inspected.

Long term, ensure that each PR that affects the storage layout of a smart contract also has a diff for the `.storage-layout` file. Additionally, explore opportunities for automating the identification and validation of storage layout changes to smart contracts.

2. The ERC721Bridge contract has fewer reserved storage slots than expected

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-OPT-MCP-2
Target: `packages/contracts-bedrock/src/L2/L2ERC721Bridge`	

Description

The `ERC721Bridge` contract has one fewer reserved storage slots than expected which may cause unexpected behavior in future contract upgrades.

The `ERC721Bridge` contract is an abstract contract that is inherited by the `L1ERC721Bridge` and `L2ERC721` contracts. Inherited (or "base") contracts, such as `ERC721Bridge`, should have a "gap" that allows for future storage variables to be introduced without overwriting storage slots of inheriting contracts. This is done by the `ERC721Bridge.__gap` variable. The `__gap` variable, In addition to the other storage variables in the contract, should lead to the contract consuming a total of 50 slots (figure 2.1). The value of 50 slots was chosen by the OP Labs team as the target number for all base contracts.

```
abstract contract ERC721Bridge is Initializable {
    /// @notice Messenger contract on this domain.
    /// @custom:network-specific
    CrossDomainMessenger public messenger;

    /// @notice Address of the bridge on the other network.
    /// @custom:legacy
    address public immutable OTHER_BRIDGE;

    /// @notice Reserve extra slots (to a total of 50) in the storage layout for
future upgrades.
    uint256[48] private __gap;
```

Figure 2.1: The `ERC721Bridge` contract only consumes 49 storage slots, instead of 50. (`ERC721Bridge.sol#L10-20`)

However, the `ERC721Bridge` contract consumes 49 slots. The `Initializable` contract, which is inherited by the `ERC721Bridge` contract, has two storage variables: `_initialized` and `_initializing`. Both are booleans that will be packed together in the first (0-th indexed) storage slot. The `messenger` storage variable will also be packed in the same slot since it consumes only 20 bytes. Thus, all three storage variables consume a total of 1 slot. This, in addition to the 48 slots consumed by the `__gap` variable leads to a total of 49 slots, instead of 50.

Recommendations

Short term, update `__gap` to be `uint256[49]`.

Long term, ensure that each base contract consumes a total of 50 slots. Additionally, for each PR that changes the storage of a base contract, perform a manual review of the `.storage-layout` file to ensure that the contract still consumes a total of 50 slots. Finally, consider writing a script that automates the process of identifying storage slot consumption and integrating it into the CI pipeline.

3. clearLegacySlot allows the theft of funds through reinitialization	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-OPT-MCP-3
Target: `packages/contracts-bedrock/src/L1/L1StandardBridge.sol`	

Description

The `clearLegacySlot` modifier allows the `L1StandardBridge` contract to be reinitialized by anyone, which would allow an attacker to set the cross-domain messenger address and steal user funds.

For the `L1StandardBridge` to use the OpenZeppelin `Initializable` implementation as part of the shift to supporting a single implementation for all OP chains, the 0th storage slot must be reset for deployments that predate the 0th slot being a spacer. The current live version of the contract is not `Initializable`, and this storage slot is a placeholder reserved to offset the former location of the cross-domain messenger address in storage, before it was made an immutable value in a more recent upgrade.

To facilitate resetting this storage slot, a `clearLegacySlot` modifier was added to the `initialize` function to reset the 0th storage slot before the `reinitialize` modifier runs and performs its validation (figure 3.1). However, the `clearLegacySlot` modifier does not contain any logic to limit it to being executed only once. This allows `initialize` to be called again since the values of the `_initialized` and `_initializing` variables, which are both packed in the 0th slot and are validated by the `reinitialize` modifier, are wiped by the `clearLegacySlot` modifier. During reinitialization, an attacker can replace the `L1CrossDomainMessenger` contract's address with their own address, allowing them to falsely finalize bridging operations to withdraw arbitrary ETH or ERC-20 tokens from the bridge contract.

```
constructor() StandardBridge(StandardBridge(payable(Predeploys.L2_STANDARD_BRIDGE)))
{
    initialize({ _messenger: CrossDomainMessenger(address(0)) });
}

/// @notice Storage slot 0 holds a legacy value on upgraded networks. It is an empty
//          placeholder slot on new networks. Manually set it to 0 so that
`Initializable`
//          can use the first storage slot. This few lines of code helps to prevent
a large
//          diff in the source code to preserve the storage layout. This should be
```

```
removed
//          during the next contract upgrade.
modifier clearLegacySlot() {
    assembly {
        sstore(0, 0)
    }
    _;
}

/// @notice Initializer
///         The fix modifier should be removed during the next contract upgrade.
function initialize(CrossDomainMessenger _messenger) public clearLegacySlot
reinitializer(2) {
    __StandardBridge_init({ _messenger: _messenger });
}
```

Figure 3.1: The `L1StandardBridge` contract allows anyone to reinitialize the contract and set the `L1CrossDomainMessenger` address. (`L1StandardBridge.sol#L75-L95`)

Exploit Scenario

The OP Mainnet `L1StandardBridge` proxy is upgraded to use the contract implementation with the `clearLegacySlot` modifier. Eve notices this and calls `initialize` to set her address as the cross-domain messenger. She can now call `finalizeBridgeETH` and `finalizeERC20Withdrawal` on the `L1StandardBridge` contract to withdraw all the assets escrowed with the bridge to an address controlled by her.

Recommendations

There are two plausible paths to work around this storage issue:

1. **Modify `Initializable` to use a custom storage slot.** This approach avoids reusing the 0th storage slot altogether and appears to be more consistent with the existing approach of using special spacer variables to mark storage associated with deprecated functionality. Though this would no longer be using an off-the-shelf contract, the implementation would be simple enough to not be a maintenance burden on its own. This approach would be the simplest, with fewer opportunities for misuse or to unintentionally break the system, and the scale of what could go wrong is more constrained. However, there would ultimately be a higher maintenance burden because the custom `Initializable` implementation would have to be used alongside the standard `Initializable` implementation due to the use of some legacy proxies.

2. **Implement two-step upgrades with custom storage setters.** This approach involves separating the upgrade process into two stages: an intermediate stage that allows storage to be manipulated directly and a second stage that contains the actual logic upgrades. This would allow the 0th storage slot of the `L1StandardBridge` to be reset, and the standard `Initializable` contract would

be usable. Manipulating storage in this way could easily introduce very serious side effects if not carried out with the utmost care. This approach could also result in downtime and unexpected behavior in other services on- or off-chain if the entire upgrade is not carried out atomically within a single transaction. A mature process and testing strategy will be necessary to safely execute an upgrade using this two-step approach.

Long term, avoid reusing existing storage even when it has been deprecated. Storage schemas like ERC-7201 can be used in future development to avoid the chance of these collisions occurring. Develop a robust process and testing framework in case extenuating circumstances, such as backward compatibility, require storage to be updated directly.

4. Insufficient post-deployment checks	
Severity: **Informational**	Difficulty: **N/A**
Type: Testing	Finding ID: TOB-OPT-MCP-4
Target: `packages/contracts-bedrock/scripts/Deploy.s.sol`	

Description

There are a number of cases where deployments or initializations of L1 contracts are missing post-deployment checks to validate that the deployment or initialization was successful. For example, when the delay is activated for the `DelayedVetoable` contracts, there are no post-condition checks to ensure that the delay has been successfully updated to the value stored in the `SuperchainConfig` contract (figure 4.1).

```
function activateDelay() public broadcast {
    address delayedVetoableForSuperchain =
mustGetAddress("DelayedVetoableForSuperchain");
    _callViaSafe({
        _safe: Safe(mustGetAddress("SuperchainConfigInitiatorSafe")),
        _target: delayedVetoableForSuperchain,
        _data: hex""
    });
    console.log("Delay activated on DelayedVetoableForSuperchain contract at %s",
delayedVetoableForSuperchain);

    address delayedVetoableForOpChain = mustGetAddress("DelayedVetoableForOpChain");
    _callViaSafe({
        _safe: Safe(mustGetAddress("SuperchainConfigInitiatorSafe")),
        _target: delayedVetoableForSuperchain,
        _data: hex""
    });
    console.log("Delay activated on SuperchainConfigInitiatorSafe contract at %s",
delayedVetoableForOpChain);

    address delayedVetoableForProtocolVersions =
mustGetAddress("DelayedVetoableForProtocolVersions");
    _callViaSafe({
        _safe: Safe(mustGetAddress("SuperchainConfigInitiatorSafe")),
        _target: delayedVetoableForProtocolVersions,
        _data: hex""
    });
    console.log(
        "Delay activated on DelayedVetoableForProtocolVersions contract at %s",
delayedVetoableForProtocolVersions
    );
}
```

Figure 4.1: The activation of the delay for the `DelayedVetoable` contracts is missing validation to ensure that the delay was successfully updated (`Deploy.s.sol#L1124-L1150`)

Similarly, during the initialization of the `SystemConfig` contract, there are checks missing to ensure that the proposer and challenger are correctly set to the values defined in the configuration file (figure 4.2).

```
function initializeSystemConfig() public broadcast {
    address systemConfigProxy = mustGetAddress("SystemConfigProxy");
    address systemConfig = mustGetAddress("SystemConfig");
    address superchainConfigProxy = mustGetAddress("SuperchainConfigProxy");
    bytes32 batcherHash = bytes32(uint256(uint160(cfg.batchSenderAddress())));
    uint256 startBlock = cfg.systemConfigStartBlock();

    _upgradeAndCallViaSafe({
        _safe: Safe(mustGetAddress("SuperchainConfigInitiatorSafe")),
        _target: mustGetAddress("DelayedVetoableForOpChain"),
        _proxy: payable(systemConfigProxy),
        _implementation: systemConfig,
        _innerCallData: abi.encodeCall(
            SystemConfig.initialize,
            (
                cfg.systemConfigOwner(),
                superchainConfigProxy,
                SystemConfig.GasConfig({ overhead: cfg.gasPriceOracleOverhead(),
scalar: cfg.gasPriceOracleScalar() }),
                batcherHash,
                uint64(cfg.l2GenesisBlockGasLimit()),
                cfg.p2pSequencerAddress(),
                Constants.DEFAULT_RESOURCE_CONFIG(),
                startBlock,
                cfg.batchInboxAddress(),
                SystemConfig.OracleRoles({
                    proposer: cfg.l2OutputOracleProposer(),
                    challenger: cfg.l2OutputOracleChallenger()
                }),
                SystemConfig.Addresses({
                    l1CrossDomainMessenger:
mustGetAddress("L1CrossDomainMessengerProxy"),
                    l1ERC721Bridge: mustGetAddress("L1ERC721BridgeProxy"),
                    l1StandardBridge: mustGetAddress("L1StandardBridgeProxy"),
                    l2OutputOracle: mustGetAddress("L2OutputOracleProxy"),
                    optimismPortal: mustGetAddress("OptimismPortalProxy"),
                    optimismMintableERC20Factory:
mustGetAddress("OptimismMintableERC20FactoryProxy")
                })
            )
        )
    });
    SystemConfig config = SystemConfig(systemConfigProxy);
    string memory version = config.version();
    console.log("SystemConfig version: %s", version);

    require(config.owner() == cfg.systemConfigOwner());
    require(config.overhead() == cfg.gasPriceOracleOverhead());
    require(config.scalar() == cfg.gasPriceOracleScalar());
    require(config.unsafeBlockSigner() == cfg.p2pSequencerAddress());
    require(config.batcherHash() == batcherHash);

    ResourceMetering.ResourceConfig memory rconfig =
Constants.DEFAULT_RESOURCE_CONFIG();
    ResourceMetering.ResourceConfig memory resourceConfig = config.resourceConfig();
    require(resourceConfig.maxResourceLimit == rconfig.maxResourceLimit);
    require(resourceConfig.elasticityMultiplier == rconfig.elasticityMultiplier);
    require(resourceConfig.baseFeeMaxChangeDenominator ==
rconfig.baseFeeMaxChangeDenominator);
    require(resourceConfig.systemTxMaxGas == rconfig.systemTxMaxGas);
    require(resourceConfig.minimumBaseFee == rconfig.minimumBaseFee);
    require(resourceConfig.maximumBaseFee == rconfig.maximumBaseFee);

    require(config.l1ERC721Bridge() == mustGetAddress("L1ERC721BridgeProxy"));
    require(config.l1StandardBridge() == mustGetAddress("L1StandardBridgeProxy"));
    require(config.l2OutputOracle() == mustGetAddress("L2OutputOracleProxy"));
    require(config.optimismPortal() == mustGetAddress("OptimismPortalProxy"));
    require(config.l1CrossDomainMessenger() ==
mustGetAddress("L1CrossDomainMessengerProxy"));

    // A non zero start block is an override
    if (startBlock != 0) {
        require(config.startBlock() == startBlock);
    } else {
        require(config.startBlock() == block.number);
    }
}
```

Figure 4.2: The initialization of the `SystemConfig` contract is missing post-condition checks to ensure that the proposer and challenger addresses are correctly instantiated (`Deploy.s.sol#L784-L855`)

Recommendations

Ensure that the deployment or initialization of each L1 contract has the associated post-deployment checks to validate that the state of the contract is correctly updated.

5. Implementation of the Challenger does not satisfy the product requirements

Severity: **Informational**	Difficulty: **N/A**
Type: Undefined Behavior	Finding ID: TOB-OPT-MCP-5
Target: `packages/contracts-bedrock/src/L1/SystemConfig.sol`	

Description

The `SystemConfig` contract stores the addresses of the OP Chain's Proposer and Challenger, who are responsible for submitting and deleting output roots, respectively. The *Security Council Product Requirements FAQ* page defines the following requirements for these roles as of Milestone 2:

> *Output Proposal requirements*
>
> - *For each OP chain, any of the following may act as the Challenger:*
>   - *OP Foundation, Security Council, the Chain Governor for that OP Chain*
> - *Challenger can replace a malicious Proposer and delete outputs.*
> - *The Challenger entity is not modifiable*
> - *The Governor can add a new Proposer*

For the first requirement, the `SystemConfig` contract provides a function called `isProposalManager` that the `L2OutputOracle` contract uses to restrict who can call the `deleteL2Outputs` function (i.e., to check if the caller is the Challenger). This function, however, checks to see if the caller is the `SystemConfig.challenger` address rather than the owner of the `SystemConfig` contract (i.e., the Chain Governor), or if the caller is either of the `SuperchainConfig.initiator` (i.e., the Security Council) address, or the `SuperchainConfig.vetoer` (i.e., the OP Foundation) address. Note that the `SystemConfig.challenger` address is set arbitrarily by the Chain Governor, there is no on-chain enforcement that this party is the Chain Governor themself. Thus, there is no guarantee that the Challenger is in fact one of the three parties mentioned in the requirements above.

The second requirement is satisfied since both `L2OutputOracle.deleteL2Outputs` and `SystemConfig.setProposer` use the `isProposalManager` function to restrict who is able to call these functions.

The third requirement is not currently satisfied as the underlying challenger address can be updated by the owner of the `SystemConfig` contract (i.e., the Chain Governor) by calling the `setChallenger` function.

The final requirement is not satisfied directly as only the Challenger may call the `setProposer` function in the `SystemConfig` contract and the `challenger` address is not necessarily the Chain Governor. However, this requirement can be satisfied indirectly since the owner of the `SystemConfig` contract is able to update the `challenger` address to one they control. Note, however, that this capability conflicts with the previous requirement being satisfiable. Additionally, the word "add" may be ambiguous as the implementation only supports one Proposer at a time.

Finally, the separate *SuperchainConfig and Chain Governor - Product Requirements* document specifies the following:

> What permissions does the Security Council have?
>
> *As the 'owner' of the SuperChainConfig contract, the Security Council may:*
>
> - *...*
> - *Update the following values (subject to delay and veto):*
>   - *The Challenger account*
>   - *...*

This requirement conflicts with the requirement above that the Challenger is not modifiable. This requirement is also not satisfied by the current implementation as the `SuperchainConfig` owner (i.e., the Security Council) does not have the capability to change the Challenger account (presumably the `SystemConfig.challenger` address, though it is not always clear which of the role or address is being referenced). This capability is only given to the owner of the `SystemConfig` contract (i.e., the Chain Governor).

Recommendations

Review the specification and implementation and update them as necessary. In the specification, prefer referring to roles over specific entities (e.g., the initiator rather than the Security Council). Avoid overloading terminology (e.g., the specification's Challenger actor is different from the implementation's `challenger` address). Additionally, rename the `isProposalManager` function to `isChallenger`.

6. LivenessModule may reduce the threshold more than expected when removing an inactive owner	
Severity: **Informational**	Difficulty: **N/A**
Type: Undefined Behavior	Finding ID: TOB-OPT-MCP-6
Target: `packages/contracts-bedrock/src/Safe/LivenessModule.sol`	

Description

When a call is made to the `LivenessModule.removeOwners` function, in addition to the specified owners being removed, the threshold of the safe is set to the minimum number of owners needed to exceed 75% which may result in a larger reduction in the threshold than expected.

```
function _removeOwner(address _prevOwner, address _ownerToRemove, uint256
_newOwnersCount) internal {
    if (_newOwnersCount > 0) {
        uint256 newThreshold = get75PercentThreshold(_newOwnersCount);
        // Remove the owner and update the threshold
        _removeOwnerSafeCall({ _prevOwner: _prevOwner, _owner: _ownerToRemove,
_threshold: newThreshold });
    } ...
```

Figure 6.1: A snippet of the internal `_removeOwner` function called during the `removeOwners` function (`LivenessModule.sol#L156-L161`)

```
/// @notice For a given number of owners, return the lowest threshold which is
greater than 75.
///         Note: this function returns 1 for numOwners == 1.
function get75PercentThreshold(uint256 _numOwners) public pure returns (uint256
threshold_) {
    threshold_ = (_numOwners * 75 + 99) / 100;
}
```

Figure 6.2: The `get75PercentThreshold` function (`LivenessModule.sol#L65-L69`)

A natural assumption when removing a single owner from an N-of-M safe might be that the safe becomes N-of-(M-1) or (N-1)-of-(M-1), depending on whether there is a preference for a higher threshold or higher availability, respectively. The constructor of the `LivenessModule` contract treats the 75% threshold as a lower bound rather than an exact requirement, suggesting that the module is intended to support modules with stricter thresholds. However, the implementation of the `removeOwners` function could result in the safe transitioning to a (N-2)-of-(M-1) threshold, or even lower, for large enough values of M.

```
constructor(
    Safe _safe,
    LivenessGuard _livenessGuard,
    uint256 _livenessInterval,
    uint256 _minOwners,
    address _fallbackOwner
) {
    SAFE = _safe;
    LIVENESS_GUARD = _livenessGuard;
    LIVENESS_INTERVAL = _livenessInterval;
    FALLBACK_OWNER = _fallbackOwner;
    MIN_OWNERS = _minOwners;
    address[] memory owners = _safe.getOwners();
    require(_minOwners <= owners.length, "LivenessModule: minOwners must be less
than the number of owners");
    require(
        _safe.getThreshold() >= get75PercentThreshold(owners.length),
        "LivenessModule: Safe must have a threshold of at least 75% of the number of
owners"
    );
}
```

Figure 6.3: The constructor of the `LivenessModule` that validates the underlying safe's parameters (`LivenessModule.sol#L45-L63`)

In contrast to the constructor, which allows a higher than necessary threshold, the `_verifyFinalState` function called at the end of the `removeOwners` function to ensure the safe is still in a consistent state enforces that the safe's threshold must be strictly equal to the amount required to meet the 75% threshold.

```
function _verifyFinalState() internal view {

    ...

    // Check that"LivenessModule: must remove all owners and transfer to fallback
owner if numOwners < minOwners"
    // the threshold is correct. This check is also correct when there is a single
    // owner, because get75PercentThreshold(1) returns 1.
    uint256 threshold = SAFE.getThreshold();
    require(
        threshold == get75PercentThreshold(numOwners),
        "LivenessModule: Safe must have a threshold of 75% of the number of owners"
    );

    ...
}
```

Figure 6.4: A snippet of the internal `_verifyFinalState` function showing the 75% being enforced as a strict requirement, not a minimum (`LivenessModule.sol#L200-L230`)

Exploit Scenario

Alice is an inactive owner of an 8-of-9 (88.8%) safe with the `LivenessModule` attached. Bob calls `LivenessModule.removeOwners` to remove Alice from the safe. Alice is removed and the safe now has a 6-of-8 (75%) signer requirement, completely skipping over 7-of-8 (87.5%) which was also a valid option. As a result, transactions may be executed through the safe with fewer signatures than expected.

Recommendations

Update the `_removeOwner` function so that it passes a value closer to the current threshold instead of dropping to the minimum possible value. Also update the `_verifyFinalState` function to enforce the 75% threshold as a lower bound instead of as strict equality.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories

Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria

Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.

Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendations are not associated with any specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

Multichain changes

- Remove the `@custom:network-specific` NatSpec tag for the `OTHER_BRIDGE` variable in the `StandardBridge` abstract contract. `OTHER_BRIDGE` is an immutable variable that will always have the same value.
- Consistently apply the naming conventions for immutable and storage variables. For example, the `PORTAL` variable in the `L1CrossDomainMessenger` contract is now a regular storage variable but still uses SCREAMING_SNAKE_CASE. On the other hand, the `messenger` variable in the `StandardBridge` contract was also updated to be a storage variable and now uses camelCase. Note that based on the documentation provided, immutable variables must use SCREAMING_SNAKE_CASE while storage variables must use camelCase.
- Update the `initialize` function for the `L2ERC721Bridge` contract to accept no arguments and set the `_messenger` value to the `Predeploys.L2_CROSS_DOMAIN_MESSENGER`. This minimizes the chance of user error since the cross-domain messenger for any L2 chain is always the predeployed `L2CrossDomainMessenger` contract.

`DelayedVetoable` contract

- Update the `@return` NatSpec tag for the `delay` getter function since it currently suggests the function returns an address instead of an unsigned integer.
- Update the inline NatSpec documentation for the `readOrHandle` modifier to highlight that any caller outside of `address(0)` (which can only be done via `eth_call`) will be directed to the `_handleCall` function rather than reverting.

`SystemConfig` contract

- The NatSpec comment above the `OracleRoles` struct describes a different struct.

`CrossDomainMessenger` contract

- Update the inline comment for the paused function to say:

```
On L1 this function will check the SuperchainConfig for its
paused status
```

D. Security Council Requirements Feedback

The OP Labs team provided Trail of Bits with documentation outlining the requirements that must be satisfied by the rollout of the Security Council. In this appendix, we've included excerpts of the requirements documents along with comments that highlight potential areas of improvement, deviations in the implementation, or areas that could use additional context.

We assume any references to the Security Council and Foundation map to the initiator and vetoer addresses, respectively, in the implementation. Our notes are included below ***italicized and bolded***.

[External] Security Council - Product Requirements FAQ

Availability requirement

- The Security Council's SLA for signing a transaction is 72 hours.
    - Thus the Security Council should never need to take action under time pressure, and so the pause should be updated to apply to the full system, not just withdrawals.

The rest of the product requirements only refer to pausing withdrawals so we assume this is mentioned as an aside as an area of future work. No SLA is explicitly provided for the Foundation but we assume it is less than the delay period.

Upgrade requirements

- Security Council signer set is expanded to at least 6 of 8
- The Security Council assumes the ability and obligation to initiate pending upgrades on-chain
- Initiated upgrades take place after a 14-day delay.
    - Transferring the ability to propose an upgrade must also be subject to a delay.
- The Foundation can veto upgrades during that 14 days.

"At least 6 of 8" is an ambiguous description of the signer set. Elsewhere it is clarified that this means quorum is 75% of signers and the signer set must have at least 8 members.

Currently the system does not directly support transferring the ability to initiate upgrades. This is implicitly subject to delay and veto on-chain as this functionality would have to be added through an upgrade. Despite this, council members could also collude to bypass this limitation by transferring custody of their keys out-of-band.

The delay is specified as part of the configuration file read by the deploy script. The current scripts are intended for devnet and testnet use and set it to 100 seconds, rather than 14 days.

Output Proposal requirements

- For each OP chain, any of the following may act as the Challenger:
  - OP Foundation, Security Council, the Chain Governor for that OP Chain
- Challenger can replace a malicious Proposer and delete outputs.
- The Challenger entity is not modifiable
- The Governor can add a new Proposer

The Challenger is not constrained to only replacing a malicious Proposer but can also replace an honest Proposer as well. For additional discrepancies between the Output Proposal requirements and implementation, see TOB-OPT-MCP-5.

Security Council membership requirements

- On a quarterly basis, all Security Council members must prove that they can access their keys within 72 hours.
  - Members who fail to do so are automatically removed.
  - When a member is removed, the threshold may also be removed to ensure that:
    - `M/N` is the lowest ratio which remains above 75%
      - These ratios are (9 of 12, 9 of 11, 8 of 10, 7 of 9, 6 of 8).
      - If the signer set is reduced below 8, then a safety mechanism is activated which hands control of the Security Council to the Foundation.

We assume there is a typo and the intention is to say "the threshold may also be [reduced]". Other aspects of the lifecycle management of the Security Council are not elaborated on here (e.g., membership constraints such as overlap with the Foundation multisig, adding new members, or off-cycle renewals).

Emergency Procedure requirements

- The Guardian role may pause the system.
- The pause initially lasts X.
  - *Design parameter TBD*
- The pause 'thaws' after X amount of time
- The Pause can be extended by calling pause again.
- Key property: it is possible to pause indefinitely until things are fixed.

The Guardian role is able to pause withdrawals and extend the pause indefinitely. In the current implementation, the Guardian also has the ability to unpause but this may be a placeholder while the pausing parameters (i.e., the max duration and who will have this role) are being finalized.

4. How do these milestones protect us from the various risks?

In this section we look at various specific risks we might face, and consider how we would be positioned to respond at each of the various milestones.

For simplicity, we only include the Milestone 2 responses to each risk.

4.1 Existential bridge vulnerability

We receive notice via our bug bounty program that a vulnerability exists in our bridge contracts which makes it possible to immediately withdraw a significant portion of the funds.

We wish to first protect the funds at risk, and then fix the vulnerability.

The best available options to respond are listed below:

1. GUARDIAN would pause withdrawals until a fix is implemented.
2. SC would propose an upgrade, which finalizes after a delay.
3. The SC would unpause or allow the pause to timeout.

Currently, the `SuperchainConfig.guardian` address is responsible for unpausing the system rather than the Security Council. However, this still maintains the spirit of this option.

4.2 Claiming L2 Assets

In this scenario the Foundation is motivated to “claim assets” from a target account.

For simplicity we assume that the assets to be claimed are escrowed in the L1 Bridge.

1. The Proposer proposes an irregular state transition transferring the asset balance from the target to the Chain Governor.
2. The SC would update the Proposer key.
3. The SC can Challenge this proposal.

The vetoer (i.e., the Foundation) is able to update the Proposer and acts as a Challenger as well. This would result in a stalemate as the Foundation and Security Council alternate updating the Proposer.

4.3 Sequencer Censorship

In this scenario the Foundation is motivated to censor interactions with a particular contract on OP Mainnet. What actions are available to censor interactions with this target contract?

1. To prevent interacting with the target contract: If the FND operated Batcher censors transactions it can be removed by the SC. However the FND cannot unilaterally upgrade the depositTransaction function, and thus cannot prevent force inclusion of transactions.
2. To disable the target contract: If the FND proposes an irregular state transition, the SC can challenge to delete the output. The SC can block an irregular state transition proposed by the FND to remove or replaces the code from the target account.

<u>4.4 Rogue council attempting to take undesirable actions</u>

In this scenario, a quorum SC members are attempting a malicious upgrade.

The delayed veto allows the foundation to indefinitely prevent the malicious upgrade. However there is no mechanism for regaining control over the system from the SC.

<u>4.5 Uncooperative council blocking desired actions</u>

In this scenario, a quorum blocking number of SC members (ie. if it's a 6 of 8, then 3 can block) have begun refusing to participate in signing transactions.

We assume that this is not because of an inaccessible key, since liveness checking will eventually resolve this.

As of August 24, 2023 the requirements above do not provide an on-chain mechanism for regaining control from the Security Council. The primary means of recourse would be social and legal.

The system's resilience to the risks listed in 4.2-4.5 above are improved under this new structure. However, the exact makeup of each of the multisigs will be a factor in determining how effective they are. The risk of collusion or coercion is greatly reduced, but is difficult to eliminate entirely.

SuperchainConfig and Chain Governor - Product Requirements

Requirements

1. The `SystemConfig` owner is now referred to as the `ChainGovernor`. Each chain's `SystemConfig` will now validate its state against the state of the `SuperChainConfig`.
2. The `SecurityCouncil` account will be able to modify config values on the new `SuperChainConfig` contract. Unless otherwise stated, any config updates will be subject to delay.

It may be beneficial to note explicitly that updates to config values in the `SuperchainConfig` contract can only be done via an upgrade.

What permissions does the ChainGovernor have?

The `ChainGovernor` can update:

- The Batcher keys (unsafe block signer and batcher hash)
  - Although these values must be contained in an allowlist in the `SuperChainConfig`
- Gas config (`overhead` and `scalar`)
- Resource config (`maxResourceLimit`, `elasticityMultiplier`, `baseFeeMaxChangeDenominator`, `minimumBaseFee`, `systemTxMaxGas`, `maximumBaseFee`)

The `ChainGovernor` also has the ability to update the `SystemConfig.challenger` address. It is unclear whether the `ChainGovernor` can also update the `SystemConfig.proposer` address (see TOB-OPT-MCP-5). Additionally, the `ChainGovernor` can update the `SystemConfig.gasLimit` variable.

What permissions does the Security Council have?

As the 'owner' of the `SuperChainConfig` contract, the Security Council may:

- Add entries to the Batcher key allow list (this is the one change not subject to delay)
- Remove entries from the Batcher key allow list
  - In the event that a chain's current Sequencer is removed, the `ChainGovernor` will need to proactively update it within this 14 day period.
- Update the following values (subject to delay and veto):
  - The Challenger account
  - The Proposer account
  - The length of the delay period

The Security Council (SC) can also perform upgrades on the following:

- *Upgrade the `SuperchainConfig` contract.*
- *Upgrade the `ProtocolVersions` contract.*
- *Upgrades to all OP chain contracts (e.g. `L1CrossChainMessenger`, `SystemConfig`, etc.)*

Additionally, the above documentation highlights that the SC can update the "Challenger account". However, this contradicts the documentation provided for Output Proposal requirements (see TOB-OPT-MCP-5). It is unclear if this documentation supersedes the other.

How is the Challenger changing?

The Challenger will become a singleton contract which can challenge outputs on any chain in the SuperChain. It will be a 1 of 3 'pass through' which will forward challenges from any of the security council, the foundation, the chain governor **for that chain.**

It is unclear whether the singleton contract is one that is already implemented or something that will be implemented in the future. In the current state, the Challenger is considered to be any of the `SystemConfig.challenger` address, the `SuperchainConfig.initiator` address, or the `SuperchainConfig.vetoer` address via the `SystemConfig.isProposalManager` function (see TOB-OPT-MCP-5 for more details).