Arbitrum Nitro Smart Contracts

Date	May 2022
Auditors	Dominik Muhs, Martin Ortner

1 Executive Summary

This report presents the results of our engagement with **Arbitrum** to review **the Nitro Smart Contract Systems**.

The review was conducted over 7 weeks, from **25 Apr 2022** to **24 Jun 2022**. A total of 63 person-days were spent.

Section 5 - Smart Contracts presents the findings in-scope for this security review. In addition to the core part of the system, the smart contracts, the system is comprised of off-chain components written in Rust and Golang. The off-chain components were out-of-scope for this review. However, the assessment team chose to communicate findings encountered along tracing the data/activity flow from the enduser/validator, into the Arbitrum node, the prover, and finally the smart contracts in a separate section of this report. Relevant out-of-scope findings can be found in Section 6 - Other Code.

2 Scope

Our review focused on the files in `contracts/src` of github/OffchainLabs/Nitro @ ae5c3ba600e6bd41b0e3a5090179f5f63cae6bb0.

The client provided the following information:

The contracts are divided into several folders:

- `bridge`: This folder contains the L1 contracts handling messaging, both L1->L2 and L2->L1.
    - `Bridge` itself is a core contract in both L1->L2 and L2->L1, but it usually isn't accessed by users directly. It's largely unchanged from our Classic contracts.
    - Inbox (also known as the "delayed inbox") is the user-facing contract for submitting L1->L2 messages. It's largely unchanged from our Classic contracts, aside from additional safety checks on retryable submission.
    - `SequencerInbox` is used by the sequencer to post its batches to L1. It's the source of truth for the canonical message order, and contains a forceInclusion method to let users force the advancement of the delayed inbox in case the sequencer fails to sequence delayed messages within its time bounds. It's largely rewritten from our Classic contracts to allow for more efficient batching, as we've removed the need for "segments" which previously subdivided batches.
    - `Outbox` is the user-facing contract used to execute L2->L1 messages. It's changed from our Classic contracts, as now a send merkle accumulator comprises the entire history of all L2->L1 messages, which differs from the previous approach wherein each merkle tree has unique messages.
- `challenge`: This folder contains the challenge manager (but not the core "one step proof" component that resolves the final execution dispute). ChallengeManager is entirely rewritten from Classic. Instead of challenges being separate contract deployments, ChallengeManager now manages all challenges. We're primarily looking for the following types of issues:
    - A way to make a challenge last for longer than the initial asserterTimeLeft + challengerTimeLeft (note that both parties to the challenge may be malicious)

    - A way to bisect to an assertion compatible with your opponent (e.g. the challenge starts with party A asserting state 1, party B challenges with state 2, but then party A somehow challenges with the same state 2, leaving B no way to win)
    - A way to win the challenge without the opponent's clock expiring. Note that the _currentWin function currently forces a timeout instead of immediately declaring a winner, so that if we lose a challenge, we can fix the issue via a smart contract upgrade instead of an incorrect assertion instantly taking effect.

- `osp`: This folder contains the "one step proof" contracts, which execute a single step of the Nitro machine. The entrypoint in OneStepProofEntry takes a previous machine hash, a step count, a max message count, and a proof. After deserializing the top level machine and the current module, it calls out to the appropriate instruction evaluator contract. For the purposes of this audit, we don't think it'd be worthwhile to audit the entire Nitro proving system, so we're primarily interested in any circumstance in which a given previous machine hash could lead to two different output machine hashes (assuming of course that the inbox state, machine step count, and max message count are fixed).

- `state`: This folder contains libraries used by the one step proof contracts to deserialize and hash the Nitro machine's state.

- `rollup`: This contains the Rollup contract (along with a few other utilities), which is responsible for managing L1's view of the L2 state, via staking and challenges.

    - Compared to the Classic contracts, we've switched from each Node (a potential L2 state) being its own contract deployment to simply being a struct in a mapping. We've also modified what L2 state is to match the Nitro model.
    - We've also replaced our Rollup user and admin logic proxy with a more standard version based on OpenZeppelin's ERC1967Upgrade proxy. The implementation of this is in ../libraries/ArbitrumProxy.sol. Note that the RollupAdminLogic is only accessible to admins, and thus does not need access checks. We currently don't check for contract existence in our

_implementation() implementation, but we are considering adding that check: https://github.com/OffchainLabs/nitro/issues/423

    - We're primarily looking for an issue where an adversary creates an assertion and is unable to be challenged (including if it is confirmed quicker than the expected confirmation delay), or where an adversary is able to delay progression of the rollup without losing their stake.

- `libraries`: This folder contains various libraries used by other contracts
- `mocks and test-helpers`: These are exclusively used for testing and can be ignored
- `node-interface and precompiles`: These are just Solidity interfaces to native Go code and can be ignored

---

2.1 Objectives

Together with the Arbitrum team, we identified the following priorities for our review:

1. Identify vulnerabilities and weaknesses relating to the passing of messages between L1 and L2.
2. Review the token bridge and other peripherals to ensure the security of funds locked in escrow.
3. Analyze potential malicious validator behavior and validate that the challenge process cannot be manipulated without punishment.
4. Ensure that the system is implemented consistently with the intended functionality and without unintended edge cases.
5. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

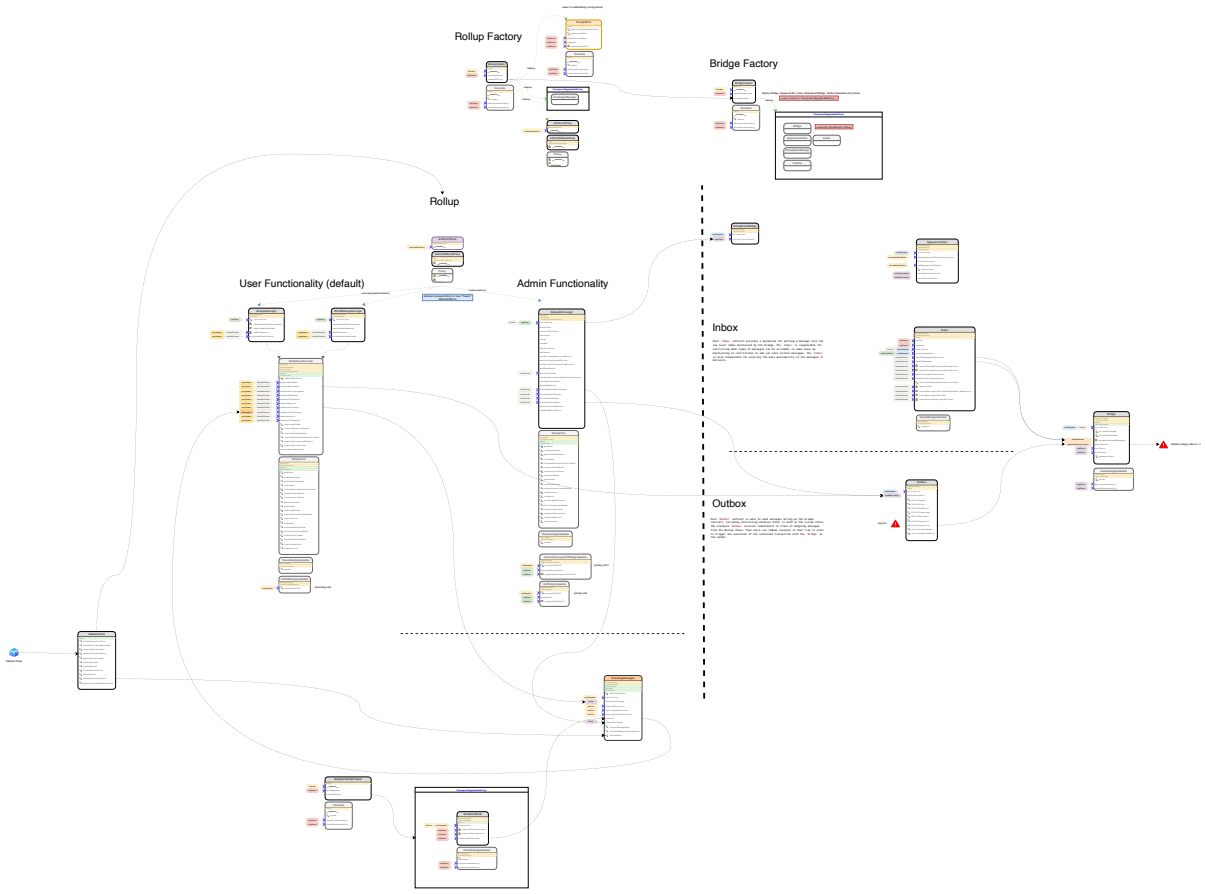
3 System Overview

This section describes the top-level/deployable contracts, their inheritance structure and interfaces, actors, permissions and important contract

interactions of the system under review. Please refer to Section 4 - Security Specification for a security-centric view on the system.

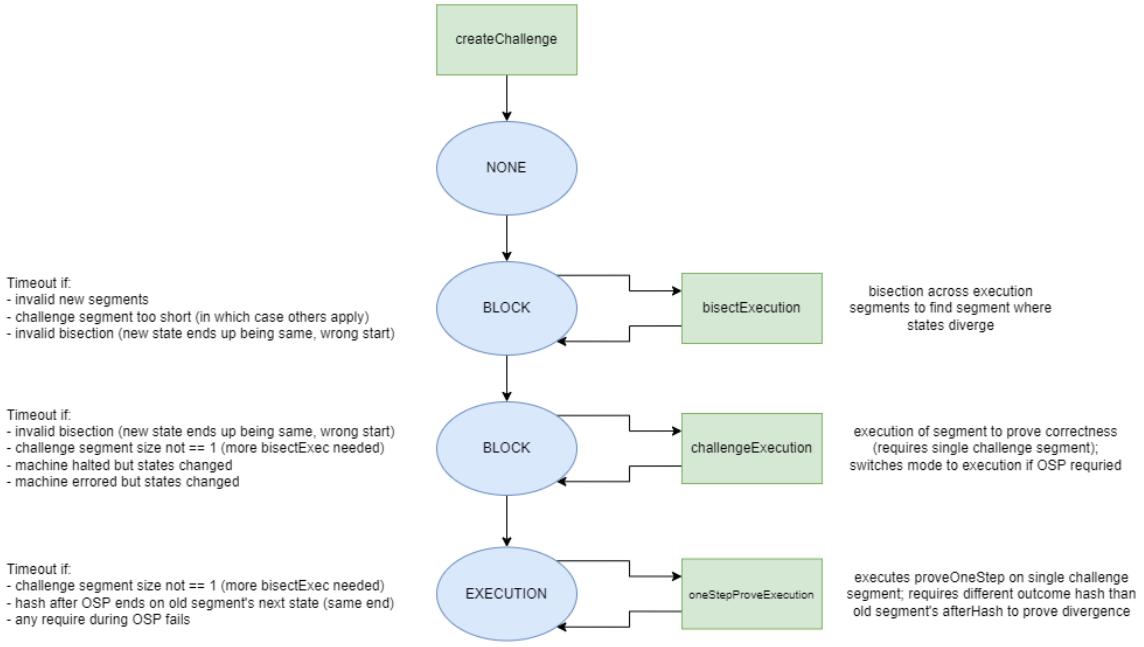
Contracts are depicted as boxes. Public reachable interface methods are outlined as rows in the box. The 🔍 icon indicates that a method is declared as non-state-changing (view/pure) while other methods may change state. A yellow dashed row at the top of the contract shows inherited contracts. A green dashed row at the top of the contract indicates that that contract is used in a usingFor declaration. Modifiers used as ACL are connected as yellow bubbles in front of methods.

3.1 Arbitrum Nitro Core Architecture



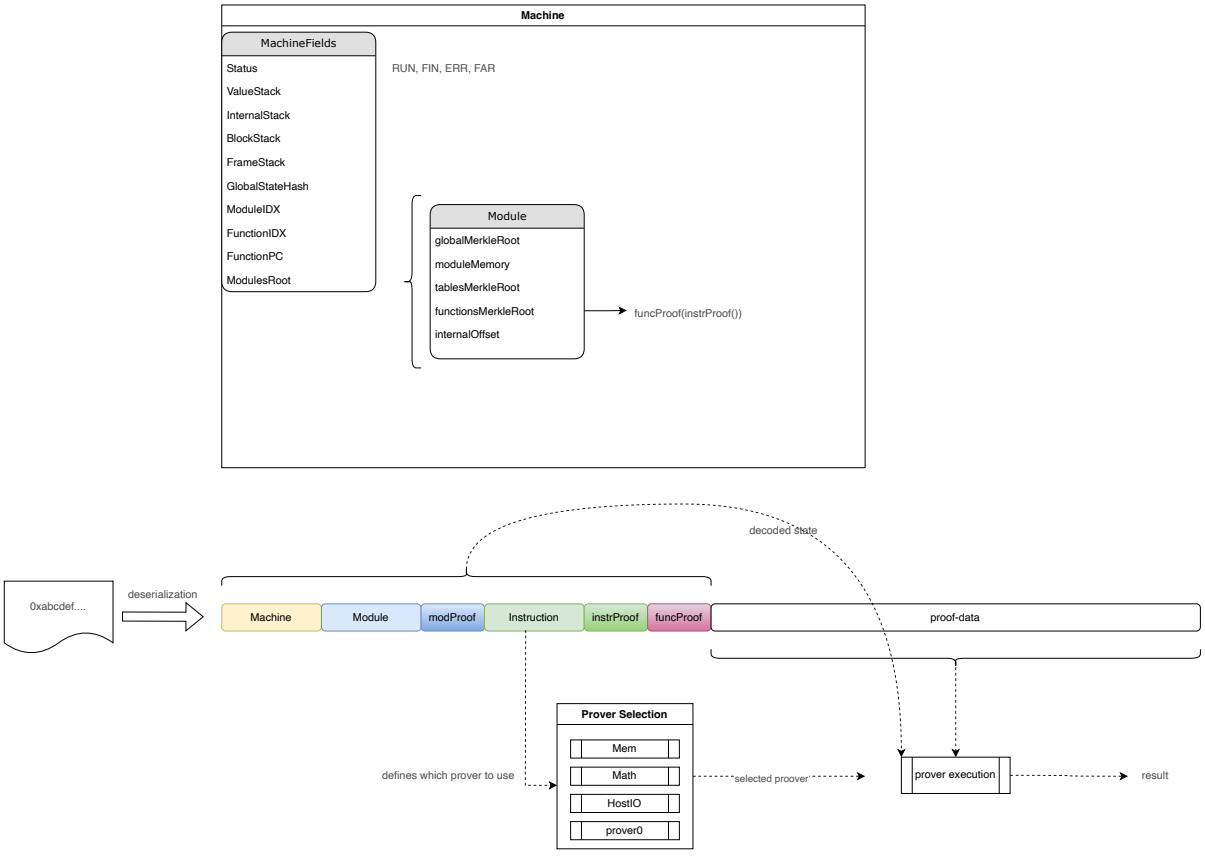
Arbitrum Nitro Core Architecture

3.2 Arbitrum Nitro Challenge Statemachine



Arbitrum Nitro Challenge Statemachine

3.3 Arbitrum Nitro OSP Deserialization



Arbitrum Nitro One Step Proof Deserialization

4 Security Specification

This section describes, from a security perspective, the system's expected behavior under review, particularly the Rollup-related smart contracts. It is not a substitute for documentation. This section aims to identify specific security properties and outline trust assumptions. While the security impact of the trust model notes is limited, they contain information that should be considered for the system's continued security.

Admin

- The rollup admin (a multisig, acc. to the client) is set by setting the `Config.owner` parameter in the `RollupCreator` factory. The `owner` (and proxy owner) is in a very exposed position (risk) as it can theoretically take over complete control of the `Bridge` contract (upgrade the contract, update settings), therefore, indirectly giving the admin access to all the escrowed funds.
- The rollup admin controls who can interact with the Rollup user functionalities (`onlyValidator` modifier).
  - It is a closed system right now that is controlled by the rollup admin ((a) via access restrictions configurable by the Admin, (b) via pausable, (c) the main rollup contracts being upgradeable/proxied).
  - The Admin decides who can challenge/confirm/reject block submissions that might undermine trust in the system. e.g., `createChallenge` can only be called by `onlyValidator`. That list is manually maintained by the Rollup Admin, while the initial assumption was that anyone should be able to call out malicious behavior and force a challenge.
- System properties can be changed anytime with little input validation via the `RollupAdminLogic` contract (risk of misconfiguration).
- The Admin can interfere with challenges up to a point where they can force a challenge outcome (`forceResolveChallenge`).
- The Admin can `forceCreateNode` and `forceConfirmNode` to submit an unchallengeable block.

- The Admin can pause the user functionalities. A similar effect might be created by an admin removing everyone from the `onlyValidator` access list.
- The contract can be upgraded due to the use of a proxy pattern. However, an admin can also change the `Admin` and `User` contracts while in use.
  - An admin can basically upgrade the contract to run any code.
  - An admin may interfere with users by unexpectedly upgrading user functionality while interacting with the system. Consider only allowing upgrades while the contract is paused.
- The ownership transfer is one-step which might pose a significant risk of losing access to the contract.
  - If this is misconfigured, the contract carrying the admin logic will not be callable anymore.
- System parameter changes are not always sanity-checked, increasing the risk for inconsistent configurations.
  - Parameter changes may interfere with users and block them from participating in the system's sub-processes, such as challenges.
  - For example, a misconfiguration of `confirmPeriodBlocks` to zero may allow anyone to initialize the contract again, setting a malicious admin facet, gaining control of the bridge, and/or destroying the Rollup contract.
  - `setBaseStake` allows setting a staking requirement of zero, which may harm the system's security. Furthermore, the `baseStake` should be multiple times higher than the minimum gas required for one party to resolve a challenge, or else this may open up a griefing vector.

onlyValidator/User

- Can add `newStake` to become **Staker**.
- Race: might block `confirmNextNode` if stake joins just before the confirm call (might be intentional, griefing). A malicious stake can grief `confirmNextNode` and may grief for `removal` with zero risk by not taking on the latest valid tip. Note that the longer someone delays block confirmation, the higher the stake requirements.
- **Anyone** can remove a new **Staker** immediately by calling `returnOldDeposit` if they don't bundle their call to `newStake` with `stakeOnExisting` / `stakeOnNewNode`.

- A zombie cannot be a **Staker** unless they are removed from their **Zombie** status.
- Can add funds to any existing **Staker**. Usually, only the staker themselves would do this.
- Can `confirmNextNode` if block deadlines are met, and **all** stakers are staked on the next node, i.e., everyone agrees.
  - Can be blocked by a single staker not advancing their stake to a new tip.
- Can `rejectNextNode` if block deadlines expired and the node does not point to `lastConfirmed` or no staker is staked.
- Can `returnOldDeposit` on any Staker (griefing: when called on new staker that does not stake on the tip).
- Can `createChallenge` for competing nodes.
  - A challenge is created in the ChallengeManager contract, and challengers are proving their nodes. First through bisection over the block numbers, then through bisection over the execution traces. Finally, by retracing a single state transition, the challenge manager resolves the challenge by submitting the result via `completeChallenge`.
  - **Note**: It is assumed that this method should not be restricted to **onlyValidator** given that anyone should be able to challenge misbehavior.
- Can `removeZombie`.
  - Removes a given zombie from nodes it is staked on.
- Can `removeOldZombies`.
  - Remove zombies whose latest stake is earlier than the first unresolved node.
- Can `withdrawStakerFunds`.
  - Stake that can be withdrawn (e.g., because `returnOldDeposit` or `reduceStake` were called) is accounted for internally until `withdrawStakerFunds` is called, allowing a staker to pull their stake off the system.

Staker

Note: Stakers should always stake way ahead of `lastConfirmed` node, or else they might open themselves up to an attack where anyone can call `returnOldDeposit` on them if they are only staked on `lastConfirmed`.

- Can `stakeOnExistingNode` (supporting a validated rollup block).
- Must stake on all previous nodes (or new staker).
  - Staker can optimize if too far off the tip by removing and re-adding themselves as staker (automatically stakes them on `latestConfirmed`) to stake at the new validated tip.
- Can `stakeOnNewNode` (submitting a rollup block).
  - Assumes the previous node is lastStakedNode.
- Can `reduceDeposit` to the minimum required stake at that moment.
- Can "block" node confirmation by not staking at the newly agreed tip.
  - May be challenged if they disagree on the new tip.
  - May be removed from stakers via `returnOldDeposit` if they do not advance to the new `tip`.

Challenge contract (called on challenge end)

- Calls `completeChallenge` with the challenge's final verdict.
  - 1/2 of the loser stake is credited to the **winner** (accounted as stake for the winner).
  - 1/2 of the loser stake is credited to the loserStakeEscrow address set in the Rollup contract system.
  - The amount of stake that can be slashed from the loser is capped at the amount the winner has staked (excess is refunded to the loser).
  - Loser is turned into a zombie and loses its **Staker** status.

Zombie

- Cannot participate in the system until they're removed from the zombie list.

5 Smart Contracts

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

5.1 Invalid assertions can be confirmed by frontrunning pause/unpause calls Major

Description

The rollup contract contains admin logic that allows authorized parties to pause and resume the system's operations:

code/contracts/src/rollup/RollupAdminLogic.sol:L117-L120

```
function pause() external override {
    _pause();
    emit OwnerFunctionCalled(3);
}
```

code/contracts/src/rollup/RollupAdminLogic.sol:L125-L128

```
function resume() external override {
    _unpause();
    emit OwnerFunctionCalled(4);
}
```

Consequently, the validator's ability to confirm, reject, or create nodes is paused as well. This includes the function to initiate challenges since it requires the contract to be unpaused.

code/contracts/src/rollup/RollupUserLogic.sol:L245-L254

```
function createChallenge(
    address[2] calldata stakers,
    uint64[2] calldata nodeNums,
    MachineStatus[2] calldata machineStatuses,
    GlobalState[2] calldata globalStates,
    uint64 numBlocks,
    bytes32 secondExecutionHash,
    uint256[2] calldata proposedTimes,
    bytes32[2] calldata wasmModuleRoots
) external onlyValidator whenNotPaused {
```

By frontrunning an admin's transaction to pause the rollup contract, an attacker is able to create a new node containing a wrong assertion. Other validators noticing the state inconsistency will be unable to create a challenge since the system is paused directly after.

Under the assumption that the system is paused just as long or longer than the block confirmation time, its containing node can be confirmed right away by the attacker by backrunning the admin's transaction calling `resume`.

An exacerbating factor to this issue is that an admin can change the rollup contract's settings in a way that lowers the block confirmation time period to a value that makes this attack feasible even for short "emergency pauses".

Recommendation

Right after calling `resume`, an admin must validate the latest chain head's correct execution and force the creation and confirmation of the latest node, referring to the last known good state. This process should be done atomically to prevent accidental or potentially malicious interference. Unconfirmed nodes before pausing should be regarded with the utmost scrutiny, or even discarded.

It can also be feasible to measure the pause time frame and add it on top of the block confirmation window.

5.2 Bridge - L2->L1 value call might lock ETH at bridge if destination is not a contract Major

Description

On L1/Ethereum mainnet, it is accepted and reasonably common to have an `ETH` value transfer with accompanying data (`calldata.length != 0`). The recipient is credited the `ETH` value, and calldata is not being executed.

When transferring value via a cross-chain L2->L1 contract call from the Arbitrum to the Ethereum chain, it is required that the L1 destination is a contract for the call to be executed successfully. If `calldata` was provided, e.g., a note sent with the transaction, and the L1 destination is not a contract, `Bridge.executeCall()` will revert, and the funds become unspendable.

Furthermore, a user cannot easily unlock the now locked value call unless Arbitrum decides to upgrade their Bridge contract or someone deploys code to the expected destination. The outbox entry has already been created and cannot be rolled back since the dispute window has already passed. Rolling back would not be an option because there is no reason to move back from a protocol perspective. Unfortunately, we could not locate any mention of this in the Arbitrum documentation.

This scenario might open up more attack vectors where someone's L2->L1 contract call with a value to a specific contract might be front-run. Note that there is plenty of time for someone to seek opportunities as L2->L1 calls are subject to a dispute period delay. An attacker can lock up the funds (griefing). In general, we suggest only performing L2->L1 contract calls with value if the caller can be confident that the L1 call cannot be forced to revert by a 3rd party.

Consider the trade-off between the following two scenarios:

1. Users might rely on the current behavior as a feature to lock up transactions until a contract is deployed at a specific contract address. This can be used for counterfactual wallets or dynamic `CREATE2` deployments.
2. The current behavior categorically excludes use-cases where `tx.data` is set on value transfers. Furthermore, rebasing protocols and other migrating systems that self-destruct their previous deployments will result in funds being locked.

Note that there is no reliable way for L2 to check if the L1 transaction would go through (in x days after the mandatory delay).

Examples

code/contracts/src/bridge/Bridge.sol:L110-L118

```
function executeCall(
    address to,
    uint256 value,
    bytes calldata data
) external override returns (bool success, bytes memory returnData) {
    if (!allowedOutboxesMap[msg.sender].allowed) revert NotOutbox(msg.sender);
    if (data.length > 0 && !to.isContract()) revert NotContract(to);
    address prevOutbox = activeOutbox;
    activeOutbox = msg.sender;
```

Recommendation

Consider providing a way for users to redeem their stuck L2->L1 transaction back to L2 if it always reverts. Make information available that describes the caveats of L2->L1 value transfers and provides guidance for the safe use of cross-chain contract interaction with value.

5.3 Admin function `forceRefundStaker` does not honor staker's state Medium

Description

In the `RollupAdminLogic` contract, the `forceRefundStaker` function allows authorized parties to unlock the funds of a given list of stakers and turn them into zombies.

This admin method fails to take into account the individual staker's state since they might be involved in a challenge at any given point in time. Assuming stakers A and B, where A is malicious, the following line of events is possible:

1. A challenge between A and B is initiated due to a malicious state transition performed by A.
2. The challenge progresses, and staker A is unable to respond anymore.

3. An admin calls the `forceRefundStaker` function and (among other addresses) converts A into a zombie. Their `amountStaked` value is set to zero. The value of their withdrawable funds now is at least the value of their previous stake.
4. Since all of staker A's funds have now been marked as withdrawable, they remove all funds from the system using `withdrawStakerFunds`.
5. Upon passing the final timeout, `completeChallengeImpl` is called, which, punishing the challenge's loser, divides the stakes and turns the losing staker into a zombie.

Staker B will not receive any rewards in the call to `completeChallengeImpl` as they don't have any actively staked funds. Furthermore, the function `turnIntoZombie` is called twice (once in step 3 and once in step 5). On the first call, the staker's address is successfully removed from the `_stakerMap`. In the second call's context, `_stakerMap[stakerAddress]` will return an uninitialized `Staker` struct. Most importantly, this will return a struct with `Staker.index` set to zero. Consequently, `deleteStaker` is called with the wrong staker, resulting in the system deleting the first staker of the core rollup contract's list:

code/contracts/src/rollup/RollupCore.sol:L496-L503

```
function deleteStaker(address stakerAddress) private {
    Staker storage staker = _stakerMap[stakerAddress];
    uint64 stakerIndex = staker.index;
    _stakerList[stakerIndex] = _stakerList[_stakerList.length - 1];
    _stakerMap[_stakerList[stakerIndex]].index = stakerIndex;
    _stakerList.pop();
    delete _stakerMap[stakerAddress];
}
```

Recommendation

Perform a check in `forceRefundStaker` requiring any given staker to not be in a challenge. This check can be accomplished by, e.g., requiring `staker.currentChallenge == 0`.

5.4 Unpredictable behavior due to immediate settings changes Medium

Description

In several cases, privileged accounts can update or upgrade things in the system without warning. Unannounced upgrades have the potential to violate the system's security goals.

Specifically, privileged roles could use front-running to make malicious changes just ahead of configuration-changing transactions, or random adverse effects could occur due to the unfortunate timing of changes.

Some instances of this issue are more significant than others, but in general, users of the system should have assurances about the behavior of the action they're about to take.

Examples

This issue mainly concerns the `RollupAdminLogic` contract where, e.g., challenge-related parameters can be changed by an authorized party, immediately affecting ongoing challenges. A further area of concern is the admin's ability to change the stake token address and the challenge loser's escrow contract address.

Other instances of admin configurable settings:

- TransparentUpgradeableProxy (`Bridge` , `Inbox` , `Outbox` , `RollupEventBridge` , `SequencerInbox` , ..
- Implementations of special proxies (`ArbitrumProxy` , `AdminFallbackProxy`)
- Configurative settings e.g. in `RollupAdminLogic` and other contracts
- Template configuration in factories

Recommendation

The underlying issue is that users of the system can't be sure of a function call's behavior. Due to potential configuration changes, the behavior can change at any time.

Consider giving users a time-locked notice of changes in advance. The first step merely tells users that the system will execute a particular change. The second step commits that change after a reasonable waiting period.

5.5 Inbox - is pausable by Rollup but Rollup does not implement any means to pause the contract Medium

Description

With the current system, the `Inbox` contract cannot be (un)paused. Pausing the inbox contract would allow the Arbitrum team to temporarily suspend new L1->L2 messages (Note that the `Bridge` / `SequencerInbox` cannot be paused).

The `Inbox` contract is `PausableUpgradeable`, with the owner of the `Bridge` contract (`Rollup`) being allowed to `[un]pause` the contract. However, the `Rollup` contract does not implement a way to pause the `Inbox` contract and `Rollup.pause()` only affects the `Rollup` contract.

Examples

- `onlyOwner == Rollup` can call `pause(), unpause()`

code/contracts/src/bridge/Inbox.sol:L34-L49

```
modifier onlyOwner() {
    // whoevever owns the Bridge, also owns the Inbox. this is usually the rollu
    address bridgeOwner = Bridge(address(bridge)).owner();
    if (msg.sender != bridgeOwner) revert NotOwner(msg.sender, bridgeOwner);
    _;
}

/// @notice pauses all inbox functionality
function pause() external onlyOwner {
    _pause();
}

/// @notice unpauses all inbox functionality
function unpause() external onlyOwner {
    _unpause();
}
```

- `Bridge(address(bridge)).owner()` is the `Rollup` contract.

code/contracts/src/rollup/BridgeCreator.sol:L109-L111

```
frame.delayedBridge.setInbox(address(frame.inbox), true);
frame.delayedBridge.transferOwnership(rollup);
```

- `RollupAdmin.[un]pause()` pauses the rollup. It does not implement a way to (un)pause the inbox.

code/contracts/src/rollup/RollupAdminLogic.sol:L112-L128

```
/**
 * @notice Pause interaction with the rollup contract.
 * The time spent paused is not incremented in the rollup's timing for node valid
 */
function pause() external override {
    _pause();
    emit OwnerFunctionCalled(3);
}

/**
 * @notice Resume interaction with the rollup contract
 */
function resume() external override {
    _unpause();
    emit OwnerFunctionCalled(4);
}
```

Recommendation

In the `RollupAdminLogic` contract, implement means to `[un]pause` the `Inbox` contract or remove the functionality. Provide guarantees that the team will unpause a paused contract at some point (timelock, document that the team can indefinitely pause the bridge).

5.6 AdminFallbackProxy - Name, Layout, Functionality Medium

Description

Various aspects of the system's proxy setup do not adhere to best practices and introduce redundant overhead, reducing readability and increasing the potential

for human error.

Fallback implementation names add significant cognitive overhead

`primary` and `secondary` have a special meaning in the context of this system and should be named explicitly as user- and admin-related routes. `implementation.primary` is the **privileged admin route**. At the same time, `implementation.secondary` is the fallback user-facing route. It also is a little counter-intuitive to have the admin route be the primary route, while in 99% of cases, the second route will be used.

code/contracts/src/libraries/AdminFallbackProxy.sol:L123-L129

```
assert(
    _IMPLEMENTATION_SECONDARY_SLOT ==
        bytes32(uint256(keccak256("eip1967.proxy.implementation.secondary")) -
);
_changeAdmin(adminAddr);
_upgradeToAndCall(adminLogic, adminData, false);
_upgradeSecondaryToAndCall(userLogic, userData, false);
```

We understand that there is a desire to keep this implementation as close as possible to the ERC1967 proxy, extending it to work with two fallback routes. However, given that the routes are not equal, with one being a privileged back-end, it is suggested to name them as such to avoid confusion.

For example, it is not immediately clear that `_upgradeToAndCall()` upgrades the **admin** implementation and `_upgradeSecondaryToAndCall()` upgrades the **user** implementation.

Rename `ArbitrumProxy`

The name `ArbitrumProxy` is misleading from an end-user perspective. It acts as **the** `Rollup` contract.

Consider renaming `ArbitrumProxy` by picking a more intuitive name like `RollupProxy`.

code/contracts/src/libraries/ArbitrumProxy.sol:L10-L20

```
contract ArbitrumProxy is AdminFallbackProxy {
    constructor(Config memory config, ContractDependencies memory connectedCont
        AdminFallbackProxy(
            address(connectedContracts.rollupAdminLogic),
            abi.encodeWithSelector(IRollupAdmin.initialize.selector, config, cc
            address(connectedContracts.rollupUserLogic),
            abi.encodeWithSelector(IRollupUserAbs.initialize.selector, config.s
            config.owner
        )
    {}
}
```

`ArbitrumProxy` source-unit location

Considering that `ArbitrumProxy` is the main `Rollup` contract it can be confusing to find it in the `libraries` sub-directory (`/contracts/src/libraries/ArbitrumProxy.sol`).

Consider moving the source unit to the rollup sub-directory as it is not a library but a top-level deployed contract.

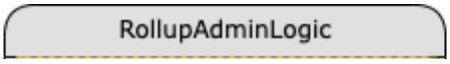
`SecondaryLogicUUPSUpgradeable.proxiableUUID` returns the primary slot

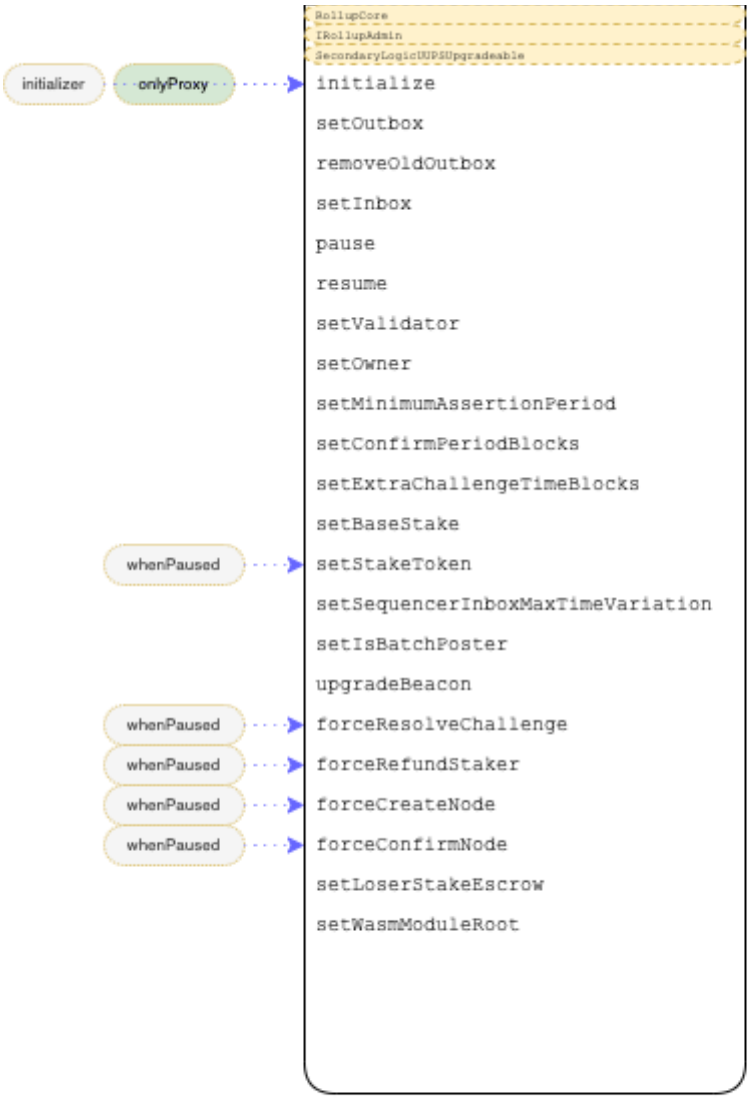
The module `SecondaryLogicUUPSUpgradeable` implements logic to upgrade the secondary implementation. However, the function `SecondaryLogicUUPSUpgradeable.proxiableUUID()` would return the primary implementation's UUID. This feels counterintuitive even though required right now as otherwise `SecondaryLogicUUPSUpgradeable.proxiableUUID` would override `UUPSUpgradeable.proxiableUUID` reference to the primary implementation (admin slot).

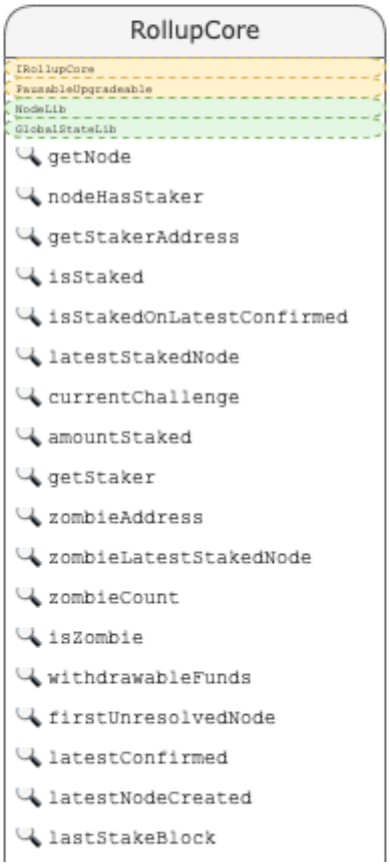
Consider renaming the method to `SecondaryLogicUUPSUpgradeable.proxyableSecondaryUUID()` to provide means to return the secondary's UUID and avoid overriding it via the contract inheritance structure.

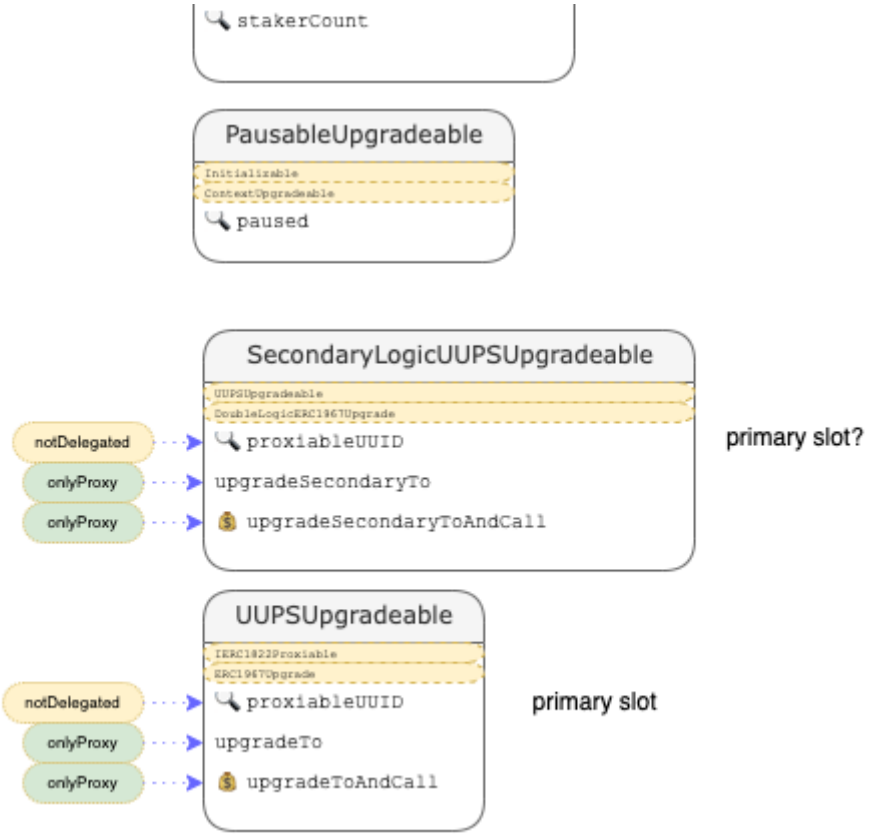
code/contracts/src/libraries/SecondaryLogicUUPSUpgradeable.sol:L14-L16

```
function proxiableUUID() external view override notDelegated returns (bytes32)
    return _IMPLEMENTATION_SLOT;
}
```




RollupCore
IRollupAdmin
SecondaryLogicUUPSUpgradeable
initializer
onlyProxy
initialize
setOutbox
removeOldOutbox
setInbox
pause
resume
setValidator
setOwner
setMinimumAssertionPeriod
setConfirmPeriodBlocks
setExtraChallengeTimeBlocks
setBaseStake
whenPaused
setStakeToken
setSequencerInboxMaxTimeVariation
setIsBatchPoster
upgradeBeacon
whenPaused
forceResolveChallenge
whenPaused
forceRefundStaker
whenPaused
forceCreateNode
whenPaused
forceConfirmNode
setLoserStakeEscrow
setWasmModuleRoot


RollupCore
IRollupCore
PausableUpgradeable
NodeLib
GlobalStateLib
getNode
nodeHasStaker
getStakerAddress
isStaked
isStakedOnLatestConfirmed
latestStakedNode
currentChallenge
amountStaked
getStaker
zombieAddress
zombieLatestStakedNode
zombieCount
isZombie
withdrawableFunds
firstUnresolvedNode
latestConfirmed
latestNodeCreated
lastStakeBlock



`DoubleLogicERC1967Upgrade` **supports a beacon slot for the primary implementation only**

`ERC1967Upgrade` provides means to return the beacon contract via `_getBeacon()` (for the primary implementation; admin logic). The implementation for beacon proxies in `DoubleLogicERC1967Upgrade` seems to be incomplete.

5.7 AdminFallbackProxy - Misleading Comment - ERC1967.admin cannot be set to zero address Minor

Resolution

This issue has been addressed by revising the comment with OffchainLabs/nitro#564.

Description

> // if the admin is disabled (set to addr zero), all calls will be forwarded to user logic

The comment suggests that the admin address can be set to `address(0)` while ERC1967.changeAdmin() -> ERC1967._setAdmin() does not allow that.

code/contracts/src/libraries/AdminFallbackProxy.sol:L132-L137

```
/// @inheritdoc Proxy
function _implementation() internal view override returns (address) {
    require(msg.data.length >= 4, "NO_FUNC_SIG");
    // if the sender is the proxy's admin, delegate to admin logic
    // if the admin is disabled (set to addr zero), all calls will be forwarded t
    address target = _getAdmin() != msg.sender
```

Recommendation

Revise the comment.

5.8 AdminFallbackProxy - Misleading Comment - admin is not secondary logic Minor

Description

A reference to the admin logic is stored in the **primary** implementation slot, the user logic in the **secondary**. However, the following comment confuses the slot usage:

code/contracts/src/libraries/AdminFallbackProxy.sol:L145-L152

```
/**
 * @dev unlike transparent upgradeable proxies, this does allow the admin to fall
 * the admin is expected to interact only with the secondary logic contract, whic
 * upgrades using the UUPS approach
 */
function _beforeFallback() internal override {
    super._beforeFallback();
}
```

> [...] the admin is expected to interact only with the secondary logic contract, [...]

Also note that this implementation of `_beforeFallback()` is basically a NO-OP.

Recommendation

The comment should read: `the admin is expected to interact with the primary logic contract`. We suggest avoiding confusion by explicitly naming the slots “fallback” and “admin” implementation (or similar; see https://github.com/ConsenSysDiligence/arbitrum-nitro-audit-2022-04/issues/5).

5.9 Inbox - duplicate initialization check Minor

Description

The `Inbox.initialize()` function carries the `initializer` modifier, which enforces that the contract can only be initialized once. Therefore, the check for `AlreadyInit()` is obsolete.

This issue is also related to issue 5.11.

Examples

code/contracts/src/bridge/Inbox.sol:L51-L55

```
function initialize(IBridge _bridge) external initializer onlyDelegated {
    if (address(bridge) != address(0)) revert AlreadyInit();
    bridge = _bridge;
    __Pausable_init();
}
```

Recommendation

Either remove the `initializer` modifier or the `AlreadyInit()` check. Check for function argument `_bridge != address(0)` instead.

5.10 Rollup - A reentrant StakeToken may allow zero-fee flash-loans Minor

Description

The `RollupUserLogic` contract allows users to stake and unstake within the same transaction context. In case a reentrant `StakeToken` is configured (i.e., `ERC-777` or

equivalent), a user can take a zero-fee flash loan by performing the following steps:

Examples

FlashLoan StakeToken from `Rollup`. Requires minimum stake to stay locked with the contract.

1. Stake on a node.
2. When `_newStake()` pulls in the token via `receiveTokens() -> token.transferFrom()`, an `ERC-777` token executes callback `from.tokensToSend()` before the token is actually transferred. The new stake, however, is already credited to the users' internal accounting.
3. In the reentrant `from.tokensToSend()` callback, the user calls `addToDeposit()` to inflate `Rollup`'s internal accounting to the desired amount. The `amount` can be up to the StakeToken balance the contract holds. Note that no tokens have been transferred yet, but the internal account credits us with the desired amount.
4. In the next reentrant `receiveTokens() -> from.tokensToSend()` callback, we call `reduceDeposit()` and `withdrawStakerFunds()` to "borrow" the stake tokens.
5. `<perform flashloan action>.`
6. Unwind and repay the tokens. This will return the loaned amount to `Rollup` and synchronize the external with the internal accounting.

code/contracts/src/rollup/RollupUserLogic.sol:L662-L672

```
function newStakeOnNewNode(
    uint256 tokenAmount,
    RollupLib.Assertion calldata assertion,
    bytes32 expectedNodeHash,
    uint256 prevNodeInboxMaxCount
) external override {
    _newStake(tokenAmount);
    stakeOnNewNode(assertion, expectedNodeHash, prevNodeInboxMaxCount);
    /// @dev This is an external call, safe because it's at the end of the funct
    receiveTokens(tokenAmount);
}
```

code/contracts/src/rollup/RollupUserLogic.sol:L706-L711

```
function receiveTokens(uint256 tokenAmount) private {
    require(
        IERC20Upgradeable(stakeToken).transferFrom(msg.sender, address(this), t
        "TRANSFER_FAIL"
    );
}
```

code/contracts/src/rollup/RollupUserLogic.sol:L679-L687

```
function addToDeposit(address stakerAddress, uint256 tokenAmount)
    external
    onlyValidator
    whenNotPaused
{
    _addToDeposit(stakerAddress, tokenAmount);
    /// @dev This is an external call, safe because it's at the end of the funct
    receiveTokens(tokenAmount);
}
```

code/contracts/src/rollup/RollupUserLogic.sol:L225-L232

```
function reduceDeposit(uint256 target) external onlyValidator whenNotPaused {
    requireUnchallengedStaker(msg.sender);
    uint256 currentRequired = currentRequiredStake();
    if (target < currentRequired) {
        target = currentRequired;
    }
    reduceStakeTo(msg.sender, target);
}
```

code/contracts/src/rollup/RollupUserLogic.sol:L689-L704

```
/**
 * @notice Withdraw uncommitted funds owned by sender from the rollup chain
 * @param destination Address to transfer the withdrawn funds to
 */
function withdrawStakerFunds(address payable destination)
    external
    override
    onlyValidator
    whenNotPaused
    returns (uint256)
{
    uint256 amount = withdrawFunds(msg.sender);
    // This is safe because it occurs after all checks and effects
    require(IERC20Upgradeable(stakeToken).transfer(destination, amount), "TRANS
    return amount;
}
```

Recommendation

Do not allow reentrant stake tokens or add a reentrancy guard to methods potentially handling tokens with callbacks. Alternatively, require a stake to be locked for at least one block, e.g., the delay between reducing stake and withdrawing stake.

5.11 Inbox/Outbox - enforce effective initialization Minor

Description

It is recommended to enforce that a contract cannot stay uninitialized after explicitly calling its `initialize()` method. For example, if the `Outbox` were to be initialized with `_rollup=address(0)`, the contract could be initialized again.

Examples

code/contracts/src/bridge/Outbox.sol:L32-L36

```
function initialize(address _rollup, IBridge _bridge) external onlyDelegated {
    if (rollup != address(0)) revert AlreadyInit();
    rollup = _rollup;
    bridge = _bridge;
}
```

code/contracts/src/rollup/RollupEventBridge.sol:L29-L33

```
function initialize(address _bridge, address _rollup) external onlyDelegated {
    require(rollup == address(0), "ALREADY_INIT");
    bridge = IBridge(_bridge);
    rollup = _rollup;
}
```

Recommendation

Enforce that function call argument `_bridge != address(0)` or else the contract stays uninitialized.

5.12 RollupUserLogic - pot. division by zero with `confirmPeriodBlocks` Minor

Description

If `confirmPeriodBlocks` is unset or set to zero, the `RollupUserLogic.currentRequiredStake()` function will revert due to a division by zero.

Neither the `RollupAdminLogic.initialize` nor `RollupAdminLogic.setConfirmPeriodBlocks` or `createRollup(..., config, ...)` functions enforce that the `confirmPeriodBlocks` is configured in a safe and non-reverting way.

Examples

code/contracts/src/rollup/RollupUserLogic.sol:L465-L467

```
uint256 firstUnresolvedAge = _blockNumber - firstUnresolvedDeadline;
uint256 periodsPassed = (firstUnresolvedAge * 10) / confirmPeriodBlocks;
uint256 baseMultiplier = 2**(periodsPassed / 10);
```

code/contracts/src/rollup/RollupAdminLogic.sol:L175-L182

```
/**
 * @notice Set number of blocks until a node is considered confirmed
 * @param newConfirmPeriod new number of blocks
 */
function setConfirmPeriodBlocks(uint64 newConfirmPeriod) external override {
    confirmPeriodBlocks = newConfirmPeriod;
    emit OwnerFunctionCalled(9);
}
```

Recommendation

Input Validation - check that `confirmPeriodBlocks` is within valid bounds.

5.13 SequencerInbox - obsolete overflow check with solidity 0.8.x Minor

Description

The `SequencerInbox.formDataHash()` function implements checks to detect if `fullDataLen` wraps in case `bytes calldata data` is too large. However, solidity `v0.8.x` standard arithmetic addition does not wrap unless placed in an `unchecked {}` block. In case of an overflow, the code would revert before the check is reached.

Examples

code/contracts/src/bridge/SequencerInbox.sol:L1-L5

```
// Copyright 2021-2022, Offchain Labs, Inc.
// For license information, see https://github.com/nitro/blob/master/LICENSE
// SPDX-License-Identifier: BUSL-1.1

pragma solidity ^0.8.0;
```

code/contracts/src/bridge/SequencerInbox.sol:L202-L209

```
function formDataHash(bytes calldata data, uint256 afterDelayedMessagesRead)
    internal
    view
    returns (bytes32, TimeBounds memory)
{
    uint256 fullDataLen = HEADER_LENGTH + data.length;
    if (fullDataLen < HEADER_LENGTH) revert DataLengthOverflow();
    if (fullDataLen > MAX_DATA_SIZE) revert DataTooLarge(fullDataLen, MAX_DATA_
```

Recommendation

It should be safe to remove the overflow check.

5.14 MerkleLib / MerkleProofLib - Inconsistent merkle tree depth check Minor

Description

Inconsistent max proof length checks

`MerkleLib` requires `proof.length <= 256` while `Outbox.recordOutputAsSpent()` requires `proof.length < 256`. Note that `Outbox.recordOutputAsSpent()` calls `MerkleLib`, thus, the `proof.length < 256` check is always enforced.

code/contracts/src/libraries/MerkleLib.sol:L28-L34

```
function calculateRoot(
    bytes32[] memory nodes,
    uint256 route,
    bytes32 item
) internal pure returns (bytes32) {
    uint256 proofItems = nodes.length;
    if (proofItems > 256) revert MerkleProofTooLong(proofItems, 256);
```

code/contracts/src/bridge/Outbox.sol:L130-L136

```
function recordOutputAsSpent(
    bytes32[] memory proof,
    uint256 index,
    bytes32 item
) internal returns (bytes32) {
    if (proof.length >= 256) revert ProofTooLong(proof.length);
```

For consistency, it is suggested to perform the same max proof depth check for both methods.

Merkle tree depth not enforced to be > 1

By indicating a zero-length proof, `Deserialize.merkleProof()` deserializes proof-data into an empty array of Merkle-proof parts.

code/contracts/src/state/Deserialize.sol:L307-L320

```
function merkleProof(bytes calldata proof, uint256 startOffset)
    internal
    pure
    returns (MerkleProof memory merkle, uint256 offset)
{
    offset = startOffset;
    uint8 length;
    (length, offset) = u8(proof, offset);
    bytes32[] memory counterparts = new bytes32[](length);
    for (uint8 i = 0; i < length; i++) {
        (counterparts[i], offset) = b32(proof, offset);
    }
    merkle = MerkleProof(counterparts);
}
```

Not providing any proof steps when computing the Merkle root for a proof is equivalent to just returning the `leafHash` in `computeRootUnsafe(proof, index, leafHash, prefix)`. The `prefix` is unused, and no additional round of hashing is performed.

code/contracts/src/state/MerkleProof.sol:L81-L98

```
// WARNING: leafHash must be computed in such a way that it cannot be a non-leaf
function computeRootUnsafe(
    MerkleProof memory proof,
    uint256 index,
    bytes32 leafHash,
    string memory prefix
) internal pure returns (bytes32 h) {
    h = leafHash;
    for (uint256 layer = 0; layer < proof.counterparts.length; layer++) {
        if (index & 1 == 0) {
            h = keccak256(abi.encodePacked(prefix, h, proof.counterparts[layer]
        } else {
            h = keccak256(abi.encodePacked(prefix, proof.counterparts[layer], h
        }
        index >>= 1;
    }
}
```

While `computeRootUnsafe` is always called with hashed data for the `leafHash` argument, one can argue, that especially for inputs with limited entropy (e.g. the combination of Instructions and their arguments) all potential pre-images can be pre-computed.

code/contracts/src/state/MerkleProof.sol:L27-L33

```
function computeRootFromInstruction(
    MerkleProof memory proof,
    uint256 index,
    Instruction memory inst
) internal pure returns (bytes32) {
    return computeRootUnsafe(proof, index, Instructions.hash(inst), "Instructio
}
```

We'd like to bring attention to this property of the system as it is unclear if one can utilize that information to cause a security-relevant impact on the system. It should be discussed, whether it would make sense to enforce that Merkle proofs always require at least one intermediary proof (`require(counterparts.length > 0)`) if - for example - the Merkle root is guaranteed to always encode > 1 elements.

5.15 Cast to enum type is implicitly bounds-checked

Description

Casting an integer to an enum will revert if the value is out of bounds. Hence, additional bounds checks are unnecessary.

Examples

- directly cast `statusU8` to `MachineStatus` will perform a boundary check

code/contracts/src/state/Machine.sol:L12-L18

```
enum MachineStatus {
    RUNNING,
    FINISHED,
    ERRORED,
    TOO_FAR
}
```

code/contracts/src/state/Deserialize.sol:L261-L273

```
uint8 statusU8;
(statusU8, offset) = u8(proof, offset);
if (statusU8 == 0) {
    status = MachineStatus.RUNNING;
} else if (statusU8 == 1) {
    status = MachineStatus.FINISHED;
} else if (statusU8 == 2) {
    status = MachineStatus.ERRORED;
} else if (statusU8 == 3) {
    status = MachineStatus.TOO_FAR;
} else {
    revert("UNKNOWN_MACH_STATUS");
}
```

- unnecessary check for `maxValueType` as `ValueType(typeInt)` will revert if it is out of bounds

code/contracts/src/state/Deserialize.sol:L100-L103

```
require(typeInt <= uint8(ValueLib.maxValueType()), "BAD_VALUE_TYPE");
uint256 contents;
(contents, offset) = u256(proof, offset);
val = Value({valueType: ValueType(typeInt), contents: contents});
```

5.16 RollupAdminLogic - avoid allowing ineffective calls for config changes

Description

Calling the `RollupAdminLogic.setValidator()` function without providing validators does not affect the system. However, it still emits an event that might trigger off-chain components to act. If such a call is performed in error - e.g., because the off-chain transaction encoding library messed up and no function argument call data was emitted, it might go undetected with the transaction not failing but succeeding instead.

Examples

code/contracts/src/rollup/RollupAdminLogic.sol:L147-L154

```
function setValidator(address[] calldata _validator, bool[] calldata _val) exte
    require(_validator.length == _val.length, "WRONG_LENGTH");

    for (uint256 i = 0; i < _validator.length; i++) {
        isValidator[_validator[i]] = _val[i];
    }
    emit OwnerFunctionCalled(6);
}
```

code/contracts/src/rollup/RollupAdminLogic.sol:L258-L273

```
function forceResolveChallenge(address[] calldata stakerA, address[] calldata s
    external
    override
    whenPaused
{
    require(stakerA.length == stakerB.length, "WRONG_LENGTH");
    for (uint256 i = 0; i < stakerA.length; i++) {
        uint64 chall = inChallenge(stakerA[i], stakerB[i]);

        require(chall != NO_CHAL_INDEX, "NOT_IN_CHALL");
        clearChallenge(stakerA[i]);
        clearChallenge(stakerB[i]);
        challengeManager.clearChallenge(chall);
    }
    emit OwnerFunctionCalled(21);
}
```

Recommendation

Revert if the method is called with no calldata to surface that this method call did not lead to any state change and was likely undesired or in error. Add `&& validator.length >0` to the input validation requirement.

5.17 GasRefundEnabled - reentrancy may theoretically allow to refund more gas than was spent

Description

The `refundsGas*` modifiers should only be used with methods that are reentrancy protected, or else an attacker might be able to reenter and potentially drain the `gasRefunder` contract of additional funds.

Note that there is no `GasRefunder.sol` in the target codebase, but we have identified a deployment on Rinkeby. The `GasRefunder` there implements a check that the refund is only applied to the first transaction in a block which should be sufficient to mitigate the attack on the gasRefunder paying out too much.

Examples

code/contracts/src/libraries/IGasRefunder.sol:L15-L34

```
abstract contract GasRefundEnabled {
    modifier refundsGasWithCalldata(IGasRefunder gasRefunder, address payable s
        uint256 startGasLeft = gasleft();
        _;
        if (address(gasRefunder) != address(0)) {
            uint256 calldataSize;
            assembly {
                calldataSize := calldatasize()
            }
            gasRefunder.onGasSpent(spender, startGasLeft - gasleft(), calldataS
        }
    }

    modifier refundsGasNoCalldata(IGasRefunder gasRefunder, address payable spe
        uint256 startGasLeft = gasleft();
        _;
        if (address(gasRefunder) != address(0)) {
            gasRefunder.onGasSpent(spender, startGasLeft - gasleft(), 0);
        }
    }
```

Recommendation

If the gas refunding functionality is not used anymore, it should be removed from the code base. Alternatively, tests and procedures should be implemented to avoid using the modifier in a context that is not protected against reentrancy attacks.

5.18 Unused constants

Description

The following constants are not referenced anywhere in the codebase. Note that if they are to be used, we suggest using an enum instead (https://github.com/ConsenSysDiligence/arbitrum-nitro-audit-2022-04/issues/2).

Examples

code/contracts/src/rollup/RollupEventBridge.sol:L15-L19

```
contract RollupEventBridge is IMessageProvider, DelegateCallAware {
    uint8 internal constant CREATE_NODE_EVENT = 0;
    uint8 internal constant CONFIRM_NODE_EVENT = 1;
    uint8 internal constant REJECT_NODE_EVENT = 2;
    uint8 internal constant STAKE_CREATED_EVENT = 3;
```

Recommendation

Remove unused constants.

5.19 Contract file system layout, source-unit structure, naming

Description

The project layout, source-unit structure, and naming conventions diverge from best practices in smart contract development.

Recommendations

We recommend the following actions to clean up the code base and make it more maintainable:

- Developers should keep contract interface declarations separately from contract implementations, e.g., in an `interfaces` subfolder. A similar approach should be taken for abstract contracts.
- If a contract/interface already exists, no sub-interfaces should be derived from it.
- Each source unit should only contain a single contract, e.g., `RollupUserLogic.sol`.
- Instead of undescriptive integers, Enums should be used:

code/contracts/src/rollup/RollupEventBridge.sol:L16-L19

```
uint8 internal constant CREATE_NODE_EVENT = 0;
uint8 internal constant CONFIRM_NODE_EVENT = 1;
uint8 internal constant REJECT_NODE_EVENT = 2;
uint8 internal constant STAKE_CREATED_EVENT = 3;
```

code/contracts/src/rollup/RollupAdminLogic.sol:L153

```
emit OwnerFunctionCalled(6);
```

code/contracts/src/rollup/RollupAdminLogic.sol:L163

```
emit OwnerFunctionCalled(7);
```

- Consider grouping public interfaces and internal/private interfaces together. E.g., public interfaces first, private/internal afterward.
- The contract should define modifiers at the top. They should not mix with public/private functions:

code/contracts/src/challenge/ChallengeManager.sol:L39-L56

```
function challengeInfo(uint64 challengeIndex)
    external
    view
    override
    returns (ChallengeLib.Challenge memory)
{
    return challenges[challengeIndex];
}

modifier takeTurn(
    uint64 challengeIndex,
    ChallengeLib.SegmentSelection calldata selection,
    ChallengeModeRequirement expectedMode
) {
    ChallengeLib.Challenge storage challenge = challenges[challengeIndex];
    require(msg.sender == currentResponder(challengeIndex), "CHAL_SENDER");
    require(!isTimedOut(challengeIndex), "CHAL_DEADLINE");
```

- Consider distinguishing non-public methods from public interfaces visually by prefixing non-public methods with an underscore. For example, both, `_deliverMessage` and `deliverToBridge` are `internal` methods but one is prefixed with an underscore while the other is not.

code/contracts/src/bridge/Inbox.sol:L394-L413

```
function _deliverMessage(
    uint8 _kind,
    address _sender,
    bytes memory _messageData
) internal returns (uint256) {
    if (_messageData.length > MAX_DATA_SIZE)
        revert DataTooLarge(_messageData.length, MAX_DATA_SIZE);
    uint256 msgNum = deliverToBridge(_kind, _sender, keccak256(_messageData));
    emit InboxMessageDelivered(msgNum, _messageData);
    return msgNum;
}

function deliverToBridge(
    uint8 kind,
    address sender,
    bytes32 messageDataHash
) internal returns (uint256) {
    return bridge.enqueueDelayedMessage{value: msg.value}(kind, sender, message
}
```

5.20 Missing NatSpec

Description

Not all external functions have NatSpec annotations. To quote the Solidity documentation: “It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI).”

Recommendation

We strongly recommend adding NatSpec annotations to every contract and every public or external function. Furthermore, critical internal and private functions should be documented with NatSpec to increase maintainability and reduce the potential for future developer errors. NatSpec documentation should primarily be enforced in the sensitive and complex components, such as proof- and challenge-related smart contracts.

6 Other Code

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

6.1 Validator - unsafe default file/folder permissions can lead to racy privilege escalation via shell-command injection Critical

Description

`validator/block-validator::writeToFile` creates a temporary shell script performing block validation. The temporary script is stored in a sub-directory of `machConf.RootPath`. This root folder is relative to the `pwd` and not necessarily within the ACL-protected location of the user's home directory.

code/validator/nitro_machine.go:L38-L42

```
var DefaultNitroMachineConfig = NitroMachineConfig{
        RootPath:             "./target/machines/",
        WavmBinaryPath:       "machine.wavm.br",
        UntilHostIoStatePath: "until-host-io-state.bin",
```

`BlockValidator::validate` calls `BlockValidator::writeToFile` which creates the directory structure with permissions `0777` (`-rwxrwxrwx`). Note that this grants anyone **read, write, and execute** permissions.

code/validator/block_validator.go:L262-L268

```
func (v *BlockValidator) writeToFile(validationEntry *validationEntry, moduleRo
        machConf := v.MachineLoader.GetConfig()
        outDirPath := filepath.Join(machConf.RootPath, v.config.OutputPath, lau
        err := os.MkdirAll(outDirPath, 0777)
        if err != nil {
                return err
        }
```

The validator then creates a new file in the newly created folder using `os.Create`, which creates files with permission `0666` (`-rw-rw-rw-`) in the folder. Note that this grants all users **read, write** access to that file. Note that all permissions are considered to be **before system-wide umask** settings.

code/validator/block_validator.go:L270-L273

```
cmdFile, err := os.Create(filepath.Join(outDirPath, "run-prover.sh"))
if err != nil {
        return err
}
```

An unprivileged local user may exploit these permissive defaults by monitoring the folder for newly created shell scripts, injecting their shell commands, and then waiting for the privileged user to execute it. This may allow an unprivileged user to execute shell commands as the validator user.

Finally, the script is explicitly set to `0777` (`-rwxrwxrwx`).

code/validator/block_validator.go:L354-L357

```
err = cmdFile.Chmod(0777)
if err != nil {
        return err
}
```

Recommendation

If there is no reason for others/group members to access or modify the shell script, it is highly recommended to restrict the permissions to only allow the `owner` to `rwx` the file (`0700`). Files not to be executed should be set to `chmod 0600`. Default folder permissions should be set to `chmod 0600`.

Avoid creating a shell script in the first place, as this may always be subject to time-of-check vs. time-of-use-related issues. It is generally less reliable than running the prover directly.

6.2 Arbnode - OOB read panic in `parseSequencerMessage()` Major

Description

A specially crafted message may lead to an OOB slice/array access in Arbnode's `parseSequencerMessage()`, potentially causing all nodes processing the message to shut down in panic.

`parseSequencerMessage()` parses data obtained from an event to `SequencerInbox.sol::addSequencerL2Batch()`. There are no input validation checks on the data emitted from that call. However, the call is access-restricted with `isBatchPoster[msg.sender]` (we found a reference that the sequencer might be a batch poster) and rollup (`RollupAdminLogic`). This leads to the assumption that - with the current system - data can only be posted by trusted entities, which led to the current severity rating.

Examples

Suppose the payload is hinting zeroHeavy encoded data, but `io.ReadAll` fails to decode the presumably zeroHeavy encoded blob. In that case, a warning message is emitted, and `payload` is set to a zero-length slice. This zero-length slice is then attempted to be accessed at index 1, which leads to an out-of-bounds access panic with the go runtime.

code/arbstate/inbox.go:L86-L96

```
if len(payload) > 0 {
        if IsZeroheavyEncodedHeaderByte(data[40]) {
                pl, err := io.ReadAll(io.LimitReader(zeroheavy.NewZeroheavyDecc
                if err != nil {
                        log.Warn("error reading from zeroheavy decoder", err.Er
                        pl = []byte{}
                }
                payload = pl
        }
        decompressed, err := arbcompress.Decompress(payload[1:], maxDecompresse
        if err == nil {
```

Recommendation

Abort handling of the payload if reading the payload fails.

6.3 seq_coordinator - `log.Crit(...)` terminates without executing deferred functions or returning from function calls

Medium

Description

Multiple locations in `seq_coordinator.go` emit **critical** log messages and return a value on error. However, code after `log.crit(...)` will not be executed as the method calls `os.Exit(1)` internally terminating the process (see go-ethereum/log) before returning a value.

Returning a value after calling `log.Crit(...)` may indicate that the terminating side-effect was not intended.

Examples

- `log.Crit(...)` terminates before returning a value

code/arbnode/seq_coordinator.go:L601-L604

```
if c.sequencer == nil {
        log.Crit("myurl main sequencer, but no sequencer exists")
        return c.noRedisError()
}
```

code/arbnode/seq_coordinator.go:L534-L537

```
if err != nil {
        log.Crit("cannot read message count", "err", err)
        return c.config.UpdateInterval
}
```

code/arbnode/seq_coordinator.go:L502-L505

```
if err != nil {
        log.Crit("coordinator cannot read message count", "err", err)
        return c.config.UpdateInterval
}
```

Recommendation

Decide whether to terminate and print a log message, bubble up the error, or return sane defaults if the condition is recoverable. Consider adding inline comments surfacing that `log.Crit(...)` terminates the process, and this behavior is intentional. Avoid using `log.Crit()` in general and terminate the process in a controlled way, explicitly, or avoid calling `log.Crit(...)` (or similar terminating methods) with `defer` statements, especially if external resources are being locked.

6.4 seq_coordinator - pot. resource exhaustion DoS due to unset `http.server` timeouts Medium

Description

The sequence coordinator can be configured to spawn a health check server. The server module used is `http.Server{}`, and no read/write timeout is set. Not setting timeouts can become problematic as malicious users may connect to the service, deliberately keeping the connection open and consuming server resources (sockets/file-descriptors). Resource exhaustion may eventually lead to a denial-of-service condition of the health check service or unspecified other

issues on the affected system (because file/socket open/close operations suddenly fail).

The following article provides more insights into why timeouts must be set: the-complete-guide-to-golang-net-http-timeouts.

Examples

- `&http.Server{}` defaults to timeouts being disabled if Read/Write/Idle-timout are not set

code/arbnode/seq_coordinator.go:L666-L684

```
func (c *SeqCoordinator) launchHealthcheckServer(ctx context.Context) {
        server := &http.Server{
                Addr:    c.config.ChosenHealthcheckAddr,
                Handler: seqCoordinatorChosenHealthcheck{c},
        }

        go func() {
                <-ctx.Done()
                err := server.Shutdown(ctx)
                if err != nil && !errors.Is(err, context.Canceled) && !errors.I
                        log.Warn("error shutting down coordinator chosen healtl
                }
        }()

        err := server.ListenAndServe()
        if err != nil && !errors.Is(err, http.ErrServerClosed) {
                log.Warn("error serving coordinator chosen healthcheck server",
        }
}
```

Recommendation

Set reasonable read/write timeouts.

```
srv := &http.Server{
    ReadTimeout: 5 * time.Second,
    WriteTimeout: 10 * time.Second,
}
```

6.5 dasrpcserver - unauthenticated read/write from localhost

Medium

Description

The `daserver` command spawns a gRPC daemon that listens on `tcp://localhost:<port>`. The transport is unauthenticated, and there is no user access control for read/write operations.

- listens on `localhost:<port>` without transport security

On a multi-user system, there is no way for a client consuming the API to verify the authenticity of the gRPC server. Other - potentially less privileged - users may be able to start a service on the expected port (depending on port range and system privileges) before the actual services does, mimicking a valid but malicious `dasrpcserver`. The attacker is in complete control of the communication channel. We, therefore, suggest enabling transport security and - at least - requiring the service to authenticate to potential clients (e.g., TLS server auth; configuring a gRPC TLS server certificate).

code/das/dasrpc/dasRpcServer.go:L21-L23

```
func StartDASRPCServer(ctx context.Context, portNum uint64, localDAS das.DataAv
        grpcServer := grpc.NewServer()
        listener, err := net.Listen("tcp", fmt.Sprintf("localhost:%d", portNum)
```

- exposes read/write API endpoints without access control

The gRPC service does not provide any means for user access control. Any gRPC client that can connect to `localhost:<port>` - including unprivileged local users - may read/write to the database.

code/das/dasrpc/dasRpcServer.go:L45-L64

```
func (serv *DASRPCServer) Store(ctx context.Context, req *StoreRequest) (*Store
        cert, err := serv.localDAS.Store(ctx, req.Message)
        if err != nil {
                return nil, err
        }
        return &StoreResponse{
                DataHash:    cert.DataHash[:],
                Timeout:     cert.Timeout,
                SignersMask: cert.SignersMask,
                Sig:         blsSignatures.SignatureToBytes(cert.Sig),
        }, nil
}

func (serv *DASRPCServer) Retrieve(ctx context.Context, req *RetrieveRequest) (
        result, err := serv.localDAS.Retrieve(ctx, req.Cert)
        if err != nil {
                return nil, err
        }
        return &RetrieveResponse{Result: result}, nil
```

Recommendation

- Enable transport security for gRPC. Configure a TLS server certificate and allow clients to verify the server's authenticity.
- Require client authentication (e.g., a simple pre-shared secret used with a challenge-response mechanism), or else anyone on the local machine with access to `localhost:<port>` may be able to read/write data.

6.6 broadcaster - OnDoBroadcast never returns an error condition Minor

Description

`SequenceNumberCatchupBuffer.OnDoBroadcast` declares to return an error, however, all `return` statements (even for error conditions) hardcode the `error` return value to `nil` and in one case the broadcaster terminates when handling an unknown message type (`log.crit()`).

code/broadcaster/broadcaster.go:L86-L97

```
func (b *SequenceNumberCatchupBuffer) OnDoBroadcast(bmi interface{}) error {
        broadcastMessage, ok := bmi.(BroadcastMessage)
        if !ok {
                log.Crit("Requested to broadcast messasge of unknown type")
        }
        defer func() { atomic.StoreInt32(&b.messageCount, int32(len(b.messages)

        if confirmMsg := broadcastMessage.ConfirmedSequenceNumberMessage; confi
                if len(b.messages) == 0 {
                        return nil
                }
```

code/broadcaster/broadcaster.go:L111-L117

```
if b.messages[confirmedIndex].SequenceNumber != confirmMsg.SequenceNumber {
        // Log instead of returning error here so that the message will be sent
        // relays to also cause them to be cleared.
        log.Error("Invariant violation: Non-sequential messages stored in Seque
        b.messages = nil
        return nil
}
```

used by: `wsbroadcastsever`

code/wsbroadcastserver/clientmanager.go:L136-L139

```
func (cm *ClientManager) doBroadcast(bm interface{}) error {
        if err := cm.catchupBuffer.OnDoBroadcast(bm); err != nil {
                return err
        }
```

Recommendation

If errors are not returned to the caller and are always hardcoded to `nil` there is no reason to return an error. Consider returning an indicator of errors and allow the caller to handle them.

6.7 precompile - failing to pack return values causes node to panic Minor

Description

Panics are generally problematic for systems that require nodes to provide reliable services. An implicit or explicit panic, `log.crit()`, or `log.fatal()` may shut down the node and thus weaken the security of the overall system. In production systems, it is recommended to avoid causing panic conditions in critical services or execution flows that may be exposed to external and potentially untrusted entities.

Examples

code/precompiles/precompile.go:L678-L683

```
encoded, err := method.template.Outputs.PackValues(result)
if err != nil {
        // in production we'll just revert, but for now this
        // will catch implementation errors
        log.Fatal("Could not encode precompile result ", err)
}
```

Recommendation

This specific example is less likely and may be less problematic as the panic should only happen if there is an implementation issue with the precompile. However, as the inline-comment states, it is recommended to implement a global switch that indicates whether the application is running in production or debug mode. Panics and fatal logs should only be raised in the latter manner to reduce the risk of production nodes terminating in case such an event occurs.

6.8 Handle recoverable exceptions in off-chain applications

Description

In the case of an attacker managing to force a Go runtime error or exception in an off-chain application, the component (e.g. Arbitrum Node) will not automatically restart itself and resume its duties to protect the protocol. The

issue applies to off-chain community applications such as validators and privileged ones like the sequencer.

Node liveness is essential, and failure to prove liveness during challenges may be punished or put the network at risk. The worst case might be someone adding information to a smart contract that causes a Go panic in the client implementation (also see https://github.com/ConsenSysDiligence/arbitrum-nitro-audit-2022-04/issues/23). This attack can destabilize the network, providing opportunities for malicious actors to profit.

Therefore, segmenting the application into recoverable and non-recoverable zones is highly recommended. For example, a recoverable exception is `parseInt` trying to parse user-tainted data and failing to do so. The application could continue in such a case by skipping this entry. The application should print the stack trace, log the event, and make the user aware. Another example would be a panic that may be triggered by a malicious client accessing the node RPC service. In production builds, any panic thrown in non-vital services like an informational RPC service should allow the service to recover. In any case, the application must continue providing its services to support the network's security.

For example, an application can achieve recoverability in Go by catching panic conditions using the Defer, Panic, Recover pattern.

6.9 Use of cgo C native library interface for interop

Description

Both Golang and Rust are memory-safe programming languages - at least at compile time. The Arbitrator prover is written and compiled in Rust and exposed as a C native library. The Arbitrum node is written in Golang and links the Rust library using the `cgo` native library interface, a low-level interface that bypasses the compilers' memory safety features. For example, data and memory pointers managed by one language must be passed to the native module in an unsafe fashion while at the same time ensuring that garbage collection does not free presumably unreferenced memory pointers that are still in use by a component. This scenario happens, e.g., when Go unmanaged data is passed into the Rust-compiled library. Furthermore, types must be converted to low-level C native

types, and the originating objects' location (stack/heap) may suddenly become important (memory might get accidentally freed). Note that the native library is not isolated from the calling Go code. A panic in the prover code may lead to the Arbitrum node terminating in error (https://github.com/ConsenSysDiligence/arbitrum-nitro-audit-2022-04/issues/25).

Examples

Assuming manual control of garbage collection

Runtime magic ensures that `m` is not garbage collected while it is being accessed from the Rust component, leading to inconsistent use:

- `AddPreimages` does not mark `m` as currently reachable

code/validator/machine.go:L265-L275

```
func (m *ArbitratorMachine) AddPreimages(preimages map[common.Hash][]byte) erro
        for _, val := range preimages {
                cbyte := CreateCByteArray(val)
                status := C.arbitrator_add_preimage(m.ptr, cbyte)
                DestroyCByteArray(cbyte)
                if status != 0 {
                        return errors.New("failed to add sequencer inbox messag
                }
        }
        return nil
}
```

- `AddDelayedInboxMessage` marks `m` as currently reachable

code/validator/machine.go:L248-L253

```
func (m *ArbitratorMachine) AddDelayedInboxMessage(index uint64, data []byte) e
        defer runtime.KeepAlive(m)

        if m.frozen {
                return errors.New("machine frozen")
        }
```

Inconsistent check for `frozen`

`addPreimages` is missing a check for `m.frozen`. The method should not allow adding preimages if the machine is frozen.

code/validator/machine.go:L265-L275

```
func (m *ArbitratorMachine) AddPreimages(preimages map[common.Hash][]byte) erro
        for _, val := range preimages {
                cbyte := CreateCByteArray(val)
                status := C.arbitrator_add_preimage(m.ptr, cbyte)
                DestroyCByteArray(cbyte)
                if status != 0 {
                        return errors.New("failed to add sequencer inbox messag
                }
        }
        return nil
}
```

Low-level manual memory management and reference tracking

code/validator/machine.go:L203-L206

```
cPath := C.CString(path)
status := C.arbitrator_serialize_state(m.ptr, cPath)
C.free(unsafe.Pointer(cPath))
```

High complexity and down-cast to unsafe/unmanaged types

Using the C native interface to glue code compiled with different compilers together introduces complexity, bypasses language safety features when passing data from one to the other component and, therefore, may increase the risk of introducing errors that may have severe consequences for the security of the system. For example, due to Go being a garbage collected language, failing to prevent the garbage collector form freeing manually managed memory allocations that are still in use may lead to problems that manifest unpredictably during runtime.

To illustrate the complexity introduced by combining the two different languages within one application we are looking at the Go method `createZeroStepMachineInternal()`. The method interfaces with code compiled in Rust to load a wavm binary, initializing and returning a pointer to machine object. This will result in the following sequence of events:

1. `createZeroStepMachineInternal` loads the wavm binary via `arbitrator_load_wavm_binary`, a method that is exposed in the linked Rust prover library.
2. The method requires the path to the wavm binary to be passed as the first argument. Since the two languages communicate via the C native library interface, the argument must be provided as a manually managed `CString`. Therefore, the Go String variable `binPath` is duplicated to an unmanaged `CString` - basically a down-cast to an unmanaged low-level string. This string must be manually freed when it is not being used anymore.
3. In the Rust code, `arbitrator_load_wavm_binary` retrieves the `CString` and converts it to a Rust string object and then proceeds creating the wavm machine.
4. Eventually, a pointer to the machine object is manually stored in the Heap via `Box::into_raw(Box::new(mach))` and a pointer to the heap structure is returned to the calling Go code. Note that the `box_rawptr` heap pointer is manually managed and must be manually freed.
5. When the Go struct is destroyed, the Go finalizer routine calls the Rust library method `arbitrator_free_machine(baseMachine)` to finally free the heap allocation.

code/validator/nitro_machine.go:L84-L100

```
func (s *loaderMachineStatus) createZeroStepMachineInternal(config NitroMachine
        defer s.signalReady()
        binPath := filepath.Join(config.getMachinePath(moduleRoot), config.Wavm
        cBinPath := C.CString(binPath)
        defer C.free(unsafe.Pointer(cBinPath))
        log.Info("creating nitro machine", "binpath", binPath)
        baseMachine := C.arbitrator_load_wavm_binary(cBinPath)
        if baseMachine == nil {
                s.err = errors.New("failed to load base machine")
                return
        }
        nitroMachine := machineFromPointer(baseMachine)
        machineModuleRoot := nitroMachine.GetModuleRoot()
        if machineModuleRoot != realModuleRoot {
                s.err = fmt.Errorf("attempting to load module root %v got machi
                return
        }
```

code/arbitrator/prover/src/lib.rs:L130-L139

```
pub unsafe extern "C" fn arbitrator_load_wavm_binary(binary_path: *const c_char
    let binary_path = cstr_to_string(binary_path);
    let binary_path = Path::new(&binary_path);
    match Machine::new_from_wavm(binary_path) {
        Ok(mach) => Box::into_raw(Box::new(mach)),
        Err(err) => {
            eprintln!("Error loading binary: {}", err);
            std::ptr::null_mut()
        }
    }
```

code/arbitrator/prover/src/lib.rs:L142-L144

```
unsafe fn cstr_to_string(c_str: *const c_char) -> String {
    CStr::from_ptr(c_str).to_string_lossy().into_owned()
}
```

Recommendation

Using the C native library interface with all the caveats introduces risk and complexity to a system that manages significant value. As such, risk and

complexity should always be reduced to the absolute minimum, low-level interaction should be avoided, and the application should be designed in a way that it can recover from recoverable exceptions (critical/non-critical sections, use of `recover()` for Go panics, ensure the rust lib does not panic-terminate the Go host for potentially user reachable panics, avoid Go `unsafe` modules)

Appendix 1 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not

extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

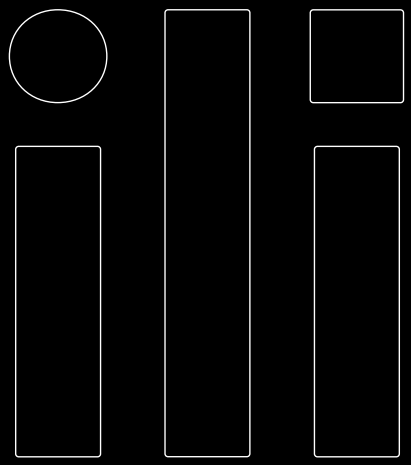
CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.

Get in touch with our team to request a quote for a smart contract audit.

CONTACT US



AUDITS

FUZZING

SCRIBBLE

BLOG

TOOLS

RESEARCH

ABOUT

CONTACT

CAREERS

PRIVACY POLICY

Subscribe to Our Newsletter

Stay up-to-date on our latest offerings, tools, and the world of blockchain security.

POWERED BY CONSENSYS