Arbitrum Governor Cancel Upgrade Review

October 19, 2025

Prepared for Dapp Hero

Conducted by:

Richie Humphrey (devtooligan)

About the Arbitrum Governor Cancel Upgrade Review

The Arbitrum Governor Cancel Upgrade project implements a governance upgrade that introduces proposal cancellation functionality to the existing Arbitrum DAO governance system. The upgrade modifies both the Core and Treasury governors through a multi-proxy upgrade mechanism, using Arbitrum's L2-to-L1 and L1-to-L2 messaging infrastructure to coordinate the cross-chain governance action.

About Offbeat Security

Offbeat Security is a boutique security company providing unique security solutions for complex and novel crypto projects. Our mission is to elevate the blockchain security landscape through invention and collaboration.

Summary & Scope

The src and scripts/forge-scripts folders were reviewed at commit df184f2.

The following **7 files** were in scope:

- src/L2ArbitrumGovernorV2.sol
- src/gov-action-contracts/gov-upgrade-contracts/upgrade-proxy/MultiProxyUpgradeAction.sol
- scripts/forge-scripts/DeployConstants.sol
- scripts/forge-scripts/DeployImplementation.s.sol
- scripts/forge-scripts/DeployMultiProxyUpgradeAction.s.sol
- scripts/forge-scripts/SubmitUpgradeProposalScript.s.sol
- scripts/forge-scripts/utils/EncodeL2ArbSysProposal.sol

Summary of Findings

The review identified 1 INFORMATIONAL severity issue related to deployment script validation. No critical security vulnerabilities were found in the upgrade implementation or proposal generation logic.

Identifier	Title	Severity	Fixed
I-01	Deployment script lacks pre-flight ownership assertions	Informational	

Proposal Calldata Formation Review

- We conducted a thorough review of the calldata formation for the proposal and validated all related addresses - complete details can be found in Appendix A: Calldata Formation Analysis

Additional Notes

- **Cancel functionality cut-over behavior**: The new `cancel()` function checks a proposer mapping that will only be populated for proposals created after the upgrade, meaning any existing pending proposals at the time of upgrade will not be cancelable through the new mechanism.

- **Retryable ticket parameters**: The L1→L2 retryable can fail to be created or auto-redeemed if `submissionCost`, `gasLimit`, `maxFeePerGas`, or `msg.value` are mis-sized, which would stall execution until the ticket is manually redeemed. We recommend establishing an operations checklist to compute submission cost using the Inbox helper, set gas limits with margin, ensure sufficient ETH is sent from the L1 Timelock, and document manual redeem as a fallback procedure.

Detailed Findings

Informational Findings

[I-01] Deployment script lacks pre-flight ownership assertions

Description

The proposal deployment script doesn't include pre-flight checks to verify ownership relationships before submitting the upgrade proposal on-chain. Since the upgrade relies on proper configuration between the Upgrade Executor, ProxyAdmin, and Governor proxies, any misconfigurations would cause the perform() call to revert when executed. Currently, the script goes straight to calling Governor.propose(...) without checking that:

- The Upgrade Executor owns the ProxyAdmin
- Both Governor proxies are administered by the ProxyAdmin
- The proposal payload structure is correctly formed
-

Recommendation

Consider adding read-only assertions to the proposal script before submitting the proposal. These checks can catch configuration errors early and prevent wasted gas on proposals that will fail during execution:

```
// 1) Verify the Upgrade Executor owns the ProxyAdmin
require(
    ProxyAdmin(L2_PROXY_ADMIN_CONTRACT).owner() == L2_UPGRADE_EXECUTOR,
    "ProxyAdmin owner mismatch"
);

// 2) Verify both Governor proxies are administered by the ProxyAdmin
require(
    ProxyAdmin(L2_PROXY_ADMIN_CONTRACT).getProxyAdmin(payable(L2_CORE_GOVERNOR))
    "Core Governor proxy admin mismatch"
);
require(
    ProxyAdmin(L2_PROXY_ADMIN_CONTRACT).getProxyAdmin(payable(L2_TREASURY_GOVERN
    "Treasury Governor proxy admin mismatch"
);

// 3) Validate the proposal payload structure
(address[] memory targets, uint256[] memory values, bytes[] memory calldatas) =
    encodeL2ArbSysProposal(PROPOSAL_DESCRIPTION, _multiProxyUpgradeAction, _minD
require(
    targets.length == 1 && targets[0] == L2_ARB_SYS &&
    values.length == 1 && values[0] == 0 &&
    calldatas.length == 1,
    "Invalid proposal arrays"
);
```

Appendix A: Calldata Formation Analysis

Arbitrum Governor Upgrade — Calldata Formation Report (Final, with Address Validation)

Scope

- `SubmitUpgradeProposalScript.s.sol` (proposal creation)
- `EncodeL2ArbSysProposal.sol` (nested calldata encoding)
- `DeployConstants.sol` (addresses)
- `MultiProxyUpgradeAction.sol` (delegate-called action)

---

1) End-to-end flow (compact diagram)

```
L2 (Arbitrum One):
==================
Governor (L2)
└─ after L2 timelock
   propose(): target = ArbSys (0x…0064)
   data = sendTxToL1(L1 Timelock,
          Timelock.schedule(
            sentinel_for_retryable, 0,
            abi.encode(Inbox, L2 UpgradeExecutor,
                       0,0,0, UpgradeExecCalldata),
            0x00, keccak256(desc), minDelay))
                          |
                          ▼
L1 (Ethereum):
==============
L1 Timelock
└─ after L1 timelock
   execute(): create L1→L2 retryable
   to: L2 UpgradeExecutor
   data: UpgradeExecutor.execute(
           upgrade = MultiProxyUpgradeAction,
           calldata = perform()
         )
                          |
                          ▼
L2 (Arbitrum One):
==================
L2 UpgradeExecutor.execute(...)
```

```
└─ delegatecall MultiProxyUpgradeAction.perform()
   ├─ ProxyAdmin.upgrade(Core Governor proxy, new impl)
   └─ ProxyAdmin.upgrade(Treasury Governor proxy, new impl)
```

L2→L1 uses the `ArbSys` precompile; L1→L2 uses the Inbox retryable; L2 execution uses the Upgrade Executor.
Docs: DAO deployment addresses, Precompiles reference.

---

2) L2 proposal builds a single `ArbSys` call

SubmitUpgradeProposalScript.s.sol (essential excerpt)

```
(_targets, _values, _calldatas) =
  encodeL2ArbSysProposal(_description, _multiProxyUpgradeAction, _minDelay);

_proposalId = GovernorUpgradeable(payable(L2_CORE_GOVERNOR))
  .propose(_targets, _values, _calldatas, _description);
```

This creates **one** L2 action targeting `ArbSys (0x…0064)` with calldata destined for the **L1 Timelock**.
Reference: ArbSys precompile at `0x000...064` in docs.

---

3) `EncodeL2ArbSysProposal`: nested payloads

(a) Final L2 payload for the executor

```
bytes memory upgradeExecutorCallData = abi.encodeWithSelector(
  IUpgradeExecutor.execute.selector,
  _oneOffUpgradeAddr,                                   // MultiProxyUpgrade
  abi.encodeWithSelector(IMultiProxyUpgradeAction.perform.selector)
);
```

This encodes `UpgradeExecutor.execute(address to, bytes data)` so that, on L2, the executor **delegatecalls** the action's `perform()`.

(b) L1 Timelock payload (schedules a retryable)

```
bytes memory l1TimelockData = abi.encodeWithSelector(
  IL1Timelock.schedule.selector,
  L2_RETRYABLE_TICKET_MAGIC, // sentinel meaning "create a retryable"
  0,                         // value ignored for L2 upgrades
  abi.encode(                // retryable parameters
    L1_ARB_ONE_DELAYED_INBOX,
    L2_UPGRADE_EXECUTOR,
    0,                          // l2CallValue
    0,                          // gasLimit (set at execution)
    0,                          // maxFeePerGas (set at execution)
    upgradeExecutorCallData  // calldata to L2 executor
  ),
  bytes32(0),
  keccak256(abi.encodePacked(_proposalDescription)),
  _minDelay
);
```

© Outermost payload to `ArbSys` (L2→L1)

```
proposalCalldata = abi.encodeWithSelector(
  IArbSys.sendTxToL1.selector,
  L1_TIMELOCK,
  l1TimelockData
);
```

`ArbSys.sendTxToL1(address,bytes)` is the standard child→parent message on Arbitrum.

---

4) L1 execution creates a retryable back to L2

When the L1 Timelock executes the scheduled call, it forwards (per the sentinel) to the **Delayed Inbox** so that a **retryable ticket** is created:

- **Destination (`to`)**: `L2_UPGRADE_EXECUTOR`
- **Calldata**: `upgradeExecutorCallData` (from §3a)
- **Funding/gas**: the Inbox call must provide sufficient **submission cost + redeem gas** for reliable auto-redeem; otherwise manual redeem is required.

Docs: Contract addresses → Cross-chain messaging → Delayed Inbox, Precompiles reference.

---

5) L2 receives the retryable → executor runs the action

The L2 Upgrade Executor's `execute(address,bytes)` **delegatecalls** the action. The action upgrades both proxies via ProxyAdmin and asserts the final `implementation()` for each proxy equals the new implementation.

MultiProxyUpgradeAction.sol (post-checks excerpt)

```
require(
  ProxyAdmin(payable(PROXY_ADMIN))
    .getProxyImplementation(TransparentUpgradeableProxy(payable(CORE_GOVERNOR_ADI
      == NEW_GOVERNOR_IMPLEMENTATION_ADDRESS,
  "Core Governor not upgraded"
);
require(
  ProxyAdmin(payable(PROXY_ADMIN))
    .getProxyImplementation(TransparentUpgradeableProxy(payable(TREASURY_GOVERNOI
      == NEW_GOVERNOR_IMPLEMENTATION_ADDRESS,
  "Treasury Governor not upgraded"
);
```

---

6) Address validation (constants vs published registries)

Source of constants in code base: `scripts/forge-scripts/DeployConstants.sol` *(reference link provided by SR)*

The table below cross-checks each constant against Arbitrum's public registries and references.

Constant	Value (hex)	Public reference	Result
`L2_ARB_TOKEN_ADDRESS`	`0x912...E6548`	DAO addresses → **$ARB Token (Arb One)**	**Matches**
`L2_CORE_GOVERNOR`	`0xf07...395B9`	DAO addresses → **Core Governor (Arb One)**	**Matches**
`L2_CORE_GOVERNOR_TIMELOCK`	`0x34d...98f0`	DAO addresses → **L2 Core Timelock**	**Matches**
`L2_TREASURY_GOVERNOR`	`0x789...e5a4`	DAO addresses → **Treasury Governor (Arb One)**	**Matches**
`L2_TREASURY_GOVERNOR_TIMELOCK`	`0xbFc...EF58`	DAO addresses → **L2 Treasury Timelock**	**Matches**
`L2_PROXY_ADMIN_CONTRACT`	`0xdb2...961e`	DAO addresses → **Arb One Proxy Admin**	**Matches**
`L2_ARB_SYS`	`0x000...00064`	Precompiles ref → **ArbSys** (`0x64`)	**Matches**
`L2_ARB_RETRYABLE_TX`	`0x000...0006E`	Precompiles ref → **ArbRetryableTx** (`0x6e`)	**Matches**
`L2_SECURITY_COUNCIL_9`	`0x423...A1641`	DAO addresses → **Security Council (Arb One, emergency)**	**Matches**

`L1_TIMELOCK`	`0xE68...C7f49`	DAO addresses → **L1 Timelock**	**Matches**
`L1_ARB_ONE_DELAYED_INBOX`	`0x4Db...BAB3f`	Contract addresses → **Delayed Inbox (Arbitrum One)**	**Matches**
`L2_UPGRADE_EXECUTOR`	`0xCF5...0A827`	DAO addresses → **Arb One Upgrade Executor**	**Matches**
`L2_RETRYABLE_TICKET_MAGIC`	`0xa72...Fa79C`	— (DAO-specific sentinel)	**DAO-specific (no public ref)**

References

- DAO contract addresses (Core/Treasury Governor, L2/L1 Timelocks, Upgrade Executor, Proxy Admin, Security Council): https://docs.arbitrum.foundation/deployment-addresses
- Precompiles reference (ArbSys `0x64`, ArbRetryableTx `0x6e`): https://docs.arbitrum.io/build-decentralized-apps/precompiles/reference
- Cross-chain messaging contracts (Delayed Inbox): https://docs.arbitrum.io/build-decentralized-apps/reference/contract-addresses

---

7) Sanity checks

- **L2 proposal**: arrays length = 1; `targets[0] == L2_ARB_SYS`; `values[0] == 0`; `calldatas[0]` decodes to `sendTxToL1(L1_TIMELOCK, <bytes>)`.
- **L1 payload**: decodes to `Timelock.schedule(target, value, data, predecessor, salt, delay)` with `target == L2_RETRYABLE_TICKET_MAGIC` and `data` = `(Inbox, L2_UPGRADE_EXECUTOR, l2CallValue, gasLimit, maxFeePerGas, upgradeExecutorCalldata)`.
- **Retryable**: sufficient funding for submission + redeem; if auto-redeem fails, manual redeem is available.
- **L2 executor**: decodes to `execute(_oneOffUpgradeAddr, abi.encodeWithSelector(perform()))`; post-checks confirm both proxies' implementations match the new address.

TL;DR

The script constructs one L2 action that uses `ArbSys.sendTxToL1` to call `Timelock.schedule` on L1 with parameters that instruct the L1 Timelock to create an L1→L2 retryable targeting the L2 Upgrade Executor. The executor `delegatecall`s the action's `perform()`, which upgrades both Governor proxies and verifies the result. All addresses used in constants have been cross-checked against Arbitrum's public registries; the `L2_RETRYABLE_TICKET_MAGIC` sentinel is a DAO-specific convention and has no external registry entry.