code4rena

Unruggable

Smart Contract Security Assessment

Version 2.0

Audit dates: Nov 22 — Dec 04, 2024

Audited by: peakbolt
spicymeatball
0xadrii

Contents

1. Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2. Executive Summary

2.1 About Unruggable

Unruggable is working to bring ENS names to every user and every chain by providing developers with open-source code and offering our services dedicated to supporting ENS in becoming more multichain and scalable. We are committed to enhancing ENS by making it possible for anyone to create their own name service for their project using ENS subnames, such as name.uni.eth.

2.2 Scope

Repository	unruggable-labs/unruggable-gateways
Commit Hash	67d92e12ece1a1d849ac671fc62db465ee4cae94
Mitigation Hash	1ded66d31b1d600f991d103fbdcd9699806afbc7

2.3 Audit Timeline

DATE	EVENT
Nov 22, 2024	Audit start
Dec 04, 2024	Audit end
Dec 06, 2024	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	5
Informational	1
Total Issues	8

3. Findings Summary

ID	DESCRIPTION	STATUS
M-1	`walkTree` incorrectly reverts when the terminal node is `NODE_LEAF_EMPTY`	Resolved

M-2	`ScrollVerifierHooks` fails to handle non-existent key in zkTrie	Resolved
L-1	`_checkWindow()` allows malicious gateway to selectively return an valid earlier block for on-chain verification	Acknowledged
L-2	`OPFaultVerifier` could face temporary DoS due to unbounded loop	Acknowledged
L-3	Verification using `OPFaultGameFinder` will fail silently if gameType value exceed 255	Resolved
L-4	Incorrect check in `GatewayFetcher::addBytes()` makes `MAX_OPS` length never be reachable	Resolved
L-5	`ops` length might surpass `MAX_OPS` when using `push` to build a request	Resolved
I-1	Possible out-of-bounds access when handling `PUSH_OUTPUT` opcode in `GatewayVM`	Resolved

4. Findings

4.1 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] `walkTree` incorrectly reverts when the terminal node is `NODE_LEAF_EMPTY`

Severity: Medium	Status: Resolved

Context: ScrollVerifierHooks.sol#L147-L157

Description:

`walkTree` will traverse down the zkTrie based on the key that we are verifying. It will either return (1) the terminal node that matches our key, proving its existence or (2) the terminal node that shares the same suffix as our key, proving its non-existence.

While traversing down the branch nodes, it will set `done = true` when the next child node (either left or right) to traverse to is a terminal node (`NODE_LEAF` or `NODE_LEAF_EMPTY`).

`walkTree() correctly hands the case when the next child node is` NODE_LEAF`, a terminal node, by ending the loop when` (done | i == 0)`.

However, it fails to exit the loop on the next child node is `NODE_LEAF_EMPTY`, which is also a terminal node. This will cause it to revert as the next `else if` case will revert with `InvalidProof()`.

The impact of this issue will cause a verification of a proven non-existent key to return an error 'invalid proof' instead of returning `NOT_A_CONTRACT` for account state verification or `bytes32(0)` for slot verification.

```
    function walkTree(
        bytes32 key,
        bytes[] memory proof,
        bytes32 rootHash
    ) internal view returns (bytes32 expectedHash, bytes memory v) {
        expectedHash = rootHash;
        bool done;
        //console.log("[WALK PROOF] %s", proof.length);
        for (uint256 i; ; i++) {
            if (i == proof.length) revert InvalidProof();
            v = proof[i];
            bool left = uint256(key >> i) & 1 == 0;
```

```
            uint256 nodeType = uint8(v[0]);
            //console.log("[%s] %s %s", i, nodeType, left ? "L" : "R");

>>>         if (nodeType == NODE_LEAF) {
                //@audit this should also handle when nodeType ==
NODE_LEAF_EMPTY
                // 20240917: tate noted 1 slot trie is just a terminal
node
                if (done || i == 0) break;
                revert InvalidProof(); // expected leaf
            } else if (
                nodeType < NODE_LEAF_LEAF ||
                nodeType > NODE_BRANCH_BRANCH ||
                v.length != 65
            ) {
                  //@audit  when nodeType == NODE_LEAF_EMPTY, it will
revert here
>>>             revert InvalidProof(); // expected node
            }
```

Recommendation: This can be resolved by handling `nodeType == NODE_LEAF_EMPTY` in `walkTree()`, similiar to `NODE_LEAF`.

Client:

Fix: @3e95abf58b9b.. and @9116aa422fdf..

- `walkTree()` supports `NODE_LEAF_EMPTY`
  - the length of bytes is ignored (it just has to be larger than 0, which is checked before `nodeType` is extracted)
  - the expected hash must be 0
- Both `verifyAccountState()` and `verifyStorageValue()` check for hash of 0
- The code is also refactored and cleaned now that we have a working solution.

Zenith:

Verified. This issue has been resolved by handling `NODE_LEAF_EMPTY` accordingly.

[M-2] `ScrollVerifierHooks` fails to handle non-existent key in zkTrie

Severity: Medium	Status: Resolved

Context:

- ScrollVerifierHooks.sol#L121
- ScrollVerifierHooks.sol#L76
- ScrollVerifierHooks.sol#L102

Description:

`isValidLeaf()` will check that the leaf node's key matches the `key` (for the account or slot), to verify that `walkTree()` retrieve the correct leaf node from the zkTrie.

Following that, both `verifyAccountState()` and `verifyStorageValue()` will check that the leaf node's `leafHash` on the zkTrie matches the key-value hash `h` that we are verifying. This ensures that the key-value pair that we are verifying is proven to exist on the zkTrie, marking its inclusion in the state.

However, it fails to handle the case where the key-value pair that we are verifying does not exists in the zkTrie. That is because the account is not used in a transaction or the slot is not written to yet.

When that happens, the proof will only contain the terminal node that shares the same suffix as the non-existent key, as documented in 1 and 2. That is because scroll does not expand fully the key on the zkTrie when a terminal node is the only node with that specific suffix (see scroll docs). This means verifying a terminal node that has the common suffix as the key will proves the absence of that specific key.

The impact of this issue is that the Verifier will return the error 'invalid proof' when the account/slot is not yet updated or written, despite using a valid proof. This could happen when the gateway return a proof that is slightly older than latest, just before the account/slot is created/written to.

```
    function isValidLeaf(
        bytes memory leaf,
        uint256 len,
        bytes32 raw,
        bytes32 key,
        bytes4 flag
    ) internal pure returns (bool) {
        if (leaf.length != len) return false;
        bytes32 temp;
        assembly {
```

```
            temp := mload(add(leaf, 33))
        }
>>>     if (temp != key) return false; // KeyMismatch
        assembly {
            temp := mload(add(leaf, 65))
        }
        if (bytes4(temp) != flag) return false; // InvalidCompressedFlag
        if (uint8(leaf[len - 33]) != 32) return false; //
InvalidKeyPreimageLength
        assembly {
            temp := mload(add(leaf, len))
        }
        return temp == raw; // InvalidKeyPreimage
    }

    function verifyAccountState(
        bytes32 stateRoot,
        address account,
        bytes memory encodedProof
    ) external view returns (bytes32 storageRoot) {
        bytes[] memory proof = abi.decode(encodedProof, (bytes[]));
        bytes32 key = poseidonHash1(bytes20(account)); // left aligned
        (bytes32 leafHash, bytes memory leaf) = walkTree(key, proof,
stateRoot);
        // HOW DO I TELL THIS DOESNT EXIST?
        if (!isValidLeaf(leaf, 230, bytes32(bytes20(account)), key,
0x05080000))
            revert InvalidProof();
        // REUSING VARIABLE #1
        bytes32 temp;
        assembly {
            temp := mload(add(leaf, 69))
        } // nonce||codesize||0
        // REUSING VARIABLE #2
        assembly {
            stateRoot := mload(add(leaf, 101))
        } // balance
        assembly {
            storageRoot := mload(add(leaf, 133))
        }
        bytes32 codeHash;
        assembly {
            codeHash := mload(add(leaf, 165))
        }
        bytes32 h = poseidonHash2(storageRoot, poseidonHash1(codeHash),
1280);
```

```
            h = poseidonHash2(poseidonHash2(temp, stateRoot, 1280), h, 1280);
            // REUSING VARIABLE #3
            assembly {
                temp := mload(add(leaf, 197))
            }
            h = poseidonHash2(h, temp, 1280);
            h = poseidonHash2(key, h, 4);
>>>         if (leafHash != h) revert InvalidProof(); //
InvalidAccountLeafNodeHash
            if (codeHash == NULL_CODE_HASH) storageRoot = NOT_A_CONTRACT;
        }

        function verifyStorageValue(
            bytes32 storageRoot,
            address /*target*/,
            uint256 slot,
            bytes memory encodedProof
        ) external view returns (bytes32 value) {
            bytes[] memory proof = abi.decode(encodedProof, (bytes[]));
            bytes32 key = poseidonHash1(bytes32(slot));
            (bytes32 leafHash, bytes memory leaf) = walkTree(
                key,
                proof,
                storageRoot
            );
            uint256 nodeType = uint8(leaf[0]);
            if (nodeType == NODE_LEAF) {
                if (!isValidLeaf(leaf, 102, bytes32(slot), key, 0x01010000))
                    revert InvalidProof();
                assembly {
                    value := mload(add(leaf, 69))
                }
                bytes32 h = poseidonHash2(key, poseidonHash1(value), 4);
>>>             if (leafHash != h) revert InvalidProof(); //
InvalidStorageLeafNodeHash
            } else if (nodeType == NODE_LEAF_EMPTY) {
                if (leaf.length != 1) revert InvalidProof();
                if (leafHash != 0) revert InvalidProof(); //
InvalidStorageEmptyLeafNodeHash
            }
        }
```

Proof of Concept

The following test case will show that `verifyAccountState()` reverts when it is verifying an non-existent account using a valid proof, instead of returning `NOT_A_CONTRACT`.

Modify `test/gateway/scroll.test.ts` as described below and run it with `bun test test/gateway/scroll.test.ts`.

```
import { ScrollRollup } from '../../src/scroll/ScrollRollup.js';
import { testScroll } from './common.js';

testScroll(ScrollRollup.mainnetConfig, {
  log: true,
  //
https://scrollscan.com/address/0x09D2233D3d109683ea95Da4546e7E9Fc17a6dfAF
#code
-  slotDataContract: '0x09D2233D3d109683ea95Da4546e7E9Fc17a6dfAF',
+    slotDataContract: '0xF85aF7D6c0B3691CCAB5A1bF80b7186f2Dd71061',
//non-existent account
  //
https://scrollscan.com/address/0x28507d851729c12F193019c7b05D916D53e9Cf57
#code
  slotDataPointer: '0x28507d851729c12F193019c7b05D916D53e9Cf57',
});
```

Recommendation: This issue can be resolved as follows:

1. Update `isValidLeaf()` to not revert when keys mis-match. When that happens, it should instead return a bool `exists = false` value just like `EthVerifierHooks` to show that the key does not exists on the zkTrie so that it can be handled in `verifyAccountState()` and `verifyStorageValue()`, to return `NOT_A_CONTRACT` and `bytes32(0)` respectively.
2. Following that, it should skip the `leafHash != h` check when the key in the leaf node does not match the key that we are looking for, as it is only provided to prove the non-existence of the key so the hashes are expected to be different.
3. Finally in `verifyStorageValue()` and `verifyAccountState()`, when the key does not exists (due to key mis-match), perform a final non-existence verification on the leaf node by performing the `poseidonHash2()` using the key hash of the leaf node in the proof.

Client: Fix: @3e95abf58b9bf.. and [@9116aa422fdf..]https://github.com/unruggable-labs/unruggable-gateways/commit/9116aa422fdf505cd3bb9475c7429a0e262ba109)

- `walkTree()` supports `NODE_LEAF` w/KeyMismatch
  - when the `key == leafKey`, we set `exists = true`
  - otherwise:
    - the proof must traverse to the same location (prefix check)
    - the actual key is extracted from the leaf so the proof can be verified

- Both `verifyAccountState()` and `verifyStorageValue()` zero the value if `!exists`
- The code is also refactored and cleaned now that we have a working solution.

Zenith: Verified. This issue has been resolved after discussion and investigation together with the client to support the non-existence proof. A good-to-have suffix check was added in `walkTree()` when keys mis-match, to reject spoofing attack using a valid proof for a different branch of the zkTrie. It was noted this is to exit early and that the `poseidonHash2()` on the leaf node would also reject such attacks.

4.2 Low Risk

A total of 5 low risk findings were identified.

[L-1] `_checkWindow()` allows malicious gateway to selectively return an valid earlier block for on-chain verification

Severity: Low	Status: Acknowledged

Context:

- AbstractVerifier.sol#L42-L45

Description:

Verifiers have an internal function `_checkWindow()`, which will check that the `got` block obtained by the gateway is within the past window period defined by `_window`.

```
    function _checkWindow(uint256 latest, uint256 got) internal view {
        if (got + _window < latest) revert CommitTooOld(latest, got,
_window);
        if (got > latest) revert CommitTooNew(latest, got);
    }
```

However, this gives the gateway the ability to 'choose' a earlier valid block within that window instead of the latest possible block.

As the Verifier is designed for general usage, this will cause an issue for time-sensitive use cases (e.g. price), where the dapp depends on an acceptable up-to-date block.

In the worst case, for optimistic rollups, the gateway can even choose a earlier known rejected block (when `minBlocks !=0`) instead of a later unresolved block that is still within that window.

An undesirable scenario could occur as follows,

1. Alice makes two `fetch()`, one for Contract A and another for Contract B separately, but both are called around the same time, thereby having the same latest context (block).
2. Alice expects gateway to return the response based on the same block for both `fetch()`.
3. However, the gateway maliciously returns the fetch response based on a earlier REJECTED block for the Contract A that is within the window. On the other hand, it returns the fetch response for Contract B using the latest valid block.

4. As window is a not a parameter, Alice will not be aware that the data from both fetches are obtained from different blocks. Furthermore, the fetch for Contract A is from a known rejected block.

Recommendation:

Consider allowing the caller to specify the window period as a parameter of `fetch()` so that it can be set to a zero or small window if required to restrict gateway's response.

Alternatively, this can be acknowledged if the deployed verifiers has a sufficient narrow window that lowers the risk of a rogue gateway that selectively returns an earlier block for malicious intent.

Client:

Around the time of a commit, there is a likely mismatch between the client (likely using latest for simulation and view calls) and the gateway (using finalized block tag). Typically, the client likely sees the new commit, and the gateway is still serving the prior commit. Additionally, the client and/or the gateway could be lagged or far apart.

For tighter bounds, I think we'd just deploy another verifier with a shorter window.

- most ZK chains could have 1 verifier, with effective 1-hr window
- optimistic chains could have 2 verifiers, one with 2-hr ("prioritize latest") and one with 6-24 hr ("prioritize resilience")

Since the units of the window parameter are non-trivial, we do not expose this to the user in fetch() although it was considered. We wanted fetch() to be rollup-agnostic, so it can be toggled from using a TrustedVerifier (immediate updates using signed state roots) to a finalized verifier.

For example, on launch day of some cross-chain mint, you might want to use a TrustedVerifier so all transfers and changes are visible immediately and then one-way switch that to OPFaultVerifier after things have calmed down.

Zenith:

Issue is acknowledged by client as the design is meant to handle when there are latency issue between client and gateway.

If required, the window period could be further restricted by deploying another verifier that has a shorter window, without any code changes.

[L-2] `OPFaultVerifier` could face temporary DoS due to unbounded loop

Severity: Low	Status: Acknowledged

Context:

- OPFaultGameFinder.sol#L46-L63

Description:

When the `OPFaultVerifier` is set to finalized mode (`minAgeSec == 0`), the while loop in `OPFaultGameFinder.findGameIndex()` could be unbounded and exceed gas limit in the rare scenario that it need to iterate through huge amount of unresolved nodes to find the latest finalized node.

One hypothetical scenario that could cause this issue to occur is when the sequencer is down for a long period of time such that all the blocks created cannot be resolved to L1. When the sequencer recovers, there could be a a huge back log of unresolved game that are created on L1, requiring `findGameIndex()` to iterate through all of them.

This issue could cause a temporary DoS and inconvenience, where the user can switch to the unfinalized verifier if required.

```
    function findGameIndex(
        IOptimismPortal portal,
        uint256 minAgeSec,
        uint256 gameTypeBitMask,
        uint256 gameCount
    ) external view virtual returns (uint256) {
        if (gameTypeBitMask == 0)
            gameTypeBitMask = 1 << portal.respectedGameType();
        IDisputeGameFactory factory = portal.disputeGameFactory();
        if (gameCount == 0) gameCount = factory.gameCount();

            //@audit this loop could be unbounded
>>>     while (gameCount > 0) {
            (
                uint256 gameType,
                uint256 created,
                IDisputeGame gameProxy
            ) = factory.gameAtIndex(--gameCount);
            if (
                _isGameUsable(
                    gameProxy,
                    gameType,
```

```
                    created,
                    gameTypeBitMask,
                    minAgeSec
                )
            ) {
                return gameCount;
            }
        }
        revert GameNotFound();
    }
```

Recommendation: Consider using `AnchorStateRegistry` to retrieve the anchor state and find the latest finalized game. This is assuming finalized mode has `_gameTypeBitMask == 0 && minAgeSec == 0` so it only requires the latest finalized game for the `respectedGameType()`. That means we do not need to loop through the games for finalized mode and prevent the issue from occurring. For this fix to work, we will have to use the `l2BlockNumber()` instead of `resolveAt()` for the `_checkWindow()` comparison in `getStorageValues()`. This is because anchor state only stores the `outputRoot` and `l2BlockNumber`.

But it is noted that this fix is not easy as `OptimismPortal` does not directly provide a mean to easily traverse the latest finalized games for the gateway/verifier.

Alternatively, this issue can be acknowledged until OP develops a function that allows traversal of latest finalized blocks.

Client:

We had a version like this, but the GameFinder design ended up being more flexible.

The gateway's Rollup design captures 2 ideas related to this:

1. what's the latest commit (head)
2. what's the commit before that commit (next)

This is necessary because around the time of a commit transaction (finalization), the view of the chain may not be consistent due to geography or latency etc between the client and the gateway. The gateway typically serves a "finalized" view of the chain, so its normally behind the client.

Client: I want `X` Gateway: here is proofs for `X-3` Client: check that `X-W ≤ X-3 ≤ X`

This can be done with block numbers but finding older games from block numbers is messy.

Additionally, as a UX annoyance, the `OptimismPortal` doesn't have a public reference to the `AnchorStateRegistry`.

Zenith: Acknowledged by client as current design allows for a more flexible mechanism to allow the gateway/verifier to handle different latest finalized blocks due to latency between the client and the gateway.

[L-3] Verification using `OPFaultGameFinder` will fail silently if gameType value exceed 255

Severity: Low	Status: Resolved

Context:

- OPFaultGameFinder.sol#L99-L114

Description: `OPFaultGameFinder` uses a `gameTypeBitMask` to match multiple game types as the `respectedGameType()` could change over time, e.g. when OP switches between permissionless and permissioned, or when a new game type is introduced.

However, the GameType is uint32, and that means it could exceed 255, causing a silient overflow in `_isGameUsable()`.

```
    function _isGameUsable(
        IDisputeGame gameProxy,
        uint256 gameType,
        uint256 created,
        uint256 gameTypeBitMask,
        uint256 minAgeSec
    ) internal view returns (bool) {
        //@audit when gameType == 256, the operation 1 << 256 will return
0 due to silent overflow.
>>>     if (gameTypeBitMask & (1 << gameType) == 0) return false;
```

The impact of this issue is that Verifier and Gateway could fail to verify against latest finalized game as described in the scenario below.

1. OptimismPortal sets new `respectedGameType() = 256`. Both `OPFaultVerifier` and gateway have `gameTypeBitMask == 0`, which will take `respectedGameType()` and set `gameTypeBitMask = 1`.
2. New game `X` with `gameType == 256` has been resolved and becomes the latest finalized game.
3. However, Gateway and `OPFaultVerifier` will not verify against latest finalized game `X` as `_isGameUsable()` will fail to match game 'X' because of the silient overflow in `_isGameUsable()` above.
4. The issue will not be noticed and gateway/verifier continues to match based on previous game type instead of 256.

Recommendation: This can be fixed by reverting when `gameType > 255` as below.

```
    function _isGameUsable(
        IDisputeGame gameProxy,
        uint256 gameType,
        uint256 created,
        uint256 gameTypeBitMask,
        uint256 minAgeSec
    ) internal view returns (bool) {
+       if (gameType > 255) revert UnsupportedGameType();
        if (gameTypeBitMask & (1 << gameType) == 0) return false;
```

This is assuming OP team will only create new gameTypes that are <= 255. But in the unlikely case that OP fail to do so, it makes sense to revert on `gameType > 255` so that the issue will be noticed and can be fixed instead failing silently.

Client: Fixed in the following commit

Zenith: Verified. Resolved by reverting when result from `gameTypeBitMask = 1 << portal.respectedGameType()` is zero.

[L-4] Incorrect check in `GatewayFetcher::addBytes()` makes `MAX_OPS` length never be reachable

Severity: Low	Status: Resolved

Context:

- `GatewayFetcher::addBytes()`

Description:

In `GatewayFetcher::addBytes()`, a check is performed to ensure that length of `ops` + the length of the to-be-added `v` bytes never surpasses `MAX_OPS`:

```
// File: GatewayFetcher.sol

function addBytes(
        GatewayRequest memory r,
        bytes memory v
    ) internal pure returns (GatewayRequest memory) {
        bytes memory buf = r.ops;
        if (r.ops.length + v.length >= MAX_OPS) revert RequestOverflow();
       ...
```

The problem is that the check should use `>` instead of `>=`, as using `>=` makes the maximum possible length of `ops` be `MAX_OPS - 1`, instead of `MAX_OPS`.

The following proof of concept demonstrates such behavior (note `MAX_OPS` was reduced to 10 for simplicity):

```
 function testIncorrectLimitCheckAddBytes() public {
        // Create a new request.
        GatewayRequest memory firstRequest =
GatewayFetcher.newRequest(0);

        console.log("-- FIRST REQUEST --");
        // Ops already contains a 0x00 due to previously appended
`outputs`. Add
        // 9 more bytes so that `ops` length increases to 10 and ops is
filled to the max allowed value.
        for (uint256 i; i < 9; i++) {
            firstRequest.addByte(uint8(i + 1));
        }
```

```
        console.log(" ");
        console.log("First request length: ", firstRequest.ops.length);

        console.log(" ");
        console.log("First request ops: ");
        console.logBytes(firstRequest.ops);

        console.log(" ");
        console.log(" ");
        console.log(
            "Is first request filled:",
            firstRequest.ops.length == GatewayFetcher.MAX_OPS
        );

        console.log("");
        console.log("-- SECOND REQUEST --");

        // Create a second request.
        GatewayRequest memory secondRequest =
GatewayFetcher.newRequest(0);

        // We add 8 bytes so that length becomes 9. It should
theoretically be possible to add one additional byte, as max allowed ops
is 10.
        for (uint256 i; i < 8; i++) {
            secondRequest.addByte(uint8(i + 1));
        }

        console.log(" ");
        console.log("Second request length: ", secondRequest.ops.length);

        console.log(" ");
        console.log("Second request ops: ");
        console.logBytes(secondRequest.ops);

        bytes memory newData = new bytes(1);
        newData[0] = 0x01;

        // The incorrect check in `addBytes` makes it impossible to add 1
byte and reach `MAX_OPS`
        vm.expectRevert(abi.encodeWithSignature("RequestOverflow()"));
        secondRequest.addBytes(newData);
    }
```

Recommendation: Update the `addBytes` function so that the length check uses `>` instead of `>=`.

Client: Fixed in [commit 34ec13f9](#).

Zenith: Verified. The length check in `addBytes` is now performed using `>` instead of `>=`.

[L-5] `ops` length might surpass `MAX_OPS` when using `push` to build a request

Severity: Low	Status: Resolved

Context:

- `GatewayFetcher.sol::push()`

Description:

The `push` function in `GatewayFetcher` is missing checks to ensure that length of `ops` does not surpass the maximum allowed `MAX_OPS`. Because of this, it is possible to create an `ops` array longer than the max allowed length.

The following proof of concept demonstrates how max ops can be bypassed. Note `MAX_OPS` was set to 10 in `GatewayFetcher` for easier visualization and simplicity.

```
 function testBypassMaxOps() public {

        // Create a new request.
        GatewayRequest memory request = GatewayFetcher.newRequest(0);

        // Ops already contains a 0x00 due to previously appended
`outputs`. Add
        // 8 more bytes so that `ops` length increases to 9.
        for(uint256 i; i < 8; i++) {
            request.addByte(uint8(i+1));
        }

        console.log(" ");
        console.log("Request length: ", request.ops.length);

        console.log(" ");
        console.log("Request ops: ");
        console.logBytes(request.ops);

        console.log(" ");
        console.log(" ");

        console.log("Trigger push and surpass length:");

        // `push` will append two items: `PUSHx` opcode (PUSH1 in this
case) and the actual value pushed (50),
```

```
        // which will be a single byte given that 50 has 31 leading
zeroes.
        request.push(50);


        console.log(" ");
        console.log("Broken request length: ", request.ops.length);


        console.log(" ");
        console.log("Broken request ops: ");
        console.logBytes(request.ops);

        console.log(" ");
        console.log("Is current ops length greater than MAX_OPS:",
request.ops.length > GatewayFetcher.MAX_OPS);


        // `addByte` won't allow to add more bytes, as it includes max
checks
        vm.expectRevert(abi.encodeWithSignature("RequestOverflow()"));
        request.addByte(0x00);

    }
```

Recommendation: Consider adding checks inside `push` function to ensure that `MAX_OPS` limit is not bypassed.

Client: Fixed in @9d2cdbed..

Zenith: The fix is correct. A check has been added to ensure that `push` does not allow `ops` to surpass `MAX_OPS` length.

4.3 Informational

A total of 1 informational findings were identified.

[I-1] Possible out-of-bounds access when handling `PUSH_OUTPUT` opcode in `GatewayVM`

Severity: Informational	Status: Resolved

Context:

- GatewayVM.sol#L372

Description: When handling the `PUSH_OUTPUT` opcode in the VM, it is not verified that the popped element from the stack is `< outputs.length`. Because of this, users could try to access slots in the outputs memory array which have not been allocated, leading to panics.

This would fall into the category of user error, as the length of `outputs` is specified when creating a new request.

Recommendation:

Consider adding a check to ensure that the popped `uint256` is `< outputs.length`. This allows to revert with a custom error instead of panicking if the popped element is greater or equal to `outputs.length`.

Client:

Whilst a user error I have added these checks for clarity in code given the minimal gas overhead. I've additionally updated the InvalidStackIndex error to return the associated index for code consistency. Added checks in commit 37f28280.

Zenith:

Verified - A check has been added and will revert with `InvalidOutputIndex` if the output accesses an out-of-bounds element. In addition, a parameter has been added to the `InvalidStackIndex` error thrown in `checkBack` if the requested `back` is greater or equal than the current stack size.