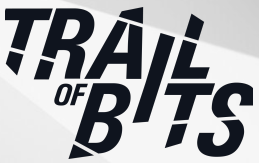

ArbOS 20 Upgrade

Security Assessment (Summary Report)

February 13, 2024

Prepared for:
Harry Kalodner, Rachel Bousfield, Lee Bousfield, Steven Goldfeder, and Ed Felten
Offchain Labs

Prepared by: **Troy Sargent and Simone Monica**

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineering director was associated with this project:

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

The following consultants were associated with this project:

Troy Sargent, Consultant
troy.sargent@trailofbits.com

Simone Monica, Consultant
simone.monica@trailofbits.com

Project Timeline

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

Date	Event
January 25, 2024	Pre-project kickoff call
February 5, 2024	Status update meeting
February 10, 2024	Delivery of report draft
February 12, 2024	Report readout meeting
February 13, 2024	Delivery of summary report

Project Targets

The engagement involved a review and testing of the targets listed below.

Nitro

Repository	https://github.com/OffchainLabs/nitro
Versions	`ab13ede45ce77cc7a08ef958ee7bfe0f2602590c` (Reviewed in week 1)
	`cf2eadfcc1039eca9594c4f71477a50f550d7749` (Reviewed in week 2)
Types	Go, Rust
Platform	Native

Nitro Contracts

Repository	https://github.com/OffchainLabs/nitro-contracts
Version	`5ebb128a8fe6053d618ab2081d7c7ccfc68a35c4`
Type	Solidity
Platform	EVM

Governance

Repository	https://github.com/ArbitrumFoundation/governance
Version	`50027863bb14f20b39686329a2d1d7c40df9f10b`
Type	Solidity
Platform	EVM

Executive Summary

Engagement Overview

Offchain Labs engaged Trail of Bits to review the security of the ArbOS 20 upgrade of the Nitro node and smart contracts, as well as the code in the governance repository that will be executed if the AIP-4844 proposal passes.

A team of two consultants conducted the review from January 29 to February 9, 2024, for a total of four engineer-weeks of effort. With full access to the source code and documentation, we performed static and dynamic testing of the project targets using automated and manual processes.

Observations and Impact

ArbOS 20 adds support for EIP-4844 blob transactions, an alternative way for rollups to store their transaction data on the base chain, and includes several other minor additions. We prioritized assessing whether the update to Offchain's implementation of `go-ethereum` and additional modifications introduced consensus issues. We also reviewed the off-chain and on-chain functionality related to blob transactions for denial-of-service vectors and missing data validation, looked for discrepancies between the Rust and Solidity implementations of the fraud prover, and checked whether the governance proposal will execute as anticipated.

Our review of the Nitro node focused specifically on new changes to the node since ArbOS 11. Our review of the Nitro smart contracts focused on changes introduced through PR #108, PR #74 , PR #109, PR #104, PR #98, and PR #90. We did not review the governance codebase in its entirety, focusing only on `AIP4844Action.sol`, `SetArbOS20VersionAction.sol`, and `SetBatchPosterManager.sol`.

We identified one high-severity consensus issue (TOB-ARBOS20-2) and gaps in dynamic testing that may have prevented this issue and could prevent others. Therefore, we recommend adding specific tests in order to decrease the likelihood that similar bugs will be missed during the development and review of future upgrades. Note that finding TOB-ARBOS20-2 was discovered during week 1 of our review; Offchain fixed the issue during the audit, and we incorporated that fix into the scope of week 2.

Recommendations

- Remediate the findings uncovered during this review.
- Implement the dynamic tests suggested in TOB-ARBOS20-1, TOB-ARBOS20-4, and appendix C.
- Create a specification for the prover that is the "source of truth" for the Rust and Solidity implementations.

Summary of Findings

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

ID	Title	Type	Severity
1	Discrepancies between checks performed by arbitrator and OneStepProverHostIo and OneStepProverO contracts	Data Validation	**Informational**
2	Missing ArbOS 20 gate on newly added GetScheduledUpgrade function	Data Validation	**High**
3	Unclear implementations of MaxInitCodeSize and MaxCodeSize restrictions	Data Validation	**Informational**
4	Rejection of unsupported transaction types is untested and fragile	Data Validation	**Informational**

Detailed Findings

1. Discrepancies between checks performed by arbitrator and OneStepProverHostIo and OneStepProver0 contracts

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARBOS20-1
Target: `arbitrator/prover/src/machine.rs`, `src/osp/OneStepProverHostIo.solm`, `src/osp/OneStepProver0.sol`	

Description

The arbitrator and the `OneStepProverHostIo` and `OneStepProver0` contracts have discrepancies between some of the checks they perform.

The arbitrator and the `OneStepProverHostIo` contract check that the preimage type is valid when executing the `ReadPreImage` opcode. However, they perform these checks in different orders, which may lead to inconsistencies.

The `OneStepProverHostIo` contract's `executeReadPreImage` function first performs other checks that could set the machine state to `Errored` (figure 1.1), which would later cause the function to revert if the preimage type is not valid.

```
function executeReadPreImage(
     ExecutionContext calldata,
     Machine memory mach,
     Module memory mod,
     Instruction calldata inst,
     bytes calldata proof
 ) internal view {
    uint256 preimageOffset = mach.valueStack.pop().assumeI32();
     uint256 ptr = mach.valueStack.pop().assumeI32();
     if (preimageOffset % 32 != 0 || ptr + 32 > mod.moduleMemory.size || ptr %
LEAF_SIZE != 0) {
         mach.status = MachineStatus.ERRORED;
         return;
     }
     ...
     if (inst.argumentData == 0) {
         ...
     } else if (inst.argumentData == 1) {
         ...
     } else if (inst.argumentData == 2) {
```

```
        ...
    } else {
        revert("UNKNOWN_PREIMAGE_TYPE");
    }
    ...
}
```

Figure 1.1: A snippet the `executeReadPreImage` function in `OneStepProverHostIo.sol#L123-L245`

The arbitrator instead first checks whether the preimage type is valid and then performs the other checks that could set the machine state to `Errored` (figure 1.2).

```
pub fn step_n(&mut self, n: u64) -> Result<()> {
    ...
    Opcode::ReadPreImage => {
        let offset = self.value_stack.pop().unwrap().assume_u32();
        let ptr = self.value_stack.pop().unwrap().assume_u32();
        let preimage_ty =
PreimageType::try_from(u8::try_from(inst.argument_data)?)?;
        // Preimage reads must be word aligned
        if offset % 32 != 0 {
            error!();
        }
        if let Some(hash) = module.memory.load_32_byte_aligned(ptr.into()) {
        ...
    ...
}
```

Figure 1.2: A snippet of the `step_n` function in `machine.rs#L1866-L1874`

This discrepancy between the order of checks in these two locations could cause inconsistencies in the machine status, such as when a preimage type is not valid and the offset is not modulo 32. However, the severity of this finding is rated as informational because, as suggested by the Offchain team, other important assumptions need to be broken; for example, the WASM module root would need to be malicious.

There are other similar discrepancies between the checks performed by the arbitrator and the `OneStepProver0` contract. For example, when handling the `argumentData` value for the `Call` operation, the contract checks the result of the downcast to ensure there is no precision loss (figure 1.3), but the arbitrator does not (figure 1.4). The arbitrator does not perform this check in the `ArbitraryJump` and `ArbitraryJumpIf` operations, either.

```
function executeCall(
    Machine memory mach,
    Module memory,
    Instruction calldata inst,
    bytes calldata
) internal pure {
```

```
    ...
    // Jump to the target
    uint32 idx = uint32(inst.argumentData);
    require(idx == inst.argumentData, "BAD_CALL_DATA");
    ...
}
```

Figure 1.3: A snippet of the `executeCall` function in `OneStepProver0.sol#L117-L136`

```
pub fn step_n(&mut self, n: u64) -> Result<()> {
    ...
                Opcode::Call => {
                    let current_frame = self.frame_stack.last().unwrap();
                    self.value_stack.push(Value::InternalRef(self.pc));
                    self.value_stack
                        .push(Value::I32(current_frame.caller_module));
                    self.value_stack
                        .push(Value::I32(current_frame.caller_module_internals));
                    self.pc.func = inst.argument_data as u32;
                    self.pc.inst = 0;
                    func = &module.funcs[self.pc.func()];
                }
    ...
}
```

Figure 1.4: A snippet of the `Call` function in `machine.rs#L1466-L1476`

Recommendations

Short term, in the arbitrator's `step_n` function, move the `preimage_ty` variable inside the `if/let` block so that it performs the same order of checks as the `OneStepProverHostIo` contract's `executeReadPreImage` function. Add the missing downcast result check to the arbitrator's `Call`, `ArbitraryJump`, and `ArbitraryJumpIf` opcodes.

Long term, when implementing the same logic multiple times, such as in these cases, make sure to implement it in the same way, even if divergences would be safe, to simplify the review process.

2. Missing ArbOS 20 gate on newly added GetScheduledUpgrade function	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBOS20-2
Target: `precompiles/precompile.go`	

Description

A new function, `GetScheduledUpgrade`, was added to the `ArbOwnerPublic` precompiled contract in ArbOS 20; however, it does not include an `arbosVersion` value to indicate that it is callable starting only from that ArbOS version. This could lead to state divergences when transactions are replayed during syncs.

When a new function is added to a custom precompiled contract or when an altogether new precompiled contract is added in a new ArbOS version, that new function or contract must check that it cannot be called before it is activated.

As shown in figure 2.1, the `Precompiles` function returns the precompiled contracts present in the system; each function in a precompiled contract has an associated `arbosVersion`, indicating the minimum, active ArbOS version necessary for the call to succeed.

```
func Precompiles() map[addr]ArbosPrecompile {
    ...
        ArbOwnerPublic := insert(MakePrecompile(templates.ArbOwnerPublicMetaData,
&ArbOwnerPublic{Address: hex("6b")}))
        ArbOwnerPublic.methodsByName["GetInfraFeeAccount"].arbosVersion = 5
        ArbOwnerPublic.methodsByName["RectifyChainOwner"].arbosVersion = 11
        ArbOwnerPublic.methodsByName["GetBrotliCompressionLevel"].arbosVersion = 20
    ...
}
```

Figure 2.1: A snippet of the `Precompiles` function in `precompile.go#L559-L562`

The `arbosVersion` is not set for the `GetScheduledUpgrade` function, allowing transactions made on ArbOS versions prior to 20 to call it.

Exploit Scenario

Eve calls the `GetScheduledUpgrade` function in the `ArbOwnerPublic` precompiled contract before ArbOS 20 is activated, and the transaction correctly reverts. When ArbOS 20 is activated, the same transaction succeeds, leading to a state divergence.

Recommendations

Short term, make the `GetScheduledUpgrade` function present in the system starting only from ArbOS 20.

Long term, add tests that attempt to call new functions added to precompiled contracts prior to the upgrade and ensure that a node can replay the historical transaction following an ArbOS upgrade without causing a state divergence.

Note that this issue was discovered during week 1 of our review; Offchain fixed the issue during the audit, and we incorporated that fix into the scope of week 2.

3. Unclear implementations of MaxInitCodeSize and MaxCodeSize restrictions

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARBOS20-3
Target: `core/state_transition.go`, `core/vm/evm.go`	

Description

The validations shown in figure 3.1 and figure 3.2 check that the length of `msg.Data` and the created contract are less than `MaxInitCodeSize` and `MaxCodeSize`, respectively. However, `MaxInitCodeSize` and `MaxCodeSize` are cast to signed integers; if they are set to values greater than 2^{63}, their signed representation will be negative; all inputs will succeed, given that their lengths will always be greater than a negative value.

In practice, these checks would function the same if the lengths of `msg.Data` and the created contract were cast to an unsigned integer instead. This is because all possible length values will succeed up to the system and language runtime's memory limits if the limits are configured to 2^{63}. Nevertheless, it would be more clear to cast the slice lengths to an unsigned integer than to rely on this subtly.

```
func (st *StateTransition) TransitionDb() (*ExecutionResult, error) {
	...
	// Check whether the init code size has been exceeded.
	if rules.IsShanghai && contractCreation && len(msg.Data) >
int(st.evm.ChainConfig().MaxInitCodeSize()) {
		return nil, fmt.Errorf("%w: code size %v limit %v",
ErrMaxInitCodeSizeExceeded, len(msg.Data),
int(st.evm.ChainConfig().MaxInitCodeSize()))
	}
	...
}
```

Figure 3.1: A snippet of the `TransitionDb` function in `state_transition.go#L450-L453`

```
func (evm *EVM) create(caller ContractRef, codeAndHash *codeAndHash, gas uint64,
value *big.Int, address common.Address, typ OpCode) ([]byte, common.Address, uint64,
error) {
	...

	// Check whether the max code size has been exceeded, assign err if the case.
	if err == nil && evm.chainRules.IsEIP158 && len(ret) >
int(evm.chainConfig.MaxCodeSize()) {
		err = ErrMaxCodeSizeExceeded
	}
```

```
        ...
}
```

Figure 3.2: A snippet of the `create` function in `evm.go#L507-L510`

Recommendations

Short term, have the two affected functions cast the lengths of `msg.Data` and the created contract to an unsigned integer instead of `MaxInitCodeSize` and `MaxCodeSize` to a signed integer.

Long term, when casting an unsigned integer to signed, always keep in mind that, depending on the values it can have, the cast value could have a negative sign. Additionally, when adding or changing conditions from the original `go-ethereum` codebase, make sure to add tests with values on the boundaries.

4. Rejection of unsupported transaction types is untested and fragile

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARBOS20-4
Target: `execution/gethexec/sequencer.go`, `execution/gethexec/tx_pre_checker.go`, `arbos/parse_l2.go`	

Description

Since Arbitrum Nitro will not support the storage of blob data in ArbOS 20, blob transaction types are rejected, and a validation is used throughout the codebase that throws an error if such transaction types are used, as shown in figures 4.1, 4.2, and 4.3. However, this validation code is currently untested, and the implementation would be more robust and future-proof if it processed only transactions that are explicitly accepted instead. That way, if an unsupported transaction type is added in the upstream `go-ethereum` implementation, the Offchain team can consider what other changes may be necessary before allowing Arbitrum Nitro to accept it.

```
if err := newTx.UnmarshalBinary(readBytes); err != nil {
	return nil, err
}
if newTx.Type() >= types.ArbitrumDepositTxType || newTx.Type() == types.BlobTxType {
	// Should be unreachable for Arbitrum types due to UnmarshalBinary not
accepting Arbitrum internal txs
	// and we want to disallow BlobTxType since Arbitrum doesn't support EIP-4844
txs yet.
	return nil, types.ErrTxTypeNotSupported
}
```

Figure 4.1: Transaction types that are blacklisted (arbos/parse_l2.go#165–172)

```
if tx.Type() >= types.ArbitrumDepositTxType || tx.Type() == types.BlobTxType {
	// Should be unreachable for Arbitrum types due to UnmarshalBinary not
accepting Arbitrum internal txs
	// and we want to disallow BlobTxType since Arbitrum doesn't support EIP-4844
txs yet.
	return types.ErrTxTypeNotSupported
}
```

Figure 4.2: Another place where the validation is used (nitro/execution/gethexec/sequencer.go#396–400)

```
if tx.Type() >= types.ArbitrumDepositTxType || tx.Type() == types.BlobTxType {
	// Should be unreachable for Arbitrum types due to UnmarshalBinary not
accepting Arbitrum internal txs
	// and we want to disallow BlobTxType since Arbitrum doesn't support EIP-4844
txs yet.
	return types.ErrTxTypeNotSupported
}
```

Figure 4.3: Another place where the validation is used (`nitro/execution/gethexec/tx_pre_checker.go#119–123`)

Recommendations

Short term, add tests to ensure this guard works as expected and to mitigate regressions.

Long term, consider having the implementation explicitly accept certain transaction types so that it does not incidentally process unsupported transaction types following merges and upgrades from upstream.

A. Vulnerability Categories

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

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

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

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

B. Code and Documentation Quality Issues

The following issues are not associated with specific vulnerabilities. However, addressing them would enhance code readability and may prevent the introduction of future vulnerabilities.

- **The `SkipAccountChecks` function's comment is incorrect.**
  - `arb_types.go#L17-L21`: The comment reads, `Returns true if nonce checks should be skipped based on inner's isFake()`, but it should read, `Returns true if nonce checks should be skipped based on inner's skipAccountChecks()`.
- **The following imported errors are never used in the `SequencerInbox` contract:**
  - `NotRollup`
  - `DataNotAuthenticated`
  - `InvalidBlobMetadata`
  - `EmptyBatchData`
- **`RollupNotChangedError` is imported twice in the `SequencerInbox` contract.**
  - It is imported first on line 26 and then on line 32.
- **There are opportunities to refactor and simplify the code:**
  - Move the line in the `addSequencerL2BatchFromBlobs` function that checks whether the host chain is Arbitrum to the start of the function.
  - Simplify the two uses of the code shown in figure B.1 to the simpler version shown in figure B.2.

```
for i := 0; i*32 < len(blobHashes); i += 1 {
	copy(versionedHashes[i][:], blobHashes[i*32:(i+1)*32])
}
```

Figure B.1: A snippet of the `parseSequencerMessage` function in `inbox.go#L100-L102` and of the `IsBlobHashesHeaderByte` function in `stateless_block_validator.go#L298-L300`

```
for i := 0; i < len(blobHashes); i += 32 {
	copy(versionedHashes[i][:], blobHashes[i:(i+32)])
}
```

Figure B.2: Proposed refactoring

- **The website documentation has not yet been updated to include the `ReadPreImage` WAVM opcode.**

C. Transaction Encoding and Decoding Fuzz Test

During our review, an issue was reported to the Nitro repository that pertained to changes relevant to this review. To prevent regressions and ensure that the fix for that issue was adequate, we developed a fuzz test that should be checked into the repository and run regularly. The fuzz test fails on a version of the codebase without the fix (figure C.2).

Run the fuzzer in Offchain's `go-ethereum` repository with the command `go test -fuzz ^FuzzEncodeDecode$ github.com/ethereum/go-ethereum/core/types -v`.

```
package types
import (
	"testing"
	"github.com/ethereum/go-ethereum/rlp"
)
func FuzzEncodeDecode(f *testing.F) {
	f.Fuzz(func(t *testing.T, data []byte) {
		newTx := new(Transaction)
		// Step 1: Decode
		err := rlp.DecodeBytes(data, newTx)
		if err != nil {
			t.Skip() // Skip on invalid input
		}
		// Step 2: Encode
		bin, err := rlp.EncodeToBytes(newTx)
		if err != nil {
			t.Fail()
		}
		// Step 3: Decode again
		newTx2 := new(Transaction)
		err = rlp.DecodeBytes(bin, newTx2)
		if err != nil {
			t.Fail()
		}
		// Step 4: Compare
		if newTx.Hash() != newTx2.Hash() {
			t.Fail()
		}
	})
}
```

Figure C.1: A fuzz test for transaction encoding and decoding (`go-ethereum/core/types/arb_types_fuzz_test.go`)

```
$ go test -run=FuzzEncodeDecode/3d4fbb7831d13f02
github.com/ethereum/go-ethereum/core/types -v

=== RUN   FuzzEncodeDecode
=== RUN   FuzzEncodeDecode/3d4fbb7831d13f02
--- FAIL: FuzzEncodeDecode (0.00s)
    --- FAIL: FuzzEncodeDecode/3d4fbb7831d13f02 (0.00s)
panic: decode called on LegacyTx) [recovered]
        panic: decode called on LegacyTx)
[...]
github.com/ethereum/go-ethereum/core/types.(*LegacyTx).decode(...)
        /audit-nitro/go-ethereum/core/types/tx_legacy.go:124
github.com/ethereum/go-ethereum/core/types.(*Transaction).decodeTyped(0x14000418ee0?
, {0x14000045580, 0x2, 0x40}, 0x1?)
        /audit-nitro/go-ethereum/core/types/transaction.go:243 +0x290
github.com/ethereum/go-ethereum/core/types.(*Transaction).DecodeRLP(0x10372b020?,
0x1400050e780?)
        /audit-nitro/go-ethereum/core/types/transaction.go:175 +0xd8
github.com/ethereum/go-ethereum/rlp.decodeDecoder(0x10372b020?, {0x1036e74c0?,
0x1400050e780?, 0x102f543ac?})
        /audit-nitro/go-ethereum/rlp/decode.go:542 +0x64
github.com/ethereum/go-ethereum/rlp.(*Stream).Decode(0x14000418ee0?, {0x10372b020?,
0x1400050e780?})
        /audit-nitro/go-ethereum/rlp/decode.go:950 +0x22c
github.com/ethereum/go-ethereum/rlp.DecodeBytes({0x140006cd590, 0x3, 0x8},
{0x10372b020, 0x1400050e780})
        /audit-nitro/go-ethereum/rlp/decode.go:99 +0x100
github.com/ethereum/go-ethereum/core/types.FuzzEncodeDecode.func1(0x14000583d40,
[...]
FAIL    github.com/ethereum/go-ethereum/core/types      0.036s
FAIL
```

Figure C.2: A replay of the crashing input found by the fuzz test (without the fix)