Offchain Nitro

Security Assessment

March 14, 2022

Prepared for:
Steven Goldfeder
Offchain Labs

Prepared by:
Josselin Feist and Gustavo Grieco

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

Table of Contents

Executive Summary

Engagement Overview

Offchain Labs engaged Trail of Bits to review the security of its Arbitrum Nitro system. From January 10 to March 11, 2022, a team of two consultants conducted a security review of the client-provided source code, with 16 person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	12
Medium	8
Low	10
Informational	13
Undetermined	4

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	2
Auditing and Logging	5
Denial of Service	1
Configuration	1
Data Validation	33
Patching	1
Undefined Behavior	4

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-NITRO-WAVM-1**
  The block-depth limit is not checked on-chain.

- **TOB-NITRO-WAVM-2**
  The arbitrator and one-step-proof contract perform two comparisons when reading inbox messages; however, because they do not perform the checks in the same order, they will emit different statuses if there is an error.

- **TOB-NITRO-WAVM-3**
  The one-step-proof contracts lack on-chain validation of the destinations of branching instructions.

- **TOB-NITRO-WAVM-4**
  The arbitrator and the one-step-proof contract adhere to different casting rules.

- **TOB-NITRO-WAVM-5**
  When executing the global state access opcodes, the arbitrator and the one-step-proof contract exhibit different behavior if they access an index out of bounds.

- **TOB-NITRO-ARBOS-5**
  An attacker could leverage the lack of error checks in the ticket-opening process to cause a node crash.

- **TOB-NITRO-ARBOS-9**
  Invalid transactions reduce the amount of gas available in a block.

- **TOB-NITRO-ARBOS-15**
  Because the errors returned by `PosterDataCost` are not checked, they can lead to a node crash.

- **TOB-NITRO-ARBOS-18**
  The `gasLeft` amount is calculated incorrectly.

- **TOB-NITRO-SC-1**
  The lack of a contract existence check on a `delegatecall` in the `AdminFallbackProxy` contract can result in unexpected behavior.

- **TOB-NITRO-SC-8**
  The `timeLeft` value, which indicates the amount of time remaining in a challenge, is never decremented.

- **TOB-NITRO-SC-9**
  There is no lower limit on the number of steps in a challenge; as a result, one participant could block the other from taking any actions, preventing the challenge from progressing.

- **TOB-NITRO-SC-10**
  The state of a challenge can change from `EXECUTION` to `BLOCK`.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

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

The following engineers were associated with this project:

Josselin Feist, Consultant
josselin@trailofbits.com

Gustavo Grieco, Consultant
gustavo.grieco@trailofbits.com

Project Timeline

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

Date	Event
January 6, 2022	Pre-project kickoff call
January 18, 2022	Status update meeting #1
January 24, 2022	Status update meeting #2
January 31, 2022	Status update meeting #3
February 7, 2022	Status update meeting #4
February 21, 2022	Status update meeting #5
February 28, 2022	Status update meeting #6
March 7, 2022	Status update meeting #7
March 14, 2022	Report readout meeting
May 25, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Offchain Labs Arbitrum Nitro components, including the Ethereum smart contracts, the arbitrator, the WAVM, and ArbOS.

We sought to answer the following questions about the Ethereum smart contracts:

- Are there gaps between the abstract protocol and its implementation?
- Are there any inconsistencies between the documentation and the implementation?
- Is it possible to steal funds from the protocol?
- Could an attacker disrupt the rollup, challenge, or confirmation process?
- Could the `outbox` execute a withdrawal without a valid confirmation?
- Is it possible to create a node that cannot be challenged?
- Is it possible to evade or delay a challenge?
- Is the `Bridge` contract flexible enough to support any smart contract?

We sought to answer the following questions regarding the Arbitrator:

- Are the semantics of valid WASM binaries preserved in the transpilation to WAVM?
- Do the WASM and WAVM implementations adhere to the WASM specification and reference implementation?
- Is the execution of WASM and WAVM code always completed in a reasonable amount of time?
- Does the WAVM behavior match the on-chain implementation?

Finally, we sought to answer the following questions regarding ArbOS:

- Does the ArbOS EVM implementation adhere to the behavior described in the Yellow Paper? If it deviates from that behavior, how do the deviations affect the correctness and security of the smart contracts deployed on Arbitrum?
- Are incoming messages properly parsed, validated, and processed?

- Could ArbOS be forced to execute a transition to an unexpected AVM status (e.g., `Errored` or `TooFar`)?
- Is the ArbOS bookkeeping correct and updated when necessary? Is there any effect from its internal state that is not properly committed or reverted?

Project Targets

The engagement involved a review and testing of the Offchain Labs Arbitrum Nitro system. We worked from the same GitHub repository (`https://github.com/OffchainLabs/nitro`) throughout the audit. However, the code was significantly updated during the audit, and we reviewed several commits for each component. The most notable commits are provided below.

`nitro/arbitrator`

Version	`c41f610b`
Type	Virtual machine
Platform	Rust

`nitro/arbos`

Versions	`5366994a` and `d8ab8da8`
Type	L2 operating system
Platform	Go

`nitro/solgen`

Version	`77a0422b` (and a fix introduced in PR #294)
Type	Smart contracts
Platform	Ethereum

Project Coverage

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

Ethereum Smart Contracts

The Arbitrum Nitro system includes Ethereum smart contracts that manage and secure a rollup chain on L1. The most relevant contracts are listed below.

Inbox. The `Inbox` allows users to send messages to ArbOS. We reviewed the inbox's receipt of L2 messages, focusing on the impact that user-controlled input can have on the whole system. We manually reviewed the use of events in the construction, validation, and delivery of messages.

`SequencerInbox` and `Bridge`. The `Bridge` contract executes cross-chain transactions sent from L2, and the `SequencerInbox` controls the inclusion of messages in the ArbOS inbox. We focused on the changes made since the last audit and reviewed the contracts' interactions with ArbOS.

`OneStepProver` and related one-step-proof (OSP) contracts. The `OneStepProver` contract emulates WAVM instructions and verifies the correctness of proofs. Our review of this and other OSP contracts was limited to their interactions with the `ChallengeManager`; we focused on identifying any flaws in the contracts' handling of basic errors as well as any deviations from the arbitrator implementation.

`Rollup`. The `Rollup` contract enables validators to stake their funds and to challenge a state. We checked how validators interact with this contract when participating in the staking and challenging protocols, particularly how users add, move, or remove stakes; create, confirm, or reject nodes; remove inactive stakers (zombies); and start and finish challenges.

`ChallengeManager`. The `ChallengeManager` enables validators to resolve challenges in a finite number of steps by using bisection and the OSP contracts. We reviewed the processes of creating, validating, advancing, and finishing challenges, focusing on the bisection of blocks and steps. We also reviewed the ways in which the OSP is invoked and their results are verified. Finally, we checked whether the timeout mechanism works as expected, effectively limiting the duration of challenges.

Arbitrator and WAVM

The arbitrator is a command-line utility that receives WASM code as input and is used by validators to execute WAVM code. Although the arbitrator assumes this WASM code to be trusted, the arbitrator still uses the WebAssembly Binary Toolkit to ensure that the WASM code is well formed before transpiling it. Validators can then use the arbitrator to execute the code and to generate and verify proofs from the code's execution.

We performed a manual review and automated testing to identify any issues in the following areas:

- The parsing, validation, and processing of WASM code
- The WASM–WAVM conversion and the processing and execution of WAVM code
- The verification, serialization, and hashing of the machine status

We looked for discrepancies in the Rust and Solidity implementations of the WAVM. We performed a manual review to look for flaws that could cause an opcode to behave differently, with a focus on the `machine.step` function and the solgen one-step-proof contracts. This part of the review was only done manually. We sought to identify any deviations in the pre- and postconditions applied to the state, as well as any missing conditions.

We also reviewed and tested the execution of WASM and WAVM code, looking for deviations between the WASM emulation and the WASM reference implementation that could affect the arbitrator's correctness and the validity of its proofs.

ArbOS

ArbOS is the trusted L2 operating system. It isolates untrusted contracts from each other, tracks and limits their resource usage, and manages the mechanism that collects fees from users to fund the operation of a chain's validators.

ArbOS handles trusted and untrusted messages originating from Ethereum. We reviewed the handling of incoming messages and the flow of assets. Our review of the escrow mechanism, which allows certain assets to be saved in order to be collected or burned later, focused on how ether is handled and how gas is tracked and burned.

We also reviewed the translation and emulation of the EVM. The ArbOS code includes the `go-ethereum` codebase, with a small number of modifications. We analyzed these modifications to ensure that the `go-ethereum` codebase adheres to the behavior described in the Yellow Paper and the EVM reference implementation. We also looked for ways to disrupt or break the processing of blocks or the gas accounting.

Additionally, we reviewed ArbOS's use of external libraries to handle untrusted data that is parsed, decompressed, or processed. These include the Brotli decompression and Recursive Length Prefix (RLP) encoding / decoding libraries. We looked for unexpected error conditions that could break important ArbOS security or correctness properties. We also checked whether ArbOS could be forced to loop or to consume an excessive amount of resources when processing new incoming messages from L1.

Finally, we reviewed the special ArbOS smart contract operations that allow privileged and unprivileged users to perform important tasks in the Arbitrum system. In particular, we analyzed how retryable tickets are redeemed, canceled, and trimmed when they expire.

We were unable to perform a comprehensive review of every ArbOS component; for example, we performed only a partial review of the ArbNode component.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform testing or comprehensive testing of certain system elements, which may warrant further review.

For example, we did not review the external `go-ethereum` and ArbOS external libraries. Thus, we did not check the correctness of the Brotli decompression implementation. Similarly, we did not review the security or correctness of the WASM external libraries such as the SoftFloat3 floating-point library.

We did not review the following system elements:

- The glue code that connect WASM and Go / Rust code
- ArbOS's Merkle tree accumulator
- ArbOS's internal storage structure
- The system's economic incentives
- The tuning of system parameters (e.g., the challenge timeout parameter)

The following elements were covered in the audit but would benefit from further review:

- The gas accounting performed by ArbOS, including in the ticket-handling process
- The `ChallengeManager`'s state machine, including the verification of segments and the handling of state transitions in the event of VM errors
- The correctness of the OSP contracts and any deviations from the WAVM implementation
- The handling of error statuses (e.g., `TooFar`)

The following directories from the Nitro codebase were not covered or were covered only partially:

- `arbnode`
- `arbstate` (only part of the inbox was covered)
- `arbutil`
- `blsSignatures`
- `broadcastclient`
- `cmd`
- `fastcache`
- `reproducible-wasm`
- `statetransfer`
- `system_tests`
- `util`
- `validator`
- `wavmio`
- `wsbroadcastserver`

Automated Testing Results

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

We used the following tools in the automated testing phase of this project:

- Slither is a static analysis framework that can statically verify algebraic relationships between Solidity variables. We used Slither to look for common Solidity flaws.

Echidna is a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation. We used Echidna to test for fundamental `Rollup` properties, devoting five weeks of CPU time to the testing of each property.

`Rollup`. Using the original `Rollup` code, Trail of Bits developed a model to replicate the behavior of the `Rollup` contracts in Solidity. This was necessary to perform property-based testing of the Rollup contract code, as detailed in appendix E.

Property	Tool	Result
If the preconditions are met, `createChallenge` never reverts.	Echidna	**Passed**
If a challenge can be created, then `commonEndTime >= proposedTimes[0]`, and `commonEndTime >= proposedTimes[1]`.	Echidna	**Passed**
If the preconditions are met, `removeOldZombies` never reverts.	Echidna	**Passed**
If the preconditions are met, `removeZombie` never reverts.	Echidna	**Failed (Code quality issue)**
For every address, x, if `isStaked(x)`, then `!isZombie(x)`.	Echidna	**Passed**
For every address, x, if `latestStakedNode(x) <= latestConfirmed()`, then	Echidna	**Passed**

`currentChallenge(x) == NO_CHAL_INDEX`.		🟩

Codebase Maturity Evaluation

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

Category	Summary	Result
Arithmetic	While ArbOS and the arbitrator appear to handle integer overflows and underflows correctly, the `Rollup` contract re-implements built-in Solidity arithmetic operations incorrectly, resulting in unreachable code. Moreover, several of the bookkeeping operations in ArbOS lack documentation and testing (including the gas- and asset-related bookkeeping operations).	Moderate
Authentication / Access Controls	The protocol's extensive composability makes it difficult to track which components should access other components. The lack of documentation on both the different actors and the numerous privileged operations increases the likelihood of mistakes.	Satisfactory
Decentralization	While the system is presented as permissionless, it is heavily centralized. For example, the `Rollup` contract's owner can pause, upgrade, or otherwise change critical parameters of the `Rollup`; moreover, the number of participating validators is limited by a validator whitelist, and the owner of a chain is allowed to upgrade ArbOS. However, the Offchain Labs team indicated that it plans to remove these privileges in the future.	Moderate

Documentation	While the Arbitrum documentation provides a good high-level overview of the system, it lacks detail on the implementation and omits several system invariants. Additionally, the documentation does not explain the workings of the ArbOS escrow account, account store, or gas system; the purpose of the escrow account is also unclear. Appendix G contains a list of the system invariants that should be documented.	**Moderate**
Low-Level Manipulations	Several low-level manipulations would benefit from better documentation and testing; these include various type-casting operations (TOB-NITRO-ARBITATOR-3), the block-depth limit (TOB-NITRO-WAVM-1), the use of Merkle trees (e.g., the maintenance of leaves' paths upon tree updates), and the use of a proxy architecture and `delegatecalls` (TOB-NITRO-SC-1).	**Moderate**
Front-Running Resistance	Several privileged operations may create undocumented / unexpected front-running risks; these include calls to the precompiled Nitro contracts that change the system's behavior (TOB-NITRO-ARBOS-6) and the censorship of transactions through the block gas limit.	**Moderate**
Testing and Verification	The system's unit test coverage should be improved. As detailed in appendix E, ArbOS would benefit from fuzzing of its individual components as well its processing of inbox messages, which triggers operations in several other components. Additionally, the arbitrator code should be stress-tested through a differential fuzzing campaign focused on WASM–WAVM translation and WASM / WAVM execution. Finally, we recommend performing property-based testing of smart contracts including the `Rollup` to ensure that their properties hold.	**Moderate**

Summary of Findings: Arbitrator

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

ID	Title	Type	Severity
1	WASM binaries lack memory protections crucial for preventing corruption	Undefined Behavior	**Low**
2	Certain WASM binaries can crash wasm-validate or the arbitrator	Data Validation	**Informational**
3	Unresolved clippy warnings	Auditing and Logging	**Informational**
4	Risk of undefined behavior caused by export of fieldless enums from Rust to C	Undefined Behavior	**Informational**

Detailed Findings: Arbitrator

1. WASM binaries lack memory protections crucial for preventing corruption	
Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-NITRO-ARBITATOR-1
Target: `go-ethereum`, `arbitrator/wasm-libraries/soft-float`	

Description

The WASM binaries do not feature modern binary protections that are available out of the box in native binaries.

Arbitrum compiles several components to WASM to resolve disputes using proofs over the executed instructions. These binaries do not feature the same security protections of the natives ones; they lack most of the common checks that native binaries perform when a memory-unsafe operation occurs:

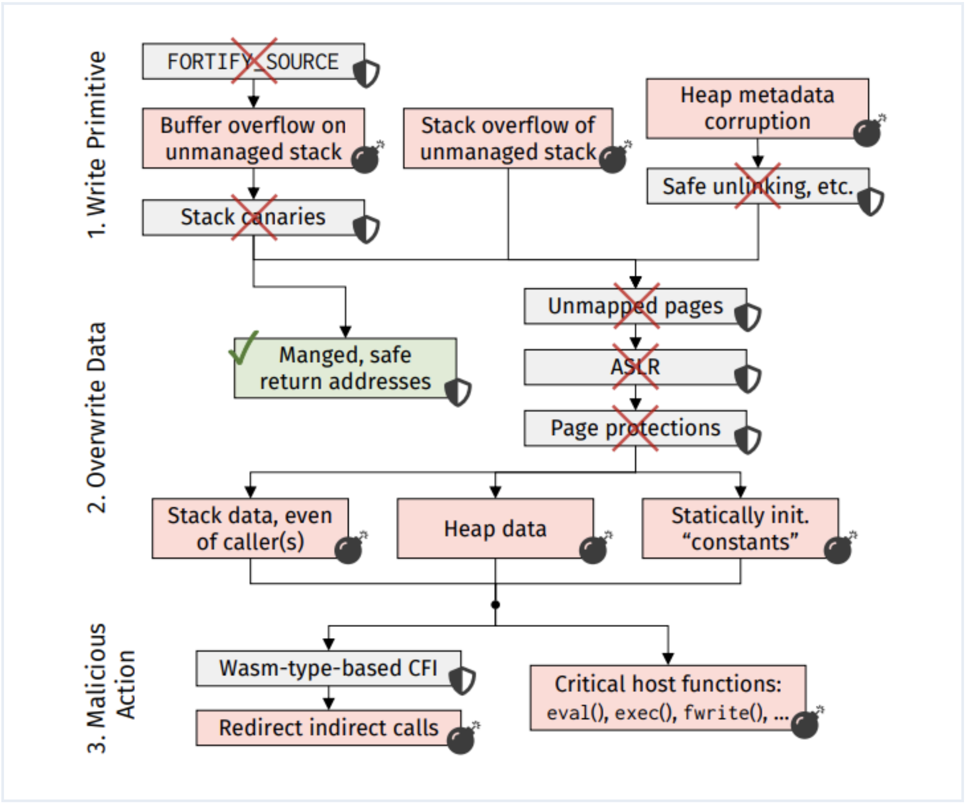


Figure 1.1: An overview of the attack primitives and the missing defenses in the WASM binaries

The USENIX 2020 paper “Everything Old is New Again: Binary Security of WebAssembly” describes in depth the binary defenses that are missing and the new attacks that can be executed in WASM binaries if memory-unsafe operations are performed.

While Go is a memory-safe language, it is still possible to write memory-unsafe code. Such code is used in some parts of geth and its C/C++ dependencies.

```
// byteArrayBytes returns a slice of the byte array v.
func byteArrayBytes(v reflect.Value, length int) []byte {
        var s []byte
        hdr := (*reflect.SliceHeader)(unsafe.Pointer(&s))
        hdr.Data = v.UnsafeAddr()
        hdr.Cap = length
        hdr.Len = length
        return s
}
```

Figure 1.2: `go-ethereum/rlp/unsafe.go#L27-L35`

Any C/C++ libraries linked in the WAVM execution (such as the softfloat library) may also be affected by the omission of these protections.

Exploit Scenario

Eve finds a memory-unsafe operation in geth or one of its dependencies. While the native versions of geth would immediately stop the execution if the stack or heap bookkeeping were invalid, the WASM version will continue the execution, enabling the creation of invalid proofs.

Recommendations

Short term, perform extensive testing of any memory-unsafe code that is compiled to WASM to protect the system against exploitable memory issues.

Long term, review the state of the WASM compiler to evaluate the maturity of the binary protections.

2. Certain WASM binaries can trigger wasm-validate and arbitrator crashes	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-ARBITATOR-2
Target: Arbitrator	

Description

The arbitrator performs the proof-generation process and invokes `wasm-validate` to ensure that WASM binaries are well formed. After a WASM binary has been validated, it is parsed and processed into a WAVM machine for execution in the generation of a proof.

```
$ ./prover --help
arbitrator-prover 0.1.0

USAGE:
    prover [FLAGS] [OPTIONS] <binary>
```

Figure 2.1: Part of the `arbitrator-prover` command line

However, certain WASM binaries can trigger a panic in the arbitrator:

```
thread 'main' panicked at 'Module has no code', prover/src/machine.rs:448:9
thread 'main' panicked at 'Cannot call `finish()` on `Err(Err::Incomplete(_))`: this
result means that the parser does not have enough data to decide, you should gather
more data and try to reapply  the parser instead',
/home/g/.cargo/registry/src/github.com-1ecc6299db9ec823/nom-7.0.0/src/internal.rs:41
:9
thread 'main' panicked at 'range end index 64 out of range for slice of length 2',
prover/src/lib.rs:51:75
thread 'main' panicked at 'range start index 1 out of range for slice of length 0',
prover/src/machine.rs:441:17
thread 'main' panicked at 'Out-of-bounds data memory init with offset 4 and size 0',
prover/src/machine.rs:388:21
thread 'main' panicked at 'range start index 1 out of range for slice of length 0',
prover/src/machine.rs:455:17
```

Figure 2.2: A list of different arbitrator panics

Certain WASM binaries can even cause a crash in `wasm-validate`:

```
$ gdb --args wasm-validate
wasm-fuzzing-corpus/wasm/d3bac5bb8061dfb6c918851a38b8af75cc14d11d
...
Program received signal SIGSEGV, Segmentation fault.
0x00005555555c2210 in wabt::ReadBinary(void const*, unsigned long,
wabt::BinaryReaderDelegate*, wabt::ReadBinaryOptions const&) ()
(gdb) bt
#0  0x00005555555c2210 in wabt::ReadBinary(void const*, unsigned long,
wabt::BinaryReaderDelegate*, wabt::ReadBinaryOptions const&) ()
#1  0x000055555559f47c in wabt::ReadBinaryIr(char const*, void const*, unsigned
long, wabt::ReadBinaryOptions const&, std::vector<wabt::Error,
std::allocator<wabt::Error> >*, wabt::Module*) ()
#2  0x0000555555597a89 in ProgramMain(int, char**) ()
#3  0x00007ffff767abf7 in __libc_start_main (main=0x5555555896a0 <main>, argc=2,
argv=0x7fffffffdd58, init=<optimized out>, fini=<optimized out>,
rtld_fini=<optimized out>, stack_end=0x7fffffffdd48)
    at ../csu/libc-start.c:310
#4  0x0000555555596f1e in _start ()
```

Figure 2.3: A stack trace showing a crash in `wasm-validate` 1.0.24

Exploit Scenario

Alice tries to generate a proof, but the arbitrator crashes, blocking her from resolving the dispute.

Recommendations

Short term, ensure that the arbitrator and `wasm-validate` will not crash regardless of their input.

Long term, use `afl.rs` or another fuzzing tool to stress-test the arbitrator code.

3. Unresolved clippy warnings

Severity: **Informational**	Difficulty: **High**
Type: Auditing and Logging	Finding ID: TOB-NITRO-ARBITATOR-3
Target: Arbitrator	

Description

We ran static analyzer Clippy over the arbitrator code to check the code for the most common Rust mistakes.

The execution of Clippy resulted in 248 warnings, including warnings regarding unsafe casting operations, the use of identical `if` and `elif` conditions, and missing documentation.

```
warning: casting `u64` to `usize` may truncate the value on targets with 32-bit wide
pointers
    --> prover/src/machine.rs:1217:36
     |
1217 |                     self.pc.inst = inst.argument_data as usize;
     |                                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^
     |
     = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#cast_possible_truncation

[...]

error: this `if` has identical blocks
    --> prover/src/machine.rs:1599:63
     |
1599 |                   if idx > self.global_state.bytes32_vals.len() {
     |  _______________________________________________________________^
1600 | |                     self.status = MachineStatus::Errored;
1601 | |                 } else if !module
     | |_________________^
     |
     = note: `#[deny(clippy::if_same_then_else)]` on by default
note: same as this
    --> prover/src/machine.rs:1604:17
```

```
     |
1604 | /                  {
1605 | |                      self.status = MachineStatus::Errored;
1606 | |                  }
     | |__________________^
     = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#if_same_then_else


[...]

warning: unsafe function's docs miss `# Safety` section
   --> prover/src/lib.rs:291:1
    |
291 | / pub unsafe extern "C" fn arbitrator_gen_proof(mach: *mut Machine) ->
RustByteArray {
292 | |     let mut proof = (*mach).serialize_proof();
293 | |     let ret = RustByteArray {
294 | |         ptr: proof.as_mut_ptr(),
...   |
299 | |     ret
300 | | }
    | |_^
    |
    = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#missing_safety_doc

[...]
```

Figure 3.1: Warnings raised in the execution of `cargo clippy --workspace -- -W clippy::pedantic`

Recommendations

Short term, address all of the warnings raised by Clippy.

Long term, integrate Clippy into the CI pipeline by using the following command: `cargo clippy --workspace -- -W clippy::pedantic`.

4. Risk of undefined behavior caused by export of fieldless enums from Rust to C

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-NITRO-ARBITATOR-4
Target: `arbitrator/prover/src/lib.rs`	

Description

The arbitrator exports fieldless enums to the C representation. In C, the size of an enum is defined by the implementation; as a result, the export of a fieldless enum may lead to undefined behavior.

For example, the arbitrator exports the fieldless `CMachineStatus` enum to its C representation.

```
 // C requires enums be represented as `int`s, so we need a new type for this :/
#[derive(Clone, Copy, PartialEq, Eq, Debug)]
#[repr(C)]
pub enum CMachineStatus {
    Running,
    Finished,
    Errored,
    TooFar,
}
```

Figure 4.1: `arbitrator/prover/src/lib.rs#L238-L246`

The Rust documentation includes the following warning:

```
repr(C) is equivalent to one of repr(u*) (see the next section) for fieldless enums.
The chosen size is the default enum size for the target platform's C application
binary interface (ABI). Note that enum representation in C is implementation
defined, so this is really a "best guess". In particular, this may be incorrect when
the C code of interest is compiled with certain flags.
```

Figure 4.2: `https://doc.rust-lang.org/nomicon/other-reprs.html#reprc`

The Rust Unsafe Code Guidelines also include a warning:

```
Note: some C compilers offer flags (e.g., -fshort-enums) that change the layout of
enums from the default settings that are standard for the platform. The integer size
selected by #[repr(C)] is defined to match the default settings for a given target,
when no such flags are supplied. If interop with code that uses other flags is
desired, then one should either specify the sizes of enums manually or else use an
alternate target definition that is tailored to the compiler flags in use.
```

Figure 4.3: *`https://github.com/rust-lang/unsafe-code-guidelines/reference/src/layout/enums.md#layout-of-a-fieldless-enum`*

Thus, the translation of `CMachineStatus` depends on the version of the C compiler and the flags it uses.

Recommendations

Short term, avoid using fieldless enums with `repr(C)`.

Long term, carefully review the Rust documentation and Unsafe Code Guidelines, particularly the warnings about code representation and evaluate any risky features used in the protocol.

Summary of Findings: WAVM

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

ID	Title	Type	Severity
1	Block-depth limit is not checked on-chain	Data Validation	High
2	Inconsistencies in arbitrator and one-step-proof contract error statuses	Data Validation	High
3	Lack of on-chain validation of branching instruction destinations	Data Validation	High
4	Divergence in the casting rules between the arbitrator and the one step-proof contract	Data Validation	High
5	Inconsistent handling of out-of-bounds index access in the execution of global state access opcodes	Data Validation	High

Detailed Findings: WAVM

1. Block-depth limit is not checked on-chain	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-WAVM-1
Target: `machine.rs`, `OneStepProver0.sol`	

Description

The arbitrator maintains a block-depth limit that is not checked on-chain. This lack of data validation could enable an attacker to prove an invalid state on-chain.

When executing block and branching instructions, the arbitrator checks that `block_depth` is not less than zero:

```
            Opcode::Block => {
                 [...]
                 self.pc.block_depth += 1;
                 [...]
             }
             Opcode::EndBlock => {
                 assert!(self.pc.block_depth > 0);
                 self.pc.block_depth -= 1;
                 [...]
             }
             Opcode::EndBlockIf => {
                 [...]
                     assert!(self.pc.block_depth > 0);
                     self.pc.block_depth -= 1;
                     [...]
                 }
             }
             [...]
             Opcode::Branch => {
                 assert!(self.pc.block_depth > 0);
                 self.pc.block_depth -= 1;
                 [...]
             }
            Opcode::BranchIf => {
                [...]
                    assert!(self.pc.block_depth > 0);
                    self.pc.block_depth -= 1;
                    [...]
                }
            }
```

Figure 1.1: `arbitrator/prover/src/machine.rs#L1179-L1235`

The one-step-proof contract lacks such a check:

```
        function executeBlock(Machine memory mach, Module memory, Instruction calldata
inst, bytes calldata) internal pure {
                uint32 targetPc = uint32(inst.argumentData);
                require(targetPc == inst.argumentData, "BAD_BLOCK_PC");
                PcStacks.push(mach.blockStack, targetPc);
        }

        function executeBranch(Machine memory mach, Module memory, Instruction
calldata, bytes calldata) internal pure {
                mach.functionPc = PcStacks.pop(mach.blockStack);
        }

        function executeBranchIf(Machine memory mach, Module memory, Instruction
calldata, bytes calldata) internal pure {
                Value memory cond = ValueStacks.pop(mach.valueStack);
                if (cond.contents != 0) {
                        // Jump to target
                        mach.functionPc = PcStacks.pop(mach.blockStack);
                }
        }
         [...]
        function executeEndBlock(Machine memory mach, Module memory, Instruction
calldata, bytes calldata) internal pure {
                PcStacks.pop(mach.blockStack);
        }
```

```
	function executeEndBlockIf(Machine memory mach, Module memory, Instruction
calldata, bytes calldata) internal pure {
		Value memory cond = ValueStacks.peek(mach.valueStack);
		if (cond.contents != 0) {
			PcStacks.pop(mach.blockStack);
		}
	}
```

Figure 1.2: `solgen/src/osp/OneStepProver0.sol#L58-L296`

As a result, the generated WAVM code will have an incorrect block-depth limit, which could enable an attacker to prove an incorrect state on-chain and wrongfully win a challenge.

Exploit Scenario

Eve finds a transaction that causes the WAVM code to execute more `EndBlock` instructions than necessary. None of the nodes can execute the transaction. While the Offchain Labs team is working on a fix, Eve creates a rollup node that includes the buggy transaction. Alice and Bob then challenge Eve's rollup node. Because the block-depth limit is not validated on-chain, Eve is able to prove the execution, effectively stealing Alice's and Bob's stakes.

Recommendations

Short term, have the one-step-proof contract track the block-depth limit.

Long term, implement the WAVM code and design recommendations outlined in appendix D.

2. Inconsistencies in arbitrator and one-step-proof contract error statuses

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-WAVM-2
Target: `machine.rs`, `OneStepProver0.sol`	

Description

The arbitrator and the `OneStepProver0` contract check two conditions when executing the `ReadInboxMessage` opcode. However, they perform these checks in different orders, which can lead to inconsistencies.

Execution of the `ReadInboxMessage` opcode can result in two error statuses: `TOO_FAR` and `Errored`.

The one-step-proof contract first checks whether the index of the inbox sequencer is too far (i.e., whether it exceeds the machine's inbox limit), in which case the `TooFar` status is triggered. It then checks whether the pointer references the correct location; if it does not, the `Errored` status is triggered.

```
        if (inst.argumentData == Instructions.INBOX_INDEX_SEQUENCER && msgIndex >=
execCtx.maxInboxMessagesRead) {
            mach.status = MachineStatus.TOO_FAR;
            return;
        }

        if (ptr + 32 > mod.moduleMemory.size || ptr % LEAF_SIZE != 0) {
            mach.status = MachineStatus.ERRORED;
            return;
        }
```

Figure 2.1: `solgen/src/osp/OneStepProverHostIo.sol#L238-L246`

The arbitrator checks these conditions in the opposite order: it first checks the pointer location and then checks the index.

```
                if ptr as u64 + 32 > module.memory.size() {
                    self.status = MachineStatus::Errored;
                } else {
                    assert!(
                        inst.argument_data <= (InboxIdentifier::Delayed as u64),
                        "Bad inbox identifier"
                    );
                    let inbox_identifier =
argument_data_to_inbox(inst.argument_data).unwrap();
                    if let Some(message) =
self.inbox_contents.get(&(inbox_identifier, msg_num)) {
                        let offset = usize::try_from(offset).unwrap();
                        let len = std::cmp::min(32,
message.len().saturating_sub(offset));
                        let read = message.get(offset..(offset +
len)).unwrap_or_default();
                        if module.memory.store_slice_aligned(ptr.into(), read) {
                            self.value_stack.push(Value::I32(len as u32));
                        } else {
                            self.status = MachineStatus::Errored;
                        }
                    } else {
                        self.status = MachineStatus::TooFar;
                    }

turn;
        }
```

Figure 2.2: `arbitrator/prover/src/machine.rs#L1659-L1678`

If both conditions evaluate to `true` (the pointer references an incorrect location and the index is too far), the arbitrator's status will be `Errored`, and the one-step-proof contract's status will be `TooFar`.

An attacker could leverage this discrepancy to make validators believe that the `Errored` status had been triggered when the on-chain contract's status was actually `TooFar`.

Exploit Scenario

Eve finds a transaction that causes the inbox pointer to reference an incorrect location and the index of the inbox sequencer to be too far. She then creates a corresponding rollup node. None of the nodes can execute the transaction. While the Offchain Labs team is working on a fix, Eve creates a node leading to the `TooFar` status. Alice and Bob challenge Eve's rollup node. Eve is able to prove the execution, effectively stealing Alice's and Bob's stakes.

Recommendations

Short term, have the arbitrator and the one-step-proof contract check the two conditions in the same order when executing `ReadInboxMessage` opcode. This will ensure that the arbitrator and the one-step-proof contract have the same status if there is an error in the execution of `ReadInboxMessage`.

Long term, implement the WAVM code and design recommendations outlined in appendix D.

3. Lack of on-chain validation of branching instruction destinations

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-WAVM-3
Target: `machine.rs`, `OneStepProver0.sol`	

Description

The arbitrator checks that the size of the next block / instruction is within the code size limit, but the one-step-proof contract does not. Without this check, an attacker might be able to prove an invalid state on-chain

The arbitrator, through `Machine::test_next_instruction`, checks that the next instruction to be executed is within the code size limit. It performs this check for all branching instructions (`ArbitraryJumpIf`, `Branch`, and `BranchIf` instructions).

```
             Opcode::ArbitraryJumpIf => {
                let x = self.value_stack.pop().unwrap();
                if !x.is_i32_zero() {
                    self.pc.inst = inst.argument_data as usize;
                    Machine::test_next_instruction(&module, &self.pc);
                }
            }
            Opcode::Branch => {
                assert!(self.pc.block_depth > 0);
                self.pc.block_depth -= 1;
                self.pc.inst = self.block_stack.pop().unwrap();
                Machine::test_next_instruction(&module, &self.pc);
            }
            Opcode::BranchIf => {
                let x = self.value_stack.pop().unwrap();
                if !x.is_i32_zero() {
                    assert!(self.pc.block_depth > 0);
                    self.pc.block_depth -= 1;
                    self.pc.inst = self.block_stack.pop().unwrap();
                    Machine::test_next_instruction(&module, &self.pc);
                }
            }
```

Figure 3.1: `arbitrator/prover/src/machine.rs#L1214-L1235`

```
fn test_next_instruction(module: &Module, pc: &ProgramCounter) {

    assert!(module.funcs[pc.func].code.len() > pc.inst);
    }
```

Figure 3.2: `arbitrator/prover/src/machine.rs#L1120-L1122`

It executes the same check when a block is created but does so without calling `test_next_instruction`:

```
              Opcode::Block => {
                 let idx = inst.argument_data as usize;
                 self.block_stack.push(idx);
                 self.pc.block_depth += 1;
                 assert!(module.funcs[self.pc.func].code.len() > idx);
             }
```

Figure 3.3: `arbitrator/prover/src/machine.rs#L1179-L1184`

This check is missing from the one-step-proof contract:

```
       function executeBlock(Machine memory mach, Module memory, Instruction calldata
inst, bytes calldata) internal pure {
              uint32 targetPc = uint32(inst.argumentData);
              require(targetPc == inst.argumentData, "BAD_BLOCK_PC");
              PcStacks.push(mach.blockStack, targetPc);
       }

       function executeBranch(Machine memory mach, Module memory, Instruction
calldata, bytes calldata) internal pure {
              mach.functionPc = PcStacks.pop(mach.blockStack);
       }

       function executeBranchIf(Machine memory mach, Module memory, Instruction
calldata, bytes calldata) internal pure {
              Value memory cond = ValueStacks.pop(mach.valueStack);
              if (cond.contents != 0) {
                     // Jump to target
                     mach.functionPc = PcStacks.pop(mach.blockStack);
              }
       }
```

Figure 3.4: `solgen/src/osp/OneStepProver0.sol#L58-L74`

As a result, if the WAVM code loads an incorrect instruction (one that exceeds the code size limit), an attacker may be able to prove an incorrect state on-chain and wrongfully win a challenge.

Exploit Scenario
Eve finds a transaction that causes the WAVM code to branch to an instruction that exceeds the code size limit. She then creates a rollup node containing the transaction. None of the nodes can execute the transaction. While the Offchain Labs team is working on a fix, Alice and Bob challenge Eve's rollup node. Eve is able to prove the execution, effectively stealing Alice's and Bob's stakes.

Recommendations
Short term, take the following steps:

- Revert in the one-step-proof contract for branching instruction and `Block` if the next block / instruction is not within the code size limit. This will ensure that the one-step-proof contract has the same behavior as the validator code.

- Have the arbitrator use `test_next_instruction` instead of an ad hoc check when executing the `Opcode::Block` opcode. The ad hoc check it currently performs is a duplicate of `test_next_instruction`, which increases the complexity of the code and the likelihood of bugs.

Long term, implement the WAVM code and design recommendations outlined in appendix D.

4. Discrepancy in the casting rules of the arbitrator and one-step-proof contract

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-WAVM-4
Target: `machine.rs`, `OneStepProver0.sol`	

Description

The arbitrator and the one-step-proof contract do not follow the same casting rules; nor do they perform the same checks for casting overflows. These inconsistencies could enable an attacker to prove an invalid state on-chain.

For example, when executing the `GetGlobalStateBytes32` opcode, the arbitrator checks that the pointer is a u32, while the one-step-proof contract does not:

```
            Opcode::GetGlobalStateBytes32 => {
                let ptr = self.value_stack.pop().unwrap().assume_u32();
                let idx = self.value_stack.pop().unwrap().assume_u32() as usize;
```

Figure 4.1: *`arbitrator/prover/src/machine.rs#L1596-L11599`*

```
    function executeGetOrSetBytes32(
        Machine memory mach,
        Module memory mod,
        GlobalState memory state,
        Instruction calldata inst,
        bytes calldata proof
     ) internal pure {
        uint256 ptr = ValueStacks.pop(mach.valueStack).contents;
        uint32 idx = Values.assumeI32(ValueStacks.pop(mach.valueStack));
```

Figure 4.2: *`solgen/src/osp/OneStepProverHostIo.sol#L38-L46`*

As a result, if a bug in the generated code causes the pointer value to exceed 2**32, the arbitrator will not be able to process the transaction, but the one-step-proof contract will be able to process it. An attacker could use such a discrepancy to steal validators' stakes.

Exploit Scenario

Eve finds a transaction that causes the size of the pointer in `GetGlobalStateBytes32` to exceed 2**32. None of the nodes can execute the transaction. While the Offchain Labs team is working on a fix, Eve creates a rollup node that includes the buggy transaction. Alice and Bob then challenge Eve's rollup node. Eve is able to prove the execution, effectively stealing Alice's and Bob's stakes.

Recommendations

Short term, ensure that for each use of `assume_X` in the arbitrator code, there exists a corresponding use of `Values.assumeI32` in the one-step-proof contract code.

Long term, implement the WAVM code and design recommendations outlined in appendix D.

5. Inconsistent handling of out-of-bounds index access in the execution of global state access opcodes

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-WAVM-5
Target: `machine.rs`, `OneStepProver0.sol`	

Description

The arbitrator and the one-step-proof contract exhibit different behavior if they access an index out of bounds when executing the global state access opcodes. This divergence could enable an attacker to prove an invalid state on-chain.

For example, when executing the `GetGlobalStateBytes32` opcode, the arbitrator will panic if the index is equal to `global_state.u64_vals.len` (the `else` branch is used for `idx == self.global_state.u64_vals.len()`.

```
                if idx > self.global_state.u64_vals.len() {
                    self.status = MachineStatus::Errored;
                } else {
                    self.value_stack
                        .push(Value::I64(self.global_state.u64_vals[idx]));
                }
```

Figure 5.1: `arbitrator/prover/src/machine.rs#L1645-L1650`

By contrast, an index of `global_state.u64_vals.len()` would trigger the `Errored` status in the one-step-proof contract:

```
        if (idx >= GlobalStates.U64_VALS_NUM) {
            mach.status = MachineStatus.ERRORED;
            return;
        }
```

Figure 5.2: `solgen/src/osp/OneStepProverHostIo.sol#L83-L86`

This means that if `GetGlobalStateBytes32` were used with an index of `global_state.u64_vals.len()`, the node would panic, but the one-step-proof contract could be used to prove the execution.

There are similar inconsistencies in the arbitrator's and one-step-proof contract's handling of the following opcodes:

- `GetGlobalStateBytes32`
- `SetGlobalStateBytes32`
- `SetGlobalStateU64`

Exploit Scenario

Eve finds a transaction that causes an index of `global_state.u64_vals.len()` to be used in the execution of `GetGlobalStateBytes32`. None of the nodes can execute the transaction. While the Offchain Labs team is working on a fix, Eve creates a rollup node that includes the buggy transaction. Alice and Bob then challenge Eve's rollup node. Eve is able to prove the execution, effectively stealing Alice's and Bob's stakes.

Recommendations

Short term, use the >= operator instead of > in the length checks when executing the `GetGlobalStateBytes32`, `SetGlobalStateBytes32`, `GetGlobalStateU64`, and `SetGlobalStateU64` opcodes. This will ensure that both implementations of the WAVM code have the same behavior.

Long term, implement the WAVM code and design recommendations outlined in appendix D.

Summary of Findings: go-ethereum

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

ID	Title	Type	Severity
1	Lack of documentation on geth changes	Auditing and Logging	**Undetermined**
2	Upstream fixes missing from geth fork	Auditing and Logging	**Undetermined**
3	totalBalanceDelta does not account for ether sent to self-destructed contracts	Data Validation	**Low**

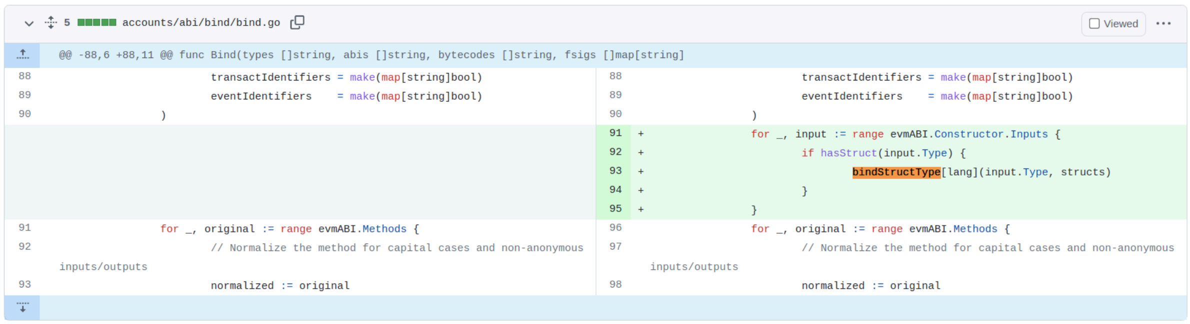
Detailed Findings: go-ethereum

1. Lack of documentation on geth changes	
Severity: **Undetermined**	Difficulty: **High**
Type: Auditing and Logging	Finding ID: TOB-NITRO-GETH-1
Target: `go-ethereum`	

Description

Offchain Labs's fork of geth includes several modifications that are undocumented..

These include the modifications made to `bind.go`, shown in figure 1.1:



```
5 accounts/abi/bind/bind.go                                                                    Viewed

@@ -88,6 +88,11 @@ func Bind(types []string, abis []string, bytecodes []string, fsigs []map[string]
                transactIdentifiers = make(map[string]bool)      |                transactIdentifiers = make(map[string]bool)
                eventIdentifiers    = make(map[string]bool)      |                eventIdentifiers    = make(map[string]bool)
        )                                                        |        )
                                                                 | +      for _, input := range evmABI.Constructor.Inputs {
                                                                 | +              if hasStruct(input.Type) {
                                                                 | +                      bindStructType[lang](input.Type, structs)
                                                                 | +              }
                                                                 | +      }
        for _, original := range evmABI.Methods {                |        for _, original := range evmABI.Methods {
                // Normalize the method for capital cases and non-anonymous |        // Normalize the method for capital cases and non-anonymous
inputs/outputs                                                   | inputs/outputs
                normalized := original                           |                normalized := original
```

Figure 1.1: The differences between `bind.go` in the Offchain Labs fork of geth and the original version of geth

The purpose and implications of these changes are unclear.

Recommendations

Short term, keep the changes made to geth to a minimum, and document all changes. Clear documentation will aid in the maintenance of the codebase and the merging of upstream changes.

Long term, establish a detailed process for applying upstream geth changes, and ensure that all changes are applied with as little delay as possible.

2. Upstream fixes missing from geth fork

Severity: **Undetermined**	Difficulty: **High**
Type: Auditing and Logging	Finding ID: TOB-NITRO-GETH-2
Target: go-ethereum	

Description

Offchain Labs's fork of geth omits the changes introduced in geth 1.10.15. These changes include a fix for a peer-to-peer connection issue that could cause nodes to become locked:

> *This release resolves a few regressions introduced by the previous release. Most importantly, it fixes an issue that could cause peer-to-peer 'eth' connections to lock up.*

Figure 2.1: An excerpt of the geth 1.10.15 release notes

Recommendations

Short term, apply the geth 1.10.15 changes, which include a fix for a peer-to-peer attack vector.

Long term, establish a detailed process for applying upstream geth upstream, and ensure that all changes are applied with as little delay as possible.

3. totalBalanceDelta does not account for ether sent to self-destructed contracts

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-NITRO-GETH-3
Target: `go-ethereum`, `arbos/block_processor.go`	

Description

A transfer of ether to a contract that has self-destructed will not be reflected in the `totalBalanceDelta` value.

Offchain Labs introduced `totalBalanceDelta` as part of its modifications to geth. The value represents the total amount of ETH that has been added to or removed from the chain.

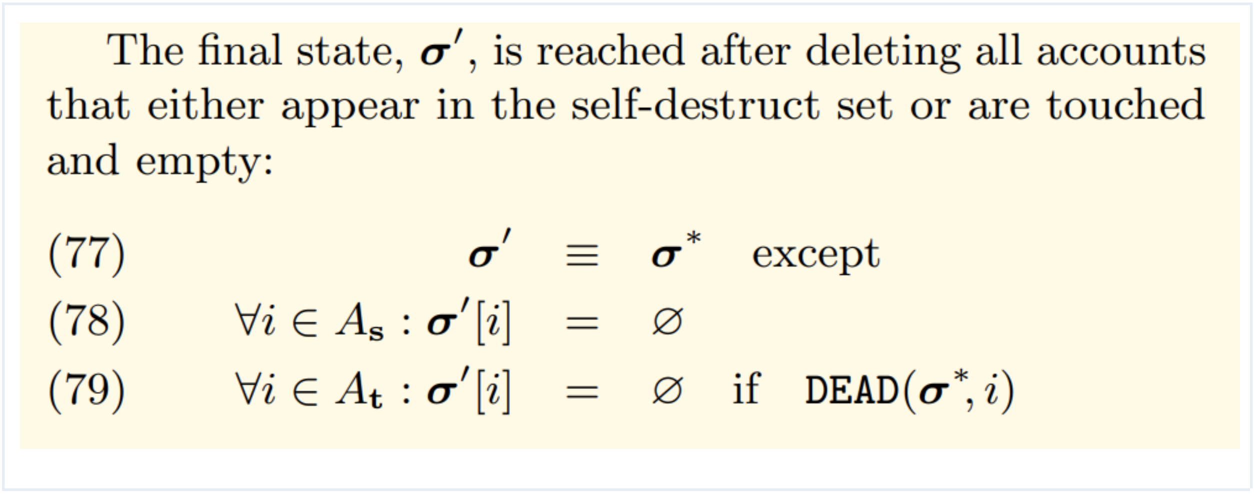
When a contract executes `selfdestruct`, it is marked for destruction, and its balance is set to zero:

```
func (s *StateDB) Suicide(addr common.Address) bool {
	stateObject := s.getStateObject(addr)
	if stateObject == nil {
		return false
	}
	s.journal.append(suicideChange{
		account:     &addr,
		prev:        stateObject.suicided,
		prevbalance: new(big.Int).Set(stateObject.Balance()),
	})
	stateObject.markSuicided()
	s.totalBalanceDelta.Sub(s.totalBalanceDelta, stateObject.data.Balance)
	stateObject.data.Balance = new(big.Int)

	return true
}
```

Figure 3.1: `core/state/statedb.go#L447-L462`

When the transaction is finalized, the accounts marked for destruction (and their balances) are deleted:



The final state, $\boldsymbol{\sigma}'$, is reached after deleting all accounts that either appear in the self-destruct set or are touched and empty:

$$
\begin{aligned}
&(77) & \boldsymbol{\sigma}' &\equiv \boldsymbol{\sigma}^* \quad \text{except} \\
&(78) & \forall i \in A_\mathbf{s} : \boldsymbol{\sigma}'[i] &= \varnothing \\
&(79) & \forall i \in A_\mathbf{t} : \boldsymbol{\sigma}'[i] &= \varnothing \quad \text{if} \quad \texttt{DEAD}(\boldsymbol{\sigma}^*, i)
\end{aligned}
$$

Figure 3.2: Yellow paper (section6.2, "Execution")

Any ether sent to an account marked for destruction will effectively be burned. However, the balance delta (tracked through `totalBalanceDelta` in the modified geth version) will not account for the burning of the ether. As a result, the value of `totalBalanceDelta` will be incorrect.

This issue does not appear to have a direct impact on ArbOS; however, it could prevent Offchain Labs from pushing the geth modifications upstream.

Exploit Scenario

A new invariant is added to the Arbitrum Nitro system. This new invariant will cause a node panic if the actual total supply of ether (based on the total of all accounts' balances) is different from the total supply tracked through `totalBalanceDelta`. Eve sends ether to a contract marked for destruction. The transaction results in a total supply discrepancy, causing the Arbitrum Nitro system to enter an invalid state.

Recommendations

Short term, revisit the tracking of `totalBalanceDelta`, and design a bookkeeping system that is robust against self-destruct corner cases. (See TOB-NITRO-ARBOS-3 and TOB-NITRO-ARBOS-7 for additional examples of corner cases.)

Long term, develop comprehensive centralized documentation on the system's invariants and thoroughly test those invariants.

Summary of Findings: ArbOS

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

ID	Title	Type	Severity
1	Unimplemented features could facilitate an ArbOS node crash	Data Validation	High
2	Confusing retryable ticket–redemption mechanism	Auditing and Logging	Low
3	Calls to selfdestruct(this) break the totalBalanceDelta bookkeeping	Data Validation	Low
4	Issues in the Go and Rust compilers' WASM back ends	Undefined Behavior	Undetermined
5	Risk of a node crash due to unchecked ticket-opening errors	Data Validation	High
6	Unspecified gas rules for precompiled Nitro contracts	Undefined Behavior	Low
7	Fragile expectedBalanceDelta bookkeeping	Data Validation	Informational
8	Processing of malformed retryable ticket messages can cause an ArbOS crash	Data Validation	Informational
9	Invalid transactions count toward the block gas limit	Data Validation	Medium
10	Confusing rule for calculating the amount of gas remaining in a block	Data Validation	Informational
11	Use of an unofficial Brotli library for message compression	Data Validation	Undetermined

12	Escrow addresses can be used to move stolen funds	Auditing and Logging	**Low**
13	Confusing EOA address remapping rules	Access Controls	**Medium**
14	Unreachable mechanism for disabling the default aggregator	Access Controls	**Informational**
15	Risk of a node crash due to unchecked PosterDataCost errors	Data Validation	**High**
16	Infinite loop caused by parsing of malformed sequencer messages	Data Validation	**Medium**
17	ArbOS bottleneck caused by RLP decoding loop	Data Validation	**Medium**
18	Broken gasLeft computation	Data Validation	**Medium**
19	Aggregators can block user transactions by setting a high fixed fee	Data Validation	**Medium**
20	Aggregator can steal the tip sent to other aggregators	Data Validation	**Medium**
21	Aggregators can steal extra fees by updating their rates	Data Validation	**Medium**
22	Aggregators can censor the redemption of retryable tickets	Data Validation	**Low**
23	Fragile retryable ticket ID scheme	Data Validation	**Informational**

Detailed Findings: ArbOS

1. Unimplemented features could facilitate an ArbOS node crash

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-1
Target: `go-ethereum`	

Description

ArbOS can receive L2 messages of different types. However, several of these message types have not been implemented. If the ArbOS nodes receive a message of an unimplemented type, they will panic; an attacker could take advantage of this behavior to remotely crash the ArbOS nodes.

The unimplemented message types are shown in figures 1.1 and 1.2.

```
	case L1MessageType_L2Message:
		return parseL2Message(bytes.NewReader(msg.L2msg), msg.Header.Poster, 
msg.Header.RequestId, 0)
	case L1MessageType_SetChainParams:
		panic("unimplemented")
	case L1MessageType_EndOfBlock:
		return nil, nil
	case L1MessageType_L2FundedByL1:
		panic("unimplemented")
	  [...]
	case L1MessageType_BatchForGasEstimation:
		panic("unimplemented")
```

Figure 1.1: `arbos/incomingmessage.go#L172-L187`

```
	case L2MessageKind_NonmutatingCall:
		panic("unimplemented")
```

```
     [...]
    case L2MessageKind_SignedCompressedTx:
        panic("unimplemented")
```

Figure 1.2: `arbos/incomingmessage.go#L235-L276`

Exploit Scenario

Eve sends a `BatchForGasEstimation` message to the chain. As a result, the node that Bob is running crashes.

Recommendations

Short term, implement all missing message types and use errors instead of panics to handle any issues caused by unimplemented features. Using panics to handle these issues is risky and could allow an attacker to remotely crash the nodes.

Long term, create a robust PR merging process and ensure that unimplemented features are not merged into the master branch of the repository.

2. Confusing retryable ticket–redemption mechanism	
Severity: **Low**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-NITRO-ARBOS-2
Target: ArbOS	

Description

The retryable ticket–redemption mechanism has several downsides that are not explained in the documentation. Users who are unaware of the risks may misuse the mechanism or break the third-party integrations.

ArbOS allows externally owned accounts (EOAs) and smart contracts to manually redeem existing retryable tickets by calling the `ArbRetryable.redeem` function. If the function is called on an existing ticket that has not expired, the call will result in the emission of a special event, which will signal ArbOS to execute the transaction in the current block.

```
func (con ArbRetryableTx) Redeem(c ctx, evm mech, ticketId bytes32) (bytes32, error)
{
	retryableState := c.state.RetryableState()
	byteCount, err := retryableState.RetryableSizeBytes(ticketId,
evm.Context.Time.Uint64())
	if err != nil {
		return hash{}, err
	}
	writeBytes := util.WordsForBytes(byteCount)
	if err := c.Burn(params.SloadGas * writeBytes); err != nil {
		return hash{}, err
	}

	retryable, err := retryableState.OpenRetryable(ticketId,
evm.Context.Time.Uint64())
	if err != nil {
		return hash{}, err
	}
	…
	err = con.RedeemScheduled(c, evm, ticketId, retryTxHash, nonce, gasToDonate,
c.caller)
	if err != nil {
		return hash{}, err
	}
    …
```

Figure 2.1: Part of the implementation of `redeem` in the precompiled contracts

To know whether a retryable ticket's redemption was successful, a user must manually check the blockchain logs using a full node. . While this is confusing for EOAs, it is more problematic for smart contracts, which may incorrectly expect that they can interact with the state of the blockchain after a retryable ticket has been executed (assuming the execution did not revert). Moreover, there can be multiple calls to the `ArbRetryable.redeem` function, and thus multiple events, for a single retryable ticket, though the redemption will be executed only once; a redemption can also be initiated and canceled in the same transaction, in which case it will not be executed.

Exploit Scenario

Alice creates a smart contract that redeems retryable tickets to execute trades in L2. She expects all of the trades to succeed, but some of them fail without her realizing it.

Recommendations

Short term, properly document the risks associated with the redemption of retryable tickets. That way, users will understand that L2 contracts are unable to observe whether a retryable ticket succeeds or fails.

Long term, review the use cases of each public component and API to ensure that they are easy to understand and in line with users' expectations.

3. Calls to selfdestruct(this) break the totalBalanceDelta bookkeeping

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-3
Target: `go-ethereum`, `arbos/block_processor.go`	

Description

By executing `selfdestruct(this)`, an attacker can break the total supply invariant and force all nodes to print an error message.

The execution of `selfdestruct(this)` can affect the tracking of the total amount of ETH that has been added to or removed from the chain (a functionality introduced in Offchain Labs's modifications of geth). ArbOS assumes that the delta will be equal to the amount added or removed through L2 transactions:

```
case *types.ArbitrumDepositTx:
	// L1->L2 deposits add eth to the system
	expectedBalanceDelta.Add(expectedBalanceDelta, txInner.Value)
case *types.ArbitrumSubmitRetryableTx:
	// Retryable submission can include a deposit which adds eth to the system
	expectedBalanceDelta.Add(expectedBalanceDelta, txInner.DepositValue)
}
```

Figure 3.1: `arbos/block_processor.go#L208-L214`

```
} else if txLog.Address == ArbSysAddress && txLog.Topics[0] ==
L2ToL1TransactionEventID {
	// L2->L1 withdrawals remove eth from the system
	event := &precompilesgen.ArbSysL2ToL1Transaction{}
	err := util.ParseL2ToL1TransactionLog(event, txLog)
	if err != nil {
		log.Error("Failed to parse L2ToL1Transaction log", "err", err)
	} else {
		expectedBalanceDelta.Sub(expectedBalanceDelta, event.Callvalue)
	}
```

Figure 3.2: `arbos/block_processor.go#L247-L255`

If the delta values computed by geth and ArbOS diverge such that the chain holds more ether than expected, the node will panic. If it holds less ether than expected, the node will print an error message.

```
if balanceDelta.Cmp(expectedBalanceDelta) != 0 {
	// Panic if funds have been minted or debug mode is enabled (i.e. this is a
test)
	if balanceDelta.Cmp(expectedBalanceDelta) > 0 || chainConfig.DebugMode() {
		panic(fmt.Sprintf("Unexpected total balance delta %v (expected %v)",
balanceDelta, expectedBalanceDelta))
	} else {
		// This is a real chain and funds were burnt, not minted, so only log
an error and don't panic
		log.Error("Unexpected total balance delta", "delta", balanceDelta,
"expected", expectedBalanceDelta)
	}
```

Figure 3.3: `arbos/block_processor.go#L288-L295`

By executing `selfdestruct(this)`, an attacker could cause the balance of an account to be burned, in which case there would be less ether than expected.

We set the severity of this finding to low because currently, the nodes would print an error and continue the execution.

Exploit Scenario
Eve executes thousands of `selfdestruct(this)` calls at 2:00 a.m. on a well-known holiday. This causes every Arbitrum node to print an error message. The Offchain Labs team is alerted to the issue and has to react quickly to ensure that the error messages can be discarded.

Recommendations
Short term, track the amount of ether burned through `selfdestruct(this)` and account for that amount in delta computations. This will ensure that unexpected decreases in the total amount of ether are reported; it will also prevent attackers from forcing the nodes to spam the system with unexpected error messages. Be mindful of the issues detailed in TOB-NITRO-GETH-3 and TOB-NITRO-ARBOS-7 when implementing this fix.

Long term, develop comprehensive centralized documentation on the system's invariants and thoroughly test those invariants.

4. Issues in the Go and Rust compilers' WASM back ends	
Severity: **Undetermined**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-NITRO-ARBOS-4
Target: ArbOS, `go-ethereum`	

Description

There are a number of issues and regressions in recent releases of the Go and Rust compilers that affect their WASM back ends.

The ArbOS and geth codebases are cross-compiled to WASM so that the code can be executed and validated during challenges. This compilation is performed by the official Go and Rust compilers. However, the WASM back end of each compiler is less battle tested than the rest of the compiler and contains a number of issues that can directly affect the compiled code.

The issue tracker for the Go compiler lists the following WASM issues:

- Using await before go.run(inst) of wasm_exec await indefinitely (`golang/go#49710`)
- wasm: large memory usage with hard-coded map/array (`golang/go#42979`)

The issue tracker for the Rust compiler also lists several WASM issues:

- Emitted memset and memcpy are really slow on WASM (`rust-lang/rust#92436`)
- Broken WASM codegen with u128 and wasm_abi (`rust-lang/rust#88207`)
- WASM float to int performance regression since 1.53.0 (`rust-lang/rust#87643`)
- Compiling to WASM with emscripten, parse exception: attempted pop from empty stack / beyond block start boundary (`rust-lang/rust#91628`)

Recommendations

Short term, review all known Go and Rust compiler issues to ensure that they do not affect the correctness or performance of the compiled code.

Long term, monitor the development and adoption of the compilers' WASM back ends to assess their maturity.

5. Risk of a node crash due to unchecked ticket-opening errors	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-5
Target: `block_processor.go`	

Description

ArbOS does not check for errors when handling `RedeemScheduledEventID` events. This means that an attacker could crash the ArbOS nodes by crafting a transaction that will cause an error.

Once a transaction has been executed, ArbOS (via the `ProduceBlock` function) goes over the transaction's receipts and calls `OpenRetryable` to open any retryable tickets associated with `RedeemScheduledEventID` events:

```
retryable, _ := state.RetryableState().OpenRetryable(event.TicketId, time)
redeem, _ := retryable.MakeTx(
        chainConfig.ChainID,
        event.SequenceNum,
        gasPrice,
        event.DonatedGas,
        event.TicketId,
        event.GasDonor,
)
redeems = append(redeems, types.NewTx(redeem))
```

Figure 5.1: `block_processor.go#L236-L245`

If `OpenRetryable` attempts to open a nonexistent ticket and experiences an error, it will return a null pointer for `retryable`. Because ArbOS does not check the errors returned by `OpenRetryable`, ArbOS will attempt to access the null pointer and will then crash.

There are multiple situations that could render an existing ticket nonexistent; these include calls to `redeem` and `cancel` on the same ticket ID within the same transaction and calls to `redeem` from within a ticket's execution.

Exploit Scenario

Eve calls `redeem` and `cancel` on the same ticket ID in a single transaction. As a result, `OpenRetryable` returns an error, causing ArbOS to crash.

Recommendations

Short term, take the following steps:

- Implement a check of the errors returned by `OpenRetryable` in `ProduceBlock`. That way, an error in the redemption of a ticket will not cause ArbOS to access a null pointer.
- Consider disallowing calls to `redeem` and `cancel` on the same ticket within the same transaction, as well as calls to `redeem` from within a ticket's execution. These corner-case operations are error-prone and unlikely to be executed by anyone other than malicious actors.

Long term, create a state-machine representation of the retryable ticket mechanism and document its invariants and the expectations surrounding state changes. Implement testing and fuzzing to ensure that these invariants hold.

6. Unspecified gas rules for precompiled Nitro contracts

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-NITRO-ARBOS-6
Target: `precompiles/ArbSys.go`	

Description

Several precompiled Nitro contracts lack clear gas rules. As a result, users may be able to execute costly operations without paying the full gas cost.

Standard precompiled contracts use `RequiredGas` to accurately compute gas costs:

```
func RunPrecompiledContract(p PrecompiledContract, input []byte, suppliedGas uint64,
advancedInfo *AdvancedPrecompileCall) (ret []byte, remainingGas uint64, err error) {
	advanced, isAdvanced := p.(AdvancedPrecompile)
	if isAdvanced {
		return advanced.RunAdvanced(input, suppliedGas, advancedInfo)
	}

	gasCost := p.RequiredGas(input)
	if suppliedGas < gasCost {
		return nil, 0, ErrOutOfGas
	}
	suppliedGas -= gasCost
	output, err := p.Run(input)
	return output, suppliedGas, err
}
```

Figure 6.1: `core/vm/contracts.go#L166-L179`

Several precompiled Nitro contracts (`RunAdvanced`) do not implement `RequiredGas` and instead use ad hoc gas computations. Because there are no specified gas costs for the execution of several precompiled Nitro contracts, users may be able to execute costly operations without paying the full gas cost.

For example, the `ArbSys.sendTxToL1` function executes `Keccak256Hash` on a user-controlled dynamic array, `calldataForL1`, but does not impose a gas charge based on the array's length:

```
// Sends a transaction to L1, adding it to the outbox
func (con *ArbSys) SendTxToL1(c ctx, evm mech, value huge, destination addr,
calldataForL1 []byte) (huge, error) {
        sendHash := crypto.Keccak256Hash(c.caller.Bytes(),
common.BigToHash(value).Bytes(), destination.Bytes(), calldataForL1)
```

Figure 6.2: The header of the `SendTxToL1` function in `ArbSys`

Exploit Scenario

Eve executes thousands of `SendTxToL1` calls, forcing the Arbitrum nodes to consume a large amount of resources—and paying only a fraction of the required gas to execute the calls.

Recommendations

Short term, evaluate and document the gas cost of executing each precompiled contract. This will help prevent a denial-of-service scenario caused by resource exhaustion.

Long term, consider implementing a `RequiredGas`-like function in the precompiled Nitro contracts to separate the calculation of gas costs from the execution logic. This will simplify the gas-related code and the architecture.

7. Fragile expectedBalanceDelta bookkeeping

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-7
Target: `block_processor.go`	

Description

The correctness of ArbOS's `expectedBalanceDelta` bookkeeping relies heavily on an assumption that certain corner cases and error paths are unreachable. This behavior is error-prone, and there may also be other cases that could cause the `expectedBalanceDelta` value to be computed incorrectly.

These corner cases include the execution of an `ArbitrumDepositTx` transaction not generated by the `arbAddress` and the reversion of an `ArbitrumDepositTx` transaction.

Specifically, when handling an `ArbitrumDepositTx` transaction, ArbOS checks that the transaction was generated by the `arbAddress` before adding the amount of the deposit to the balance:

```
case *types.ArbitrumDepositTx:
	if p.msg.From() != arbAddress {
		return false, 0, errors.New("deposit not from arbAddress"), nil
	}
	p.evm.StateDB.AddBalance(*p.msg.To(), p.msg.Value())
	return true, 0, nil, nil
```

Figure 7.1: `tx_processor.go#L90-L95`

However, it does not perform this check before incrementing the `expectedBalanceDelta` value; instead, it adds the deposit amount to that value regardless of whether the transaction was generated by the `arbAddress`:

```
		switch txInner := tx.GetInner().(type) {
		case *types.ArbitrumDepositTx:
			// L1->L2 deposits add eth to the system
			expectedBalanceDelta.Add(expectedBalanceDelta, txInner.Value)
```

Figure 7.2: `block_processor.go#L207-L210`

Similarly, ArbOS assumes that `ArbitrumDepositTx` transactions will never revert. (See figure 7.1.) If an `ArbitrumDepositTx` transaction did revert, the `expectedBalanceDelta` value would be incorrect.

We did not find any ways to execute these corner cases; however, the current way of handling `expectedBalanceDelta` bookkeeping is inherently risky, and there may be other corner cases that could also break the bookkeeping.

Recommendations

Short term, improve the process of updating the `expectedBalanceDelta` value upon an `ArbitrumDepositTx` transaction; to do this, implement a check to ensure that the transaction was generated by the `arbAddress` and did not revert. This will make the bookkeeping functionality more robust against ArbOS errors.

Long term, develop comprehensive centralized documentation on the system's invariants and thoroughly test those invariants.

8. Processing of malformed retryable ticket messages can cause an ArbOS crash

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-8
Target: `arbos/incomingmessage.go`	

Description

An attacker could cause a node crash by submitting a retryable ticket message that forces ArbOS to allocate a large amount of memory.

ArbOS receives on-chain data in the form of messages. There are a number of message types, including one for retryable ticket messages.

```
func (msg *L1IncomingMessage) ParseL2Transactions(chainId *big.Int)
(types.Transactions, error) {
        …
        switch msg.Header.Kind {
        case L1MessageType_L2Message:
                …
        case L1MessageType_EndOfBlock:
                …
        case L1MessageType_L2FundedByL1:
                …
        case L1MessageType_SubmitRetryable:
                tx, err := parseSubmitRetryableMessage(bytes.NewReader(msg.L2msg),
msg.Header, chainId)
                if err != nil {
                        return nil, err
                }
                    return types.Transactions{tx}, nil
        …
```

Figure 8.1: Part of the `ParseL2Transactions` function

Retryable ticket messages contain fields that are extracted directly from the raw data:

```
func parseSubmitRetryableMessage(rd io.Reader, header *L1IncomingMessageHeader,
chainId *big.Int) (*types.Transaction, error) {
        destAddr, err := util.AddressFrom256FromReader(rd)
        if err != nil {
                return nil, err
        }
        pDestAddr := &destAddr
        if destAddr == (common.Address{}) {
                pDestAddr = nil
        }
        callvalue, err := util.HashFromReader(rd)
        if err != nil {
                return nil, err
        }
        depositValue, err := util.HashFromReader(rd)
        if err != nil {
                return nil, err
        }
        …
        dataLengthBig := dataLength256.Big()
        if !dataLengthBig.IsUint64() {
                return nil, errors.New("data length field too large")
        }
        dataLength := dataLengthBig.Uint64()
        data := make([]byte, dataLength)
        …
```

Figure 8.2: Part of the `ParseL2Transactions` function

However, the `dataLength` field is not properly validated, and a large enough `dataLength` value could cause a panic.

```
panic: runtime error: makeslice: len out of range

goroutine 17 [running, locked to thread]:
github.com/offchainlabs/arbstate/arbos.parseSubmitRetryableMessage({0xd763a0,
0xc0001ad710}, 0xc0001bb400, 0xc0001a3ac0)
       arbos/incomingmessage.go:445 +0x830
```

```
github.com/offchainlabs/arbstate/arbos.(*L1IncomingMessage).ParseL2Transactions(0xc0
001a3aa0, 0xc0001a3ac0)
       arbos/incomingmessage.go:200 +0x857
github.com/offchainlabs/arbstate/cmd/fuzz.Fuzz({0x377fdf0, 0x22b, 0x22b})
       cmd/fuzz/fuzz.go:17 +0x152
main.LLVMFuzzerTestOneInput(0x0, 0xc000000601)
       cmd/fuzz/go.fuzz.main/main.go:35 +0x47
==15936== ERROR: libFuzzer: deadly signal
    #0 0x4adfc0 in __sanitizer_print_stack_trace (cmd/fuzz/fuzz.libfuzzer+0x4adfc0)
    #1 0x45a2c8 in fuzzer::PrintStackTrace() (cmd/fuzz/fuzz.libfuzzer+0x45a2c8)
    #2 0x43f413 in fuzzer::Fuzzer::CrashCallback()
(cmd/fuzz/fuzz.libfuzzer+0x43f413)
    #3 0x7fa1e20ed97f  (/lib/x86_64-linux-gnu/libpthread.so.0+0x1297f)
    #4 0x515680 in runtime.raise.abi0 runtime/sys_linux_amd64.s:164
```

Figure 8.3: ArbOS panics when trying to parse a malformed message

We set the severity of this finding to informational because the smart contract currently validates the message fields.

Recommendations

Short term, implement proper validation of the `dataLength` field to prevent panics.

Long term, execute extensive fuzz testing to catch panics and unhandled exceptions in the Go code.

9. Invalid transactions count toward the block gas limit

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-9
Target: `arbos/block_processor.go`	

Description

Although an invalid transaction will not be included in a block, it will still decrease the amount of gas available in the block being processed.

The `ProduceBlockAdvanced` function uses `gasLeft` to track the gas limit per block on L2:

```
gasLeft, _ := state.L2PricingState().PerBlockGasLimit()
```

Figure 9.1: `arbos/block_processor.go#L146`

The amount of gas that will remain in a block upon a transaction's completion (`gasLeft`) is computed before the execution of the transaction:

```
computeGas := tx.Gas() - dataGas

if computeGas > gasLeft && isUserTx && userTxsCompleted > 0 {
    return nil, nil, core.ErrGasLimitReached
}


[..]

gasLeft -= computeGas

receipt, result, err := core.ApplyTransaction(
    chainConfig,
    chainContext,
    &header.Coinbase,
    &gasPool,
    statedb,
    header,
    tx,
    &header.GasUsed,
    vm.Config{},
)
if err != nil {
    // Ignore this transaction if it's invalid under the state transition function
    statedb.RevertToSnapshot(snap)
    return nil, nil, err
}
```

Figure 9.2: `arbos/block_processor.go#L235-L261`

If the transaction is invalid, it will not be included in the block. However, the gas cost of the transaction will still count toward the block gas limit, causing the block to be filled more quickly than it should be.

Exploit Scenario

Eve submits thousands of invalid transactions. By submitting these transactions, Eve slows the generation of Arbitrum blocks—and pays only the L1 costs to do so.

Recommendations

Short term, decrease the `gasLeft` value only after a transaction has been executed in `ProduceBlockAdvanced`. That way, transactions will not decrease the amount of gas available in a block unless they are included in the block.

Long term, document the L2 gas rules and the modifications made to the geth gas metrics, and ensure that the test suite covers any related corner cases.

10. Confusing rule for calculating the amount of gas remaining in a block

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-10
Target: `block_processor.go`	

Description

ArbOS does not include the poster's gas cost when calculating the amount of gas left in a block. This nonstandard gas rule has no direct effect on the calculation's outcome, as ArbOS also implements the original geth rule.

ArbOS implements a custom rule for calculating the amount of gas left in a block: instead of including `tx.Gas()` in its calculation, it includes `tx.Gas()` subtracted from the poster's gas cost.

```
if gasPrice.Sign() > 0 {
    dataGas = math.MaxUint64
    pricing := state.L1PricingState()
    posterCost, _ := pricing.PosterDataCost(sender, aggregator, tx.Data())
    posterCostInL2Gas := new(big.Int).Div(posterCost, gasPrice)
    if posterCostInL2Gas.IsUint64() {
        dataGas = posterCostInL2Gas.Uint64()
    } else {
        log.Error("Could not get poster cost in L2 terms", posterCost, gasPrice)
    }
}

if dataGas > tx.Gas() {
    // this txn is going to be rejected later
    if hooks.RequireDataGas {
        return nil, nil, core.ErrIntrinsicGas
    }
    dataGas = 0
}

computeGas := tx.Gas() - dataGas

if computeGas > gasLeft && isUserTx && userTxsCompleted > 0 {
    return nil, nil, core.ErrGasLimitReached
}
```

Figure 10.1: `block_processor.go#L215-L219`

However, ArbOS also executes geth's `preCheck` function:

```
func (st *StateTransition) transitionDbImpl() (*ExecutionResult, error) {
	[...]

	// Check clauses 1-3, buy gas if everything is correct
	if err := st.preCheck(); err != nil {
		return nil, err
	}
```

Figure 10.2: `core/state_transition.go#L286-L305`

This function executes buyGas:

```
func (st *StateTransition) preCheck() error {
	[...]
	return st.buyGas()
```

Figure 10.3: `core/state_transition.go#L225-L270`

The buyGas function subtracts the gas cost of a transaction from the amount of gas left in the block and returns an error if the block is left without any gas:

```
func (st *StateTransition) buyGas() error {
	[...]
	if err := st.gp.SubGas(st.msg.Gas()); err != nil {
		return err
	}
```

Figure 10.4: `core/state_transition.go#L203-L217`

Thus, while ArbOS does not include the poster's gas cost in the operation, the omission has no effect on the outcome.

Recommendations

Short term, clarify how the poster's gas cost should be handled in regard to the block gas limit and create a standard rule for it.

Long term, document the L2 gas rules and the modifications made to the geth gas metrics, and ensure that the test suite covers any related corner cases.

11. Use of an unofficial Brotli library for message compression

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-11
Target: `incomingmessage.go`, `andybalholm/brotli`	

Description

The parsing of compressed incoming messages for ArbOS uses a Brotli library that is not the official one and it lacks thorough testing.

Certain ArbOS messages could be compressed using Brotli:

```
func parseL2Message(rd io.Reader, poster common.Address, requestId common.Hash,
depth int) (types.Transactions, error) {
…
case L2MessageKind_BrotliCompressed:
   if depth > 0 { // ignore compressed messages if not top level
         return nil, errors.New("can only compress top level batch")
   }
   reader := io.LimitReader(brotli.NewReader(rd), MaxL2MessageSize)
   return parseL2Message(reader, poster, requestId, depth+1)
…
```

Figure 11.1: `incomingmessage.go#L294-L299`

ArbOS uses an unofficial library instead of the Google recommended one, which has been extensively tested through Google's `oss-fuzz` tool. By contrast, the unofficial library was converted semi-automatically from C code via the c2go tool and has not been extensively tested.

Moreover, when we ran a fuzzing campaign on the Brotli library, our fuzzer was unable to reach several parts of the codebase. It is unclear whether these parts of the codebase are unreachable or whether they could be reached with additional exploration.

```
func copyUncompressedBlockToOutput(available_out *uint, next_out *[]byte, total_out
*uint, s *Reader) int {
        /* TODO: avoid allocation for single uncompressed block. */
        if !ensureRingBuffer(s) {
                return decoderErrorAllocRingBuffer1
        }
…
```

Figure 11.2: The code not covered in our fuzzing of `brotli/decode.go`

Exploit Scenario
Alice, the maintainer of the unofficial Brotli library used by ArbOS, decides to disown the project. Eve takes over its maintenance and creates a new release. She claims that the release contains a critical fix but has actually added a backdoor to the code. When the Arbitrum team sees the new release, it updates the ArbOS code to use the new version of the library.

Recommendations
Short term, consider using Google's official library for Brotli compression or avoiding the use of Brotli compression altogether to reduce ArbOS's attack surface.

Long term, review all external dependencies to ensure that they are well maintained and tested.

12. Escrow addresses can be used to move stolen funds

Severity: **Low**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-NITRO-ARBOS-12
Target: `retryable.go`	

Description

ArbOS's handling of escrowed funds could enable an attacker to obfuscate a movement of stolen ETH.

ArbOS defines special addresses to hold retryable ticket funds in escrow:

```
func RetryableEscrowAddress(ticketId common.Hash) common.Address {
        return common.BytesToAddress(crypto.Keccak256([]byte("retryable escrow"),
ticketId.Bytes()))
}
```

Figure 12.1: `retryable/retryable.go#L324-L326`

When a retryable ticket is executed or canceled, the balance of the escrow address associated with the ticket is transferred to the beneficiary's address:

```
func OpenRetryableState(sto *storage.Storage, statedb vm.StateDB) *RetryableState {
        payFromEscrow := func(ticketId common.Hash, destination common.Address) {
                escrowAddress := RetryableEscrowAddress(ticketId)
                arbos_util.TransferEverything(escrowAddress, destination, statedb)
        }
        …
}
func (rs *RetryableState) DeleteRetryable(id common.Hash) (bool, error) {
        retStorage := rs.retryables.OpenSubStorage(id.Bytes())
        timeout, err := retStorage.GetByUint64(timeoutOffset)
        if timeout == (common.Hash{}) || err != nil {
                return false, err
        }
        // move any funds in escrow to the beneficiary (should be none if the retry
succeeded -- see EndTxHook)
       beneficiary, _ := retStorage.GetByUint64(beneficiaryOffset)
        rs.payFromEscrow(id, common.BytesToAddress(beneficiary[:]))
        …
 }
```

Figure 12.2: The `OpenRetryableState` and `DeleteRetryable` functions in `retryable/retryable.go`

Although full nodes can replay every ArbOS operation, it is difficult for them to identify whether an ETH balance movement corresponds to a ticket refund. This is because the balance movement will not be directly translated into an EVM transaction (and thus reflected in an off-chain event) and will instead be triggered when the retryable ticket is deleted. This can be very useful to attackers seeking to move stolen funds without leaving an obvious trace.

Exploit Scenario

Eve hacks a contract and obtains a large amount of ETH in the Arbitrum system. Eve then creates a retryable ticket and transfers the stolen funds to a corresponding escrow address. Eventually, Eve's retryable ticket expires and is ready to be refunded. Alice, a user interacting with the retryable ticket system, triggers the deletion of Eve's expired ticket. The stolen funds then disappear from the escrow address and appear in Eve's wallet.

Recommendations

Short term, consider emitting an event for retryable ticket refunds, including refunds executed in the context of other operations (such as transactions from one of ArbOS's special addresses).

Long term, closely monitor the ways in which users interact with the special ArbOS addresses (e.g., escrow and gas network wallet addresses) to ensure that they are not being used in unintended ways.

13. Confusing EOA address remapping rules

Severity: **Medium**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-NITRO-ARBOS-13
Target: ArbOS	

Description
The confusing nature of the EOA remapping rules could cause users to interact with an L2 contract incorrectly.

To prevent the use of a smart contract to send an L2 transaction (i.e., the issue detailed in finding TOB-ARB-BRIDGE-013 of the September 10, 2021 Arbitrum security assessment), ArbOS now remaps the `from` address of any transaction originating from an L1 contract. The Arbitrum team also implemented the following rules for the remapping of EOA addresses:

- If the transaction sent by an EOA is signed, the EOA's address is not remapped.
- If the transaction is a retryable ticket generated through a call to `depositEth`, the address is not remapped.
- In all other situations, the address is remapped.

The divergence from the remapping rule for L1 contracts is not well documented and could result in user confusion and the use of an incorrect `from` address in an L2 transaction.

Exploit Scenario
Bob submits a signed transaction to take out a collateralized loan on L2. The value of the collateral then decreases, so Bob needs to deposit additional funds to prevent the liquidation of his loan. To speed up the process, Bob calls the `createRetryableTicket` function directly rather than using the sequencer to transfer the funds. Instead of increasing Bob's collateral, the call causes Bob's address to be remapped and a new position to be created. The value of Bob's collateral continues to decrease, and Bob's loan is liquidated.

Recommendations
Short term, implement a single remapping rule for EOA addresses or clearly document the EOA remapping rules. Currently, the rules are error-prone and undocumented.

Long term, review all corner cases that could occur in the handling of cross-chain transactions and ensure that users are aware of them.

14. Unreachable mechanism for disabling the default aggregator

Severity: **Informational**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-NITRO-ARBOS-14
Target: `arbos/block_processor.go`	

Description

ArbOS users are not required to use the default aggregator. However, the code involved in disabling the default aggregator is not reachable and is therefore dead code.

When checking whether it needs to reimburse an aggregator for a transaction, ArbOS checks whether the transaction's sender has disabled the default aggregator:

```
// Get the aggregator who is eligible to be reimbursed for L1 costs of txs from
sender, or nil if there is none.
func (ps *L1PricingState) ReimbursableAggregatorForSender(sender common.Address)
(*common.Address, error) {
	fromTable, err := ps.UserSpecifiedAggregator(sender)
	if err != nil {
		return nil, err
	}
	if fromTable != nil {
		return fromTable, nil
	}

	refuses, err := ps.RefusesDefaultAggregator(sender)
	if err != nil || refuses {
		return nil, err
	}
```

Figure 14.1: `arbos/l1pricing/l1pricing.go#L133-L146`

Disabling the default aggregator requires a call to the `SetRefusesDefaultAggregator` function:

```
func (ps *L1PricingState) SetRefusesDefaultAggregator(addr common.Address, refuses
bool) error {
	val := uint64(0)
	if refuses {
		val = 1
	}
	return ps.refuseDefaultAggregator.Set(common.BytesToHash(addr.Bytes()),
common.BigToHash(new(big.Int).SetUint64(val)))
}
```

Figure 14.1: `arbos/l1pricing/l1pricing.go#L133-L146`

However, this function is never called. Thus, the relevant code is dead code, and the mechanism for disabling the default aggregator is not implemented.

Recommendations

Short term, either remove the unreachable code related to the disabling of the default aggregator, or implement the `SetRefusesDefaultAggregator` function in a precompiled contract. Because this function is currently unreachable, it is unclear whether code is missing from the codebase or whether the disabling option was meant to be removed.

Long term, improve the documentation on the use of aggregators. Specifically, explain the extent of their control over users' funds and actions and ensure that users are aware of the inherent risks of using an aggregator.

15. Risk of a node crash due to unchecked PosterDataCost errors

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-15
Target: `block_processor.go`, `tx_processor.go`	

Description

ArbOS invokes the `PosterDataCost` function to compute the cost of each transaction. However, it does not check the errors returned by this function. This omission could enable an attacker to crash the ArbOS nodes by deliberately causing an error.

The `PosterDataCost` function uses the addresses of a transaction's sender and aggregator and the data of the actual transaction to compute the transaction's cost.

```
if gasPrice.Sign() > 0 {
    dataGas = math.MaxUint64
    pricing := state.L1PricingState()
    posterCost, _ := pricing.PosterDataCost(sender, aggregator, tx.Data())
    posterCostInL2Gas := new(big.Int).Div(posterCost, gasPrice)
```

Figure 15.1: `block_processor.go#L215-L219`

However, various situations could cause `PosterDataCost` to experience an error and return a null pointer. These include the disabling of the default aggregator (TOB-NITRO-ARBOS-14).

```
func (ps *L1PricingState) PosterDataCost(
       sender common.Address,
       aggregator *common.Address,
       data []byte,
) (*big.Int, error) {
       …
       reimbursableAggregator, err := ps.ReimbursableAggregatorForSender(sender)
       if err != nil {
              return nil, err
       }
       …

       bytesToCharge := uint64(len(data) + TxFixedCost)

       ratio, err := ps.AggregatorCompressionRatio(*reimbursableAggregator)
       if err != nil {
              return nil, err
       }
```

```
        dataGas := 16 * bytesToCharge * ratio / DataWasNotCompressed

        // add 5% to protect the aggregator from bad price fluctuation luck
        dataGas = dataGas * 21 / 20

        price, err := ps.L1GasPriceEstimateWei()
        if err != nil {
               return nil, err
        }

        baseCharge, err := ps.FixedChargeForAggregatorWei(*reimbursableAggregator)
        if err != nil {
               return nil, err
        }

        …
}
```

Figure 15.2: Potential error conditions in `PosterDataCost`

Because ArbOS does not check the errors returned by `PosterDataCost`, it may attempt to access a null pointer, in which case ArbOS will crash.

Similarly, if `PosterDataCost` experiences an error, the `tx_processor.GasChargingHook` function will log the error but may still return a null pointer for `posterCost`.

Exploit Scenario

Alice posts a transaction that causes `PosterDataCost` to return an error. ArbOS discards the error and accesses a null pointer, which causes it to crash.

Recommendations

Short term, have each one of `PosterDataCost`'s callers check for errors returned by the function. This will prevent an error in a gas cost computation from causing ArbOS to access a null pointer.

Long term, review all of the functions that can return errors and ensure that their callers properly handle those errors instead of dereferencing a null pointer.

16. Infinite loop caused by parsing of malformed sequencer messages

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-16
Target: `inbox.go`	

Description

The lack of data validation in the parsing of messages from the sequencer can cause ArbOS to enter a loop, preventing the production of new blocks.

ArbOS parses the messages it receives from the sequencer without checking whether they are malformed:

```
// This does *not* return parse errors, those are transformed into invalid messages
func (r *inboxMultiplexer) Pop() (*MessageWithMetadata, error) {
	if r.cachedSequencerMessage == nil {
		bytes, realErr := r.backend.PeekSequencerInbox()
		if realErr != nil {
			return nil, realErr
		}
		r.cachedSequencerMessageNum = r.backend.GetSequencerInboxPosition()
		r.cachedSequencerMessage = parseSequencerMessage(bytes)
	}
	msg, err := r.getNextMsg()
	// advance even if there was an error
```

Figure 16.1: `inbox.go#L148-L159`

If `getNextMsg` fails to decode an unsigned integer encoded as a Recursive Length Prefix (RLP) string, it will not increase the segment number, which is necessary for the parsing to continue.

```
// Returns a message, the segment number that had this message, and real/backend
errors
// parsing errors will be reported to log, return nil msg and nil error
func (r *inboxMultiplexer) getNextMsg() (*MessageWithMetadata, error) {
	…
	for {
		if segmentNum >= uint64(len(seqMsg.segments)) {
			break
		}
		segment = seqMsg.segments[int(segmentNum)]
		if len(segment) == 0 {
```

```
                    segmentNum++
                    continue
            }
            segmentKind := segment[0]
            if segmentKind == BatchSegmentKindAdvanceTimestamp || segmentKind ==
BatchSegmentKindAdvanceL1BlockNumber {
                    rd := bytes.NewReader(segment[1:])
                    advancing, err := rlp.NewStream(rd, 16).Uint()
                    if err != nil {
                            log.Warn("error parsing sequencer advancing segment",
"err", err)
                            continue
                    }
```

Figure 16.2: Part of the `getNextMsg` function

Thus, if the sequencer sends a malformed message, ArbOS will become stuck in a loop, and no new blocks will be produced.

Exploit Scenario

The sequencer posts a malformed message, which causes ArbOS to enter an infinite loop and ultimately crashes the validators.

Recommendations

Short term, ensure that `segmentNum` is incremented even if the message being parsed is invalid.

Long term, use a fuzzer to ensure that an invalid or incomplete message will not disrupt ArbOS's expected behavior.

17. ArbOS bottleneck caused by RLP decoding loop

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-17
Target: `inbox.go`	

Description

By posting a large enough message, the sequencer could force ArbOS to complete numerous iterations of the message-decoding loop, degrading the quality of its service.

ArbOS parses the messages it receives from the sequencer without checking whether they are malformed:

```
// This does *not* return parse errors, those are transformed into invalid messages
func (r *inboxMultiplexer) Pop() (*MessageWithMetadata, error) {
	if r.cachedSequencerMessage == nil {
		bytes, realErr := r.backend.PeekSequencerInbox()
		if realErr != nil {
			return nil, realErr
		}
		r.cachedSequencerMessageNum = r.backend.GetSequencerInboxPosition()
		r.cachedSequencerMessage = parseSequencerMessage(bytes)
	}
	msg, err := r.getNextMsg()
	// advance even if there was an error
```

Figure 17.1: `inbox.go#L148-L159`

The `parseSequencerMessage` function decompresses the messages and, using a `for` loop, decodes them into a list of RLP bytes:

```
func parseSequencerMessage(data []byte) *sequencerMessage {
	…
	if len(data) >= 41 {
		if data[40] == 0 {
			reader :=
io.LimitReader(brotli.NewReader(bytes.NewReader(data[41:])), maxDecompressedLen)
			stream := rlp.NewStream(reader, uint64(maxDecompressedLen))
			for {
				var segment []byte
				err := stream.Decode(&segment)
				if err != nil {
					if !errors.Is(err, io.EOF) && !errors.Is(err,
io.ErrUnexpectedEOF) {
                                        log.Warn("error parsing sequencer message
segment", "err", err.Error())
                                 }
                                 break
                          }
                          segments = append(segments, segment)
                    }
```

Figure 17.2: Part of the `getNextMsg` function

However, this loop can contain numerous items, and the sequencer could add empty RLP segments (each consisting of only one byte) to increase that number and cause a slowdown. There is a limit on the size of the decompressed data (16 MB, or 1024 * 1024 * 16 bytes), and it could be very expensive to post an on-chain message that would cause the function to reach that limit. However, the use of Brotli compression could enable the sequencer to produce very small files (of around 15 bytes), making the attack very cheap.

Exploit Scenario

The sequencer posts a compressed message with numerous empty segments, causing ArbOS to experience a severe slowdown and degrading the quality of its service.

Recommendations

Short term, consider limiting the number of segments that the sequencer can send.

Long term, use a fuzzer to ensure that an invalid or incomplete message will not disrupt ArbOS's expected behavior.

18. **Broken gasLeft computation**	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-18
Target: `arbos/block_processor.go`	

Description

When calculating a transaction's effect on the `gasLeft` value, `ProduceBlockAdvanced` twice subtracts the amount of gas used from that value. As a result, blocks are filled more quickly than expected.

The `ProduceBlockAdvanced` function uses `gasLeft` to track the gas limit per block on L2:

```
gasLeft, _ := state.L2PricingState().PerBlockGasLimit()
```

Figure 18.1: `arbos/block_processor.go#L146`

The `gasLeft` value is calculated twice with each transaction: the first calculation uses the total cost incurred to complete the transaction, and the second, the amount of gas used by the transaction.

```
computeGas := tx.Gas() - dataGas

if computeGas > gasLeft && isUserTx && userTxsCompleted > 0 {
    return nil, nil, core.ErrGasLimitReached
}


[..]

gasLeft -= computeGas

[..]

gasLeft -= gasUsed - dataGas
```

Figure 18.2: `arbos/block_processor.go#L235-L320`

While twice counting the amount of gas used by a transaction would be problematic, the use of the entire transaction gas cost in the first operation causes L2 blocks to be filled even more quickly.

Exploit Scenario

The amount of gas left in an L2 block decreases twice with every transaction that is executed. As a result, the Arbitrum system becomes congested, and users have to pay higher-than-expected fees.

Recommendations

Short term, have `ProduceBlockAdvanced` decrease the `gasLeft` value only once per transaction. This will prevent L2 blocks from being filled twice as fast as they should be.

Long term, document the L2 gas rules and the modifications made to the geth gas metrics and ensure that the test suite covers any related corner cases.

19. Aggregators can block user transactions by setting a high fixed fee

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-19
Target: `arbos/block_processor.go`	

Description

The L1 fees paid to an aggregator include a fixed fee that can be set by the aggregator. Because there is no limit on this fixed fee, a malicious aggregator could set a high fee to prevent transactions from being included in a block.

The aggregator's fixed fee is added to the dynamic fee, which is based on the transaction input's length:

```
	baseCharge, err := ps.FixedChargeForAggregatorWei(*reimbursableAggregator)
	if err != nil {
		return nil, err
	}

	chargeForBytes := new(big.Int).Mul(big.NewInt(int64(dataGas)), price)
	return new(big.Int).Add(baseCharge, chargeForBytes), nil
```

Figure 19.1: `arbos/l1pricing/l1pricing.go#L252-L258`

The fixed fee can be set to an arbitrary value by the aggregator (or the system's owner):

```
// Sets an aggregator's fixed fee (caller must be the aggregator, its fee collector,
or an owner)
func (con ArbAggregator) SetTxBaseFee(c ctx, evm mech, aggregator addr, feeInL1Gas
huge) error {
	allowed, err := accountIsAggregatorOrCollectorOrOwner(c.caller, aggregator,
c.state)
	if err != nil {
		return err
	}
	if !allowed {
```

```
            return errors.New("Only an aggregator (or its fee collector / chain
owner) may change its fee collector")
       }
       return c.state.L1PricingState().SetFixedChargeForAggregatorL1Gas(aggregator,
feeInL1Gas)
}
```

Figure 19.2: `precompiles/ArbAggregator.go#L108-L118`

Although there is no limit on this fee, the total L1 cost of a transaction (the sum of the fixed and dynamic fees) is capped at 2**64:

```
posterCostInL2Gas := new(big.Int).Div(posterCost, gasPrice) // the cost as if it
were an amount of gas

if !posterCostInL2Gas.IsUint64() {

       posterCostInL2Gas = new(big.Int).SetUint64(math.MaxUint64)

}
```

Figure 19.3: `arbos/tx_processor.go#L247-L250`

An aggregator could thus prevent users from sending L2 transactions by setting a high fixed fee. To switch to another aggregator, users would need to pay a transaction cost of ~2**64 wei (~USD 40,000–50,000 at the current ether price).

Exploit Scenario

Over time, the owner drops its capacity to control the fixed fee of an aggregator. Eve changes the fixed fee of her aggregator to 2**128. As a result, to continue using the Arbitrum system, the aggregator's users are forced to pay the high fee or to pay USD 50,000 to change their aggregator.

Recommendations

Short term, implement a cap on fixed aggregator fees. This will prevent malicious aggregators from setting high fees.

Long term, document the L2 gas rules, the modifications made to the geth gas metrics, and the risks associated with using an aggregator. Additionally, ensure that the test suite covers any related corner cases.

20. Aggregators can steal each other's tips

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-20
Target: `arbos/block_processor.go`	

Description

As of EIP-1559, every transaction can include a tip. However, because all Arbitrum tips are sent to `block.coinbase`, one aggregator could steal another's tip by front-running the transaction and including it in a block.

```
	effectiveTip := st.gasPrice
	if london {
		effectiveTip = cmath.BigMin(st.gasTipCap,
new(big.Int).Sub(st.gasFeeCap, st.evm.Context.BaseFee))
	}
	st.state.AddBalance(st.evm.Context.Coinbase,
new(big.Int).Mul(new(big.Int).SetUint64(st.gasUsed()), effectiveTip))
```

Figure 20.1: `go-ethereum/core/state_transition.go#L356-L360`

The Arbitrum documentation includes the following guidance on tips:

```
While tips are not advised for those using the sequencer, which prioritizes
transactions on a first-come first-served basis, 3rd-party aggregators may choose to
order txes based on tips. A user specifies a tip by setting a gas price in excess of
the basefee and will pay that difference on the amount of gas the tx uses.

A poster receives the tip only when the user has set them as their preferred
aggregator. Otherwise the tip goes to the network fee collector. This disincentives
unpreferred aggregators from racing to post txes with large tips.
```

Figure 20.2: `docs/arbos/Gas.md#tips-in-l2`

The claims in the second paragraph are incorrect: in Arbitrum, every tip is sent to the coinbase, even if the associated transaction was included by an aggregator other than the preferred aggregator. This incentivizes aggregators to race to post transactions with large tips.

Exploit Scenario
Alice's aggregator. She posts a transaction with a 0.1 ETH tip. Eve front-runs Alice's transaction and includes it in its own block, stealing Alice's tip.

Recommendations
Short term, investigate ways to prevent the theft of tips. A simple solution would be updating the geth code to send a tip to the user's preferred aggregator or to the coinbase in the absence of a preferred aggregator. However, that would require implementing a change in geth that could be difficult to push upstream.

Long term, document the L2 gas rules, the modifications made to the geth gas metrics, and the risks associated with using an aggregator. Additionally, ensure that the test suite covers any related corner cases.

21. Aggregators can steal extra fees by updating their rates

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-21
Target: `arbos/block_processor.go`	

Description
An aggregator can update its compression ratio or fixed charge at any time. By executing a well-timed update, an aggregator could trick a user into paying a higher-than-expected poster cost.

Every transaction can carry a poster cost, which is based on the aggregator's compression ratio and fixed charge:

```
	baseCharge, err := ps.FixedChargeForAggregatorWei(*reimbursableAggregator)
	if err != nil {
		return nil, err
	}

	chargeForBytes := new(big.Int).Mul(big.NewInt(int64(dataGas)), price)
	return new(big.Int).Add(baseCharge, chargeForBytes), nil
```

Figure 21.1: `arbos/l1pricing/l1pricing.go#L252-L258`

An aggregator can change its ratio at any time by calling `ArbAggregator.SetCompressionRatio` or `ArbAggregator.SetTxBaseFee`. A well-timed change could allow an aggregator to steal user funds (any amount of funds up to the transaction's gas limit).

Exploit Scenario
Alice, who holds 1 ETH, is using Eve's aggregator. She sends a transaction with no gas limit. Eve's aggregator receives Alice's transaction and creates a block with a previous transaction. Eve then updates the aggregator's fixed cost to 1 ETH, which enables her to steal Alice's ether.

Recommendations
Short term, document the risks associated with malicious aggregators and ensure that users set appropriate gas limits.

Long term, document the L2 gas rules, the modifications made to the geth gas metrics, and the risks associated with using an aggregator. Additionally, ensure that the test suite covers any related corner cases.

22. Aggregators can censor the redemption of retryable tickets

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-22
Target: `arbos/block_processor.go`	

Description

The sequencer or an aggregator could cause the redemption of a retryable ticket to be rejected by including the transaction's execution at the end of a block.

The redemption of a retryable ticket triggers the creation of a new transaction, which will be executed after the redemption call:

```
			if len(redeems) > 0 {
				tx = redeems[0]
				redeems = redeems[1:]

				retry, ok := (tx.GetInner()).(*types.ArbitrumRetryTx)
				if !ok {
					panic("retryable tx is somehow not a retryable")
				}
				retryable, _ :=
state.RetryableState().OpenRetryable(retry.TicketId, time)
				if retryable == nil {
					// retryable was already deleted
					continue
				}
```

Figure 22.1: `arbos/block_processor.go#L174-L186`

If the current block has reached its gas limit, the transaction will be discarded:

```
if computeGas > gasLeft && isUserTx && userTxsCompleted > 0 {
	return nil, nil, core.ErrGasLimitReached
 }
```

Figure 22.2: `arbos/block_processor.go#L237-L239`

The sequencer or an aggregator could thus force a transaction to be discarded by putting it at the end of a block.

Exploit Scenario

Eve is a malicious actor who runs an aggregator. Eve wants to prevent the execution of tickets created by Alice, who sends her transactions to Bob's aggregator. Whenever Alice sends a transaction, Eve front-runs all of the transactions submitted by Bob's aggregator and includes Alice's transaction at the end of her block. As a result, Alice's attempts to redeem her tickets always fail.

Recommendations

Short term, document this issue and ensure that users are aware that they can call the bridge directly if an aggregator or the sequencer is censoring their retryable tickets.

Long term, develop centralized documentation highlighting the risks associated with using the sequencer or an aggregator.

23. Fragile retryable ticket ID scheme	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-ARBOS-23
Target: `arbos/block_processor.go`	

Description

The retryable ticket ID scheme would allow for a `RequestId` collision if it were possible for more than one retryable ticket to be submitted in the same block.

The `RequestId` field of a retryable ticket, which indicates the ticket's unique identifier, is set to the block header's `RequestId`:

```
	tx := &types.ArbitrumSubmitRetryableTx{
		ChainId:          chainId,
		RequestId:        header.RequestId,
```

Figure 23.1: `arbos/incomingmessage.go#L454-L456`

The block header's `RequestId` is based on `segmentNum` and `cachedSequencerMessageNum`:

```
		var requestId common.Hash
		// TODO: a consistent request id. Right now we just don't set the
request id when it isn't needed.
		if len(segment) < 2 || (segment[1] != arbos.L2MessageKind_SignedTx &&
segment[1] != arbos.L2MessageKind_UnsignedUserTx) {
			requestId[0] = 1 << 6
			binary.BigEndian.PutUint64(requestId[(32-16):(32-8)],
r.cachedSequencerMessageNum)
			binary.BigEndian.PutUint64(requestId[(32-8):], segmentNum)
		}
```

Figure 23.2: `arbos/incomingmessage.go#L454-L456`

This means that if two tickets were created within the same block, they would have the same `RequestId`.

It is not currently possible for two tickets to be created in the same block. However, if an update to the protocol enabled that option, every ticket in the same block would have the same `RequestId`.

Recommendations

Short term, update the retryable ticket ID scheme such that every ticket has a unique ID. That way, if it becomes possible to create multiple tickets within the same block, the system will maintain its integrity.

Long term, create a state-machine representation of the retryable ticket system and document the invariants related to every state and transition. This should include the ticket-creation process, the ID scheme, calls to the precompiled contracts, and the gas refund functionality.

Summary of Findings: ArbNode

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

ID	Title	Type	Severity
1	Use of time.After() in select statements can lead to memory leaks	Data Validation	Low
2	Broadcast client configuration allows the use of insecure TLS versions	Configuration	Low

Detailed Findings: ArbNode

1. Use of time.After() in select statements can lead to memory leaks	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-ARBNODE-1
Target: ArbNode	

Description

ArbNode's use of the `time.After` function can lead to memory leaks that worsen over time and ultimately cause an out-of-memory error.

ArbNode uses `time.After` in `select` statements within loops:

```
for {
	err := s.createBlocks(ctx)
	if err != nil && !errors.Is(err, context.Canceled) {
		log.Error("error creating blocks", "err", err.Error())
	}
	select {
	case <-ctx.Done():
		return
	case <-s.newMessageNotifier:
	case <-time.After(10 * time.Second):
	}
}
```

Figure 1.1: `arbnode/transaction_streamer.go#L457-L468`

The following parts of the codebase contain that same pattern:

- `arbnode/batch_poster.go#L342-L351`
- `arbnode/delayed_sequencer.go#L172-L181`
- `arbnode/inbox_reader.go#L62-L71`

- `arbnode/inbox_reader.go#L90-L99`
- `arbnode/transaction_streamer.go#L457-L466`
- `arbnode/util.go#L70-L79`
- `arbnode/util.go#156-L165`
- `arbnode/util.go#163-L172`
- `broadcastclient/broadcastclient.go#L2-L81`

The use of `time.After` can cause memory issues because, as its documentation indicates, "The underlying Timer is not recovered by the garbage collector until the timer fires."

If the other switch statements are executed, the `time.After` object will not be directly freed. If the select statements are called frequently, the amount of memory that is used may increase over time, disrupting the node's operation.

Exploit Scenario

Bob is running an Arbitrum node on hardware with a limited amount of memory. The node eventually requires 10 times the amount of RAM that it used upon its deployment and stops working.

Recommendations

Short term, replace the `time.After` function with `time.NewTime`, and use `time.Reset` and `time.Stop` to control the timer's progress. This will prevent memory leaks caused by the use of a timer.

Long term, integrate Semgrep and the `semgrep-go` rules into the CI pipeline to catch potential security issues.

References

- A story of a memory leak in GO: How to properly use time.After()
- Golang <-time.After() is not garbage collected before expiry

2. Broadcast client configuration allows the use of insecure TLS versions	
Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-NITRO-ARBNODE-2
Target: `go-ethereum`	

Description

The broadcast client uses Transport Layer Security (TLS) without setting a minimum TLS version (a `tls.Config.MinVersion` value) in `ws.Dialer`.

```
timeoutDialer := ws.Dialer{
        Timeout: 10 * time.Second,
 }
```

Figure 2.1: `broadcastclient/broadcastclient.go#L95-L97`

This allows the client to use any TLS version—including versions that are considered insecure.

Exploit Scenario

Eve forces Bob's node to connect to the inbox message broadcaster via TLS 1.0. This enables Eve to compromise the encrypted data sent by Bob's node and to disrupt the node's operations.

Recommendations

Short term, use at least version 1.2 of TLS in the broadcast client.

Long term, integrate CodeQL into the CI pipeline to catch potential security issues.

Summary of Findings: Smart Contracts

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

ID	Title	Type	Severity
1	Lack of contract existence check on delegatecall will result in unexpected behavior	Data Validation	High
2	Unreachable overflow checks in currentRequiredStake	Data Validation	Low
3	ERC20Rollup is incompatible with non-standard ERC20s	Data Validation	Informational
4	Integer type inconsistency	Data Validation	Informational
5	Inclusion of dead code	Patching	Informational
6	Unclear expectations surrounding updates to the stake requirements	Data Validation	Medium
7	Process for removing old stakes is not scalable	Denial of Service	Informational
8	Failure to decrement the amount of time remaining in a challenge	Data Validation	High
9	Lack of a lower limit on numSteps in a challenge enables attackers to halt a challenge's progress	Data Validation	High
10	Challenges can move from the EXECUTION state to the BLOCK state	Data Validation	High

Detailed Findings: Smart Contracts

1. Lack of a contract existence check on delegatecall will result in unexpected behavior	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-SC-1
Target: `AdminFallbackProxy`	

Description

The `AdminFallbackProxy` contract uses the `delegatecall` proxy pattern. If the implementation contract is self-destructed, the proxy may not detect failed executions.

A `delegatecall` to a destructed contract will return success as part of the EVM specification. The Solidity documentation includes the following warning:

> *The low-level call, delegatecall and callcode will return success if the called account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.*

Figure 1.1: A snippet of the Solidity documentation detailing unexpected behavior related to `delegatecall`

`AdminFallbackProxy` checks for the implementation contract's existence only when the implementation is set:

```
    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
to user logic
         address target = _getAdmin() != msg.sender
```

```
            ? DoubleLogicERC1967Upgrade._getSecondaryImplementation()
            : ERC1967Upgrade._getImplementation();
        // implementation setters already do an existence check
        // require(target.isContract(), "TARGET_NOT_CONTRACT");
        return target;
    }
```

Figure 1.2: `libraries/AdminFallbackProxy.sol#L136-L151`

If the implementation is destroyed after the contract's deployment, the proxy will not throw an error; instead, it will return success even though no code was executed.

Exploit Scenario

`AdminFallbackProxy`'s implementation contract is destroyed. However, each `delegatecall` returns success without executing any code.

Recommendations

Short term, implement a contract existence check before any `delegatecall`. Document the fact that using `suicide` and `selfdestruct` can lead to unexpected behavior, and prevent future upgrades from introducing these functions.

Long term, carefully review the Solidity documentation, especially the "Warnings" section, as well as the pitfalls of using the `delegatecall` proxy pattern.

2. Unreachable overflow checks in currentRequiredStake

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-SC-2
Target: `RollupUserLogic.sol`	

Description

The Rollup contracts contain unreachable overflow checks. This dead code increases the code's complexity without providing any benefit and can increase the likelihood of mistakes.

The Rollup contracts require that validators deposit a certain amount of assets to place stakes (and thus secure the Arbitrum network). The computation of this amount is performed in `currentRequiredStake`:

```
    function currentRequiredStake(
        uint256 _blockNumber,
        uint64 _firstUnresolvedNodeNum,
        uint256 _latestCreatedNode
    ) internal view returns (uint256) {
        …
        uint256 firstUnresolvedAge = _blockNumber - firstUnresolvedDeadline;
        uint256 periodsPassed = (firstUnresolvedAge * 10) / confirmPeriodBlocks;
        // Overflow check
        if (periodsPassed / 10 >= 255) {
            return type(uint256).max;
        }
        uint256 baseMultiplier = 2**(periodsPassed / 10);
        uint256 withNumerator = baseMultiplier * numerators[periodsPassed % 10];
        // Overflow check
        if (withNumerator / baseMultiplier != numerators[periodsPassed % 10]) {
            return type(uint256).max;
        }
        uint256 multiplier = withNumerator / denominators[periodsPassed % 10];
        if (multiplier == 0) {
            multiplier = 1;
        }
        uint256 fullStake = baseStake * multiplier;
        // Overflow check
        if (fullStake / baseStake != multiplier) {
            return type(uint256).max;
        }
        return fullStake;
    }
```

Figure 2.1: Part of the `currentRequiredStake` function in `RollupUserLogic.sol`

There are manual overflow checks for most of the arithmetic operations involved in this calculation, and the function will return 2**256-1 if an overflow is detected. However, these overflow checks are unreachable, since, as of Solidity 0.8.0, an overflow will automatically cause a revert.

Exploit Scenario

A Rollup enters a state in which certain operations in `currentRequiredStake` overflow, causing an unavoidable revert. Each revert will stop any interactions that depend on the computation of the required stake.

Recommendations

Short term, consider removing the redundant overflow checks in `currentRequiredStake`. Ensure that the arithmetic operations are protected against overflows and implement testing to ensure that if any corner-case overflows do occur, they cause a revert.

Long term, carefully review the breaking changes introduced by each version of Solidity.

3. ERC20Rollup is incompatible with nonstandard ERC20s

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-SC-3
Target: `RollupUserLogic.sol`	

Description

The `ERC20RollupUserLogic` contract is not compatible with ERC20 tokens that do not return a boolean on calls to the `transfer` and `transferFrom` functions.

`ERC20RollupUserLogic` always checks the values returned by the `transferFrom` and `transfer` functions:

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

Figure 3.1: `rollup/RollupUserLogic.sol#L769-L777`

This ensures that the rollup contract will work correctly with ERC20 tokens that follow the standard. However, several tokens (including high-profile tokens) deviate from the ERC20 standard and do not return a boolean on certain calls. (For more information, see "Missing return value bug — At least 130 tokens affected".)

Similarly, if the rollup calls a token whose `transfer` function takes a fee, the rollup may be left with fewer tokens than expected.

Exploit Scenario

The rollup contract is updated to use USDT, which does not return a boolean on a call to `transferFrom` or `transfer`. As a result, calls to those functions will revert, making it impossible to move USDT into or out of the rollup.

Recommendations

Short term, document the fact that the rollup is incompatible with nonstandard ERC20 tokens.

Long term, review the Token Integration Checklist and ensure that the system can handle any deviations from the ERC20 standard.

4. Integer type inconsistency

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-SC-4
Target: `RollupUserLogic.sol`	

Description

The `ChallengeManager` and `RollupCore` contracts use different integer types to represent the inbox count at the end of a rollup (`afterInboxCount`). If the code is refactored, this inconsistency may lead to an issue.

`RollupCore` uses a `uint256` to represent `afterInboxCount`:

```
            uint256 afterInboxCount = assertion
                .afterState
                .globalState
                .getInboxPosition();
             [...]
                // The current inbox message was read
                afterInboxCount++;
```

Figure 4.1: `src/rollup/RollupCore.sol#L598-L613`

The contract's `getInboxPosition` function returns a `uint64`, which the `ChallengeManager` contract uses to represent `afterInboxCount`:

```
        uint64 maxInboxMessagesRead = startAndEndGlobalStates_[1].getInboxPosition();
         if (startAndEndMachineStatuses_[1] == MachineStatus.ERRORED ||
startAndEndGlobalStates_[1].getPositionInMessage() > 0) {
             maxInboxMessagesRead++;
         }
```

Figure 4.2: `ChallengeManager.sol#L105-L108`

In both cases, the variable can be incremented by one.

Currently, `RollupCore` ensures that the inbox count is less than or equal to the current inbox size. Without this check, an attacker could create a node asserting that the inbox size at the end of the execution is 2**64-1, and all attempts to challenge the node would be unsuccessful.

```
            require(
                 afterInboxCount <= memoryFrame.currentInboxSize,
                 "INBOX_PAST_END"
             );
```

Figure 4.3: `rollup/RollupCore.sol#L615-L618`

Exploit Scenario

As part of a codebase update, the constraint requiring that the inbox count be less than or equal to the current inbox size is removed for gas optimization purposes. This enables Eve to create a false assertion that no one can challenge.

Recommendations

Short term, use the `uint64` type for `afterInboxCount` in `RollupCore`. That way, the same type will be used when a node is created and if it is challenged.

Long term, create a list of the system invariants related to the creation of rollup nodes and resolution of challenges.

5. Inclusion of dead code	
Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-NITRO-SC-5
Target: Smart contracts	

Description

The contracts contain various unused functions. This dead code increases the code's complexity and the likelihood of mistakes.

These functions include the following:

- `ChallengeManager.timeUsedSinceLastMove(uint64)` `(src/challenge/ChallengeManager.sol#282-284)`
- `Instructions.newNop()` `(src/state/Instructions.sol#160-162)`
- `MerkleLib.generateRoot(bytes32[])` `(src/libraries/MerkleLib.sol#24-40)`
- `PcArrayLib.set(PcArray,uint256,uint32)` `(src/state/PcArray.sol#13-15)`
- `RollupCore.max(uint256,uint256)` `(src/rollup/RollupCore.sol#557-559)`
- `SequencerInbox.getTimeBounds()` `(src/bridge/SequencerInbox.sol#48-59)`
- `ValueLib.isNumeric(Value)` `(src/state/Value.sol#33-35)`
- `ValueStackLib.hasProvenDepthLessThan(ValueStack,uint256)`
- `ValueStackLib.isEmpty(ValueStack)` `(src/state/ValueStack.sol#37-39)`
- `CryptographyPrimitives.sha256Block(uint256[2],uint256)` `(src/libraries/CryptographyPrimitives.sol#201-330)`

Removing certain of these functions would also make it possible to remove certain contracts. For example, `MerkleLib` has only two functions: `generateRoot`, which is never used, and `calculateRoot`, which is used only once, in the `Outbox` contract. Removing the unused function and adding `calculateRoot` directly to the `Outbox` contract would allow for the removal of the library too.

Recommendations

Short term, remove the unused functions listed in this finding. This will simplify the code and decrease the likelihood of mistakes.

Long term, integrate Slither into the CI pipeline, and use Slither's `dead-code` detector, which found this issue (but be mindful of its current limitations)

6. Unclear expectations surrounding updates to the stake requirements

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NITRO-SC-6
Target: `RollupUserLogic.sol`, `RollupAdminLogic.sol`	

Description

To place stakes on nodes (and thus secure the Arbitrum network), validators must deposit a certain amount of ether or ERC20 tokens. However, when validators move their stakes from one node to another, the system does not check whether the required stake has changed since their initial deposit.

```
    function _newStake(uint256 depositAmount)
        internal
        onlyValidator
        whenNotPaused
    {
        // Verify that sender is not already a staker
        require(!isStaked(msg.sender), "ALREADY_STAKED");
        require(!isZombie(msg.sender), "STAKER_IS_ZOMBIE");
        require(depositAmount >= currentRequiredStake(), "NOT_ENOUGH_STAKE");

        createNewStake(msg.sender, depositAmount);

        rollupEventBridge.stakeCreated(msg.sender, latestConfirmed());
    }
```

Figure 6.1: `rollup/RollupUserLogic.sol#L128-L141`

The amount of this deposit is computed by the `currentRequiredStake` function and depends on the base stake and other values:

```
    function currentRequiredStake(
        uint256 _blockNumber,
        uint64 _firstUnresolvedNodeNum,
        uint256 _latestCreatedNode
    ) internal view returns (uint256) {
        // If there are no unresolved nodes, then you can use the base stake
        if (_firstUnresolvedNodeNum - 1 == _latestCreatedNode) {
            return baseStake;
        }
     …
```

Figure 6.2: The header of the `currentRequiredStake` function in `rollup/RollupUserLogic.sol`

The rollup admin can change the base stake at any time by calling `setBaseStake`:

```
    function setBaseStake(uint256 newBaseStake) external override {
        baseStake = newBaseStake;
        emit OwnerFunctionCalled(12);
    }
```

Figure 6.3: `rollup/RollupAdminLogic.sol#L200-L203`

However, if a validator already has a stake on a confirmed node, the validator can place a stake on other existing nodes regardless of whether the current required stake or even the base stake has changed since the validator's initial deposit:

```
    function stakeOnExistingNode(uint64 nodeNum, bytes32 nodeHash)
        public
        onlyValidator
        whenNotPaused
    {
        require(isStaked(msg.sender), "NOT_STAKED");

        require(
            nodeNum >= firstUnresolvedNode() && nodeNum <= latestNodeCreated(),
            "NODE_NUM_OUT_OF_RANGE"
        );
        Node storage node = getNodeStorage(nodeNum);
        require(node.nodeHash == nodeHash, "NODE_REORG");
        require(
            latestStakedNode(msg.sender) == node.prevNum,
            "NOT_STAKED_PREV"
        );
        stakeOnNode(msg.sender, nodeNum);
    }
```

Figure 6.4: `rollup/RollupUserLogic.sol#L148-L166`

It is unclear whether this loophole in the payment of the required stake is intentional.

Exploit Scenario

Eve participates in the Arbitrum protocol by running multiple validators and has made a base stake payment for each one. In response to a significant decrease in the value of ETH, the Arbitrum team increases the base stake amount to make sure that the network is secure. Eve's validators are still able to place stakes on nodes even though the payments Eve made do not meet the new base stake requirement.

Recommendations

Short term, consider adding more checks to ensure that validators have deposited at least the required stake amount.

Long term, properly document the system invariants related to the rollups and validators and use a fuzzer like Echidna to ensure that they hold.

7. Process for removing old stakes is not scalable

Severity: **Informational**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-NITRO-SC-7
Target: `RollupUserLogic.sol`	

Description

To confirm a new node, a validator must remove any old stakes on previous valid nodes (and any stakes on previous unresolved nodes must be challenged). To do so, the validator must make a manual call to a smart contract for each old stake that needs to be removed. This process will not scale to accommodate a large number of stakers.

```
    function confirmNextNode(bytes32 blockHash, bytes32 sendRoot)
        external
        onlyValidator
        whenNotPaused
    {
        ...

        // All non-zombie stakers are staked on this node
        require(
            node.stakerCount == stakerCount() + countStakedZombies(nodeNum),
            "NOT_ALL_STAKED"
        );

        confirmNode(nodeNum, blockHash, sendRoot);
    }
```

Figure 7.1: Part of the `confirmNextNode` function in `RollupUserLogic.sol`

If there are validators with stakes in previous valid blocks, the validator seeking to confirm a new node must call `returnOldDeposit` to force them to remove their stakes:

```
    function returnOldDeposit(address stakerAddress)
        external
    {
        require(
            latestStakedNode(stakerAddress) <= latestConfirmed(),
            "TOO_RECENT"
        );
        requireUnchallengedStaker(stakerAddress);
        withdrawStaker(stakerAddress);
    }
```

Figure 7.2: The `returnOldDeposit` function in `RollupUserLogic.sol`

Because the number of validators is limited by a whitelist, this approach is not currently an issue. However, the approach would not be practical without a whitelist (or with a larger whitelist); if there were numerous stakers and no good incentives for them to move their stakes, validators would need to track which validators were staked on previous nodes and to call `returnOldDeposit` for each one of them.

Exploit Scenario
Alice wants to confirm a node. She first needs to perform hundreds of calls to remove old stakes. After learning of that requirement, she desists from attempting to validate the Arbitrum network.

Recommendations
Short term, consider either incentivizing stakers to place stakes on the latest node or providing a function that can perform batched calls to `returnOldDeposit` on validators with outdated stakes.

Long term, review the requirements and incentives of the challenge protocol to ensure that they will scale with the number of users, validators, and states.

8. Failure to decrement the amount of time remaining in a challenge

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-NITRO-SC-8
Target: `ChallengeManager.sol`	

Description

To avoid an infinite loop, challenges should have an overall time limit after which they time out. Without a timeout feature, an attacker may be able to prevent the progress of a rollup.

When a staker starts a challenge, the protocol starts tracking the amount of time each participant has spent on the challenge (which is limited to one week).

```
    function createChallenge(
    ..
    ) external onlyValidator whenNotPaused {
        …
        // Calculate upper limit for allowed node proposal time:
        uint256 commonEndTime = getNodeStorage(node1.prevNum).firstChildBlock +
            // Dispute start: dispute timer for a node starts when its first child
is created
            (node1.deadlineBlock - proposedTimes[0]) +
            extraChallengeTimeBlocks; // add dispute window to dispute start time
        …
```

Figure 8.1: Part of `createChallenge` in `RollupUserLogic.sol`

During a challenge, the timestamp of the last move is used to calculate how much time has passed:

```
    function timeUsedSinceLastMove(Challenge storage challenge) internal view
returns (uint256) {
        return block.timestamp - challenge.lastMoveTimestamp;
    }
```

Figure 8.2: The `timeUsedSinceLastMove` function in `ChallengeLib.sol#L40-L42`

The result is compared to the amount of time left:

```
function isTimedOut(Challenge storage challenge) internal view returns (bool) {
    return challenge.timeUsedSinceLastMove() > challenge.current.timeLeft;
}
```

Figure 8.3: The `isTimeOut` function in `ChallengeLib.sol#L44-L46`

However, because the `timeLeft` value is never decremented, each staker can spend one week (the maximum amount of time) on each step.

Exploit Scenario
Eve creates two validators, which place stakes on two nodes with the same parent. She uses one validator to start a challenge against the other and crafts the challenge such that the bisection will have 50 steps. Each time there is a bisection, the amount of time since the last move is reset, allowing her to delay the challenge's resolution with each step (i.e., for 50 weeks). No new nodes can be confirmed until the challenge is over, and users will not be able to withdraw funds from the bridge.

Recommendations
Short term, ensure that `timeLeft` is decremented in each step of the challenge.

Long term, document the important invariants in the codebase and use randomized testing to verify that they cannot be broken. (See Appendix G.)

9. Lack of a lower limit on numSteps in a challenge enables attackers to halt a challenge’s progress

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-NITRO-SC-9
Target: `ChallengeManager.sol`	

Description

There is no lower bound on the number of steps (`numSteps`) in the “execution challenge” phase of a challenge. This means that an attacker could create a segment with a length of zero to prevent his or her opponent from executing any further challenge operations.

The `challengeExecution` function is called to start the “execution challenge” phase:

```
function challengeExecution(
     uint64 challengeIndex,
     ChallengeLib.SegmentSelection calldata selection,
     MachineStatus[2] calldata machineStatuses,
     bytes32[2] calldata globalStateHashes,
     uint256 numSteps
 ) external takeTurn(challengeIndex, selection) {
     [...]

     completeBisection(
         challengeIndex,
         0,
         numSteps,
         segments
     );
```

Figure 9.1: `challenge/ChallengeManager.sol#L169-L226`

The new segment length is defined by `numSteps`, which has no lower bound and can therefore be set to zero. If it is, the second challenge participant will not be able to perform any actions and will lose the challenge when it times out.

Exploit Scenario

Bob and Eve are involved in a challenge. Eve calls `challengeExecution` with `numSteps==0`. As a result, Bob cannot make any moves, and Eve wins the challenge.

Recommendations

Short term, have the `challengeExecution` function check that `numSteps` is non-zero. This will ensure that the caller cannot create an empty segment.

Long term, document the important invariants in the codebase and use randomized testing to verify that they cannot be broken. (See Appendix G.)

10. Challenges can move from the EXECUTION state to the BLOCK state

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-NITRO-SC-10
Target: `ChallengeManager.sol`	

Description

Because of a lack of machine state validation, a challenge can move from the `EXECUTION` state to the `BLOCK` state and loop between the two states indefinitely. An attacker could leverage this flaw to prove an incorrect claim.

The challenge state machine assumes that once a challenge has reached the `EXECUTION` state, it cannot move back to the `BLOCK` state:

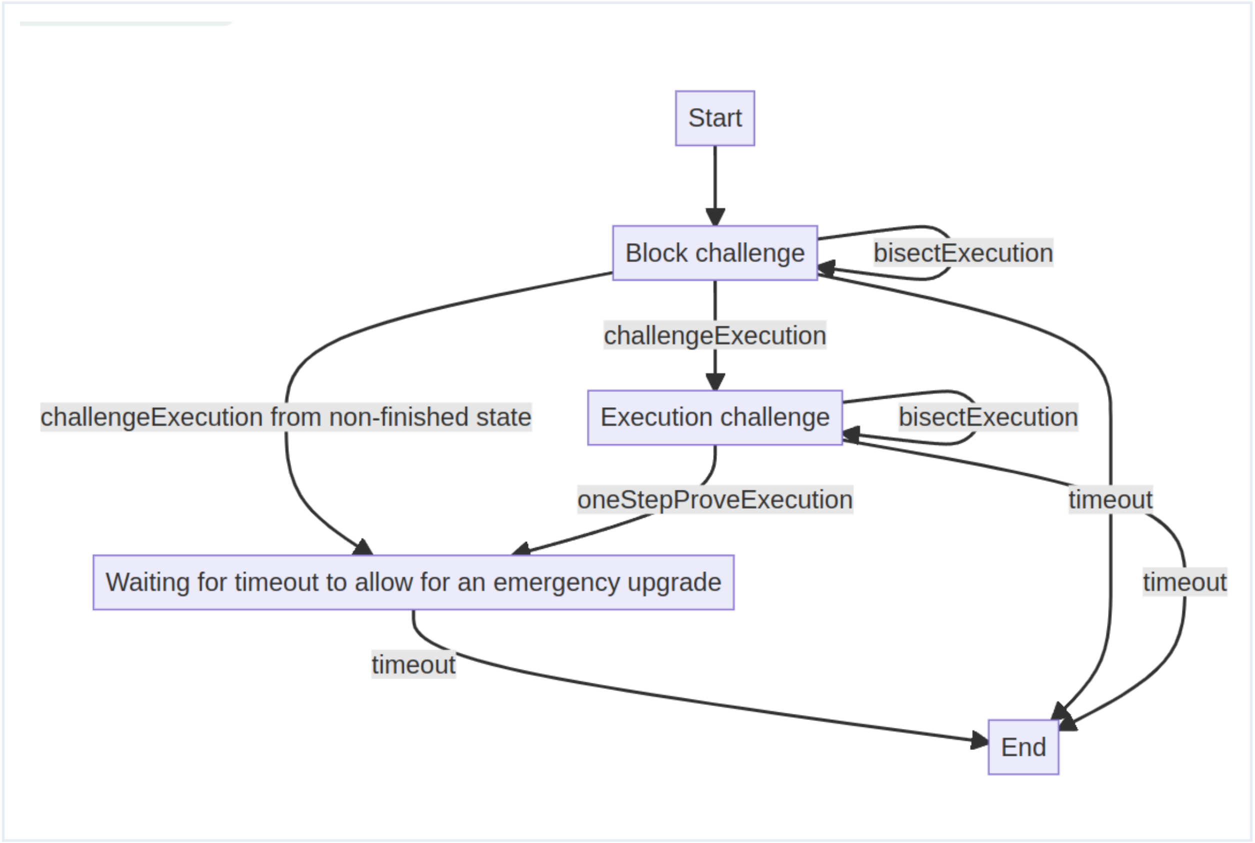


Figure 10.1: `docs/proving/ChallengeManager.md#challengemanager`

When a challenge is created, its state is `BLOCK`:

```
challenge.mode = ChallengeLib.ChallengeMode.BLOCK;
```

Figure 10.2: `challenge/ChallengeManager.sol#L119`

When `challengeExecution` is called, the challenge's mode is set to `EXECUTION`:

```
challenge.mode = ChallengeLib.ChallengeMode.EXECUTION;
```

Figure 10.3: `challenge/ChallengeManager.sol#L219`

However, because the challenge's mode is never read, the challenge can loop between the `BLOCK` and `EXECUTION` states indefinitely. This creates two ways to abuse the protocol:

1. An attacker could start a challenge with an incorrect claim and then start the third round of bisection positioned on a correct claim. Note, though, that `challengeExecution` cannot be called if the status of the machine is `Running` (because that would cause `blockStateHash` to revert). If the other participant was honest, the attacker could call `challengeExecution` if the bisection ended on the first instruction of a block to validate this condition (i.e., that the machine status is not `Running)`.

2. An attacker could create an infinite loop if both participants in the challenge are malicious.

Exploit Scenario

Eve creates a rollup in which the first instruction of block 2 leads to an incorrect outcome. Bob challenges Eve's rollup. After the bisection in the "execution challenge" phase, the challenge progresses to the beginning of block 2, and it is Eve's turn.

Eve is supposed to prove the execution by calling `oneStepProveExecution`. Instead, she calls `challengeExecution`, resetting the challenge in the "challenge execution" phase and starting the bisection with the correct claim. On his next turn, Bob needs to prove that the correct execution is impossible, which he cannot do. As a result, Eve wins the challenge.

Recommendations

Short term, have `challengeExecution` check that the mode of a challenge is `BLOCK`. This will prevent `challengeExecution` from being called multiple times in the same challenge.

Long term, document the important invariants in the codebase and use randomized testing to verify that they cannot be broken. (See appendix G.)

Summary of Recommendations

The Arbitrum Nitro system is a work in progress with multiple planned iterations. Trail of Bits recommends that Offchain Labs address the findings detailed in this report and take the following additional steps prior to deployment:

- Freeze the codebase and create centralized documentation on the system invariants, including those listed in Appendix G.
- Using the fuzzing work detailed in Appendix E as a starting point, integrate fuzz testing into the development process. Specifically,
    - use the model developed by Trail of Bits and invariants listed in Appendix G to fuzz the Solidity `Rollup` and `ChallengeManager` contracts;
    - perform differential fuzzing to compare the Solidity OSP contracts and the Arbitrator code; and
    - Fuzz the ArbOS entry points (e.g., the L2 message-parsing process and the inbox).
- Continue improving the system's documentation, particularly that on the gas rules, the parsing of L2 messages, the error statuses (e.g., `TooFar`), and the aggregators' expected roles.
- Use a WASM reference implementation as the ground truth in testing and fuzzing of Nitro's version of WASM
- Perform additional security reviews focused on the areas highlighted in the "Coverage Limitations" subsection of the Project Coverage section
- Ensure that all known issues have been fixed and that no TODOs are left.

A. Vulnerability Categories

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

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

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

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

B. Code Maturity Categories

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

Code Maturity Categories

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

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

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

General

Use the same terminology across the modules. For example, the smart contracts use `sender` and `inboxSeqNum` to refer to the sender and ID of a message (`solgen/src/bridge/Messages.sol#L10-L16`), while ArbOS uses `poster` and `requestId` (`arbos/incomingmessage.go#L38-L44`).

Smart Contracts

Consider using the less than operator in `require(zombieNum <= zombieCount(), "NO_SUCH_ZOMBIE")` in `AbsRollupUserLogic.removeZombie`. This will prevent the use of a zombie identifier that is invalid. Although the transaction would revert in the next line if an invalid identifier were used, the user would not receive a proper revert message (`NO_SUCH_ZOMBIE`); instead, the user would receive an out-of-bounds Solidity error.

D. WAVM Code and Design Recommendations

We identified several discrepancies between the VM implementations of the arbitrator and the one-step-proof contracts caused by edge cases and undefined behavior. This appendix contains recommendations on improving the WAVM design and testing.

Specification

- **Create a specification for each opcode.** While most of the opcodes follow the WASM specification, it is unclear how the implementations should handle edge cases and implementation-specific details.

- **Explicitly detail how to handle errors.** For example, specify whether errors should trigger the `Errored` status or whether the state transition should be blocked or reverted.

- **Add pre- and postconditions for each opcode.** Having pre- and postconditions would facilitate the review and testing of the implementations.

Design

- **Use the same code architecture for the arbitrator and the one-step-proof contract.** Currently, the one-step-proof opcodes are split across four contracts, while the arbitrator opcodes are concentrated in one contract. Using the same code architecture would make it easier to review the code and to identify any divergence.

Testing

- **Create standalone comparison tests for every opcode.** This would enable the team to test both implementations of every opcode at the same time. However, it would be necessary to ensure that the code architecture allows for standalone runs.

- **Create a differential fuzzer to identify deviations between the VM implementations of the arbitrator and the one-step-proof contract. Fuzz every opcode in standalone.** The differential fuzzer will help identify divergences.

E. Fuzzer-Based Test Cases for Arbitrum Nitro

Trail of Bits developed test cases to perform fuzz testing of the Arbitrum Nitro codebase with Echidna, AFL.rs, and go-fuzz. Fuzzing provides a way to explore the input space of software in order to uncover robustness and correctness issues. We focused our fuzzing efforts on ArbOS and the arbitrator.

Fuzzing the Rollup System Invariants

To perform property-based testing of the `Rollup` contracts, Trail of Bits used the original code to develop a model replicating their behavior in Solidity. It was not possible to use the unmodified `Rollup` contract code in this testing, as certain state-related assertions have conditions that are difficult to explore through random testing (e.g., hash matching). The smart contract model simplifies the creation of nodes, for example, because it does not require any conditions on the related assertion itself:

```
function stakeOnNewNode(
    Assertion calldata assertion,
    bytes32 expectedNodeHash,
    uint256 prevNodeInboxMaxCount
) public {
    require(isStaked(msg.sender), "NOT_STAKED");
    // Ensure staker is staked on the previous node
    uint64 prevNode = latestStakedNode(msg.sender);

    {
        uint256 timeSinceLastNode = block.number -
            getNode(prevNode).createdAtBlock;
        // Verify that assertion meets the minimum Delta time requirement
        require(timeSinceLastNode >= minimumAssertionPeriod, "TIME_DELTA");
    }
    createNewNode(assertion, prevNode, prevNodeInboxMaxCount, expectedNodeHash);

    stakeOnNode(msg.sender, latestNodeCreated());
}
```

Figure E.1: Part of an Echidna test for the `Rollup` contract code

Similarly, it was not possible to use the `ChallengeManager` code directly in random testing. Our model makes it possible to perform fuzz testing because an arbitrary winner is selected for every challenge:

```
function completeChallenge(address winningStaker, address losingStaker)
    external
{
    require(winningStaker != losingStaker);
    require (NO_CHAL_INDEX != inChallenge(winningStaker, losingStaker));
    completeChallengeImpl(winningStaker, losingStaker);
    }
```

Figure E.2: Part of an Echidna test for the `Rollup` code

This code allowed us to test system properties including the following:

- If the preconditions are met, `createChallenge` never reverts.
- If a challenge can be created, then `commonEndTime` >= `proposedTimes[0]`, and `commonEndTime` >= `proposedTimes[1]`.
- If the preconditions are met, `removeOldZombies` never reverts.
- If the preconditions are met, `removeZombie` never reverts.
- For every address, x, if `isStaked(x)`, then `!isZombie(x)`.
- For every address, x, if `(latestStakedNode(x)` <= `latestConfirmed())`, then `currentChallenge(x)` == NO_CHAL_INDEX.

Fuzzing ArbOS to Detect Crashes

Trail of Bits's fuzz testing covered essential ArbOS functionalities including the following:

- L1 and L2 message parsing
- Inbox parsing and processing
- Delayed-message parsing and processing
- Block production
- Transaction and hook execution

Figure E.3 shows part of a test that injects random inputs into the sequencer and delayed-message inbox:

```
func FuzzInbox(input []byte) int {
        arbstate.RequireHookedGeth()
        chainDb := rawdb.NewMemoryDatabase()
        stateRoot, err := arbosState.InitializeArbosInDatabase(chainDb,
&statetransfer.ArbosInitializationInfo{})
        if err != nil {
                panic(err)
        }
        statedb, err := state.New(stateRoot, state.NewDatabase(chainDb), nil)
        if err != nil {
                panic(err)
        }
        …
       delayedMessages := [][]byte{input}
        seqBatch := make([]byte, 40)
        binary.BigEndian.PutUint64(seqBatch[32:], uint64(len(delayedMessages)))
        seqBatch = append(seqBatch, input...)
        inbox := &inboxBackend{
                batchSeqNum:           0,
                batches:               [][]byte{seqBatch},
                positionWithinMessage: 0,
                delayedMessages:       delayedMessages,
        }
        _, err = BuildBlock(statedb, genesis, noopChainContext{},
params.ArbitrumOneChainConfig(), inbox)
        if err != nil {
                // With the fixed header it shouldn't be possible to read a delayed
message,
                // and no other type of error should be possible.
                panic(err)
        }

        return 0
}
```

Figure E.3: Part of a `go-fuzz` test for ArbOS

Our fuzzing campaigns identified the issues detailed in TOB-NITRO-ARBOS-8, TOB-NITRO-ARBOS-16, and TOB-NITRO-ARBOS-17.

Fuzzing the Precompiled ArbOS Contracts

Trail of Bits modified the fuzz test code provided by the Offchain Labs team to run the precompiled contract calls, seeking to increase its coverage. We made the following important changes:

- Using a non-null `from` address to send transactions to the precompiled contracts
- Using a `from` address for chain owners
- Adding expired and unexpired retryable tickets so that the fuzzer would execute retryable ticket operations correctly
- Using a `from` address for the `beneficiary` to enable the fuzzer to cancel retryable tickets

```
func FuzzPrecompiles(input []byte) int {
        …
        _, err = state.RetryableState().CreateRetryable(lastTimestamp, id, timeout,
from, &to, callvalue, beneficiary, calldata)
        …
        id = common.BytesToHash([]byte{1})
        _, err = state.RetryableState().CreateRetryable(lastTimestamp, id,
lastTimestamp-1, from, &to, callvalue, beneficiary, calldata)
        …
        id = common.BytesToHash([]byte{2})
        _, err = state.RetryableState().CreateRetryable(lastTimestamp, id,
lastTimestamp-1, from, &to, callvalue, beneficiary, calldata)
        …

}
```

Figure E.4: Part of a `go-fuzz` test for ArbOS

Fuzzing the External Libraries Used in ArbOS

Certain parts of the ArbOS code rely on external code to parse inputs. While these parts of the code could technically be reached through other kinds of testing, it is best to use dedicated fuzz tests to make sure that they are covered extensively.

```
func FuzzBrotli(input []byte) int {
        reader := brotli.NewReader(bytes.NewReader(input))
        if _, err := ioutil.ReadAll(reader); err != nil {
                return 1
        }
        return 0
}

func FuzzPublicKeyFromBytes(input []byte) int {
        blsSignatures.PublicKeyFromBytes(input, false);
        blsSignatures.PublicKeyFromBytes(input, true);
        return 0
}
```

Figure E.5: Two `go-fuzz` tests dedicated to ArbOS

Setting up Fuzzing in Go

Because the ArbOS code has been rewritten in Go, Trail of Bits tested the code through `go-fuzz`, a coverage-guided fuzzing solution for Go packages.

To install and build an ArbOS fuzz test using the FUNC function (`FuzzPrecompiles`, `FuzzPublicKeyFromBytes`, `FuzzIncomingMessage`, or `FuzzBrotli`), invoke the following:

```
$ go-fuzz-build -libfuzzer -func $FUNC
$ clang -fsanitize=fuzzer reflect-fuzz.a -o fuzz.$FUNC.libfuzzer
```

Once the build has finished, execute the generated binary directly to start the fuzzing campaign. The following commands can be used to run the fuzzer with an empty corpus:

```
$ mkdir corpus.$FUNC
$ ./fuzz.$FUNC.libfuzzer corpus.$FUNC
```

See the libFuzzer documentation for information on adjusting the parameters.

Measuring Coverage in Go

We used the code coverage tool provided by the Golang compiler to measure the coverage of our testing. The first step in this process is selecting a FUNC to use (`FuzzPrecompiles`, `FuzzPublicKeyFromBytes`, `FuzzIncomingMessage`, or `FuzzBrotli`) and replacing with the appropriate name in `precompile_fuzz_test.go`:

```
$ FUZZ_CORPUS_DIR=corpus.$FUNC/
$ go test . -cover -coverprofile coverage -coverpkg=$FUNCPKGS
```

The `FUNCPKGS` parameter depends on the fuzz test selected by the user. For instance, when one is using `FuzzBrotli`, this important parameter should be set to `github.com/andybalholm/brotli`. However, when fuzzing an ArbOS-specific package, it is necessary to specify a local path (e.g., `../../arbos`). If an incorrect or invalid package is used, the tool will not identify any coverage.

After the coverage data has been collected, the user can view it in an HTML file by invoking the following command:

```
$ go tool cover -html=coverage
```

The tool will then open a web browser displaying the coverage results.

Manually annotating branches that would be unreachable under normal circumstances (i.e., everything but out-of-memory errors) makes it easier to monitor a fuzzing campaign during development or a security review. By using this approach, we found that the `timeout` error branch may be unreachable:

```
// Gets the timestamp for when ticket will expire
func (con ArbRetryableTx) GetTimeout(c ctx, evm mech, ticketId bytes32) (huge,
error) {
        retryableState := c.state.RetryableState()
        retryable, err := retryableState.OpenRetryable(ticketId,
evm.Context.Time.Uint64())
        if err != nil {
                return nil, err
        }
        if retryable == nil {
                return nil, ErrNotFound
        }
        timeout, err := retryable.Timeout()
        if err != nil {
                return nil, err
        }
        return big.NewInt(int64(timeout)), nil
}
```

Figure E.6: An unreachable branch in `GetTimeout`

Fuzzing the Arbitrator to Detect Crashes

Trail of Bits also developed fuzz tests covering most of the Arbitrator functionalities, including the following:

- WASM parsing, validation, and processing
- WASM–WAVM conversion, processing and execution (up to 100 steps)
- Machine status verification, serialization, and hashing

Through our fuzzing campaigns, we found that the Arbitrator is susceptible to crashes caused by the improper handling of WASM binaries (as detailed in TOB-NITRO-ARBITATOR-2).

Figure E.7 shows part of a test created to check the robustness of the arbitrator code.

```
fn main() -> Result<()> {
    …
    if opts.fuzz {
        afl::fuzz!(|data: &[u8]| {
          if let Err(err) = fuzz_inner(data, libraries.clone()) {
           eprintln!("Non-fatal error: {}", err);
          }
        });

        Ok(())
    }
    …
}

fn fuzz_inner(bin: &[u8], libraries: Vec<WasmBinary>) -> Result<()> {
    let main_bin = parse_binary_from_bytes(bin)?;
    let mut mach = Machine::from_binary(
        libraries,
        main_bin,
        false,
        false,
        GlobalState::default(),
        HashMap::default(),
        HashMap::default(),
    )?;
    while !mach.is_halted() && mach.get_steps() < 1000 {
        mach.serialize_proof();
        mach.step();
    }

    Ok(())
}
```

Figure E.7: Part of an AFL.rs test for the arbitrator

Executing Differential Fuzzing of the WASM and WAVM Code

Trail of Bits also developed a differential fuzzer to compare the arbitrator's emulation of WAVM code to an external WASM implementation. We used differential fuzzing to identify discrepancies between the WASM and WAVM implementation, which could have a severe impact on the arbitrator's correctness and the validity of its proofs.

We used the `wasmi` crate for the external WASM implementation. There are several main reasons that we chose this crate:

- It purportedly provides correct and deterministic WebAssembly execution.
- It is well tested and is actively being developed.
- It has low overhead requirements and offers cross-platform support.
- It can easily be compiled with AFL.rs instrumentation.

Our main goal in developing this differential fuzzer was to evaluate how a WASM execution succeeds or fails and to identify the last WAVM opcode in an execution. If opcodes do not match, there is a divergence in the executions.

Building this differential fuzzer was not a straightforward task. We faced a few challenges stemming from the following discrepancies in the WASM and WAVM implementations:

- The WASM binaries may not be processed in the exact same way as the WAVM binaries processed by the arbitrator. To mitigate this issue, we needed to ensure that the WASM imports and exports were processed in the exact same way.
- The WASM reference implementation lacks Arbitrum's internal instructions, such as those related to the handling of preimages and the inbox. Thus, we discarded executions that use Arbitrum's internal instructions.
- The WASM and WAVM emulations may count steps in different ways, making it more difficult to compare executions with a limited number of steps. To mitigate that limitation, after the WASM and WAVM executions ended, we used their return values or the resultant stacks to compare the machines.

To avoid any issues, we compared executions that do not loop or include any Arbitrum-specific opcodes. We also accounted for the documented WASM and WAVM discrepancies to avoid false positives:

```
let wasmi_start_mach = wasmi_start_mach_instance.unwrap().run_start_with_stack(&mut
NopExternals, &mut StackRecycler::default());
if let Err(trap) = wasmi_start_mach {
  println!("This was binary failed with {}", trap);
  match (trap.kind().clone(), last_opcode) {
      (TrapKind::Unreachable, Opcode::Unreachable) => println!("Unreachable."),
      (TrapKind::UnexpectedSignature, Opcode::CallIndirect) =>
println!("UnexpectedSignature."),
      (TrapKind::ElemUninitialized, Opcode::CallIndirect) =>
println!("ElemUninitialized."),
      (TrapKind::TableAccessOutOfBounds, Opcode::CallIndirect) =>
println!("TableAccessOutOfBounds."),
      (TrapKind::MemoryAccessOutOfBounds, Opcode::MemoryLoad { .. }) =>
println!("MemoryAccessOutOfBounds."),
      (TrapKind::MemoryAccessOutOfBounds, Opcode::MemoryStore { .. }) =>
println!("MemoryAccessOutOfBounds."),
      (TrapKind::DivisionByZero, _) => println!("DivisionByZero."),
      (TrapKind::StackOverflow, _) => println!("StackOverflow."),
      _ => panic!("This binary execution failed: {} with opcode: {}", trap,
last_opcode.repr()),
   }
} else {
  if mach.get_status() == MachineStatus::Errored && last_opcode != Opcode::Return {
    panic!("Machine should not return error!");
  }
}
```

Figure E.8: Part of an AFL.rs test used in differential fuzzing of the WASM and WAVM implementations

Setting Up Fuzzing in Rust

Because the arbitrator was developed in Rust, Trail of Bits used AFL.rs to implement fuzz tests of the component. AFL.rs is an in-process coverage-guided evolutionary fuzzing engine based on AFL.

To build fuzz tests in the arbitrator code, invoke the following `cargo` commands:

```
$ cargo install afl
$ cargo afl build
```

After the build has finished, use `cargo afl fuzz` to execute the fuzzer. To run the fuzzer with a single seed, invoke the following code:

```
$ mkdir in
$ echo a > in/seed
$ AFL_MAP_SIZE=500000
cargo afl fuzz -i in -o out target/debug/prover -- --fuzz dummy
afl-fuzz++3.14c based on afl by Michal Zalewski and a large online
community
[+] afl++ is maintained by Marc "van Hauser" Heuse, Heiko "hexcoder"
Eißfeldt, Andrea Fioraldi and Dominik Maier
[+] afl++ is open source, get it at
https://github.com/AFLplusplus/AFLplusplus
[+] NOTE: This is v3.x which changes defaults and behaviours - see
README.md
[*] Getting to work...
[+] Using exponential power schedule (FAST)
[+] Enabled testcache with 50 MB
...
```

See the AFL.rs documentation for information on adjusting the parameters.

We recommend using a small number of precompiled WASM files from the `arbitrator/prover/test-cases` directory as the initial seeds. This will enable the fuzzer to explore the input space of the arbitrator more efficiently.

Measuring Code Coverage in Rust

Regardless of how inputs are generated, it is important to measure the coverage of a fuzzing campaign. To measure the coverage of our Rust fuzz testing, we used `grcov`'s source-based code coverage feature. It is important to disable the AFL.rs instrumentation to avoid conflicts with the coverage measurement code. This requires compiling the codebase without using AFL.rs:

```
$ export CARGO_INCREMENTAL=0
$ export RUSTFLAGS="-Zprofile -Ccodegen-units=1 -Copt-level=0
-Clink-dead-code -Coverflow-checks=off -Zpanic_abort_tests
-Cpanic=abort"
$ export RUSTDOCFLAGS="-Cpanic=abort"
$ cargo build
```

After the build is complete, use the resulting binary to execute each file from the corpus. To generate an HTML coverage report, use the following command:

```
$ grcov . -s . --binary-path ./target/debug/ -t html --branch
--ignore-not-existing -o ./target/debug/coverage/
```

The result will be available in `./target/debug/coverage/index.html`.

Integrating Fuzzing and Coverage Measurement into the Development Cycle

Once the fuzzing process has been tuned to be fast and efficient, it should be integrated into the development cycle and used to catch bugs. We recommend using a CI system and adopting the following process:

1. After running the initial fuzzing campaign, save the corpus generated for every test. Trail of Bits provided these initial corpora.

2. Re-run the fuzzing campaign for each internal milestone, new feature, or public release. Start with the current corpora for each test and run the campaign for at least for 24 hours.[1]

3. Update the corpora to include the new inputs generated by the fuzzer.

Note that over time, the corpora will come to represent thousands of CPU hours of refinement and be a very valuable tool for efficiently guiding code coverage during fuzz testing. However, because an attacker could use the corpora to quickly identify vulnerable code, we recommend storing the fuzzing corpora in an access-controlled location rather than in a public repository. Some CI systems allow maintainers to keep a cache to accelerate building and testing. The corpora can be included in such a cache if they are not very large.

As an alternative to in-house fuzz testing, the Offchain Labs team could use OSS-Fuzz. OSS-Fuzz makes it possible to execute automated fuzzing campaigns using Google's extensive testing infrastructure. OSS-Fuzz is free for widely used open-source software. We believe OSS-Fuzz would accept the Arbitrium Nitro system as a project.

Using OSS-Fuzz is beneficial because Google provides access to all of its infrastructure for free and will notify a project’s maintainers any time that a change in the source code introduces a new issue. Moreover, the reports it provides include essential information such as information on test case minimization and backtraces. However, there are some downsides: If OSS-Fuzz discovers critical issues, Google employees will learn of the issues before the project’s own developers. Google policy also requires bug reports to be made public after 90 days, which may not be in Offchain Labs’s best interests. The team should weigh these risks against the benefits when deciding whether to request use of Google’s free fuzzing resources.

[1] For more on fuzz-driven development, see the CppCon 2017 talk delivered by Kostya Serebryany of Google.

F. Detecting Destructible Contracts

Trail of Bits wrote a custom Slither script to ensure that the Nitro smart contracts' upgradeability does not lead to the arbitrary destruction of the contracts' implementation. We recommend adding this script to the CI pipeline to prevent future code updates from introducing vulnerabilities.

```
from slither import Slither
from slither.core.declarations import SolidityFunction
from slither.slithir.operations import LowLevelCall, SolidityCall



def main():
    sl = Slither(".", ignore_compile=True)

    for compilation_unit in sl.compilation_units:
        for contract in compilation_unit.contracts:
            if contract.name == "TransparentUpgradeableProxy":
                continue
            for function in contract.functions_entry_points:
                if function.is_constructor:
                    continue
                for ir in function.all_slithir_operations():
                    if isinstance(ir, LowLevelCall) and ir.function_name in [
                        "delegatecall",
                        "codecall",
                    ]:
                        if "onlyProxy" in [m.name for m in function.modifiers]:
                            continue
                        print(
                            f"{function} uses an unprocted delegatecall:
{ir.node.source_mapping_str}"
                        )
                        exit(-1)
                    elif isinstance(ir, SolidityCall) and ir.function in [
                        SolidityFunction("selfdestruct(address)"),
                        SolidityFunction("suicide(address)"),
                    ]:
                        print(
                            f"{function} uses selfdestruct:
{ir.node.source_mapping_str}"
                        )
                        exit(-1)
    print("No issue was found")


if __name__ == "__main__":
    main()
```

Figure F.1: `Check_destruct.py`

G. System Invariants

The Arbitrum Nitro system is a moving target that relies on complex invariants. The maintenance of certain of these invariants depends on data validation performed by multiple components (e.g., the smart contracts and ArbOS); this inherent complexity makes it difficult to review the codebase and to avoid breaking the invariants during code updates. We recommend that Offchain Labs freeze the codebase and document all known system invariants. The documentation on each invariant should include information on the components it is related to and the ways in which it is checked (e.g., manualfly or through unit tests, static analysis, fuzzing, etc.); any broken invariants should be identified as such.

Certain of the invariants identified during our review are listed below. This list includes invariants that are broken or could not be confirmed to hold.

Smart Contracts

General

- **With the exception of the proxy, the contracts cannot self-destruct.** *(Checked via static analysis.)*
    - See Appendix F.

Challenge

- **If the machine's status is `RUNNING`, the challenge will time out.** *(Checked manually.)*
    - The `challengeExecution` function calls `blockStateHash`, which will revert if the status is `RUNNING` (`challenge/ChallengeManager.sol#L179-L182`).
- **If neither participant advances a challenge, it will time out.** *(Invariant broken.)*
    - See TOB-NITRO-SC-8.
- **Every challenge has a unique index that is different from the index of `NO_CHALLENGE`.** *(Checked manually.)*
    - The index is based on the `totalChallengesCreated` counter, which will always be greater than zero (`challenge/ChallengeManager.sol#L98`).

Rollup

- **A node cannot challenge itself.** *(Checked manually.)*

    - When a challenge is created, the nodes involved in the challenge have different identifiers (`rollup/RollupUserLogic.sol#L288`).

- **For every address, x, if `isStaked(x)`, then `!isZombie(x)`.** *(Checked manually.)*
    - This applies to each address, x, to which a stake is moved or for which a stake is created (`rollup/RollupUserLogic.sol#L135`).

- **For every address, x, if (`latestStakedNode(x) <= latestConfirmed()`), then `currentChallenge(x) == NO_CHAL_INDEX`.** *(Checked manually.)*
    - A validator staked on a node older than the latest confirmed node cannot start a challenge (`RollupUserLogic.sol#L290`). Similarly, no new nodes can be confirmed if any validators remain staked on an old node or involved in a challenge over an old node (`RollupUserLogic.sol#L116-L119`).

Rollup - Challenge

- **Block number operations use `uint64` values.** *(Checked manually.)*
    - `src/rollup/RollupLib.sol#L96`
    - `challenge/ChallengeManager.sol#L87`

- **Inbox count operations use `uint64` values.** *(Invariant broken.)*
    - See TOB-NITRO-SC-4.

- **The first bisection starts with a length of at least one.** *(Checked manually.)*
    - When a node is created, it has at least one block (`rollup/RollupUserLogic.sol#L200`, `challenge/ChallengeManager.sol#L129`).

- **A node challenge starts in the FINISHED state.** *(Checked manually.)*
    - Nodes are created only if the status at the start of the challenge is FINISHED (`rollup/RollupUserLogic.sol#L206`).

- **A node challenge ends in the FINISHED or ERRORED state.** *(Checked manually.)*
    - Nodes are created only if the status at the end of the challenge is FINISHED or ERRORED (`rollup/RollupCore.sol#L577-L580`).

ArbOS Bridge

- **When the root is updated, the updated version contains the previous leaves at the same index.** *(Unconfirmed.)*

    - The ArbOS Merkle tree construction maintains subtrees from left to right.

ArbOS Smart Contracts

- **There is no more than one retryable ticket in each block.** *(Checked manually.)*
    - This invariant is necessary because the hash of a retryable ticket is the hash of the block's header (`arbstate/inbox.go#L284-L299`).

WAVM

- **ArbOS's WASM implementation follows the WASM specification.** *(Checked through differential fuzzing.)*
    - See Appendix E.

ArbOS

- **Retryable tickets can be executed correctly only once.** *(Checked manually.)*
    - When a retryable ticket is executed without reverting, it is deleted at the end of the transaction hook.
- **The amount of ETH in L2 is equal to the amount of ETH indicated by the `expectedBalanceDelta` value.** *(Invariant broken.)*
    - See TOB-NITRO-GETH-3, TOB-NITRO-ARBOS-3, and TOB-NITRO-ARBOS-7.