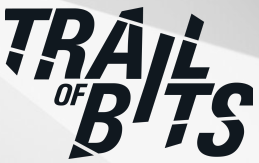

Scroll l2geth Diff Review

Security Assessment (Summary Report)

October 6, 2023

Prepared for:
Haichen Shen
Scroll

Prepared by: **Anish Naik, Nat Chin, and Vara Prasad Bandaru**

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 100+ 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

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

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

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

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Brooke Langhorne, Project Manager
brooke.langhorne@trailofbits.com

The following engineering director was associated with this project:

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

The following consultants were associated with this project:

Anish Naik, Consultant
anish.naik@trailofbits.com

Nat Chin, Consultant
natalie.chin@trailofbits.com

Vara Prasad Bandaru, Consultant
vara.bandaru@trailofbits.com

Project Timeline

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

Date	Event
August 18, 2023	Pre-project kickoff call
August 28, 2023	Delivery of summary report draft
August 28, 2023	Report readout meeting
September 26, 2023	Delivery of summary report
October 6, 2023	Delivery of summary report with detailed findings

Executive Summary

Scroll engaged Trail of Bits to review the security of its `l2geth` implementation at commit hash `be1600f`. `l2geth` is a fork of `go-ethereum` that was developed to support Scroll's zero-knowledge (ZK) rollup. This review was a follow-up review to a previous `l2geth` security assessment. The core focus of this audit was the introduction of the circuit capacity checker (CCC), which is responsible for identifying whether a specific transaction or block is "unprovable."

A team of three consultants conducted the review from August 21 to August 25, 2023, for a total of two engineer-weeks of effort. With full access to the source code, we performed a manual review of the diff from the previous audit.

Within our main area of focus, we sought to answer the following non-exhaustive list of questions:

- Does the CCC correctly handle unprovable L1 transactions, L2 transactions, and blocks?
- Does the introduction of the CCC adversely affect the miner or block validation code paths?
- Does the system correctly log and handle errors?
- Does the introduction of the custom tracer cause unexpected state changes or undefined behavior?

This audit was a best-effort review of the above goals. Due to time limitations, we were unable to identify and validate the impact of all edge cases that the CCC may introduce into the system. Additionally, we were provided with only the API for the CCC, and the true implementation of the CCC was treated as a black box. Thus, any issues that may arise from an incorrect implementation of the feature were considered out of scope. Finally, the `l2geth` diff introduced a variety of other changes, outside of the CCC, that were also considered out of scope for this audit.

We identified a few patterns that caused the issues discovered during this audit. First, we identified a high-severity denial-of-service (DoS) attack that would prevent the system from fully taking advantage of the block size. This was due to a bug within the CCC implementation. Transaction ordering risks, as well as DoS attack vectors, must be continually considered as the codebase evolves and matures.

Second, we identified a large amount of dead or unused code. These are artifacts from the original `go-ethereum` implementation. This technical debt makes the codebase harder to review and increases the effort required to maintain the codebase.

Finally, there was a lack of documentation around the introduction of the CCC. Since the CCC significantly changes the functionality of critical code paths within the system, identifying the invariants that must be upheld is critical.

Going forward, we recommend that the Scroll team invest further efforts into improving documentation and maturing the test suite. The documentation should hold the system's expected behavior to identify deviations and changes that were made from the initial `go-ethereum` fork and should also hold any critical function- and system-level invariants that arise from those changes. Additionally, we recommend that the team invest more time into integration testing with a true implementation of the CCC, instead of a mock one, to ensure that the system works as expected.

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	The diff from the previous audit did not introduce any significant changes to the arithmetic operations performed in the codebase.	**Not Considered**
Auditing	The logging pattern provided in the diff remains consistent with that of the original `go-ethereum`. Sufficient trace, debug, error, and critical log events are used to effectively triage and identify any issues within the system.	**Satisfactory**
Authentication / Access Controls	No meaningful changes to `go-ethereum`'s access controls were made in `l2geth`.	**Not Considered**
Complexity Management	As mentioned in the previous audit of `l2geth`, the system carries a large amount of technical debt from `go-ethereum`. This makes it harder to review the `l2geth` code and determine whether a specific code quirk reflects a quality issue with `l2geth` or a workaround for a legacy feature that `go-ethereum` maintains. `l2geth`'s changes also contain many unresolved "TODO" statements and code quality issues, likely stemming from a fast development life cycle. Full inclusion of the `go-ethereum` source code makes code atrophy from these changes much harder to detect against the background of historical `go-ethereum` quirks.	**Weak**

Cryptography and Key Management	No meaningful changes to `go-ethereum`'s cryptography or key management were made in the diff of `l2geth`.	**Not Considered**
Decentralization	As mentioned in the previous audit of `l2geth`, the system runs using a centralized sequencer, which represents a single point of failure and exposes off-chain applications to potential double-spend attacks. The diff that was audited did not make any changes to any code paths that would affect decentralization.	**Not Considered**
Documentation	There is a lack of documentation regarding the new CCC integration. Given the criticality of the CCC for block production and validation, as well as the nuanced behavior of the feature, we recommend increasing documentation in these areas to ensure that expected behavior and properties are clear.	**Weak**
Memory Safety and Error Handling	We identified many instances where an error is caught and logged but not returned to the caller of the function. Based on discussions with Scroll, these instances operate as expected. However, this coding practice may lead to the introduction of latent bugs and cause unhandled errors as development continues. We recommend that Scroll create internal guidelines to identify when and where errors should be returned versus when they should be only logged.	**Moderate**
Low-Level Manipulation	`l2geth`'s modifications do not introduce any low-level manipulation.	**Not Considered**
Testing and Verification	The changes made to the miner code paths were sufficiently tested with unit tests. However, changes to the block validation and tracing code paths were not directly tested with unit tests. We recommend that Scroll extend its integration testing suite so that the `l2geth` system can be tested in tandem with a true implementation of the CCC, instead of a mock one.	**Moderate**

Transaction Ordering	The prioritization of L1 transactions to be the first in a block allowed the DoS attack that we identified. This prioritization may also lead to additional DoS vectors or transaction ordering–related risks. Due to time constraints, we were unable to examine these other potential attack vectors.	**Further Investigation Required**

Summary of Findings

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

ID	Title	Type	Severity
1	Attacker can prevent L2 transactions from being added to a block	Denial of Service	High
2	Unused and dead code	Undefined Behavior	Informational
3	Lack of documentation	Undefined Behavior	Informational

Detailed Findings

1. Attacker can prevent L2 transactions from being added to a block

Severity: **High**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-L2GETH-1
Target: `miner/worker.go`	

Description

The `commitTransactions` function returns a flag that determines whether to halt transaction production, even if the block has room for more transactions to be added.

If the circuit checker returns an error either for row consumption being too high or reasons unknown, the `circuitCapacityReached` flag is set to `true` (figure 1.1).

```
case (errors.Is(err, circuitcapacitychecker.ErrTxRowConsumptionOverflow) &&
tx.IsL1MessageTx()):
	// Circuit capacity check: L1MessageTx row consumption too high, shift to the
next from the account,
	// because we shouldn't skip the entire txs from the same account.
	// This is also useful for skipping "problematic" L1MessageTxs.
	queueIndex := tx.AsL1MessageTx().QueueIndex
	log.Trace("Circuit capacity limit reached for a single tx", "tx",
tx.Hash().String(), "queueIndex", queueIndex)
	log.Info("Skipping L1 message", "queueIndex", queueIndex, "tx",
tx.Hash().String(), "block", w.current.header.Number, "reason", "row consumption
overflow")
	w.current.nextL1MsgIndex = queueIndex + 1

	// after `ErrTxRowConsumptionOverflow`, ccc might not revert updates
	// associated with this transaction so we cannot pack more transactions.
	// TODO: fix this in ccc and change these lines back to `txs.Shift()`
	circuitCapacityReached = true
	break loop

case (errors.Is(err, circuitcapacitychecker.ErrTxRowConsumptionOverflow) &&
!tx.IsL1MessageTx()):
	// Circuit capacity check: L2MessageTx row consumption too high, skip the
account.
	// This is also useful for skipping "problematic" L2MessageTxs.
	log.Trace("Circuit capacity limit reached for a single tx", "tx",
tx.Hash().String())

	// after `ErrTxRowConsumptionOverflow`, ccc might not revert updates
```

```
        // associated with this transaction so we cannot pack more transactions.
        // TODO: fix this in ccc and change these lines back to `txs.Pop()`
        circuitCapacityReached = true
        break loop

case (errors.Is(err, circuitcapacitychecker.ErrUnknown) && tx.IsL1MessageTx()):
        // Circuit capacity check: unknown circuit capacity checker error for
L1MessageTx,
        // shift to the next from the account because we shouldn't skip the entire txs
from the same account
        queueIndex := tx.AsL1MessageTx().QueueIndex
        log.Trace("Unknown circuit capacity checker error for L1MessageTx", "tx",
tx.Hash().String(), "queueIndex", queueIndex)
        log.Info("Skipping L1 message", "queueIndex", queueIndex, "tx",
tx.Hash().String(), "block", w.current.header.Number, "reason", "unknown row
consumption error")
        w.current.nextL1MsgIndex = queueIndex + 1

        // after `ErrUnknown`, ccc might not revert updates associated
        // with this transaction so we cannot pack more transactions.
        // TODO: fix this in ccc and change these lines back to `txs.Shift()`
        circuitCapacityReached = true
        break loop

case (errors.Is(err, circuitcapacitychecker.ErrUnknown) && !tx.IsL1MessageTx()):
        // Circuit capacity check: unknown circuit capacity checker error for
L2MessageTx, skip the account
        log.Trace("Unknown circuit capacity checker error for L2MessageTx", "tx",
tx.Hash().String())

        // after `ErrUnknown`, ccc might not revert updates associated
        // with this transaction so we cannot pack more transactions.
        // TODO: fix this in ccc and change these lines back to `txs.Pop()`
        circuitCapacityReached = true
        break loop
```

Figure 1.1: Error handling for the circuit capacity checker (worker.go#L1073-L1121)

When this flag is set to `true`, no new transactions will be added even if there is room for additional transactions in the block (figure 1.2).

```
// Fill the block with all available pending transactions.
pending := w.eth.TxPool().Pending(true)
// Short circuit if there is no available pending transactions.
// But if we disable empty precommit already, ignore it. Since
// empty block is necessary to keep the liveness of the network.
if len(pending) == 0 && pendingL1Txs == 0 && atomic.LoadUint32(&w.noempty) == 0 {
        w.updateSnapshot()
        return
}
// Split the pending transactions into locals and remotes
localTxs, remoteTxs := make(map[common.Address]types.Transactions), pending
```

```
for _, account := range w.eth.TxPool().Locals() {
	if txs := remoteTxs[account]; len(txs) > 0 {
		delete(remoteTxs, account)
		localTxs[account] = txs
	}
}
var skipCommit, circuitCapacityReached bool
if w.chainConfig.Scroll.ShouldIncludeL1Messages() && len(l1Txs) > 0 {
	log.Trace("Processing L1 messages for inclusion", "count", pendingL1Txs)
	txs := types.NewTransactionsByPriceAndNonce(w.current.signer, l1Txs,
header.BaseFee)
	skipCommit, circuitCapacityReached = w.commitTransactions(txs, w.coinbase,
interrupt)
	if skipCommit {
		return
	}
}
if len(localTxs) > 0 && !circuitCapacityReached {
	txs := types.NewTransactionsByPriceAndNonce(w.current.signer, localTxs,
header.BaseFee)
	skipCommit, circuitCapacityReached = w.commitTransactions(txs, w.coinbase,
interrupt)
	if skipCommit {
		return
	}
}
if len(remoteTxs) > 0 && !circuitCapacityReached {
	txs := types.NewTransactionsByPriceAndNonce(w.current.signer, remoteTxs,
header.BaseFee)
	// don't need to get `circuitCapacityReached` here because we don't have
further `commitTransactions`
	// after this one, and if we assign it won't take effect (`ineffassign`)
	skipCommit, _ = w.commitTransactions(txs, w.coinbase, interrupt)
	if skipCommit {
		return
	}
}

// do not produce empty blocks
if w.current.tcount == 0 {
	return
}

w.commit(uncles, w.fullTaskHook, true, tstart)
```

Figure 1.2: Pending transactions are not added if the circuit capacity has been reached. (worker.go#L1284-L1332)

Exploit Scenario

Eve, an attacker, sends an L2 transaction that uses `ecrecover` many times. The transaction is provided to the mempool with enough gas to be the first L2 transaction in

the blockchain. Because this causes an error in the circuit checker, it prevents all other L2 transactions from being executed in this block.

Recommendations

Short term, implement a snapshotting mechanism in the circuit checker to roll back unexpected changes made as a result of incorrect or incomplete computation.

Long term, analyze and document all impacts of error handling across the system to ensure that these errors are handled gracefully. Additionally, clearly document all expected invariants of how the system is expected to behave to ensure that in interactions with other components, these invariants hold throughout the system.

2. Unused and dead code	
Severity: **Informational**	Difficulty: **N/A**
Type: Undefined Behavior	Finding ID: TOB-L2GETH-2
Target: Throughout the code	

Description

Due to the infrastructure setup of this network and the use of a single node clique setup, this fork of `geth` contains a significant amount of unused logic. Continuing to maintain this code can be problematic and may lead to issues.

The following are examples of unused and dead code:

- Uncle blocks—with a single node clique network, there is no chance for uncle blocks to exist, so all the logic that handles and interacts with uncle blocks can be dropped.
- Redundant logic around updating the L1 queue index
- A redundant check on empty blocks in the `worker.go` file

Recommendations

Short term, remove anything that is no longer relevant for the current `go-etheruem` implementation and be sure to document all the changes to the codebase.

Long term, remove all unused code from the codebase.

3. Lack of documentation

Severity: **Informational**	Difficulty: **N/A**
Type: Unexpected Behavior	Finding ID: TOB-L2GETH-3
Target: `miner/worker.go`	

Description
Certain areas of the codebase lack documentation, high-level descriptions, and examples, which makes the contracts difficult to review and increases the likelihood of user mistakes.

Areas that would benefit from being expanded and clarified in code and documentation include the following:

- **Internals of the CCC**. Despite being treated as a black box, the code relies on stateful changes made from `geth` calls. This suggests that the internal states of the miner's work and the CCC overlap. The lack of documentation regarding these states creates a lack of visibility in evaluating whether there are potential state corruptions or unexpected behavior.
- **Circumstances where transactions are skipped and how they are expected to be handled.** During the course of the review, we attempted to reverse engineer the intended behavior of transactions considered skipped by the CCC. The lack of documentation in these areas results in unclear expectations for this code.
- **Error handling standard throughout the system.** The codebase handles system errors differently—in some cases, logging an error and continuing execution or logging traces. Listing out all instances where errors are identified and documenting how they are handled can help ensure that there is no unexpected behavior related to error handling.

The documentation should include all expected properties and assumptions relevant to the aforementioned aspects of the codebase.

Recommendations
Short term, review and properly document the aforementioned aspects of the codebase. In addition to external documentation, NatSpec and inline code comments could help clarify complexities.

Long term, consider writing a formal specification of the protocol.

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.

- **Consider simplifying the switch case statements that check whether transactions are L1 messages or not.** The current error handling in `commitTransactions` makes the codebase hard to read.
- **Separate logic intended for production deployment from logic introduced for testing purposes.** For example, the `worker.go` file contains functions and `if` conditions for zero-period cliques or side-chain events, which are used only in testing. Removing these from the core codebase would make the contracts significantly more readable.
- **Fix the spelling error on the `ErrNonceTooHigh` log with the correct spelling of `height` instead of `hight`.** Correct spelling ensures that expected system behavior is clear.