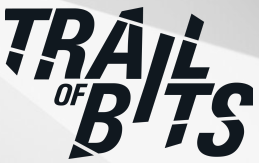

Scroll, l2geth

Security Assessment

September 12, 2023

Prepared for:
Haichen Shen
Scroll

Prepared by: **Benjamin Samuels and Damilola Edwards**

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

Executive Summary

Engagement Overview

Scroll engaged Trail of Bits to review the security of `l2geth`, Scroll's fork of `go-ethereum` (the Go implementation of the Ethereum protocol). The `l2geth` sequencer is an L2 transaction sequencer that collects L2 transactions, sequences them into blocks, and emits tracing information that a set of circuits can use to generate zero-knowledge proofs.

A team of two consultants conducted the review from July 10 to July 21, 2023, for a total of four engineer-weeks of effort. Our testing efforts focused on identifying any violated invariants and reviewing the modifications made to the original `go-ethereum` code to ensure that they do not introduce new security vulnerabilities. With full access to the source code and documentation, we performed static and dynamic testing of `l2geth`, using automated and manual processes.

Observations and Impact

Overall, we observed that in its current state, the codebase is relatively more immature than expected at this point in the development life cycle. The changes to `go-ethereum` contain many TODO statements, incorrectly handled errors, and MVP-specific configuration settings that are not relevant to the production network.

`l2geth` currently operates using a single, centralized sequencer in Clique consensus mode; however, using a single node in this consensus mode puts off-chain applications at risk of double-spend attacks if the sequencer is compromised (TOB-L2GETH-3).

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that the Scroll team take the following steps before deployment:

- **Remediate the findings disclosed in this report.** Various identified issues may lead to deviations from invariants established within the `l2geth` specification. These findings should be addressed as part of direct remediation or as part of any refactoring that may occur when addressing other recommendations.

- **Expand the existing test suite.** The Scroll team should take the following actions to expand the existing test suite: improve the test cases to cover a wider range of inputs and edge cases, design negative tests to assess error handling, use mutation testing to evaluate test effectiveness, create integration tests to validate component interactions, and deploy the node in a testnet environment for end-to-end validation. We anticipate that an expanded test suite will uncover additional issues

related to insufficient data validation or error handling, similar to those disclosed in this report.

- **Resolve all TODO statements currently in `l2geth`.** All TODO statements must be fully resolved before production deployment. TODO statements that cannot be resolved before production deployment should be captured as tickets/GitHub issues to ensure that they are prioritized and their risk is accounted for.

- **Expand `l2geth`'s CI pipeline.** The Scroll team should run `l2geth`'s full test suite in the CI pipeline for all pull requests. In addition, Go static analysis tools should be added to prevent common, easy-to-detect issues from being introduced into the codebase. Recommended static analysis tools include `go vet`, `staticcheck`, `ineffassign`, `errcheck`, `exportloopref`, and `gokart`.

- **Generate invalid execution traces and attempt to prove them.** `l2geth`'s execution traces are consumed by the circuits to generate zero-knowledge proofs of the execution, and if the trace is invalid or incorrect, the circuit should fail to generate a proof. Scroll should consider using a fuzzer to generate invalid execution traces to verify that the circuits will not generate a proof for an invalid trace.

- **Generate random transactions and attempt to prove them.** Scroll should generate random transactions, such as contract creation transactions and calls to certain functions, and verify that the circuits always generate a valid proof.

- **Create larger-scale testnet deployments and generate transactions on them.** Scroll should invest in a larger-scale testnet deployment that uses randomly submitted transactions, bridge operations, random starts and stops of the sequencer, and follower sync nodes. A more effective testnet strategy can help detect issues such as TOB-L2GETH-4, TOB-L2GETH-5, and TOB-L2GETH-6. Some existing tools such as tx-fuzz may be helpful for facilitating the described testnet.

Over the long term, Trail of Bits recommends that the Scroll team focus on the following initiatives to increase `l2geth`'s security maturity:

- **Remove unused code from `l2geth`.** The `go-ethereum` fork weighs in at roughly half a million lines of code and carries a large amount of technical debt. Much of this code is not necessary for `l2geth` and will actively make `l2geth` harder to maintain and audit as newer, more complex changes are made to the codebase. Scroll should consider a large-scale refactor of the codebase to remove unnecessary features and laser-focus the codebase on its targeted use case.

- **Invest in additional sequencer and node clients**. Over the long term, Scroll should invest in additional sequencer and node clients to help make the network more robust. Multiple client implementations enable differential fuzzing and other

advanced testing techniques that can drastically improve the system's security assurance. If an investment in a multi-client ecosystem is planned, it should be accompanied by a formal specification of the protocol to help identify the key invariants and properties of the system, laying the groundwork for property-based testing of each client.

The following tables provide the number of findings by severity and category:

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	1
Low	4
Informational	2
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	2
Denial of Service	2
Error Reporting	1
Undefined Behavior	2

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

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

The following engineers were associated with this project:

Benjamin Samuels, Consultant
benjamin.samuels@trailofbits.com

Damilola Edwards, Consultant
damilola.edwards@trailofbits.com

Project Timeline

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

Date	Event
July 5, 2023	Pre-project kickoff call
July 14, 2023	Status update meeting #1
July 24, 2023	Delivery of report draft
July 24, 2023	Report readout meeting
September 12, 2023	Delivery of comprehensive report

Project Goals

The engagement was scoped to provide a security assessment of Scroll's `l2geth`. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can a freshly synced follower node reach the same state root as the sequencer?
- Is there any way to bypass L1 or L2 fees when submitting L1 or L2 transactions?
- What kinds of new restrictions does `l2geth` apply to blocks and transactions, and are those restrictions enforced during block production and sync?
- Are there any downstream issues introduced by the use of Clique consensus for L2 transactions?
- Is there a way to force the sequencer to replay an L1 transaction?
- Does the L1 sync service correctly query the finalization condition for the L1 chain?
- Do any new features in `l2geth` introduce nondeterminism that may lend itself to a chain split or the inability to continue generating new blocks and advancing the chain?
- Were pre-existing providers within `go-ethereum` appropriately updated to handle any new changes introduced within `l2geth`?
- Do any of the changes to `go-ethereum` impact the determinism properties of consensus?

Project Targets

The engagement involved a review and testing of the following target.

l2geth

Repository	github.com/scroll-tech/go-ethereum
Version	`audit-v1@2dcc60a082ff89d1c57e497f23daad4823b2fdea`
Type	Go

Project Coverage

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

- **State account changes:** Two new fields, `PoseidonCodeHash` and `CodeSize`, were added to the state account object in `l2geth`. We manually investigated the implementation and use of these fields to ensure their correct functioning and relevance within the state account structure.
- **State and storage trie replacement:** We manually reviewed the replacement of the original Merkle Patricia Trie with zkTrie for both state and storage. We focused on cross-comparing the two implementations to verify the correct integration of zkTrie to maintain consistent data representation and retrieval.
- **EVM opcode modifications:** We manually reviewed changes to several EVM opcodes, including `BLOCKHASH`, `COINBASE`, `DIFFICULTY`, `PREVRANDAO`, `BASEFEE`, and `SELFDESTRUCT` to ensure accurate opcode behavior, appropriate handling of fee calculation, and consistent functionality across the codebase. Changes to the `DIFFICULTY` opcode were verified not to impact block headers.
- **Modifications to precompiled contracts:** We manually reviewed the modifications made to the precompiled contracts `modexp` and `ecPairing`. Additionally, we examined the disabling of the precompiled contracts `SHA2-256`, `RIPEMD-160`, and `blake2f`. We checked for proper input restrictions and usage limitations for these precompiled contracts to ensure the overall security and general correctness of their implementation.
- **L1 and L2 transaction fee calculation:** `l2geth` adds an additional fee payment mechanism to pay for L1 fees. We manually reviewed the L2 transaction fee calculation, including L2 execution fees, L1 data fees, and the use of the `L1GasPriceOracle` contract for the proper estimation of the L1 data fee based on transaction data. We searched for methods by which a transaction may avoid paying L1 or L2 fees, and we reviewed the various filtering mechanisms that `l2geth` uses to rank transactions.
- **`L1MessageTx`:** We manually reviewed the introduction of a new transaction type, `L1MessageTx`, for L1-initiated transactions. We reviewed the transaction payload definition, its behavior, and the implementation of the new block validation rules related to `L1MessageTx` transactions to ensure that the required front-of-block placement and matching QueueIndex order are properly implemented.

- **Consensus:** We manually reviewed the adoption of the Clique consensus mechanism to ensure that these changes were appropriately implemented and aligned with the specification. We assembled a list of new consensus rules introduced by `l2geth` and verified that `l2geth` checks for the rules during block construction and sync. The following consensus rules were identified and verified during block production and sync:
    - L1 transactions must be at the top of the block and must be contiguous.
    - L1 transactions must be ordered by queue index and be sequential to that of the prior block.
    - The maximum initialization code size is enforced.
    - The maximum block payload size is enforced.
    - L2 transaction gas payment must match that defined by the `VerifyFee()` function.
    - L1 transactions must be present on L1, and syncing nodes must not implicitly trust the sequencer.
    - The maximum number of L2 transactions per block is enforced.
- **L1 sync service:** We reviewed the L1 sync service, a component used to identify L1 and sync transactions from the base chain. We reviewed this component for general correctness and to ensure that it uses the correct definition of finality for a given base chain.
- **General logic:** We reviewed the components in scope for a number of common issues, using both manual and automated program analysis:
    - **Data manipulation:** We reviewed the data manipulation operations, such as copying, updating, and storing the transactions and block data, for correctness.
    - **Loop iterations:** We reviewed the loop iterations for correctness. We checked the loop conditions, indices, and break conditions to ensure they are bounded and accurate.
    - **Error handling:** We checked whether errors are handled properly and appropriate actions are taken.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- This review was conducted in the context of this pull request; any Geth code that is unrelated to the changes introduced in the `l2geth` fork falls outside the scope of this review. The evaluation was specifically focused on the modifications and additions made to the Geth codebase to implement the Scroll protocol and rollup features. As a result, code changes or functionalities not directly linked to these specific additions were not included in the review process.
- Many Merkle Patricia Trie properties that applied to the original `go-ethereum` implementation could not be verified in the zkTrie implementation, because the code is in a library and was out of scope.
- Mock implementations used in unit tests were not evaluated in depth to determine test efficacy.
- Risks of gas-related denial-of-service attacks were not fully evaluated during the assessment.
- Related Solidity smart contracts were out of scope.
- The implications of a trustless sequencer were not examined in depth because `l2geth` supports only centralized sequencing at this time.
- The `go-ethereum` fork is a relatively large codebase, and given the time-boxed nature of the audit and the deep changes made to its logic, there may be undiscovered bugs that may impact the security or liveness properties of the network.

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	We found that `l2geth` has issues with fully integrating fee modifications across the codebase, specifically in filtering/ordering code (TOB-L2GETH-1). At the time of the audit, the test suite had little-to-no arithmetic tests or negative tests for L1 transaction fees.	**Moderate**
Auditing	We found that the auditing and logging patterns used in `l2geth` closely resemble those of existing logging providers within `go-ethereum`. We identified no issues with the pattern of logging. Additional logging should be performed for increased verbosity, but for production, logging was found to be sufficient.	**Satisfactory**
Authentication / Access Controls	No meaningful changes to `go-ethereum`'s access controls were made in `l2geth`.	**Not Considered**
Complexity Management	`l2geth` builds on top of `go-ethereum`, a large and varied codebase that carries a large amount of technical debt. This technical debt makes it much harder to make sweeping architectural changes to the software without introducing hard-to-detect issues of varying severities (TOB-L2GETH-1, TOB-L2GETH-4). The technical debt carried over from `go-ethereum` makes it harder to review the `l2geth` code and to 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	**Weak**

	these changes much harder to detect against the background of historical `go-ethereum` quirks.	
Cryptography and Key Management	No meaningful changes to `go-ethereum`'s cryptography or key management were made in `l2geth`. We reviewed `l2geth`'s changes to the Merkle Patricia Trie only for code quality and high-level issues, as the core implementation for the zkTrie was out of scope for this review.	**Not Considered**
Decentralization	`l2geth` runs using a centralized sequencer, which represents a single point of failure. Using a single sequencer with Clique consensus can expose off-chain applications to double-spend attacks (TOB-L2GETH-3), which at this time cannot be mitigated unless off-chain applications manually query checkpoints on L1.	**Moderate**
Documentation	Scroll's public documentation is relatively thorough for an early-stage project and contains detailed information about the changes made to `go-ethereum`'s codebase; however, other categories of documentation are entirely missing: • Information on how to determine whether a transaction on L2 is final (required for safe exchange and bridge integration) • A specification detailing `l2geth`'s modifications in enough detail that a third-party developer could modify an Ethereum client and join the network as a follower ◦ Some of this information is already present in the documentation and simply needs to be expanded. • Information on how to run a follower node	**Moderate**
Transaction Ordering Risks	`l2geth` may have transaction-ordering risks due to how L1 transactions are ordered in a block. We were unable to examine these risks due to time constraints.	**Further Investigation Required**
Memory Safety and Error	While many of the unchecked errors described in finding TOB-L2GETH-2 are from the original	**Moderate**

Handling	`go-ethereum` code, we found that several issues stem from `l2geth`'s changes, including instances in which `panic` is used instead of Go's error management system (TOB-L2GETH-4) and an instance in which no error management is used at all—the program simply dereferences a pointer (TOB-L2GETH-5).	
Low-Level Manipulation	`l2geth`'s modifications do not introduce any low-level manipulation.	**Not Considered**
Testing and Verification	While the unit test coverage is decent, many of the tests are inherited from `go-ethereum` and may not be adequate to test the new features (even once they are modified). Many findings could have been prevented with a more comprehensive test suite (TOB-L2GETH-1, TOB-L2GETH-7). Multiple unit tests were not working at the time of the review, but they have been fixed in a newer branch.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Transaction pool fails to drop transactions that cannot afford L1 fees	Data Validation	**Informational**
2	Multiple instances of unchecked errors	Undefined Behavior	**Low**
3	Risk of double-spend attacks due to use of single-node Clique consensus without finality API	Undefined Behavior	**Medium**
4	Improper use of panic	Denial of Service	**Low**
5	Risk of panic from nil dereference due to flawed error reporting in addressToKey	Error Reporting	**Informational**
6	Risk of transaction pool admission denial of service	Denial of Service	**Low**
7	Syncing nodes fail to check consensus rule for L1 message count	Data Validation	**Low**

Detailed Findings

1. Transaction pool fails to drop transactions that cannot afford L1 fees	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-L2GETH-1
Target: `core/types/transaction.go`	

Description

`l2geth` defines two fees that must be paid for L2 transactions: an L2 fee and an L1 fee. However, the code fails to account for L1 fees; as a result, transactions that cannot afford the combined L1 and L2 fees may be included in a block rather than demoted, as intended.

The `transaction.go` file defines a `Cost()` function that returns the amount of ether that a transaction consumes, as shown in figure 1.1. The current implementation of `Cost()` does not account for L1 fees, causing other parts of the codebase to misjudge the balance requirements to execute a transaction. The correct implementation of `Cost()` should match the implementation of `VerifyFee()`, which correctly checks for L1 fees.

```
// Cost returns gas * gasPrice + value.
func (tx *Transaction) Cost() *big.Int {
	total := new(big.Int).Mul(tx.GasPrice(), new(big.Int).SetUint64(tx.Gas()))
	total.Add(total, tx.Value())
	return total
}
```

Figure 1.1: The `Cost()` function does not include L1 fees in its calculation. (go-ethereum/core/types/transaction.go#318–323)

Most notably, `Cost()` is consumed by the `tx_list.Filter()` function, which is used to prune un-executable transactions (transactions that cannot afford the fees), as shown in figure 1.2. The failure to account for L1 fees in `Cost()` could cause `tx_list.Filter()` to fail to demote such transactions, causing them to be incorrectly included in the block.

```
func (l *txList) Filter(costLimit *big.Int, gasLimit uint64) (types.Transactions,
types.Transactions) {
	// If all transactions are below the threshold, short circuit
	if l.costcap.Cmp(costLimit) <= 0 && l.gascap <= gasLimit {
		return nil, nil
	}
	l.costcap = new(big.Int).Set(costLimit) // Lower the caps to the thresholds
```

```
	l.gascap = gasLimit

	// Filter out all the transactions above the account's funds
	removed := l.txs.Filter(func(tx *types.Transaction) bool {
		return tx.Gas() > gasLimit || tx.Cost().Cmp(costLimit) > 0
	})
```

Figure 1.2: `Filter()` uses `Cost()` to determine which transactions to demote. (go-ethereum/core/tx_list.go#332–343)

Exploit Scenario

A user creates an L2 transaction that can just barely afford the L1 and L2 fees in the next upcoming block. Their transaction is delayed due to full blocks and is included in a future block in which the L1 fees have risen. Their transaction reverts due to the increased L1 fees instead of being ejected from the transaction pool.

Recommendations

Short term, refactor the `Cost()` function to account for L1 fees, as is done in the `VerifyFee()` function; alternatively, have the transaction list structure use `VerifyFee()` or a similar function instead of `Cost()`.

Long term, add additional tests to verify complex state transitions such as a transaction becoming un-executable due to changes in L1 fees.

2. Multiple instances of unchecked errors

Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-L2GETH-2
Targets: `trie/zkproof/writer.go`, `trie/sync.go`, `trie/proof.go`, `trie/committer.go`, `trie/database.go`	

Description

There are multiple instances of unchecked errors in the `l2geth` codebase, which could lead to undefined behavior when errors are raised. One such unhandled error is shown in figure 2.1. A comprehensive list of unchecked errors is provided in appendix C.

```
if len(requests) == 0 && req.deps == 0 {
        s.commit(req)
} else {
```

Figure 2.1: The Sync.commit() function returns an error that is unhandled, which could lead to invalid commitments or a frozen chain. (go-ethereum/trie/sync.go#296–298)

Unchecked errors also make the system vulnerable to denial-of-service attacks; they could allow attackers to trigger `nil` dereference panics in the sequencer node.

Exploit Scenario

An attacker identifies a way to cause a zkTrie commitment to fail, allowing invalid data to be silently committed by the sequencer.

Recommendations

Short term, add error checks to all functions that can emit Go errors.

Long term, add the tools `errcheck` and `ineffassign` to `l2geth`'s build pipeline. These tools can be used to detect errors and prevent builds containing unchecked errors from being deployed.

3. Risk of double-spend attacks due to use of single-node Clique consensus without finality API

Severity: **Medium**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-L2GETH-3
Target: N/A	

Description

`l2geth` uses the proof-of-authority Clique consensus protocol, defined by EIP-255. This consensus type is not designed for single-node networks, and an attacker-controlled sequencer node may produce multiple conflicting forks of the chain to facilitate double-spend attacks.

The severity of this finding is compounded by the fact that there is no API for an end user to determine whether their transaction has been finalized by L1, forcing L2 users to use ineffective block/time delays to determine finality.

Clique consensus was originally designed as a replacement for proof-of-work consensus for Ethereum testnets. It uses the same fork choice rule as Ethereum's proof-of-work consensus; the fork with the highest "difficulty" should be considered the canonical fork. Clique consensus does not use proof-of-work and cannot update block difficulty using the traditional calculation; instead, block difficulty may be one of two values:

- "1" if the block was mined by the designated signer for the block height
- "2" if the block was mined by a non-designated signer for the block height

This means that in a network with only one authorized signer, all of the blocks and forks produced by the sequencer will have the same difficulty value, making it impossible for syncing nodes to determine which fork is canonical at the given block height.

In a normal proof-of-work network, one of the proposed blocks will have a higher difficulty value, causing syncing nodes to re-organize and drop the block with the lower difficulty value. In a single-validator proof-of-authority network, neither block will be preferred, so each syncing node will simply prefer the first block they received.

This finding is not unique to `l2geth`; it will be endemic to all L2 systems that have only one authorized sequencer.

Exploit Scenario

An attacker acquires control over `l2geth`'s centralized sequencer node. The attacker modifies the node to prove two forks: one fork containing a deposit transaction to a centralized exchange, and one fork with no such deposit transaction. The attacker publishes the first fork, and the centralized exchange picks up and processes the deposit transaction. The attacker continues to produce blocks on the second private fork. Once the exchange processes the deposit, the attacker stops generating blocks on the public fork, generates an extra block to make the private fork longer than the public fork, then publishes the private fork to cause a re-organization across syncing nodes. This attack must be completed before the sequencer is required to publish a proof to L1.

Recommendations

Short term, add API methods and documentation to ensure that bridges and centralized exchanges query only for transactions that have been proved and finalized on the L1 network.

Long term, decentralize the sequencer in such a way that a majority of sequencers must collude in order to successfully execute a double-spend attack. This design should be accompanied by a slashing mechanism to penalize sequencers that sign conflicting blocks.

4. Improper use of panic

Severity: **Low**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-L2GETH-4
Target: Various areas of the codebase	

Description

`l2geth` overuses Go's `panic` mechanism in lieu of Go's built-in error propagation system, introducing opportunities for denial of service.

Go has two primary methods through which errors can be reported or propagated up the call stack: the `panic` method and Go errors. The use of `panic` is not recommended, as it is unrecoverable: when an operation panics, the Go program is terminated and must be restarted. The use of `panic` creates a denial-of-service vector that is especially applicable to a centralized sequencer, as a restart of the sequencer would effectively halt the L2 network until the sequencer recovers.

Some example uses of `panic` are presented in figures 4.1 to 4.3. These do not represent an exhaustive list of `panic` statements in the codebase, and the Scroll team should investigate each use of `panic` in its modified code to verify whether it truly represents an unrecoverable error.

```
func sanityCheckByte32Key(b []byte) {
	if len(b) != 32 && len(b) != 20 {
		panic(fmt.Errorf("do not support length except for 120bit and 256bit
now. data: %v len: %v", b, len(b)))
	}
}
```

Figure 4.1: The `sanityCheckByte32Key` function panics when a trie key does not match the expected size. This function may be called during the execution of certain RPC requests. (`go-ethereum/trie/zk_trie.go#44–48`)

```
func (s *StateAccount) MarshalFields() ([]zkt.Byte32, uint32) {
	fields := make([]zkt.Byte32, 5)

	if s.Balance == nil {
		panic("StateAccount balance nil")
	}

	if !utils.CheckBigIntInField(s.Balance) {
		panic("StateAccount balance overflow")
```

```
	}

	if !utils.CheckBigIntInField(s.Root.Big()) {
		panic("StateAccount root overflow")
	}

	if !utils.CheckBigIntInField(new(big.Int).SetBytes(s.PoseidonCodeHash)) {
		panic("StateAccount poseidonCodeHash overflow")
	}
```

Figure 4.2: The `MarshalFields` function panics when attempting to marshal an object that does not match certain requirements. This function may be called during the execution of certain RPC requests. (go-ethereum/core/types/state_account_marshalling.go#47–64)

```
func (t *ProofTracer) MarkDeletion(key []byte) {
	if path, existed := t.emptyTermPaths[string(key)]; existed {
		// copy empty node terminated path for final scanning
		t.rawPaths[string(key)] = path
	} else if path, existed = t.rawPaths[string(key)]; existed {
		// sanity check
		leafNode := path[len(path)-1]

		if leafNode.Type != zktrie.NodeTypeLeaf {
			panic("all path recorded in proofTrace should be ended with
leafNode")
		}
```

Figure 4.3: The `MarkDeletion` function panics when the proof tracer contains a path that does not terminate in a leaf node. This function may be called when a syncing node attempts to process an invalid, malicious proof that an attacker has gossiped on the network. (go-ethereum/trie/zktrie_deletionproof.go#120–130)

Exploit Scenario

An attacker identifies an error path that terminates with a panic that can be triggered by a malformed RPC request or proof payload. The attacker leverages this issue to either disrupt the sequencer's operation or prevent follower/syncing nodes from operating properly.

Recommendations

Short term, review all uses of `panic` that have been introduced by Scroll's changes to `go-ethereum`. Ensure that these uses of `panic` truly represent unrecoverable errors, and if not, add error handling logic to recover from the errors.

Long term, annotate all valid uses of `panic` with explanations for why the errors are unrecoverable and, if applicable, how to prevent the unrecoverable conditions from being triggered. `l2geth`'s code review process must also be updated to verify that this documentation exists for new uses of `panic` that are introduced later.

5. Risk of panic from nil dereference due to flawed error reporting in addressToKey

Severity: **Informational**	Difficulty: **N/A**
Type: Error Reporting	Finding ID: TOB-L2GETH-5
Target: `trie/zkproof/writer.go`	

Description

The `addressToKey` function, shown in figure 5.1, returns a `nil` pointer instead of a Go error when an error is returned by the `preImage.Hash()` function, which will cause a `nil` dereference panic in the `NewZkTrieProofWriter` function, as shown in figure 5.2.

If the error generated by `preImage.Hash()` is unrecoverable, the `addressToKey` function should panic instead of silently returning a `nil` pointer.

```
func addressToKey(addr common.Address) *zkt.Hash {
	var preImage zkt.Byte32
	copy(preImage[:], addr.Bytes())

	h, err := preImage.Hash()
	if err != nil {
		log.Error("hash failure", "preImage", hexutil.Encode(preImage[:]))
		return nil
	}
	return zkt.NewHashFromBigInt(h)
}
```

Figure 5.1: The `addressToKey` function returns a `nil` pointer to `zkt.Hash` when an error is returned by `preImage.Hash()`. (go-ethereum/trie/zkproof/writer.go#31–41)

```
func NewZkTrieProofWriter(storage *types.StorageTrace) (*zktrieProofWriter, error) {

	underlayerDb := memorydb.New()
	zkDb := trie.NewZktrieDatabase(underlayerDb)

	accounts := make(map[common.Address]*types.StateAccount)

	// resuming proof bytes to underlayerDb
	for addrs, proof := range storage.Proofs {
		if n := resumeProofs(proof, underlayerDb); n != nil {
			addr := common.HexToAddress(addrs)
			if n.Type == zktrie.NodeTypeEmpty {
				accounts[addr] = nil
			} else if acc, err := types.UnmarshalStateAccount(n.Data()); err
```

```
== nil {
                        if bytes.Equal(n.NodeKey[:], addressToKey(addr)[:]) {
                                accounts[addr] = acc
```

Figure 5.2: The `addressToKey` function is consumed by `NewZkTrieProofWriter`, which will attempt to dereference the `nil` pointer and generate a system panic. (go-ethereum/trie/zkproof/writer.go#152–167)

Recommendations

Short term, modify `addressToKey` so that it either returns an error that its calling functions can propagate or, if the error is unrecoverable, panics instead of returning a `nil` pointer.

Long term, update Scroll's code review and style guidelines to reflect that errors must be propagated by Go's error system or must halt the program by using `panic`.

6. Risk of transaction pool admission denial of service

Severity: **Low**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-L2GETH-6
Target: `core/tx_pool.go`	

Description

`l2geth`'s changes to the transaction pool include an ECDSA recovery operation at the beginning of the pool's transaction validation logic, introducing a denial-of-service vector: an attacker could generate invalid transactions to exhaust the sequencer's resources.

`l2geth` adds an L1 fee that all L2 transactions must pay. To verify that an L2 transaction can afford the L1 fee, the transaction pool calls `fees.VerifyFee()`, as shown in figure 6.1. `VerifyFee()` performs an ECDSA recovery operation to determine the account that will pay the L1 fee, as shown in figure 6.2.

ECDSA key recovery is an expensive operation that should be executed as late in the transaction validation process as possible in order to reduce the impact of denial-of-service attacks.

```
func (pool *TxPool) add(tx *types.Transaction, local bool) (replaced bool, err
error) {
	// If the transaction is already known, discard it
	hash := tx.Hash()
	if pool.all.Get(hash) != nil {
		log.Trace("Discarding already known transaction", "hash", hash)
		knownTxMeter.Mark(1)
		return false, ErrAlreadyKnown
	}
	// Make the local flag. If it's from local source or it's from the network
but
	// the sender is marked as local previously, treat it as the local
transaction.
	isLocal := local || pool.locals.containsTx(tx)

	if pool.chainconfig.Scroll.FeeVaultEnabled() {
		if err := fees.VerifyFee(pool.signer, tx, pool.currentState); err !=
nil {
```

Figure 6.1: `TxPool.add()` calls `fees.VerifyFee()` before any other transaction validators are called. (go-ethereum/core/tx_pool.go#684–697)

```
func VerifyFee(signer types.Signer, tx *types.Transaction, state StateDB) error {
```

```
from, err := types.Sender(signer, tx)
if err != nil {
        return errors.New("invalid transaction: invalid sender")
}
```

Figure 6.2: `VerifyFee()` initiates an ECDSA recovery operation via `types.Sender()`. (go-ethereum/rollup/fees/rollup_fee.go#198–202)

Exploit Scenario

An attacker generates a denial-of-service attack against the sequencer by submitting extraordinarily large transactions. Because ECDSA recovery is a CPU-intensive operation and is executed before the transaction size is checked, the attacker is able to exhaust the memory resources of the sequencer.

Recommendations

Short term, modify the code to check for L1 fees in the `TxPool.validateTx()` function immediately after that function calls `types.Sender()`. This will ensure that other, less expensive-to-check validations are performed before the ECDSA signature is recovered.

Long term, exercise caution when making changes to code paths that validate information received from public sources or gossip. For changes to the transaction pool, a good general rule of thumb is to validate transaction criteria in the following order:

1. Simple, in-memory criteria that do not require disk reads or data manipulation
2. Criteria that require simple, in-memory manipulations of the data such as checks of the transaction size
3. Criteria that require an in-memory state trie to be checked
4. ECDSA recovery operations
5. Criteria that require an on-disk state trie to be checked

However, note that sometimes these criteria must be checked out of order; for example, the ECDSA recovery operation to identify the origin account may need to be performed before the state trie is checked to determine whether the account can afford the transaction.

7. Syncing nodes fail to check consensus rule for L1 message count

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-L2GETH-7
Target: `core/block_validator.go`	

Description

`l2geth` adds a consensus rule requiring that there be no more than `L1Config.NumL1MessagesPerBlock` number of L1 messages per L2 block. This rule is checked by the sequencer when building new blocks but is not checked by syncing nodes through the `ValidateL1Messages` function, as shown in figure 7.1.

```
// TODO: consider adding a rule to enforce L1Config.NumL1MessagesPerBlock.
// If there are L1 messages available, sequencer nodes should include them.
// However, this is hard to enforce as different nodes might have different views of
L1.
```

Figure 7.1: The `ValidateL1Messages` function does not check the `NumL1MessagesPerBlock` restriction. (`go-ethereum/core/block_validator.go#145–147`)

The TODO comment shown in the figure expresses a concern that syncing nodes cannot enforce `NumL1MessagesPerBlock` due to the different view of L1 that the nodes may have; however, this issue does not prevent syncing nodes from simply checking the number of L1 messages included in the block.

Exploit Scenario

A malicious sequencer ignores the `NumL1MessagesPerBlock` restriction while constructing a block, thus bypassing the consensus rules. Follower nodes consider the block to be valid even though the consensus rule is violated.

Recommendations

Short term, add a check to `ValidateL1Messages` to check the maximum number of L1 messages per block restriction.

Long term, document and check all changes to the system's consensus rules to ensure that both nodes that construct blocks and nodes that sync blocks check the consensus rules. This includes having syncing nodes check whether an L1 transaction actually exists on the L1, a concern expressed in comments further up in the `ValidateL1Messages` function.

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
Transaction Ordering Risks	The system's resistance to transaction re-ordering attacks
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
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. Unchecked Errors List

The following table lists the various unhandled Go errors, as reported in finding TOB-L2GETH-2.

Location	Functions with Unchecked Errors
trie/zkproof/writer.go, line 226	underlayerDb.Put()
trie/zkproof/writer.go, line 57	db.Put()
trie/sync.go, line 297	s.commit()
trie/sync.go, line 279	s.commit()
trie/proof.go, line 567	tr.TryUpdate()
trie/proof.go, line 497	tr.TryUpdate()
trie/proof.go, line 86	proofDb.Put()
trie/committer.go, line 229	c.onleaf()
trie/committer.go, line 235	c.onleaf()
trie/committer.go, line 245	c.sha.Write()
trie/committer.go, line 256	c.sha.Read()
trie/database.go, line 581	db.preimages.commit()
trie/database.go, line 659	batch.Put()
trie/database.go, line 676	db.preimages.commit()
trie/database.go, line 695	batch.Replay()
trie/database.go, line 743	batch.Replay()
trie/database.go, line 850	db.saveCache()

D. Code Quality Findings

This appendix contains findings that do not have immediate or obvious security implications. However, they may facilitate exploit chains targeting other vulnerabilities or become easily exploitable in future releases.

- **Avoid the use of magic values.** The use of hard-coded magic values within code introduces ambiguity because they lack clear context, making it challenging for developers/auditors to understand their purpose and significance. By using variables or constants, meaningful names can be provided to these values, offering self-documentation and improving code comprehension.

```
func toWordSize(size uint64) uint64 {
	if size > math.MaxUint64-31 {
		return math.MaxUint64/32 + 1
	}

	return (size + 31) / 32
}
```

Figure D.1: go-ethereum/core/state_transition.go#173–179

```
func IterateL1MessagesFrom(db ethdb.Iteratee, fromQueueIndex uint64)
L1MessageIterator {
	start := encodeQueueIndex(fromQueueIndex)
	it := db.NewIterator(l1MessagePrefix, start)
	keyLength := len(l1MessagePrefix) + 8
```

Figure D.2: go-ethereum/core/rawdb/accessors_l1_message.go#108–112

- **Remove redundant code.** Clean up the redundant code related to handling uncle blocks and forks, as the codebase does not recognize or produce uncle blocks. Removing this unnecessary functionality will simplify the code, making it easier to understand and maintain.

- **Address TODO statements or create tickets for them.** TODO statements should remain in a codebase only while the codebase is in a pre-release state. Address TODO statements that can be addressed immediately; for those that cannot be addressed before production deployment, create tickets to ensure that they are prioritized.

- **Use feature flags to control network behavior instead of using reflection or magic hashes.** Some features in `l2geth` use reflection or the presence of a specific hash in a database to determine whether certain features are enabled, as in the `VerifyProof` function, shown in figure D.3. Feature flags provide a better, more unified interface to enable or disable specific features or functionality. If a function

needs to operate in multiple, simultaneous modes, an encapsulating function should be used to make it clear that the behavior is not specific to a network feature.

```
func VerifyProof(rootHash common.Hash, key []byte, proofDb ethdb.KeyValueReader)
(value []byte, err error) {

	// test the type of proof (for trie or SMT)
	if buf, _ := proofDb.Get(magicHash); buf != nil {
		return VerifyProofSMT(rootHash, key, proofDb)
	}

	key = keybytesToHex(key)
	wantHash := rootHash
	for i := 0; ; i++ {
```

Figure D.3: The `VerifyProof` function uses the presence of a magic hash to determine what kind of proof to verify. (`go-ethereum/trie/proof.go#106–115`)