HACKEN

gcEVM-Based Blockchain Protocol Security Analysis Report

Customer: COTI and Soda Labs

Date: 12/08/2025

We express our gratitude to the COTI and Soda Labs team for the collaborative engagement that enabled the execution of this Blockchain Protocol Security Assessment.

gcEVM is a solution designed to enhance Ethereum-compatible environments by integrating privacy-preserving features through Secure Multiparty Computation. It ensures encrypted, privacy-preserving computations while supporting seamless integration with existing Ethereum-based applications. Developers can utilize gcEVM to create privacy-focused decentralized applications by leveraging new data types and security measures without compromising performance.

Document

Name	gcEVM-Based Blockchain Protocol Review and Security Analysis Report for COTI and Soda Labs
Audited By	Nino Lipartiia, Tanuj Soni, Hamza Sajid
Approved By	Stepan Chekhovskoi
Website	https://www.sodalabs.xyz/ and https://coti.io/
Language	Golang
Tags	Go-Ethereum, EVM, MPC
Methodology	https://hackenio.cc/blockchain_methodology

Review Scope

Repository	https://github.com/soda-mpc/go-ethereum
Review Commit	c53489958e34ca2dcc57bf5e8246aa0ca9015a88
Commit After Fixes	76a7d1eecac00143ba28df45e4ccae92dfaa0739

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

15	9	4	2
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	1
Medium	1
Low	8

Vulnerability	Severity
F-2025-12034 - Executor Panics on Sequencer Transactions Exceeding Gas Limits	High
F-2025-9399 - Gas Cost Discrepancy for Several Secure Operations	Medium
F-2025-9028 - Inadequate Handling of gRPC Errors	Low
F-2025-9034 - Vulnerabilities in Go External Dependencies & Standard Library	Low
F-2025-9036 - Deprecated & Insecure Functions in Key Operations	Low
F-2025-9180 - Unsafe CGO Integration in Consensus Mechanism	Low
F-2025-9212 - Unreliable Signature Verification Process	Low
F-2025-9214 - Unnecessary Insertion to Authenticated Memory After offBoardToUser	Low
F-2025-9294 - Centralized Token Emission Distribution Creates Critical Single Point of Failure	Low
F-2025-9403 - Questionable Validation of SodaMPCExecutionError	Low
F-2025-12033 - Missing gcEVM Context Initialization Reduces Tracing Reliability	Info
F-2025-9027 - Absence of Authenticated Memory Emptiness Validation Upon EVM Depth Increase	Info
F-2025-9035 - Errors Identified in Unit Tests Compilation	Info
F-2025-9211 - Eliminate Unused Variables in common Package	Info
F-2025-9222 - Duplicate Verification Calls in Executor Block Insertion	Info

Documentation quality

- Comprehensive documentation is provided, including a white paper.
- Additional documents covering transaction flow, block propagation, and minting and inflation are present.
- Module-level documentation clearly explains the functionality of each component.
- The codebase includes helpful comments to improve clarity.
- While build instructions were not available in the repository for security reasons, they were supplied for audit testing purposes.

Code quality

- The codebase maintains high quality standards.
- Tests were failing in the initial audit review commit but were subsequently fixed during the remediation phase.
- Comprehensive end-to-end tests are in place to improve coverage.
- The code contains several unresolved "SODO" (TODO) items that, while not urgent, highlight areas for future refinement.

Architecture quality

- Built upon the reliable Go-Ethereum source code, providing a strong foundation.
- Implements a design for private data management and secure computation on ciphertexts without prior decryption, resembling homomorphic encryption but utilizing a Garbled Circuit-based MPC protocol.
- Modifications to the EVM are implemented as a precompiled contract.
- Security is enhanced through custom-designed authenticated memory and storage.
- Transaction processing and block production are handled by a sequencer and two executors, while validators are responsible for block validation.
- The gcEVM network uses a custom Proof of Authority (PoA) consensus called CO2, ensuring secure and efficient transaction processing.
- The current model is centralized, with plans for decentralization in the future.

Table of Contents

System Overview

Soda gcEVM is a modified Geth fork designed to enhance confidentiality and security in blockchain operations. It incorporates significant changes to the consensus layer, network layer, and virtual machine (VM).

Node Types and Roles

The Soda gcEVM network consists of three distinct node types, each fulfilling specific roles:

- **Sequencer**: Responsible for managing the mempool and generating the "red block," which includes valid transactions, some involving private data. It validates non-private transactions and sends the red block to executors.
- **Executors**: Validate the red block and perform privacy operations via Multi-Party Computation (MPC). Upon success, they create the "black block" and forward it to both the sequencer and validators.
- **Validators**: Validate the "black block" to ensure the block's correctness before finalizing the process.

Key Components and Modifications

- **Consensus (CO2)**: A customized version of the Clique consensus mechanism, introducing the new roles of Executor and Sequencer to enhance functionality.
- **SES**: A proprietary library that utilizes CGO to invoke C code directly from Go, playing a pivotal role in the consensus mechanism.
- **Soda-MPC**: Implements Multi-Party Computation (MPC) with garbled circuits to secure data privacy across the network. Though outside the scope of this audit, it integrates secure operations as an EVM extension.
- **EVM Extension**: Enhances the virtual machine layer with a precompiled contract that integrates MPC computation for secure operations.

Risks

- **Single Point of Failure**: If either the sequencer or one of the executors misbehaves or disconnects, the entire chain will halt. While the design prevents invalid transactions, this centralization introduces significant risks to system stability and availability.
- **Centralization Risk**: The system's architecture is highly centralized, relying on a single sequencer and two executors, all managed by trusted network parties. The development team has indicated plans to modify the architecture to achieve greater distribution and decentralization, which may mitigate these risks in the future.

Findings

Vulnerability Details

F-2025-12034 - Executor Panics on Sequencer Transactions Exceeding Gas Limits - High

Description:

An issue was identified by the Soda team in the executor's handling of sequencer-submitted block transactions.

The root cause lies in the `fillOrderedBlockTxs` method, which continues processing and sorting transactions even when individual transactions fail validation due to insufficient gas:

```
for _, tx := range txs {
log.Warn("transaction started")
if env.gasPool.Gas() < params.TxGas {
log.Warn("Not enough gas for further transactions", "have", env.gas
Pool, "want", params.TxGas)
// This is a temporary solution, this means the transaction is not
included in the Black block even
// though it was sequenced as part of the Red block. In the future
we should take steps to avoid this
// discrepancy.
continue
}
// If we don't have enough space for the next transaction, skip the
account.
if env.gasPool.Gas() < tx.Gas() {
log.Warn("Not enough gas left for transaction", "hash", tx.Hash, "l
eft", env.gasPool.Gas(), "needed", tx.Gas)
// This is a temporary solution, this means the transaction is not
included in the Black block even
// though it was sequenced as part of the Red block. In the future
we should take steps to avoid this
// discrepancy.
continue
}
if left := uint64(params.MaxBlobGasPerBlock - env.blobs*params.Blob
TxBlobGasPerBlob); left < tx.BlobGas() {
log.Warn("Not enough blob gas left for transaction", "hash", tx.Has
h, "left", left, "needed", tx.BlobGas)
continue
}
...
}
```

This behavior introduces risk, especially in private computation environments where block execution semantics assume that earlier transactions are valid and executable. Preserving transaction order is critical in such contexts. However, the current logic allows the executor to emit a fragmented transaction list, skipping invalid transactions instead of halting and rejecting the following transaction. This may break assumptions relied upon by downstream execution or state transition logic.

In addition, if `fillOrderedBlockTxs` encounters a failure during `commitTransaction`, the error propagates to `commitExecutorWork`, which can trigger a panic:

```
func (w *worker) mainLoop() {
...
err := w.commitExecutorWork(block)
if err != nil {
log.Warn("Error committing executor work", "error", err)
if !errors.Is(err, core.ErrMPCRestartBlock) {
panic(fmt.Sprintf("Extremely bad! error: %s", err.Error()))
}
}
...
}
```

This panic initiates the mining process and stalls the chain, potentially halting block production until manual recovery.

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 4/5

Severity: High

Recommendations

Remediation: The following actions are recommended:

- Handle `commitTransaction` errors gracefully to prevent triggering a panic in `commitExecutorWork`.
- Ensure the sequencer constructs blocks using the same transaction validation rules and gas calculations as the executor, to avoid execution discrepancies.

While updating `fillOrderedBlockTxs` to halt processing on transaction failure can prevent fragmented transaction lists, the core security concern lies in proper error handling. Panics during execution must be avoided to maintain node stability and protect the integrity of the consensus process.

Additionally, consider performing gas and blob gas checks prior to transaction ordering, so that invalid or unexecutable transactions are excluded early. This further strengthens the executor's resilience and consistency.

These changes will help maintain transaction list integrity, preserve correct execution order, and prevent chain stalls or unintended halts in block production.

Resolution: The issue was fixed in PR#329 and merged in commit 58d88d9238ea8e109f5fc1c85e7ac599c872d30a.

The fix modifies the `fillOrderedBlockTxs` logic to stop adding new transactions as soon as any transaction fails. Error handling was also improved by correctly reporting the `core.ErrMPCRestartBlock` error, preventing a panic in the streamline processing path.

The explicit `panic` in `commitExecutorWork` remains unchanged. The client confirmed that it is still appropriate for true logic or invariant violations where producing an invalid block is unacceptable.

While there is no dedicated recovery mechanism, the team described the current behavior as follows:

> The recovery mechanism now works as follows:
> - **Non-`ErrMPCRestartBlock` errors:** The transaction that failed is excluded from the current black block, and processing stops there. The block is sealed with all transactions up to that point. The failed transaction is returned to the transaction pool and will be retried in a future block.
> - `ErrMPCRestartBlock` **errors:** The error is propagated, triggering a block discard and executor restart. This is intentional because these are execution-level restart conditions, not logic errors.
> - In effect, failed transactions that would have caused nonce gaps (like in the reproduction scenario) will now be automatically retried later, and the node no longer crashes in these cases.

F-2025-9399 - Gas Cost Discrepancy for Several Secure Operations - Medium

Description:

Several operations' gas requirements differ significantly from the documented and expected values. Specifically, it is anticipated that the `GetUserKey` operation has a gas cost of 47,039, and the `Shl` and `Shr` operations have costs defined as follows:

Operation	gtUint8	gtUint16	gtUint32	gtUint64
`Shl`	12,620	14,571	22,467	54,233
`Shr`	12,969	15,960	28,009	76,377

However, the current implementation assigns these values differently:

core/vm/soda_extension_mpc.go:148-152, 284-292:

```
// requiredGasMap maps a function signature along with the number o
f bits to the corresponding required gas value.
requiredGasMap = map[struct {
signature uint32
bits int
}]uint64{
// ...
{signature: signatureGetKey}: 200,
{signature: signatureShl, bits: 8}: 100,
{signature: signatureShl, bits: 16}: 100,
{signature: signatureShl, bits: 32}: 100,
{signature: signatureShl, bits: 64}: 100,
{signature: signatureShr, bits: 8}: 100,
{signature: signatureShr, bits: 16}: 100,
{signature: signatureShr, bits: 32}: 100,
{signature: signatureShr, bits: 64}: 100,
// ...
}
```

This significant discrepancy may pose a security risk by enabling the creation of transactions that exploit these underpriced operations repeatedly, potentially overwhelming the system and leading to potential halts.

Assets:

- gcEVM

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 3/5

Severity: Medium

Recommendations

Remediation: To mitigate potential security risks, it is advisable to reevaluate and adjust the gas costs associated with the `GetUserKey`, `Shl`, and `Shr` operations to align with their documented and expected values. This adjustment will prevent the exploitation of underpriced operations that could overwhelm the system.

Resolution: The issue was fixed at https://github.com/soda-mpc/go-ethereum/pull/296 by assigning correct amount of gas to `GetUserKey`, `Shl`, and `Shr` operations in the `requiredGasMap.`

F-2025-9028 - Inadequate Handling of gRPC Errors - Low

Description:

In the implementation of the `runOpcode` function, there is a risk that an error returned from a gRPC call may be omitted. Specifically, the returned error can be:

core/vm/soda_extension_mpc.go:775:

```
return nil, MPCExecutionResult[st.Message()]
```

where `MPCExecutionResult` is defined as

core/vm/soda_extension_mpc.go:328:

```
MPCExecutionResult = map[string]error{
"CONNECTION_ERROR": ErrMPCConnection,
"EXECUTION_ERROR": ErrMPCExecution,
"MALICIOUS_ERROR": ErrMPCMaliciousError,
"BAD_ALLOCATION_ERROR": ErrAllocatingMemory,
"HASH_NOT_FOUND_ERROR": ErrMPCHashNotFoundError,
"INVALID_ARGUMENT": ErrInvalidArgument,
"OTHER_ERROR": ErrMPCUnknownError,
}
```

However, if `st.Message()` does not match any key in the `MPCExecutionResult` map, the returned error will be `nil`.

Although reviewing all possible statuses and errors returned by `RunMPC` falls outside the scope of this audit, this implementation remains a potential risk. Even if `MPCExecutionResult` currently covers all possible error values, this has not been explicitly confirmed in this audit. However, future updates to this repository or the MPC repository could introduce inconsistencies, leading to improper error handling and unexpected behavior.

Assets:

- gcEVM

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation: A more robust error-handling mechanism should be implemented in the `runOpcode` function to ensure that if `MPCExecutionResult[st.Message()]` is `nil`, the function still returns a valid error.

Resolution: The issue was resolved at https://github.com/soda-mpc/go-ethereum/pull/300 by implementing more robust error handling - specifically, returning `ErrMPCUnknownError` whenever `MPCExecutionResult[st.Message()]` is `nil`.

F-2025-9034 - Vulnerabilities in Go External Dependencies & Standard Library - Low

Description:

During static analysis using `govulncheck`, several vulnerabilities were identified in the current code base. While these vulnerabilities may not pose immediate risks, they could compromise the security posture of the project. These issues include vulnerabilities in the Go standard library and an external Go dependency.

GO-2024-3250: Improper error handling in ParseWithClaims

- Issue: Improper error handling in ParseWithClaims and bad documentation may cause dangerous situations in *github.com/golang-jwt/jwt*
- Vulnerable version: *github.com/golang-jwt/jwt/v4@v4.5.0*
- Fixed in: *github.com/golang-jwt/jwt/v4@v4.5.11*
- CVSS Score: 3.1/10
- Details:
- https://pkg.go.dev/vuln/GO-2024-3250
- https://github.com/golang-jwt/jwt/security/advisories/GHSA-29wx-vh33-7x7r

GO-2025-3447 Timing sidechannel for P-256 on ppc64le in crypto/internal/nistec

- Issue: A timing sidechannel vulnerability in P-256 elliptic curve cryptography could allow an attacker to extract private keys. This issue is present in *crypto/internal/nistec.*
- Vulnerable version:
- <= 1.22.11
- >= 1.23.0 < 1.23.6
- >= 1.24.0 < 1.24.0-rc.3
- CVSS Score 5.3/10
- Details:
- https://pkg.go.dev/vuln/GO-2025-3447

GO-2025-3420 Sensitive headers incorrectly sent after cross-domain redirect in net/http

- Issue: Sensitive HTTP headers are being incorrectly sent after cross-domain redirects, which could potentially leak sensitive data.
- Vulnerable version:
- <= go1.22.10
- go1.23.0-0 <= version < go1.23.5
- go1.24.0-0 <= version < go1.24.0-rc.2
- CVSS Score: 6.1/10

- Details:
- https://pkg.go.dev/vuln/GO-2025-3420

GO-2025-3373 Usage of IPv6 zone IDs can bypass URI name constraints in crypto/x509

- Issue: The use of IPv6 zone IDs in URI validation can bypass name constraints in the `crypto/x509` package.
- Vulnerable version:
- <= go1.22.10
- go1.23.0-0 <= version < go1.23.5
- go1.24.0-0 <= version < go1.24.0-rc.2
- CVSS Score: 6.1/10
- Details:
- https://pkg.go.dev/vuln/GO-2025-3373

Assets:

- Dependencies

Status: Mitigated

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation:

- **Upgrade** `github.com/golang-jwt/jwt` to v4.5.1 to resolve improper error handling in `ParseWithClaims`. While the CVSS score is low (3.1/10), this upgrade mitigates potential security risks and improves the overall security posture of the project.
- **Update the Go version** to at least go1.23.6, go1.24.0-rc.3 or later versions to address vulnerabilities in the Go standard library, including those in `crypto/internal/nistec`, `net/http`, and `crypto/x509`. This will ensure that the fixes for the vulnerabilities mentioned above are applied and that your project benefits from the latest security patches.
- **Embed static analysis tools** like `govulncheck` and `staticcheck` into your CI pipeline. This will allow continuous monitoring and detection of potential vulnerabilities in the codebase, ensuring proactive security management.

Resolution:

GO-2024-3250 (JWT `ParseWithClaims`)

- **Status:** Not fixed, still using version v4.5.0; v4.5.11 is required.
- **Soda Labs Team Response:** JWT authentication is optional and not used in production. The Soda team has promised to warn external operators accordingly.

GO-2025-3447 (P-256 Timing Side Channel)

- **Status:** Not fixed, currently using Go 1.21 / 1.22.3; Go 1.23.6 or later is required.
- **Soda Labs Team Response:** The issue only affects the `ppc64le` architecture (used by ~1% of processors). The team will advise against deploying on this architecture.

GO-2025-3420 (HTTP Sensitive Headers)

- **Status:** Not fixed, using Go 1.21 / 1.22.3; Go 1.23.6 or later is required.
- **Soda Labs Team Response:** The affected code is only used in non-production environments and does not process sensitive headers.

GO-2025-3373 (IPv6 Zone Identifiers)

- **Status:** Not fixed, using Go 1.21 / 1.22.3; Go 1.23.6 or later is required.
- **Soda Labs Team Response:** Deployments do not use private PKI. The team will alert operators.

Soda labs team Response:-

> **Regarding GO-2024-3250: Improper error handling in ParseWithClaims:**
> It seems this vulnerability is for the authenticated http handler. Authenticating the http sessions with jwt is an option which is determined in the command line by the node operator. We currently do not secure our nodes using a jwt (nor do we allow for public connections directly to any of our internal nodes and do not advise it to any external node operator).
> In any case, any external node operator will be made aware to this vulnerability alongside our discouragement to open the node to public connection. With this said, outside node operators do not risk any data leakage since these nodes do not have any effect on the network and are only used to *validate* received blocks.
>
> **Regarding the GO-2025-3447 Timing side-channel**:
> It seems that this side-channel would only be available in less then 1% of processors (the ones that use the ppc64le architecture, which are most commonly HPC systems)

Additionally it is stated in the vulnerability report that `“Due to the way this function is used, **we do not believe** this leakage is enough to allow recovery of the private key when P-256 is used in any well known protocols”`

We will instruct against deploying the node on any system with a processor built with the ppc64le architecture

Regarding GO-2025-3420 Sensitive headers incorrectly sent:
The usage of the net/http package you specified is in go-ethereum p2p simulations (which is not used in production), in the faucet (which we do not use) and in the ci process, which is used to install Golang on a machine. The usage is for public web pages which do not require any sensitive headers, so even if they were compromised the redirect chain would not yield any sensitive header leakage.
Furthermore the checksum file used to verify the golang source files has this statement made by the go-ethereum developers: `"This version **is fine to be old and full of security holes**, we just use it to build the latest Go. Don't change it."`

Additionally while reviewing the code for the CI I found no usage of any sensitive headers for the requests. With that said we would also instruct users not to edit any of the CI files to make sure this package is not used for any other purpose by mistake.

Regarding GO-2025-3373 Usage of IPv6 zone IDs:
I could not find the calls you specified in any of the locations you mentioned, could we be looking at different code versions?
Anyway we do not use private PKI in anywhere in our deployments and will make sure any operator is made aware of this vulnerability before deploying.

F-2025-9036 - Deprecated & Insecure Functions in Key Operations - Low

Description:

During a comprehensive code analysis, several deprecated types, functions, and packages were identified that could potentially impact the cryptographic integrity of critical operations, such as wallet connections, key generation, and encryption. These deprecated elements include:

- `elliptic.Marshal` and `elliptic.Unmarshal`, both deprecated methods, are actively used in multiple locations across the codebase, affecting core elliptic curve operations.
- Locations:
- *go-ethereum/accounts/scwallet/securechannel.go#L79*
- *go-ethereum/consensus/co2/consensus_test.go#L1556*
- *go-ethereum/crypto/crypto.go#L179*
- *go-ethereum/crypto/ecies/ecies.go#L258*
- *go-ethereum/crypto/secp256k1/secp256_test.go#L24*
- *go-ethereum/p2p/rlpx/rlpx.go#L667*
- *go-ethereum/soda_tests/soda_chain_maker.go#L295*
- *go-ethereum/crypto/crypto.go#L168*
- *go-ethereum/crypto/signature_cgo.go#L44*
- *go-ethereum/crypto/ecies/ecies.go#L300*
- `elliptic.GenerateKey`, another deprecated function, is still in use, contributing to key generation processes.
- Locations:
- *go-ethereum/crypto/ecies/ecies.go#L98*
- The deprecated `reflect.SliceHeader` and `reflect.PtrTo` are utilized in key sections of the code.
- Locations:
- *go-ethereum/rlp/unsafe.go#L30*
- *go-ethereum/rlp/decode.go*
- *go-ethereum/rlp/encode.go*
- `key.Curve.ScalarMult` has also been marked deprecated but continues to play a role in elliptic curve cryptography.
- Locations:
- *go-ethereum/accounts/scwallet/securechannel.go#L75*
- Furthermore, `grpc.DialContext`, a deprecated method, is present in the codebase, possibly affecting secure communication channels.
- Locations:
- *go-ethereum/core/vm/soda_extension_mpc.go#L421*

Elliptic curve-related functions like `key.Curve.ScalarMult`, `elliptic.Marshal`, and `elliptic.Unmarshal` are used in critical areas, such as wallet

connection implementations, private and public key generation, and decryption functions.

```
func NewSecureChannelSession(card *pcsc.Card, keyData []byte) (*Sec
ureChannelSession, error)

func decrypt(key, r, ct []byte) ([]byte, error)

func GenerateKey(rand io.Reader, curve elliptic.Curve, params *ECIE
SParams) (prv *PrivateKey, err error)
```

While the likelihood of exploitation is low, the use of deprecated functions in critical areas - such as wallet connections, key generation, and decryption operations - could potentially weaken the protocol's security posture. It's advisable to replace these outdated methods with current, supported alternatives to maintain optimal security.

Assets:

- Code Quality

Status: Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation: To address the deprecated functions identified in your codebase, consider implementing fixes similar to those applied in the go-ethereum. Reviewing the following commits can provide guidance:

Elliptic Curve Functions:

- The go-ethereum replaced references to deprecated elliptic curve functions with custom implementations in the `crypto` package. See commit hash [ab49f228ad6f37ba78be66b34aa5fee740245f57]

Unsafe Slice Usage:

- To replace the deprecated `reflect.SliceHeader`, the project utilized `unsafe.Slice`, enhancing memory handling efficiency. See commit hash [a970295956d602c348dccce034712c14aedce5e0].

Pointer Type Reflection:

- The deprecated `reflect.PtrTo` was replaced with `reflect.PointerTo`, aligning with updated Go language standards. See commit hash [f447de936c31e6a64470f3c102da85f245fe9640](f447de936c31e6a64470f3c102da85f245fe9640).

grpc dialcontext:

- grpc.DialContext is deprecated: use NewClient instead. This new method will use passthrough as a parameter rather than resolving domain in to ip using dns. As mentioned in this github thread.

Implementing similar updates in your codebase will help maintain compatibility with current Go standards and enhance overall security.

Resolution: The Soda team has indicated that they will consider integrating these changes in a future version.

F-2025-9180 - Unsafe CGO Integration in Consensus Mechanism - Low

Description:

The Soda consensus mechanism (CO2), implemented in *consensus/co2/consensus.go*, integrates with an external C library (`libses`) using CGO without proper memory safety measures, error handling, or input validation. The `libses` library performs core cryptographic operations and handles sensitive MPC (Multi-Party Computation) functions that are essential to the security of the consensus mechanism.

1. Unsafe Memory Management

The CGO interface uses direct memory access with unsafe pointers without proper bounds checking:

```
func initSES(update []byte) {
log.Info("Initializing SES")
C.InitSES((*C.uchar)(&update[0]), C.int(len(update)))
}
```

This creates a risk of buffer overflows, use-after-free vulnerabilities, and memory corruption issues.

2. Absence of Error Handling

C function calls lack proper error checking and handling:

```
C.InitSES((*C.uchar)(&update[0]), C.int(len(update)))
```

If the C library encounters errors, the Go code has no way to detect or respond to them, potentially leading to undefined behavior or silent failures with security implications.

3. Unchecked Type Conversions

The code converts between Go and C types without validation:

```
C.InitSES((*C.uchar)(&update[0]), C.int(len(update)))
```

This could lead to integer overflow or truncation if the Go slice is larger than what a C int can represent, potentially resulting in buffer overflow vulnerabilities.

These vulnerabilities in the CGO integration could lead to several severe security consequences:

1. **Memory Corruption**: Buffer overflows or use-after-free vulnerabilities could lead to arbitrary code execution.

2. **Consensus Failures**: Vulnerabilities in transcript handling could allow manipulation of transaction execution, potentially enabling double-spending or transaction censorship.
3. **Node Crashes**: Memory leaks or crashes in the C library could cause node failures.

Assets:

- SES module

Status: Mitigated

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation:

1. Use `runtime/cgo.Handle` for Memory Safety:

Replace direct pointer passing with Go's `runtime/cgo.Handle` mechanism, which provides a safe way to pass Go values to C code without exposing raw pointers.

```
// Current unsafe implementation
func initSES(update []byte) {
log.Info("Initializing SES")
C.InitSES((*C.uchar)(&update[0]), C.int(len(update)))
}

// Safer implementation using runtime/cgo.Handle
func initSES(update []byte) {
log.Info("Initializing SES")
h := cgo.NewHandle(update)
status := C.InitSES_Safe(C.uintptr_t(h), C.int(len(update)))
if status != 0 {
log.Error("SES initialization failed with status code", "status", s
tatus)
}
h.Delete() // Release the handle when done
}

// C side interface would need to be modified to accept handle IDs
instead of direct pointers
```

Benefits of this approach include:

- Elimination of direct pointer passing, and protection against buffer overflows.
- Ensuring Go values remain alive during C function calls.
- Explicit memory lifecycle management.
- Enforcement of a cleaner separation between Go and C memory spaces.

2. Implement Input Validation: Add thorough validation for all inputs passed to C functions.

3. Add Error Handling: Modify all C function calls to check and handle return values and error codes.

4. Implement Memory Safety Wrappers: Create safer interfaces that handle memory allocation/deallocation properly.

Resolution: The Soda Labs Team has addressed input validation and error handling in PR #305 in `go-ethereum` and PR #4 in `ses`.

Here is the response from the The Soda Labs Team:-

> Our CGO implementation follows a synchronous request-response pattern. All pointers passed into C functions are scoped strictly within the lifetime of each call, with no pointers stored or used after the call returns. This ensures the memory remains pinned and protected from garbage-collector movement during execution. Additionally, we explicitly manage memory allocated on the C side by freeing pointers with `C.free()`, eliminating potential memory leaks. This design effectively mitigates any memory corruption risks.

F-2025-9212 - Unreliable Signature Verification Process - Low

Description:

The `verifySignatureAndSigner` function in `core/vm/soda_extension_mpc.go` attempts to recover the public key from a signature using the `Ecrecover` function. If this initial attempt fails, the function modifies the message and signature before retrying. This approach presents potential issues:

- **Unvalidated Signature Modification:** The `updateSignature` function alters the `signature[64]` byte without verifying its initial value. This practice risks changing the recovery identifier from valid (0 or 1) to invalid, potentially leading to failures.
- **Unconditional Message Prefixing:** The `updateMessageWithPrefix` function prepends the EIP-191 prefix to the message without checking if it's already present. This could result in improper message formatting, affecting signature verification.

These practices may cause valid signatures to be incorrectly rejected or allow both prefixed and non-prefixed message signatures to be accepted, compromising the integrity of the signature verification process.

Assets:

- gcEVM

Status: Accepted

Classification

Impact Rate: 3/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation:

Revise the signature verification process as follows:

- Independently validate the formats of both the message and the signature.
- Modify the message or signature only if they deviate from the expected formats:
  - Apply the EIP-191 prefix only if the message doesn't adhere to EIP-191 standards.

    - If `v` value of signature is 27 or 28, adjust it to 0 or 1, respectively.
- After ensuring the correctness of the message and signature formats, use `Ecrecover` once to retrieve the signer's public key.

Implementing these changes will enhance the reliability and security of the signature verification process.

Resolution:

The Soda team clarified that their SDKs are responsible for ensuring the correct signature format is used based on the message type. For raw messages, the SDK produces signatures with (v) in 0/1 format; for EIP-191-prefixed messages, it uses 27/28, which is normalized during verification. The underlying assumption is that the signer (via SDK) is aware of the intended signing method and provides the appropriate signature accordingly.

While this approach is secure and does not introduce any vulnerabilities, there may still be some ambiguity around what constitutes an invalid signature in practice. For instance, a signature with (v) = 0/1 over an EIP-191-prefixed message might be valid, but the current logic may not handle this case explicitly. This could lead to valid signatures being rejected if the format is misinterpreted, particularly by clients not using the official SDKs.

One potential improvement could be to explicitly validate and, if necessary, normalize the signature's (v) value prior to `Ecrecover`. However, implementing such a change would require a fork, which was deemed unnecessary.

While no security risk were confirmed, to avoid confusion and improve interoperability, it is recommended to document the expected signature and message formats more thoroughly, and consider adding explicit checks (e.g., for the presence of the EIP-191 prefix) to make the verification process more predictable.

F-2025-9214 - Unnecessary Insertion to Authenticated Memory After offBoardToUser - Low

Description:

The `offBoardToUser` function offboards the given garbledtext, creating a ciphertext under user's symmetric AES key. This ciphertext allows the AES key owner to retrieve the private data; however, it should not be used otherwise. Specifically, there appears to be no reason to insert it into authenticated memory.

This assertion aligns with the gcEVM whitepaper, which states that a ciphertext offboarded to a user is not considered authenticated.

Nonetheless, the current implementation inserts the ciphertext into authenticated memory:

core/vm/soda_extension_mpc.go:1413:

```
func (c *mpcContract) callOffboardToUser(opName string, input []byt
e, evm *EVM) ([]byte, error) {
// Input validation and runOpcode
var tempArr [VAR_SIZE]byte
copy(tempArr[:], encryptedValue)

// Insert the cipher-text into the validated cipher-text memory
insertToAuthenticatedMemory(evm, tempArr)
}
```

While inserting the ciphertext into authenticated memory is unlikely to introduce security vulnerabilities, it is unnecessary and could lead to confusion regarding its intended use.

Assets:

- gcEVM

Status: Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation: To enhance security and clarity, avoid inserting ciphertexts offboarded to users into authenticated memory. Instead, ensure that such ciphertexts are handled appropriately, maintaining their

confidentiality and integrity without unnecessary inclusion in authenticated memory spaces.

Resolution: It has been acknowledged that the insertion into authenticated memory after `offBoardToUser` is unnecessary, but it does not pose any security risks. Therefore, the Soda team has decided not to implement a fix, as it would require a fork or chain reset due to changes in how data is stored in the contract.

F-2025-9294 - Centralized Token Emission Distribution Creates Critical Single Point of Failure - Low

Description:

The implementation of the Co2 consensus engine's token minting mechanism directs 100% of newly minted tokens to a single address defined as `FoundationAccount` in the genesis configuration. Analysis of `mintCoins()` function in `go-ethereum/consensus/co2/consensus.go` reveals that this account receives all emissions without any distribution logic or fallback mechanisms.

```
func (co2 *Co2) mintCoins(stateDB *state.StateDB, AmountInWei *big.
Int) {
// Conjure up some coins and give them to the foundation
prevBalance := stateDB.GetBalance(co2.emissions.FoundationAccount)
stateDB.AddBalance(co2.emissions.FoundationAccount, AmountInWei)
currentBalance := stateDB.GetBalance(co2.emissions.FoundationAccoun
t)
log.Info("Minted coins", "amount in Wei", AmountInWei,
"prev balance", prevBalance, "current balance", currentBalance)
}
```

Two issues compound this vulnerability:

1. **Immutable Configuration:** There is no mechanism in the codebase to update the FoundationAccount after chain genesis. If the account is compromised, there is no way to redirect future emissions to a secure address.
2. **EOA Compatibility:** The `FoundationAccount` can be an Externally Owned Account (EOA). If an EOA is used, security depends entirely on a single private key.

If the `FoundationAccount` private key is compromised, it would allow an attacker to:

- Gain control of 100% of newly minted tokens
- Potentially destabilize the token economy through sudden large sell-offs

The absence of multi-signature requirements or distribution contracts creates a significant centralization vector in what should be a decentralized system.

Assets:

- Minting & inflation

Status: Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation: Replace the direct transfer to a single address with a more robust distribution system. Consider implement a distribution contract as the `FoundationAccount` with:

1. Multi-signature requirements.
2. Time-locked withdrawals with cooling periods.
3. Transparent allocation logic for different ecosystem needs (dev, treasury, staking).

Resolution: The issue was acknowledged with the following comment:

> This is a “minting package” commit that was added especially for our design partner request.
> For other partners we offer two options:
> 1. Changing the “Foundation” address to be an array rather than a single address (requires code change and re-audit).
> 2. Remaining with a single address but having this address be managed by a DAO, this can be done by:
> - having this address be a smart contract wallet (e.g., gnosis safe) - that works with multisig
> - using a threshold sig EOA with secret key shares held by the DAO's members.

F-2025-9403 - Questionable Validation of SodaMPCExecutionError - Low

Description:

The `SodaMPCExecutionError` field in the `EVM` is designed to store the first error encountered during operations on private data. Several issues have been identified in the current error-handling approach:

1. **Inconsistent Panic Usage**: Panics are used for specific errors, which could be acceptable for unrecoverable errors that indicate critical node state issues. However, several errors in the `applyTransaction` function, such as those found in the `MPCUnrecoverableErrors` map, trigger panics. Notably, errors like `ErrMPCUnknownError`, `ErrMPCMaliciousError`, and `ErrAllocatingMemory` raise concerns. While a full review of errors from `client.RunMPC` is outside this audit's scope, it is advisable to reconsider the necessity of panics, particularly if such errors could be triggered by external malicious actors attempting to halt the chain.
2. **Documentation-Implementation Mismatch**: Errors like `ErrRunCalledOnEVMAware` and `ErrEVMIsNIL` are documented as causing panics. However, both errors appear in the `MPCTxReversionErrors` map, which corresponds to reverted transactions. Although the implementation may not need to change, clearer documentation of the expected behavior would improve consistency.
3. **Missing Logic for** `ErrInvalidCall`: The `ErrInvalidCall` error is neither present in the `MPCUnrecoverableErrors` nor the `MPCTxReversionErrors` map, suggesting a potential oversight or missing logic.
4. **Unused Error in** `EVMAwareRun`: In the `EVMAwareRun` function, if the execution type is `TranscriptEvaluation`, the `SodaMPCExecutionError` may be set. However, this error is only utilized in `applyTransaction` for the `MPCExecution` type. As a result, saving the error during `TranscriptEvaluation` is redundant, as it will not be used by the sequencer or validator.

Assets:

- gcEVM

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation: To enhance error handling, the following is suggested:

- **Reserve Panics for Critical Failures**: Use panics only for unrecoverable errors, such as critical system failures. For expected errors, return error values to maintain stability.
- **Align Documentation with Behavior**: Ensure documentation matches the actual behavior of error handling, specifying whether errors cause panics or are returned for handling.
- **Utilize Errors Consistently**: Ensure that errors like `SodaMPCExecutionError` are used consistently across different execution types, avoiding redundancy and ensuring they are actionable.

Implementing these practices will lead to more consistent and maintainable error handling.

Resolution:

1. The Soda team has indicated a preference for failing immediately and performing manual investigation and remediation in the event of unrecoverable errors, due to the potential impact on chain state and overall security. Although not typical in most decentralized system scenarios, we agree that in this context it is a valid and reliable approach.
2. The issue has been addressed in PR #315.
3. The comment on `ErrInvalidCall` has been updated to clarify the expected behavior. Specifically, a view call that triggers this condition should return `nil, ErrInvalidCall`. Since `ErrInvalidCall` is used solely for this purpose and for logging, it does not need to be included in either the `MPCUnrecoverableErrors` or `MPCTxReversionErrors` maps.
4. The Soda team acknowledged the redundancy but chose to retain it for potential debugging purposes in the future. We agree that this redundancy is harmless.

F-2025-12033 - Missing gcEVM Context Initialization Reduces Tracing Reliability - Info

Description:

The issue reported by the Soda team concerns the `debug_traceBlockByNumber` functionality. The problem originates from missing gcEVM adjustments in the *eth/tracers/* module.

Specifically, the tracing logic does not set the execution type, the VM configuration variable, leaving `execType == nil`. In addition, the EVM's `Transcript` is never initialized. As a result, tracing operates without an essential execution context, which disrupts the operational consistency of the process and leads to incorrect evaluation of calls to private on-chain operations. This also leads to a panic in `debug_traceBlockByNumber` when the execution type is `nil`.

This issue is assessed as having no direct security impact, as debug RPCs are not intended to be exposed in production environments. However, addressing it would significantly improve the functionality, performance, and reliability of the tracing system.

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation:

Update the *eth/tracers/* implementation to:

- Properly set the execution type (`execType`) in the VM configuration during tracing initialization.
- Ensure the EVM's Transcript is initialized before executing trace operations.

In addition, operators should ensure that debug RPC methods remain disabled in production environments, as they are intended solely for development and diagnostic use.

Implementing these changes will provide the necessary execution context for consistent and performant tracing, improving the accuracy, reliability, while avoiding operational inconsistencies.

Resolution: The issue was fixed in PR#330 and PR#332.

Regarding the debug RPC methods, the Soda team provided the following description:

> In our production environment, tracing is enabled **only on validator nodes**, which are non-block-producing nodes that passively maintain chain state. These nodes:
> - Contain **only black blocks** (finalized blocks with complete transcripts).
> - Never contain **red blocks** (intermediate state), which exist only on sequencer/executor infrastructure.
> - Have transcripts of at least length 1 even in “empty” blocks, as they always include MPC circuit status.
>
> On sequencer and executor nodes, where red blocks exist, the debug API is disabled entirely.

F-2025-9027 - Absence of Authenticated Memory Emptiness Validation Upon EVM Depth Increase - Info

Description:

According to the documentation of gcEVM, when a function call occurs and the EVM depth increases, a validation step should ensure that the authenticated memory for the new depth level is empty.

> Similarly, when depth is increased from **d** to **d+1** (on a function call), we make sure that **M(d+1)** is empty.

Currently, this check is missing from the codebase, which could increase susceptibility to unintended memory manipulation.

This omission is unlikely to result in security vulnerabilities since authenticated memory is cleared when the depth decreases. However, implementing this validation would enhance the system's robustness and reliability while ensuring consistency with the documentation.

Assets:

- gcEVM

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Introduce a validation check within the `Run` method implementation of `EVMInterpreter` to verify that the value associated with the increased depth in the `in.validCts` map is empty.

Resolution: The issue was fixed at https://github.com/soda-mpc/go-ethereum/pull/301.

F-2025-9035 - Errors Identified in Unit Tests Compilation - Info

Description:

During the general analysis of code it was identified that some of the important test cases are unable to be compiled and run unit test due to some undefined variables. Some of the example are given below:

github.com/ethereum/go-ethereum/consensus/co2

```
consensus/co2/consensus_test.go:44:31: undefined: ExtraDataLength
consensus/co2/consensus_test.go:266:28: undefined: ExtraVanityLengt
h
consensus/co2/consensus_test.go:282:37: undefined: ExtraVanityLengt
h
consensus/co2/consensus_test.go:282:73: undefined: ExtraSeal
consensus/co2/consensus_test.go:283:49: undefined: ExtraVanityLengt
h
consensus/co2/consensus_test.go:285:31: undefined: ExtraVanityLengt
h
consensus/co2/consensus_test.go:295:28: undefined: ExtraDataLength
consensus/co2/consensus_test.go:306:23: undefined: ExtraVanityLengt
h
consensus/co2/consensus_test.go:306:77: undefined: ExtraSeal
consensus/co2/consensus_test.go:323:37: undefined: ExtraDataLength
```

github.com/ethereum/go-ethereum/core/vm

```
core/vm/contracts_test.go:841:22: multiple-value getSize(testCase.b
its) (value of type (int, error)) in single-value context
core/vm/contracts_test.go:912:6: mpc.SetParams undefined (type *mpc
Contract has no field or method SetParams)
core/vm/contracts_test.go:1128:44: undefined: splitIntoThreeSlices
core/vm/contracts_test.go:1304:56: undefined: splitIntoFourSlices
core/vm/contracts_test.go:1564:8: mpc.SetParams undefined (type *mp
cContract has no field or method SetParams)
core/vm/contracts_test.go:1567:8: mpc.evm undefined (type *mpcContr
act has no field or method evm)
core/vm/contracts_test.go:1568:8: mpc.evm undefined (type *mpcContr
act has no field or method evm)
core/vm/contracts_test.go:1583:16: undefined: genUserKeyFromSeed
core/vm/contracts_test.go:1589:15: undefined: encrypt
core/vm/contracts_test.go:1604:48: mpc.caller undefined (type *mpcC
ontract has no field or method caller)
```

github.com/ethereum/go-ethereum/soda_tests

```
soda_tests/soda_chain_maker.go:83:20: cannot use cm.cliqueConf (var
iable of type *params.CliqueConfig) as *params.Co2Config value in a
rgument to co2.New
soda_tests/soda_chain_maker.go:94:29: cannot use config (variable o
f type *params.CliqueConfig) as *params.Co2Config value in argument
to co2.New
soda_tests/soda_chain_maker.go:97:29: cannot use config (variable o
f type *params.CliqueConfig) as *params.Co2Config value in argument
to co2.New
soda_tests/soda_chain_maker.go:100:29: cannot use config (variable
of type *params.CliqueConfig) as *params.Co2Config value in argumen
t to co2.New
soda_tests/soda_chain_maker.go:330:44: undefined: co2.Signature1
soda_tests/soda_chain_maker.go:332:43: undefined: co2.Signature2
```

Assets:

- Code Quality

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: The identified test cases are currently failing to compile due to undefined variables, missing methods, and type mismatches. These issues prevent the execution of unit tests, which is essential for verifying the logic of the code. The following actions are recommended:

- Resolve the undefined variable errors, such as `ExtraDataLength`, `ExtraVanityLength`, and `ExtraSeal` in the `consensus/co2/consensus_test.go` file.
- Address method definition issues like `mpc.SetParams` and the undefined `mpc.evm` in the `core/vm/contracts_test.go` file.
- Fix type mismatches, especially the incorrect use of `*params.CliqueConfig` as `*params.Co2Config` in `soda_tests/soda_chain_maker.go`.
- Ensure missing functions like `genUserKeyFromSeed`, `encrypt`, and `splitIntoThreeSlices` are properly defined and accessible.

Ensuring that tests compile and pass is essential for maintaining a robust test suite. This practice helps catch bugs and identifying issues promptly.

Resolution: The Soda Labs Team fixed the compilation errors in unit tests in two PRs:

1. PR #307:

- Fixed issues in the `co2` consensus package by properly defining variables like `ExtraDataLength`, `ExtraVanityLength`, and `ExtraSeal` in `shared/shared.go`.
- Corrected type mismatches in `soda_chain_maker.go` where `CliqueConfig` was incorrectly used instead of `Co2Config`.
- Addressed signature-related errors in consensus tests.

2. PR #316:

- Removed problematic MPC-related test functions from `core/vm/contracts_test.go` as these tests required integration-level setup rather than unit testing.
- Eliminated unused imports and functions with undefined methods (`mpc.SetParams`, `mpc.evm`).

- Removed references to undefined helper functions like `genUserKeyFromSeed`, `encrypt`, and the `splitIntoThreeSlices`.

F-2025-9211 - Eliminate Unused Variables in common Package - Info

Description: In the `common` package, the variables `SodaPrecompileExecutionResult` and `SodaTranscript` are declared but not utilized meaningfully within the codebase. Their sole usage occurs in the `ResetTranscript` function, where they are merely assigned empty values, rendering their presence ineffective.

Unused variables can clutter the code, making it harder for developers to navigate and maintain. They may also indicate unfinished code, mistakes during implementation or refactoring, and can reduce readability.

Assets:

- Code Quality

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: To enhance code clarity and maintainability, it is advisable to remove these unused variables. Eliminating them will reduce confusion and align with Go's best practices, which discourage the inclusion of unused variables in the codebase.

Resolution: The issue was fixed at https://github.com/soda-mpc/go-ethereum/pull/306.

F-2025-9222 - Duplicate Verification Calls in Executor Block Insertion - Info

Description:

In the `validateExecutorBlockForExecutorInsertion` function, a duplicate call to `verifyExecutorBlock` occurs when `receivedHeaderNum > existingHeaderNum`. This redundancy leads to unnecessary repetition of the same check, potentially impacting system efficiency.

ses/main.go:379-401:

```
case receivedHeaderNum > existingHeaderNum:
// The block we received has a higher number than the current head:
// We should first validate it.
log.Warn("Executor received Executor block with a number higher tha
n the current head",
"receivedHeaderNum", receivedHeaderNum, "existingHeaderNum", existi
ngHeaderNum)
if err := verifyExecutorBlock(blockToAccept); err != nil {
// The block is not valid! We return an error.
log.Error("Block is not valid", "hash", headerToAccept.Hash(), "num
ber", receivedHeaderNum)
return err // This error should raise an alert since we got a bad b
lock.
}
// If we're in the middle of a sync we accept the block without val
idating
// the chain is ordered by the SequencerBlock:x ExecutorBlock:x Seq
uencerBlock:x+1..
// since during syncing only Executor blocks are sent.

log.Warn("Executor is syncing, accepting block without red-black va
lidation")
if err := verifyExecutorBlock(blockToAccept); err != nil {
log.Error("Block is not valid", "hash", headerToAccept.Hash(), "num
ber", receivedHeaderNum, "error", err)
return err // This error should raise an alert since we got a bad b
lock.
}
// OK the block is valid, but we still can't insert it because it i
s too far ahead.
// We should return an error.
log.Error("Block is too far ahead", "hash", headerToAccept.Hash(),
"number", receivedHeaderNum)
return errCheckSyncSkipBlock
```

Additionally, comments suggest that during synchronization, blocks are accepted without validation. However, the implementation performs validation and returns an error, leading to potential confusion.

Assets:

- Code Quality
- SES module

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: To enhance efficiency and clarity:

- **Eliminate Duplicate Function Calls**: Restructure the conditional logic to ensure that `verifyExecutorBlock(blockToAccept)` is invoked only once when necessary. This approach reduces redundancy and streamlines the codebase.
- **Reevaluate Synchronization Logic**: Review the current implementation and comments regarding block acceptance during the synchronization process. Ensure that the code accurately reflects the intended behavior, particularly concerning the validation steps during synchronization.

Resolution: Fixed via changes in SES codebase PR:- https://github.com/soda-mpc/ses/pull/3

Observation Details

Disclaimers

Hacken Disclaimer

The blockchain protocol given for audit has been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in the protocol source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other protocol statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the blockchain protocol.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

As part of Hacken’s ongoing quality assurance process, we may conduct re-audits of select projects. These re-audits are performed independently from the original audit and are intended solely for internal quality control and improvement. Updated reports resulting from such re-audits will be shared privately with the respective clients and may be published on the Hacken website only with their explicit consent.
The sole authoritative source for finalized and most up-to-date versions of all reports remains the Audits section at https://hacken.io/audits/.

Technical Disclaimer

Blockchain protocols are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the protocol can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited blockchain protocol.

Appendix 1. Severity Definitions

Severity	Description
Critical	Vulnerabilities that can lead to a complete breakdown of the blockchain network's security, privacy, integrity, or availability fall under this category. They can disrupt the consensus mechanism, enabling a malicious entity to take control of the majority of nodes or facilitate 51% attacks. In addition, issues that could lead to widespread crashing of nodes, leading to a complete breakdown or significant halt of the network, are also considered critical along with issues that can lead to a massive theft of assets. Immediate attention and mitigation are required.
High	High severity vulnerabilities are those that do not immediately risk the complete security or integrity of the network but can cause substantial harm. These are issues that could cause the crashing of several nodes, leading to temporary disruption of the network, or could manipulate the consensus mechanism to a certain extent, but not enough to execute a 51% attack. Partial breaches of privacy, unauthorized but limited access to sensitive information, and affecting the reliable execution of smart contracts also fall under this category.
Medium	Medium severity vulnerabilities could negatively affect the blockchain protocol but are usually not capable of causing catastrophic damage. These could include vulnerabilities that allow minor breaches of user privacy, can slow down transaction processing, or can lead to relatively small financial losses. It may be possible to exploit these vulnerabilities under specific circumstances, or they may require a high level of access to exploit effectively.
Low	Low severity vulnerabilities are minor flaws in the blockchain protocol that might not have a direct impact on security but could cause minor inefficiencies in transaction processing or slight delays in block propagation. They might include vulnerabilities that allow attackers to cause nuisance-level disruptions or are only exploitable under extremely rare and specific conditions. These vulnerabilities should be corrected but do not represent an immediate threat to the system.

Appendix 2. Scope

The scope of the project includes the following components from the provided repository:

Scope Details	
Repository	https://github.com/soda-mpc/go-ethereum
Commit	c53489958e34ca2dcc57bf5e8246aa0ca9015a88
Whitepaper	https://www.sodalabs.xyz/wp-content/uploads/2024/10/gcEVM-v.-0.1-1.pdf

Components in Scope

The scope consists of the whole codebase, with primarily focus on:

- Modification of EVM, allowing private data and secure operations.
- Soda co2 consensus engine.
- gcEVM node roles.
- SES module.
- Minting & inflation.

The remediation check has been conducted based on commit hash 76a7d1e, which reflects the status of each issue following this process. It is important to acknowledge that this commit may include changes made subsequent to the initial review commit, which were not part of the audit assessment.