Zellic

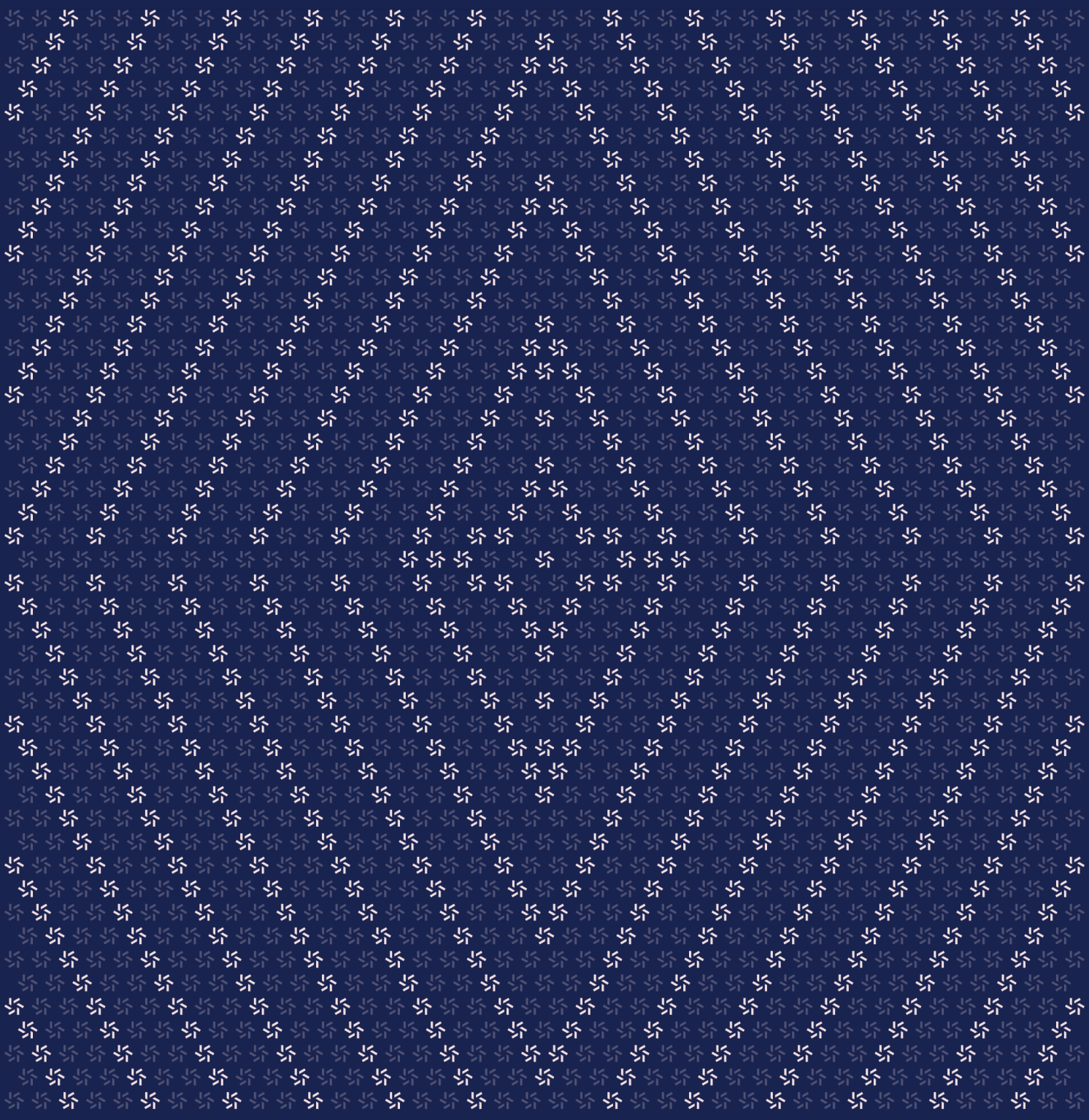
Prepared for
Jordan Oroshiba
Astria

Prepared by
Filippo Cremonese
Daniel Lu
Zellic

June 24, 2024

Astria Geth

Differential Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Astria from June 10th to June 19th, 2024. During this engagement, Zellic reviewed Astria Geth's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

This assessment was conducted by reviewing the differences between the official Ethereum Golang implementation (Geth) and Astria's fork. Specifically, our review compared Geth release tag v1.14.3 (commit `ab48ba42f4f34873d65fd1737fabac5c680baff6`) with the Astria Geth code found at PR #21 (commit `5f9724be5ad41500855c9c6e6f76037e438f320c` at the time of our review).

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

1.4. Results

During our assessment on the scoped Astria Geth modules, we discovered one finding, which was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Astria's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	1

2. Introduction

2.1. About Astria Geth

Astria contributed the following description of Astria Geth:

> Astria is a decentralized sequencing layer that can be shared amongst multiple rollups. It does this by separating transaction sequencing and execution.
>
> Astria geth is a fork of the official Ethereum golang implementation which enables the use of geth as an EVM-compatible rollup transaction executor.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the modules.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case

basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped modules itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Astria Geth Modules

Type	Golang
Platform	EVM-compatible
Target	astria-geth
Repository	https://github.com/astriaorg/astria-geth ↗
Version	`a30fd3d23b31dff26ded8abf373e0bc8050d08c7`
Programs	`geth`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.4 person-weeks. The assessment was conducted by two consultants over the course of eight calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Filippo Cremonese
Engineer
fcremo@zellic.io ↗

Daniel Lu
Engineer
daniel@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

June 10, 2024	Start of primary review period
June 19, 2024	End of primary review period

3. Detailed Findings

3.1. Incorrect ERC-20 deposit handling

- Target: state_transition.go
- Category: Coding Mistakes
- Severity: High
- Likelihood: N/A
- Impact: Informational

Description

The pull request under review rebases Astria preexisting changes on top of official Geth 1.14.3. At the time of the start of this engagement, PR #21 HEAD was at commit `a30fd3d23b31dff26ded8abf373e0bc8050d08c7`. During the first two days of our engagement, Astria independently found and fixed some issues (seemingly originating from incorrect manual merge-conflict resolutions), updating the PR HEAD to `5f9724be5ad41500855c9c6e6f76037e438f320c`. This included a relatively severe issue in `state_transition.go::TransitionDb`.

The function was modified by Astria to handle the newly introduced deposit-transaction type. In the initial version of the code, the function code appeared as follows:

```
func (st *StateTransition) TransitionDb() (*ExecutionResult, error) {
    // if this is a deposit tx, we only need to mint funds and no gas is used.
    if st.msg.IsDepositTx {
        log.Debug("deposit tx minting funds", "to", *st.msg.To, "value",
    st.msg.Value)
        st.state.AddBalance(*st.msg.To, uint256.MustFromBig(st.msg.Value),
    tracing.BalanceIncreaseAstriaDepositTx)
        return &ExecutionResult{
            UsedGas: 0,
            Err: nil,
            ReturnData: nil,
        }, nil
    }

    if st.msg.IsDepositTx {
        st.initialGas = st.msg.GasLimit
        st.gasRemaining = st.msg.GasLimit
        log.Debug("deposit tx minting erc20", "to", *st.msg.To, "value",
    st.msg.Value)
    }
    /// ...
```

The initial condition is intended to handle only native deposit transactions, and it was incorrect. ERC-20 deposits always have a `msg.Value` of zero but a nonempty `msg.Data` containing the calldata,

which invokes the `mint` function of the destination contract that manages the bridged asset.

Impact

The `TransitionDb` function was incorrectly handling ERC-20 deposit transactions as native deposit transactions. This would have prevented successful bridging of ERC-20 assets, leading to the loss of the deposited assets.

Recommendations

None. (The issue was already independently found and fixed.)

Remediation

This issue has been acknowledged by Astria, and a fix was implemented in commit `5f9724be` ↗.

The issue was fixed as follows:

```
func (st *StateTransition) TransitionDb() (*ExecutionResult, error) {
    // if this is a deposit tx, we only need to mint funds and no gas is used.
   if st.msg.IsDepositTx {
   if st.msg.IsDepositTx && len(st.msg.Data) == 0 {
        log.Debug("deposit tx minting funds", "to", *st.msg.To, "value",
    st.msg.Value)
        st.state.AddBalance(*st.msg.To, uint256.MustFromBig(st.msg.Value),
    tracing.BalanceIncreaseAstriaDepositTx)
        return &ExecutionResult{
            UsedGas: 0,
            Err: nil,
            ReturnData: nil,
        }, nil
    }

    if st.msg.IsDepositTx {
        st.initialGas = st.msg.GasLimit
        st.gasRemaining = st.msg.GasLimit
        log.Debug("deposit tx minting erc20", "to", *st.msg.To, "value",
    st.msg.Value)
    }
    /// ...
```

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Discussion of the most significant changes

This section documents noteworthy security-relevant differences between Astria Geth and the official repository.

New gRPC service

The Geth codebase was modified to add a gRPC endpoint exposing the executor interface required by Astria (specifically, by the Astria conductor component).

This interface allows to create new blocks by executing transactions (via `ExecuteBlock`) and to update the blockchain head (via `UpdateCommitmentState`).

The gRPC service is intended to be exposed only to a trusted conductor service, and by default it binds only to the local loopback interface.

Mempool and miner changes

The codebase significantly simplified the block miner so that transaction ordering is determined by the order in which they are received via the gRPC service from the conductor, which in turn receives ordered transactions from the sequencer.

Deposit transactions

Astria added the new `DepositTx` transaction type to Geth, used to represent incoming bridge transactions.

Gas charges

Deposit transactions do not charge any intrinsic gas; native deposits do not involve actual bytecode execution and do not consume any gas, while ERC-20 deposits have a low gas budget (16,000) and invoke the contract that represents the deposited asset. The gas budget is fixed and not taken from any balance. Since all deposit transactions are not externally sourced but are instead generated by the sequencer, no gas-related denial-of-service issues can occur.

Since deposit transactions do not consume any gas, they also do not originate gas refunds.

Validation

Deposit transactions skip several checks, entirely bypassing the `StateTransition::preCheck`

function. Skipped checks include nonce verification, ensuring that the transaction is sent from an EOA and not from a contract (which should be impossible as the private key for a contract address is not recoverable) as well as minimum gas requirements. Even more notably, deposit transactions are not subject to any sort of signature check; there is no need to provide a valid sender signature.

Skipping these checks is justified since deposit transactions are not externally sourced but automatically generated from deposit events by the sequencer (and validated by the conductor).

Execution

Deposit transactions are added to the mempool alongside other Ethereum transactions. Their execution is handled by the same function that handles the other preexisting transaction types.

However, deposits are distinguished from normal transaction types via a boolean field `IsDepositTx`. The most crucial part of deposits handling is implemented in `state_transition.go::StateTransition::TransitionDb`. Deposits are divided into two kinds, handled differently: native deposits and ERC-20 deposits.

Native deposits grant the recipient some balance denominated in the native currency of the chain. These deposits are handled by increasing the recipient native balance and returning early.

ERC-20 deposits are handled almost identically to any regular transaction invoking a contract. This is because when an ERC-20 deposit transaction generated by the sequencer is received from the conductor, it is converted into a transaction invoking the `mint` function of the ERC-20 contract that represents the bridged asset. The only changes made to `TransitionDb` to handle ERC-20 deposits concern gas budget, gas refunds, and coinbase tips. Due to modifications in other functions (specifically, skipping `preCheck` leading to not invoke `buyGas`), the gas budget of ERC-20 deposit transactions is not set; therefore, a block of code that sets the appropriate gas limit was introduced.

Additionally, the function returns early to avoid originating gas refunds and distributing coinbase tips.

4.2. Testing

The forked `go-ethereum` repo includes a comprehensive test suite, and the Astria team has added some additional tests to cover the functionality related to the protocol. However, a more comprehensive test suite would be beneficial to ensure the correctness of changes made to the codebase.

In particular, we pointed out before the engagement that the `ExecuteBlock` function in grpc/execution/server.go omitted a check that special transactions (such as deposit transactions) should not be included in sequenced data. More explicit reasoning about incorrect behavior through tests would have helped catch this issue before it was merged.

Similarly, Finding 3.1. ↗ was in fact a regression, where the commit `5f9724be` ↗ removed a correct check that was previously added. A negative test case for the original logic could have helped catch this issue before it was merged as well.

5. Assessment Results

At the time of our assessment, the reviewed code was not in use on mainnet.

During our assessment on the scoped Astria Geth modules, we discovered one finding, which was informational in nature.

5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.