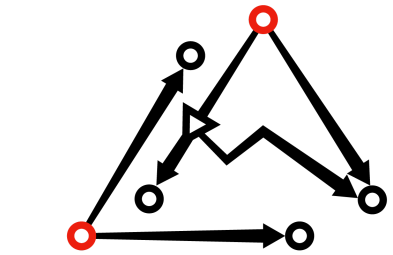

Least Authority

PRIVACY MATTERS

Gravity Bridge

Security Audit Report

Althea

Final Audit Report: 11 April 2022

Table of Contents

Overview

Background

Althea has requested that Least Authority perform a security audit of their Gravity Bridge, a bridge between Ethereum and Cosmos-based blockchains. The Gravity Bridge facilitates the transfer of ERC-20 tokens originating on Ethereum to Cosmos and back to Ethereum.

Project Dates

- **November 1 - December 10**: Code Review *(Completed)*
- **December 15**: Delivery of Initial Audit Report *(Completed)*
- **April 7 - 8**: Verification Review *(Completed)*
- **April 11**: Delivery of Final Audit Report *(Completed)*

Review Team

- Suyash Bagad, Security Researcher and Engineer
- David Braun, Security Researcher and Engineer
- Steven Jeung, Security Researcher and Engineer
- DK, Security Researcher and Engineer
- Dylan Lott, Security Researcher and Engineer
- ElHassan Wanas, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and a review of the Gravity Bridge followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:

- Gravity Bridge: https://github.com/althea-net/gravity-private

Specifically, we examined the Git revision for our initial review:

`c9562fbb157f0f741c5086aa5154659fc372efd1`

For the verification, we examined the Git revision:

`a583f30dc8c4307f727b7987161491f7db64f041`

For the review, this repository was cloned for use during the audit and for reference in this report:

https://github.com/LeastAuthority/althea-net-gravity-private

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in-scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- README.md: https://github.com/LeastAuthority/althea-net-gravity-private/blob/main/readme.md
- Blogpost, "How Gravity Works": https://blog.althea.net/how-gravity-works/

In addition, this audit report references the following documents:

- P. Robinson, 2019, "The merits of using Ethereum MainNet as a Coordination Blockchain for Ethereum Private Sidechains," *arXiv:1906.04421v2 [cs.CR]* [R19]

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Common and case-specific implementation errors;
- Adversarial actions and other attacks on the bridge;
- Attacks that impact funds such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) attacks and security exploits that would impact or disrupt execution of the bridge;
- Vulnerabilities within individual components as well as secure interaction between the components;
- Exposure of any critical information during interaction with any external libraries;
- Proper management of encryption and signing keys;
- Protection against malicious attacks and other methods of exploitation;
- Data privacy, data leaking, and information integrity;
- Inappropriate permissions and excess authority; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

The Gravity Bridge enables the transfer of ERC-20 tokens on Ethereum and Cosmos assets between the Cosmos and Ethereum blockchains. Tokens originating on Ethereum are transferred to Cosmos by locking them up in the `Gravity.sol` smart contract, then minting an equivalent amount of tokens on Cosmos. Newly minted tokens on Cosmos would be burnt on request to transfer back the original tokens on Ethereum. Similarly, Cosmos assets can be transferred to Ethereum by deploying their respective ERC-20 representations on Ethereum. The Gravity Bridge is a Cosmos-based module that is implemented in Go. On the Ethereum side of the bridge, the `Gravity.sol` smart contract performs Gravity Bridge transactions on the Ethereum blockchain.

The interaction between the Cosmos and Ethereum components of the Gravity Bridge is facilitated and controlled by the Orchestrator, an off-chain component implemented in Rust. The Orchestrator groups transactions into batches in order to reduce gas costs. The batched transactions to be performed are validated by the validator set, a set of Cosmos nodes running the Gravity Bridge module. Two thirds of the validator set must provide approval in order to perform the transactions. Another critical function of the Orchestrator is updating the validator set in the `Gravity.sol` smart contract, the group of addresses that can approve a batch transaction on the Ethereum blockchain.

Our team performed a broad and comprehensive review of the Gravity Bridge system and found that security has been considered in the design of the Gravity Bridge, as demonstrated by a sound design and a commendable effort to break down complex functionality into well-defined components. However, we identified several issues in the design and implementation of the system that could put user assets at

risk. In addition, we identified several suggestions for improvement that will contribute to the quality of the implementation and increase the overall security of the Gravity Bridge system.

Areas of Investigation

We began our review with the `Gravity.sol` smart contract in order to supplement our understanding of the functionalities supported on the Ethereum and Cosmos blockchains. We found that the `Gravity.sol` smart contract is well written and provides valuable insights into the Gravity Bridge system design. We subsequently performed a close examination of the Orchestrator module, which forms an interface between the `Gravity.sol` smart contract and the Gravity Bridge module.

We then closely examined the Gravity Bridge module and its packages. We investigated the transaction batch handling process and its respective components. We checked `OutgoingTxBatchExecuted` for double spend vulnerabilities. In addition, we checked for atomicity and ordering issues within the `Attest` function in the attestation handler for the Keeper. Within the `abci.go` file, we examined validator set management, slashing of malicious validators in transaction batch creation and validator set updates, and EndBlocker calls and attestation handling. We did not identify security vulnerabilities in these areas of the Gravity Bridge module implementation. However, given their security critical functionality, we recommend continuing to closely monitor batch handling, attestations, slashing and validator set voting for potential issues, and vulnerabilities.

In addition, we specifically examined the Gravity Bridge module for instances where voting could be manipulated. Although manipulating voting successfully is likely difficult to execute, attempts could result in unexpected behavior of slashing and batching code. As a result, we advise continued monitoring and testing of all slashing and voting code.

System Design

We found that the general architecture of the Gravity Bridge system is well designed. The system is built of several modules, each of which has a clearly defined role and supports a set of specific functionalities. This facilitated our ability to reason about each individual module and its implementation. However, reasoning about the entire overarching system and the interactions between the different modules introduces additional complexity, thus increasing the difficulty of investigating the security characteristics of the system. Bugs and vulnerabilities in large and complex systems are more difficult to identify and the probability of implementation errors going unnoticed is high. As a result, we advise limiting the complexity of the system to the extent possible as the design and implementation scales.

In our investigation of the system design and implementation, we identified a number of issues and suggestions for improvement.

Handling of User Secrets

We identified a pattern of issues where secret key material is insufficiently secured in the system implementation, as detailed below. The insecure handling of key materials creates potential attack vectors that can lead to the compromise of user private keys, which can result in the total loss of user assets.

In the current implementation, the Orchestrator stores private keys in cleartext without any encryption and the Gravity Bridge module encrypts private keys with hard coded passwords, which can be easily derived and cannot be changed by the user. As a result, an attacker may compromise the filesystem and access user private keys in order to sign transactions, allowing the transfer of tokens between blockchains. As a result, we recommend implementing encrypted storage for secp256k1 private keys in the filesystem for both components, in addition to constraining passwords in adherence with industry standards (Issue G). Moreover, the Orchestrator logs private keys that are generated by the implementation. An attacker could

compromise the secret keys by reading the logs, which would lead to the loss of control of the compromised accounts. We advise against console logging of user secrets or user identifying data in any logs generated by the system (Issue H).

We found that secret key material is not sufficiently cleared from memory, leaving it exposed to attackers. We recommend that unencrypted private keys be cleared from memory appropriately (Issue C). Furthermore, the current `keys.json` file permissions configuration exposes user private keys to anyone with access to the filesystem. We recommend restricting the `keys.json` file permissions configuration to protect user private keys (Issue D).

In addition to insecure handling of secret key material in the implementation, our team found that private keys are used as inputs to derive public keys. Private keys should only be used in the implementation when it is absolutely necessary (e.g. to sign transactions) in order to minimize the attack surface. We advise against the unnecessary use of private keys as argument inputs in the implementation (Suggestion 12).

Block Finality

The Orchestrator component implements a function that sets a block delay 6 blocks deep for the Ethereum mainnet. A block delay reduces the probability of the Gravity Bridge system processing blocks that are not sufficiently finalized (i.e. could change). A low threshold for block confirmations could make the system susceptible to race conditions and double spend attacks. As a result, we recommend increasing the block delay value for the Ethereum mainnet to 12 blocks, in accordance with best practices (Issue B).

In addition, the function `get_block_delay` that sets the block delay in the Orchestrator is called upon each event, which creates unnecessary computation and increases the complexity of the code. We recommend setting the block delay during initialization (Suggestion 6).

Use of Nonce

An `EventNonce` is created by the `Gravity.sol` smart contract upon each event and is used to keep the components of the Gravity Bridge informed of the ordering of events and check the correct ordering of transactions to prevent replay attacks. We examined the `EventNonce` mechanism and did not identify any errors in the implementation. However, for the events `Erc20DeployedEvent` and `ValsetUpdatedEvent`, we recommend implementing a sanity check to confirm that the block height and nonce are within reasonable range to avoid unexpected behavior (Issue F).

In the current implementation, a call is made to a single validator to retrieve the last event nonce upon initialization of the Orchestrator, which enables the node to derive the information on the state of the Gravity Bridge. If this single validator becomes disconnected from the network, relay nodes would not be able to initialize. The validator may also be compromised and its sent value can be incorrect, resulting in increased risk for non-optimal message communications or race conditions. We recommend retrieving the last `EventNonce` from a more trusted source than a single validator (Suggestion 16).

Finally, in order to prevent the potential loss of events, we recommend asserting that events' nonces are not duplicated while forwarding Ethereum events to the Cosmos module (Suggestion 17). Without this check, events with a duplicate nonce will not be forwarded.

Arbitrary Logic Functionality

The Gravity Bridge implements functionality that is capable of performing arbitrary logic for transfers of assets from Cosmos to Ethereum. This feature's powerful functionality extends to potentially delegating all the assets controlled by the system to an arbitrary smart contract. The Althea team has noted that this functionality can only be initiated manually by the Gravity Bridge module or another Cosmos module making a call to the arbitrary logic functionality. We recommend that caution be taken in the use of this

feature as the general-purpose action allowed by the function greatly increases the attack surface. In addition, we recommend any smart contract that is set as a delegate undergo a security audit by an independent security auditing team. Finally, we recommend performing a risk-to-benefit analysis prior to deploying this functionality and more thoroughly documenting the arbitrary logic function (Suggestion 23).

Code Quality

The Gravity Bridge codebase is well organized and generally adheres to best practices. We identified several issues and suggestions in the coded implementation of the Gravity Bridge, as detailed below.

Rust/Orchestrator

We identified an instance of an incorrectly implemented function in the Orchestrator component of the system, which may result in a caller getting an incorrect balance affecting the system logic. We recommend correctly implementing the `get_eth_balances_with_retry` function (Issue A). In addition, the `rust-toolchain` file is missing, which can cause unintended behavior of the Orchestrator. We recommend that the Rust toolchain version be specified in adherence to best practices, as recommended by the Rust documentation (Suggestion 4).

Go/Gravity Bridge Module

In the Gravity Bridge module implementation, some variables and functions are not named explicitly, which may lead to the incorrect use of the function that may result in security issues. We recommend improving the naming convention used for variables and functions such that the names convey their intended behavior. This aids maintainers and security researchers in building a better understanding of the system and minimizes the potential for confusion or errors (Suggestion 13). In addition, the functions are excessively long and we recommend that they be simplified to improve readability and minimize the potential for implementation errors (Suggestion 19).

The compiled `gravity` binary is missing key exploit mitigation mechanisms. Failure to use these security flags in compiling increases the likelihood of an attacker exploiting the `gravity` binary. As a result, we recommend compiling the Go binary with appropriate security flags, in accordance with best practices (Suggestion 2).

Solidity/Gravity.sol and CosmosToken.sol Smart Contracts

There are several opportunities to remove unnecessary computation and storage usage in the `Gravity.sol` smart contract and the `CosmosToken.sol` smart contract, which is imported in `Gravity.sol`, in order to potentially reduce gas cost. We recommend the following optimizations in order to reduce computation:

- Remove `rewardAmount` and `rewardToken` from the inputs to the hash function to compute the checkpoint (Suggestion 18);
- Remove unnecessary constraints (Suggestion 21); and
- Use a `constant` type variable to store a constant value rather than a `storage` type (Suggestion 22).

Error Handling

There are instances of errors being ignored in the implementation, which could cause unexpected behavior and prevent developers from identifying the source of errors. In addition, inappropriate handling of errors increases the difficulty of identifying bugs and implementation errors.

There are multiple instances in the Gravity Bridge module where returned errors are not checked and are ignored, which may lead to undefined behavior. We recommend improving error handling in the Gravity

Bridge module implementation such that all errors are checked and handled appropriately, returning useful information about the cause of the error (Issue E). In addition, the Gravity Bridge module implementation inappropriately triggers a panic rather than handling the error. We recommend removing panics where possible to prevent denial of service if an error occurs (Suggestion 11).

We also found that the Orchestrator implementation does not consistently make use of correct error handling and instead utilizes a panic-based mechanism that may result in denial of service if an error occurs. We recommend handling errors appropriately and avoiding the use of panics as a response to errors (Suggestion 7).

The function `check_if_valsets_differ` is inconsistent with the currently implemented error handling approach and does not adhere to idiomatic Rust code. We recommend improving the error handling implementation, in addition to adding information regarding possible panic to the description of the function (Suggestion 9).

Tests

The `Gravity.sol` smart contract implements sufficient test coverage. However, the Orchestrator and Gravity Bridge module components implement a small percentage of test coverage. While some integration and unit tests exist, we identified an issue (see Issue A) that could have been easily detected, given appropriate unit test coverage. A robust test suite is comprehensive, covers all security critical components, and includes tests for success, failure, and edge cases. This aids developers and security researchers in identifying implementation errors and security vulnerabilities. As a result, we recommend expanding unit and integration tests, specifically around invariants for the arbitrary call logic and batching operations (Suggestion 8).

The Gravity Bridge module and the Orchestrator implement identical functions that must always return identical results. Given that the functions are implemented in different languages, it is uncertain if the functions will consistently return identical outputs. We recommend implementing property based tests to increase confidence that the output of both implementations is always identical (Suggestion 20).

Documentation

The Gravity Bridge project documentation included in the repository is up-to-date and comprehensive. The documentation clearly and accurately explains the different components and their interactions, including reasoning and justification for design decisions. This aided our ability to reason about the security characteristics of the system. However, as previously noted, we recommend expanding the documentation of the arbitrary logic functionality and providing examples of potential use cases (Suggestion 23).

Code Comments

The documentation within the code is sufficient and clearly describes the intended behavior of each of the components that are critical to the functionality and security of the system.

Scope

The in-scope repository was sufficient and included all the security critical components of the Gravity Bridge system. In addition, the Althea team provided clear and explicit key areas of concern regarding the design and implementation of the Gravity Bridge, which aided our team's investigation.

Use of Dependencies

Our team performed `cargo audit` and `go-mod-outdated` tests, which reported several security critical vulnerabilities, including dependencies that have been deprecated. Maintaining up-to-date versions of dependencies, which include fixes to known bugs and security patches for known security

vulnerabilities, minimizes the likelihood of issues being introduced into the codebase. As a result, we recommend updating dependencies and monitoring the latest security developments in the dependencies utilized by the Gravity Bridge system (Suggestion 1; Suggestion 10).

Finally, a deprecated package, `io/ioutil`, is being utilized in the implementation. The use of deprecated packages that are no longer being maintained and updated may result in errors and vulnerabilities. We recommend using similar definitions from the `os` and `io` packages outlined in the Go recommendations (Suggestion 5).

Specific Issues & Suggestions

We list the issues and suggestions found during the review in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Incorrect Implementation of get_eth_balances_with_retry Function	Resolved
Issue B: The Minimum Number of Block Confirmations for Ethereum Mainnet is Low	Resolved
Issue C: Sensitive Information is Not Cleared	Unresolved
Issue D: keys.json File Permissions Configurations for Keys is Insecure	Unresolved
Issue E: Returned Errors in Go are Not Checked	Unresolved
Issue F: Block Height and Event Nonce Not Range-Checked in from_logs	Partially Resolved
Issue G: Private Keys are Stored in Cleartext or Encrypted with a Hard Coded Password	Unresolved
Issue H: Private Keys are Logged to Console	Unresolved
Suggestion 1: Update Outdated Dependencies in Go	Resolved
Suggestion 2: Use Exploit Mitigation Mechanisms for Go Binary	Resolved
Suggestion 3: Warn Users if Transport Layer Security is Not Used	Unresolved
Suggestion 4: Specify Rust Toolchain Version	Unresolved
Suggestion 5: Do Not Use io/ioutil Package	Resolved
Suggestion 6: Get the Block Delay Number on Initialization	Resolved
Suggestion 7: Improve Error Handling and Limit Using Panics in Rust	Unresolved
Suggestion 8: Expand Test Coverage	Unresolved
Suggestion 9: Make check_if_valsets_differ Function Consistent	Unresolved

Suggestion 10: Update Deprecated or Vulnerable Dependencies in Rust	Resolved
Suggestion 11: Improve Error Handling, Limit and Avoid Using Panics in Go	Unresolved
Suggestion 12: Unnecessary Use of Private Key as Function Argument	Resolved
Suggestion 13: Make Variable and Function Naming More Explicit	Unresolved
Suggestion 14: Assert ERC-20 Token and Fee Token to be the Same	Resolved
Suggestion 15: Provide a Refund Mechanism in Case of Failures	Unresolved
Suggestion 16: Retrieve Last Event Nonce from Replicated State on Cosmos	Unresolved
Suggestion 17: Add Assertion to Prevent Event Loss	Resolved
Suggestion 18: Remove (rewardAmount,rewardToken) from Checkpoint Computation	Unresolved
Suggestion 19: Simplify Functions	Partially Resolved
Suggestion 20: Write Tests to Compare Outputs of PowerDiff and power_diff Functions	Unresolved
Suggestion 21: Remove Unnecessary Check	Unresolved
Suggestion 22: Use a Constant Instead of Storage	Resolved
Suggestion 23: Thoroughly Document and Audit the Arbitrary Logic Functionality	Unresolved

Issue A: Incorrect Implementation of `get_eth_balances_with_retry` Function

Location

`orchestrator/gravity_utils/src/get_with_retry.rs#L23`

Synopsis

The `get_eth_balances_with_retry` function calls `web3.eth_get_balance` to get an initial value. It then calls the `web3.eth_block_number` function, which returns the block number rather than the balance.

Impact

The caller would get an incorrect balance, affecting the logic of the system.

Mitigation

We recommend using `web3.eth_get_balance` function on the subsequent call of the `get_eth_balances_with_retry` function.

Status

The Althea team has implemented the `web3.eth_get_balance` function, as suggested.

Verification

Resolved.

Issue B: The Minimum Number of Block Confirmations for Ethereum Mainnet is Low

Location

`orchestrator/orchestrator/src/ethereum_event_watcher.rs#L240-L257`

Synopsis

The number of block confirmations for Ethereum mainnet used by the Gravity Bridge is 6. Consequently, the system is not processing blocks, as they are written to the chain, to update its state. Instead, it is implementing a delay and reading a block once it has been confirmed 6 times, or since 5 more blocks have been written to the chain.

Block finality is a crucial mechanism for blockchain and bridge safety. According to the `overview.md` definition, "EthBlockDelay - Is an agreed upon number of Ethereum blocks all oracle attestations are delayed by. The current value being considered is 50 blocks". However, the `EthBlockDelay` variable and its value are noted only once in the `overview.md` and the codebase does not utilize such a constant or variable.

In the coded implementation, the function is named `get_block_delay` and the description of the function justifies that the block confirmation should be equal to 6 blocks for Ethereum mainnet. The implementation also contains the following statement: "The value used here for Ethereum is a balance between being reasonably fast and reasonably secure". A 6 block finality threshold for Ethereum mainnet is the absolute minimum and could potentially be ineffective in preventing race conditions and double spend attacks.

Impact

A low threshold of block confirmation could result in race conditions and double spend attacks against the Gravity Bridge, which could prevent the system from functioning as intended and lead to the loss of funds.

Feasibility

Since the Ethereum blockchain does not have a finality property, and reorganizations and forks occur regularly, attacks targeting non-repudiation properties are feasible if the number of block confirmations is low.

Remediation

We recommend increasing the block delay value for the Ethereum Mainnet to 12 blocks [R19], in adherence with best practice. We also recommend reconsidering block delay values for other testnets in scope.

Status

The Althea team has increased the block delay value to 13 blocks.

Verification

Resolved.

Issue C: Sensitive Information is Not Cleared

Location

Examples (non-exhaustive):

`orchestrator/gbt/src/orchestrator.rs`

`orchestrator/gbt/src/keys/register_orchestrator_address.rs`

`orchestrator/cosmos_gravity/src/send.rs`

`orchestrator/orchestrator/src/main_loop.rs#L45`

`module/cmd/gravity/cmd/eth_keys.go`

`module/x/gravity/types/ethereum_signer.go`

Synopsis

If an attacker is able to access memory (e.g. accessing core dump, using debuggers, and exploiting vulnerabilities such as Heartbleed), the attacker may be able to retrieve non-zeroized sensitive information in cleartext, such as Ethereum or Cosmos private keys.

Impact

Leakage of Ethereum or Cosmos private keys could result in the loss of user funds.

Preconditions

An attacker must be able to read memory regions containing sensitive data.

Mitigation

We recommend performing zeroization for passwords, local secrets, private keys, authentication tokens, and other sensitive information. In Rust, we recommend using the `zeroize` crate to derive the `zeroize-on-drop` trait. In Go, we recommend using `SetFinalizer` and monitoring the current activities in that area for future improvements.

Status

The Althea team has responded that they plan to implement this mitigation in the future. However, at the time of the verification, the suggested mitigation has not been resolved.

Verification

Unresolved.

Issue D: `keys.json` File Permissions Configuration is Insecure

Location

`orchestrator/gbt/src/config.rs#L39-L57`

Synopsis

The Orchestrator stores private keys in the `keys.json` file with `644` permissions. That file is stored in the `.gbt` directory with `755` permissions inside the user directory. These permissions give any user on the host access to read the private keys.

Impact

An attacker with access to the user's file system can access the user's private keys.

Preconditions

An attacker must be able to have local or remote access to the user's filesystem.

Feasibility

If the preconditions are met, the attacker would need to perform trivial actions in order to steal the private keys.

Mitigation

We recommend creating the `keys.json` file with `600` permissions.

Status

The Althea team has responded that they plan to implement this mitigation in the future. However, at the time of the verification, the suggested mitigation has not been resolved.

Verification

Unresolved.

Issue E: Returned Errors in Go are Not Checked

Location

Examples (non-exhaustive):

`gravity/keeper/pool.go#L194`

`gravity/module.go#L88`

`gravity/abci.go#L172`

`gravity/keeper/genesis.go#L120`

`gravity/keeper/msg_server.go#L49`

`gravity/types/msgs.go#L435`

Synopsis

There are multiple instances in the Gravity Bridge module code where returned errors are not checked and ignored. This may lead to undefined behavior in the case when the result value is *nil* and the error is not *nil* but ignored.

The following is an example of the code where errors are not handled:

```
val, _ := sdk.ValAddressFromBech32(msg.Validator)
orch, _ := sdk.AccAddressFromBech32(msg.Orchestrator)
addr, _ := types.NewEthAddress(msg.EthAddress)
```

The idiomatic coding in Go suggests that the result of a function is unsafe until the error value is checked and then properly handled or propagated.

Impact

The failure to detect and report errors appropriately makes it difficult to identify bugs and implementation errors, which inhibits the implementation of correct functionality.

Mitigation

We recommend appropriately checking and handling or propagating all returned errors. In addition, we recommend integrating Go linters such as `semgrep` or `golangci-lint` with appropriate modules (e.g. ineffassign) and rules to check that all errors are handled correctly.

Status

The Althea team has responded that adding Go linters is in progress. However, it has not yet been fully implemented at the time of verification.

Verification

Unresolved.

Issue F: Block Height and Event Nonce Not Range-Checked in `from_logs`

Location

`orchestrator/gravity_utils/src/types/ethereum_events.rs#L533`

`orchestrator/gravity_utils/src/types/ethereum_events.rs#L202`

Synopsis

In the Orchestrator implementation, the `event_nonce` and `block_height` must be checked to verify if they are constrained to 64 bits in the `from_log` function in the Ethereum events structs `Erc20DeployedEvent` and `ValsetUpdatedEvent`. The code should throw an error if any of these exceeds 64 bits, which is already being done for events like `TransactionBatchExecutedEvent` and `SendToCosmosEvent`.

Impact

In the file `orchestrator/gravity_utils/src/types/ethereum_events.rs`, the events emitted from the Ethereum blockchain are parsed to decode the type of the event and the individual fields as a part of the event. Therefore, this implementation is critical to confirm that the events are correctly parsed. There is no existing Ethereum Application Binary Interface (ABI) unpacking implementation. Thus, it is crucial for the methods in this file to work as expected.

For events like `Erc20DeployedEvent` and `ValsetUpdatedEvent`, a failure to check if the event nonce and the block height are valid 64-bit numbers could lead to unexpected behavior. Gauging the direct impact of the nonce and block height not being range constrained is non-trivial.

Remediation

We recommend adding the error conditions to `Erc20DeployedEvent` and `ValsetUpdatedEvent`. In addition, we recommend maintaining consistency in error reporting in the event that nonce or block height is a number greater than 64 bits for the `Erc20DeployedEvent` and `ValsetUpdatedEvent` events.

Status

The Althea team has implemented checks for `event_nonce` and `block_height` except for this assignment.

Verification

Partially Resolved.

Issue G: Private Keys are Stored in Cleartext or Encrypted with a Hard Coded Password

Location

`orchestrator/gbt/src/config.rs#L118`

`module/cmd/gravity/cmd/eth_keys.go#L31`

Synopsis

The Orchestrator and Gravity Bridge module components store private keys in the filesystem insecurely. The Orchestrator stores private keys in cleartext without any encryption. The Gravity Bridge module encrypts private keys with a hard-coded password that can be derived by anyone and cannot be changed by the user.

Impact

An attacker that is able to gain access to the filesystem can access the user's private keys and sign transactions transferring tokens between blockchains.

Mitigation

We recommend implementing encrypted storage for secp256k1 private keys in the filesystem for both components. In addition, we recommend that passwords adhere to industry standards such as the NIST guidelines on memorized secrets.

Status

The Althea team has responded that this issue is irresolvable given that in all POS networks, validator key security must be balanced against liveness. However, our team recommends that the security of private keys be prioritized and the suggested mitigation be implemented.

Verification

Unresolved.

Issue H: Private Keys are Logged to Console

Location

`orchestrator/gbt/src/keys/register_orchestrator_address.rs`

`orchestrator/gbt/src/keys/register_orchestrator_address.rs`

Synopsis

In the event that a user does not provide private keys, the system generates them and outputs them to console via a logging mechanism. Once complete, the keys cannot be controlled or cleared from the console by the user. However, the secret keys are not persisted in a log file.

Impact

An attacker that is able to gain access to the system can compromise the user's private keys.

Preconditions

An attacker must be able to have local or remote access to the system and be able to read logs outputted to the user's console.

Mitigation

We recommend outputting the private keys into a file with `600` permissions located in the `.gbt` directory. In addition, we suggest outputting a log message containing the path to the file with the private keys and the shell command required to obtain them.

Status

The Althea team has responded that the implemented command generates private keys and those keys are provided to the user through the console by design. However, we believe that the security of the key generation mechanism could be effectively improved with the proposed mitigation.

Verification

Unresolved.

Suggestions

Suggestion 1: Update Outdated Dependencies in Go

Synopsis

The `go-ethereum`, `ibc-go`, `cosmos-sdk`, and `tendermint` dependencies are reported as outdated by `go-mod-outdated` utility and should be updated. Outdated dependencies may have software bugs and vulnerabilities, which may impact the security of the entire system.

Mitigation

We recommend updating the aforementioned dependencies. We also recommend regularly running the `go-mod-outdated` tool to capture any outdated dependencies.

Status

The Althea team has updated the dependencies, as suggested.

Verification

Resolved.

Suggestion 2: Use Exploit Mitigation Mechanisms for Go Binary

Synopsis

The compiled `gravity` binary is missing the `STACK CANARY`, `PIE`, and `FORTIFY` exploit mitigation mechanisms, as reported by the `checksec.sh` utility.

The failure to use these security flags in compiling minimizes the system's security against attackers aiming to exploit the `gravity` binary. The description of the exploit mitigations and compiler flags can be found in the security packaging guideline.

Mitigation

We recommend compiling the Go binary with the corresponding security flags:

- export GOFLAGS='-buildmode=pie'
- export CGO_CPPFLAGS="-D_FORTIFY_SOURCE=2"
- export CGO_LDFLAGS="-Wl,-z,relro,-z,now"
- export CGO_LDFLAGS='-fstack-protector'

Status

The Althea team has implemented the suggested security mechanism by using the flags in compilation.

Verification

Resolved.

Suggestion 3: Warn Users if Transport Layer Security is Not Used

Location

`orchestrator/gravity_utils/src/connection_prep.rs`

Synopsis

The `create_rpc_connections` function establishes Remote Procedure Call (RPC) connections with Ethereum and Cosmos nodes. By default, it connects to the corresponding services on the `localhost`. At the same time, a user may provide addresses of external services located on the Internet. In such an instance, it is critical to access the services over Transport Layer Security (TLS) since a user's node may otherwise establish a network connection with a malicious node.

Mitigation

We recommend establishing connections to remote RPC services over TLS by default. To establish a connection without TLS, a user must be required to enter flag `--without-tls`.

Status

The Althea team has responded that they will not be implementing the suggested mitigation. However, they will automatically upgrade if TLS is available.

Verification

Unresolved.

Suggestion 4: Specify Rust Toolchain Version

Synopsis

A `rust-toolchain` file is missing. An unspecified toolchain version may cause deviations and different behavior of the Orchestrator deployed in different environments.

Mitigation

We recommend placing the `rust-toolchain` file with the explicit Rust toolchain in the Orchestrator directory alongside the `Cargo.toml` file, in adherence with Rust best practices.

Status

The Althea team has responded that they will not be implementing this mitigation.

Verification

Unresolved.

Suggestion 5: Do Not Use `io/ioutil` Package

Synopsis

The Gravity Bridge module component uses the `io/ioutil` package deprecated in Go version 1.16. This package is poorly defined and misused. Using deprecated packages that are no longer maintained may result in security vulnerabilities.

Mitigation

We recommend using similar definitions from the `os` and `io` packages outlined in the Go recommendations:

- `Discard => io.Discard`
- `NopCloser => io.NopCloser`
- `ReadAll => io.ReadAll`
- `ReadDir => os.ReadDir (note: returns a slice of os.DirEntry rather than a slice of fs.FileInfo)`
- `ReadFile => os.ReadFile`
- `TempDir => os.MkdirTemp`
- `TempFile => os.CreateTemp`
- `WriteFile => os.WriteFile`

Status

The Althea team has implemented the suggested mitigation.

Verification

Resolved.

Suggestion 6: Retrieve the Block Delay Number on Initialization

Location

`orchestrator/orchestrator/src/ethereum_event_watcher.rs`

Synopsis

Retrieval of the block delay number is performed in the main loop calling `get_block_delay` function on each event, which results in unnecessary computation.

Mitigation

We recommend retrieving the block delay number once, during the initialization of the system, for the Ethereum blockchain.

Status

The Althea team has implemented the suggested mitigation.

Verification

Resolved.

Suggestion 7: Improve Error Handling and Limit Using Panics in Rust

Location

Examples (non-exhaustive):

`orchestrator/orchestrator/src/oracle_resync.rs`

`orchestrator/relayer/src/main_loop.rs`

`orchestrator/relayer/src/find_latest_valset.rs`

`orchestrator/gravity_utils/src/types/valsets.rs`

`orchestrator/proto_build/src/main.rs`

Synopsis

The Orchestrator does not consistently use correct error handling and uses a panic-based mechanism instead, which may lead to different application behavior for different environments. Triggering a panic by the Orchestrator is semantically equal to a denial of service state. Although panic-based mechanisms could be used in a trusted environment, by design, the considered environment is not trusted since Cosmos validators and Ethereum nodes communicate within a Byzantine Fault Tolerance (BFT) model.

Mitigation

We recommend using explicit error handling based on `Result` type. Panic-based mechanisms should not be called in functions returning a `Result`.

Status

The Althea team has responded that they assume that the user will run the relayer or orchestrator against a trusted Cosmos SDK node.

Verification

Unresolved.

Suggestion 8: Expand Test Coverage

Synopsis

The Orchestrator and the Gravity Bridge module components have insufficient test coverage. The total test coverage for the Go module is only 8% and the total test coverage for the Rust component is only 18%. Furthermore, in the Orchestrator, only the `gravity_utils` package has unit tests. Unit tests assert the behavior of the different components, aiding in the understanding of the intended functionality of these components and the system overall.

Mitigation

We recommend implementing a comprehensive test suite for all system components, including tests for success, failure, and edge cases to check that the implementation behaves as expected and to identify potential errors and vulnerabilities. In particular, we recommend increasing test coverage and invariant testing for the arbitrary call logic system, batching, transferring, and validator set voting functions.

Status

The Althea team has responded that integration tests that cover most of the codebase are implemented. However, we found the tests implemented to still be insufficient.

Verification

Unresolved.

Suggestion 9: Make `check_if_valsets_differ` Function Consistent

Location

`orchestrator/relayer/src/find_latest_valset.rs`

Synopsis

The description of the function `check_if_valsets_differ` states: "This function exists to provide a warning if Cosmos and Ethereum have different validator sets for a given nonce". In addition to warnings, the function can behave in three different ways: panic, log events with two different levels (info and error), and return without logging. Furthermore, the function logs an error but does not propagate the error to the caller. As a result, the function is inconsistent with the current coding style and error handling strategy, and with idiomatic Rust code.

Mitigation

We recommend refactoring the function by adding explicit error handling and adding information regarding possible panic to the description of the function.

Status

The Althea team has responded that they will not be implementing this suggestion.

Verification

Unresolved.

Suggestion 10: Update Deprecated or Vulnerable Dependencies in Rust

Synopsis

The `tempdir`, `stdweb`, and `net2` dependencies are reported as deprecated by `cargo audit` and should be replaced. The `chrono 0.4.19`, `prost-types 0.7.0`, and `time 0.1.43` dependencies are reported as vulnerable by `cargo audit` and should be updated. Regular maintenance and updates of dependencies should be part of the ongoing development process, in order to minimize the risk of introducing known vulnerabilities into the codebase.

Mitigation

We recommend updating or replacing the reported dependencies. If a dependency is used by an upstream dependency, the relevant upstream package should be updated. In addition, we recommend regularly running `cargo audit` and `cargo outdated` tools.

Status

The Althea team has updated dependencies in Rust.

Verification

Resolved.

Suggestion 11: Improve Error Handling, Limit and Avoid Using Panics in Go

Location

Examples (non-exhaustive):

`module/x/gravity/keeper/attestation_handler.go`

`module/x/gravity/keeper/attestation.go`

`module/x/gravity/keeper/cosmos-originated.go#L29`

`module/x/gravity/types/validation.go`

`module/x/gravity/types/msgs.go`

`module/cmd/gravity/cmd/root.go`

Synopsis

There are multiple instances in the Gravity Bridge `module` code triggering a panic if an error occurs. Functions that can cause the code to panic at runtime may lead to denial of service.

Mitigation

We recommend refactoring the code and removing panics where possible. One of the possible improvements is to propagate errors to the caller and handle them on the upper layers. However, error handling does not exclude using panics. In addition, if a caller can return an error, the callee function may not panic but instead propagate an error to the caller.

Status

The Althea team has responded they will not be implementing the suggested mitigation.

Verification

Unresolved.

Suggestion 12: Unnecessary Use of Private Key as Function Argument

Location

Examples (non-exhaustive):

`orchestrator/ethereum_gravity/src/logic_call.rs`

`orchestrator/ethereum_gravity/src/submit_batch.rs`

Synopsis

In some functions, the Ethereum private key and Cosmos secret phrase are passed in as function arguments where they are only used to generate public keys. For functions that need to sign transactions or use the `web3.send_transaction` method, private keys are necessary and must be passed in as function arguments. In all other instances, it should be avoided to reduce the potential attack surface.

Mitigation

We recommend passing corresponding public keys as function arguments (instead of having them derived from the private key) for all instances where private keys are used as function arguments but are not used to sign a transaction.

Status

The public keys are now passed as function arguments rather than being derived from the private key.

Verification

Resolved.

Suggestion 13: Make Variable and Function Naming More Explicit

Location

Examples (non-exhaustive):

`orchestrator/gbt/src/args.rs#L77-L79`

`orchestrator/gbt/src/args.rs#L135-L137`

`orchestrator/gbt/src/args.rs#L110-L112`

`orchestrator/gbt/src/args.rs#L214-L216`

`orchestrator/gbt/src/orchestrator.rs#L28-L29`

`orchestrator/ethereum_gravity/src/utils.rs#L10`

`orchestrator/ethereum_gravity/src/utils.rs#L26`

Synopsis

The naming of variables and functions in the current implementation are not sufficiently explicit, which may prohibit the reader from understanding the intended behavior of the variable or function from its name. For example, in the first two locations above, `ethereum_key` can be changed to `ethereum_private_key`. In the next three locations, `cosmos_phrase` can be changed to `cosmos_secret_phrase`. Finally, in the last two locations, the function `downcast_uint256` can be renamed to `downcast_uint256_to_uint64`, and `downcast_to_u128` to `downcast_uint256_to_u128`.

Mitigation

We recommend improving the naming convention used for variables and functions such that the names convey their intended functionality. This aids maintainers and security researchers in building a better understanding of the system and minimizes the potential for confusion or errors.

Status

The Althea team has responded that this suggestion has been resolved. However, we identified the following areas that would benefit from improved variable and function naming:

- https://github.com/Gravity-Bridge/Gravity-Bridge/blob/42a07e9fb3c95eb7064430bc1a7aba682a7f6a3c/orchestrator/gbt/src/args.rs#L75
- https://github.com/Gravity-Bridge/Gravity-Bridge/blob/42a07e9fb3c95eb7064430bc1a7aba682a7f6a3c/orchestrator/gbt/src/args.rs#L139

- https://github.com/Gravity-Bridge/Gravity-Bridge/blob/42a07e9fb3c95eb7064430bc1a7aba682a7f6a3c/orchestrator/gbt/src/args.rs#L115
- https://github.com/Gravity-Bridge/Gravity-Bridge/blob/42a07e9fb3c95eb7064430bc1a7aba682a7f6a3c/orchestrator/gbt/src/args.rs#L206
- https://github.com/Gravity-Bridge/Gravity-Bridge/blob/42a07e9fb3c95eb7064430bc1a7aba682a7f6a3c/orchestrator/gbt/src/orchestrator.rs#L31-L32
- https://github.com/Gravity-Bridge/Gravity-Bridge/blob/42a07e9fb3c95eb7064430bc1a7aba682a7f6a3c/orchestrator/gravity_utils/src/num_conversion.rs#L23
- https://github.com/Gravity-Bridge/Gravity-Bridge/blob/42a07e9fb3c95eb7064430bc1a7aba682a7f6a3c/orchestrator/gravity_utils/src/num_conversion.rs#L39

Verification

Unresolved.

Suggestion 14: Assert ERC-20 Token and Fee Token to be the Same

Location

`orchestrator/gravity_utils/src/types/batches.rs#L137-L152`

Synopsis

An individual transaction to be sent from Cosmos to Ethereum is represented by the struct `BatchTransaction` in the Orchestrator. Groups of `BatchTransactions` are batched together in a struct called `TransactionBatch`. The actual transactions are not batched by the Orchestrator. The method `try_from` fills the fields in a `BatchTransaction` from a given `OutgoingTransferTx` (which is defined in `gravity/types/batch.pb.go`).

For a given transaction from Cosmos to Ethereum, the fee token and the ERC-20 token to be transferred must be the same (i.e. the fee is to be paid to the relayer in the same ERC-20 token). This constraint is enforced in the `send_to_eth` method in `orchestrator/src/send.rs`.

To achieve correctness and completeness, the same constraint should be enforced while creating a `BatchTransaction` from `OutgoingTransferTx`. Implementing the necessary constraints in all relevant locations is a best practice as long as it does not hamper performance.

Mitigation

We recommend adding the constraint to check the equality of the ERC-20 token smart contract and the fee token smart contract at the aforementioned location in Orchestrator.

Status

The Althea team has added the necessary check in the `try_from` method in the `BatchTransaction`.

Verification

Resolved.

Suggestion 15: Provide a Refund Mechanism in Case of Failures

Synopsis

In the absence of a refund mechanism, failures, errors, or subtle bugs in the current implementations of the Orchestrator, Gravity Bridge module, and `Gravity.sol` smart contract components may lead to tokens being locked indefinitely.

Mitigation

We recommend designing and implementing a refund mechanism that would allow users to refund their locked tokens.

Status

The Althea team has responded that they will not be implementing the suggested mitigation as locked tokens can be retrieved via arbitrary logic or governance.

Verification

Unresolved.

Suggestion 16: Retrieve Last Event Nonce from Replicated State on Cosmos

Synopsis

A relayer gets the last event nonce from the corresponding validator using the `get_last_event_nonce_for_validator` function. According to the documentation, each Orchestrator is expected to run against a trusted full node. However, that does not necessarily mean that each validator will have the real (committed) last nonce. For example, a validator may not have the real last nonce due to network partitioning or recovering from a crash.

If this single validator becomes disconnected from the network, relay nodes would not be able to initialize. The validator may also be compromised and its sent value can be incorrect, resulting in increased risk for non-optimal message communications or race conditions. We recommend retrieving the last `EventNonce` from a more trusted source than a single validator.

Mitigation

We recommend retrieving last event nonces from the replicated state agreed on by the honest majority of validators.

Status

The Althea team has responded that this will require the implementation of a light client. While this is planned, it has not yet been implemented at the time of the verification.

Verification

Unresolved.

Suggestion 17: Add Assertion to Prevent Event Loss

Location

`cosmos_gravity/src/send.rs#L214`

Synopsis

In `send_ethereum_claims`, the implementation takes events from `deposits`, `withdraws`, `erc20_deploys`, `logic_calls`, and `valsets` and inserts them into a `HashMap` with event nonces as keys. If a key already existed from one of the previous insertions, the event will be lost with subsequent insertions, which would cause events on the `Gravity.sol` smart contract to not be forwarded to the Cosmos module.

Mitigation

We recommend asserting that there were no elements for a given key while filling the `HashMap`.

For example:

```
assert!(unordered_msgs.insert(deposit.event_nonce, msg).is_none());
```

This would cause a panic in the Orchestrator in the event that such an input is given to the `send_ethereum_claims` function. Alternatively, the implementation could return an `Error` that provides useful information to the caller.

Status

In all insertions to the event `HashMap`, `unordered_msgs` assert that there was no value before the insertion. This would cause a panic in the case where events with duplicate nonces are being processed.

Verification

Resolved.

Suggestion 18: Remove (`rewardAmount`, `rewardToken`) from Checkpoint Computation

Location

[solidity/contracts/Gravity.sol#L183-L204]

Synopsis

A checkpoint is a hash of all relevant information about a validator set. A validator set consists of the following:

```
struct ValsetArgs {
    // the validators in this set, represented by an Ethereum address
    address[] validators;
    // the powers of the given validators in the same order as above
    uint256[] powers;
    // the nonce of this validator set
    uint256 valsetNonce;
    // the reward amount denominated in reward token, can be set to zero
    uint256 rewardAmount;
    // the reward token, should be set to the zero address if not being
used
    address rewardToken;
}
```

A checkpoint can be completely defined from the list of validators, their powers, and the valset nonce:

```
checkpoint = sha3(gravityId, "checkpoint", valsetNonce, validators[],
powers[])
```

However, in the implementation, the `rewardAmount` and `rewardToken` are also included in the inputs to the hash function:

[althea-net-gravity-private/solidity/contracts/Gravity.sol]

```
bytes32 checkpoint = keccak256(
      abi.encode(
            _gravityId,
            methodName,
            _valsetArgs.valsetNonce,
            _valsetArgs.validators,
            _valsetArgs.powers,
            _valsetArgs.rewardAmount,
            _valsetArgs.rewardToken
      )
);
```

The `rewardAmount` and `rewardToken` are the reward parameters paid to the one who calls the function `updateValset` to update the validator set. These quantities need not be included in the checkpoint computation because the data (`validators`, `powers`, `valsetNonce`) is sufficient to uniquely encode the information of a validator set at a given time.

Mitigation

We recommend removing `rewardAmount` and `rewardToken` from the inputs to the hash function to compute the checkpoint in order to conform to the expected definition of checkpoint and save a small amount of gas.

Status

The Althea team has acknowledged the suggestion and responded that they will not implement the suggested mitigation.

Verification

Unresolved.

Suggestion 19: Simplify Functions

Location

Examples (non-exhaustive):

`module/app/app.go#L217-L556`

`module/x/gravity/keeper/attestation_handler.go#L36-L260`

`module/cmd/gravity/cmd/gentx.go#L42-L245`

Synopsis

There are a number of long functions within the codebase. For example, the first function noted above is 340 lines in length, the second is 225 lines in length, and the third is 204 lines in length. It is considered best practice to limit function size for readability and to encourage better code design (i.e. small functions composed together).

Mitigation

We recommend refactoring long functions by decomposing them such that functions fit within the size of a standard screen.

Status

The Althea team has partially resolved this suggestion. The functions `NewGravityApp` and `GenTxCmd` remain to be refactored.

Verification

Partially Resolved.

Suggestion 20: Write Tests to Compare Outputs of PowerDiff and power_diff Functions

Location

`module/x/gravity/types/validation.go#L98-L136`

`orchestrator/gravity_utils/src/types/valsets.rs#L243-L288`

Synopsis

The `PowerDiff` (Cosmos module) and `power_diff` (Orchestrator) functions compute the difference in validator set voting power and have been identified by the Althea team as susceptible to vulnerabilities resulting from logic errors. The comment for `power_diff` reads "In theory an error here, if unnoticed for long enough, could allow funds to be stolen from the bridge without the validators in question still having stake to lose".

It is important that the two implementations of the function generate the same outputs so that the Gravity Bridge module and Orchestrator components have a shared understanding of when a validator set should be updated. However, in addition to being written in two different programming languages, the implementations use different algorithms to generate their results, resulting in increased difficulty to confirm that they are in fact the same function.

Mitigation

We suggest writing property based tests to compare the outputs of both functions in order to increase confidence that they generate the same outputs. In addition, we suggest considering using the same algorithm in both languages to reduce the risk of them producing different results.

Status

The Althea team has responded that the implementation of this mitigation is in progress. However, at the time of the verification, the suggested mitigation has not been resolved.

Verification

Unresolved.

Suggestion 21: Remove Unnecessary Check

Location

`solidity/contracts/Gravity.sol#L277-L282`

Synopsis

The `Gravity.sol` smart contract implements a constraint that the length of the validators array does not equal 0. However, the sum of the power array that has the same length should be bigger than the

`uint256` constant `constant_powerThreshold` found in solidity/contracts/Gravity.sol#L289-L301. As a result, the check 0 is unnecessary.

A similar check is found in solidity/contracts/Gravity.sol#L622-L624. These checks increase gas cost but do not enhance the security of the implementation.

Mitigation

We recommend removing unnecessary checks.

Status

The Althea team has responded that the implementation of this mitigation is in progress. However, at the time of the verification, the suggested mitigation has not been resolved.

Verification

Unresolved.

Suggestion 22: Use a Constant Instead of Storage

Location

solidity/contracts/CosmosToken.sol#L6

Synopsis

A `storage` type variable is used for storing a constant value and reading it, which results in unnecessary gas consumption.

Mitigation

We recommend using the `constant` type to store constant values.

Status

The Althea team has converted the variable type used for storing constants to a `constant` type.

Verification

Resolved.

Suggestion 23: Thoroughly Document and Audit the Arbitrary Logic Functionality

Location

docs/design/arbitrary-logic.md

Synopsis

The Gravity Bridge includes functionality to make arbitrary calls out to other Ethereum smart contracts. This can be used to allow the Cosmos blockchain to take general-purpose actions on Ethereum. Allowing general-purpose actions increases the attack surface.

The concept of a bridge to trade assets between two blockchains is conceptually simple. It is also security sensitive because it will be handling large amounts of value. As a result, the implementation should remain simple to the extent possible and act as a building block upon which higher order functionality can be composed for more general behavior.

The Gravity Bridge's use cases requiring the arbitrary logic functionality, and the security implications of such a general-purpose mechanism, remain unclear to our team.

Mitigation

We recommend that caution be taken in the use of this feature as the general-purpose action allowed by the function significantly increases the attack surface. Specifically, we recommend the following:

- Conducting a comprehensive security audit of any smart contract that is set as a delegate by an independent security auditing team;
- Performing a rigorous risk-to-benefit analysis prior to deploying this arbitrary functionality; and
- Thoroughly documenting the arbitrary logic function such that it includes use cases and a comprehensive explanation of its intended use.

Status

The Althea team responded that the implementation of the mitigation is in progress. However, at the time of the verification, the suggested mitigation has not been resolved.

Verification

Unresolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.