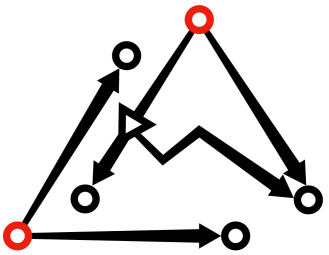

Least Authority

PRIVACY MATTERS

Peggo Orchestrator

Security Audit Report

Umee

Updated Final Audit Report: 6 June 2022

Table of Contents

Overview

Background

Umee has requested that Least Authority perform a security audit of the Peggo Orchestrator.

Project Dates

- **February 28 - March 25**: Initial Review *(Completed)*
- **March 30**: Delivery of Initial Audit Report *(Completed)*
- **April 27 - 28**: Verification Review *(Completed)*
- **April 29**: Delivery of Final Audit Report *(Completed)*
- **June 7:** Delivery of Updated Final Audit Report *(Completed)*

Review Team

- David Braun, Security Researcher and Engineer
- Gabrielle Hibbert, Security Researcher and Engineer
- Denis Kolegov, Security Researcher and Engineer
- Nishit Majithia, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Peggo Orchestrator 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:

- Peggo Orchestrator: https://github.com/umee-network/peggo
- Gravity Bridge: https://github.com/Gravity-Bridge/Gravity-Bridge

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

Peggo Orchestrator: `2e6b903c038df3cc1d3195555c0247ce6b1e04d2`

Gravity Bridge: `f4be2bb36f48c32622080b2fe23b0e759cca9389`

For the verification, we examined the Git revisions:

Peggo Orchestrator: `53e9a8e95a4367426f719e6e8f9db115e2e11f95`

Gravity Bridge: `780daf986516905af4b12373f96935dfaf856999`

For the review, these repositories were cloned for use during the audit and for reference in this report:

Peggo Orchestrator: https://github.com/LeastAuthority/Peggo-Orchestrator

Gravity Bridge: https://github.com/LeastAuthority/Gravity-Bridge-

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/umee-network/peggo/tree/a4ef47c4a380bcada75aff894bece34e9d13be63#readme
- Umee Whitepaper: https://umee.cc/umee-whitepaper/
- Umee Docs: https://docs.umee.cc/umee/
- Umee blockchain - Final Report Draft.pdf *(shared with Least Authority via Slack on 23 February 2022)*
- Peggo's main loops document *(shared with Least Authority via Slack on 16 March 2022)*

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 Orchestrator;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service attacks and security exploits that would impact or disrupt execution of the Orchestrator;
- 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

Our team performed a 4 week timebox review of Umee's Peggo Orchestrator, a Go implementation of the original Gravity Bridge Orchestrator implemented by Althea. Peggo is an off-chain relayer that facilitates the transfer of ERC-20 and Cosmos SDK based tokens between the Ethereum and Umee blockchains. Peggo is operated by a set of validators that scan for events emitted by the Solidity smart contract component of the system and relay the events as messages to Umee. Similarly, transactions originating on the Umee blockchain are grouped into batches to reduce transaction costs, and are relayed to the smart contract component of the bridge upon approval. In addition, the Peggo Orchestrator updates the Gravity Bridge smart contract on Ethereum with the latest validator set changes on the Cosmos blockchain.

Peggo bridges two different blockchains that use two types of consensus algorithms and borrows the authority mechanism from Cosmos (validators) to use on Ethereum. In particular, the algorithm for updating validator set changes has to accommodate the differences (e.g. speed, finality, etc.) between the two blockchains. As a result, it is challenging for the Cosmos side of the bridge to be in sync with the Ethereum side.

We closely investigated the functionality in the implementation responsible for sending updates of the validator set from Cosmos to the Ethereum smart contracts. Specifically, we checked whether the updates could be sent in the wrong order, or if the updates could cause any kind of disruption to the bridge's functionality. We did not identify any vulnerabilities in the functionality updating the validator set. However, we suggest two possible approaches to implement additional safeguards to increase the safety of this mechanism.

The hash of the set of Peggo validators is stored in the Ethereum smart contract component of the system and is used to verify the authenticity of transaction requests sent by Peggo. This validator set must be ordered consistently to produce the same hash that is stored in the smart contract. Incorrect ordering would result in a hash being rejected by the smart contract, which can stop the bridge or prevent it from functioning as intended. As a result, the ordering of the validator set should be checked before a hash is generated. We recommend two alternative approaches to verify the correct ordering of the validators (Suggestion 1; Suggestion 3).

System Design

We found that security has been taken into consideration in the design and implementation of the Peggo Orchestrator as demonstrated by a codebase that is written in a defensive style and implements checks for possible insecure scenarios (e.g. a hijacking of the validator set). The design follows the original Rust implementation of the Orchestrator in the Althea Gravity Bridge. The translation to Go reduces the number of programming languages used in the bridge from three to two, which reduces the risk of unexpected behavior in the interaction between the components of the bridge. However, we identified several issues and suggestions, as detailed below, which if resolved, will result in a more robust implementation and improve the overall security of the system.

Upon set up of the Peggo Orchestrator, the Cosmos private key is input into the command line interface. The command line interface is vulnerable to snooping whereby an attacker can read command line processes without any privileges. It is a recommended practice to avoid passing secrets in the command line. We recommend utilizing the system keyring to provide the private key (Issue B).

Recent successful bridge attacks have culminated in the attacker draining bridge assets in one large transaction. As a result, we recommend implementing a feature to detect and flag such large transactions so that preventative action can be taken by the validators (Suggestion 5).

The Peggo Orchestrator depends on a single price oracle (CoinGecko), which is a single point of failure (SPOF). A disruption in this service could prevent the Peggo Orchestrator from functioning, or in the case of inaccurate price information, could result in unprofitable transactions being performed. We recommend that price information from more than one oracle be used to prevent system dependence on a SPOF (Issue A).

Code Quality

We found the Peggo Orchestrator implementation to be well-organized. However, there are several functions in the codebase that are excessively large. This reduces readability of the code and as a result, makes reasoning about the security of these functions and their interaction with the rest of the system more difficult. We recommend adhering to best practices and refactoring large functions (Suggestion 4).

Furthermore, we found instances of errors utilizing panics, which can lead to failure. We recommend that errors be handled appropriately and return useful information about the cause of the error, and that the use of panic be avoided in accordance with best practice (Suggestion 2).

Tests

The Peggo Orchestrator has a minimal number of tests, with only 10% of the codebase covered. A robust test suite tests for success and failure cases, which helps to identify potential edge cases, and helps prevent errors and bugs, which may lead to vulnerabilities or exploits. We recommend improving the test suite (Suggestion 7).

Documentation

The project documentation includes a simple but accurate README file that describes the set up and use of the Peggo Orchestrator. However, we found the documentation to be insufficient in describing the general design of the system and how that design is realized in the implementation. We recommend creating additional developer documentation including an architecture diagram and detailed descriptions of the system components and how they interact with each other. This will enhance developer comprehension of this security sensitive code (Suggestion 6).

Code Comments

The documentation within the codebase provides sufficient description of the intended behavior of critical functions and components.

Scope

Although, the in-scope repository for this timebox review was sufficient, we recommend a comprehensive review of the entire Peggo system. The Peggo system is composed of a relatively large codebase (20K lines of code). As a result, our time constrained review limited our team's ability to sufficiently review the entire scope as thoroughly as preferred.

Dependencies

Our team did not identify any security vulnerabilities in the use of dependencies. However, the dependency `google.golang.org/grpc` is 12 minor versions out of date. Up-to-date dependencies include bug fixes and security patches that can improve the overall security of the implementation.

Specific Issues

Issues are vulnerabilities our researchers identified that may compromise the security of the system. In the table below, we list the issues 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	STATUS
Issue A: Griefing Attack Possible from Use of Single Price Oracle	Completed
Issue B: Cosmos Account Private Key is Passed as Command Line Argument	Completed

Issue A: Griefing Attack Possible from Use of Single Price Oracle

Location

orchestrator/coingecko/coingecko.go

Synopsis

The Peggo Orchestrator sends transactions from Cosmos to Ethereum in batches. Before sending a batch, each relayer first determines if the batch will be profitable based on the expected cost of the transactions and the included fees in the batch. The batch cost is estimated by using pricing data from the CoinGecko oracle. The over-reliance on one oracle introduces a SPOF, which could compromise the Peggo Orchestrator by accident or through a griefing attack.

Impact

If the single price oracle malfunctioned or was attacked, it is possible that batches of transactions may not be relayed because they are deemed unprofitable, disrupting the functionality of the bridge. It is also possible that unprofitable batches may be relayed, draining value from the system (a griefing attack).

Preconditions

The CoinGecko oracle (or the communication channel to it) is compromised.

Feasibility

There is precedent for a price oracle malfunctioning and resulting in a significant impact (see the Synthetix oracle incident).

Remediation

We recommend sampling prices from several oracles, discarding any outliers, and using the average of the remaining prices.

Status

The Umee team has generalized the oracle code to sample multiple feeds.

Verification

Resolved.

Issue B: Cosmos Account Private Key is Passed as Command Line Argument

Location

cmd/peggo/flags.go#L76

Synopsis

The Peggo Orchestrator command line interface takes the following argument:

```
    --cosmos-pk string   Specify a Cosmos account private key of the
orchestrator in hex
```

Passing secrets on the command line is not a recommended practice as it is vulnerable to snooping.

Impact

An attacker in control of the Cosmos private key could move the tokens on the Cosmos blockchain to other accounts, draining the bridge of its funds.

Preconditions

An attack requires access to a Peggo Orchestrator validator's machine as an unprivileged user.

Feasibility

Given access to a Peggo validator's machine, it would be relatively straightforward to execute the attack.

Technical Details

An attacker with access to the validator's machine could use a tool such as pspy to steal the private key from the command line argument.

Remediation

We recommend that the Cosmos private key be retrieved from the system keyring rather than being input as a command line argument.

Status

The Umee team updated the code to now accept the private key through an environment variable instead of an argument on the command line.

Verification

Resolved.

Specific Suggestions

Suggestions provide improvements to the overall security and quality of the implementation and better adherence to industry standards and best practices. Suggestions are not security critical, and therefore accepting or rejecting them does not have a significant security impact on the system. In the table below, we list suggestions found during the review, in the order we reported them.

SUGGESTION	STATUS
Suggestion 1: Use Differential Testing to Verify Compatibility of Gravity Bridge and Peggo Orchestrators	Planned
Suggestion 2: Improve Error Handling, Limit and Avoid Using Panics	Planned
Suggestion 3: Ensure Validators Are Ordered Correctly During Valset Validation	Not Completed
Suggestion 4: Refactor Large Functions	Planned
Suggestion 5: Consider Delaying Large Transactions and Adding Circuit Breakers	Planned
Suggestion 6: Improve Project Documentation	Planned
Suggestion 7: Increase Test Coverage	Planned

Suggestion 1: Use Differential Testing to Verify Compatibility of Gravity Bridge and Peggo Orchestrators

Synopsis

The Peggo Orchestrator is a Go implementation of Althea's Gravity Bridge Orchestrator, which is implemented in Rust. As a result, the Gravity Bridge must support two implementations of the same component. It is known that different implementations of the same protocol or mechanism defined semi-formally or even formally (TCP, web server, load balancer, etc.) can influence the security of each other in a positive way.

Mitigation

We recommend implementing a combination of two approaches:

- differential testing and/or differential fuzzing
- lightweight formal verification

The first entails creating two testnets: the first consisting of orchestrators implemented in Go and the second of orchestrators implemented in Rust. The target property to be tested is that both implementations behave identically if inputs and events (e.g. faults) are the same.

Secondly, both the Gravity Bridge (Rust) and Peggo Orchestrator (Go) implementations could be deployed and used interchangeably since they must implement the same interfaces. This approach is used by Ethereum, which has several different clients.

Status

The Umee team has replied that the mitigation will be implemented in the future.

Verification

Unresolved.

Suggestion 2: Improve Error Handling, Limit and Avoid Using Panics

Location

Examples (non-exhaustive):

`orchestrator/coingecko/coingecko.go:40`

`orchestrator/coingecko/coingecko.go:54`

`orchestrator/coingecko/coingecko.go:66`

`orchestrator/coingecko/coingecko.go:103`

`orchestrator/coingecko/coingecko.go:115`

`orchestrator/ethereum/gravity/message_signatures.go:22`

`orchestrator/ethereum/gravity/message_signatures.go:44`

`orchestrator/ethereum/gravity/message_signatures.go:66`

orchestrator/ethereum/gravity/message_signatures.go:84

orchestrator/relayer/main_loop.go:42

orchestrator/main_loops.go:63

orchestrator/oracle_resync.go:183

orchestrator/ethereum/gravity/pending_transactions.go:77

orchestrator/ethereum/gravity/pending_transactions.go:87

Synopsis

There are multiple instances in the Peggo Orchestrator code triggering a panic or exit if an error occurs. Functions that can cause the code to panic at runtime may lead to total denial of service or impact the liveness of the system.

Mitigation

We recommend refactoring the code and removing all panics and fatals where possible. One of the possible improvements is to propagate errors to the caller and handle them on the upper layers. Note that error handling does not exclude using panics. In addition, if a caller returns an error, the callee function may not panic, but instead propagate an error to the caller. Panics and fatals should be used in the following cases: initialization code, CLI code, or if a potential safety violation can occur.

Status

The Umee team has responded that the mitigation will be implemented in the future.

Verification

Unresolved.

Suggestion 3: Ensure Validators Are Ordered Correctly During Valset Validation

Location

Gravity-Bridge-/blob/master/solidity/contracts/Gravity.sol#L164-L173

Synopsis

Validator sets are frequently updated on the Cosmos blockchain and updates are relayed to the Ethereum smart contract to keep it in sync. To save gas, the Ethereum smart contract stores a hash of the validator set (the *checkpoint*) rather than storing the entire set. If a validator set is sent using the same validators but in a different ordering than in the previous update, the `checkpoint` will not match and the update will fail, which can stop the bridge or prevent it from functioning as intended. Checking the order of the validators in each update set can act as a safeguard to prevent this failure.

Mitigation

We recommend the addition of code to the `validateValset` function to check the order of the validators:

```
        // Check that the validators are sorted by power and address.
        for (uint256 i = 0; i < _newValset.validators.length - 1; i++) {
          if (
            _newValset.powers[i] > _newValset.powers[i + 1] ||
```

```
            (
              _newValset.powers[i] == _newValset.powers[i + 1] &&
              _newValset.validators[i] >= _newValset.validators[i + 1]
            )
          ) {
            revert MalformedNewValidatorSet();
          }
        }
```

We have provided sample code in the form of a pull request.

Status

Since the Gravity.sol contract of the Gravity Bridge is deployed as non-updatable, the Umee team has responded that they cannot implement the suggested mitigation.

Verification

Unresolved.

Suggestion 4: Refactor Large Functions

Location

Examples (non-exhaustive):

`getOrchestratorCmd`

`initEthereumAccountsManager`

`deployERC20Cmd`

Synopsis

There are several large functions within the codebase that can be refactored into smaller functions to improve the overall quality and readability of the code. Excessively large functions reduce readability and increase the possibility of implementation errors going unnoticed. Furthermore, smaller functions make the code more structured and easier to change.

Mitigation

We recommend that large functions be refactored into multiple smaller ones and that the implementation adhere to the informal guidelines that a function should not exceed a single text editor page.

Status

The Umee team has responded that the mitigation will be implemented in the future.

Verification

Unresolved.

Suggestion 5: Consider Delaying Large Transactions and Adding Circuit Breakers

Synopsis

Bridges between blockchains can carry very high amounts of value by design. If the Umee Bridge is compromised, the attacker may attempt to remove large amounts of assets quickly, as was done in

attacks on the Ronin (USD 625 million) and Solana Wormhole (USD 326 million) bridges. A circuit breaker mechanism could have detected large transactions and interrupted the transfers.

Mitigation

We recommend adding a new feature that would detect and flag large or suspicious transactions, or a rapid decrease in the bridge's fund balance, for manual review.

Status

The Umee team has not implemented the suggested mitigation, noting it will be technically challenging because of interactions with the validator slashing mechanism. However, they stated this may be completed in the future.

Verification

Unresolved.

Suggestion 6: Improve Project Documentation

Synopsis

Our team found the project documentation insufficient in accurately describing the system, its components, and internal and external interactions. Comprehensive documentation should provide developers, reviewers, and users of the system an accurate and detailed explanation of the system that facilitates understanding and reasoning about the system's security characteristics. This helps improve the security of the protocol, and gives users the necessary information to make informed decisions.

Mitigation

We recommend that developer documentation, including an architectural diagram, be created as well as a detailed explanation of how the Peggo Orchestrator functions and interacts with the bridge at large.

Status

The Umee team has responded that the mitigation will be implemented in the future.

Verification

Unresolved.

Suggestion 7: Increase Test Coverage

Synopsis

The Peggo Orchestrator implements insufficient test coverage. Comprehensive test coverage helps to identify implementation errors and to check that components behave as intended. In addition, a comprehensive test suite allows developers and security researchers to gain a better understanding of the functionality of the system, facilitating the discovery of potential security vulnerabilities.

Mitigation

We recommend increasing test coverage to test for success, failure, and edge case scenarios.

Status

The Umee team has responded that the mitigation will be implemented in the future.

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.