



Umee

Security Assessment

March 14, 2022

Prepared for:

Brent Xu

Umee

Prepared by:

Dominik Czarnota and Paweł Płatek

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at <https://github.com/trailofbits/publications>, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow [@trailofbits](#) on Twitter and explore our public repositories at <https://github.com/trailofbits>. To engage us directly, visit our "Contact" page at <https://www.trailofbits.com/contact>, or email us at info@trailofbits.com.

Trail of Bits, Inc.

228 Park Ave S #80688

New York, NY 10003

<https://www.trailofbits.com>

info@trailofbits.com

Notices and Remarks

Copyright and Distribution

© 2022 by Trail of Bits, Inc.

All rights reserved. Trail of Bits hereby asserts its right to be identified as the creator of this report in the United Kingdom.

This report is considered by Trail of Bits to be public information; it is licensed to Umee under the terms of the project statement of work and has been made public at Umee's request. Material within this report may not be reproduced or distributed in part or in whole without the express written permission of Trail of Bits.

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and mutually agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

About Trail of Bits	1
Notices and Remarks	2
Table of Contents	3
Executive Summary	6
Project Summary	9
Project Goals	10
Project Targets	11
Project Coverage	12
Automated Testing	14
Codebase Maturity Evaluation	16
Summary of Findings	19
Detailed Findings	22
1. Integer overflow in Peggo's deploy-erc20-raw command	22
2. Rounding of the standard deviation value may deprive voters of rewards	24
3. Vulnerabilities in exchange rate commitment scheme	27
4. Validators can crash other nodes by triggering an integer overflow	30
5. The repayValue variable is not used after being modified	32
6. Inconsistent error checks in GetSigners methods	33
7. Incorrect price assumption in the GetExchangeRateBase function	34
8. Oracle price-feeder is vulnerable to manipulation by a single malicious price feed	36
9. Oracle rewards may not be distributed	37
10. Risk of server-side request forgery attacks	38

11. Incorrect comparison in SetCollateralSetting method	39
12. Voters' ability to overwrite their own pre-votes is not documented	40
13. Lack of user-controlled limits for input amount in LiquidateBorrow	42
14. Lack of simulation and fuzzing of leverage module invariants	43
15. Attempts to overdraw collateral cause WithdrawAsset to panic	46
16. Division by zero causes the LiquidateBorrow function to panic	48
17. Architecture-dependent code	50
18. Weak cross-origin resource sharing settings	51
19. price-feeder is at risk of rate limiting by public APIs	52
20. Lack of prioritization of oracle messages	53
21. Risk of token/uToken exchange rate manipulation	54
22. Collateral dust prevents the designation of defaulted loans as bad debt	56
23. Users can borrow assets that they are actively using as collateral	57
24. Providing additional collateral may be detrimental to borrowers in default	58
25. Insecure storage of price-feeder keyring passwords	59
26. Insufficient validation of genesis parameters	61
27. Potential overflows in Peggo's current block calculations	62
28. Peggo does not validate Ethereum address formats	63
29. Peggo takes an Ethereum private key as a command-line argument	65
30. Peggo allows the use of non-local unencrypted URL schemes	66
31. Lack of prioritization of Peggo orchestrator messages	67
32. Failure of a single broadcast Ethereum transaction causes a batch-wide failure	68
33. Peggo orchestrator's IsBatchProfitable function uses only one price oracle	69
34. Rounding errors may cause the module to incur losses	71

35. Outdated and vulnerable dependencies	74
A. Vulnerability Categories	76
B. Code Maturity Categories	78
C. Calculating a Dec Value's Square Root	80
D. Incorrect Exchange-Rate Parsing	82
E. Code Quality Recommendations	83
F. Automated Testing	86
G. Fixed-Point Rounding Recommendations	87
H. Token Integration Checklist	89
I. Fix Log	94

Executive Summary

Engagement Overview

Umee engaged Trail of Bits to review the security of the Umee blockchain and Peggo orchestrator. From January 17 to February 11, 2022, a team of two consultants conducted a security review of the client-provided source code, with eight person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered a few significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	Count
High	6
Medium	9
Low	6
Informational	8
Undetermined	6

CATEGORY BREAKDOWN

Category	Count
Data Validation	17
Configuration	9
Undefined Behavior	2
Timing	2
Data Exposure	2
Testing	1
Cryptography	1
Patching	1

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-UMEE-11**
An invalid comparison in the `SetCollateralSetting` method could allow an attacker to disable the use of an asset as collateral and to regain the collateral he or she had provided when borrowing coins.
- **TOB-UMEE-21**
The token/uToken exchange rate may be susceptible to unexpected manipulation, which would violate system invariants and enable certain malicious behavior.
- **TOB-UMEE-7**
The `GetExchangeRateBase` function returns an exchange rate of 1 if the denominator contains the string "USD." As a result, the function may return incorrect prices for assets like stablecoins.
- **TOB-UMEE-8**
The price sent to the oracle could be manipulated by a single compromised or malfunctioning price provider if most of the validators used that provider.

- **TOB-UMEE-14**
The system lacks comprehensive simulation and invariant testing, which would aid in checking various edge cases as the system is further developed.
- **TOB-UMEE-3**
Vulnerabilities in the exchange rate commitment scheme could allow an attacker to predict the prices submitted by other voters; alternatively, if Umee increased the salt length without applying other fixes, an attacker would be able to send two prices for an asset in a pre-vote message hash and to then commit to either one.
- **TOB-UMEE-4**
By sending a large exchange rate value, a validator could cause an overflow in the standard deviation calculation, which would result in a panic and a node crash.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineers were associated with this project:

Dominik Czarnota, Consultant
dominik.czarnota@trailofbits.com

Paweł Płatek, Consultant
pawel.platek@trailofbits.com

Project Timeline

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

Date	Event
December 21, 2021	Pre-project kickoff call
January 13, 2022	Project kickoff call
January 24, 2022	Status update meeting #1
January 31, 2022	Status update meeting #2
February 7, 2022	Status update meeting #3
February 14, 2022	Final report readout
March 14, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Umee blockchain and its Peggo orchestrator. Specifically, we sought to answer the following non-exhaustive list of questions:

- Could users steal funds, or could their funds be lost or locked?
- Could users, validators, or third parties manipulate asset prices in unexpected ways?
- Are there any issues in the design of the leverage module?
- Could a validator discern an exchange rate submitted by another validator during pre-voting?
- Could a validator manipulate the exchange rate commitment scheme?
- Are oracle messages prioritized for inclusion in a block (to mitigate spam attacks)?
- Are all inputs and system parameters properly validated?
- Are the arithmetic operations correct and prevented from overflowing?
- Can third-party integrators rely on the system to emit events when appropriate?
- Could the system experience a denial of service?
- Does the test suite cover malicious inputs and edge cases and include testing of the system's invariants?

Project Targets

The engagement involved a review and testing of the Umee blockchain (including its x/oracle and x/leverage modules and price-feeder) and the Peggo orchestrator, which relays messages between Umee and Ethereum blockchains.

UMEE (x/oracle and x/leverage modules and price-feeder)

Repository	https://github.com/umee-network/umee
Version	7eee4cc8bda457cfc8a27fae493e58273ec30b79
Type	Blockchain
Platform	Go

Peggo (Gravity Bridge Orchestrator)

Repository	https://github.com/umee-network/peggo
Version	98be9d710691db55749ab3ebd3f5a739f4e0b965
Type	Blockchain relayer
Platform	Go

We also reviewed pull requests [umee#434](#) and [umee#483](#), which removed the interest epoch parameter in favor of an adjusted borrow rate and interest scalar value.

Project Coverage

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

- **General review.** We performed a manual review of the Umee system's use of Cosmos SDK features. We reviewed the input validation performed by the system and the use of application blockchain interface (ABCI) hooks, among other system elements; we also checked whether events are emitted when appropriate and whether the chain can panic unexpectedly in contexts other than the execution of a transaction. We read the Umee whitepaper and documentation and ran static analysis tools over the codebase, as described in the "Automated Testing" section.
- **Umee's x/oracle and x/leverage modules.** We analyzed the modules' tests and manually reviewed their code, basing our review on the corresponding specification documents in the spec directories. We also performed simple non-exhaustive fuzzing of Umee's transaction-handling functionality, using a Cosmos SDK simulation based on newer code developed by Umee. However, we were unable to perform proper randomized testing (fuzzing) of the modules' invariants.
- **price-feeder.** We conducted a manual code review and dynamic testing. We also reviewed the use of price provider APIs.
- **Peggo.** We conducted a shallow non-holistic review of the Peggo orchestrator code, focusing on identifying significant bugs.
- **Umee's migration away from the interest epoch.** We reviewed the correctness of the new interest-storage method as a mathematical abstraction and checked whether the implementation introduced any rounding errors.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- **Umee's x/oracle and x/leverage modules**
 - Any changes introduced by pull requests [umee#434](#) and [umee#483](#) beyond those described in the specification
 - The impact of using a stablecoin rather than USD as a quote in certain exchange rates (that is, the impact of reading an exchange rate as token/USD instead of token/USDC)

- Whether the price feed APIs use the expected number of decimals for the tokens supported by the system
- **Peggo**
 - The Ethereum-Cosmos communication logic, including the use of transaction signing to verify the integrity and authenticity of data and the use of private keys
 - The handling of Ethereum event logs, the use of go-ethereum APIs (e.g., the handling of **tokens that return false** instead of reverting), the handling of blockchain reorganizations, and the resistance to blockchain-specific attacks like eclipse attacks as well as other attacks that could affect the event logs provided to Peggo
 - Discrepancies between the specification and the actual implementation
 - The holistic security of the system, including the relayers, Gravity Bridge module, and Ethereum smart contracts
 - Modifications from the original (forked) implementation
 - The economic incentives meant to ensure that malicious behavior is not profitable and the research on new maximum extractable value (MEV) opportunities
 - The token/uToken exchange rate's susceptibility to manipulation through misuse of Peggo (e.g., misuse similar to that described in **TOB-UMEE-21**)
- **Umee's migration away from the interest epoch**
 - The consistency of the new feature's use (e.g., whether parts of the code still use the old method or treat adjusted values as non-adjusted, or vice versa)
 - The impact of the change on uTokenExchangeRate computations
 - Whether the new invariant (TotalAdjustedBorrow is equal to the sum of all AdjustedBorrow values) always holds
 - The effect of an ever-increasing InterestScalar value (i.e., whether a high enough value could introduce vulnerabilities)
- **Umee's x/ibctransfer module and Solidity smart contracts**

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

We used the following tools in the automated testing phase of this project. We recommend adding them to the CI/CD pipelines of the project so that they will be run automatically on each new version of the code.

Tool	Description	Policy
<code>go-vet</code>	The go-vet tool is a popular static analysis utility that searches for Go-specific problems in a codebase, such as issues related to closures, marshaling, and unsafe pointers. The go-vet tool is integrated into the go command itself and provides support for other tools through the vettool command-line flag.	Executed as <code>go vet ./...</code>
<code>go-sec</code>	The go-sec static analysis utility looks for various problems in Go codebases. Notably, go-sec can identify stored credentials, unhandled errors, cryptographically problematic packages, and similar issues.	Executed as <code>gosec ./...</code>
<code>Staticcheck</code>	Staticcheck is a static analysis utility that identifies both stylistic problems and implementation problems in Go codebases.	Executed as <code>staticcheck ./...</code>
<code>nancy</code>	This open-source static analysis tool is used to check for vulnerabilities in the Go dependencies used by a project.	Executed as <code>go list -json -m all nancy sleuth</code>
<code>CodeQL</code>	CodeQL is a code analysis engine developed by GitHub to automate security checks. CodeQL	Appendix F

	allows one to query code in a way similar to how one would query an SQL database. CodeQL is free for research and open source.	
GoLand	The GoLand integrated development environment's (IDE) code inspection feature is used to look for issues in a codebase. While it often detects issues similar to those detected by go-vet, it also has analyzers for problematic regular expressions and issues that can occur in different file formats.	GoLand's code inspection documentation
Semgrep	Semgrep is an open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time.	Appendix F
GCatch	GCatch is a research tool for the automatic detection of concurrency bugs in Go codebases. It often finds issues that would not be detected by any other tool or by the Go race detector, such as blocking misuse-of-channel bugs, deadlocks, races on struct fields, and races caused by errors during the use of a testing package.	<p>We were not able to run the tool on every module; when it did work, it did not provide any results.</p> <p>We filed an issue, GCatch#38, regarding GCatch's installation.</p>

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The use of the Dec type for fixed-point arithmetic may cause rounding errors.	Moderate
Auditing	We did not find any issues related to events or logging. The storage and utilization of logs were outside the scope of the audit.	Satisfactory
Authentication / Access Controls	The x/oracle and x/leverage modules use strict access controls implemented through standard Cosmos SDK methods. However, Peggo, the relayer subcomponent, and the Tendermint, Cosmos, and Ethereum APIs have weak access controls or none whatsoever and therefore lack protection against local attacks.	Moderate
Complexity Management	The project builds on Cosmos SDK, which has a clear structure.	Satisfactory
Configuration	Configurable parameters are subject to basic validation; however, that validation is not strict enough and may allow some parameters to be set to invalid edge-case values. It would be beneficial to check parameters' values after the network is started / restarted or after the system undergoes changes like the addition of a new token.	Further Investigation Required

Cryptography and Key Management	Private keys may be provided to the system in an insecure fashion. Additional development of the key-handling functionality would be beneficial; for example, requiring the use of secure methods for storing keys and passing keys to applications, ensuring that no sensitive data is exposed via the operating system or application logs, and introducing support for key revocation and rotation would provide additional security. It would also be beneficial to further review the private keys used by Peggo (e.g., for ways in which a user could trick a validator into signing arbitrary data).	Further Investigation Required
Data Handling	We found multiple issues related to the insufficient validation of data processed by the system; these primarily concern the handling of edge-case values in the x/leverage module and Peggo.	Moderate
Documentation	Although the x/oracle and x/leverage modules have helpful specifications, certain areas of the project lack operational documentation. For example, the documentation on setting up and running Peggo should be improved; the documentation should also cover the flow of messages in Peggo and between various components (e.g., relayers, the Gravity Bridge, smart contracts, etc.). Lastly, all system invariants and assumptions should be clearly specified.	Weak
Front-Running Resistance	Crucial operations such as price updates could be blocked if they were performed in the last possible block and front-run by many transactions. The ability to prioritize certain transactions would be beneficial to the system, and MEV opportunities introduced by the Umee system (especially relayers) should be researched and documented. The functions should have arguments for their minimum and maximum values, which would reduce the risk of front-running. Moreover, the practice of handling system updates (e.g., interest scalar updates) only after a block is mined and the ability to include many messages in one transaction should be evaluated further; it is possible that users could abuse these aspects of the system (for example, to take	Moderate

	out free short-term loans).	
Memory Safety and Error Handling	<p>We identified a few error-handling issues, though they are mostly related to insufficient data validation. The system's tests should be extended to cover all possible error paths.</p> <p>Memory safety was outside the scope of this review.</p>	Moderate
Testing and Verification	The unit tests are not exhaustive, and many check only happy paths. The unit tests should ensure that expected states are triggered when errors occur. Randomized fuzz testing of the entire project (via the simulation module, for example) could help to battle-test the assumptions and invariants of the system.	Weak

Summary of Findings

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

ID	Title	Type	Severity
1	Integer overflow in Peggo's deploy-erc20-raw command	Configuration	Informational
2	Rounding of the standard deviation value may deprive voters of rewards	Configuration	Low
3	Vulnerabilities in exchange rate commitment scheme	Data Validation	Low
4	Validators can crash other nodes by triggering an integer overflow	Data Validation	High
5	The repayValue variable is not used after being modified	Undefined Behavior	Undetermined
6	Inconsistent error checks in GetSigners methods	Data Validation	Informational
7	Incorrect price assumption in the GetExchangeRateBase function	Configuration	High
8	Oracle price-feeder is vulnerable to manipulation by a single malicious price feed	Data Validation	High
9	Oracle rewards may not be distributed	Configuration	Informational
10	Risk of server-side request forgery attacks	Data Validation	Medium
11	Incorrect comparison in SetCollateralSetting method	Data Validation	High
12	Voters' ability to overwrite their own pre-votes is not documented	Data Validation	Informational

13	Lack of user-controlled limits for input amount in LiquidateBorrow	Data Validation	Medium
14	Lack of simulation and fuzzing of leverage module invariants	Testing	Medium
15	Attempts to overdraw collateral cause WithdrawAsset to panic	Data Validation	Low
16	Division by zero causes the LiquidateBorrow function to panic	Data Validation	Low
17	Architecture-dependent code	Data Validation	Informational
18	Weak cross-origin resource sharing settings	Configuration	Informational
19	price-feeder is at risk of rate limiting by public APIs	Configuration	Medium
20	Lack of prioritization of oracle messages	Timing	Medium
21	Risk of token/uToken exchange rate manipulation	Undefined Behavior	High
22	Collateral dust prevents the designation of defaulted loans as bad debt	Data Validation	Low
23	Users can borrow assets that they are actively using as collateral	Configuration	Undetermined
24	Providing additional collateral may be detrimental to borrowers in default	Configuration	Informational
25	Insecure storage of price-feeder keyring passwords	Data Exposure	Medium
26	Insufficient validation of genesis parameters	Data Validation	Low

27	Potential overflows in Peggo's current block calculations	Data Validation	Informational
28	Peggo does not validate Ethereum address formats	Data Validation	Undetermined
29	Peggo takes an Ethereum private key as a command-line argument	Data Exposure	Medium
30	Peggo allows the use of non-local unencrypted URL schemes	Cryptography	Medium
31	Lack of prioritization of Peggo orchestrator messages	Timing	Undetermined
32	Failure of a single broadcast Ethereum transaction causes a batch-wide failure	Configuration	Undetermined
33	Peggo orchestrator's IsBatchProfitable function uses only one price oracle	Data Validation	Medium
34	Rounding errors may cause the module to incur losses	Data Validation	High
35	Outdated and vulnerable dependencies	Patching	Undetermined

Detailed Findings

1. Integer overflow in Peggo's deploy-erc20-raw command

Severity: Informational

Difficulty: High

Type: Configuration

Finding ID: TOB-UMEE-1

Target: peggo/cmd/peggo/bridge.go#L348-L353

Description

The denom-decimals argument of the deploy-erc20-raw command (in the deployERC20RawCmd function) may experience an integer overflow. The argument is first parsed into a value of the int type by the strconv.Atoi function and then cast to a value of the uint8 type (figure 1.1). If the denom-decimals argument with which deploy-erc20-raw is invoked is a negative value or a value that is too large, the casting operation will cause an overflow; however, the user will not receive an error, and the execution will proceed with the overflow value.

```
func deployERC20RawCmd() *cobra.Command {
    return &cobra.Command{
        Use:   "deploy-erc20-raw [gravity-addr] [denom-base] [denom-name] [denom-symbol] [denom-decimals]",
        /* (...) */
        RunE: func(cmd *cobra.Command, args []string) error {
            denomDecimals, err := strconv.Atoi(args[4])
            if err != nil {
                return fmt.Errorf("invalid denom decimals: %w", err)
            }

            tx, err := gravityContract.DeployERC20(auth, denomBase, denomName, denomSymbol,
                uint8(denomDecimals))
```

Figure 1.1: peggo/cmd/peggo/bridge.go#L348-L353

We identified this issue by running CodeQL's `IncorrectIntegerConversionQuery.q1` query.

Recommendations

Short term, fix the integer overflow in Peggo's deployERC20RawCmd function by using the `strconv.ParseUint` function to parse the denom-decimals argument. To do this, use the patch in figure 1.2.

```
diff --git a/cmd/peggo/bridge.go b/cmd/peggo/bridge.go
```

```

index 49aabc5..4b3bc6a 100644
--- a/cmd/peggo/bridge.go
+++ b/cmd/peggo/bridge.go
@@ -345,7 +345,7 @@ network starting`,
        denomBase := args[1]
        denomName := args[2]
        denomSymbol := args[3]
-       denomDecimals, err := strconv.Atoi(args[4])
+       denomDecimals, err := strconv.ParseUint(args[4], 10, 8)
        if err != nil {
                return fmt.Errorf("invalid denom decimals: %w", err)
        }

```

Figure 1.2: A patch for the integer overflow issue in Peggo's deploy-erc20-raw command

Long term, integrate CodeQL into the CI/CD pipeline to find similar issues in the future.

2. Rounding of the standard deviation value may deprive voters of rewards

Severity: Low

Difficulty: High

Type: Configuration

Finding ID: TOB-UMEE-2

Target: `umee/x/oracle/types/ballot.go#L89-L97`

Description

The `ExchangeRateBallot.StandardDeviation` function calculates the standard deviation of the exchange rates submitted by voters. To do this, it converts the variance into a float, prints its square root to a string, and parses it into a `Dec` value (figure 2.1). This logic rounds down the standard deviation value, which is likely unexpected behavior; if the exchange rate is within the reward spread value, voters may not receive the rewards they are owed.

The rounding operation is performed by the `fmt.Sprintf("%f", floatNum)` function, which, as shown in [Appendix C](#), may cut off decimal places from the square root value.

```
// StandardDeviation returns the standard deviation by the power of the ExchangeRateVote.
func (pb ExchangeRateBallot) StandardDeviation() (sdk.Dec, error) {
    // (...)
    variance := sum.QuoInt64(int64(len(pb)))

    floatNum, err := strconv.ParseFloat(variance.String(), 64)
    if err != nil { /* (...) */ }

    floatNum = math.Sqrt(floatNum)
    standardDeviation, err := sdk.NewDecFromStr(fmt.Sprintf("%f", floatNum))
    if err != nil { /* (...) */ }

    return standardDeviation, nil
}
```

Figure 2.1: Inaccurate float conversions (`umee/x/oracle/types/ballot.go#L89-L97`)

Exploit Scenario

A voter reports a price that should be within the reward spread. However, because the standard deviation value is rounded, the price is not within the reward spread, and the voter does not receive a reward.

Recommendations

Short term, have the `ExchangeRateBallot.StandardDeviation` function use the `Dec.ApproxSqrt` method to calculate the standard deviation instead of parsing the variance into a float, calculating the square root, and parsing the formatted float back into a value of the `Dec` type. That way, users who vote for exchange rates close to the correct

reward spread will receive the rewards they are owed. Figure 2.2 shows a patch for this issue.

```
diff --git a/x/oracle/types/ballot.go b/x/oracle/types/ballot.go
index 6b201c2..9f6b579 100644
--- a/x/oracle/types/ballot.go
+++ b/x/oracle/types/ballot.go
@@ -1,12 +1,8 @@
package types

import (
-   "fmt"
-   "math"
-   "sort"
-   "strconv"
+   "sort"
+   sdk "github.com/cosmos/cosmos-sdk/types"
)

// VoteForTally is a convenience wrapper to reduce redundant lookup cost.
@@ -88,13 +84,8 @@ func (pb ExchangeRateBallot) StandardDeviation() (sdk.Dec, error) {

    variance := sum.QuoInt64(int64(len(pb)))

-   floatNum, err := strconv.ParseFloat(variance.String(), 64)
-   if err != nil {
-       return sdk.ZeroDec(), err
-   }
+   standardDeviation, err := variance.ApproxSqrt()

-   floatNum = math.Sqrt(floatNum)
-   standardDeviation, err := sdk.NewDecFromStr(fmt.Sprintf("%f", floatNum))
-   if err != nil {
-       return sdk.ZeroDec(), err
-   }
diff --git a/x/oracle/types/ballot_test.go b/x/oracle/types/ballot_test.go
index 0cd09d8..0dd1f1a 100644
--- a/x/oracle/types/ballot_test.go
+++ b/x/oracle/types/ballot_test.go
@@ -177,21 +177,21 @@ func TestPBStandardDeviation(t *testing.T) {
    []float64{1.0, 2.0, 10.0, 100000.0},
    []int64{1, 1, 100, 1},
    []bool{true, true, true, true},
-   sdk.NewDecWithPrec(4999500036300, OracleDecPrecision),
+   sdk.MustNewDecFromStr("49995.000362536252310906"),
},
{
    // Adding fake validator doesn't change outcome
    []float64{1.0, 2.0, 10.0, 100000.0, 10000000000.0},
    []int64{1, 1, 100, 1, 10000},
    []bool{true, true, true, true, false},
-   sdk.NewDecWithPrec(447213595075100600, OracleDecPrecision),
+   sdk.MustNewDecFromStr("4472135950.751005519905537611"),
},
{
    // Tie votes
    []float64{1.0, 2.0, 3.0, 4.0},
    []int64{1, 100, 100, 1},
```

```
    []bool{true, true, true, true},  
-   sdk.NewDecWithPrec(122474500, OracleDecPrecision),  
+   sdk.MustNewDecFromStr("1.224744871391589049"),  
    },  
    {  
        // No votes
```

Figure 2.2: A patch for this issue

3. Vulnerabilities in exchange rate commitment scheme

Severity: Low

Difficulty: Medium

Type: Data Validation

Finding ID: TOB-UMEE-3

Target: `umee/x/oracle`

Description

The Umee oracle implements a commitment scheme in which users vote on new exchange rates by submitting "pre-vote" and "vote" messages. However, vulnerabilities in this scheme could allow an attacker to (1) predict the prices to which other voters have committed and (2) send two prices for an asset in a pre-vote message hash and then submit one of the prices in the vote message. (Note that predicting other prices would likely require the attacker to make some correct guesses about those prices.)

The first issue is that the random salt used in the scheme is too short. The salt is generated as two random bytes (figure 3.1) and is later hex-encoded and limited to four bytes (figure 3.2). As a result, an attacker could pre-compute the pre-vote commitment hash of every salt value (and thus the expected exchange rate), effectively violating the "hiding" property of the scheme.

```
salt, err := GenerateSalt(2)
```

Figure 3.1: The salt-generation code ([umee/price-feeder/oracle/oracle.go#358](#))

```
if len(msg.Salt) > 4 || len(msg.Salt) < 1 {  
    return sdkerrors.Wrap(ErrInvalidSaltLength, "salt length must be [1, 4]")  
}
```

Figure 3.2: The salt-validation logic ([umee/x/oracle/types/msgs.go#148-150](#))

The second issue is the lack of proper salt validation, which would guarantee sufficient domain separation between a random salt and the exchange rate when the commitment hash is calculated. The domain separator string consists of a colon character, as shown in figure 3.3. However, there is no verification of whether the salt is a hex-encoded string or whether it contains the separator character; only the length of the salt is validated. This bug could allow an attacker to reveal an exchange rate other than the one the attacker had committed to, violating the "binding" property of the scheme.

```
func GetAggregateVoteHash(salt string, exchangeRatesStr string, voter  
sdk.ValAddress) AggregateVoteHash {  
    hash := tmhash.NewTruncated()
```

```
sourceStr := fmt.Sprintf("%s:%s:%s", salt, exchangeRatesStr, voter.String())
```

Figure 3.3: The generation of a commitment hash ([umee/x/oracle/types/hash.go#23–25](#))

The last vulnerability in the scheme is the insufficient validation of exchange rate strings: the strings undergo unnecessary trimming, and the code checks only that `len(denomAmountStr)` is less than two (figure 3.4), rather than performing a stricter check to confirm that it is not equal to two. This could allow an attacker to exploit the second bug described in this finding.

```
func ParseExchangeRateTuples(tuplesStr string) (ExchangeRateTuples, error) {
    tuplesStr = strings.TrimSpace(tuplesStr)
    if len(tuplesStr) == 0 {
        return nil, nil
    }

    tupleStrs := strings.Split(tuplesStr, ",")
    // (...)
    for i, tupleStr := range tupleStrs {
        denomAmountStr := strings.Split(tupleStr, ":")
        if len(denomAmountStr) < 2 {
            return nil, fmt.Errorf("invalid exchange rate %s", tupleStr)
        }
    }
    // (...)
}
```

Figure 3.4: The code that parses exchange rates ([umee/x/oracle/types/vote.go#72–86](#))

Exploit Scenario

The maximum salt length of two is increased. During a subsequent pre-voting period, a malicious validator submits the following commitment hash: `sha256("whatever:UMEE:123:spUMEE:456,USDC:789:addr")`. (Note that "sp" represents a normal whitespace character.) Then, during the voting period, the attacker waits for all other validators to reveal their exchange rates and salts and then chooses the UMEE price that he will reveal (123 or 456). In this way, the attacker can manipulate the exchange rate to his advantage.

If the attacker chooses to reveal a price of 123, the following will occur:

1. The salt will be set to `whatever`.
2. The attacker will submit an exchange rate string of `UMEE:123:spUMEE:456,USDC:789`.
3. The value will be hashed as `sha256(whatever:UMEE:123:spUMEE:456,USDC:789:addr)`.

4. The exchange rate will then be parsed as 123/789 (UMEE/USDC).

Note that `UMEE = 456` (with its leading whitespace character) will be ignored. This is because of the insufficient validation of exchange rate strings (as described above) and the fact that only the first and second items of `denomAmountStr` are used. (See the screenshot in [Appendix D](#)).

If the attacker chooses to reveal a price of 456, the following will occur:

1. The salt will be set to `whatever:UMEE:123`.
2. The exchange rate string will be set to `UMEE:456,USDC:789`.
3. The value will be hashed as `sha256(whatever:UMEE:123: UMEE:456,USDC:789:addr)`.
4. Because exchange rate strings undergo space trimming, the exchange rate will be parsed as 456/789 (UMEE/USDC).

Recommendations

Short term, take the following steps:

- Increase the salt length to prevent brute-force attacks. To ensure a security level of X bits, use salts of 2*X random bits. For example, for a 128-bit security level, use salts of 256 bits (32 bytes).
- Ensure domain separation by implementing validation of a salt's format and accepting only hex-encoded strings.
- Implement stricter validation of exchange rates by ensuring that every exchange rate substring contains exactly one colon character and checking whether all denominations are included in the list of accepted denominations; also avoid trimming whitespaces at the beginning of the parsing process.

Long term, consider replacing the truncated SHA-256 hash function with a SHA-512/256 or HMAC-SHA256 function. This will increase the level of security from 80 bits to about 128, which will help prevent collision and length extension attacks.

4. Validators can crash other nodes by triggering an integer overflow

Severity: High

Difficulty: Medium

Type: Data Validation

Finding ID: TOB-UMEE-4

Target: `umee/x/oracle/types/ballot.go`

Description

By submitting a large exchange rate value, a validator can trigger an integer overflow that will cause a Go panic and a node crash.

The Umee oracle code checks that each exchange rate submitted by a validator is a positive value with a bit size of less than or equal to 256 (figures 4.1 and 4.2). The `StandardDeviation` method iterates over all exchange rates and adds up their squares (figure 4.3) but does not check for an overflow. A large exchange rate value will cause the `StandardDeviation` method to panic when performing **multiplication** or **addition**.

```
func ParseExchangeRateTuples(tuplesStr string) (ExchangeRateTuples, error) {
    // (...)
    for i, tupleStr := range tupleStrs {
        // (...)
        decCoin, err := sdk.NewDecFromStr(denomAmountStr[1])
        // (...)
        if !decCoin.IsPositive() {
            return nil, types.ErrInvalidOraclePrice
        }
    }
}
```

Figure 4.1: The check of whether the exchange rate values are positive
(`umee/x/oracle/types/vote.go#L71-L96`)

```
func (msg MsgAggregateExchangeRateVote) ValidateBasic() error {
    // (...)
    exchangeRates, err := ParseExchangeRateTuples(msg.ExchangeRates)
    if err != nil { /* (...) - returns wrapped error */ }

    for _, exchangeRate := range exchangeRates {
        // check overflow bit length
        if exchangeRate.ExchangeRate.BigInt().BitLen() > 255+sdk.DecimalPrecisionBits
        // (...) - returns error
    }
}
```

Figure 4.2: The check of the exchange rate values' bit lengths
(`umee/x/oracle/types/messages.go#L136-L146`)

```
sum := sdk.ZeroDec()
for _, v := range pb {
    deviation := v.ExchangeRate.Sub(median)
}
```

```
    sum = sum.Add(deviation.Mul(deviation))  
}
```

*Figure 4.3: Part of the StandardDeviation method
([umee/x/oracle/types/ballot.go#83-87](https://github.com/umee/x-oracle/types/blob/master/ballot.go#L83-87))*

The StandardDeviation method is called by the Tally function, which is called in the EndBlocker function. This means that an attacker could trigger an overflow remotely in another validator node.

Exploit Scenario

A malicious validator commits to and then sends a large UMEE exchange rate value. As a result, all validator nodes crash, and the Umee blockchain network stops working.

Recommendations

Short term, implement overflow checks for all arithmetic operations involving exchange rates.

Long term, use fuzzing to ensure that no other parts of the code are vulnerable to overflows.

5. The repayValue variable is not used after being modified

Severity: **Undetermined**

Difficulty: **High**

Type: Undefined Behavior

Finding ID: TOB-UMEE-5

Target: umee/x/leverage/keeper/keeper.go

Description

The `Keeper.LiquidateBorrow` function uses the local variable `repayValue` to calculate the `repayment.Amount` value. If `repayValue` is greater than or equal to `maxRepayValue`, it is changed to that value. However, the `repayValue` variable is not used again after being modified, which suggests that the modification could be a bug or a code quality issue.

```
func (k Keeper) LiquidateBorrow(
    // (...)
    // repayment cannot exceed borrowed value * close factor
    maxRepayValue := borrowValue.Mul(closeFactor)
    repayValue, err := k.TokenValue(ctx, repayment)
    if err != nil {
        return sdk.ZeroInt(), sdk.ZeroInt(), err
    }

    if repayValue.GTE(maxRepayValue) {
        // repayment *= (maxRepayValue / repayValue)
        repayment.Amount =
        repayment.Amount.ToDec().Mul(maxRepayValue).Quo(repayValue).TruncateInt()
        repayValue = maxRepayValue
    }
    // (...)
)
```

Figure 5.1: `umee/x/leverage/keeper/keeper.go#L446-L456`

We identified this issue by running CodeQL's `DeadStoreOfLocal.q1` query.

Recommendations

Short term, review and fix the `repayValue` variable in the `Keeper.LiquidateBorrow` function, which is not used after being modified, to prevent related issues in the future.

6. Inconsistent error checks in GetSigners methods

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-UMEE-6

Target: `umee/x/leverage` and `umee/x/oracle`

Description

The `GetSigners` methods in the `x/oracle` and `x/leverage` modules exhibit different error-handling behavior when parsing strings into validator or account addresses. The `GetSigners` methods in the `x/oracle` module always panic upon an error, while the methods in the `x/leverage` module explicitly ignore parsing errors. Figures 6.1 and 6.2 show examples of the `GetSigners` methods in those modules.

We set the severity of this finding to informational because message addresses parsed in the `x/leverage` module's `GetSigners` methods are also validated in the `ValidateBasic` methods. As a result, the issue is not currently exploitable.

```
// GetSigners implements sdk.Msg
func (msg MsgDelegateFeedConsent) GetSigners() []sdk.AccAddress {
    operator, err := sdk.ValAddressFromBech32(msg.Operator)
    if err != nil {
        panic(err)
    }

    return []sdk.AccAddress{sdk.AccAddress(operator)}
}
```

Figure 6.1: `umee/x/oracle/types/messages.go#L174-L182`

```
func (msg *MsgLendAsset) GetSigners() []sdk.AccAddress {
    lender, _ := sdk.AccAddressFromBech32(msg.GetLender())
    return []sdk.AccAddress{lender}
}
```

Figure 6.2: `umee/x/leverage/types/tx.go#L30-L33`

Recommendations

Short term, use a consistent error-handling process in the `x/oracle` and `x/leverage` modules' `GetSigners` methods. The `x/leverage` module's `GetSigners` functions should handle errors in the same way that the `x/oracle` methods do—by panicking.

7. Incorrect price assumption in the GetExchangeRateBase function

Severity: High

Difficulty: Medium

Type: Configuration

Finding ID: TOB-UMEE-7

Target: `umee/x/oracle/keeper/keeper.go`

Description

If the denominator string passed to the `GetExchangeRateBase` function contains the substring "USD" (figure 7.1), the function returns 1, presumably to indicate that the denominator is a stablecoin. If the system accepts an ERC20 token that is not a stablecoin but has a name containing "USD," the system will report an incorrect exchange rate for the asset, which may enable token theft. Moreover, the price of an actual USD stablecoin may vary from USD 1. Therefore, if a stablecoin used as collateral for a loan loses its peg, the loan may not be liquidated correctly.

```
// GetExchangeRateBase gets the consensus exchange rate of an asset
// in the base denom (e.g. ATOM -> uatom)
func (k Keeper) GetExchangeRateBase(ctx sdk.Context, denom string) (sdk.Dec, error)
{
    if strings.Contains(strings.ToUpper(denom), types.USDDenom) {
        return sdk.OneDec(), nil
    }
    // (...)
```

Figure 7.1: `umee/x/oracle/keeper/keeper.go#L89-L94`

```
func (k Keeper) TokenPrice(ctx sdk.Context, denom string) (sdk.Dec, error) {
    if !k.IsAcceptedToken(ctx, denom) {
        return sdk.ZeroDec(), sdkerrors.Wrap(types.ErrInvalidAsset, denom)
    }
    price, err := k.oracleKeeper.GetExchangeRateBase(ctx, denom)
    // (...)
    return price, nil
}
```

Figure 7.2: `umee/x/leverage/keeper/oracle.go#L12-L34`

Exploit Scenario

Umee adds the `cUSDC ERC20 token` as an accepted token. Upon its addition, its price is USD 0.02, not USD 1. However, because of the incorrect price assumption, the system sets

its price to USD 1. This enables an attacker to create an undercollateralized loan and to draw funds from the system.

Exploit Scenario 2

The price of a stablecoin drops significantly. However, the `x/leverage` module fails to detect the change and reports the price as USD 1. This enables an attacker to create an undercollateralized loan and to draw funds from the system.

Recommendations

Short term, remove the condition that causes the `GetExchangeRateBase` function to return a price of USD 1 for any asset whose name contains "USD."

8. Oracle price-feeder is vulnerable to manipulation by a single malicious price feed

Severity: High

Difficulty: High

Type: Data Validation

Finding ID: TOB-UMEE-8

Target: price-feeder and umee/x/oracle

Description

The price-feeder component uses a volume-weighted average price (VWAP) formula to compute average prices from various third-party providers. The price it determines is then sent to the x/oracle module, which commits it on-chain. However, an asset price could easily be manipulated by only one compromised or malfunctioning third-party provider.

Exploit Scenario

Most validators are using the Binance API as one of their price providers. The API is compromised by an attacker and suddenly starts to report prices that are much higher than those reported by other providers. However, the price-feeder instances being used by the validators do not detect the discrepancies in the Binance API prices. As a result, the VWAP value computed by the price-feeder and committed on-chain is much higher than it should be. Moreover, because most validators have committed the wrong price, the average computed on-chain is also wrong. The attacker then draws funds from the system.

Recommendations

Short term, implement a price-feeder mechanism for detecting the submission of wildly incorrect prices by a third-party provider. Have the system temporarily disable the use of the malfunctioning provider(s) and issue an alert calling for an investigation. If it is not possible to automatically identify the malfunctioning provider(s), stop committing prices. (Note, though, that this may result in a loss of interest for validators.) Consider implementing a similar mechanism in the x/oracle module so that it can identify when the exchange rates committed by validators are too similar to one another or to old values.

References

- [Synthetix Response to Oracle Incident](#)

9. Oracle rewards may not be distributed

Severity: Informational

Difficulty: High

Type: Configuration

Finding ID: TOB-UMEE-9

Target: `umee/x/oracle`

Description

If the `x/oracle` module lacks the coins to cover a reward payout, the rewards will not be distributed or registered for payment in the future.

```
var periodRewards sdk.DecCoins
for _, denom := range rewardDenoms {
    rewardPool := k.GetRewardPool(ctx, denom)

    // return if there's no rewards to give out
    if rewardPool.IsZero() {
        continue
    }

    periodRewards = periodRewards.Add(sdk.NewDecCoinFromDec(
        denom,
        sdk.NewDecFromInt(rewardPool.Amount).Mul(distributionRatio),
    ))
}
```

Figure 9.1: A loop in the code that calculates oracle rewards
([umee/x/oracle/keeper/reward.go#43-56](#))

Recommendations

Short term, document the fact that oracle rewards will not be distributed when the `x/oracle` module does not have enough coins to cover the rewards.

10. Risk of server-side request forgery attacks

Severity: **Medium**

Difficulty: **High**

Type: Data Validation

Finding ID: TOB-UMEE-10

Target: price-feeder

Description

The price-feeder sends HTTP requests to configured providers' APIs. If any of the HTTP responses is a redirect response (e.g., one with HTTP response code 301), the module will automatically issue a new request to the address provided in the response's header. The new address may point to a local address, potentially one that provides access to restricted services.

Exploit Scenario

An attacker gains control over the Osmosis API. He changes the endpoint used by the price-feeder such that it responds with a redirect like that shown in figure 10.1, with the goal of **removing a transaction** from a Tendermint validator's mempool. The price-feeder automatically issues a new request to the Tendermint REST API. Because the API does not require authentication and is running on the same machine as the price-feeder, the request is successful, and the target transaction is removed from the validator's mempool.

```
HTTP/1.1 301 Moved Permanently
Location: http://localhost:26657/remove_tx?txKey=aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 10.1: The redirect response

Recommendations

Short term, use a function such as **CheckRedirect** to disable redirects, or at least redirects to local services, in all HTTP clients.

11. Incorrect comparison in SetCollateralSetting method

Severity: High

Difficulty: Low

Type: Data Validation

Finding ID: TOB-UMEE-11

Target: umee/x/leverage

Description

Umee users can send a `SetCollateral` message to disable the use of a certain asset as collateral. The messages are handled by the `SetCollateralSetting` method (figure 11.1), which should ensure that the borrow limit will not drop below the amount borrowed. However, the function uses an incorrect comparison, checking that the borrow limit will be greater than, not less than, that amount.

```
// Return error if borrow limit would drop below borrowed value
if newBorrowLimit.GT(borrowedValue) {
    return sdkerrors.Wrap(types.ErrBorrowLimitLow, newBorrowLimit.String())
}
```

*Figure 11.1: The incorrect comparison in the SetCollateralSetting method
([umee/x/leverage/keeper/keeper.go#343-346](#))*

Exploit Scenario

An attacker provides collateral to the Umee system and borrows some coins. Then the attacker disables the use of the collateral asset; because of the incorrect comparison in the `SetCollateralSetting` method, the disable operation succeeds, and the collateral is sent back to the attacker.

Recommendations

Short term, correct the comparison in the `SetCollateralSetting` method.

Long term, implement tests to check whether basic functionality works as expected.

12. Voters' ability to overwrite their own pre-votes is not documented

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-UMEE-12

Target: `umee/x/oracle`

Description

The `x/oracle` module allows voters to submit more than one pre-vote message during the same pre-voting period, overwriting their previous pre-vote messages (figure 12.1). This feature is not documented; while it does not constitute a direct security risk, it may be unintended behavior.

Third parties may incorrectly assume that validators cannot change their pre-vote messages. Monitoring systems may detect only the first pre-vote event for a validator's pre-vote messages, while voters may trust the exchange rates and salts revealed by other voters to be final.

On the other hand, this feature may be an intentional one meant to allow voters to update the exchange rates they submit as they obtain more accurate pricing information.

```
func (ms msgServer) AggregateExchangeRatePrevote(
    goCtx context.Context,
    msg *types.MsgAggregateExchangeRatePrevote,
) (*types.MsgAggregateExchangeRatePrevoteResponse, error) {
    // (...)
    aggregatePrevote := types.NewAggregateExchangeRatePrevote(voteHash, valAddr,
    uint64(ctx.BlockHeight()))

    // This call overwrites previous pre-vote if there was one
    ms.SetAggregateExchangeRatePrevote(ctx, valAddr, aggregatePrevote)

    ctx.EventManager().EmitEvents(sdk.Events{
        // (...) - emit EventTypeAggregatePrevote and EventTypeMessage
    })
    return &types.MsgAggregateExchangeRatePrevoteResponse{}, nil
}
```

Figure 12.1: `umee/x/oracle/keeper/msg_server.go#L23-L66`

Recommendations

Short term, document the fact that a pre-vote message can be submitted and overwritten in the same voting period. Alternatively, disallow this behavior by having the

AggregateExchangeRatePrevote function return an error if a validator attempts to submit an additional exchange rate pre-vote message.

Long term, add tests to check for this behavior.

13. Lack of user-controlled limits for input amount in LiquidateBorrow

Severity: **Medium**

Difficulty: **High**

Type: Data Validation

Finding ID: TOB-UMEE-13

Target: `umee/x/leverage`

Description

The `x/leverage` module's `LiquidateBorrow` function computes the amount of funds that will be transferred from the module to the function's caller in a liquidation. The computation uses asset prices retrieved from an oracle.

There is no guarantee that the amount returned by the module will correspond to the current market price, as a transaction that updates the price feed could be mined before the call to `LiquidateBorrow`.

Adding a lower limit to the amount sent by the module would enable the caller to explicitly state his or her assumptions about the liquidation and to ensure that the collateral payout is as profitable as expected. It would also provide additional protection against the misreporting of oracle prices. Since such a scenario is unlikely, we set the difficulty level of this finding to high.

Using caller-controlled limits for the amount of a transfer is a best practice commonly employed by large DeFi protocols such as Uniswap.

Exploit Scenario

Alice calls the `LiquidateBorrow` function. Due to an oracle malfunction, the amount of collateral transferred from the module is much lower than the amount she would receive on another market.

Recommendations

Short term, introduce a `minRewardAmount` parameter and add a check verifying that the reward value is greater than or equal to the `minRewardAmount` value.

Long term, always allow the caller to control the amount of a transfer. This is especially important for transfer amounts that depend on factors that can change between transactions. Enable the caller to add a lower limit for a transfer from a module and an upper limit for a transfer of the caller's funds to a module.

14. Lack of simulation and fuzzing of leverage module invariants

Severity: Medium

Difficulty: High

Type: Testing

Finding ID: TOB-UMEE-14

Target: Umee test suite

Description

The Umee system lacks comprehensive **Cosmos SDK simulations** and invariants for its `x/oracle` and `x/leverage` modules. More thorough use of the simulation feature would facilitate fuzz testing of the entire blockchain and help ensure that the invariants hold.

Additionally, the current simulation module may need to be modified for the following reasons:

- It **exits on the first transaction error**. To avoid an early exit, it could skip transactions that are expected to fail when they are generated; however, that could also cause it to skip logic that contains issues.
- The **numKeys argument**, which determines how many accounts it will use, can range from 2 to 2,500. Using too many accounts may hinder the detection of bugs that require multiple transactions to be executed by a few accounts.
- By default, it is configured to use a **"stake" currency**, which may not be used in the final Umee system.
- Running it with a small number of accounts and a large block size for many blocks could quickly cause all validators to be unbonded. To avoid this issue, the simulation would need the ability to run for a longer time.

Trail of Bits attempted to use the simulation module by modifying the recent changes to the Umee codebase, which introduce simulations for the `x/oracle` and `x/leverage` modules (commit [f22b2c7f8e](#)). We enabled the `x/leverage` module simulation and modified the Cosmos SDK codebase locally so that the framework would use fewer accounts and log errors via **Fatal logs** instead of exiting. The framework helped us find the issue described in [TOB-UMEE-15](#), but the setup and tests we implemented were not exhaustive. We sent the codebase changes we made to the Umee team via an internal chat.

Recommendations

Short term, identify, document, and test all invariants that are important for the system's security, and identify and document the arbitrage opportunities created by the system.

Enable simulation of the x/oracle and x/leverage modules and ensure that the following assertions and invariants are checked during simulation runs:

1. In the `UpdateExchangeRates` function, the token supply value corresponds to the uToken supply value. Implement the following check:

```
if uTokenSupply != 0 { assert(tokenSupply != 0) }
```

2. In the `LiquidateBorrow` function (after the line `"if !repayment.Amount.IsPositive()"`), the following comparisons evaluate to true:

```
ExchangeUToken(reward) == EquivalentTokenValue(repayment, baseRewardDenom)
TokenValue(ExchangeUToken(ctx, reward)) == TokenValue(repayment)
borrowed.AmountOf(repayment.Denom) >= repayment.Amount
collateral.AmountOf(rewardDenom) >= reward.Amount
module's collateral amount >= reward.Amount
repayment <= desiredRepayment
```

3. The x/leverage module is never significantly undercollateralized at the end of a transaction. Implement a check,

```
total collateral value * X >= total borrows value,
```

in which X is close to 1. (It may make sense for the value of X to be greater than or equal to 1 to account for module reserves.) It may be acceptable for the module to be slightly undercollateralized, as it may mean that some liquidations have yet to be executed.

4. The amount of reserves remains above a certain minimum value, or new loans cannot be issued if the amount of reserves drops below a certain value.
5. The interest on a loan is less than or equal to the borrowing fee. (This invariant is related to the issue described in [TOB-UMEE-23](#).)
6. It is impossible to borrow funds without paying a fee. Currently, when four messages (lend, borrow, repay, and withdraw messages) are sent in one transaction, the `EndBlocker` method will not collect borrowing fees.
7. Token/uToken exchange rates are always greater than or equal to 1 and are less than an expected maximum. To avoid rapid significant price increases and decreases, ensure that the rates do not change more quickly than expected.
8. The `exchangeRate` value cannot be changed by public (user-callable) methods like `LendAsset` and `WithdrawAsset`. Pay special attention to rounding errors and make sure that the module is the beneficiary of all rounding operations.

9. It is impossible to liquidate more than the `closeFactor` in a single liquidation transaction for a defaulted loan; be mindful of the fact that a single transaction can include more than one message.

Long term, extend the simulation module to cover all operations that may occur in a real Umee deployment, along with all potential error states, and run it many times before each release. Ensure the following:

- All modules and operations are included in the simulation module.
- The simulation uses a small number of accounts (e.g., between 5 and 20) to increase the likelihood of an interesting state change.
- The simulation uses the currencies/tokens that will be used in the production network.
- Oracle price changes are properly simulated. (In addition to a mode in which prices are changed randomly, implement a mode in which prices are changed only slightly, a mode in which prices are highly volatile, and a mode in which prices decrease or increase continuously for a long time period.)
- The simulation continues running when a transaction triggers an error.
- All transaction code paths are executed. (Enable `code coverage` to see how often individual lines are executed.)

15. Attempts to overdraw collateral cause WithdrawAsset to panic

Severity: Low

Difficulty: Low

Type: Data Validation

Finding ID: TOB-UMEE-15

Target: umee/x/leverage/keeper/keeper.go

Description

The `WithdrawAsset` function panics when an account attempts to withdraw more collateral than the account holds. While panics triggered during transaction runs **are recovered by the Cosmos SDK**, they should be used only to handle unexpected events that should not occur in normal blockchain operations. The function should instead check the `collateralToWithdraw` value and return an error if it is too large.

The panic occurs in the `Dec.Sub` method when the calculation it performs results in an overflow (figure 15.1).

```
func (k Keeper) WithdrawAsset(/* (...) */) error {
    // (...)
    if amountFromCollateral.IsPositive() {
        if k.GetCollateralSetting(ctx, lenderAddr, uToken.Denom) {
            // (...)
            // Calculate what borrow limit will be AFTER this withdrawal
            collateral := k.GetBorrowerCollateral(ctx, lenderAddr)
            collateralToWithdraw := sdk.NewCoins(sdk.NewCoin(uToken.Denom,
                amountFromCollateral))
            newBorrowLimit, err := k.CalculateBorrowLimit(ctx,
                collateral.Sub(collateralToWithdraw))
        }
    }
}
```

Figure 15.1: `umee/x/leverage/keeper/keeper.go#L124-L159`

To reproduce this issue, use the test shown in figure 15.2.

Exploit Scenario

A user of the Umee system who has enabled the collateral setting lends 1,000 UMEE tokens. The user later tries to withdraw 1,001 UMEE tokens. Due to the lack of validation of the `collateralToWithdraw` value, the transaction causes a panic. However, the panic is recovered, and the transaction finishes with a **panic error**. Because the system does not provide a proper error message, the user is confused about why the transaction failed.

Recommendations

Short term, when a user attempts to withdraw collateral, have the `WithdrawAsset` function check whether the `collateralToWithdraw` value is less than or equal to the

collateral balance of the user's account and return an error if it is not. This will prevent the function from panicking if the withdrawal amount is too large.

Long term, integrate the test shown in figure 15.2 into the codebase and extend it with additional assertions to verify other program states.

```
func (s *IntegrationTestSuite) TestWithdrawAsset_InsufficientCollateral() {
    app, ctx := s.app, s.ctx
    lenderAddr := sdk.AccAddress([]byte("addr_-----"))
    lenderAcc := app.AccountKeeper.NewAccountWithAddress(ctx, lenderAddr)
    app.AccountKeeper.SetAccount(ctx, lenderAcc)

    // mint and send coins
    s.Require().NoError(app.BankKeeper.MintCoins(ctx, minttypes.ModuleName, initCoins))
    s.Require().NoError(app.BankKeeper.SendCoinsFromModuleToAccount(ctx, minttypes.ModuleName,
lenderAddr, initCoins))

    // mint additional coins for just the leverage module; this way it will have available
reserve
    // to meet conditions in the withdrawal logic
    s.Require().NoError(app.BankKeeper.MintCoins(ctx, types.ModuleName, initCoins))

    // set collateral setting for the account
    uTokenDenom := types.UTokenFromTokenDenom(umeeapp.BondDenom)
    err := s.app.LeverageKeeper.SetCollateralSetting(ctx, lenderAddr, uTokenDenom, true)
    s.Require().NoError(err)

    // lend asset
    err = s.app.LeverageKeeper.LendAsset(ctx, lenderAddr, sdk.NewInt64Coin(umeeapp.BondDenom,
1000000000)) // 1k umee
    s.Require().NoError(err)

    // verify collateral amount and total supply of minted uTokens
    collateral := s.app.LeverageKeeper.GetCollateralAmount(ctx, lenderAddr, uTokenDenom)
    expected := sdk.NewInt64Coin(uTokenDenom, 1000000000) // 1k u/umee
    s.Require().Equal(collateral, expected)
    supply := s.app.LeverageKeeper.TotalUTokenSupply(ctx, uTokenDenom)
    s.Require().Equal(expected, supply)

    // withdraw more collateral than having - this panics currently
    uToken := collateral.Add(sdk.NewInt64Coin(uTokenDenom, 1))
    err = s.app.LeverageKeeper.WithdrawAsset(ctx, lenderAddr, uToken)
    s.Require().EqualError(err, "TODO/FIXME: set proper error string here after fixing panic
error")

    // TODO/FIXME: add asserts to verify all other program state
}
```

Figure 15.2: A test that can be used to reproduce this issue

16. Division by zero causes the LiquidateBorrow function to panic

Severity: Low

Difficulty: Low

Type: Data Validation

Finding ID: TOB-UMEE-16

Target: umee/x/leverage

Description

Two operations in the x/leverage module's LiquidateBorrow method may involve division by zero and lead to a panic.

The first operation is shown in figure 16.1. If both repayValue and maxRepayValue are zero, the GTE (greater-than-or-equal-to) comparison will succeed, and the Quo method will panic. The repayValue variable will be set to zero if liquidatorBalance is set to zero; maxRepayValue will be set to zero if either closeFactor or borrowValue is set to zero.

```
if repayValue.GTE(maxRepayValue) {  
    // repayment *= (maxRepayValue / repayValue)  
    repayment.Amount =  
    repayment.Amount.ToDec().Mul(maxRepayValue).Quo(repayValue).TruncateInt()  
    repayValue = maxRepayValue  
}
```

*Figure 16.1: A potential instance of division by zero
(umee/x/leverage/keeper/keeper.go#452–456)*

The second operation is shown in figure 16.2. If both reward.Amount and collateral.AmountOf(rewardDenom) are set to zero, the GTE comparison will succeed, and the Quo method will panic. The collateral.AmountOf(rewardDenom) variable can easily be set to zero, as the user may not have any collateral in the denomination indicated by the variable; reward.Amount will be set to zero if liquidatorBalance is set to zero.

```
// reward amount cannot exceed available collateral  
if reward.Amount.GTE(collateral.AmountOf(rewardDenom)) {  
    // reduce repayment.Amount to the maximum value permitted by the available  
    collateral reward  
    repayment.Amount =  
    repayment.Amount.Mul(collateral.AmountOf(rewardDenom)).Quo(reward.Amount)  
    // use all collateral of reward denom  
    reward.Amount = collateral.AmountOf(rewardDenom)  
}
```

*Figure 16.2: A potential instance of division by zero
(umee/x/leverage/keeper/keeper.go#474–480)*

Exploit Scenario

A user tries to liquidate a loan. For reasons that are unclear to the user, the transaction fails with a panic. Because the error message is not specific, the user has difficulty debugging the error.

Recommendations

Short term, replace the GTE comparison with a strict inequality GT (greater-than) comparison.

Long term, carefully validate variables in the `LiquidateBorrow` method to ensure that every variable stays within the expected range during the entire computation. Write negative tests with edge-case values to ensure that the methods handle errors gracefully.

17. Architecture-dependent code

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-UMEE-17

Target: `umee/x/oracle`

Description

In the Go programming language, the bit size of an `int` variable depends on the platform on which the code is executed. On a 32-bit platform, it will be 32 bits, and on a 64-bit platform, 64 bits. Validators running on different architectures will therefore interpret `int` types differently, which may lead to transaction-parsing discrepancies and ultimately to a consensus failure or chain split.

One use of the `int` type is shown in figure 17.1. Because casting the `maxValidators` variable to the `int` type should not cause it to exceed the maximum `int` value for a 32-bit platform, we set the severity of this finding to informational.

```
for ; iterator.Valid() && i < int(maxValidators); iterator.Next() {
```

Figure 17.1: An architecture-dependent loop condition in the `EndBlocker` method (`umee/x/oracle/abci.go#34`)

Exploit Scenario

The `maxValidators` variable (a variable of the `uint32` type) is set to its maximum value, 4,294,967,296. During the execution of the `x/oracle` module's `EndBlocker` method, some validators cast the variable to a negative number, while others cast it to a large positive integer. The chain then stops working because the validators cannot reach a consensus.

Recommendations

Short term, ensure that architecture-dependent types are not used in the codebase.

Long term, test the system with parameters set to various edge-case values, including the maximum and minimum values of different integer types. Test the system on all common architectures (e.g., architectures with 32- and 64-bit CPUs), or develop documentation specifying the architecture(s) used in testing.

18. Weak cross-origin resource sharing settings

Severity: Informational

Difficulty: High

Type: Configuration

Finding ID: TOB-UMEE-18

Target: price-feeder

Description

In the price-feeder's cross-origin resource sharing (CORS) settings, most of the same-origin policy protections are disabled. This increases the severity of vulnerabilities like cross-site request forgery.

```
v1Router.Methods("OPTIONS").HandlerFunc(func(w http.ResponseWriter, r *http.Request)
{
    w.Header().Set("Access-Control-Allow-Origin", r.Header.Get("Origin"))
    w.Header().Set("Access-Control-Allow-Methods", "GET, PUT, POST, DELETE,
OPTIONS")
    w.Header().Set("Access-Control-Allow-Headers", "Content-Type,
Access-Control-Allow-Headers, Authorization, X-Requested-With")
    w.Header().Set("Access-Control-Allow-Credentials", "true")
    w.WriteHeader(http.StatusOK)
})
```

*Figure 18.1: The current CORS configuration
(umee/price-feeder/router/v1/router.go#46-52)*

We set the severity of this finding to informational because no sensitive endpoints are exposed by the price-feeder router.

Exploit Scenario

A new endpoint is added to the price-feeder API. It accepts PUT requests that can update the tool's provider list. An attacker uses phishing to lure the price-feeder's operator to a malicious website. The website triggers an HTTP PUT request to the API, changing the provider list to a list in which all addresses are controlled by the attacker. The attacker then repeats the attack against most of the validators, manipulates on-chain prices, and drains the system's funds.

Recommendations

Short term, use strong default values in the CORS settings.

Long term, ensure that APIs exposed by the price-feeder have proper protections against web vulnerabilities.

19. price-feeder is at risk of rate limiting by public APIs

Severity: **Medium**

Difficulty: **Medium**

Type: Configuration

Finding ID: TOB-UMEE-19

Target: price-feeder

Description

Price providers used by the price-feeder tool may enforce limits on the number of requests served to them. After reaching a limit, the tool should take certain actions to avoid a prolonged or even permanent ban. Moreover, using API keys or non-HTTP access channels would decrease the price-feeder's chance of being rate limited.

Every API has its own rules, which should be reviewed and respected. The rules of three APIs are summarized below.

- Binance has **hard, machine-learning, and web application firewall limits**. Users are required to stop sending requests if they receive a **429 HTTP response code**.
- Kraken implements **rate limiting based on "call counters"** and recommends using the WebSockets API instead of the REST API.
- Huopi restricts the number of requests to **10 per second** and recommends using an API key.

Exploit Scenario

A price-feeder exceeds the limits of the Binance API. It is rate limited and receives a 429 HTTP response code from the API. The tool does not notice the response code and continues to spam the API. As a result, it receives a permanent ban. The validator using the price-feeder then starts reporting imprecise exchange rates and gets slashed.

Recommendations

Short term, review the requirements and recommendations of all APIs supported by the system. Enforce their requirements in a user-friendly manner; for example, allow users to set and rotate API keys, delay HTTP requests so that the price-feeder will avoid rate limiting but still report accurate prices, and log informative error messages upon reaching rate limits.

Long term, perform stress-testing to ensure that the implemented safety checks work properly and are robust.

20. Lack of prioritization of oracle messages

Severity: **Medium**

Difficulty: **Medium**

Type: Timing

Finding ID: TOB-UMEE-20

Target: `umee/x/oracle`

Description

Oracle messages are not prioritized over other transactions for inclusion in a block. If the network is highly congested, the messages may not be included in a block. Although the Umee system could increase the fee charged for including an oracle message in a block, that solution is suboptimal and may not work.

Tactics for prioritizing important transactions include the following:

- Using the custom **CheckTx** implementation introduced in **Tendermint version 0.35**, which returns a **priority** argument
- **Reimplementing part of the Tendermint engine**, as Terra Money did
- **Using Substrate's dispatch classes**, which allow developers to mark transactions as **normal**, **operational**, or **mandatory**

Exploit Scenario

The Umee network is congested. Validators send their exchange rate votes, but the exchange rates are not included in a block. An attacker then exploits the situation by draining the network of its tokens.

Recommendations

Short term, use a custom CheckTx method to prioritize oracle messages. This will help prevent validators' votes from being left out of a block and ignored by an oracle.

Long term, ensure that operations that affect the whole system cannot be front-run or delayed by attackers or blocked by network congestion.

21. Risk of token/uToken exchange rate manipulation

Severity: High

Difficulty: Medium

Type: Undefined Behavior

Finding ID: TOB-UMEE-21

Target: umee/x/leverage

Description

The Umee specification states that the **token/uToken exchange rate** can be affected only by the accrual of interest (not by Lend, Withdraw, Borrow, Repay, or Liquidate transactions). However, this invariant can be broken:

- When tokens are burned or minted through an Inter-Blockchain Communication (IBC) transfer, the `ibc-go` library accesses the `x/bank` module's keeper interface, which changes the total token supply (as shown in figure 21.2). This behavior is mentioned in a comment shown in figure 22.1.
- Sending tokens directly to the module through an **x/bank message** also affects the exchange rate.

```
func (k Keeper) TotalUTokenSupply(ctx sdk.Context, uTokenDenom string) sdk.Coin {
    if k.IsAcceptedUToken(ctx, uTokenDenom) {
        return k.bankKeeper.GetSupply(ctx, uTokenDenom)
        // TODO - Question: Does bank module still track balances sent (locked) via IBC?
        // If it doesn't then the balance returned here would decrease when the tokens
        // are sent off, which is not what we want. In that case, the keeper should keep
        // an sdk.Int total supply for each uToken type.
    }
    return sdk.NewCoin(uTokenDenom, sdk.ZeroInt())
}
```

Figure 21.1: The method vulnerable to unexpected IBC transfers
([umee/x/leverage/keeper/keeper.go#65-73](#))

```
if err := k.bankKeeper.BurnCoins(
    ctx, types.ModuleName, sdk.NewCoins(token),
```

Figure 21.2: The IBC library code that accesses the `x/bank` module's keeper interface
([ibc-go/modules/apps/transfer/keeper/relay.go#136-137](#))

Exploit Scenario

An attacker with two Umee accounts lends tokens through the system and receives a commensurate number of uTokens. He temporarily sends the uTokens from one of the

accounts to another chain ("chain B"), decreasing the total supply and increasing the token/uToken exchange rate. The attacker uses the second account to withdraw more tokens than he otherwise could and then sends uTokens back from chain B to the first account. In this way, he drains funds from the module.

Recommendations

Short term, ensure that the `TotalUTokenSupply` method accounts for IBC transfers. Use the [Cosmos SDK's blocklisting feature](#) to disable direct transfers to the leverage and oracle modules. Consider setting `DefaultSendEnabled` to `false` and explicitly enabling certain tokens' transfer capabilities.

Long term, follow GitHub issues [#10386](#) and [#5931](#), which concern functionalities that may enable module developers to make token accounting more reliable. Additionally, ensure that the system accounts for [inflation](#).

22. Collateral dust prevents the designation of defaulted loans as bad debt

Severity: Low

Difficulty: Medium

Type: Data Validation

Finding ID: TOB-UMEE-22

Target: umee/x/leverage

Description

An account's debt is considered bad debt only if its collateral balance drops to zero. The debt is then repaid from the module's reserves. However, users may liquidate the majority of an account's assets but leave a small amount of debt unpaid. In that case, the transaction fees may make liquidation of the remaining collateral unprofitable. As a result, the bad debt will not be paid from the module's reserves and will linger in the system indefinitely.

Exploit Scenario

A large loan taken out by a user becomes highly undercollateralized. An attacker liquidates most of the user's collateral to repay the loan but leaves a very small amount of the collateral unliquidated. As a result, the loan is not considered bad debt and is not paid from the reserves. The rest of the tokens borrowed by the user remain out of circulation, preventing other users from withdrawing their funds.

Recommendations

Short term, establish a lower limit on the amount of collateral that must be liquidated in one transaction to prevent accounts from holding dust collateral.

Long term, establish a lower limit on the number of tokens to be used in every system operation. That way, even if the system's economic incentives are lacking, the operations will not result in dust tokens.

23. Users can borrow assets that they are actively using as collateral

Severity: **Undetermined**

Difficulty: **Low**

Type: Configuration

Finding ID: TOB-UMEE-23

Target: umee/x/leverage

Description

When a user calls the `BorrowAsset` function to take out a loan, the function does not check whether the user is borrowing the same type of asset as the collateral he or she supplied. In other words, a user can borrow tokens from the collateral that the user supplied.

The Umee system prohibits users from borrowing assets worth more than the collateral they have provided, so a user cannot directly exploit this issue to borrow more funds than the user should be able to borrow. However, a user can borrow the vast majority of his or her collateral to continue accumulating lending rewards while largely avoiding the risks of providing collateral.

Exploit Scenario

An attacker provides 10 ATOMs to the protocol as collateral and then immediately borrows 9 ATOMs. He continues to earn lending rewards on his collateral but retains the use of most of the collateral. The attacker, through flash loans, could also resupply the borrowed amount as collateral and then immediately take out another loan, repeating the process until the amount he had borrowed asymptotically approached the amount of liquidity he had provided.

Recommendations

Short term, determine whether borrowers' ability to borrow their own collateral is an issue. (Note that Compound's front end disallows such operations, but its actual contracts do not.) If it is, have `BorrowAsset` check whether a user is attempting to borrow the same asset that he or she staked as collateral and block the operation if so. Alternatively, ensure that borrow fees are greater than profits from lending.

Long term, assess whether the liquidity-mining incentives accomplish their intended purpose, and ensure that the lending incentives and borrowing costs work well together.

24. Providing additional collateral may be detrimental to borrowers in default

Severity: Informational

Difficulty: High

Type: Configuration

Finding ID: TOB-UMEE-24

Target: umee/x/leverage

Description

When a user who is in default on a loan deposits additional collateral, the collateral will be immediately liquidable. This may be surprising to users and may affect their satisfaction with the system.

Exploit Scenario

A user funds a loan and plans to use the coins he deposited as the collateral on a new loan. However, the user does not realize that he defaulted on a previous loan. As a result, bots instantly liquidate the new collateral he provided.

Recommendations

Short term, if a user is in default on a loan, consider blocking the user from calling the `LendAsset` or `SetCollateralSetting` function with an amount of collateral insufficient to collateralize the defaulted position. Alternatively, document the risks associated with calling these functions when a user has defaulted on a loan.

Long term, ensure that users cannot incur unexpected financial damage, or document the financial risks that users face.

25. Insecure storage of price-feeder keyring passwords

Severity: Medium

Difficulty: High

Type: Data Exposure

Finding ID: TOB-UMEE-25

Target: price-feeder

Description

Users can store oracle keyring passwords in the price-feeder configuration file. However, the price-feeder stores these passwords in plaintext and does not provide a warning if the configuration file has overly broad permissions (like those shown in figure 25.1).

Additionally, neither the price-feeder README nor the relevant documentation string instructs users to provide keyring passwords via standard input (figure 25.2), which is a safer approach. Moreover, neither source provides information on different keyring back ends, and the example price-feeder configuration **uses the "test" back end**.

An attacker with access to the configuration file on a user's system, or to a backup of the configuration file, could steal the user's keyring information and hijack the price-feeder oracle instance.

```
$ ls -la ./price-feeder/price-feeder.example.toml
-rwxrwxrwx 1 dc dc 848 Feb  6 10:37 ./price-feeder/price-feeder.example.toml

$ grep pass ./price-feeder/price-feeder.example.toml
pass = "exampleKeyringPassword"

$ ~/go/bin/price-feeder ./price-feeder/price-feeder.example.toml
10:42AM INF starting price-feeder oracle...
10:42AM ERR oracle tick failed error="key with
addressA4F324A31DECC0172A83E57A3625AF4B89A91F1Fnot found: key not found"
module=oracle
10:42AM INF starting price-feeder server... listen_addr=0.0.0.0:7171
```

Figure 25.1: The price-feeder does not warn the user if the configuration file used to store the keyring password in plaintext has overly broad permissions.

```
// CreateClientContext creates an SDK client Context instance used for transaction
// generation, signing and broadcasting.
func (oc OracleClient) CreateClientContext() (client.Context, error) {
    var keyringInput io.Reader
    if len(oc.KeyringPass) > 0 {
        keyringInput = newPassReader(oc.KeyringPass)
    } else {
        keyringInput = os.Stdin
    }
}
```

Figure 25.2: The price-feeder supports the use of standard input to provide keyring passwords. ([umee/price-feeder-oracle/client/client.go#L184-L192](https://github.com/umee/price-feeder-oracle/client/client.go#L184-L192))

Exploit Scenario

A user sets up a price-feeder oracle and stores the keyring password in the price-feeder configuration file, which has been misconfigured with overly broad permissions. An attacker gains access to another user account on the user's machine and is able to read the price-feeder oracle's keyring password. The attacker uses that password to access the keyring data and can then control the user's oracle account.

Recommendations

Short term, take the following steps:

- Recommend that users provide keyring passwords via standard input.
- Check the permissions of the configuration file. If the permissions are too broad, provide an error warning the user of the issue, as **openssh** does when it finds that a private key file has overly broad permissions.
- Document the risks associated with storing a keyring password in the configuration file.
- Improve the price-feeder's keyring-related documentation. Include a link to the [Cosmos SDK keyring documentation](#) so that users can learn about different keyring back ends and the addition of keyring entries, among other concepts.

26. Insufficient validation of genesis parameters

Severity: Low

Difficulty: High

Type: Data Validation

Finding ID: TOB-UMEE-26

Target: Genesis parameters

Description

A few system parameters must be set correctly for the system to function properly. The system checks the parameter input against minimum and maximum values (not always correctly) but does not check the correctness of the parameters' dependencies.

Exploit Scenario

When preparing a protocol upgrade, the Umee team accidentally introduces an invalid value into the configuration file. As a result, the upgrade is deployed with an invalid or unexpected parameter.

Recommendations

Short term, implement proper validation of configurable values to ensure that the following expected invariants hold:

- `BaseBorrowRate <= KinkBorrowRate <= MaxBorrowRate`
- `LiquidationIncentive <= some maximum`
- `CompleteLiquidationThreshold > 0`

(The third invariant is meant to prevent division by zero in the `Interpolate` method.)

27. Potential overflows in Peggo's current block calculations

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-UMEE-27

Target: Peggo's block number calculations

Description

In a few code paths, Peggo calculates the number of a delayed block by subtracting a delay value from the latest block number. This subtraction will result in an overflow and cause Peggo to operate incorrectly if it is run against a blockchain node whose latest block number is less than the delay value.

We set the severity of this finding to informational because the issue is unlikely to occur in practice; moreover, it is easy to have Peggo wait to perform the calculation until the latest block number is one that will not cause an overflow.

An overflow may occur in the following methods:

- `gravityOrchestrator.GetLastCheckedBlock` (figure 27.1)
- `gravityOrchestrator.CheckForEvents`
- `gravityOrchestrator.EthOracleMainLoop`
- `gravityRelayer.FindLatestValset`

```
// add delay to ensure minimum confirmations are received and block is finalized
currentBlock := latestHeader.Number.Uint64() - ethBlockConfirmationDelay
```

Figure 27.1: *peggo/orchestrator/oracle_resync.go#L35-L42*

Recommendations

Short term, have Peggo wait to calculate the current block number until the blockchain for which Peggo was configured reaches a block number that will not cause an overflow.

28. Peggo does not validate Ethereum address formats

Severity: **Undetermined**

Difficulty: **High**

Type: Data Validation

Finding ID: TOB-UMEE-28

Target: Peggo

Description

In several code paths in the Peggo codebase, the `go-ethereum HexToAddress` function (figure 28.1) is used to parse Ethereum addresses. This function does not return an error when the format of the address passed to it is incorrect.

The `HexToAddress` function is used in tests as well as in the following parts of the codebase:

- `peggo/cmd/peggo/bridge.go#L143` (in the peggo deploy-gravity command, to parse addresses fetched from `gravityQueryClient`)
- `peggo/cmd/peggo/bridge.go#L403` (in parsing of the peggo send-to-cosmos command's token-address argument)
- `peggo/cmd/peggo/orchestrator.go#L150` (in the peggo orchestrator [gravity-addr] command)
- `peggo/cmd/peggo/bridge.go#L536` and twice in `#L545-L555`
- `peggo/cmd/peggo/keys.go#L199`, `#L274`, and `#L299`
- `peggo/orchestrator/ethereum/gravity/message_signatures.go#L36`, `#L40`, `#L102`, and `#L117`
- `peggo/orchestrator/ethereum/gravity/submit_batch.go#L53`, `#L72`, `#L94`, `#L136`, and `#L144`
- `peggo/orchestrator/ethereum/gravity/valset_update.go#L37`, `#L55`, and `#L87`
- `peggo/orchestrator/main_loops.go#L307`
- `peggo/orchestrator/relayer/batch_relaying.go#L81-L82`, `#L237`, and `#L250`

We set the severity of this finding to undetermined because time constraints prevented us from verifying the impact of the issue. However, without additional validation of the addresses fetched from external sources, Peggo may operate on an incorrect Ethereum address.

```
// HexToAddress returns Address with byte values of s.
// If s is larger than len(h), s will be cropped from the left.
func HexToAddress(s string) Address { return BytesToAddress(FromHex(s)) }

// FromHex returns the bytes represented by the hexadecimal string s.
// s may be prefixed with "0x".
func FromHex(s string) []byte {
    if has0xPrefix(s) { s = s[2:] }
    if len(s)%2 == 1 { s = "0" + s }
    return Hex2Bytes(s)
}

// Hex2Bytes returns the bytes represented by the hexadecimal string str.
func Hex2Bytes(str string) []byte {
    h, _ := hex.DecodeString(str)
    return h
}
```

Figure 28.1: The `HexToAddress` function, which calls the `BytesToAddress`, `FromHex`, and `Hex2Bytes` functions, ignores any errors that occur during hex-decoding.

Recommendations

Short term, review the code paths that use the `HexToAddress` function, and use a function like `ValidateEthAddress` to validate Ethereum address string formats before calls to `HexToAddress`.

Long term, add tests to ensure that all code paths that use the `HexToAddress` function properly validate Ethereum address strings before parsing them.

29. Peggo takes an Ethereum private key as a command-line argument

Severity: Medium

Difficulty: High

Type: Data Exposure

Finding ID: TOB-UMEE-29

Target: Peggo's command line

Description

Certain Peggo commands take an Ethereum private key (`--eth-pk`) as a command-line argument. If an attacker gained access to a user account on a system running Peggo, the attacker would also gain access to any Ethereum private key passed through the command line. The attacker could then use the key to steal funds from the Ethereum account.

```
$ peggo orchestrator {gravityAddress} \  
  --eth-pk=$ETH_PK \  
  --eth-rpc=$ETH_RPC \  
  --relay-batches=true \  
  --relay-valsets=true \  
  --cosmos-chain-id=... \  
  --cosmos-grpc="tcp://..." \  
  --tendermint-rpc="http://..." \  
  --cosmos-keyring=... \  
  --cosmos-keyring-dir=... \  
  --cosmos-from=...
```

Figure 29.1: An example of a Peggo command line

In Linux, all users can inspect other users' commands and their arguments. A user can enable the proc filesystem's `hidepid=2 gid=0 mount options` to hide metadata about spawned processes from users who are not members of the specified group. However, in many Linux distributions, those options are not enabled by default.

Exploit Scenario

An attacker gains access to an unprivileged user account on a system running the Peggo orchestrator. The attacker then uses a tool such as `pspy` to inspect processes run on the system. When a user or script launches the Peggo orchestrator, the attacker steals the Ethereum private key passed to the orchestrator.

Recommendations

Short term, avoid using a command-line argument to pass an Ethereum private key to the Peggo program. Instead, fetch the private key from the keyring.

30. Peggo allows the use of non-local unencrypted URL schemes

Severity: Medium

Difficulty: High

Type: Cryptography

Finding ID: TOB-UMEE-30

Target: Peggo

Description

The peggo orchestrator command takes `--tendermint-rpc` and `--cosmos-grpc` flags specifying Tendermint and Cosmos remote procedure call (RPC) URLs. If an unencrypted non-local URL scheme (such as `http://<some-external-ip>/`) is passed to one of those flags, Peggo will not reject it or issue a warning to the user. As a result, an attacker connected to the same local network as the system running Peggo could launch a **man-in-the-middle attack**, intercepting and modifying the network traffic of the device.

```
$ peggo orchestrator {gravityAddress} \  
  --eth-pk=$ETH_PK \  
  --eth-rpc=$ETH_RPC \  
  --relay-batches=true \  
  --relay-valsets=true \  
  --cosmos-chain-id=... \  
  --cosmos-grpc="tcp://..." \  
  --tendermint-rpc="http://..." \  
  --cosmos-keyring=... \  
  --cosmos-keyring-dir=... \  
  --cosmos-from=...
```

Figure 30.1: The problematic flags

Exploit Scenario

A user sets up Peggo with an external Tendermint RPC address and an unencrypted URL scheme (`http://`). An attacker on the same network performs a man-in-the-middle attack, modifying the values sent to the Peggo orchestrator to his advantage.

Recommendations

Short term, warn users that they risk a man-in-the-middle attack if they set the RPC endpoint addresses to external hosts that use unencrypted schemes such as `http://`.

31. Lack of prioritization of Peggo orchestrator messages

Severity: **Undetermined**

Difficulty: **Medium**

Type: Timing

Finding ID: TOB-UMEE-31

Target: Peggo orchestrator

Description

Peggo orchestrator messages, like oracle messages ([TOB-UMEE-20](#)), are not prioritized over other transactions for inclusion in a block. As a result, if the network is highly congested, orchestrator transactions may not be included in the earliest possible block. Although the Umee system could increase the fee charged for including a Peggo orchestrator message in a block, that solution is suboptimal and may not work.

Tactics for prioritizing important transactions include the following:

- Using the custom [CheckTx](#) implementation introduced in [Tendermint version 0.35](#), which returns a priority argument
- [Reimplementing part of the Tendermint engine](#), as Terra Money did
- [Using Substrate's dispatch classes](#), which allow developers to mark transactions as normal, operational, or mandatory

Exploit Scenario

A user sends tokens from Ethereum to Umee by calling Gravity Bridge's `sendToCosmos` function. When validators notice the transaction in the Ethereum logs, they send `MsgSendToCosmosClaim` messages to Umee. However, 34% of the messages are front-run by an attacker, effectively stopping Umee from acknowledging the token transfer.

Recommendations

Short term, use a custom `CheckTx` method to prioritize Peggo orchestrator messages.

Long term, ensure that operations that affect the whole system cannot be front-run or delayed by attackers or blocked by network congestion.

32. Failure of a single broadcast Ethereum transaction causes a batch-wide failure

Severity: Undetermined

Difficulty: High

Type: Configuration

Finding ID: TOB-UMEE-32

Target: Peggo orchestrator

Description

The Peggo orchestrator broadcasts Ethereum events as Cosmos messages and sends them in batches of 10 (**at least by default**). According to a code comment (figure 32.1), if the execution of a single message fails on the Umee side, all of the other messages in the batch will also be ignored.

We set the severity of this finding to undetermined because it is unclear whether it is exploitable.

```
// runTx processes a transaction within a given execution mode, encoded transaction
// bytes, and the decoded transaction itself. All state transitions occur through
// a cached Context depending on the mode provided. State only gets persisted
// if all messages get executed successfully and the execution mode is DeliverTx.
// Note, gas execution info is always returned. A reference to a Result is
// returned if the tx does not run out of gas and if all the messages are valid
// and execute successfully. An error is returned otherwise.
func (app *BaseApp) runTx(mode runTxMode, txBytes []byte, tx sdk.Tx) (gInfo
sdk.GasInfo, result *sdk.Result, err error) {
```

Figure 32.1: [cosmos-sdk/v0.45.1/baseapp/baseapp.go#L568-L575](https://github.com/cosmos-sdk/v0.45.1/baseapp/baseapp.go#L568-L575)

Recommendations

Short term, review the practice of ignoring an entire batch of Peggo-broadcast Ethereum events when the execution of one of them fails on the Umee side, and ensure that it does not create a denial-of-service risk. Alternatively, change the system such that it can identify any messages that will fail and exclude them from the batch.

Long term, generate random messages corresponding to Ethereum events and use them in testing to check the system's handling of failed messages.

33. Peggo orchestrator's IsBatchProfitable function uses only one price oracle

Severity: Medium

Difficulty: High

Type: Data Validation

Finding ID: TOB-UMEE-33

Target: peggo/orchestrator/relayer/batch_relaying.go

Description

The Peggo orchestrator relays batches of Ethereum transactions only when doing so will be profitable (figure 33.1). To determine an operation's profitability, it uses the price of ETH in USD, which is fetched from a single source—the CoinGecko API. This creates a single point of failure, as a hacker with control of the API could effectively choose which batches Peggo would relay by manipulating the price.

The IsBatchProfitable function (figure 33.2) fetches the ETH/USD price; the gravityRelayer.priceFeeder field it uses is set earlier in the getOrchestratorCmd function (figure 33.3).

```
func (s *gravityRelayer) RelayBatches(/* (...) */) error {
    // (...)
    for tokenContract, batches := range possibleBatches {
        // (...)
        // Now we iterate through batches per token type.
        for _, batch := range batches {
            // (...)
            // If the batch is not profitable, move on to the next one.
            if !s.IsBatchProfitable(ctx, batch.Batch, estimatedGasCost,
gasPrice, s.profitMultiplier) {
                continue
            }
            // (...)
        }
    }
}
```

Figure 33.1: *peggo/orchestrator/relayer/batch_relaying.go#L173-L176*

```
func (s *gravityRelayer) IsBatchProfitable(/ * (...) */) bool {
    // (...)
    // First we get the cost of the transaction in USD
    usdEthPrice, err := s.priceFeeder.QueryETHUSDPrice()
```

Figure 33.2: *peggo/orchestrator/relayer/batch_relaying.go#L211-L223*

```

func getOrchestratorCmd() *cobra.Command {
    cmd := &cobra.Command{
        Use:   "orchestrator [gravity-addr]",
        Args:  cobra.ExactArgs(1),
        Short: "Starts the orchestrator",
        RunE: func(cmd *cobra.Command, args []string) error {
            // (...)
            coingeckoAPI := konfig.String(flagCoinGeckoAPI)
            coingeckoFeed := coingecko.NewCoingeckoPriceFeed(/* (...) */)
            // (...)
            relayer := relayer.NewGravityRelayer(
                /* (...) */,
                relayer.SetPriceFeeder(coingeckoFeed),
            )
        },
    }
}

```

Figure 33.3: [peggo/cmd/peggo/orchestrator.go#L162-L188](#)

Exploit Scenario

All Peggo orchestrator instances depend on the CoinGecko API. An attacker hacks the CoinGecko API and falsifies the ETH/USD prices provided to the Peggo relayers, causing them to relay unprofitable batches.

Recommendations

Short term, address the Peggo orchestrator's reliance on a single ETH/USD price feed. Consider using the price-feeder tool to fetch pricing information or reading prices from the Umee blockchain.

Long term, implement protections against extreme ETH/USD price changes; if the ETH/USD price changes by too large a margin, have the system stop fetching prices and require an operator to investigate whether the issue was caused by malicious behavior. Additionally, implement tests to check the orchestrator's handling of random and extreme changes in the prices reported by the price feed.

References

- [Check Coingecko prices separately from BatchRequesterLoop \(GitHub issue\)](#)

34. Rounding errors may cause the module to incur losses

Severity: **High**

Difficulty: **Medium**

Type: Data Validation

Finding ID: TOB-UMEE-34

Target: [PR #434](#) and [PR #483](#) (changes in Peggo's interest-rate calculations)

Description

The amount that a user has borrowed is calculated using `AdjustedBorrow` data and an `InterestScalar` value. Because the system uses fixed-precision decimal numbers that are truncated to integer values, there may be small rounding errors in the computation of those amounts. If an error occurs, it will benefit the user, whose repayment will be slightly lower than the amount the user borrowed.

Figure 34.1 shows a test case demonstrating this vulnerability. It should be added to the `umee/x/leverage/keeper/keeper_test.go` file. [Appendix G](#) discusses general rounding recommendations.


```

// Test rounding error bug - users can repay less than have borrowed
// It should pass
func (s *IntegrationTestSuite) TestTruncationBug() {
    lenderAddr, _ := s.initBorrowScenario()
    app, ctx := s.app, s.ctx

    // set some interesting interest scalar
    _ = s.app.LeverageKeeper.SetInterestScalar(s.ctx, umeeapp.BondDenom,
sdk.MustNewDecFromStr("2.9"))

    // save initial balances
    initialSupply := s.app.BankKeeper.GetSupply(s.ctx, umeeapp.BondDenom)
    s.Require().Equal(initialSupply.Amount.Int64(), int64(10000000000))
    initialModuleBalance := s.app.LeverageKeeper.ModuleBalance(s.ctx,
umeeapp.BondDenom)

    // lender borrows 20 umee
    err := s.app.LeverageKeeper.BorrowAsset(ctx, lenderAddr,
sdk.NewInt64Coin(umeeapp.BondDenom, 20000000))
    s.Require().NoError(err)

    // lender repays in a few transactions
    iters := int64(99)
    payOneIter := int64(2000)
    amountDelta := int64(99) // borrowed expects to "earn" this amount
    for i := int64(0); i < iters; i++ {
        repaid, err := s.app.LeverageKeeper.RepayAsset(ctx, lenderAddr,
sdk.NewInt64Coin(umeeapp.BondDenom, payOneIter))
        s.Require().NoError(err)
        s.Require().Equal(sdk.NewInt(payOneIter), repaid)
    }

    // lender repays remaining debt - less than he borrowed
    // we send 90000000, because it will be truncated to the actually owned
amount
    repaid, err := s.app.LeverageKeeper.RepayAsset(ctx, lenderAddr,
sdk.NewInt64Coin(umeeapp.BondDenom, 90000000))
    s.Require().NoError(err)
    s.Require().Equal(repaid.Int64(), 20000000-(iters*payOneIter)-amountDelta)

    // verify lender's new loan amount in the correct denom (zero)
    loanBalance := s.app.LeverageKeeper.GetBorrow(ctx, lenderAddr,
umeeapp.BondDenom)
    s.Require().Equal(loanBalance, sdk.NewInt64Coin(umeeapp.BondDenom, 0))

    // we expect total supply to not change

```

```

finalSupply := s.app.BankKeeper.GetSupply(s.ctx, umeeapp.BondDenom)
s.Require().Equal(initialSupply, finalSupply)

// verify lender's new umee balance
// should be 10 - 1k from initial + 20 from loan - 20 repaid = 9000 umee
// it is more -> borrower benefits
tokenBalance := app.BankKeeper.GetBalance(ctx, lenderAddr, umeeapp.BondDenom)
s.Require().Equal(tokenBalance, sdk.NewInt64Coin(umeeapp.BondDenom,
9000000000+amountDelta))

// in test, we didn't pay interest, so module balance should not have changed
// but it did because of rounding
moduleBalance := s.app.LeverageKeeper.ModuleBalance(s.ctx, umeeapp.BondDenom)
s.Require().NotEqual(moduleBalance, initialModuleBalance)
s.Require().Equal(moduleBalance.Int64(), int64(10000000000-amountDelta))
}

```

Figure 34.1: A test case demonstrating the rounding bug

Exploit Scenario

An attacker identifies a high-value coin. He takes out a loan and repays it in a single transaction and then repeats the process again and again. By using a single transaction for both operations, he evades the borrowing fee (i.e., the interest scalar is not increased). Because of rounding errors in the system's calculations, he turns a profit by repaying less than he borrowed each time. His profits exceed the transaction fees, and he continues his attack until he has completely drained the module of its funds.

Exploit Scenario 2

The Umee system has numerous users. Each user executes many transactions, so the system must perform many calculations. Each calculation with a rounding error causes it to lose a small amount of tokens, but eventually, the small losses add up and leave the system without the essential funds.

Recommendations

Short term, always use the rounding direction that will benefit the module rather than the user.

Long term, to ensure that users pay the necessary fees, consider prohibiting them from borrowing and repaying a loan in the same block. Additionally, use fuzz testing to ensure that it is not possible for users to secure free tokens.

References

- [How to Become a Millionaire, 0.000001 BTC at a Time](#)

35. Outdated and vulnerable dependencies

Severity: Undetermined	Difficulty: High
Type: Patching	Finding ID: TOB-UMEE-35
Target: Umee and Peggo	

Description

Both Umee and Peggo rely on outdated and vulnerable dependencies. The table below lists the problematic packages used by Umee dependencies; the yellow rows indicate packages that were also detected in Peggo dependencies.

We set the severity of this finding to undetermined because we could not confirm whether these vulnerabilities affect Umee or Peggo. However, they likely do not, since most of the CVEs are related to binaries or components that are not run in the Umee or Peggo code.

Package	Vulnerabilities
golang/github.com/coreos/etcd@3.3.13	CVE-2020-15114 CVE-2020-15136 CVE-2020-15115
pkg:golang/github.com/dgrijalva/jwt-go@3.2.0	CVE-2020-26160
golang/github.com/microcosm-cc/bluemonday@1.0.4	#111 (CWE-79)
golang/k8s.io/kubernetes@1.13.0	CVE-2020-8558, CVE-2019-11248, CVE-2019-11247, CVE-2019-11243, CVE-2021-25741, CVE-2019-9946, CVE-2020-8552, CVE-2019-11253, CVE-2020-8559, CVE-2021-25735, CVE-2019-11250, CVE-2019-11254, CVE-2019-11249, CVE-2019-11246, CVE-2019-1002100, CVE-2020-8555, CWE-601, CVE-2019-11251, CVE-2019-1002101, CVE-2020-8563, CVE-2020-8557, CVE-2019-11244

Recommendations

Short term, update the outdated and vulnerable dependencies. Even if they do not currently affect Umee or Peggo, a change in the way they are used could introduce a bug.

Long term, integrate a dependency-checking tool such as nancy into the CI/CD pipeline. Frequently update any direct dependencies, and ensure that any indirect dependencies in upstream libraries remain up to date.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	Description
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	Description
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Calculating a Dec Value's Square Root

This appendix provides a minimal example of the issue described in [TOB-UMEE-2](#). Figure C.1 shows the use of two different methods to calculate the square root of a number stored in the Cosmos SDK's Dec type.

The first method, taken from the UMEE code, uses the `strconv.ParseFloat`, `math.Sqrt`, and `NewDecFromStr(fmt.Sprintf("%f", floatNum))` scheme. The second uses the `variance.ApproxSqrt()` method. The code also contains debug prints.

The output of the code is shown in figure C.2. The value formatted through the `fmt.Sprintf("%f", floatNum)` call has lower precision than the original square root value obtained through the `math.Sqrt(floatNum)` call.

```
package main

import (
    sdk "github.com/cosmos/cosmos-sdk/types"
    "fmt"
    "math"
    "strconv"
)

func main() {
    sum, _ := sdk.NewDecFromStr("400")
    fmt.Printf("sum=%v\n", sum)

    pb := []int{1,2,3,4,5,6,7,8,9,10}
    variance := sum.QuoInt64(int64(len(pb)))
    fmt.Printf("variance=%v\n", variance)

    // Calculate square root as in UMEE
    floatNum, _ := strconv.ParseFloat(variance.String(), 64)
    fmt.Printf("floatNum=%v\n", floatNum)
    floatNum = math.Sqrt(floatNum)
    fmt.Printf("floatNum squared=%v, sprintfed=%s\n", floatNum, fmt.Sprintf("%f",
floatNum))
    standardDev, _ := sdk.NewDecFromStr(fmt.Sprintf("%f", floatNum))
    fmt.Printf("stdDev=%v\n", standardDev)

    // Calculate square root via Dec type's .ApproxSqrt() func
    standardDev2, _ := variance.ApproxSqrt()
    fmt.Printf("stdDev2=%v\n", standardDev2)
}
```

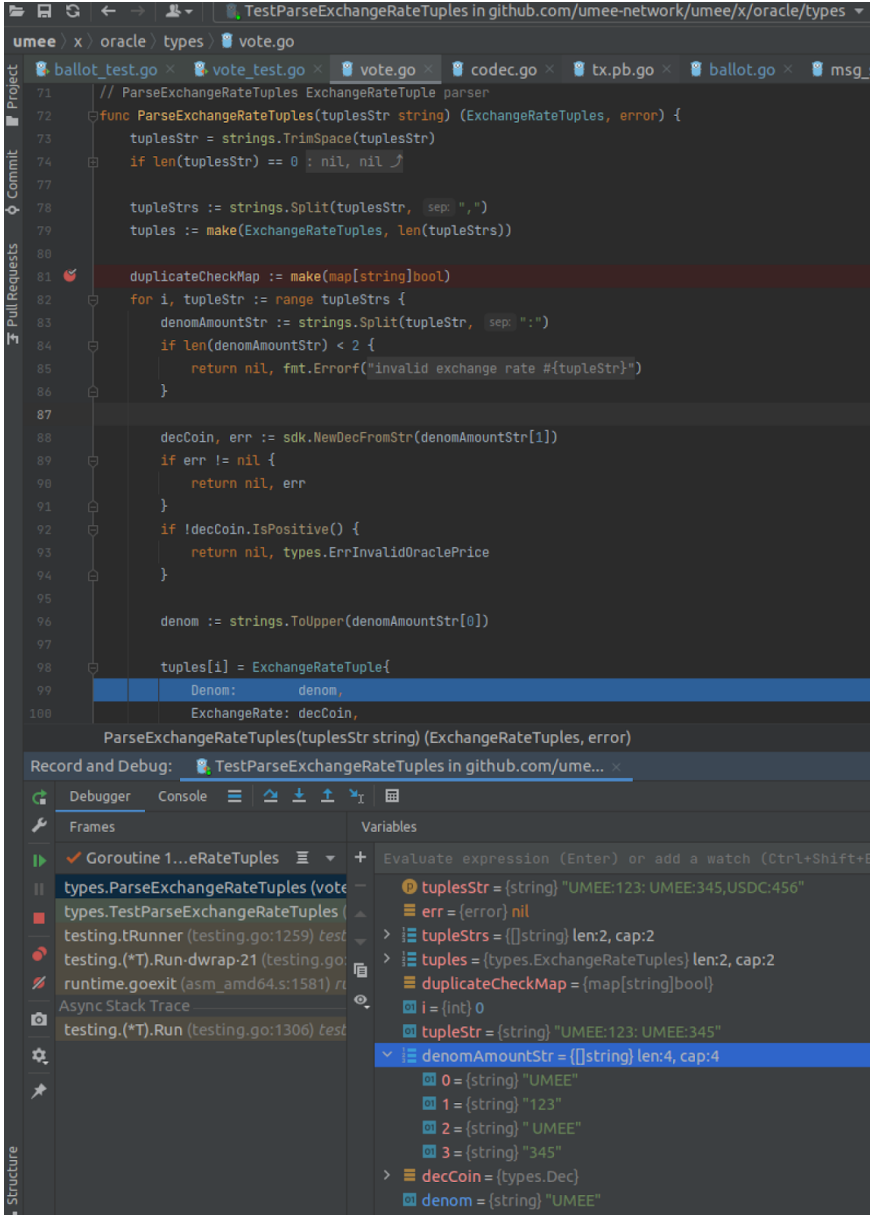
Figure C.1: Code showing the issue in the square root calculation

```
$ go run main.go
sum=400.000000000000000000000000000000
variance=40.000000000000000000000000000000
floatNum=40
floatNum squared=6.32455320336759, sprintfed=6.324555
stdDev=6.324555000000000000000000000000
stdDev2=6.32455320336758664
```

Figure C.2: The output of the code in figure C.1

D. Incorrect Exchange-Rate Parsing

The screenshot in figure D.1 demonstrates the exchange-rate parsing issue described in the Exploit Scenario of **TOB-UMEE-3**. The screenshot shows a debugging session of the `TestParseExchangeRateTuples` test, with a modified exchange rate string of "UMEE:123:UMEE:345,USDC:456". The parsed `denomAmountStr` value is a slice of length four, but only two of the values ("UMEE" and "123") are used; the other two ("UMEE" and "456") are discarded.



```
umee x oracle types vote.go
ballot_test.go x vote_test.go x vote.go x codec.go x tx.pb.go x ballot.go x msg_
71 // ParseExchangeRateTuples ExchangeRateTuple parser
72 func ParseExchangeRateTuples(tuplesStr string) (ExchangeRateTuples, error) {
73     tuplesStr = strings.TrimSpace(tuplesStr)
74     if len(tuplesStr) == 0 { nil, nil }
77
78     tupleStrs := strings.Split(tuplesStr, ",")
79     tuples := make(ExchangeRateTuples, len(tupleStrs))
80
81     duplicateCheckMap := make(map[string]bool)
82     for i, tupleStr := range tupleStrs {
83         denomAmountStr := strings.Split(tupleStr, ":")
84         if len(denomAmountStr) < 2 {
85             return nil, fmt.Errorf("invalid exchange rate #{tupleStr}")
86         }
87
88         decCoin, err := sdk.NewDecFromStr(denomAmountStr[1])
89         if err != nil {
90             return nil, err
91         }
92         if !decCoin.IsPositive() {
93             return nil, types.ErrInvalidOraclePrice
94         }
95
96         denom := strings.ToUpper(denomAmountStr[0])
97
98         tuples[i] = ExchangeRateTuple{
99             Denom: denom,
100             ExchangeRate: decCoin,
101         }
102     }
103     return tuples, nil
104 }
105
106 ParseExchangeRateTuples(tuplesStr string) (ExchangeRateTuples, error)
Record and Debug: TestParseExchangeRateTuples in github.com/umee-network/umee/x/oracle/types
Debugger Console
Frames Variables
Goroutine 1...eRateTuples
types.ParseExchangeRateTuples (vote
types.TestParseExchangeRateTuples
testing.tRunner (testing.go:1259) test
testing.(*T).Run-dwrap-21 (testing.go
runtime.goexit (asm_amd64.s:1581) r
Async Stack Trace
testing.(*T).Run (testing.go:1306) test
tuplesStr = {string} "UMEE:123:UMEE:345,USDC:456"
err = {error} nil
tupleStrs = {[string] len:2, cap:2
tuples = {types.ExchangeRateTuples len:2, cap:2
duplicateCheckMap = {map[string]bool}
i = {int} 0
tupleStr = {string} "UMEE:123:UMEE:345"
denomAmountStr = {[string] len:4, cap:4
0 = {string} "UMEE"
1 = {string} "123"
2 = {string} "UMEE"
3 = {string} "345"
decCoin = {types.Dec}
denom = {string} "UMEE"
```

Figure D.1: Code demonstrating the exchange-rate parsing issue

E. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Remove the duplicated error check in `umee/price-feeder/oracle/client/client.go#L70-L77`.**

```
func NewOracleClient(/* (...) */) (OracleClient, error) {
    oracleAddr, err := sdk.AccAddressFromBech32(oracleAddrString)
    if err != nil {
        return OracleClient{}, err
    }
    validatorAddr := sdk.ValAddress(validatorAddrString)
    if err != nil {
        return OracleClient{}, err
    }
}
```

- **Remove the redundant error check in `umee/x/oracle/keeper/keeper.go#L278-L281`.**

```
err := sdkerrors.Wrap(types.ErrNoAggregatePrevote, voter.String())
if err != nil {
    return types.AggregateExchangeRatePrevote{}, err
}
```

- **In `umee/x/oracle/keeper/reward.go#L61-L70`, if `receiverVal` is nil, consider continuing the loop early instead of computing the `rewardCoins` value.**

```
for _, winner := range ballotWinners {
    receiverVal := k.StakingKeeper.Validator(ctx, winner.Recipient)

    // reflects contribution
    rewardCoins, _ :=
periodRewards.MulDec(sdk.NewDec(winner.Weight).QuoInt64(ballotPowerSum)).TruncateDec
imal()

    // in case absence of the validator, we just skip distribution
    if receiverVal != nil && !rewardCoins.IsZero() {
        // (...)
    }
}
```

- **Use well-defined contexts instead of `context.TODO`.** There are a few parts of the codebase in which `context.TODO()` is passed as the context argument instead of a valid context.
- **Use `os.ReadFile` and `os.WriteFile` instead of `ioutil.ReadFile` and `ioutil.WriteFile`.** As of version 1.16 of Go, the latter functions call the former ones and are deprecated.
- **Use `io.ReadAll` instead of `ioutil.ReadAll`.** As of version 1.16 of Go, the latter function calls the former one and is deprecated.
- **Use `os.MkdirTemp` instead of `ioutil.TempDir`.** As of version 1.17 of Go, the latter function calls the former one.
- **Address the following issues flagged by the `Staticcheck` static analysis tool.** Additionally, review the unused append functions in the `reward_test.go` file. That code appears to be redundant, but it could have been meant to assert the `voteTargets` variable.

```
dc@ubuntu:~/audit-umee/peggo$ ~/go/bin/staticcheck ./...
test/e2e/chain.go:94:17: func (*chain).createAndInitValidatorsWithMnemonics is
unused (U1000)
test/e2e/chain.go:142:17: func (*chain).createAndInitOrchestratorsWithMnemonics is
unused (U1000)
test/e2e/e2e_setup_test.go:626:32: func
(*IntegrationTestSuite).registerValidatorOrchAddresses is unused (U1000)
test/e2e/e2e_util_test.go:122:32: func (*IntegrationTestSuite).registerOrchAddresses
is unused (U1000)
test/e2e/e2e_util_test.go:339:6: func queryUmeeAllBalances is unused (U1000)
test/e2e/keys.go:24:6: func createMemoryKey is unused (U1000)
test/e2e/keys.go:38:6: func createMemoryKeyFromMnemonic is unused (U1000)

dc@ubuntu:~/audit-umee/umee$ ~/go/bin/staticcheck ./...
ante/ante_test.go:24:2: field anteHandler is unused (U1000)
ante/fee_test.go:46:2: this value of err is never used (SA4006)
cmd/umee/cmd/root.go:210:6: func overwriteFlagDefaults is unused (U1000)
tests/e2e/chain.go:95:17: func (*chain).createAndInitValidatorsWithMnemonics is
unused (U1000)
tests/e2e/chain.go:161:17: func (*chain).createAndInitOrchestratorsWithMnemonics is
unused (U1000)
tests/e2e/e2e_setup_test.go:727:32: func
(*IntegrationTestSuite).registerValidatorOrchAddresses is unused (U1000)
tests/e2e/e2e_util_test.go:122:32: func
(*IntegrationTestSuite).registerOrchAddresses is unused (U1000)
tests/e2e/validator.go:39:2: field consensusPrivKey is unused (U1000)
x/ibctransfer/keeper/keeper_test.go:17:2: package
"github.com/cosmos/ibc-go/v2/modules/apps/transfer/types" is being imported more
than once (ST1019)
x/ibctransfer/keeper/keeper_test.go:18:2: other import of
```

```

"github.com/cosmos/ibc-go/v2/modules/apps/transfer/types"
x/leverage/types/query.pb.gw.go:16:2: package github.com/golang/protobuf/descriptor
is deprecated: See the "google.golang.org/protobuf/reflect/protorelect" package for
how to obtain an EnumDescriptor or MessageDescriptor in order to programatically
interact with the protobuf type system. (SA1019)
x/leverage/types/query.pb.gw.go:17:2: package github.com/golang/protobuf/proto is
deprecated: Use the "google.golang.org/protobuf/proto" package instead. (SA1019)
x/leverage/types/query.pb.gw.go:33:9: descriptor.ForMessage is deprecated: Not all
concrete message types satisfy the Message interface. Use MessageDescriptorProto
instead. If possible, the calling code should be rewritten to use protobuf
reflection instead. See package "google.golang.org/protobuf/reflect/protorelect"
for details. (SA1019)
x/oracle/keeper/msg_server_test.go:11:2: package
"github.com/umee-network/umee/x/oracle/types" is being imported more than once
(ST1019)
    x/oracle/keeper/msg_server_test.go:12:2: other import of
"github.com/umee-network/umee/x/oracle/types"
x/oracle/keeper/msg_server_test.go:122:2: this value of err is never used (SA4006)
x/oracle/keeper/reward_test.go:29:22: this result of append is never used, except
maybe in other appends (SA4010)
x/oracle/types/query.pb.gw.go:16:2: package github.com/golang/protobuf/descriptor is
deprecated: See the "google.golang.org/protobuf/reflect/protorelect" package for
how to obtain an EnumDescriptor or MessageDescriptor in order to programatically
interact with the protobuf type system. (SA1019)
x/oracle/types/query.pb.gw.go:17:2: package github.com/golang/protobuf/proto is
deprecated: Use the "google.golang.org/protobuf/proto" package instead. (SA1019)
x/oracle/types/query.pb.gw.go:33:9: descriptor.ForMessage is deprecated: Not all
concrete message types satisfy the Message interface. Use MessageDescriptorProto
instead. If possible, the calling code should be rewritten to use protobuf
reflection instead. See package "google.golang.org/protobuf/reflect/protorelect"
for details. (SA1019)

```

F. Automated Testing

This appendix describes the setup of the automated analysis tools used in this audit.

CodeQL

We installed CodeQL by following the ["Getting started with the CodeQL CLI" guide](#) and installing the [codeql-go extractor and libraries](#).

After installing CodeQL, we ran the following command to create the project database:

```
codeql database create codeql.db --language=go
```

We then ran the following command to query the database:

```
codeql database analyze codeql.db --additional-packs  
~/codeql/codeql-repo --format=sarif-latest --output=codeql_log.sarif  
-- tob-go-all
```

We ran CodeQL on the Umee and Peggo repositories using our `tob-go-all` query pack, which includes queries from the `codeql/go-all` built-in query pack and a few of our private queries. This enabled us to identify the issues in [TOB-UMEE-1](#) and [TOB-UMEE-5](#).

Semgrep

To install Semgrep, we used `pip` by running `python3 -m pip install semgrep`. To run Semgrep on the codebase (and to exclude all files in the `static` directory), we simply ran `semgrep --config "<CONFIGURATION>" --exclude static` in the root directory of the project.

We ran Semgrep on the Umee and Peggo repositories with our private Semgrep rules and the following configurations:

- `p/security-audit`
- `p/r2c-bug-scan`
- `p/gosec`
- `p/trailofbits`

This helped us identify a few of the code quality issues reported in [Appendix E](#).

G. Fixed-Point Rounding Recommendations

The Umee codebase uses the Dec type to implement fixed-point arithmetic. As a result, the same rounding direction (down) is used throughout. This can be beneficial to the user, as it can leave the user with dust and enable him or her to steal assets from the system (TOB-UMEE-34).

We recommend using the rounding direction that will be beneficial to the system (the module) rather than the user. The process of determining the rounding direction for each operation is described below.

Fixed-Point Primitives

In fixed-point arithmetic, both division and multiplication operations require rounding. Four functions are defined for these operations.

1. Rounding down in division (the default behavior)

$$\text{div}_{\text{down}}(a, b) = \frac{a}{b}$$

2. Rounding up in division (see [Number Conversion, Roland Backhouse](#))

$$\text{div}_{\text{up}}(a, b) = \frac{a + b - 1}{b}$$

3. Rounding down in multiplication (the default behavior)

$$\text{mul}_{\text{down}}(a, b) = \frac{a * b}{\text{precision}}$$

4. Rounding up in multiplication

$$\text{mul}_{\text{up}}(a, b) = \frac{a * b + \text{precision} - 1}{\text{precision}}$$

Determining the Rounding Direction

To determine the rounding direction of an operation (i.e., up or down), reason out the operation's outcome.

For example, the function below determines the number of a tokens that one must loan to receive a certain number of c tokens:

$$a = b * \left(1 - \frac{b1}{c - \frac{b1}{b2}}\right)$$

To benefit the module, a must tend toward a high value (\nearrow), resulting in the following:

- $b * \left(1 - \frac{b1}{c - \frac{b1}{b2}}\right)$ must \nearrow
- $\left(1 - \frac{b1}{c - \frac{b1}{b2}}\right)$ must \nearrow
- $\frac{b1}{c - \frac{b1}{b2}}$ must \searrow
- $c - \frac{b1}{b2}$ must \nearrow
- $\frac{b1}{b2}$ must \searrow

So the following formula must apply:

$$a = b *_{\nearrow} \left(1 - \frac{b1}{c - \frac{b1}{b2}}_{\searrow} \right)$$

The same analysis can be applied in all of the system's formulas.

H. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. For an up-to-date version of the checklist, see [crytic/building-secure-contracts](#).

For convenience, all [Slither](#) utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc ERC20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc ERC721
```

To follow this checklist, use the below output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❑ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❑ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on [blockchain-security-contacts](#).
- ❑ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

Contract Composition

- ❑ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's [human-summary](#) printer to identify complex code.
- ❑ **The contract uses SafeMath.** Contracts that do not use SafeMath require a higher standard of review. Inspect the contract by hand for SafeMath usage.
- ❑ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's [contract-summary](#) printer to broadly review the code used in the contract.

- ❑ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❑ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.
- ❑ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❑ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❑ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❑ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❑ **Transfer and transferFrom return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❑ **The name, decimals, and symbol functions are present if used.** These functions are optional in the ERC20 standard and may not be present.
- ❑ **Decimals returns a uint8.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.
- ❑ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❑ **The contract passes all unit tests and security properties from slither-prop.** Run the generated unit tests and then check the properties with **Echidna** and **Manticore**.

Risks of ERC20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❑ **The token is not an ERC777 token and has no external function call in transfer or transferFrom.** External calls in the transfer functions can lead to reentrancies.
- ❑ **Transfer and transferFrom should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❑ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❑ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❑ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❑ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❑ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❑ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

ERC721 Tokens

ERC721 Conformity Checks

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❑ **Transfers of tokens to the 0x0 address revert.** Several tokens allow transfers to 0x0 and consider tokens transferred to that address to have been burned; however, the ERC721 standard requires that such transfers revert.
- ❑ **safeTransferFrom functions are implemented with the correct signature.** Several token contracts do not implement these functions. A transfer of NFTs to one of those contracts can result in a loss of assets.
- ❑ **The name, decimals, and symbol functions are present if used.** These functions are optional in the ERC721 standard and may not be present.
- ❑ **If it is used, decimals returns a uint8(0).** Other values are invalid.
- ❑ **The name and symbol functions can return an empty string.** This behavior is allowed by the standard.
- ❑ **The ownerOf function reverts if the tokenId is invalid or is set to a token that has already been burned.** The function cannot return 0x0. This behavior is required by the standard, but it is not always properly implemented.
- ❑ **A transfer of an NFT clears its approvals.** This is required by the standard.
- ❑ **The token ID of an NFT cannot be changed during its lifetime.** This is required by the standard.

Common Risks of the ERC721 Standard

To mitigate the risks associated with ERC721 contracts, conduct a manual review of the following conditions:

- ❑ **The onERC721Received callback is taken into account.** External calls in the transfer functions can lead to reentrancies, especially when the callback is not explicit (e.g., in `safeMint` calls).

- ❑ **When an NFT is minted, it is safely transferred to a smart contract.** If there is a minting function, it should behave similarly to `safeTransferFrom` and properly handle the minting of new tokens to a smart contract. This will prevent a loss of assets.
- ❑ **The burning of a token clears its approvals.** If there is a burning function, it should clear the token's previous approvals.

I. Fix Log

Trail of Bits reviewed the fixes and mitigations implemented by the Umee team for issues identified in this report. See the Detailed Fix Log for information on findings that were not fixed or were fixed in ways that introduce other risks.

ID	Title	Type	Severity	Fix Status
1	Integer overflow in Peggo's deploy-erc20-raw command	Configuration	Informational	Fixed (#139)
2	Rounding of the standard deviation value may deprive voters of rewards	Configuration	Low	Fixed (#406)
3	Vulnerabilities in exchange rate commitment scheme	Data Validation	Medium	Fixed (#429, #430)
4	Validators can crash other nodes by triggering an integer overflow	Data Validation	High	Fixed (#493)
5	The repayValue variable is not used after being modified	Undefined Behavior	Undetermined	Fixed (#425)
6	Inconsistent error checks in GetSigners methods	Data Validation	Informational	Fixed (#468)
7	Incorrect price assumption in the GetExchangeRateBase function	Data Validation	High	Fixed (#436)
8	Oracle price-feeder is vulnerable to manipulation by a single malicious price feed	Data Validation	High	Fixed (#502, #536)
9	Oracle rewards may not be distributed	Configuration	Informational	Fixed (#461)

10	Risk of server-side request forgery attacks	Configuration	Medium	Fixed (#503)
11	Incorrect comparison in SetCollateralSetting method	Data Validation	Medium	Fixed (#458)
12	Voters' ability to overwrite their own pre-votes is not documented	Data Validation	Informational	Fixed (#460)
13	Lack of user-controlled limits for input amount in LiquidateBorrow	Data Validation	Medium	Fixed (#579)
14	Lack of simulation and fuzzing of leverage module invariants	Data Validation	High	Fixed (#386, #392, #401, #433)
15	Attempts to overdraw collateral cause WithdrawAsset to panic	Data Validation	Low	Fixed (#491)
16	Division by zero causes the LiquidateBorrow function to panic	Configuration	Low	Fixed (#508)
17	Architecture-dependent code	Data Validation	Informational	Not fixed
18	Weak cross-origin resource sharing settings	Data Validation	Informational	Fixed (#573)
19	price-feeder is at risk of rate limiting by public APIs	Data Validation	Medium	Fixed (#522, #551, #569, #580)
20	Lack of prioritization of oracle messages	Timing	Medium	Not fixed (#510)
21	Risk of token/uToken exchange rate manipulation	Undefined Behavior	High	Fixed (#504)

22	Collateral dust prevents the designation of defaulted loans as bad debt	Data Validation	Low	Risk accepted (#513)
23	Users can borrow assets that they are actively using as collateral	Data Validation	Undetermined	Not an issue
24	Providing additional collateral may be detrimental to borrowers in default	Configuration	Informational	Fixed (#533)
25	Insecure storage of price-feeder keyring passwords	Data Exposure	Medium	Fixed, but introduces another risk (#540)
26	Insufficient validation of genesis parameters	Data Validation	Medium	Fixed (#532)
27	Potential overflows in Peggo's current block calculations	Data Validation	Informational	Risk accepted (#176)
28	Peggo does not validate Ethereum address formats	Data Validation	Undetermined	Fixed (#217)
29	Peggo takes an Ethereum private key as a command-line argument	Data Exposure	Medium	Fixed, but introduces another risk (#174)
30	Peggo allows the use of non-local unencrypted URL schemes	Cryptography	Medium	Fixed (#205)
31	Lack of prioritization of Peggo orchestrator messages	Timing	Undetermined	Not fixed (#179)
32	Failure of a single broadcast Ethereum transaction causes a batch-wide failure	Configuration	Undetermined	Risk accepted (#180)

33	Peggo orchestrator's IsBatchProfitable function uses only one price oracle	Data Validation	Medium	Not fixed (#181)
34	Rounding errors may cause the module to incur losses	Data Validation	High	Fixed (#559)
35	Outdated and vulnerable dependencies	Patching	Undetermined	Not fixed

Detailed Fix Log

TOB-UMEE-17: Architecture-dependent code

Not fixed. Umee indicated that it will develop documentation on the architecture-dependent code.

TOB-UMEE-20: Lack of prioritization of oracle messages

Not fixed. Umee will address this issue by updating the Cosmos SDK version it uses to v0.46 when **that version is released**.

TOB-UMEE-22: Collateral dust prevents the designation of defaulted loans as bad debt

Risk accepted. Umee has **accepted this risk** and decided to manually address any dust-related issues that occur.

TOB-UMEE-23: Users can borrow assets that they are actively using as collateral

Not an issue. Umee indicated that this is intended behavior.

TOB-UMEE-25: Insecure storage of price-feeder keyring passwords

Fixed, but introduces another risk

TOB-UMEE-29: Peggo takes an Ethereum private key as a command-line argument

Fixed, but introduces another risk

Both of these issues have been fixed. However, sensitive values (keyring passwords and Ethereum private keys) are now passed in via environment variables and may be leaked in other ways:

- Environment variables are often dumped to external services through crash-logging mechanisms.
- All processes started by a user can read environment variables from the `/proc/$pid/environ` file. Attackers often use this ability to dump sensitive values passed in through environment variables (though this requires finding an arbitrary file read vulnerability in the application).

- An application can also overwrite the contents of a special `/proc/$pid/envron` file. However, overwriting the file is not as simple as calling `setenv(SECRET, "*****")`, because runtimes copy environment variables upon initialization and then operate on the copy. To clear environment variables from that special environ file, one must either overwrite the stack data in which they are located or make a low-level `prctl` system call with the `PR_SET_MM_ENV_START` and `PR_SET_MM_ENV_END` flags enabled to change the memory address of the content the file is rendered from.

We recommend that Umee take one of the following steps:

1. Document the risks of providing sensitive values through environment variables.
2. Strongly encourage users to pass sensitive values through standard input or to use a launcher that can fetch them from a service like HashiCorp Vault.
3. Allow users to pass in those values from a configuration file, document the fact that the configuration file should not be saved in backups, and provide a warning if the file has overly broad permissions when the program is started.

TOB-UMEE-27: Potential overflows in Peggo's current block calculations

Risk accepted. Umee indicated that the calculations will overflow only if the `--bridge-start-height` flag is not provided.

TOB-UMEE-31: Lack of prioritization of Peggo orchestrator messages

Not fixed. Umee stated that the update to version v0.46 of the Cosmos SDK may fix this issue; however, until that version is released, it cannot be fixed.

TOB-UMEE-32: Failure of a single broadcast Ethereum transaction causes a batch-wide failure

Risk accepted. Umee acknowledged this issue, stating that it cannot be fixed in the Umee code and that Peggo can retry any failed broadcasting operations.

TOB-UMEE-33: Peggo orchestrator's `IsBatchProfitable` function uses only one price oracle

Not fixed. Umee **acknowledged** that the system will continue to use the CoinGecko API until it begins using the Chainlink protocol.

TOB-UMEE-35: Outdated and vulnerable dependencies

Not fixed. The dependencies have not been updated.