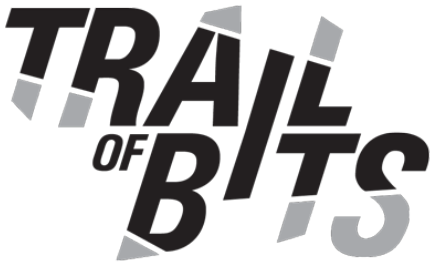

THORChain

Security Assessment

August 23, 2021

Prepared For:
THORchain | *THORChain*
dev@thorchain.org

Prepared By:
Alex Useche | *Trail of Bits*
alex.useche@trailofbits.com

Gabrielle Beck | *Trail of Bits*
gabrielle.beck@trailofbits.com

David Pokora | *Trail of Bits*
david.pokora@trailofbits.com

Opal Wright | *Trail of Bits*
opal.wright@trailofbits.com

Changelog:
August 23, 2021: Initial report draft
September 23, 2021: Added Appendix F: Fix Log

Executive Summary

From July 19 to August 20, 2021, THORChain engaged Trail of Bits to review the security of the THORChain node, threshold signature scheme (TSS) libraries, Ethereum router code, and related Terraform definitions. The Trail of Bits team conducted the assessment working from the following repositories, branches, and commit hashes:

Repository	Commit Hash
`thorchain/thornode`	91719cb1
`thorchain/tss/go-tss`	c42abc86
`thorchain/tss/tss-lib`	09a0a259
`thorchain/devops/cluster-launcher`	b7498990
`thorchain/devops/node-launcher`	25e08289
`thorchain/ethereum/eth-router`	6941c319

In the first week of the assessment, we familiarized ourselves with the codebase, read through the documentation, and scoped out the system's critical components and functionality. We performed a preliminary static analysis using `terrascan` and `semgrep` to uncover low-hanging-fruit bugs in `thornode` and in the infrastructure defined within `cluster-launcher` Terraform definitions. We focused our manual review on low-hanging fruit, such as endemic data validation issues in endpoints, insufficient error handling, and misconfiguration.

In the second week of the assessment, we began reviewing the `THORChain_Router`, `ETH_RUNE`, and `SeedService` smart contracts in the `eth-router` repository, relevant Ethereum chain client code in `thornode`, and general operations for adding and removing nodes and swapping tokens.

We spent the third and fourth weeks of the assessment reviewing deposit, withdrawal, and swap operations as well as client code for Binance and unspent transaction output (UTXO)-based chains. We began reviewing transaction and message types that could be derived from chain clients, looking for data validation issues that could lead to panics or invalid state transitions. Additionally, we conducted a partial review of the mimir logic, fee calculations, slashing logic, use of key-value storage, and API endpoints that accept user input. We also reviewed the TSS library for general correctness. Finally, we expanded our review to focus on components such as Bifrost, consensus protocols, message handlers, and managers.

In the final week of the assessment, due to our increased focus on additional components in the previous week, we broadened the scope of the assessment to uncover general correctness, data validation, error reporting, misconfiguration, and timing issues. We performed a preliminary review of the `cluster-launcher` and `node-launcher` repositories,

triaged outstanding concerns in `thornode`, and performed a final review of consensus and validation routines.

While we discovered no significant state machine issues during the audit, the complexity of the project and the maturity concerns described in the Code Maturity Evaluation section suggest that a robust testing strategy is needed to uncover potential issues. As such, we recommend investing time in documenting system invariants and in code-refactoring efforts to increase code clarity. This will ensure that invariants are enforced throughout the system and that state machine components are correct. Furthermore, expanding the test suite to include additional unit, integration, and property tests would be beneficial. Although the `heimdall` repository provides additional tests, we recommend building on these tests to cover additional edge cases. The use of property tests end to end may uncover more issues in data validation and undefined node behavior.

Update: On September 18, 2021 Trail of Bits reviewed fixes implemented for issues presented in this report. See the detailed fixes status in Appendix F.

Project Dashboard

Application Summary

Name	THORChain
Versions	`thornode` 91719cb1 `go-tss` c42abc86 `tss-lib` 09a0a259 `cluster-launcher` b7498990 `eth-router` 6941c319
Types	Go, Solidity, Terraform
Platforms	macOS, Windows, Linux, Ethereum, Web

Engagement Summary

Dates	July 19, 2021–August 20, 2021
Method	Full knowledge
Consultants Engaged	3
Level of Effort	12 person-weeks

Vulnerability Summary

Total High-Severity Issues	3	■ ■ ■
Total Medium-Severity Issues	7	■ ■ ■ ■ ■ ■ ■
Total Low-Severity Issues	3	■ ■ ■
Total Informational-Severity Issues	8	■ ■ ■ ■ ■ ■ ■ ■
Total Undetermined-Severity Issues	3	■ ■ ■
Total	24	

Category Breakdown

Access Controls	2	■ ■
Auditing and Logging	1	■
Configuration	1	■
Cryptography	6	■ ■ ■ ■ ■ ■
Data Validation	7	■ ■ ■ ■ ■ ■ ■
Error Reporting	2	■ ■

Patching	1	■
Timing	1	■
Undefined Behavior	3	■ ■ ■
Total	24	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Strong.** Access controls throughout the THORChain system seem appropriate. Asymmetric key pairs are used to validate identity, mitigating most access control concerns. However, stronger controls could be implemented to ensure the integrity of services such as `midgard` (TOB-THOR-001) and to prevent phishing attempts in the `eth-router` contracts (TOB-THOR-011).
Arithmetic	**Moderate.** The arithmetic throughout the system is underdocumented. Many variables use signed types in areas in which negative numbers are not possible. Casting between signed and unsigned types may introduce errors. Several functions rely on "magic numbers" and literals that are repeated throughout the codebase.
Configuration	**Moderate**. The codebase lacks checks for weak user-supplied configurations (TOB-THOR-019). Additional configuration options could be provided to limit allowed origins (TOB-THOR-001) and modify rate-limiting configuration values. Not all mimir values are documented or listed in a centralized file (e.g., `HALTBCHCHAIN`, `HALTBTCCHAIN`).
Data Validation	**Moderate.** Data validation such as transaction memo validation is inconsistent. In some cases, validation occurs when strings are converted into memos and in other cases when memos are converted into cosmos messages. The `eth-router` contracts lack checks on some transfer operations (TOB-THOR-008, TOB-THOR-009, TOB-THOR-010). We also found data validation issues in API endpoints (TOB-THOR-004) and key pair derivation methods (TOB-THOR-007).
Monitoring	**Moderate.** The codebase includes logging statements that are saved to text files; however, this is the extent of the centralized logging strategy.
Error Handling	**Moderate.** We identified several cases of insufficient error handling across components (TOB-THOR-012, TOB-THOR-015, TOB-THOR-016). Generally, it is important to ensure that user output is sanitized before it is logged.

Cryptography	**Weak.** Overall, the use of cryptography in the codebase is appropriate. However, we identified cases in which cryptographic algorithms are not configured according to best practices. We also identified the use of weak cryptographic schemes (TOB-THOR-006), the use of inappropriately configured schemes (TOB-THOR-005), and other correctness concerns (TOB-THOR-012, TOB-THOR-013). Cryptographic code should be more thoroughly tested to ensure that it meets modern security recommendations.
Function Composition	**Moderate.** Overall, the code is easy to understand. However, several functions are overly long and complex, resulting in code with high cyclomatic and cognitive complexity. Furthermore, several functions break the single-responsibility principle by performing tasks that could be accomplished by helper functions. The THORChain package is overly complex, containing managers, handlers, API routers, and other logic.
Resource & Rate Limiting	**Satisfactory.** Through dynamic testing, we confirmed that the project uses the tollbooth library to implement rate limiting for most API endpoints. However, we recommend taking additional measures to prevent resource exhaustion due to large packets.
Specification	**Moderate.** While detailed external documentation for `thornode` is available, other areas of the codebase lack documentation. For instance, not all memo types are documented. The documentation mentions the mimir utility but does not provide guidance on how to use it. Furthermore, the documentation does not provide guidance on the importance of creating strong passwords when deploying nodes.
Testing & Verification	**Moderate.** Total unit test coverage is low (30.5%). However, because deprecated functions remain in the codebase, this percentage is inaccurate. Furthermore, integration tests are managed in a separate repository. These deficiencies in testing can obscure the actual test coverage.

Engagement Goals

The engagement was scoped to provide a security assessment of the THORChain node, threshold signature scheme (TSS) libraries, Ethereum router code, and related Terraform definitions.

Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the chain client code interact with chain nodes appropriately? Are THORChain transactions derived appropriately from chain node events?
- Is the logic that handles deposits, withdrawals, and swaps of assets generally sound?
- Are the bond-slashing routines for misbehaving nodes appropriate?
- Is the system prone to denial of service attacks? Does the system perform data validation to prevent such attacks?
- Are the `eth-router` smart contracts secure? Do they handle funds correctly? Do they emit appropriate events for THORChain?
- Do the API endpoints for the THORChain node properly perform data validation and error handling?
- Are cryptographic providers properly configured throughout the system?

Coverage

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

- Our review of the server configurations for `thornode` revealed overly permissive Cross-Origin-Request-Sharing settings. (TOB-THOR-001).
- Our assessment of the THORChain Bifrost chain client code did not result in any immediate concerns.
- Our assessment of the `eth-router` contracts revealed data validation issues that could result in a loss of funds (TOB-THOR-008, TOB-THOR-009, TOB-THOR-010) and a design choice that may put users at increased risk of phishing attempts (TOB-THOR-011).
- Our review of the system's error handling procedures revealed insufficient error handling in cryptographic algorithms (TOB-THOR-012), unsanitized variables written into logs (TOB-THOR-015), and a failure to return mimir errors when mimir is disabled (TOB-THOR-016).
- Our analysis of the bond-slashing logic used to repay victims of stolen funds revealed that a vault holding multiple assets whose only non-zero balance is `RUNE` may lead to a panic (TOB-THOR-021).

- Our assessment of the message type validation in the codebase did not result in any concerns. Handler validation is generally appropriate, and we were unable to identify meaningful invalid state transitions. However, when fuzzing message handlers, we found that the helper function `common.GetShare` may be prone to previously unconsidered panics (TOB-THOR-024).
- Our review of the use of cryptography revealed a weak message authentication code (MAC) construction scheme (TOB-THOR-005), a code correctness concern in the hierarchical deterministic key derivation methods (TOB-THOR-006), a potential segfault in an elliptic curve cryptography routine (TOB-THOR-012), and a non-binding hash commitment scheme (TOB-THOR-013).
- Our partial review of the `cluster-launcher` and `node-launcher` repositories did not reveal any immediate vulnerabilities, although we recommend enforcing passphrase-complexity requirements (TOB-THOR-019) and enabling automatic upgrades in Google Cloud Platform (GCP) Kubernetes nodes (TOB-THOR-022).

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

❑ **Add an option to the APIs that limits origins to a set of whitelisted addresses via configuration files or environment variables.** The Cosmos SDK documentation also suggests using a reverse proxy whenever possible. TOB-THOR-001

❑ **Make channel `ch` a buffered channel of size `1`.** That way, the goroutine will be cleaned and destroyed when the function returns regardless of which case occurs first. TOB-THOR-002

❑ **Implement centralized error-handling logic that verifies and sanitizes error messages before they are returned to users.** For instance, assign types to custom errors and add a method that returns generic error messages when the type cannot be determined. TOB-THOR-003

❑ **Ensure that fee values are checked against `nil` before they are used in the `ValidateBasic` method.** TOB-THOR-004

❑ **Isolate the key material between the encryption scheme and the MAC and use a dedicated MAC algorithm, such as HMAC, rather than SHA-3.** TOB-THOR-005

❑ **Employ Argon2id or scrypt to prevent brute-force attacks against encryption keys.** These primitives offer sufficient computational complexity and are memory-hard. TOB-THOR-006

❑ **Consider using `strconv.ParseUint` instead of `strconv.Atoi` for `string`-to-`int` conversions.** This function takes arguments that specify the expected integer base and bit size of the index, enabling the check to emit errors if the parsed value is out of range. TOB-THOR-007

❑ **Add checks to `THORChain_Router`'s `transferOut` function to ensure that calls to the token's `transfer` method return a `true` (success) status, if return data is available.** This will ensure that the system can appropriately handle tokens that return a boolean value and those that return no value and simply revert. Consider using OpenZeppelin's libraries for `SafeERC20` calls. TOB-THOR-008

❑ **Add checks to `THORChain_Router`'s `_routerDeposit` function to ensure that calls to the token's `approve` method return a `true` (success) status if return data is available.** This will ensure that the system can appropriately handle tokens that return a boolean value indicating success and those that return no value and simply revert. Consider using OpenZeppelin's library for safe ERC20 calls. TOB-THOR-009

❑ **Add a check for the return value of `transferTo`.** Alternatively, if the return value has no significance, remove it and document the function's intended behavior. TOB-THOR-010

❑ **Consider replacing the `transferTo` method with one that requires users to explicitly approve funds for transfer or that whitelists trusted contracts to move funds on behalf of users.** Alternatively, provide operating procedures advising users not to use any key pair holding `RUNE` to call contracts outside of THORChain. TOB-THOR-011

❑ **Ensure that the system always checks for errors before attempting to perform any operations on a returned value.** If all points should be in the large prime order subgroup, ensure that the system checks for this condition instead of modifying the point and continuing with a new value. TOB-THOR-012

❑ **Use pre authentication encoding (PAE) and replace the safety delimiter with the lengths of the bytes that are being hashed.** This will prevent the parts of the value to be hashed from being shuffled across different parameters. TOB-THOR-013

❑ **Document all API endpoints exposed by the `thornode` API and remove any endpoints that are no longer required or should not be exposed to users.** TOB-THOR-014

❑ **Use the `keyvalue` (rather than the `msg`) parameter when logging variables with unexpected values using functions such as `Logger().Error()` and `Logger.Info()`.** Avoid interpolating unknown values into strings using format specifiers in functions such as `fmt.Sprintf()`. TOB-THOR-015

❑ **Update the functions responsible for handling mimir updates so that errors are returned when appropriate, including when `SetMimir` operations fail as a result of the `ReleaseTheKraken` flag.** TOB-THOR-016

❑ **Remove unused and unnecessary logic from the `UpdateNetwork` method to improve the readability of the code.** TOB-THOR-017

❑ **Consider performing staged updates of critical data files.** Because validation can occur against the staged updates and the existing critical data files before the updates are aggregated permanently, this could help prevent irrevocable faulting by corrupted data files. TOB-THOR-018

❑ **Implement passphrase-complexity requirements for the `thornode-password`, including length requirements and the mandatory use of numbers or symbols.** TOB-THOR-019

❑ **Require the use of a newer version of Go when building a THORChain node.** TOB-THOR-020

❑ **Revise the `subsidizePoolWithSlashBond` methods to check whether `yggTotalStolen` is zero before entering the `for` loop.** If the value is zero, the method should exit early without entering the loop. TOB-THOR-021

❑ **Enable automatic node upgrades on GCP nodes using the Terraform definition shown in figure 22.1.** TOB-THOR-022

❑ **Correct the THORChain documentation to reflect the format expected for `ADD` memos with affiliate fees.** TOB-THOR-023

❑ **Ensure that the result of the inner `Quo` operation in `common.GetShare` cannot result in zero and cause a panic after the outer `Quo` computation.** Ensure that all calling methods handle such errors appropriately, if they are bubbled up to the caller. TOB-THOR-024

Long term

❑ **Use libraries such as `gorilla/handlers` or `rs/cors` to centralize the configuration of CORS settings.** TOB-THOR-001

❑ **Run GCatch against goroutine-heavy packages to detect the mishandling of channel bugs.** Basic instances of this issue can also be detected by running this Semgrep rule. TOB-THOR-002

❑ **Add unit tests that check for different forms of invalid input and fuzz tests to identify any other errors that are returned to users unsanitized.** TOB-THOR-003

❑ **Review the codebase for potential `nil` pointer dereferences.** TOB-THOR-004

❑ **Consider using Argon2id or scrypt rather than PBKDF2 for key derivation, as they better protect against brute-force dictionary attacks using dedicated hardware.** TOB-THOR-005

❑ **Continue reviewing the use of cryptography throughout the THORChain system to ensure that modern encryption and hashing schemes are used and configured appropriately.** TOB-THOR-006

❑ **Ensure that the system checks all return values indicating success throughout the codebase.** To detect missing return value checks from high-level calls to ERC20 functions, use `slither`, a static analyzer for Solidity smart contracts. For low-level calls, perform a manual review of the codebase to ensure that successful calls return the expected values, where applicable. TOB-THOR-008

❑ **Ensure that the system checks all return values indicating success throughout the codebase.** For high-level calls to ERC20 functions, missing return value checks can be exposed using `slither`, a static analyzer for Solidity smart contracts. For low-level calls, perform a manual review to ensure that both the call succeeds and that it returns the expected success value, where applicable. TOB-THOR-009

❑ **Ensure that the system checks all return values indicating success throughout the codebase.** TOB-THOR-010

❑ **Run the Semgrep rule included in Appendix C as part of the CI/CD pipeline for the `thornode` application.** TOB-THOR-015

❑ **Run the unit test included in Appendix D to verify that requests to update mimir values return errors when `ReleaseTheKraken` is set.** TOB-THOR-016

❑ **Ensure that all critical files are validated before the `thornode` boots.** This can prevent the system from operating in an invalid state. TOB-THOR-018

❑ **Ensure that all cryptographic routines that derive hashes or keys from a user-supplied passphrase enforce passphrase-complexity requirements.** TOB-THOR-019

❑ **Ensure that all dependencies and tools used to build in THORChain are up to date.** Consider integrating dependency bots into your CI/CD pipeline to expose vulnerable dependencies. For out-of-date software that is not easily discovered by static analyzers, employ operating procedures to ensure that software is kept up to date. TOB-THOR-020

❑ **Employ property tests to uncover edge cases in data validation.** Ensure that appropriate validation and error handling exists such that future changes to code do not introduce reachable vulnerabilities. TOB-THOR-021, TOB-THOR-024

❑ **Consider employing `terrascan` to uncover common flaws, new bugs, and regressions in Terraform definition files as they are introduced.** TOB-THOR-022

Findings Summary

#	Title	Type	Severity
1	Overly permissive `Access-Control-Allow-Origin` headers	Access Controls	Medium
2	Goroutine leaks in signer process	Timing	Medium
3	Unsanitized errors returned by `thornode` API	Error Reporting	Informational
4	Nil pointer dereference in internal `thornode` API	Data Validation	Informational
5	Weak MAC construction in `tss-recovery`'s `exportKeyStore`	Cryptography	Low
6	Weak key derivation in `thornode` passphrase-based encryption routine	Cryptography	Medium
7	Insufficient data validation in hierarchical deterministic key pair derivation method	Cryptography	Informational
8	`THORChain_Router`'s `transferOut` does not check transfer return values	Data Validation	High
9	`THORChain_Router`'s `_routerDeposit` does not check `approve` return values	Data Validation	Medium
10	`THORChain_Router`'s `deposit` does not check `iRUNE.transferTo` return values	Data Validation	Informational
11	`ETH_RUNE`'s `transferTo` function can be used to extract funds from a victim calling unrelated code	Access Controls	High
12	Sending EC point not on curve results in segfault	Cryptography	High
13	Hash Based Commitment Scheme is Non-Binding	Cryptography	Undetermined
14	Undocumented API endpoints	Undefined Behavior	Informational
15	Insecure usage of logger function could lead to log injection	Auditing and Logging	Low

16	`SetMimir` fails silently when mimir is disabled via `ReleaseTheKraken`	Error Reporting	Medium
17	Unused bond reward and total reserve calculations	Data Validation	Informational
18	`thornode` can be faulted into a state which produces no blocks	Undefined Behavior	Medium
19	Lack of passphrase complexity requirements for `SIGNER_PASSWD`	Cryptography	Medium
20	Minimum Go version enforced allows vulnerable versions of Go	Patching	Low
21	`subsidizePoolWithSlashBond` is prone to division-by-zero panics	Data Validation	Undetermined
22	`cluster-launcher`'s GCP nodes do not have automatic upgrades enabled	Configuration	Informational
23	`ADD` memo with affiliate fees is incorrectly documented	Undefined Behavior	Informational
24	`common.GetShare` is prone to division-by-zero panics	Data Validation	Undetermined

1. Overly permissive Access-Control-Allow-Origin headers

Severity: Medium
Type: Access Controls
Target: `thornode/x/thorchain/client/rest/rest.go`

Difficulty: Medium
Finding ID: TOB-THOR-001

Description
The `thornode` and `midgard` APIs use cross-origin resource sharing (CORS) headers that are overly permissive. In both APIs, the `Access-Control-Allow-Origin` (ACAO) header is set to a hard-coded wildcard flag (*). As a result, the applications can receive and respond to requests originating from risky endpoints.

```
func customCORSHeader() mux.MiddlewareFunc {
   return func(next http.Handler) http.Handler {
       return http.HandlerFunc(func(w http.ResponseWriter, req *http.Request) {
           w.Header().Set("Access-Control-Allow-Origin", "*")
           next.ServeHTTP(w, req)
       })
   }
}
```

Figure 1.1: ACAO header set to a wildcard flag in the thornode API (thornode/x/thorchain/client/rest/rest.go#L87-L94)

```
func corsHandler(h http.Handler) http.Handler {
   return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
       if !strings.HasPrefix(r.URL.Path, proxiedPrefix) {
           w.Header().Set("Access-Control-Allow-Origin", "*")
       }
       h.ServeHTTP(w, r)
   })
}
```

Figure 1.2: ACAO header set to a wildcard flag in the midgard API (midgard/internal/api/api.go#L148-L155)

Exploit Scenario
A user sets up a THORChain node and exposes the API only through his loopback interface. While the API is running, he visits a malicious site. Because CORS is enabled, the malicious site is able to issue requests to the user's node over the localhost.

Recommendations
Short term, add an option to the APIs that limits origins to a set of whitelisted addresses via configuration files or environment variables. The Cosmos SDK documentation also suggests using a reverse proxy whenever possible.

Long term, use libraries such as `gorilla/handlers` or `rs/cors` to centralize the configuration of CORS settings.

2. Goroutine leaks in the signer process

Severity: Medium
Type: Timing
Target: `thornode/bifrost/signer/sign.go`

Difficulty: High
Finding ID: TOB-THOR-002

Description
The `thornode` signer process causes goroutine leaks whenever a timeout occurs before a transaction has been signed. In some situations, this could lead to an exhaustion of resources.

The `runWithContext` function takes as parameters the `signAndBroadcast` function, responsible for signing transactions, and a `context` object with a timeout of five minutes. If the transaction is signed within five minutes and `signAndBroadcast` returns an error, the error will be received by an unbuffered channel, the function will return, and the goroutine will be destroyed. However, if a transaction is not signed within five minutes, the `ctx.Done()` channel will be read, and the function will return without cleaning the goroutine resources. The more often this situation occurs, the more resources will be held up by the signer process.

```
func (s *Signer) runWithContext(ctx context.Context, fn func() error) error {
    ch := make(chan error)
    go func() {
        ch <- fn()
    }()
    select {
    case <-ctx.Done():
        return ctx.Err()
    case err := <-ch:
        return err
    }
}
```

Figure 2.1: Goroutine leaks occur whenever the function returns before receiving data from channel `ch` (thornode/bifrost/signer/sign.go#L180-L191).

Exploit Scenario
An issue occurs on the server hosting the node, causing the signer process to slow down and experience timeouts. Each timeout results in goroutine and resource leaks, eventually causing the node to crash due to memory exhaustion.

Recommendations
Short term, make channel `ch` a buffered channel of size `1`. That way, the goroutine will be cleaned and destroyed when the function returns regardless of which case occurs first.

Long term, run GCatch against goroutine-heavy packages to detect the mishandling of channel bugs. Basic instances of this issue can also be detected by running this Semgrep rule.

3. Unsanitized errors returned by thornode API

Severity: Informational
Type: Error Reporting
Target: `thornode/x/thorchain/client/rest/query.go`

Difficulty: Low
Finding ID: TOB-THOR-003

Description
Unexpected runtime errors are returned to users unsanitized. Unsanitized errors do not necessarily reveal sensitive implementation information; attackers could deduce similar implementation information by examining open-source code. However, unsanitized errors could help attackers formulate other attack vectors.

```
func getHandlerWrapper(q query.Query, storeName string, cliCtx client.Context)
http.HandlerFunc {
   return func(w http.ResponseWriter, r *http.Request) {
       heightStr, ok := r.URL.Query()["height"]
       if ok && len(heightStr) > 0 {
           height, err := strconv.ParseInt(heightStr[0], 10, 64)
           if err != nil {
               rest.WriteErrorResponse(w, http.StatusBadRequest, err.Error())
               return
           }
        [...]
       }
       param := mux.Vars(r)[restURLParam]
       text, err := r.URL.MarshalBinary()
       if err != nil {
           rest.WriteErrorResponse(w, http.StatusInternalServerError, err.Error())
           return
       }

       res, _, err := cliCtx.QueryWithData(q.Path(storeName, param,
mux.Vars(r)[restURLParam2]), text)
       if err != nil {
           rest.WriteErrorResponse(w, http.StatusNotFound, err.Error())
           return
       }
       [...]
}
```

Figure 2.1: Errors are returned to users without sanitization or verification (thornode/x/thorchain/client/rest/query.go#L23-L51).

Recommendations
Short term, implement centralized error-handling logic that verifies and sanitizes error messages before they are returned to users. For instance, assign types to custom errors and add a method that returns generic error messages when the type cannot be determined.

Long term, add unit tests that check for different forms of invalid input and fuzz tests to identify any other errors that are returned to users unsanitized.

4. Nil pointer dereference in internal thornode API

Severity: Informational
Type: Data Validation
Target: `thornode/x/thorchain/client/rest/tx.go`

Difficulty: N/A
Finding ID: TOB-THOR-004

Description
A `nil` pointer dereference error occurs when POST requests without `fee` amounts are issued to internal `thornode` API endpoints.

When a POST request is received by the `thornode` API, the `ValidateBasic` method—implemented in the `cosmos-sdk` library—is called on a `request` object. This method in turn calls `isZero` on an array of fees submitted by users. The system does not check whether the `amount` value is `nil` before `isZero` checks whether the `amount` value is zero. As a result, fees with missing `amount` values will result in `nil` dereference errors.

```
func newDepositHandler(cliCtx client.Context) http.HandlerFunc {
   return func(w http.ResponseWriter, r *http.Request) {
       var req deposit
       [...]
       baseReq := req.BaseReq.Sanitize()
       if !baseReq.ValidateBasic(w) {
           return
       }
```

Figure 4.1: The API calls `ValidateBasic` on the `request` object (thornode/x/thorchain/client/rest/tx.go#L21-L33).

```
// IsZero returns if this represents no money
func (coin Coin) IsZero() bool {
   return coin.Amount.IsZero()
}
```

Figure 4.2: A fee amount is checked against zero without first being checked against `nil` (cosmos/cosmos-sdk/types/coin.go#L60-L62).

Recommendations
Short term, ensure that fee values are checked against `nil` before they are used in the `ValidateBasic` method.

Long term, review the codebase for potential `nil` pointer dereferences.

5. Weak MAC construction in tss-recovery's exportKeyStore

Severity: Low
Type: Cryptography
Target: `go-tss/cmd/tss-recovery/key.go`

Difficulty: High
Finding ID: TOB-THOR-005

Description
When the keystore is exported in the `tss-recovery` command in `go-tss`, AES-CTR key material is derived from a PBKDF2 hash of a user-provided password. However, the message authentication code (MAC) is also derived from the same key material as the encryption key, as shown in figure 5.1.

```
derivedKey := pbkdf2.Key([]byte(password), salt, 262144, 32, sha256.New)
 encryptKey := derivedKey[:32]
 cipherText, err := aesCTRXOR(encryptKey, privKey, iv)
 if err != nil {
     return nil, err
 }

 hasher := sha3.NewLegacyKeccak512()
 _, err = hasher.Write(derivedKey[16:32])
```

Figure 5.1: AES-CTR encryption key material is reused for MAC construction (go-tss/cmd/tss-recovery/key.go#L48-56).

It is not best practice to reuse key material across primitives, as security will be based on the weaker of the two primitives. Reusing key material for encryption and authentication is acceptable when using AES-GCM because the key is fed into the same block cipher. In this case, however, the key material is being used by two different primitives.

In addition to reusing key material across two different primitives, the keystore does not use a dedicated MAC primitive, such as a hash-based message authentication code (HMAC). As shown in figure 5.1, the ciphertext is authenticated with a SHA-3 hash computed over the key material and the ciphertext; however, use of the HMAC primitive rather than SHA-3 would strengthen the MAC.

Exploit Scenario
Eve, an attacker, realizes that `go-tss` reuses key material for the AES-CTR encryption key and the MAC in the `exportKeyStore` routine, and she shares this information with other attackers. Later, details of the MAC construction are exposed due to a leak or another cryptographic weakness. This could lead to the weakening or exposure of the AES-CTR encryption key.

Recommendations
Short term, isolate the key material between the encryption scheme and the MAC and use a dedicated MAC algorithm, such as HMAC, rather than SHA-3.

Long term, consider using Argon2id or scrypt rather than PBKDF2 for key derivation, as they better protect against brute-force dictionary attacks using dedicated hardware.

6. Weak key derivation in thornode passphrase-based encryption routine

Severity: Medium
Type: Cryptography
Target: `thornode/common/encryption.go`

Difficulty: High
Finding ID: TOB-THOR-006

Description
The `common` directory in the `thornode` repository contains a passphrase-based encryption provider. Although the use of AES-GCM in this scheme is appropriate, the keys for the encryption routines are derived from simple MD5 hashes of user passphrases. MD5 is an insecure hash function and should never be used in a security-relevant context. Because the key material is generated from a human-provided passphrase, possibly one with low entropy, a dedicated password-based key derivation function that can prevent brute-force dictionary attacks would be more secure.

```
func createHash(key string) (string, error) {
   hasher := md5.New()
   _, err := hasher.Write([]byte(key))
   return hex.EncodeToString(hasher.Sum(nil)), err
}

// Encrypt the input data with passphrase
func Encrypt(data []byte, passphrase string) ([]byte, error) {
   hash, err := createHash(passphrase)
   if err != nil {
       return nil, err
   }

   block, _ := aes.NewCipher([]byte(hash))
   gcm, err := cipher.NewGCM(block)
   if err != nil {
       return nil, err
   }
```

Figure 6.1: SignerStore uses MD5 hash passwords as AES-GCM keys (thornode/common/encryption.go#L12-29).

Exploit Scenario
Eve, an attacker, notes that `thornode` uses the common encryption helper for sensitive data and a single-pass MD5 hash key derivation process on a user-provided passphrase. Due to the speed at which AES ciphers can be executed and the lack of computational complexity in the key derivation process, Eve is able to brute-force passphrases for encrypted data at a very fast rate.

Recommendations
Short term, employ Argon2id or scrypt to prevent brute-force attacks against encryption keys. These primitives offer sufficient computational complexity and are memory-hard.

Long term, continue reviewing the use of cryptography throughout the THORChain system to ensure that modern encryption and hashing schemes are used and configured appropriately.

7. Insufficient data validation in hierarchical deterministic key pair derivation method

Severity: Informational
Type: Cryptography
Target: `thornode/cmd/thornode/cmd/ed25519_keys.go`

Difficulty: High
Finding ID: TOB-THOR-007

Description

The `thornode` repository provides BIP32-compliant hierarchical deterministic key pair derivation functions. The use of these functions in the codebase is not problematic; however, these functions could be exploitable if they were used in a context in which a bug enabled the derivation of the same key pair from two different key paths.

```
func derivePrivateKeyForPath(privKeyBytes, chainCode [32]byte, path string) ([32]byte,
error) {
   data := privKeyBytes
   parts := strings.Split(path, "/")
   for _, part := range parts {
       [...]
       idx, err := strconv.Atoi(part)
       if err != nil {
           return [32]byte{}, fmt.Errorf("invalid BIP 32 path: %s", err)
       }
       if idx < 0 {
           return [32]byte{}, errors.New("invalid BIP 32 path: index negative ot too large")
       }
       data, chainCode = derivePrivateKey(data, chainCode, uint32(idx), harden)
   [...]
   return derivedKey, nil
}
```

Figure 7.1: The hierarchical deterministic key pair derivation method could parse a 64-bit integer for the key path index but cast it to an unsigned 32-bit integer prior to key derivation (thornode/cmd/thornode/cmd/ed25519_keys.go#L171-197).

The `strconv.Atoi` function converts the `string` provided for the key path index to an `int`. This `int` is later cast to a `uint32`. However, the `int` type's bit size is architecture dependent: on a 64-bit system, the `int` will be 64 bits wide. This means that if `4294967303` is provided for the key path index, it would be appropriately parsed into an `int64`; however, the `uint32` casting operation would result in a value of `7`. If one key path index is `7` while another is `4294967303`, they will produce the same key.

While the codebase includes a check that should emit an error for indexes that are negative or too large, the condition that triggers this error does not capture values that are too large.

Recommendations

Short term, consider using `strconv.ParseUint` instead of `strconv.Atoi` for `string`-to-`int` conversions. This function takes arguments that specify the expected integer base and bit size of the index, enabling the check to emit errors if the parsed value is out of range.

8. THORChain_Router's transferOut does not check transfer return values

Severity: High
Type: Data Validation
Target: `eth-router/contracts/THORChain_Router.sol`

Difficulty: Medium
Finding ID: TOB-THOR-008

Description
ERC20 tokens expose functions that execute and approve fund transfers. The ERC20 interface defines the `transfer`, `transferFrom`, and `approve` functions as returning a boolean to indicate whether fund transfers were successful. However, many popular tokens have implemented this interface incorrectly: the token functions have no return values and simply revert when transfers fail. Contracts using earlier versions of Solidity take as the return value the random data stored from the memory slot in which the return value should be stored. However, the Solidity compiler has since been updated with checks for expected return values, causing contracts to error out in case of a failure.

To work around this issue, `THORChain_Router` uses functions that perform low-level calls to the `transfer`, `transferFrom`, and `approve` functions to ensure that the calls succeed and to check the return values, if they exist, much like OpenZeppelin's SafeERC20 functions. However, when calling the ERC20-defined `transfer` function, `THORChain_Router`'s `transferOut` function checks whether the call succeeded but does not check the return value. If a call to the `transfer` function succeeds but returns `false`—indicating that the transfer itself failed—the `transferOut` function will falsely indicate that the transfer succeeded.

```
   // Any vault calls to transfer any asset to any recipient.
   function transferOut(address payable to, address asset, uint amount, string memory memo)
public payable nonReentrant {
       uint safeAmount; bool success;
       if(asset == address(0)){
           safeAmount = msg.value;
           (success,) = to.call{value:msg.value}(""); // Send ETH
       } else {
           vaultAllowance[msg.sender][asset] -= amount; // Reduce allowance
           (success,) = asset.call(abi.encodeWithSignature("transfer(address,uint256)" , to,
amount));
           safeAmount = amount;
       }
       require(success);
       emit TransferOut(msg.sender, to, asset, safeAmount, memo);
   }
```

Figure 8.1: `THORChain_Router` does not check the return value of the low-level `transfer` call (eth-router/contracts/THORChain_Router.sol#L99-112).

If `THORChain_Router` assumes that failed fund transfers were successful, funds could be lost. The router would decrease the asset's allowance for the given vault, preventing the vault from allowing a user to withdraw the funds again.

Exploit Scenario
Bob, a user of THORChain, initiates a transfer of a compatible ERC20 token in an attempt to swap it. The transfer fails. Due to the lack of a return value check (figure 8.1), `THORChain_Router` reports that the funds have been successfully transferred. This may affect THORChain's asset accountability and cause Bob or a user attempting to buy his asset to incur losses.

Recommendations
Short term, add checks to `THORChain_Router`'s `transferOut` function to ensure that calls to the token's `transfer` method return a `true` (success) status, if return data is available. This will ensure that the system can appropriately handle tokens that return a boolean value and those that return no value and simply revert. Consider using OpenZeppelin's libraries for `SafeERC20` calls.

Long term, ensure that the system checks all return values indicating success throughout the codebase. To detect missing return value checks from high-level calls to ERC20 functions, use `slither`, a static analyzer for Solidity smart contracts. For low-level calls, perform a manual review of the codebase to ensure that successful calls return the expected values, where applicable.

9. THORChain_Router's _routerDeposit does not check approve return values

Severity: Medium
Type: Data Validation
Target: `eth-router/contracts/THORChain_Router.sol`

Difficulty: High
Finding ID: TOB-THOR-009

Description

`THORChain_Router`'s `_routerDeposit` method enables users to transfer funds from the current router to a new one. The method calls the ERC20 `approve` method and then the router's `depositWithExpiry` method, which uses the ERC20 `transferFrom` method to facilitate the approved transfer. Because many popular ERC20 tokens have implemented these methods incorrectly—the tokens simply revert upon failure instead of returning a boolean—many systems perform low-level calls to check the return data if it exists.

`THORChain_Router` checks whether the call to `approve` succeeded but does not check the return value. If the `approve` function successfully executes but returns `false`—indicating that the operation itself failed—`THORChain_Router`'s check will falsely indicate that the funds have been approved for transfer.

```
   // Adjust allowance and forwards funds to new router, credits allowance to desired vault
   function _routerDeposit(address _router, address _vault, address _asset, uint _amount,
string memory _memo) internal {
       vaultAllowance[msg.sender][_asset] -= _amount;
       (bool success,) = _asset.call(abi.encodeWithSignature("approve(address,uint256)",
_router, _amount)); // Approve to transfer
       require(success);
       iROUTER(_router).depositWithExpiry(_vault, _asset, _amount, _memo, type(uint).max);
// Transfer by depositing
   }
```

Figure 9.1: `THORChain_Router` does not check return value of the low-level approve call (eth-router/contracts/THORChain_Router.sol#L156-163).

This issue should not have a meaningful impact on its own. While `THORChain_Router`'s check may falsely indicate that transfers have been approved, the subsequent calls to `depositWithExpiry` should fail due to the lack of allowance. However, if the receiving router provides a different `deposit` implementation, the issue may result in the loss of funds.

Exploit Scenario

Bob, a user of THORChain, initiates a `transferAllowance` call to transfer assets from the current router to a new one. Because of the lack of return value checks on the `approve` function, shown in figure 9.1, `THORChain_Router` falsely reports that the funds have been approved for transfer. If the receiving router does not appropriately check return values, execution of the transfer will continue, despite the lack of approval, and Bob will lose funds.

Recommendations
Short term, add checks to `THORChain_Router`'s `_routerDeposit` function to ensure that calls to the token's `approve` method return a `true` (success) status if return data is available. This will ensure that the system can appropriately handle tokens that return a boolean value indicating success and those that return no value and simply revert. Consider using OpenZeppelin's library for safe ERC20 calls.

Long term, ensure that the system checks all return values indicating success throughout the codebase. For high-level calls to ERC20 functions, missing return value checks can be exposed using `slither`, a static analyzer for Solidity smart contracts. For low-level calls, perform a manual review to ensure that both the call succeeds and that it returns the expected success value, where applicable.

10. THORChain_Router's deposit does not check iRUNE.transferTo return values

Severity: Informational
Type: Data Validation
Target: `eth-router/contracts/THORChain_Router.sol`

Difficulty: N/A
Finding ID: TOB-THOR-010

Description
The `THORChain_Router` contract contains a `deposit` function that allows users to deposit their funds into a vault through the router. To transfer `RUNE`, users call the `ETH_RUNE.transferTo` method, which returns a boolean indicating whether the transfer succeeded. However, the `deposit` function does not check the return value of `transferTo`.

```
   // Deposit an asset with a memo. ETH is forwarded, ERC-20 stays in ROUTER
   function deposit(address payable vault, address asset, uint amount, string memory memo)
public payable nonReentrant{
       uint safeAmount;
       if(asset == address(0)){
           safeAmount = msg.value;
           (bool success,) = vault.call{value:safeAmount}("");
           require(success);
       } else if(asset == RUNE) {
           safeAmount = amount;
           iRUNE(RUNE).transferTo(address(this), amount);
           iERC20(RUNE).burn(amount);
```

Figure 10.1: `THORChain_Router`'s `deposit` method does not check the return value of `transferTo`, which indicates whether the transfer succeeded (eth-router/contracts/THORChain_Router.sol#L67-77).

Because `ETH_RUNE` reverts on a transfer's failure, this issue currently has no impact on the security of the system. However, future changes to the codebase may introduce an issue.

Recommendations
Short term, add a check for the return value of `transferTo`. Alternatively, if the return value has no significance, remove it and document the function's intended behavior.

Long term, ensure that the system checks all return values indicating success throughout the codebase.

11. ETH_RUNE's transferTo function can be used to extract funds from a victim calling unrelated code

Severity: High
Type: Access Controls
Target: `eth-router/contracts/eth_rune.sol`

Difficulty: Medium
Finding ID: TOB-THOR-011

Description
The `ETH_RUNE` smart contract represents `RUNE` on Ethereum. This contract defines a function, `transferTo`, that can send the funds of the `tx.origin` address (the address that triggered the entire call stack) to any other address. This means that any smart contract that a user calls can call this function to drain the user's `RUNE`.

Consider the following method:

```
 /**
  * Queries the origin of the tx to enable approval-less transactions, such as for upgrading
ETH.RUNE to THOR.RUNE.
  * Beware phishing contracts that could steal tokens by intercepting tx.origin.
  * The risks of this are the same as infinite-approved contracts which are widespread.
  * Acknowledge it is non-standard, but the ERC-20 standard is less-than-desired.
  */
 function transferTo(address recipient, uint256 amount) public returns (bool) {
   _transfer(tx.origin, recipient, amount);
   return true;
 }
```

Figure 11.1: `ETH_RUNE` provides a method to transfer `tx.origin`'s funds to any other address; this function can be used to extract a user's funds (eth-router/contracts/eth_rune.sol#L153-162).

Although the developers have warned users of phishing contracts that can steal tokens, the developers should provide operating procedures advising users not to use their private keys holding `RUNE` for interactions with any other system.

Exploit Scenario
Bob is a user of THORChain who has a large amount of `RUNE` associated with his key pair. Eve, an attacker, maintains another smart contract, which Bob interacts with. Knowing `transferTo` is vulnerable, Bob checks the source code of every contract he interacts with on Etherscan. After verifying that Eve's contract is not malicious, he sends a transaction to interact with it. Eve notices that there is a pending transaction from a user who holds a large amount of `RUNE`. She sends another transaction with a larger gas fee to upgrade her contract code such that it will steal Bob's funds. This leads to a race condition in which Bob's pending call to Eve's contract is fulfilled after the contract code is changed. As a result, Bob loses his funds, despite being careful to watch for phishing contracts.

Recommendations

Short term, consider replacing the `transferTo` method with one that requires users to explicitly approve funds for transfer or that whitelists trusted contracts to move funds on behalf of users. Alternatively, provide operating procedures advising users not to use any key pair holding `RUNE` to call contracts outside of THORChain.

12. Sending an off-curve point results in a segfault

Severity: High | Difficulty: Low
Type: Cryptography | Finding ID: TOB-THOR-012
Target: `tss-lib/eddsa/signing/round_3.go`, `tss-lib/eddsa/keygen/round_3.go`

Description
When executing the key-generation and signing algorithms for the EdDSA TSS, users must send elliptic curve points to one another in particular rounds of the operation. These curve points are reconstructed as `ECPoint` objects in memory. If a party commits to a point that is not on the curve, the function `NewECPoint` returns an error and a `nil` pointer value.

```
// Creates a new ECPoint and checks that the given coordinates are on the elliptic curve.
func NewECPoint(curve elliptic.Curve, X, Y *big.Int) (*ECPoint, error) {
   if !isOnCurve(curve, X, Y) {
       return nil, fmt.Errorf("NewECPoint: the given point is not on the elliptic curve")
   }
   return &ECPoint{curve, [2]*big.Int{X, Y}, 1}, nil
}
```

Figure 12.1: If a point is not on the curve, the pointer value is `nil` (tss-lib/crypto/ecpoint.go#L40-46).

In several parts of the code, the system does not check this error value before attempting to modify the point to be part of the large prime order subgroup of curve25519 (the elliptic curve operation specified in `EightInvEight`). Therefore, if the point is set by a malicious party, a segfault can occur.

```
       if !ok {
           return round.WrapError(errors.New("de-commitment verify failed"))
       }
       if len(coordinates) != 2 {
           return round.WrapError(errors.New("length of de-commitment should be 2"))
       }

       Rj, err := crypto.NewECPoint(tss.EC(), coordinates[0], coordinates[1])
       Rj = Rj.EightInvEight()
       if err != nil {
           return round.WrapError(errors.Wrapf(err, "NewECPoint(Rj)"), Pj)
       }
```

Figure 12.2: In round 3 of the signing algorithm, operations are done on elliptic curve points before testing whether the initialization of the point was successful (tss-lib/eddsa/signing/round_3.go#L44-56).

```
           ok, flatPolyGs := cmtDeCmt.DeCommit()
           if !ok || flatPolyGs == nil {
               ch <- vssOut{errors.New("de-commitment verify failed"), nil}
               return
```

```
        }
        PjVs, err := crypto.UnFlattenECPoints(tss.EC(), flatPolyGs)
        for i, PjV := range PjVs {
            PjVs[i] = PjV.EightInvEight()
        }
        if err != nil {
            ch <- vssOut{err, nil}
            return
        }
```

Figure 12.3: In round 3 of the key generation algorithm, operations are done on elliptic curve points before testing whether the initialization was successful (tss-lib/eddsa/keygen/round_3.go#L75-87).

Exploit Scenario

Alice joins the THORChain network as a node, staking some number of tokens. During a batch signing operation, she commits to an off-curve point. Every other node that is part of the signing operation experiences a segfault and is unable to complete signing transactions. Due to the code segfaulting and depending on how blame is assigned, programmatically identifying Alice may be impossible, and it may be necessary to consult logs. Depending on how the broadcast channel is implemented, a small group may also be able to mount a 51% attack by selectively handing out off-curve points to cause nodes to segfault and to increase its chances of being chosen from the signer pool.

Recommendation

Short term, ensure that the system always checks for errors before attempting to perform any operations on a returned value. If all points should be in the large prime order subgroup, ensure that the system checks for this condition instead of modifying the point and continuing with a new value.

13. Hash-based commitment scheme is non-binding

Severity: Undetermined
Type: Cryptography
Target: `tss-lib/crypto/commitments/hashed.go`

Difficulty: High
Finding ID: TOB-THOR-013

Description
The hash-based commitment scheme implemented in the TSS library is non-binding due to a canonicalization failure in the use of the hash function. The only messages that users can commit to are values at structured elliptic curve *x* and *y* coordinates, so this issue is not easily exploitable. However, because the scheme is non-binding, an attacker could change the message he has committed to in a multiparty computation, compromising the cryptographic security offered by the protocol.

Normally, the hash-based commitment scheme constructed as `C = H(r || m)`—where `H` is a random oracle, `r` is randomness, and `m` is the message—is secure. However, care must be taken to separate the domain of the randomness, `r`, from the message, `m`. For example, an attacker could claim that he is committing to "`dogs`" using randomness "`[garbage string] I love`" when he has actually committed to "`I love dogs`" using "`[garbage string]`." Because these strings are the same length, `H([garbage string] I love || dogs) = H([garbage string] || I love dogs)`. Note that in this example, `||` represents concatenation and is not a delimiter or an actual character in the string. Therefore, it is imperative that a set number of random bytes always be used for this commitment scheme.

In the `tss-lib` codebase, the commitment scheme simply prepends the randomness, `r`, to the slice representing the message, `m`.

```
func NewHashCommitmentWithRandomness(r *big.Int, secrets ...*big.Int) *HashCommitDecommit {
   parts := make([]*big.Int, len(secrets)+1)
   parts[0] = r
   for i := 1; i < len(parts); i++ {
       parts[i] = secrets[i-1]
   }
   hash := common.SHA512_256i(parts...)

   cmt := &HashCommitDecommit{}
   cmt.C = hash
   cmt.D = parts
   return cmt
}
```

Figure 13.1: When creating a new `HashCommitDecommit` object, the randomness is just another element in the slice of `int`s called `parts` passed to the hash function (tss-lib/crypto/commitments/hashed.go#L32-44).

The hash function attempts to implement domain separation by hashing the length of `parts` and providing a safety delimiter, `$`, between each element in the slice. For example, if

`parts = [a,b]`, where `a` is the randomness and `b` is the message, the commitment, `C`, is `H(2a$b$)`. However, the use of the character `$` as a delimiter is not safe because it is itself a valid `int` value. Moreover, due to the use of `Bytes` instead of `FillBytes`, the lengths of `a` and `b` can vary.

```
func SHA512_256i(in ...*big.Int) *big.Int {
   [...]
   binary.LittleEndian.PutUint64(inLenBz, uint64(inLen))
   ptrs := make([][]byte, inLen)
   for i, n := range in {
       ptrs[i] = n.Bytes()
       bzSize += len(ptrs[i])
   }
   data = make([]byte, 0, len(inLenBz)+bzSize+inLen)
   data = append(data, inLenBz...)
   for i := range in {
       data = append(data, ptrs[i]...)
       data = append(data, hashInputDelimiter) // safety delimiter
   }
   [...]
   if _, err := state.Write(data); err != nil {
       Logger.Errorf("SHA512_256i Write() failed: %v", err)
       return nil
   }
   return new(big.Int).SetBytes(state.Sum(nil))
```

Figure 13.2: The use of `Bytes()` allows the number of digits in each part of the hash to vary; the safety delimiter is not safe because it is a valid value for the `int`s themselves (tss-lib/common/hash.go#L53-83).

This means that a party can commit to a value such as `$$$$$b` and later, after seeing the values that other parties have committed to, claim that the value was `$$$$b`, `$$$b`, `$$b`, and so on. Because many higher-level primitives could rely on this commitment scheme, this security issue could be pervasive in the codebase.

Exploit Scenario

A developer at THORChain modifies the codebase to use a more succinct representation of curve25519 points that includes only the representation of the *x* coordinate (i.e., X25519). Because the commitment scheme is non-binding and every 32-byte value is a valid *x* coordinate, a participant in the multiparty computation protocol can commit to an *x* value of `$$$$$$$$a` and later claim that his value was any value of the following form: `${repeated <= 8 times}a`. This incorrect use of the hash function presents no clear risk. However, this implementation is brittle and could lead to problems in the future.

Recommendations

Short term, use pre authentication encoding (PAE) and replace the safety delimiter with the lengths of the bytes that are being hashed. This will prevent the parts of the value to be hashed from being shuffled across different parameters.

14. Undocumented API endpoints

Severity: Informational
Type: Undefined Behavior
Target: `thornode`

Difficulty: N/A
Finding ID: TOB-THOR-014

Description
The following API endpoints exposed by the `thornode` API are omitted from the generated swagger documentation:

- `/thorchain/mimir`
- `/thorchain/mimir/{key}`
- `/thorchain/metrics`
- `/thorchain/ban/{address}`

While this deficiency in the documentation does not present a security risk, it is best practice to explicitly document the intended purposes and behaviors of all API endpoints.

Furthermore, keeping an up-to-date inventory of all API endpoints would help the THORChain team track the endpoints that should be deprecated and the functionality that should not be exposed to users.

Recommendations
Short term, document all API endpoints exposed by the `thornode` API and remove any endpoints that are no longer required or should not be exposed to users.

15. Insecure use of logger function could lead to log injection

Severity: Low
Type: Auditing and Logging
Target: `thornode`

Difficulty: High
Finding ID: TOB-THOR-015

Description
The `thornode` code uses logging functionality provided by the Tendermint library to log messages of different levels. Each level (`Debug`, `Info`, `Error`) takes in a message of type `string` and a variadic number of key-values of type `interface{}`. While Tendermind sanitizes and encodes values in each of the key-value parameters, it does not sanitize the `message` string. As a result, using functions like `fmt.Sprintf` that are used to interpolate non-constant values into a log message could cause messages to be printed in an unexpected format. Furthermore, an attacker could inject log messages if they are able to control the variables used in logger functions.

```
func (h WithdrawLiquidityHandler) Run(ctx cosmos.Context, m cosmos.Msg) (*cosmos.Result,
error) {
   msg, ok := m.(*MsgWithdrawLiquidity)
   [...]
   ctx.Logger().Info(fmt.Sprintf("receive MsgWithdrawLiquidity from : %s(%s) withdraw (%s)",
*msg, msg.WithdrawAddress, msg.BasisPoints))
```

Figure 15.1: `fmt.Sprintf` is used to interpolate values from a transaction message into a log (thornode/x/thorchain/handler_withdraw.go#L28-L33).

No instances of logging functionality in the code appear to be directly exploitable. However, if the system continues to log messages by interpolating non-constant values into the message parameter of Tendemint's logger functions, as the `thornode` code grows in size and complexity, developers could introduce log injection vulnerabilities.

Exploit Scenario
An attacker performs illegal actions in the `thornode` application. They inject custom logs into the application via memo messages to obfuscate their malicious activities, hampering any investigation.

Recommendations
Short term, use the `keyvalue` (rather than the `msg`) parameter when logging variables with unexpected values using functions such as `Logger().Error()` and `Logger.Info()`. Avoid interpolating unknown values into strings using format specifiers in functions such as `fmt.Sprintf()`.

Long term, run the Semgrep rule included in Appendix C as part of the CI/CD pipeline for the `thornode` application.

16. SetMimir fails silently when mimir is disabled via ReleaseTheKraken

Severity: Medium
Type: Error Reporting
Target: `thornode/x/thorchain/handler_mimir.go`

Difficulty: Medium
Finding ID: TOB-THOR-016

Description
Through a utility called mimir, node administrators can update global node variables that affect the behavior of node operations. This feature can be disabled by setting a variable called `ReleaseTheKraken` to a value higher than `-1`. When `ReleaseTheKraken` is set, any attempts to read or write global mimir values will fail. However, when `ReleaseTheKraken` is set and a user requests to update mimir values, the user receives no indication of failure. Without an indication of failure, node administrators who attempt to change mimir values when `ReleaseTheKraken` is set may be led to believe that their requests to update the values were successful.

```
func (h MimirHandler) handleCurrent(ctx cosmos.Context, msg MsgMimir) error {
   h.mgr.Keeper().SetMimir(ctx, msg.Key, msg.Value)

   ctx.EventManager().EmitEvent(
       cosmos.NewEvent("set_mimir",
           cosmos.NewAttribute("key", msg.Key),
           cosmos.NewAttribute("value", strconv.FormatInt(msg.Value, 10))))

   return nil
}
```

Figure 16.1: The `handleCurrent` function always returns a `nil` error (thornode/x/thorchain/handler_mimir.go#L83-92).

```
// SetMimir save a mimir value to key value store
func (k KVStore) SetMimir(ctx cosmos.Context, key string, value int64) {
   // if we have the kraken, mimir is no more, ignore him
   if k.haveKraken(ctx) {
       return
   }
   k.setInt64(ctx, k.GetKey(ctx, prefixMimir, key), value)
}
```

Figure 16.2: The `SetMirmir` function returns when `ReleaseTheKraken` is set (thornode/x/thorchain/keeper/v1/keeper_mimir.go#L27-33).

Exploit Scenario
A node administrator attempts to halt `BNB` trading by using the mimir utility to set the `HaltBNBTrading` to `1`. Unbeknownst to the administrator, `ReleaseTheKraken` was set by another node administrator, so the operation fails. The administrator does not realize that the operation failed, and `BNB` continues to be enabled.

Recommendations
Short term, update the functions responsible for handling mimir updates so that errors are returned when appropriate, including when `SetMimir` operations fail as a result of the `ReleaseTheKraken` flag.

Long term, run the unit test included in Appendix D to verify that requests to update mimir values return errors when `ReleaseTheKraken` is set.

17. Unused bond reward and total reserve calculations

Severity: Informational
Type: Data Validation
Target: `thornode/x/thorchain/manager_network_v59.go`

Difficulty: N/A
Finding ID: TOB-THOR-017

Description
The `UpdateNetwork` method, responsible for network data based on block changes, uses a `totalReserve` variable to calculate block rewards. It later calculates the total rewards and updates the total reserve values; however, it does not use the result of the total rewards calculation in any subsequent calculations or to perform any node state changes. Furthermore, none of the methods that use `totalRewards` or `totalReserve` take those values by pointer reference.

```
   bondReward, totalPoolRewards, lpDeficit := vm.calcBlockRewards(totalProvidedLiquidity,
totalBonded, totalReserve, totalLiquidityFees, emissionCurve, incentiveCurve, blocksOerYear)
   // [...]
   // Move Rune from the Reserve to the Bond and Pool Rewards
   totalRewards := bondReward.Add(totalPoolRewards)
   if totalRewards.GT(totalReserve) {
       totalRewards = totalReserve
   }
   totalReserve = common.SafeSub(totalReserve, totalRewards)
   coin := common.NewCoin(common.RuneNative, bondReward)
   if !bondReward.IsZero() {
       if err := vm.k.SendFromModuleToModule(ctx, ReserveName, BondName,
common.NewCoins(coin)); err != nil {
           ctx.Logger().Error("fail to transfer funds from reserve to bond", "error", err)
           return fmt.Errorf("fail to transfer funds from reserve to bond: %w", err)
       }
   }
   network.BondRewardRune = network.BondRewardRune.Add(bondReward) // Add here for
individual Node collection later
```

Figure 17.1: Updated calculations for `totalRewards` and `totalReserve` are unused (thornode/x/thorchain/manager_network_v59.go#L650-L667).

These variables were left over from previous code, in which similar logic was performed and set using vaults instead of network objects. These variables remained when additional versions of the same function were added to the codebase.

Recommendations
Short term, remove unused and unnecessary logic from the `UpdateNetwork` method to improve the readability of the code.

18. thornode can be faulted into a state that produces no blocks

Severity: Medium
Type: Undefined Behavior
Target: `thornode`

Difficulty: High
Finding ID: TOB-THOR-018

Description
To test error handling within `thornode`, we used KRF to randomly fault system calls (syscalls) performed by the daemon. Specifically, the `read` and `write` syscalls were randomly faulted; the daemon crashed critically, requiring human intervention to restart. As a result, upon restarting, `thornode` entered a state in which it no longer produced blocks. The error displayed upon restarting the node is shown in figure 18.1.

```
Setting THORNode as genesis
1:32AM INF starting ABCI with Tendermint
1:32AM INF Starting multiAppConn service impl=multiAppConn module=proxy
1:32AM INF Starting localClient service connection=query impl=localClient module=abci-client
1:32AM INF Starting localClient service connection=snapshot impl=localClient
module=abci-client
1:32AM INF Starting localClient service connection=mempool impl=localClient
module=abci-client
1:32AM INF Starting localClient service connection=consensus impl=localClient
module=abci-client
1:32AM INF Starting EventBus service impl=EventBus module=events
1:32AM INF Starting PubSub service impl=PubSub module=pubsub
1:32AM INF Version info block=11 p2p=8 software=
1:32AM INF Starting Node service impl=Node
1:32AM INF Starting pprof server laddr=localhost:6060
1:32AM INF Starting BlockchainReactor service impl=BlockchainReactor module=blockchain
1:32AM ERR failed to process message err="error adding vote" height=293 module=consensus
msg={"Vote":{"block_id":{"hash":"92958032459C7025B4DC6A65F6130D092F23DFCDC3B3516846C64526A78
78A41","parts":{"hash":"A7BE801EC1500A4436A2046638A352D648527E47F6E3E5455668633F8BA89059","t
otal":1}},"height":293,"round":0,"signature":"2jU7yuzZeaGS9jMtFw0Tg3nSR0eVkn2gdvHKciEmol6Kdl
/o7SlLUL+UE+qIy1hctApexMJ8kczOP9dUnjyTAg==","timestamp":"2021-08-20T01:10:09.366420793Z","ty
pe":1,"validator_address":"6907DDDE95D7F06413B4B2ED0CC2EE4B7FE42281","validator_index":0}}
peer= round=0
```

Figure 18.1: `thornode` logs indicating that no blocks are being produced

While this is not directly a vulnerability in itself, the fact that system faults are not handled well enough to ensure data integrity and recovery upon failure is concerning. Snippets of specific errors that were triggered during this run are shown in figures 18.2 and 18.3.

```
12:46AM ERR WriteSync failed to flush consensus wal.
               WARNING: may result in creating alternative proposals / votes for the current
height iff the node restarted err="write /root/.thornode/data/cs.wal/wal: disk quota
exceeded" module=consensus wal=/root/.thornode/data/cs.wal/wal
12:46AM ERR CONSENSUS FAILURE!!! err="failed to write {[Proposal Proposal{237/0
```

```
(DB432D29751C812D084036B0B2E85CD99B694FF36C1E166FB406E927EA926E27:1:3558E8175073, -1)
85E206EFA4B1 @ 2021-08-20T00:46:50.35373526Z}] } msg to consensus WAL due to write
/root/.thornode/data/cs.wal/wal: disk quota exceeded; check your file system and restart the
node" module=consensus stack="goroutine 84 [running]:\nruntime/debug.Stack(0xc003491a40,
0x1cd7060, 0xc003ca9d60)\n\t/usr/local/go/src/runtime/debug/stack.go:24
github.com/tendermint/tendermint/consensus.(*State).OnStart\n\t/go/pkg/mod/github.com/tender
mint/tendermint@v0.34.8/consensus/state.go:378 +0x896\n"
...
```

Figure 18.2: An example of an error occurring when flushing consensus wal to disk

```
12:54AM INF Version info block=11 p2p=8 software=
12:54AM INF Starting Node service impl=Node
12:54AM INF Starting pprof server laddr=localhost:6060
12:54AM ERR Corrupted entry. Skipping... err="DataCorruptionError[failed to read checksum:
read /root/.thornode/data/cs.wal/wal: bad file descriptor]" module=consensus
wal=/root/.thornode/data/cs.wal/wal
12:54AM INF Starting StateSync service impl=StateSync module=statesync
12:54AM INF Starting BlockchainReactor service impl=BlockchainReactor module=blockchain
12:54AM ERR CONSENSUS FAILURE!!! err="write
/root/.thornode/data/write-file-atomic-04442787901459131641: operation not permitted"
module=consensus stack="goroutine 131 [running]:\nruntime/debug.Stack(0xc005212ff8,
0x1da2f80, 0xc0015f93b0)\n\t/usr/local/go/src/runtime/debug/stack.go:24
+0x9f\ngithub.com/tendermint/tendermint/consensus.(*State).receiveRoutine.func2(0xc0004cc700
, 0x26fbbc0)\n\t/go/pkg/mod/github.com/tendermint/tendermint@v0.34.8/consensus/state.go:726
+0x57\npanic(0x1da2f80, 0xc0015f93b0)\n\t/usr/local/go/src/runtime/panic.go:969
...
```

Figure 18.3: An example of a failing file descriptor when writing to a data file

Exploit Scenario
Alice configures a `thornode`. During normal operation, the system faults due to an external application or OS error. As a result, the node no longer produces blocks, and Alice is no longer rewarded for being a member of the network.

Recommendations
Short term, consider performing staged updates of critical data files. Because validation can occur against the staged updates and the existing critical data files before the updates are aggregated permanently, this could help prevent irrevocable faulting by corrupted data files.

Long term, ensure that all critical files are validated before the `thornode` boots. This can prevent the system from operating in an invalid state.

19. Lack of passphrase-complexity requirements for SIGNER_PASSWD

Severity: Medium
Type: Cryptography
Target: `node-launcher/scripts/core.sh`

Difficulty: Low
Finding ID: TOB-THOR-019

Description
The `node-launcher` repository contains deployment scripts for THORChain nodes. When setting up a node, a `thornode-password` is established in the deployment scripts to be later used as the `SIGNER_PASSWD` environment variable. This protects the Cosmos SDK keyring and is used to generate passphrase hashes to access keys. However, the system does not include passphrase-complexity checks. If the passphrase is weak, an attacker could discern sensitive key material.

```
create_password() {
 [ "$NET" = "testnet" ] && return
 local pwd
 local pwdconf
 if ! kubectl get -n "$NAME" secrets/thornode-password >/dev/null 2>&1; then
   echo "=> Creating THORNode Password"
   read -r -s -p "Enter password: " pwd
   echo
   read -r -s -p "Confirm password: " pwdconf
   echo
   [ "$pwd" != "$pwdconf" ] && die "Passwords mismatch"
   kubectl -n "$NAME" create secret generic thornode-password --from-literal=password="$pwd"
   echo
 fi
}
```

Figure 19.1: The `create_password` routine in the `core.sh` deployment script in `node-launcher` does not perform passphrase-complexity checks (node-launcher/scripts/core.sh#L105-119).

Exploit Scenario
Bob deploys infrastructure to host a THORChain node. When prompted to create a `thornode-password`, Bob supplies a weak, easy-to-remember passphrase. As a result, Eve, an attacker who gains access to Bob's `thornode` instance, is able to crack Bob's passphrase and compromise the underlying Cosmos SDK keyring.

Recommendations
Short term, implement passphrase-complexity requirements for the `thornode-password`, including length requirements and the mandatory use of numbers or symbols.

Long term, ensure that all cryptographic routines that derive hashes or keys from a user-supplied passphrase enforce passphrase-complexity requirements.

20. thornode allows the use of vulnerable versions of Go

Severity: Low
Type: Patching
Target: `thornode/go.mod`

Difficulty: Low
Finding ID: TOB-THOR-020

Description
The `go.mod` file in `thornode` enforces a minimum version requirement of `go 1.15`. Unfortunately, these versions of Go contain vulnerabilities that may affect the confidentiality, integrity, or availability of an application built with it.

```
module gitlab.com/thorchain/thornode

go 1.15

require (
[...]
```

Figure 20.1: The `go.mod` file in `thornode` enforces a minimum Go version of `go 1.15`, which may allow vulnerable versions of Go (`thornode/go.mod#L1-5`).

An example of a bug that could have a severe impact on networked applications is CVE-2021-31525, which enables remote attackers to cause a denial of service via a large header to `ReadRequest` or `ReadResponse` in `net/http`.

Exploit Scenario
Bob, an operator of a THORChain node, builds and deploys the application with `go 1.15`. Bob's software is now built with the vulnerabilities of `go 1.15`.

Recommendations
Short term, require the use of a newer version of Go when building a THORChain node.

Long term, ensure that all dependencies and tools used to build in THORChain are up to date. Consider integrating dependency bots into your CI/CD pipeline to expose vulnerable dependencies. For out-of-date software that is not easily discovered by static analyzers, employ operating procedures to ensure that software is kept up to date.

21. subsidizePoolWithSlashBond is prone to division-by-zero panics

Severity: Undetermined
Type: Data Validation
Target: `thornode/x/thorchain/helpers.go`

Difficulty: High
Finding ID: TOB-THOR-021

Description
When malicious actors misbehave, their collateral bond is slashed at 1.5 times the rate of any funds they have stolen. This disincentivizes misbehavior within the system. `RUNE` and other assets in an attacker's bonded vault are used to repay victims who have lost funds. However, if `subsidizePoolWithSlashBondV46` is called and the vault lists assets that have balances of zero while `RUNE` has a balance of non-zero, `yggTotalStolen` will be zero, indicating that no non-`RUNE` assets need to be used for repayment. This value will be used in a `Quo` operation, which will result in a division-by-zero panic.

```
func subsidizePoolWithSlashBondV46(ctx cosmos.Context, ygg Vault, yggTotalStolen,
slashRuneAmt cosmos.Uint, mgr Manager) error {
   // Thorchain did not slash the node account
   if slashRuneAmt.IsZero() {
       return nil
   }
   stolenRUNE := ygg.GetCoin(common.RuneAsset()).Amount
   slashRuneAmt = common.SafeSub(slashRuneAmt, stolenRUNE)
   yggTotalStolen = common.SafeSub(yggTotalStolen, stolenRUNE)
   [...]
   for _, coin := range ygg.Coins {
       asset := coin.Asset
       if coin.Asset.IsRune() {
           // when the asset is RUNE, thorchain don't need to update the RUNE balance on
pool
           continue
       }
       [...]
       // slashRune * (stealAssetRuneValue /totalStealAssetRuneValue)
       subsidizeAmt := slashRuneAmt.Mul(runeValue).Quo(yggTotalStolen)
       f.subsidiseRune = f.subsidiseRune.Add(subsidizeAmt)
       subsidizeAmounts[asset] = f
   }
```

Figure 21.1: `subsidizePoolWithSlashBond` methods perform a `Quo` operation on `yggTotalStolen`, which, if zero, can result in a panic (thornode/x/thorchain/helpers.go#L275-316).

This panic is unlikely to occur, given that a `Yggdrasil` vault listing other assets with balances of zero must contain only a non-zero `RUNE` amount when being considered for a subsidy. However, this error should be handled gracefully in the event that the division-by-zero panic becomes increasingly reachable in the future.

Recommendation
Short term, revise the `subsidizePoolWithSlashBond` methods to check whether `yggTotalStolen` is zero before entering the `for` loop. If the value is zero, the method should exit early without entering the loop.

Long term, employ property tests to uncover edge cases in data validation. Ensure that appropriate validation and error handling exists such that future changes to code do not introduce reachable vulnerabilities.

22. cluster-launcher's GCP nodes do not have automatic upgrades enabled

Severity: Informational
Type: Configuration
Target: `cluster-launcher/gcp/main.tf`

Difficulty: High
Finding ID: TOB-THOR-022

Description
The Google Cloud Platform (GCP) node pool does not define any settings for automatic node upgrades. This means that while a control plane may be updated, node infrastructure may not be.

Automatic node upgrades can be enabled by adding the following keys under the `google_container_node_pool` defined in `gcp/main.tf`:

```
management {
  # https://cloud.google.com/kubernetes-engine/docs/how-to/node-auto-upgrades
  auto_upgrade = true
}
```

Figure 22.1: The Terraform definition needed to enable automatic node upgrades in GCP nodes

This will ensure that as vulnerabilities are found in node infrastructure, nodes will be upgraded.

Recommendations
Short term, enable automatic node upgrades on GCP nodes using the Terraform definition shown in figure 22.1.

Long term, consider employing `terrascan` to uncover common flaws, new bugs, and regressions in Terraform definition files as they are introduced.

23. ADD memo with affiliate fees is incorrectly documented

Severity: Informational
Type: Undefined Behavior
Target: `thornode/x/thorchain/memo/memo_add.go`

Difficulty: Low
Finding ID: TOB-THOR-023

Description

The THORChain documentation states that `ADD` memos that include an affiliate fee should have the format `ADD:ASSET:AFFILIATE:FEE`, where each field is separated by a colon (`:`). Based on this notation, a user may assume that the third field is the affiliate address and the fourth field is the affiliate fee. However, as shown in figure 23.1, the code for parsing `ADD` memos specifies that the fourth field at index 3 is the affiliate address and the fifth field at index 4 is the fee. Users attempting to add funds with affiliate fees may submit memos in an incorrect format, leading to unexpected results.

```
func ParseAddLiquidityMemo(ctx cosmos.Context, keeper keeper.Keeper, asset common.Asset,
parts []string) (AddLiquidityMemo, error) {
   var err error
   addr := common.NoAddress
   affAddr := common.NoAddress
   affPts := cosmos.ZeroUint()
   if len(parts) >= 3 && len(parts[2]) > 0 {
       if keeper == nil {
           addr, err = common.NewAddress(parts[2])
       } else {
           addr, err = FetchAddress(ctx, keeper, parts[2], asset.Chain)
       }
       if err != nil {
           return AddLiquidityMemo{}, err
       }
   }

   if len(parts) > 4 && len(parts[3]) > 0 && len(parts[4]) > 0 {
       if keeper == nil {
           affAddr, err = common.NewAddress(parts[3])
       } else {
           affAddr, err = FetchAddress(ctx, keeper, parts[3], common.THORChain)
       }
       if err != nil {
           return AddLiquidityMemo{}, err
       }
       pts, err := strconv.ParseUint(parts[4], 10, 64)
       if err != nil {
           return AddLiquidityMemo{}, err
       }
       affPts = cosmos.NewUint(pts)
   }
```

```
    return NewAddLiquidityMemo(asset, addr, affAddr, affPts), nil
}
```

Figure 23.1: Logic for parsing ADD memos (thornode/x/thorchain/memo/memo_add.go#L29-L61)

Recommendations

Short term, correct the THORChain documentation to reflect the format expected for `ADD` memos with affiliate fees.

24. common.GetShare is prone to division-by-zero panics

Severity: Undetermined
Type: Data Validation
Target: `thornode/common/type_convert.go`

Difficulty: High
Finding ID: TOB-THOR-024

Description
The `common.GetShare` method is used when calculating shares in a total pool. To prevent division-by-zero panics, the method checks that the provided `part` and `total` fields are non-zero. However, the method performs two `Quo` operations, using the result of one operation as the denominator in the other. In the event that the inner `Quo` operation results in zero, this will result in a division-by-zero panic.

```
// GetShare this method will panic if any of the input parameter can't be convert to
cosmos.Dec
// which shouldn't happen
func GetShare(part, total, allocation cosmos.Uint) cosmos.Uint {
   if part.IsZero() || total.IsZero() {
       return cosmos.ZeroUint()
   }

   // use string to convert cosmos.Uint to cosmos.Dec is the only way I can find out without
being constrain to uint64
   // cosmos.Uint can hold values way larger than uint64 , because it is using big.Int
internally
   aD, err := cosmos.NewDecFromStr(allocation.String())
   if err != nil {
       panic(fmt.Errorf("fail to convert %s to cosmos.Dec: %w", allocation.String(), err))
   }

   pD, err := cosmos.NewDecFromStr(part.String())
   if err != nil {
       panic(fmt.Errorf("fatil to convert %s to cosmos.Dec: %w", part.String(), err))
   }
   tD, err := cosmos.NewDecFromStr(total.String())
   if err != nil {
       panic(fmt.Errorf("fail to convert%s to cosmos.Dec: %w", total.String(), err))
   }
   // A / (Total / part) == A * (part/Total) but safer when part < Totals
   result := aD.Quo(tD.Quo(pD))
   return cosmos.NewUintFromBigInt(result.RoundInt().BigInt())
}
```

Figure 24.1: `GetShare` performs a `Quo` operation inside of another `Quo` operation, which may result in a division-by-zero panic if the inner result is zero (thornode/common/type_convert.go#L17-42).

As noted in the code comments above the method in figure 24.1, this case was not considered as a potential source of a panic.

The issue was discovered using a fuzzing test written for message handlers, as validation routines in `MsgAddLiquidity`'s handler call `common.GetShare`. A unit test that can be used to verify that this issue exists can be found in Appendix D.

Exploit Scenario
Bob is a node operator in the THORChain network. Eve, an attacker, notes that `common.GetShare` is vulnerable to panics and can be leveraged to perform a denial of service attack. She discovers a call path that she can leverage to trigger a panic and take Bob's node offline.

Recommendations
Short term, ensure that the result of the inner `Quo` operation in `common.GetShare` cannot result in zero and cause a panic after the outer `Quo` computation. Ensure that all calling methods handle such errors appropriately, if they are bubbled up to the caller.

Long term, employ property tests to uncover edge cases in data validation. Ensure that appropriate validation and error handling exists such that future changes to code do not introduce reachable vulnerabilities.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing a system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Testing	Related to test methodology or test coverage
Timing	Related to race conditions, locking, or the order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is relatively small or is not a risk the customer has indicated is important.

Medium	Individual users' information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client.
High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

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

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components
Arithmetic	Related to the proper use of mathematical operations and semantics
Assembly Use	Related to the use of inline assembly
Centralization	Related to the existence of a single point of failure
Upgradeability	Related to contract upgradeability
Function Composition	Related to separation of the logic into functions with clear purposes
Front-Running	Related to resilience against front-running
Key Management	Related to the existence of proper procedures for key generation, distribution, and access
Monitoring	Related to the use of events and monitoring procedures
Specification	Related to the expected codebase documentation
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.)

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed, and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

B. Code Quality Recommendations

This appendix contains recommendations for addressing findings that do not have immediate security implications, including documentation and code quality concerns.

eth-router

- **Refactor smart contracts to remove duplicate code.** Each Solidity source file contains a flattened version of the contract source code and all contract dependencies. This results in lots of duplicate code, which negatively impacts the readability and maintainability of the codebase. For example, rather than redefining the `SafeMath` library and ERC20 interfaces in the codebase, they can simply be sourced from OpenZeppelin's libraries.

- **Refactor hard-coded function selectors.** Throughout `THORChain_Router`, low-level calls are made to external contracts (ERC20 tokens) by providing hard-coded function selector strings (e.g., #1, #2, #3). Although this is not currently problematic, hard-coded function selectors are more prone to developer error. Instead, use the `.selector` property exposed for each function definition.

thornode

- **Function `getContractABI` returns two unnamed values of the same type.** `thornode` calls the `getContractABI` function when starting an Ethereum chain client. The same function returns two contracts of type `*abi.ABI`, one for the Ethereum client vault and one for the ERC20 contract. Developers who use this function in the future may confuse the two contracts. This breaks Go's coding convention of making everything as explicit as possible. Instead, consider breaking the `getContractABI` function into two or assigning explicit names to the return values in the function signature.

- **Functions always return `nil` errors.** There are several functions that have a signature indicating that an error type could be returned. However, the function bodies are written in such a way that errors will always be `nil`. The following functions always return `nil` errors:
  - `(*Client).extractTxs`, `bifrost/pkg/chainclients/dogecoin/client.go:767`
  - `(*Client).extractTxs`, `bifrost/pkg/chainclients/bitcoincash/client.go:752`
  - `(*Client).extractTxs`, `bifrost/pkg/chainclients/litecoin/client.go:729`

    - `(*ETHScanner).extractTxs`, `bifrost/pkg/chainclients/ethereum/ethereum_block_scanner.go:388`
    - `getDOGEPrivateKey`, `bifrost/pkg/chainclients/dogecoin/signer.go:42`
    - `getBCHPrivateKey`, `bifrost/pkg/chainclients/bitcoincash/signer.go:40`
    - `getLTCPrivateKey`, `bifrost/pkg/chainclients/litecoin/signer.go:41`
    - `getBTCPrivateKey`, `bifrost/pkg/chainclients/bitcoin/signer.go:41`

- **Refactor repetitive use of integer literals throughout the codebase.** Important values are often repeated as integer literals instead of sourcing a single-source constant or variable. For example, the literal `10000` is repeated throughout the codebase as basis points to describe a full share during share calculations. In some cases, constants such as `MaxWithdrawBasisPoints` are used, albeit inconsistently.

C. Semgrep Rules

This section describes a set of Semgrep rules used by Trail of Bits during the assessment to uncover low-hanging fruit.

Insecure use of logger function

The following rule can be used to find where Tendermint's `Logger()` functions are called insecurely (TOB-THOR-015):

```
rules:
- id: insecure-logging
 pattern: <... ctx.Logger().$LEVEL(fmt.Sprintf(...)) ...>
 message: Insecure usage of logging function could lead to log injection
 languages: [go]
 severity: WARNING
```

D. Unit Tests

This section includes unit tests that verify some of the findings discussed in this report. These tests should be incorporated into the existing test suite to ensure that all findings have been addressed accordingly.

SetMimir fails silently when mimir is disabled

The following test can be used to verify that requests to update mimir values when `ReleaseTheKraken` is enabled return errors (TOB-THOR-016).

```
func (s *HandlerMimirSuite) TestSetMimir(c *C){
   ctx, keeper := setupKeeperForTest(c)

   handler := NewMimirHandler(NewDummyMgrWithKeeper(keeper))

   // Make sure things work as they should
   msg := NewMsgMimir("foo", 55, GetRandomBech32Addr())
   sdkErr := handler.handle(ctx, *msg)
   c.Assert(sdkErr, IsNil)
   val, err := keeper.GetMimir(ctx, "foo")
   c.Assert(err, IsNil)
   c.Check(val, Equals, int64(55))


   // Release the kraken (AKA disable mimir)
   krakenMsg := NewMsgMimir("ReleaseTheKraken", 1, GetRandomBech32Addr())
   krakenSdkErr := handler.handle(ctx, *krakenMsg)
   c.Assert(krakenSdkErr, IsNil)

   // Now make set a new value
   barMsg := NewMsgMimir("bar", 1984, GetRandomBech32Addr())
   barErr := handler.handle(ctx, *barMsg)
   c.Assert(barErr, NotNil) // this fails

   barVal, err := keeper.GetMimir(ctx, "bar")
   c.Assert(barErr, IsNil)
   c.Check(barVal, Equals, int64(1984)) // This fails, because we never set "bar"
succesfully
}
```

subsidizePoolWithSlashBond can lead to a division-by-zero panic

The following test exposes the division-by-zero panic that could occur when the vault passed to the `subsidizePoolWithSlashBond` function references multiple coins but only contains a non-zero balance of `RUNE`. (TOB-THOR-021)

```
func (s *HelperSuite) TestSubsidizedPoolWithRUNEOnlySlashBond(c *C) {
   ctx, mgr := setupManagerForTest(c)
   ygg := GetRandomVault()

   poolBNB := NewPool()
   poolBNB.Asset = common.BNBAsset
   poolBNB.BalanceRune = cosmos.NewUint(100 * common.One)
   poolBNB.BalanceAsset = cosmos.NewUint(100 * common.One)
   poolBNB.Status = PoolAvailable
   c.Assert(mgr.Keeper().SetPool(ctx, poolBNB), IsNil)

   ygg.Type = YggdrasilVault
   ygg.Coins = common.Coins{
       common.NewCoin(common.RuneAsset(), cosmos.NewUint(1*common.One)), // 1
       common.NewCoin(common.BNBAsset, cosmos.NewUint(0)),
   }
   totalRuneLeft, err := getTotalYggValueInRune(ctx, mgr.Keeper(), ygg)
   c.Assert(err, IsNil)

   slashAmt := totalRuneLeft.MulUint64(3).QuoUint64(2)
   c.Assert(subsidizePoolWithSlashBond(ctx, ygg, totalRuneLeft, slashAmt, mgr), IsNil)
}
```

common.GetShare is prone to division-by-zero panics

The following test exposes the division-by-zero panic that could occur when the result of the inner `Quo` computation in the `common.GetShare` function results in zero. (TOB-THOR-024)

```
package thorchain

import (
   "math/big"
   "os"

   . "gopkg.in/check.v1"

   "gitlab.com/thorchain/thornode/common"
   "gitlab.com/thorchain/thornode/common/cosmos"
)

type GetShareTestSuite struct{}

var _ = Suite(&GetShareTestSuite{})

func (s *GetShareTestSuite) SetUpSuite(c *C) {
   err := os.Setenv("NET", "other")
   c.Assert(err, IsNil)
   SetupConfigForTest()
}

func (s *GetShareTestSuite) TestAddLiquidity(c *C) {
   // Setup our manager for testing.
   SetupConfigForTest()
   ctx, mgr := setupManagerForTest(c)
   ctx = ctx.WithBlockHeight(1024)

   x := cosmos.NewUint(0)
   data := []byte {
       0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30,
0x30, 0x30,
       0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30,
0x30, 0x30,
   }
   z2 := new(big.Int)
   z2.SetBytes(data)
   y := cosmos.NewUintFromBigInt(z2)

   // Create our message and call the underlying handler.
   msg := NewMsgAddLiquidity(GetRandomTx(), common.BNBAsset, x, y, GetRandomBNBAddress(),
GetRandomBNBAddress(), GetRandomTHORAddress(), y, GetRandomBech32Addr())
   handler := NewInternalHandler(mgr)
   handler(ctx, msg)
}

func (s *GetShareTestSuite) TestGetShare(c *C) {
   x := cosmos.NewUint(0)
   data := []byte {
```

```
       0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30,
0x30, 0x30,
       0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30,
0x30, 0x30,
   }
   z2 := new(big.Int)
   z2.SetBytes(data)
   y := cosmos.NewUintFromBigInt(z2)

   common.GetShare(y, cosmos.NewUint(10000), x)
}
```

E. Distributed Fault-Injection Testing of thornode via KRF

Trail of Bits performed fault-injection testing using the KRF tool on `thornode` local nodes. KRF helps to find issues related to incorrect error handling by making syscalls used by the application return their error values and then seeing whether those errors lead to panics.

The testing setup was built with Vagrant on an Ubuntu Bionic x64 virtual machine (VM). It can be reproduced by using the Vagrantfile included in figure E.1. To run the mocknet `thornode` node, we updated the default configuration to use 8096 MB of RAM and 8 virtual CPUs. This configuration can be changed by setting the `KRF_VAGRANT_RAM` and `KRF_VAGRANT_CPUS` environment variables to the desired values before building the VM using the `vagrant up` command.

The following steps can be used to perform fault-injection testing of `thornode` nodes using KRF:

1. Clone KRF and update the Vagrantfile with the configuration shown in figure E.1.
2. Spin up the VM using `vagrant up` and connect to it via Secure Shell (SSH) using `vagrant ssh`.
3. To install KRF on the VM, follow the KRF build steps in the KRF README under "if you're using Vagrant."
4. Clone the `thornode` repository in the home directory using `git clone`: `https://gitlab.com/thorchain/thornode`.
5. Replace `thornode/build/docker/Dockerfile` with the content shown in figure E.2.
6. Update the shebang in `thornode/build/scripts/genesis.sh` and `thornode/build/scripts/bifrost.sh` to use bash with `#!/bin/bash`.
7. For compatibility with the latest version of `docker-compose`, remove all instances of `network_mode: "host"` from the following files:
    a. `thornode/build/docker/components/fullnode.Linux.yml`
    b. `thornode/build/docker/components/midgard.Linux.yml`
    c. `thornode/build/docker/components/standalone.Linux.yml`
    d. `thornode/build/docker/components/validator.Linux.yml`
8. To run the `thornode` container in privileged mode and to run the binary with KRF, update `thornode/build/docker/components/standalone.base.yml` with the changes in figure E.3.
9. To configure KRF, run the shell script in figure E.4.
10. Build the containers using `sudo BUILDTAG=mocknet BRANCH=mocknet make docker-gitlab-build`.
11. Run the containers using `sudo BLOCK_TIME=2m make -C build/docker reset-mocknet-standalone`.

12. In our tests, we occasionally needed to restart the `ethereum-localnet` container using `sudo docker restart ethereum-localnet`, followed by `sudo docker restart thornode`.
13. You can also change the faulting probability reciprocal at any time using `sudo krfctl -p 500`. For example, a faulting probability of 500 indicates a probability of 1/500, or that, on average, every 500th syscall will be faulted.
14. Inspect the `thornode` logs using `sudo docker logs --follow thornode`.
15. Faults can be stopped at any time using `sudo krfctl -c -C`.

```
Vagrant.configure("2") do |config|
 config.vm.provider :virtualbox do |vb|
   vb.memory = ENV["KRF_VAGRANT_RAM"] || 8048
   vb.cpus = ENV["KRF_VAGRANT_CPUS"] || 8
 end

 config.vm.define "linux" do |linux|
   linux.vm.box = "hashicorp/bionic64"
   linux.vm.provision :shell, inline: <<~PROVISION
     sudo apt update
     sudo DEBIAN_FRONTEND=noninteractive apt upgrade -y
     sudo DEBIAN_FRONTEND=noninteractive apt install -y libelf-dev build-essential ruby
linux-headers-$(uname -r)
     sudo apt autoremove apport apport-systems
     echo "/tmp/core_%e.krf.%p" | sudo tee /proc/sys/kernel/core_pattern
   PROVISION

   linux.vm.provider :virtualbox do |vb|
     vb.customize ["modifyvm", :id, "--uartmode1", "disconnected"]
   end
 end
end
```

Figure E.1: Vagrantfile used to set up the KRF VM

```
FROM golang:1.15.5 AS build

ENV DEBIAN_FRONTEND="noninteractive"

ENV GOBIN=/go/bin
ENV GOPATH=/go
ENV CGO_ENABLED=0
ENV GOOS=linux
ENV DEBIAN_FRONTEND=noninteractive

RUN apt update -q && apt install -y protobuf-compiler

WORKDIR /app
COPY go.mod go.sum ./
```

```
RUN go mod download

COPY . .

ARG TAG=mainnet
RUN make install

FROM ubuntu:18.04

RUN DEBIAN_FRONTEND="noninteractive" apt-get update
RUN DEBIAN_FRONTEND="noninteractive" apt-get -y install sudo git gcc musl-dev python3-dev
libffi-dev openssl libssl-dev jq curl dnsutils python3-pip autoconf automake libtool curl
make g++ unzip

RUN pip3 install requests==2.22.0 web3==5.12.0

RUN DEBIAN_FRONTEND=noninteractive apt upgrade -y
RUN DEBIAN_FRONTEND=noninteractive apt install -y kmod libelf-dev build-essential ruby
linux-headers-$(uname -r)
RUN sudo apt autoremove apport

RUN cd ~ && git clone https://github.com/trailofbits/krf
RUN cd ~/krf && make -j1 && sudo make install

COPY --from=build /go/bin/generate /usr/bin/
COPY --from=build /go/bin/thornode /usr/bin/
COPY --from=build /go/bin/bifrost /usr/bin/

COPY build/scripts /scripts
```

Figure E.2: Updated Dockerfile to run with Ubuntu instead of Alpine

```
version: '3'
services:
 # [...]
 thornode:
     # [...]
     privileged: true
     command: ["krfexec", "thornode",
"start","--log_level","info","--log_format","plain","--rpc.laddr", "tcp://0.0.0.0:26657",
"--trace"]
```

Figure E.3: Changes made to `thornode/build/docker/components/standalone.base.yml`

```
#!/bin/bash
sudo krfctl -c -C
sudo krfctl -T personality=28
sudo krfctl -p 500
sudo krfctl -L
```

```
sudo krfctl -F 'read,write'
sudo krfctl -F io
sudo krfctl -P sys
sudo krfctl -P time
```

Figure E.4: Script to configure KRF

F. Fix Log

After the assessment, THORchain informed Trail of Bits that it had addressed issues identified in the audit through various pull requests. The audit team verified each fix to ensure that it would appropriately address the corresponding issue. The results of this audit are provided in the table below.

ID	Title	Severity	Status
1	Overly permissive Access-Control-Allow-Origin headers	Medium	Not Fixed
2	Goroutine leaks in the signer process	Medium	Fixed
3	Unsanitized errors returned by thornode API	Informational	Not Fixed
4	Nil pointer dereference in internal thornode API	Informational	Fixed
5	Weak MAC construction in tss-recovery's exportKeyStore	Low	Fixed
6	Weak key derivation in thornode passphrase-based encryption routine	Medium	Fixed
7	Insufficient data validation in hierarchical deterministic key pair derivation method	Informational	Fixed
8	THORChain_Router's transferOut does not check transfer return values	High	Not Fixed
9	THORChain_Router's _routerDeposit does not check approve return values	Medium	Not Fixed
10	THORChain_Router's deposit does not check iRUNE.transferTo return values	Informational	Not Fixed
11	ETH_RUNE's transferTo function can be used to extract funds from a victim calling unrelated code	High	Not Fixed
12	Sending an off-curve point results in a segfault	High	Fixed
13	Hash-based commitment scheme is non-binding	Undetermined	Fixed
14	Undocumented API endpoints	Informational	Not Fixed
15	Insecure use of logger function could lead to log injection	Low	Fixed

16	SetMimir fails silently when mimir is disabled via ReleaseTheKraken	Medium	Not Fixed
17	Unused bond reward and total reserve calculations	Informational	Fixed
18	thornode can be faulted into a state that produces no blocks	Medium	Not Fixed
19	Lack of passphrase-complexity requirements for SIGNER_PASSWD	Medium	Not Fixed
20	thornode allows the use of vulnerable versions of Go	Low	Fixed
21	subsidizePoolWithSlashBond is prone to division-by-zero panics	Undetermined	Fixed
22	cluster-launcher's GCP nodes do not have automatic upgrades enabled	Informational	Not Fixed
23	ADD memo with affiliate fees is incorrectly documented	Informational	Not Fixed
24	common.GetShare is prone to division-by-zero panics	Undetermined	Fixed

For additional information on each fix, please refer to the detailed fix log on the following page.

Detailed Fix Log

Finding 1: Overly permissive Access-Control-Allow-Origin headers
Not fixed. While the THORChain team has not taken action on this finding, it plans to address the recommendations in the future.

Finding 2: Goroutine leaks in the signer process
Fixed. The channel that was causing the leak was converted to a buffered channel.

Finding 3: Unsanitized errors returned by thornode API
Not fixed. The THORChain team will not address this issue, positing that it would make troubleshooting more difficult to swallow the error. While this finding does not pose a security impact, sanitizing the error is still recommended as a best practice.

Finding 4: Nil pointer dereference in internal thornode API
Fixed. THORChain added a helper function, `hasNilAmountInBaseRequestValid`, which validates `Fees` and `GasPrices` and verifies that `Amount` values are not `nil`.

Finding 5: Weak MAC construction in tss-recovery's exportKeyStore
Fixed. The MPC node code now generates primes of `L/2` bits, no longer dropping one bit of entropy. The prime-generation algorithm also checks that the resulting primes are less than `2^(L/2)`.

Finding 6: Weak key derivation in thornode passphrase-based encryption routine
Fixed. The key is now derived using `scrypt.Key` rather than MD5.

Finding 7: Insufficient data validation in hierarchical deterministic key pair derivation method
Fixed. The `strconv.ParseUint` method is now used instead of `strconv.Atoi` for `string`-to-`int` conversions.

Finding 8: THORChain_Router's transferOut does not check transfer return values
Not fixed. The THORChain team has opted not to fix this issue.

Finding 9: THORChain_Router's _routerDeposit does not check approve return values
Not fixed. The THORChain team has opted not to fix this issue.

Finding 10: THORChain_Router's deposit does not check iRUNE.transferTo return values
Not fixed. The THORChain team has opted not to fix this issue.

Finding 11: ETH_RUNE's transferTo function can be used to extract funds from a victim calling unrelated code
Not fixed. The THORChain team has opted not to fix this issue.

Finding 12: Sending an off-curve point results in a segfault
Fixed. The affected functions now perform error checks after calling the relevant `crypto` library functions and before using their returned values, preventing segfault errors should an off-curve point be submitted.

Finding 13: Hash-based commitment scheme is non-binding
Fixed. The team replaced the use of safety delimiters between byte arrays when hashing with byte arrays to ensure values from the tail of an earlier byte array are not moved to the head of the next, while producing the same hash.

Finding 14: Undocumented API endpoints
Not fixed. While the THORChain team has not taken action on this finding, it has indicated a plan to address it in the future.

Finding 15: Insecure use of logger function could lead to log injection
Fixed. `fmt.Sprintf` is no longer used before passing values to `Logger` functions. Furthermore, a Semgrep rule that detects this issue was added to the CI pipeline.

Finding 16: SetMimir fails silently when mimir is disabled via ReleaseTheKraken
Not fixed. The THORChain team will not address this finding, stating that this is the behavior intended by its original design.

Finding 17: Unused bond reward and total reserve calculations
Fixed. The unused code was removed from all versions of the network manager's `UpdateNetwork` function.

Finding 18: thornode can be faulted into a state that produces no blocks
Not fixed. This finding will not be addressed by THORChain, given that this finding is the result of operating system faults and is not directly addressable in the `thornode` code. Moreover, the likelihood of this issue occurring is very low, and the errors documented indicate that the fault originates in the supporting libraries used by `thornode`.

Finding 19: Lack of passphrase complexity requirements for SIGNER_PASSWD
Not fixed. This finding has not been addressed by THORChain.

Finding 20: thornode allows the use of vulnerable versions of Go
Fixed. The `go.mod` file is now annotated with Go version 1.17. The `Dockerfile` was also updated to build `thornode` with Go 1.17.

Finding 21: subsidizePoolWithSlashBond is prone to division-by-zero panics
Fixed. The function `subsidizePoolWithSlashBondV46` now verifies that `yggTotalStolen` is not zero, preventing possible division-by-zero panics.

Finding 22: cluster-launcher's GCP nodes do not have automatic upgrades enabled
Not fixed. This finding has not been addressed by THORChain.

Finding 23: ADD memo with affiliate fees is incorrectly documented
Not fixed. This finding has not been addressed by THORChain.

Finding 24: common.GetShare is prone to division-by-zero panics
Fixed. A deferred goroutine block that calls `recover()` on panic was added to the `GetShare` function, which prevents the function from causing a panic should a division-by-zero error occur.