Wormhole Governor and Watchers

Security Assessment

April 7, 2023

Prepared for:
Wormhole Foundation

Prepared by: **Samuel Moelius and Troy Sargent**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

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

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

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

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

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

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

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

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through any source other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

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

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

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

Table of Contents

Executive Summary

Engagement Overview

Wormhole Foundation engaged Trail of Bits to review the security of the governor and watcher components of the Wormhole blockchain interoperability protocol. From February 6 to March 10, 2023, a team of two consultants conducted a security review of the client-provided source code, with eight person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

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

Summary of Findings

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

EXPOSURE ANALYSIS

Severity	*Count*
Low	4
Informational	10
Undetermined	3

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	5
Denial of Service	2
Patching	5
Testing	1
Timing	4

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager dan@trailofbits.com	**Name of PM**, Project Manager Name@trailofbits.com

The following engineers were associated with this project:

Samuel Moelius, Consultant samuel.moelius@trailofbits.com	**Troy Sargent**, Consultant troy.sargent@trailofbits.com

Project Timeline

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

Date	Event
February 2, 2023	Pre-project kickoff call
February 10, 2023	Status update meeting #1
February 17, 2023	Status update meeting #2
February 24, 2023	Status update meeting #3
March 10, 2023	Delivery of report draft
March 10, 2023	Report readout meeting
April 7, 2023	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the governor and watcher components of the Wormhole blockchain interoperability protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can the governor be used to forge messages?
- Is there any way to bypass the governor's safety checks?
- Can the governor be abused for denial of service?
- Can a watcher be used to forge messages?
- Do the watchers properly check for finality on their respective chains?
- Do the watchers properly handle/check for duplicate messages?

Project Targets

The engagement involved a review and testing of the guardian and watcher components of the Wormhole blockchain interoperability protocol.

Wormhole

Repository	https://github.com/wormhole-foundation/wormhole
Version	`3b8de17d02ca60e2cac3f4d86d3112c5f572ba96`
Type	Go
Platform	POSIX

Project Coverage

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

- **Documentation review.** We carefully reviewed the Governor whitepaper.
- **Static analysis.** We ran `golangci-lint` with option `--enable-all`, `sempgrep` with the `dgryski/semgrep-go` rules enabled, and `codeql` with our own custom rules enabled.
- **Test coverage review.** We verified that all existing unit tests pass, and we reviewed the test coverage of the governor tests and the `evm` finalizer tests.
- **Fuzzing.** We developed fuzzers for `ProcessMsg` and for the processing of `queryCoinGecko` responses.
- **Manual review.** We reviewed the source code for the governor, the `evm` watchers, and the Solana, Cosmwasm, and Algorand watchers.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Non-`evm` watchers other than Solana, Cosmwasm, and Algorand were largely uncovered. More specifically, with respect to the following watchers, only static analysis tools were applied, and their results were only cursorily examined:
  - Near
  - Sui
  - Aptos
- Due to time constraints, we did not perform fuzzing of Solana, Cosmwasm, or Algorand watcher code.
- We did not review the node's interaction with the governor and watchers via CLI or REST API.

Automated Testing

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

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

- `Golangci-lint`: A fast Go linters runner that includes dozens of linters, including `go vet`, `gosec`, `errcheck`, and `ineffassign`
- Semgrep: An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time
- CodeQL: A code analysis engine developed by GitHub to automate security checks

Fuzzing

We developed fuzzers for the targets listed below. Fuzzing `ProcessMsg` did not reveal any findings. Fuzzing the processing of `queryCoinGecko` responses resulted in one low-severity finding involving an unchecked type assertion.

Fuzz Target	Resulting Finding(s)
`ProcessMsg`	None
Processing of `queryCoinGecko` responses	TOB-WORMGUWA-4

Codebase Maturity Evaluation

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

Category	Summary	Result
Arithmetic	We found no issues involving arithmetic.	Satisfactory
Auditing	Wormhole makes use of verbose logging for critical actions like startup and receiving/sending messages. This provides insight into potential issues and creates an audit log should there be a need to investigate unexpected behavior.	Strong
Authentication / Access Controls	We found no issues involving access controls.	Satisfactory
Complexity Management	Wormhole supports a larger number of chains and configures settings for each within a function that is difficult to review (TOB-WORMGUWA-9).	Weak
Cryptography and Key Management	We found no issues related to cryptography or key management.	Satisfactory
Decentralization	The governor relies on CoinGecko as the sole source of truth for asset prices (TOB-WORMGUWA-5).	Moderate
Documentation	Function comments are sparse (TOB-WORMGUWA-1, TOB-WORMGUWA-2), and important assumptions about the data returned from external sources are not documented (see the long-term recommendation for TOB-WORMGUWA-8). Although there is some high-level documentation for the governor, the configuration and	Weak

	idiosyncrasies of each chain's finality and transaction validation should be documented. Appendix D gives an example of how this could be done.	
Front-Running Resistance	Front-running was not a factor for the code under review.	**Not Applicable**
Low-Level Manipulation	Several of this report's findings involved mishandling of channels (TOB-WORMGUWA-7, TOB-WORMGUWA-8, TOB-WORMGUWA-11) or contexts (TOB-WORMGUWA-6, TOB-WORMGUWA-15, TOB-WORMGUWA-16).	**Moderate**
Testing and Verification	Wormhole relies primarily on integration tests that make it difficult to identify what is tested (TOB-WORMGUWA-12). Unit test coverage should be increased to cover as much of the program as possible. In addition, fuzz testing should be adopted, especially for functions that handle untrusted data (TOB-WORMGUWA-4).	**Weak**

Summary of Findings

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

ID	Title	Type	Severity
1	Lack of doc comments	Patching	Informational
2	Fields protected by mutex are not documented	Patching	Informational
3	Potential nil pointer dereference in reloadPendingTransfer	Data Validation	Low
4	Unchecked type assertion in queryCoinGecko	Data Validation	Low
5	Governor relies on a single external source of truth for asset prices	Data Validation	Informational
6	Potential resource leak	Denial of Service	Informational
7	PolygonConnector does not properly use channels	Timing	Undetermined
8	Receiver closes channel, contradicting Golang guidance	Timing	Undetermined
9	Watcher configuration is overly complex	Data Validation	Informational
10	evm.Watcher.Run’s default behavior could hide bugs	Patching	Informational
11	Race condition in TestBlockPoller	Timing	Informational
12	Unconventional test structure	Testing	Informational

13	Vulnerable Go packages	Patching	**Undetermined**
14	Wormhole node does not build with latest Go version	Patching	**Informational**
15	Missing or wrong context	Timing	**Low**
16	Use of defer in a loop	Denial of Service	**Low**
17	Finalizer is allowed to be nil	Data Validation	**Informational**

Detailed Findings

1. Lack of doc comments	
Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-WORMGUWA-1
Target: `node/pkg/governor/governor.go`, various source files in `node/pkg/watchers`	

Description

Publicly accessible functions within the governor and watcher code generally lack doc comments. Inadequately documented code can be misunderstood, which increases the likelihood of an improper bug fix or a mis-implemented feature.

There are ten publicly accessible functions within `governor.go`. However, only one such function has a comment preceding it (see figure 1.1).

```
// Returns true if the message can be published, false if it has been added to the
pending list.
func (gov *ChainGovernor) ProcessMsg(msg *common.MessagePublication) bool {
```

Figure 1.1: `node/pkg/governor/governor.go#L281–L282`

Similarly, there are at least 28 publicly accessible functions among the non-`evm` watchers. However, only seven of them are preceded by doc comments, and only one of the seven is not in the Near watcher code (see figure 1.2).

```
// GetLatestFinalizedBlockNumber() returns the latest published block.
func (s *SolanaWatcher) GetLatestFinalizedBlockNumber() uint64 {
```

Figure 1.2: `node/pkg/watchers/solana/client.go#L846–L847`

Go's official documentation on doc comments states the following:

> *A func's doc comment should explain what the function returns or, for functions called for side effects, what it does.*

Exploit Scenario

Alice, a Wormhole developer, implements a new node feature involving the governor. Alice misunderstands how the functions called by her new feature work. Alice introduces a vulnerability into the node as a result.

Recommendations

Short term, add doc comments to each function that are accessible from outside of the package in which the function is defined. This will facilitate code review and reduce the likelihood that a developer introduces a bug into the code because of a misunderstanding.

Long term, regularly review code comments to ensure they are accurate. Documentation must be kept up to date to be beneficial.

References

- Go Doc Comments

2. Fields protected by mutex are not documented

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-WORMGUWA-2
Target: `node/pkg/governor/governor.go`	

Description

The fields protected by the governor's mutex are not documented. A developer adding functionality to the governor is unlikely to know whether the mutex must be locked for their application.

The `ChainGovernor` struct appears in figure 2.1. The Wormhole Foundation communicated to us privately that the mutex protects the fields highlighted in yellow. Note that, because there are 13 fields in `ChainGovernor` (not counting the mutex itself), the likelihood of a developer guessing exactly the set of highlighted fields is small.

```
type ChainGovernor struct {
        db                    db.GovernorDB
        logger                *zap.Logger
        mutex                 sync.Mutex
        tokens                map[tokenKey]*tokenEntry
        tokensByCoinGeckoId   map[string][]*tokenEntry
        chains                map[vaa.ChainID]*chainEntry
        msgsSeen              map[string]bool // Key is hash, payload is consts
transferComplete and transferEnqueued.
        msgsToPublish         []*common.MessagePublication
        dayLengthInMinutes    int
        coinGeckoQuery        string
        env                   int
        nextStatusPublishTime time.Time
        nextConfigPublishTime time.Time
        statusPublishCounter  int64
        configPublishCounter  int64
}
```

Figure 2.1: `node/pkg/governor/governor.go#L119–L135`

Exploit Scenario

Alice, a Wormhole developer, adds a new function to the governor.

- Case 1: Alice does not lock the mutex, believing that her function operates only on fields that are not protected by the mutex. However, by not locking the mutex, Alice introduces a race condition into the governor.

- Case 2: Alice locks the mutex "just to be safe." However, the fields on which Alice's function operates are not protected by the mutex. Alice introduces a deadlock into the code as a result.

Recommendations

Short term, document the fields within `ChainGovernor` that are protected by the mutex. This will reduce the likelihood that a developer incorrectly locks, or does not lock, the mutex.

Long term, regularly review code comments to ensure they are accurate. Documentation must be kept up to date to be beneficial.

3. Potential nil pointer dereference in reloadPendingTransfer

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-WORMGUWA-3
Target: `node/pkg/governor/governor_db.go`	

Description

A potential `nil` pointer dereference exists in `reloadPendingTransfer`. The bug could be triggered if invalid data were stored within a node's database, and could make it impossible to restart the node.

The relevant code appears in figures 3.1 and 3.2 When `DecodeTransferPayloadHdr` returns an error, the payload that is also returned is used to construct the error message (figure 3.1). However, as shown in figure 3.2, the returned payload can be `nil`.

```
payload, err := vaa.DecodeTransferPayloadHdr(msg.Payload)
if err != nil {
	gov.logger.Error("cgov: failed to parse payload for reloaded pending transfer, 
dropping it",
		zap.String("MsgID", msg.MessageIDString()),
		zap.Stringer("TxHash", msg.TxHash),
		zap.Stringer("Timestamp", msg.Timestamp),
		zap.Uint32("Nonce", msg.Nonce),
		zap.Uint64("Sequence", msg.Sequence),
		zap.Uint8("ConsistencyLevel", msg.ConsistencyLevel),
		zap.Stringer("EmitterChain", msg.EmitterChain),
		zap.Stringer("EmitterAddress", msg.EmitterAddress),
		zap.Stringer("tokenChain", payload.OriginChain),
		zap.Stringer("tokenAddress", payload.OriginAddress),
		zap.Error(err),
	)
	return
}
```

Figure 3.1: *`node/pkg/governor/governor_db.go#L90–L106`*

```
func DecodeTransferPayloadHdr(payload []byte) (*TransferPayloadHdr, error) {
	if !IsTransfer(payload) {
		return nil, fmt.Errorf("unsupported payload type")
	}
```

Figure 3.2: *`sdk/vaa/structs.go#L962–L965`*

Exploit Scenario

Eve finds a code path that allows her to store erroneous payloads within the database of Alice's node. Alice is unable to restart her node, as it tries to dereference a `nil` pointer on each attempt.

Recommendations

Short term, either eliminate the use of payload when constructing the error message, or verify that the payload is not `nil` before attempting to dereference it. This will eliminate a potential `nil` pointer dereference.

Long term, add tests to exercise additional error paths within `governor_db.go`. This could help to expose bugs like this one.

4. Unchecked type assertion in queryCoinGecko

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-WORMGUWA-4
Target: `node/pkg/governor/governor_prices.go`	

Description

The code that processes CoinGecko responses contains an unchecked type assertion. The bug is triggered when CoinGecko returns invalid data, and could be exploited for denial of service (DoS).

The relevant code appears in figure 4.1. The `data` object that is returned as part of CoinGecko's response to a query is cast to a map `m` of type `map[string]interface{}` (yellow). However, the cast's success is not verified. As a result, a `nil` pointer dereference can occur when `m` is accessed (red).

```
m := data.(map[string]interface{})
if len(m) != 0 {
        var ok bool
        price, ok = m["usd"].(float64)
        if !ok {
                gov.logger.Error("cgov: failed to parse coin gecko response, reverting
to configured price for this token", zap.String("coinGeckoId", coinGeckoId))
                // By continuing, we leave this one in the local map so the price will
get reverted below.
                continue
        }
}
```

Figure 4.1: `node/pkg/governor/governor_prices.go#L144–L153`

Note that if the access to `m` is successful, the resulting value is cast to a `float64`. In this case, the cast's success is verified. A similar check should be performed for the earlier cast.

Exploit Scenario

Eve, a malicious insider at CoinGecko, sends invalid data to Wormhole nodes, causing them to crash.

Recommendations

Short term, in the code in figure 4.1, verify that the cast in yellow is successful by adding a check similar to the one highlighted in green. This will eliminate the possibility of a node crashing because CoinGecko returns invalid data.

Long term, consider enabling the `forcetypeassert` lint in CI. This bug was initially flagged by that lint, and then confirmed by our `queryCoinGecko` response fuzzer. Enabling the lint could help to expose additional bugs like this one.

5. Governor relies on a single external source of truth for asset prices

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-WORMGUWA-5
Target: `node/pkg/governor/governor_prices.go`	

Description

The governor relies on a single external source (CoinGecko) for asset prices, which could enable an attacker to transfer more than they would otherwise be allowed.

The governor fetches an asset's price from CoinGecko, compares the price to a hard-coded default, and uses whichever is larger (figure 5.1). However, if an asset's price were to grow much larger than the hard-coded default, the hard-coded default would essentially be meaningless, and CoinGecko would become the sole source of truth for the price of that asset. Such a situation could be problematic, for example, if the asset's price were volatile and CoinGecko had trouble keeping up with the price changes.

```
// We should use the max(coinGeckoPrice, configuredPrice) as our price for computing
notional value.
func (te tokenEntry) updatePrice() {
	if (te.coinGeckoPrice == nil) || (te.coinGeckoPrice.Cmp(te.cfgPrice) < 0) {
		te.price.Set(te.cfgPrice)
	} else {
		te.price.Set(te.coinGeckoPrice)
	}
}
```

Figure 5.1: `node/pkg/governor/governor_prices.go#L205–L212`

Exploit Scenario

Eve obtains a large quantity of AliceCoin from a hack. AliceCoin's price is both highly volatile and much larger than what was hard-coded in the last Wormhole release. CoinGecko has trouble keeping up with the current price of AliceCoin. Eve identifies a point in time when the price that CoinGecko reports is low (but still higher than the hard-coded default). Eve uses the opportunity to move more of her maliciously obtained AliceCoin than Wormhole would allow if CoinGecko had reported the correct price.

Recommendations

Short term, monitor the price of assets supported by Wormhole. If the price of an asset increases substantially, consider issuing a release that takes into account the new price.

This will help to avoid situations where CoinGecko becomes the sole source of truth of the price of an asset.

Long term, incorporate additional price oracles besides CoinGecko. This will provide more robust protection than requiring a human to monitor prices and issue point releases.

6. Potential resource leak

Severity: **Informational**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-WORMGUWA-6
Target: `node/pkg/watchers/evm/watcher.go`	

Description

Calls to some `Contexts`' `cancel` functions are missing along certain code paths involving panics. If an attacker were able to exercise these code paths in rapid succession, they could exhaust system resources and cause a DoS.

Within `watcher.go`, `WithTimeout` is essentially used in one of two ways, using the pattern shown in either figure 6.1 or 6.2. The pattern of figure 6.1 is problematic because `cancel` will not be called if a panic occurs in `MessageEventsForTransaction`. By comparison, `cancel` will be called if a panic occurs after the `defer` statement in figure 6.2. Note that if a panic occurred in either figure 6.1 or 6.2, `RunWithScissors` (figure 6.3) would prevent the program from terminating.

```
timeout, cancel := context.WithTimeout(ctx, 5*time.Second)
blockNumber, msgs, err := MessageEventsForTransaction(timeout, w.ethConn,
w.contract, w.chainID, tx)
cancel()
```

Figure 6.1: `node/pkg/watchers/evm/watcher.go#L395–L397`

```
timeout, cancel := context.WithTimeout(ctx, 15*time.Second)
defer cancel()
```

Figure 6.2: `node/pkg/watchers/evm/watcher.go#L186–L187`

```
// Start a go routine with recovering from any panic by sending an error to a error
channel
func RunWithScissors(ctx context.Context, errC chan error, name string, runnable
supervisor.Runnable) {
	ScissorsErrors.WithLabelValues("scissors", name).Add(0)
	go func() {
		defer func() {
			if r := recover(); r != nil {
				switch x := r.(type) {
				case error:
					errC <- fmt.Errorf("%s: %w", name, x)
				default:
					errC <- fmt.Errorf("%s: %v", name, x)
```

```
                }
                ScissorsErrors.WithLabelValues("scissors", name).Inc()

            }
        }()
        err := runnable(ctx)
        if err != nil {
            errC <- err
        }
    }()
}
```

Figure 6.3: `node/pkg/common/scissors.go#L20–L41`

Golang's official `Context` documentation states:

> *The `WithCancel`, `WithDeadline`, and `WithTimeout` functions take a `Context` (the parent) and return a derived `Context` (the child) and a `CancelFunc`. … Failing to call the `CancelFunc` leaks the child and its children until the parent is canceled or the timer fires. …*

In light of the above guidance, it seems prudent to call the `cancel` function, even along panicking paths.

Note that the problem described applies to three locations in `watch.go`: one involving a call to `MessageEventsForTransaction` (figure 6.1), one involving a call to `TimeOfBlockByHash`, and one involving a call to `TransactionReceipt`.

Exploit Scenario

Eve discovers a code path she can call in rapid succession, which induces a panic in the call to `MessageEventsForTransaction` (figure 6.1). Eve exploits this code path to crash Wormhole nodes.

Recommendations

Short term, use the `defer cancel()` pattern (figure 6.2) wherever `WithTimeout` is used. This will help to prevent DoS conditions.

Long term, regard all code involving `Contexts` with heightened scrutiny. `Contexts` are frequently a source of resource leaks in Go programs, and deserve elevated attention.

References

- Golang Context WithTimeout Example

7. PolygonConnector does not properly use channels

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Timing	Finding ID: TOB-WORMGUWA-7
Target: `node/pkg/watchers/evm/connectors/polygon.go`	

Description

The Polygon connector does not read from the `PollSubscription.quit` channel, nor does it write to the `PollSubscription.unsubDone` channel. A caller who calls `Unsubscribe` on the `PollSubscription` could hang.

A `PollSubscription` struct contains three channels: `err`, `quit`, and `unsubDone` (figure 7.1). Based on our understanding of the code, the entity that fulfills the subscription writes to the `err` and `unsubDone` channels, and reads from the `quit` channel. Conversely, the entity that consumes the subscription reads from the `err` and `unsubDone` channels, and writes to the `quit` channel.[1]

```
type PollSubscription struct {
	errOnce   sync.Once
	err       chan error
	quit      chan error
	unsubDone chan struct{}
}
```

Figure 7.1: `node/pkg/watchers/evm/connectors/common.go#L38–L43`

More specifically, the consumer can call `PollSubscription.Unsubscribe`, which writes `ErrUnsubscribed` to the `quit` channel and waits for a message on the `unsubDone` channel (figure 7.2).

```
func (sub *PollSubscription) Unsubscribe() {
	sub.errOnce.Do(func() {
		select {
		case sub.quit <- ErrUnsubscribed:
			<-sub.unsubDone
		case <-sub.unsubDone:
		}
		close(sub.err)
	})
}
```

Figure 7.2: `node/pkg/watchers/evm/connectors/common.go#L59–L68`

[1] If our understanding is correct, we recommend documenting these facts.

However, the Polygon connector does not read from the `quit` channel, nor does it write to the `unsubDone` channel (figure 7.3). This is unlike `BlockPollConnector` (figure 7.4), for example. Thus, if a caller tries to call `Unsubscribe` on the Polygon connector `PollSubscription`, the caller may hang.

```
select {
case <-ctx.Done():
	return nil
case err := <-messageSub.Err():
	sub.err <- err
case checkpoint := <-messageC:
	if err := c.processCheckpoint(ctx, sink, checkpoint); err != nil {
		sub.err <- fmt.Errorf("failed to process checkpoint: %w", err)
	}
}
```

Figure 7.3: `node/pkg/watchers/evm/connectors/polygon.go#L120–L129`

```
select {
case <-ctx.Done():
	blockSub.Unsubscribe()
	innerErrSub.Unsubscribe()
	return nil
case <-sub.quit:
	blockSub.Unsubscribe()
	innerErrSub.Unsubscribe()
	sub.unsubDone <- struct{}{}
	return nil
case v := <-innerErrSink:
	sub.err <- fmt.Errorf(v)
}
```

Figure 7.4: `node/pkg/watchers/evm/connectors/poller.go#L180–L192`

Exploit Scenario

Alice, a Wormhole developer, adds a code path that involves calling `Unsubscribe` on a Polygon connector's `PollSubscription`. By doing so, Alice introduces a deadlock into the code.

Recommendations

Short term, adjust the code in figure 7.3 so that it reads from the `quit` channel and writes to the `unsubDone` channel, similar to how the code in figure 7.4 does. This will eliminate a class of code paths along which hangs or deadlocks could occur.

Long term, consider refactoring the code so that the `select` statements in figures 7.3 and 7.4, as well as a similar statement in `LogPollConnector`, are consolidated under a single function. The three statements appear similar in their behavior; combining them would make the code more robust against future changes and could help to prevent bugs like this one.

8. Receiver closes channel, contradicting Golang guidance

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Timing	Finding ID: TOB-WORMGUWA-8
Target: `node/pkg/watchers/evm/connectors/common.go`	

Description

According to Golang's official guidance, "Only the sender should close a channel, never the receiver. Sending on a closed channel will cause a panic." However, along some code paths within the watcher code, the receiver of a channel closes the channel.

When `PollSubscription.Unscbscribe` is called, it closes the `err` channel (figure 8.1). However, in `logpoller.go` (figure 8.2), the caller of `Unsubscribe` (red) is clearly an `err` channel receiver (green).

```
func (sub *PollSubscription) Err() <-chan error {
	return sub.err
}

func (sub *PollSubscription) Unsubscribe() {
	sub.errOnce.Do(func() {
		select {
		case sub.quit <- ErrUnsubscribed:
			<-sub.unsubDone
		case <-sub.unsubDone:
		}
		close(sub.err)
	})
}
```

Figure 8.1: `node/pkg/watchers/evm/connectors/common.go#L55–L68`

```
sub, err := l.SubscribeForBlocks(ctx, errC, blockChan)
if err != nil {
	return err
}
defer sub.Unsubscribe()

supervisor.Signal(ctx, supervisor.SignalHealthy)
for {
	select {
	case <-ctx.Done():
		return ctx.Err()
	case err := <-sub.Err():
```

```
                return err
        case err := <-errC:
                return err
        case block := <-blockChan:
                if err := l.processBlock(ctx, logger, block); err != nil {
                        l.errFeed.Send(err.Error())
                }
        }
}
```

Figure 8.2: `node/pkg/watchers/evm/connectors/logpoller.go#L49–L69`

Exploit Scenario

Eve discovers a code path along which a sender tries to send to an already closed `err` channel and panics. `RunWithScissors` (see TOB-WORMGUWA-6) prevents the node from terminating, but the node is left in an undetermined state.

Recommendations

Short term, eliminate the call to `close` in figure 8.1. This will eliminate a class of code paths along which the `err` channel's sender(s) could panic.

Long term, for each channel, document who the expected senders and receivers are. This will help catch bugs like this one.

References

- A Tour of Go: Range and Close

9. Watcher configuration is overly complex

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-WORMGUWA-9
Target: `node/pkg/watchers/evm/watcher.go`	

Description

The Run function of the `Watcher` configures each chain's connection based on its fields, `unsafeDevMode` and `chainID`. This is done in a series of nested if-statements that span over 100 lines, amounting to a cyclomatic complexity over 90 which far exceeds what is considered complex. In order to make the code easier to understand, test, and maintain, it should be refactored. Rather than handling all of the business logic in a monolithic function, the logic for each chain should be isolated within a dedicated helper function. This would make the code easier to follow and reduce the likelihood that an update to one chain's configuration inadvertently introduces a bug for other chains.

```
if w.chainID == vaa.ChainIDCelo && !w.unsafeDevMode {
	// When we are running in mainnet or testnet, we need to use the Celo ethereum library rather than go-ethereum.
	// However, in devnet, we currently run the standard ETH node for Celo, so we need to use the standard go-ethereum.
	w.ethConn, err = connectors.NewCeloConnector(timeout, w.networkName, w.url, w.contract, logger)
	if err != nil {
		ethConnectionErrors.WithLabelValues(w.networkName, "dial_error").Inc()
		p2p.DefaultRegistry.AddErrorCount(w.chainID, 1)
		return fmt.Errorf("dialing eth client failed: %w", err)
	}
} else if useFinalizedBlocks {
	if w.chainID == vaa.ChainIDEthereum && !w.unsafeDevMode {
		safeBlocksSupported = true
		logger.Info("using finalized blocks, will publish safe blocks")
	} else {
		logger.Info("using finalized blocks")
	}

[...] /* many more nested branches */
```

Figure 9.1: `node/pkg/watchers/evm/watcher.go#L192–L326`

Exploit Scenario

Alice, a wormhole developer, introduces a bug that causes guardians to run in unsafe mode in production while adding support for a new evm chain due to the difficulty of modifying and testing the nested code.

Recommendations

Short term, isolate each chain’s configuration into a helper function and document how the configurations were determined.

Long term, run linters in CI to identify code with high cyclomatic complexity and consider whether complex code can be simplified during code reviews.

10. evm.Watcher.Run's default behavior could hide bugs

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-WORMGUWA-10
Target: `node/{cmd/guardiand/node.go, pkg/watchers/evm/watcher.go}`	

Description

`evm.Watcher.Run` tries to create an `evm` watcher, even if called with a `ChainID` that does not correspond to an `evm` chain. Additional checks should be added to `evm.Watcher.Run` to reject such `ChainID`s.

Approximately 60 watchers are started in `node/cmd/guardiand/node.go` (figure 10.1). Fifteen of those starts result in calls to `evm.Watcher.Run`. Given the substantial number of `ChainID`s, one can imagine a bug where a developer tries to create an `evm` watcher with a `ChainID` that is not for an `evm` chain. Such a `ChainID` would be handled by the blanket `else` in figure 10.2, which tries to create an `evm` watcher. Such behavior could allow the bug to go unnoticed. To avoid this possibility, `evm.Watcher.Run`'s default behavior should be to fail rather than to create a watcher.

```
if shouldStart(ethRPC) {
	...
	ethWatcher = evm.NewEthWatcher(*ethRPC, ethContractAddr, "eth",
common.ReadinessEthSyncing, vaa.ChainIDEthereum, chainMsgC[vaa.ChainIDEthereum],
setWriteC, chainObsvReqC[vaa.ChainIDEthereum], *unsafeDevMode)
	...
}

if shouldStart(bscRPC) {
	...
	bscWatcher := evm.NewEthWatcher(*bscRPC, bscContractAddr, "bsc",
common.ReadinessBSCSyncing, vaa.ChainIDBSC, chainMsgC[vaa.ChainIDBSC], nil,
chainObsvReqC[vaa.ChainIDBSC], *unsafeDevMode)
	...
}

if shouldStart(polygonRPC) {
	...
	polygonWatcher := evm.NewEthWatcher(*polygonRPC, polygonContractAddr,
"polygon", common.ReadinessPolygonSyncing, vaa.ChainIDPolygon,
chainMsgC[vaa.ChainIDPolygon], nil, chainObsvReqC[vaa.ChainIDPolygon],
*unsafeDevMode)
}
...
```

Figure 10.1: node/cmd/guardiand/node.go#L1065–L1104

```
...
} else if w.chainID == vaa.ChainIDOptimism && !w.unsafeDevMode {
        ...
} else if w.chainID == vaa.ChainIDPolygon && w.usePolygonCheckpointing() {
        ...
} else {
        w.ethConn, err = connectors.NewEthereumConnector(timeout, w.networkName,
w.url, w.contract, logger)
        if err != nil {
                ethConnectionErrors.WithLabelValues(w.networkName, "dial_error").Inc()
                p2p.DefaultRegistry.AddErrorCount(w.chainID, 1)
                return fmt.Errorf("dialing eth client failed: %w", err)
        }
}
```

Figure 10.2: node/pkg/watchers/evm/watcher.go#L192–L326

Exploit Scenario

Alice, a Wormhole developer, introduces a call to `NewEvmWatcher` with a `ChainID` that is not for an `evm` chain. `evm.Watcher.Run` accepts the invalid `ChainID`, and the error goes unnoticed.

Recommendations

Short term, rewrite `evm.Watcher.Run` so that a new watcher is created only when a `ChainID` for an `evm` chain is passed. When a `ChainID` for some other chain is passed, `evm.Watcher.Run` should return an error. Adopting such a strategy will help protect against bugs in `node/cmd/guardiand/node.go`.

Long term:

- Add tests to the `guardiand` package to verify that the right watcher is created for each `ChainID`. This will help ensure the package's correctness.
- Consider whether TOB-WORMGUWA-9's recommendations should also apply to `node/cmd/guardiand/node.go`. That is, consider whether the watcher configuration should be handled in `node/cmd/guardiand/node.go`, as opposed to `evm.Watcher.Run`. The file `node/cmd/guardiand/node.go` appears to suffer from similar complexity issues. It is possible that a single strategy could address the shortcomings of both pieces of code.

11. Race condition in TestBlockPoller

Severity: **Informational**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-WORMGUWA-11
Target: `node/pkg/watchers/evm/connectors/poller_test.go`	

Description

A race condition causes `TestBlockPoller` to fail sporadically with the error message in figure 11.1. For a test to be of value, it must be reliable.

```
poller_test.go:300:
            Error Trace:    .../node/pkg/watchers/evm/connectors/poller_test.go:300
            Error:          Received unexpected error:
                            polling encountered an error: failed to look up latest
block: RPC failed
            Test:           TestBlockPoller
```

Figure 11.1: Error produced when TestBlockPoller fails

A potential code interleaving causing the above error appears in figure 11.2. The interleaving can be explained as follows:

- The main thread sets the `baseConnector`'s error and yields (left column).
- The go routine declared at `poller_test.go:189` retrieves the error, sets the `err` variable, loops, retrieves the error a second time, and yields (right column).
- The main thread locks the mutex, verifies that `err` is set, clears `err`, and unlocks the mutex (left).
- The go routine sets the `err` variable a second time (right).
- The main thread locks the mutex and panics because `err` is set (left).

`baseConnector.setError(fmt.Errorf("RPC failed"))`	

	`case thisErr :=`
	`<-headerSubscription.Err():`
	`      mutex.Lock()`
	`      err = thisErr`
	`      mutex.Unlock()`
	`...`
	`case thisErr :=`
	`<-headerSubscription.Err():`

`time.Sleep(10 * time.Millisecond)`	

`mutex.Lock()`
`require.Equal(t, 1, pollerStatus)`
`assert.Error(t, err)`
`assert.Nil(t, block)`
`baseConnector.setError(nil)`
`err = nil`
`mutex.Unlock()`	

	`mutex.Lock()`
	`err = thisErr`
	`mutex.Unlock()`

`// Post the next block and verify we get it (so we survived the RPC error).`
`baseConnector.setBlockNumber(0x309a10)`
 	
`time.Sleep(10 * time.Millisecond)`
`mutex.Lock()`
`require.Equal(t, 1, pollerStatus)`
`require.NoError(t, err)`	

Figure 11.2: Interleaving of `node/pkg/watchers/evm/connectors/poller_test.go#L283–L300` *(left) and* `node/pkg/watchers/evm/connectors/poller_test.go#L198–L201` *(right) that causes an error*

Exploit Scenario

Alice, a Wormhole developer, ignores `TestBlockPoller` failures because she believes the test to be flaky. In reality, the test is flagging a bug in Alice's code, which she commits to the Wormhole repository.

Recommendations

Short term:

- Use different synchronization mechanisms in order to eliminate the race condition described above. This will increase `TestBlockPoller`'s reliability.
- Have the main thread sleep for random rather than fixed intervals. This will help to expose bugs like this one.

Long term, investigate automated tools for finding concurrency bugs in Go programs. This bug is not flagged by Go's race detector. As a result, different analyses are needed.

12. Unconventional test structure	
Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-WORMGUWA-12
Target: Various source files in `node/pkg/watchers`	

Description

Tilt is the primary means of testing Wormhole watchers. Relying on such a coarse testing mechanism makes it difficult to know that all necessary conditions and edge cases are tested.

The following are some conditions that should be checked by native Go unit tests:

- The right watcher is created for each `ChainID` (TOB-WORMGUWA-10).
- The `evm` watchers' connectors behave correctly (similar to how the `evm` finalizers' correct behavior is now tested).[2]
- The `evm` watchers' `logpoller` behaves correctly (similar to how the `poller`'s correct behavior is now tested by `poller_test.go`).
- There are no off-by-one errors in any inequality involving a block or round number. Examples of such inequalities include the following:
  - `node/pkg/watchers/algorand/watcher.go#L225`
  - `node/pkg/watchers/algorand/watcher.go#L243`
  - `node/pkg/watchers/solana/client.go#L363`
  - `node/pkg/watchers/solana/client.go#L841`

To be clear, we are not suggesting that the Tilt tests be discarded. However, the Tilt tests should not be the sole means of testing the watchers for any given chain.

Exploit Scenario

Alice, a Wormhole developer, introduces a bug into the codebase. The bug is not exposed by the Tilt tests.

Recommendations

Short term, develop unit tests for the watcher code. Get as close to 100% code coverage as possible. Develop specific unit tests for conditions that seem especially problematic. These steps will help ensure the correctness of the watcher code.

[2] Note that the `evm` watchers' finalizers have nearly 100% code coverage by unit tests.

Long term, regularly review test coverage to help identify gaps in the tests as the code evolves.

13. Vulnerable Go packages	
Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Patching	Finding ID: TOB-WORMGUWA-13
Target: `node/go.mod`	

Description

`govulncheck` reports that the packages used by Wormhole in table 13.1 have known vulnerabilities, which are described in the following table.

Package	Vulnerability	Description excerpt
`path/filepath`	`GO-2023-1568`	A path traversal vulnerability exists in filepath.Clean on Windows. ...
`mime/multipart`	`GO-2023-1569`	A denial of service is possible from excessive resource consumption in net/http and mime/multipart. ...
`crypto/tls`	`GO-2023-1570`	Large handshake records may cause panics in crypto/tls. ...
`golang.org/x/net`	`GO-2023-1571`	A maliciously crafted HTTP/2 stream could cause excessive CPU consumption in the HPACK decoder, sufficient to cause a denial of service from a small number of small requests.

Table 13.1: Vulnerabilities in dependencies reported by `govulncheck`

Exploit Scenario

Eve discovers an exploitable code path involving one of the vulnerabilities in table 13.1 and uses it to crash Wormhole nodes.

Recommendations

Short term, update Wormhole to Go version 1.20.1. This will mitigate all of the vulnerabilities in table 13.1, according to the vulnerability descriptions.

Long term, run `govulncheck` as part of Wormhole’s CI process. This will help to identify vulnerable dependencies as they arise.

References

- Vulnerability Management for Go

14. Wormhole node does not build with latest Go version

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-WORMGUWA-14
Target: Various source files	

Description

Attempting to build a Wormhole node with the latest Go version (1.20.1) produces the error in figure 14.1. Go's release policy states, "Each major Go release is supported until there are two newer major releases." By not building with the latest Go version, Wormhole's ability to receive updates will expire.

```
cannot use "The version of quic-go you're using can't be built on Go 1.20 yet. For
more details, please see
https://github.com/lucas-clemente/quic-go/wiki/quic-go-and-Go-versions." (untyped
string constant "The version of quic-go you're using can't be built on Go 1.20 yet.
F...) as int value in variable declaration
```

Figure 14.1: Error produced when one tries to build the Wormhole with the latest Go version (1.20)

It is unclear when Go 1.21 will be released. Go 1.20 was released on February 1, 2023 (a few days prior to the start of the audit), and new versions appear to be released about every six months. We found a thread discussing Go 1.21, but it does not mention dates.

Exploit Scenario

Alice attempts to build a Wormhole node with Go version 1.20. When her attempt fails, Alice switches to Go version 1.19. Go 1.21 is released, and Go 1.19 ceases to receive updates. A vulnerability is found in a Go 1.19 package, and Alice is left vulnerable.

Recommendations

Short term, adapt the code so that it builds with Go version 1.20. This will allow Wormhole to receive updates for a greater period of time than if it builds only with Go version 1.19.

Long term, test with the latest Go version in CI. This will help identify incompatibilities like this one sooner.

References

- Go Release History (see Release Policy)
- Planning Go 1.21

15. Missing or wrong context

Severity: **Low**	Difficulty: **High**
Type: Timing	Finding ID: TOB-WORMGUWA-15
Target: `node/pkg/watchers/{algorand, cosmwasm, sui, wormchain}/watcher.go`	

Description

In several places where a `Context` is required, the Wormhole node creates a new background `Context` rather than using the passed-in `Context`. If the passed-in `Context` is canceled or times out, a `go` routine using the background `Context` will not detect this, and resources will be leaked.

The aforementioned problem is flagged by the `contextcheck` lint. For each of the locations named in figure 15.1, a `Context` is passed in to the enclosing function, but the passed-in `Context` is not used. Rather, a new background `Context` is created.

```
algorand/watcher.go:172:51: Non-inherited new context, use function like
`context.WithXXX` instead (contextcheck)
        status, err := algodClient.StatusAfterBlock(0).Do(context.Background())
                                                         ^
algorand/watcher.go:196:139: Non-inherited new context, use function like
`context.WithXXX` instead (contextcheck)
                        result, err :=
indexerClient.SearchForTransactions().TXID(base32.StdEncoding.WithPadding(base32.NoP
adding).EncodeToString(r.TxHash)).Do(context.Background())

^
algorand/watcher.go:205:42: Non-inherited new context, use function like
`context.WithXXX` instead (contextcheck)
                                block, err :=
algodClient.Block(r).Do(context.Background())
                                                                     ^
```

Figure 15.1: Warnings produced by `contextcheck`

A closely related problem is flagged by the `noctx` lint. In each of the locations named in figure 15.2, `http.Get` or `http.Post` is used. These functions do not take a `Context` argument. As such, if the `Context` passed in to the enclosing function is canceled, the `Get` or `Post` will not similarly be canceled.

```
cosmwasm/watcher.go:198:28: (*net/http.Client).Get must not be called (noctx)
```

```
                                resp, err := client.Get(fmt.Sprintf("%s/%s",
e.urlLCD, e.latestBlockURL))
                                                       ^
cosmwasm/watcher.go:246:28: (*net/http.Client).Get must not be called (noctx)
                                resp, err :=
client.Get(fmt.Sprintf("%s/cosmos/tx/v1beta1/txs/%s", e.urlLCD, tx))
                                                       ^
sui/watcher.go:315:26: net/http.Post must not be called (noctx)
                        resp, err := http.Post(e.suiRPC, "application/json",
strings.NewReader(buf))
                                              ^
sui/watcher.go:378:26: net/http.Post must not be called (noctx)
                        resp, err := http.Post(e.suiRPC, "application/json",
strings.NewReader(`{"jsonrpc":"2.0", "id": 1, "method": "sui_getCommitteeInfo",
"params": []}`))
                                              ^
wormchain/watcher.go:136:27: (*net/http.Client).Get must not be called (noctx)
                        resp, err := client.Get(fmt.Sprintf("%s/blocks/latest",
e.urlLCD))
                                               ^
```

Figure 15.2: Warnings produced by `noctx`

Exploit Scenario

A bug causes Alice's Algorand, Cosmwasm, Sui, or Wormchain node to hang. The bug triggers repeatedly. The connections from Alice's Wormhole node to the respective blockchain nodes hang, causing unnecessary resource consumption.

Recommendations

Short term, take the following steps:

- For each location named in figure 15.1, use the passed-in `Context` rather than creating a new background `Context`.
- For each location named in figure 15.2, rewrite the code to use `http.Client.Do`.

Taking these steps will help to prevent unnecessary resource consumption and potential denial of service.

Long term, enable the `contextcheck` and `notctx` lints in CI. The problems highlighted in this finding were uncovered by those lints. Running them regularly could help to identify similar problems.

References

- `checkcontext`
- `noctx`

16. Use of defer in a loop

Severity: **Low**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-WORMGUWA-16
Target: `node/pkg/watchers/solana/client.go`	

Description

The Solana watcher uses `defer` within an infinite loop (figure 16.1). Deferred calls are executed when their enclosing function returns. Since the enclosing loop is not exited under normal circumstances, the deferred calls are never executed and constitute a waste of resources.

```
for {
        select {
        case <-ctx.Done():
                return nil
        default:
                rCtx, cancel := context.WithTimeout(ctx, time.Second*300) // 5 minute
                defer cancel()
                ...
        }
}
```

Figure 16.1: `node/pkg/watchers/solana/client.go#L244–L271`

Sample code demonstrating the problem appears in appendix E.

Exploit Scenario

Alice runs her Wormhole node in an environment with constrained resources. Alice finds that her node is not able to achieve the same uptime as other Wormhole nodes. The underlying cause is resource exhaustion caused by the Solana watcher.

Recommendations

Short term, rewrite the code in figure 16.1 to eliminate the use of `defer` in the `for` loop. The easiest and most straightforward way would likely be to move the code in the `default` case into its own named function. Eliminating this use of `defer` in a loop will eliminate a potential source of resource exhaustion.

Long term, regularly review uses of `defer` to ensure they do not appear in a loop. To the best of our knowledge, there are not publicly available detectors for problems like this. However, regular manual review should be sufficient to spot them.

17. Finalizer is allowed to be nil

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-WORMGUWA-17
Target: `node/pkg/watchers/evm/connectors/poller.go`	

Description

The configuration of a chain's watcher can allow a finalizer to be `nil`, which may allow newly introduced bugs to go unnoticed.

Whenever a chain's RPC does not have a notion of "safe" or "finalized" blocks, the watcher polls the chain for the latest block using `BlockPollConnector`. After fetching a block, the watcher checks whether it is "final" in accordance with the respective chain's `PollFinalizer` implementation.

```
// BlockPollConnector polls for new blocks instead of subscribing when using
SubscribeForBlocks. It allows to specify a
// finalizer which will be used to only return finalized blocks on subscriptions.
type BlockPollConnector struct {
        Connector
        Delay             time.Duration
        useFinalized      bool
        publishSafeBlocks bool
        finalizer         PollFinalizer
        blockFeed         ethEvent.Feed
        errFeed           ethEvent.Feed
}
```

Figure 17.1: `node/pkg/watchers/evm/connectors/poller.go#L24–L34`

However, the method `pollBlocks` allows `BlockPollConnector` to have a `nil` `PollFinalizer` (see figure 17.2). This is unnecessary and may permit edge cases that could otherwise be avoided by requiring all `BlockPollConnectors` to use the `DefaultFinalizer` explicitly if a finalizer is not required (the default finalizer accepts all blocks as final). This will ensure that the watcher does not incidentally process a block received from `blockFeed` that is not in the canonical chain .

```
if b.finalizer != nil {
        finalized, err := b.finalizer.IsBlockFinalized(timeout, block)
        if err != nil {
                logger.Error("failed to check block finalization",
                        zap.Uint64("block", block.Number.Uint64()), zap.Error(err))
```

```
              return lastPublishedBlock, fmt.Errorf("failed to check block
finalization (%d): %w", block.Number.Uint64(), err)
       }

       if !finalized {
              break
       }
}

b.blockFeed.Send(block)
lastPublishedBlock = block
```

Figure 17.2: `node/pkg/watchers/evm/connectors/poller.go#L149–L164`

Exploit Scenario

A developer adds a new chain to the watcher using `BlockPollConnector` and forgets to add a `PollFinalizer`. Because a finalizer is not required to receive the latest blocks, transactions that were not included in the blockchain are considered valid, and funds are incorrectly transferred without corresponding deposits.

Recommendations

Short term, rewrite the block poller to require a finalizer. This makes the configuration of the block poller explicit and clarifies that a `DefaultFinalizer` is being used, indicating that no extra validations are being performed.

Long term, document the configuration and assumptions of each chain. Then, see if any changes could be made to the code to clarify the developers' intentions.

Summary of Recommendations

Trail of Bits recommends that Wormhole Foundation address the findings detailed in this report and take the following additional steps prior to deployment:

- Improve the code's documentation. We recommend, at a minimum, documenting every function accessible from outside of a package. (TOB-WORMGUWA-1, TOB-WORMGUWA-2)

- Review official and third-party guidance for handling channels. Several of this report's findings involved channels (TOB-WORMGUWA-7, TOB-WORMGUWA-8, TOB-WORMGUWA-11). Appendix F gives specific recommendations for avoiding channel-related problems.

- Simplify watcher and/or guardian configuration. There is considerable variation among how the watchers are launched and run (see Appendix D). The existing means of configuring the watchers is a complex set of nested if-statements. We recommend pursuing a simpler solution, potentially one that also extends to `guardiand`. (TOB-WORMGUWA-9, TOB-WORMGUWA-10)

- Develop more granular tests for the watchers. The present, primary means of testing the watchers involves Tilt. However, relying on such a high-level testing mechanism makes it difficult to know that all necessary conditions and edge cases are tested. (TOB-WORMGUWA-12)

A. Vulnerability Categories

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

Vulnerability Categories

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

Severity Levels

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

Difficulty Levels

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

B. Code Maturity Categories

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

Code Maturity Categories

Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria

Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Non-Security-Related Findings

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

- Eliminate the following unnecessary conversions identified by the `unconvert` lint:

```
governor.go:56:38: unnecessary conversion (unconvert)
const maxEnqueuedTime = time.Duration(time.Hour * 24)
                                     ^
governor_monitoring.go:442:16: unnecessary conversion (unconvert)
                ).Set(float64(available))
                             ^
governor_monitoring.go:448:16: unnecessary conversion (unconvert)
                ).Set(float64(numPending))
                             ^
near/nearapi/types.go:68:14: unnecessary conversion (unconvert)
        ts := uint64(ts_nanosec) / 1_000_000_000
                    ^
near/nearapi/types.go:87:15: unnecessary conversion (unconvert)
        return uint64(ts_nanosec) / 1000000000
                     ^
solana/client.go:230:16: unnecessary conversion (unconvert)
        if vaa.ChainID(s.chainID) != vaa.ChainIDPythNet {
                      ^
```

- Eliminate the following unused parameters identified by the `unparam` lint:

```
governor_db.go:59:78: `(*ChainGovernor).reloadPendingTransfer` - `now` is
unused (unparam)
func (gov *ChainGovernor) reloadPendingTransfer(pending *db.PendingTransfer,
now time.Time) {

^
governor_db.go:161:61: `(*ChainGovernor).reloadTransfer` - `now` is unused
(unparam)
func (gov *ChainGovernor) reloadTransfer(xfer *db.Transfer, now time.Time,
startTime time.Time) {
                                                            ^
solana/client.go:725:56: `(*SolanaWatcher).processAccountSubscriptionData` -
`ctx` is unused (unparam)
func (s *SolanaWatcher) processAccountSubscriptionData(ctx context.Context,
logger *zap.Logger, data []byte) error {
                                                       ^
```

- Use named constants in place of the following numeric literals identified by the go-mnd lint:

```
governor_monitoring.go:455:73: mnd: Magic number: 5, in <argument> detected
(gomnd)
                gov.nextConfigPublishTime = startTime.Add(time.Minute *
time.Duration(5))

^
governor.go:199:12: mnd: Magic number: 8, in <condition> detected (gomnd)
                if dec > 8 {
                         ^
governor_monitoring.go:543:21: mnd: Magic number: 20, in <condition> detected
(gomnd)
                        if numEnqueued < 20 {
                                         ^
```

The go-mnd lint also flags numerous magic numbers within the node/pkg/watchers directory.

- Either eliminate the use of the now variable, or set it after the mutex has been acquired, in the following code:

```
now := time.Now()
gov.mutex.Lock()
defer gov.mutex.Unlock()
```

If the node waits too long to acquire the mutex, the value of now could be inaccurate. Note that now is used to set a tokenEntry's priceTime field. However, the priceTime field does not appear to be currently used:

```
te.coinGeckoPrice = big.NewFloat(price)
te.updatePrice()
te.priceTime = now
```

- Eliminate the use of named returns in the following code, given that getAcalaMode returns immediately after setting either variable:

```
func (w *Watcher) getAcalaMode(ctx context.Context) (useFinalizedBlocks bool,
errRet error) {
        ...
        if err == nil {
                useFinalizedBlocks = true
                return
        }
```

The use of named returns could similarly be eliminated in the following code. In this case, `pollBlocks` can set `lastPublishedBlock` and return later. But the code would likely be clearer if `lastPublishedBlock` were a local variable featured explicitly in a return:

```
func (b *BlockPollConnector) pollBlocks(ctx context.Context, logger
*zap.Logger, lastBlock *NewBlock, safe bool) (lastPublishedBlock *NewBlock,
retErr error) {
        ...
        for {
                ...
                lastPublishedBlock = block
        }

        return
}
```

The use of named returns in the following code is unnecessary, since all returned values are explicitly named:

```
func (e *Watcher) ReadFinalChunksSince(logger *zap.Logger, ctx
context.Context, startHeight uint64, chunkSink chan<- nearapi.ChunkHeader)
(newestFinalHeight uint64, err error) {

        finalBlock, err := e.nearAPI.GetFinalBlock(ctx)
        if err != nil {
                // We can supress this error because this is equivalent to
saying that we haven't found any blocks since.
                return startHeight, nil
        }

        newestFinalHeight = finalBlock.Header.Height

        if newestFinalHeight > startHeight {
                ...
                if err != nil {
                        return startHeight, err
                }
        }

        return newestFinalHeight, nil
}
```

Finally, within the following two pieces of code, the named returns, ok and `retryable`, are not referenced at all:

```
func (s *SolanaWatcher) fetchBlock(ctx context.Context, logger *zap.Logger,
slot uint64, emptyRetry uint) (ok bool) {
```

```
func (s *SolanaWatcher) fetchMessageAccount(ctx context.Context, logger
*zap.Logger, acc solana.PublicKey, slot uint64) (retryable bool) {
```

- Have `ArbitrumConnector.TimeOfBlockByHash` check the first two bytes of the timestamp, rather than assume they are `0x`:

```
num, err := strconv.ParseUint(m.Time[2:], 16, 64)
if err != nil {
	return 0, fmt.Errorf("failed to parse time %s: %w", m.Time, err)
}
```

For comparison, Geth performs such a check:

```
func checkNumberText(input []byte) (raw []byte, err error) {
	if len(input) == 0 {
		return nil, nil // empty strings are allowed
	}
	if !bytesHave0xPrefix(input) {
		return nil, ErrMissingPrefix
	}
	input = input[2:]
	if len(input) == 0 {
		return nil, ErrEmptyNumber
	}
	if len(input) > 1 && input[0] == '0' {
		return nil, ErrLeadingZero
	}
	return input, nil
}
```

Similarly, have `ParseMessagePublicationAccount` check the first three bytes of `data`, rather than assume they are `msg`:

```
// Skip the b"msg" prefix
if err := borsh.Deserialize(prop, data[3:]); err != nil {
```

- Create a helper function to replace the following duplicated code:

```
// Note: Using Add() with a negative value because Sub() takes a time and
returns a duration, which is not what we want.
startTime := now.Add(-time.Minute * time.Duration(gov.dayLengthInMinutes))
```

For example, the code is also used again here:

```
startTime := time.Now().Add(-time.Minute *
time.Duration(gov.dayLengthInMinutes))
```

- Convert the following for loop to a while loop:

```
if e.next_round <= status.LastRound {
        for {
                block, err :=
algodClient.Block(e.next_round).Do(context.Background())
                if err != nil {
                        logger.Error(fmt.Sprintf("algodClient.Block %d: %s",
e.next_round, err.Error()))
                        p2p.DefaultRegistry.AddErrorCount(vaa.ChainIDAlgorand, 1)
                        break
                }

                if block.Round == 0 {
                        break
                }

                for _, element := range block.Payset {
                        lookAtTxn(e, element, block, logger)
                }
                e.next_round = e.next_round + 1

                if e.next_round > status.LastRound {
                        break
                }
        }
}
```

- Avoid single character variable names in favor of descriptive names:

```
func lookAtTxn(e *Watcher, t types.SignedTxnInBlock, b types.Block, logger
*zap.Logger)
```

- Eliminate the following uses of `fmt.Printf` and `fmt.Println`, and replace them with calls to a logger:

```
if newChunk.Hash != chunkHeader.Hash {
        fmt.Printf("queried hash=%s, return_hash=%s", chunkHeader.Hash,
newChunk.Hash)
        return Chunk{}, errors.New("Returned chunk hash does not equal queried
chunk hash")
}
```

```
res, err := base64.StdEncoding.DecodeString(value) //hex.DecodeString(value)
fmt.Println("address after second decode " + string(res))
```

- Fix the following two typos, which should be “suppress” and “unexpected”:

```
// We can supress this error because this is equivalent to saying that we
haven't found any blocks since.
return startHeight, nil
```

```
if err != nil {
        logger.Error("unspected error while parsing chunk data in event",
zap.Error(err))
}
```

- Within the following code, replace `200` with `http.StatusOK`:

```
if resp.StatusCode == 200 {
        return result, err
}
```

- Preallocate the variable result, as suggested by the `prealloc` lint (see also this StackOverflow answer on the topic):

```
var result []*transactionProcessingJob
...
for _, tx := range txns {
        result = append(result, newTransactionProcessingJob(tx.Hash,
tx.SignerId))
}
```

- In the following code, add a use of `defer ws.Close()` following the check that `err != nil`, to ensure the websocket is eventually closed:

```
ws, _, err := websocket.DefaultDialer.Dial(u.String(), nil)
if err != nil {
        logger.Error(fmt.Sprintf("e.suiWS: %s", err.Error()))
        return err
}
```

D. evm Watcher Configurations

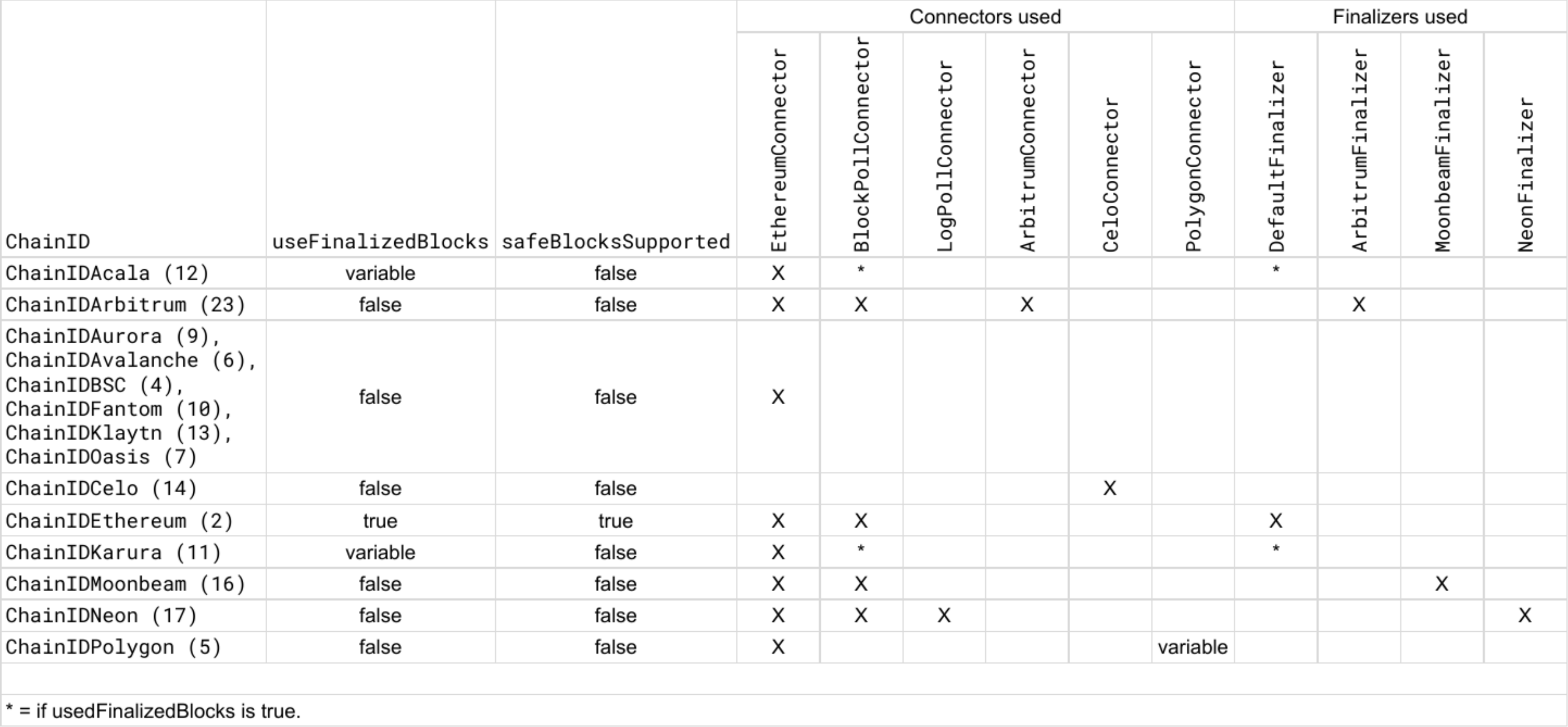
This appendix presents table D.2, which details the behavior of the evm watchers for various ChainIDs.

Table D.2 was constructed essentially by adding fmt.Println statements to indicate when connectors and finalizers were created, and by running the code in figure D.1.

```
func TestRun(t *testing.T) {
        for _, chainID := range []vaa.ChainID{
                vaa.ChainIDAcala,
                vaa.ChainIDArbitrum,
                vaa.ChainIDAurora,
                vaa.ChainIDAvalanche,
                vaa.ChainIDBSC,
                vaa.ChainIDCelo,
                vaa.ChainIDEthereum,
                vaa.ChainIDFantom,
                vaa.ChainIDKarura,
                vaa.ChainIDKlaytn,
                vaa.ChainIDMoonbeam,
                vaa.ChainIDNeon,
                vaa.ChainIDOasis,
                vaa.ChainIDPolygon,
        } {
                c := make(chan struct{})
                logger := zap.NewNop()
                supervisor.New(context.Background(), logger, func(ctx context.Context)
error {
                        supervisor.Signal(ctx, supervisor.SignalHealthy)
                        watcher := Watcher{
                                chainID:     chainID,
                                l1Finalizer: &mockL1Finalizer{},
                        }
                        if err := watcher.Run(ctx); err != nil {
                                panic(err)
                        }
                        supervisor.Signal(ctx, supervisor.SignalDone)
                        c <- struct{}{}
                        return nil
                })
                <-c
        }
}
```

Figure D.1: Code used to produce table D.1

Note that Acala and Karura required special treatment, since the connectors and finalizers they use can vary. For these two chains, we instrumented the getAcalaMode function to always return true or always return false, and ran the code in figure D.1 both ways. As indicated in table D.2, the two watchers use `LogPollConnector` and `DefaultFinalizer` when `useFinalizedBlocks` is `true`.



			Connectors used						Finalizers used			
ChainID	useFinalizedBlocks	safeBlocksSupported	EthereumConnector	BlockPollConnector	LogPollConnector	ArbitrumConnector	CeloConnector	PolygonConnector	DefaultFinalizer	ArbitrumFinalizer	MoonbeamFinalizer	NeonFinalizer
ChainIDAcala (12)	variable	false	X	*					*			
ChainIDArbitrum (23)	false	false	X	X		X				X		
ChainIDAurora (9), ChainIDAvalanche (6), ChainIDBSC (4), ChainIDFantom (10), ChainIDKlaytn (13), ChainIDOasis (7)	false	false	X									
ChainIDCelo (14)	false	false					X					
ChainIDEthereum (2)	true	true	X	X					X			
ChainIDKarura (11)	variable	false	X	*					*			
ChainIDMoonbeam (16)	false	false	X	X							X	
ChainIDNeon (17)	false	false	X	X	X							X
ChainIDPolygon (5)	false	false	X					variable				

* = if usedFinalizedBlocks is true.

Table D.2: The behavior of `evm.Watcher` with respect to the Ethereum ChainIDs

Key:

- `ChainID`: Named constant and its associated value from `sdk/vaa/structs.go` for an evm chain
- `usedFinalizedBlock`: Value of the `usedFinalizedBlock` variable at the conclusion of `evm.Watcher.Run`
- `safeBlocksSupported`: Value of the `safeBlocksSupported` variable at the conclusion of `evm.Watcher.Run`
- Connectors used: Connectors constructed in `evm.Watcher.Run`
- Finalizers used: Finalizers constructed in `evm.Watcher.Run`

E. Code Used to Verify TOB-WORMGUWA-16

This appendix contains code used to verify the problem described in TOB-WORMGUWA-16.

The code in figure E.1 contains two functions, `main` and `foo`, that behave as follows:

- `main` prints the total allocated heap memory, calls `foo()`, forces a garbage collect, and prints the total allocated heap memory again.
- `foo` prints the total allocated heap memory, defers a call to `cancel()` for a newly created `Context` 1,000 times, forces a garbage collect, and prints the total allocated heap memory again.

```
package main

import (
        "context"
        "fmt"
        "runtime"
)

func main() {
        ctx := context.Background()

        var before runtime.MemStats
        runtime.ReadMemStats(&before)
        fmt.Printf("before call: %v\n", before.Alloc)

        foo(ctx)

        runtime.GC()

        var after runtime.MemStats
        runtime.ReadMemStats(&after)
        fmt.Printf(" after call: %v\n", after.Alloc)
}

func foo(ctx context.Context) {
        var before runtime.MemStats
        runtime.ReadMemStats(&before)
        fmt.Printf("before loop: %v\n", before.Alloc)

        for i := 0; i < 1000; i++ {
                _, cancel := context.WithCancel(ctx)
                defer cancel()
                // cancel()
        }

        runtime.GC()

        var after runtime.MemStats
        runtime.ReadMemStats(&after)
        fmt.Printf(" after loop: %v\n", after.Alloc)
}
```

Figure E.1: Code used to verify the problem described in TOB-WORMGUWA-16

Figure E.2 shows the sample output produced by running the code in figure E.1. As shown in the lines labeled `after loop` and `after call`, significant heap memory is allocated during the loop. Moreover, that memory is freed when the call returns, suggesting the use of `defer cancel()` is the cause.

```
before call: 103760
before loop: 105064
 after loop: 283920
 after call: 109840
```

Figure E.2: Sample output from the code in figure E.1 (unmodified)

For control purposes, we also ran the code in figure E.1 with the `defer cancel()` replaced with a direct call to `cancel()`. As shown in figure E.3, some memory is still allocated by the loop, but the amount is considerably less, and none of that memory is freed when the call returns.

```
before call: 101696
before loop: 103000
 after loop: 105856
 after call: 105856
```

Figure E.3: Sample output from the code in figure E.1 with the `defer cancel()` replaced by a direct call to `cancel()`

F. Channel-Related Recommendations

Multiple findings in this report involved channels (e.g., TOB-WORMGUWA-7, TOB-WORMGUWA-8, TOB-WORMGUWA-11). This appendix provides general recommendations for avoiding channel-related problems.

- **Review official Golang guidance on how to use channels.** A Tour of Go offers a multi-part module on concurrency in Go. Parts 2 through 6 address channels, specifically.
- **Review third-party documentation on how to use channels.** Unofficial but valuable resources on the use of Go channels include the following:
  - Principles of designing Go APIs with channels
  - Understanding real-world concurrency bugs in Go
- **Document who each channel's senders and receivers are.** For each channel, document *who* is expected to write to it, and *who* is expected to read from it. "Who" could be a function, object, component, etc.—whatever makes sense for the situation. Providing such information to readers of the code will greatly increase the code's clarity.
- **Avoid calling close on a channel unless absolutely necessary.** As mentioned in part 4 of A Tour of Go, calling close on a channel is needed in only exceptional circumstances. Moreover, calling close on a channel can cause panics.
- **Use callbacks instead of channels where possible.** The two constructs are not interchangeable. However, for situations where a callback could be used, we recommend doing so instead of using a channel. Using a callback has the advantage that it is generally easy to tell where the arguments of the callback are used, i.e., within the function body. By comparison, it can be difficult to tell who the readers of a channel are, and thus difficult to tell where values written to the channel are used.
- **To the extent possible, store channels in local variables only, and avoid storing them in heap variables, passing them as function parameters, or returning them from functions.** The more a channel is moved around, the more difficult it becomes to tell who the channel's readers and writers are. Ideally, each channel is created within a function, and the channel is destroyed when the function returns.
- **Stay abreast of new tool developments for finding concurrency bugs in Go programs.** There appear to be few up-to-date tools for catching concurrency bugs in Go programs. We experimented with three of them; our results are described in the following bullets. This appears to be an area in need of further attention from the Go community.

    - The Go race detector is a tried-and-true dynamic analysis tool for catching certain currency bugs in Go programs. However, it does not catch any of the bugs described in this report.
    - GCatch is a static analysis tool for catching concurrency bugs in Go programs. Its support for Go modules is in beta. Moreover, we were unable to use it to produce meaningful results for the Wormhole codebase.
    - GoAT is a combined static and dynamic concurrency testing and analysis tool for Go programs. We were able to run it on some, but not all, of the Wormhole codebase. It alerted us to one potential deadlock in a test, though a seemingly uninteresting one.

References

- Introducing the Go Race Detector
- Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems: Extended Abstract
- GoAT: Automated Concurrency Analysis and Debugging Tool for Go