Wormhole

Security Assessment

September 22, 2022

Prepared for:
Wormhole Foundation

Prepared by: **Samuel Moelius, Maciej Domanski, 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 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

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 its Wormhole cross-chain messaging platform, specifically the Solana and CosmWasm contracts. From July 14 to August 19, 2022, a team of three 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. We conducted this audit with full knowledge of the codebase, including access to documentation. We performed static and dynamic testing of the codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered flaws that could impact confidentiality, integrity, or availability, though all but one appear to be highly difficult to exploit. A summary of the findings is provided on the next page.

Wormhole has elected not to use its optional fix review from the project statement of work and will be instead sourcing fixes from its community via public GitHub issues, listed in appendix E. As a result, Trail of Bits cannot confirm any finding's resolution past September 22, 2022.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	0
Low	4
Informational	11
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	2
Cryptography	1
Data Validation	4
Denial of Service	2
Error Reporting	1
Patching	3
Testing	2
Undefined Behavior	1

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

Samuel Moelius, Consultant
samuel.moelius@trailofbits.com

Maciej Domanski, Consultant
maciej.domanski@trailofbits.com

Troy Sargent, Consultant
troy.sargent@trailofbits.com

Project Timeline

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

Date	Event
July 14, 2022	Pre-project kickoff call
July 18, 2022	Review of "Wormhole: Life of a Message"
July 18, 2022	Solana contract walk-through
July 22, 2022	Status update meeting #1
July 29, 2022	Status update meeting #2
August 12, 2022	Status update meeting #3
August 19, 2022	Delivery of report draft
August 19, 2022	Report readout meeting
September 22, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of Wormhole's Solana and CosmWasm token bridge contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the codebase contain any "infinite mint" bugs?
- Is it possible to "brick" (i.e., make non-withdrawable) locked funds?
- Does the codebase contain denial-of-service vectors?
- Could an unauthorized user withdraw locked funds?
- Is it possible to forge an invalid verifiable action approval (VAA)?

Project Targets

The engagement involved a review and testing of the targets listed below.

Solana Contracts

Repository	https://github.com/certusone/wormhole
Version	c2db1116293ab81ddad4e87fc9af175656ab40e1
Directory	`solana`
Type	Rust
Platform	Solana

CosmWasm Contracts

Repository	https://github.com/certusone/wormhole
Version	c2db1116293ab81ddad4e87fc9af175656ab40e1
Directory	`cosmwasm`
Type	Rust
Platform	Cosmos

The Ethereum contracts were not within the scope of this review. However, we compared the Ethereum contracts to the Solana contracts to help us understand the latter. While doing so, we identified one issue in the Ethereum contracts (TOB-WORM-16).

Project Coverage

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

- **Static analysis with `cargo-audit`, `cargo-outdated`, and Clippy**
    - We ran `cargo-audit` and `cargo-outdated` over all of the `Cargo.lock` files. We ran Clippy with `-W clippy::pedantic` enabled over all of the Rust source files. We reviewed the results of each run.
- **Review of CosmWasm tests**
    - We ran the CosmWasm tests and verified that they passed.
- **Manual review**
    - We manually reviewed the Solana and CosmWasm contracts, with a focus on answering the questions listed under Project Goals.

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 elements that may warrant further review:

- We were unable to run the Solana contracts' tests. Thus, our assessment of those contracts was limited to static analysis and a manual review.

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:

Tool	Description	Results
`cargo-audit`	A Cargo subcommand for auditing `Cargo.lock` files for crates with security vulnerabilities reported to the RustSec Advisory Database	Only known false positives
`cargo-outdated`	A Cargo subcommand for displaying outdated Rust dependencies	TOB-WORM-7
`rust-clippy`	A collection of lints to catch common Rust mistakes and to improve Rust code	No significant results
`test-fuzz`	A collection of Rust macros and a Cargo subcommand that automate certain fuzzing-related tasks	Appendix D

Areas of Focus

Our automated testing and verification work focused on identifying the following:

- Vulnerable or outdated dependencies
- Common Rust mistakes
- The impacts of potential type cosplay vulnerabilities

Test Results

We found several outdated dependencies in the Solana contracts; the details are described in finding TOB-WORM-7. Running Clippy over the codebase produced no significant results. Our investigation into the impacts of finding TOB-WORM-5 are detailed in appendix D.

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	Unchecked arithmetic is used in several places without justification (TOB-WORM-15). One use of unchecked arithmetic could cause the associated contract to become unavailable (TOB-WORM-14). In some places in which *checked* arithmetic is used, panics are used in the event of overflows or underflows (TOB-WORM-4). Some arithmetic could be simplified (TOB-WORM-11).	**Moderate**
Auditing	We did not consider event auditing and logging during our review, as the guardians—Wormhole's monitoring nodes—were out of scope.	**Not Considered**
Authentication / Access Controls	We identified only minor issues related to access controls (TOB-WORM-11, TOB-WORM-12).	**Satisfactory**
Complexity Management	The project would benefit from additional documentation to manage the codebase's complexity, such as state transition diagrams. Additionally, the projects' tests should be made easier to run, and we identified some opportunities for code consolidation.	**Moderate**
Cryptography and Key Management	We identified one cryptography issue, specifically related to the possibility of hash collisions (TOB-WORM-2).	**Moderate**
Decentralization	Decentralization was only partially considered during our review, as we did not review the guardian-related code. However, we found that operations related to	**Further Investigation**

	governance appear to require the signatures of a quorum of guardians, as one would expect.	**Required**
Documentation	The Solana contracts' documentation is inadequate. Furthermore, both the Solana and CosmWasm contracts would benefit from additional code comments and a standard naming convention to make the codebase more clear.	**Moderate**
Front-Running Resistance	We identified no issues related to front-running.	**Satisfactory**
Low-Level Manipulation	Additional protections should be implemented for the transparent upgradeable proxy pattern used in the codebase (TOB-WORM-16).	**Moderate**
Testing and Verification	The minimal requirements for running the tests are too high (TOB-WORM-1). Furthermore, the `cw20-wrapped` tests would not build (TOB-WORM-9). The project would benefit from property and fuzz testing.	**Weak**

Summary of Findings

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

ID	Title	Type	Severity
1	Unconventional test structure	Testing	Informational
2	Risk of collisions due to hashing of dynamically sized data	Cryptography	Low
3	Use of deprecated methods	Patching	Low
4	Use of panics on invalid inputs	Error Reporting	Informational
5	No general protection against type cosplay in bridge	Data Validation	Undetermined
6	Insufficient documentation of Solana contracts	Patching	Informational
7	Outdated dependencies	Patching	Informational
8	AccountMeta::new constructor used for non-writable accounts	Denial of Service	Low
9	cw20-wrapped tests do not build	Testing	Informational
10	CosmWasm token bridge does not distinguish between loading errors	Data Validation	Informational
11	Quorum calculation can be simplified and made more robust	Access Controls	Informational
12	Bridge contract addresses cannot be updated	Access Controls	Low

13	Wormhole archive is checked unnecessarily	Data Validation	**Informational**
14	Overflow of native transfer counters may cause reverts	Denial of Service	**Informational**
15	Inconsistent use of checked math	Undefined Behavior	**Informational**
16	Insufficient safeguards against destruction in the proxy	Data Validation	**Informational**

Detailed Findings

1. Unconventional test structure	
Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-WORM-1
Target: Various	

Description
The sole method for testing Wormhole's Solana contracts relies on Tilt. As a result, tests that are complicated or rely on complex frameworks are less likely to be run.

Wormhole's Tilt tests involve launching a node for each supported chain. So, for example, one must launch Algorand and Ethereum nodes to test the Solana contracts.

Additionally, Tilt by itself is extremely heavyweight, as it relies on Kubernetes, and Wormhole's Tilt tests rely on Docker. One can try to use Docker directly to run the tests without Tilt, but even Docker is considerably heavyweight and its behavior can vary across platforms.

The Wormhole team shared with us the beginnings of a Node.js-based framework that could be used in place of Tilt for testing. However, the Node.js-based setup does not deploy the Solana contracts; one must still use Docker to do this.

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 contracts on any given chain.

Exploit Scenario
Alice, a Wormhole developer, introduces a bug into the codebase. Bob reviews the code but does not spot the bug, in part because he is unable to run the tests.

Recommendations
Short term, take the following steps:

- Develop tests that require minimal tooling whose behavior does not vary across platforms. For the Solana contracts specifically, use Solana's `solana_program_test` framework to test native compilations of the contracts.
- Ensure that deployment scripts are self-contained and are supplemented by instructions that can be performed by a human.

If the tests are easy to run, developers are more likely to run them.

Long term, run the tests on multiple platforms (e.g., Linux, macOS, Windows) in the CI process. Doing so will ensure that the tests are run regularly and will help to identify platform-specific incompatibilities.

References

- `solana_program_test::ProgramTest::add_program`

2. Risk of collisions due to hashing of dynamically sized data

Severity: **Low**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-WORM-2
Target: `solana/bridge/program/src/api/post_vaa.rs#L219-L238`	

Description

Whenever a message receives the necessary signatures from a quorum of guardians, relayers call `post_vaa` to announce the message's validity. The `post_vaa` function performs several validation checks, one of which (the `check_integrity` function, shown in figure 2.1) hashes the `PostVAAData` structure to ensure that it matches the structure signed by the guardians. Each field of the structure is written into a vector of bytes and then hashed.

However, `check_integrity` uses the `write_all` function improperly in creating the structure to be hashed: `write_all` is used to write adjacent, dynamically sized fields (`emitter_address` and `payload`), which could allow hash collisions to occur. The risk of a hash collision breaks the assumption that each verifiable action approval (VAA) corresponds to one message. If the `emitter_address` and `payload` fields, highlighted in figure 2.1, are moved in the future, this risk would be exacerbated.

```
let body = {
    let mut v = Cursor::new(Vec::new());
    v.write_u32::<BigEndian>(vaa.timestamp)?;
    v.write_u32::<BigEndian>(vaa.nonce)?;
    v.write_u16::<BigEndian>(vaa.emitter_chain)?;
    v.write_all(&vaa.emitter_address)?;
    v.write_u64::<BigEndian>(vaa.sequence)?;
    v.write_u8(vaa.consistency_level)?;
    v.write_all(&vaa.payload)?;
    v.into_inner()
};

// Hash this body, which is expected to be the same as the hash currently stored in
the
// signature account, binding that set of signatures to this VAA.
let body_hash: [u8; 32] = {
    let mut h = sha3::Keccak256::default();
    h.write(body.as_slice())
        .map_err(|_| ProgramError::InvalidArgument)?;
    h.finalize().into()
};
```

Figure 2.1: Part of the `check_integrity` function

Exploit Scenario

An attacker generates VAA data that is invalid but results in the same hash as one that has received signatures from a quorum of guardians. The attacker then submits the VAA data to a third-party program that uses Wormhole's core messaging protocol. Since the program assumes that only one valid structure exists for each VAA hash, it permits the attacker's invalid input, executing actions based on falsified data. The following is an example of a hash collision:

```
let vaa0 = PostVAAData {
    version: 1,
    guardian_set_index: 2,

    // Body part
    timestamp: 3,
    nonce: 4,
    emitter_chain: 5,
    emitter_address: Vec::from([2, 3, 4]),
    sequence: 7,
    consistency_level: 255,
    payload: Vec::from([0, 0, 0, 0, 0, 0, 0, 7, 255, 5, 6, 7, 8]),
};
let vaa1 = PostVAAData {
    version: 1,
    guardian_set_index: 2,

    // Body part
    timestamp: 3,
    nonce: 4,
    emitter_chain: 5,
    emitter_address: Vec::from([2, 3, 4, 0, 0, 0, 0, 0, 0, 0, 7, 255]),
    sequence: 7,
    consistency_level: 255,
    payload: Vec::from([5, 6, 7, 8]),
};
```

Figure 2.2: An example hash collision

Recommendations

Short term, modify the `check_integrity` function so that the lengths of the two dynamically sized fields are inserted into the vector before hashing the vector of bytes. Doing so will help prevent hash collisions.

Long term, review the cardinality assumptions of VAAs and messages to ensure that undefined behavior is unlikely to occur.

3. Use of deprecated methods

Severity: **Low**	Difficulty: **High**
Type: Patching	Finding ID: TOB-WORM-3
Target: `wormhole/solana/bridge/client/src/main.rs#L106`, `wormhole/solana/bridge/client/src/main.rs#L170`	

Description

The `command_deploy_bridge` and `command_post_message` functions use the deprecated `calculate_fee` function (figure 3.1). The comment highlighted in figure 3.1 justifies the use of another deprecated function, `get_recent_blockhash`, through the `#[allow(deprecated)]` attribute, but the comment does not justify the use of `calculate_fee`. As a result, developers could overlook this deprecated function.

```
// [`get_recent_blockhash`] is deprecated, but devnet deployment hangs using the
// recommended method, so allowing deprecated here. This is only the client, so
// no risk.
#[allow(deprecated)]
fn command_deploy_bridge(
    config: &Config,
    bridge: &Pubkey,
    initial_guardians: Vec<[u8; 20]>,
    guardian_expiration: u32,
    message_fee: u64,
) -> CommmandResult {
// (...)
fee_calculator.calculate_fee(transaction.message()),
```

Figure 3.1: solana/bridge/client/src/main.rs#75–106

Exploit Scenario

Developers overlook a deprecated method because of the `#[allow(deprecated)]` attribute. An attacker finds a deprecated method that was replaced for security reasons, exploits it, and steals funds.

Recommendations

Short term, revise the code so that the `calculate_fee` function is not needed. This will make the code compatible with future versions of Solana.

Long term, regularly test the Wormhole code with the most recent version of Solana available for potential deprecated methods. Testing with recent versions of Solana will help alert the team to deprecations affecting the code.

4. Use of panics on invalid inputs

Severity: **Informational**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-WORM-4
Target: Various	

Description

In several places, the code panics when an arithmetic overflow or underflow occurs. Panics should be reserved for programmer errors (e.g., assertion violations). Panicking on user errors dilutes the utility of the panic operation.

An example appears in figure 4.1. The `complete_native` function uses `checked_sub` to verify that the transfer includes sufficient funds to cover the fee. However, `complete_native` panics if there are insufficient funds. Thus, the function could panic on either a programmer error or a user error.

```
pub fn complete_native(
    ctx: &ExecutionContext,
    accs: &mut CompleteNative,
    _data: CompleteNativeData,
) -> Result<()> {
    ...
    // Transfer tokens
    let transfer_ix = spl_token::instruction::transfer(
        &spl_token::id(),
        accs.custody.info().key,
        accs.to.info().key,
        accs.custody_signer.key,
        &[],
        amount.checked_sub(fee).unwrap(),
    )?;
```

Figure 4.1: solana/modules/token_bridge/program/src/api/complete_transfer.rs#L74-L140

We noticed the use of unwrap with checked arithmetic in at least the following locations:

- `modules/token_bridge/program/src/api/complete_transfer.rs:139`
- `modules/token_bridge/program/src/api/complete_transfer.rs:251`
- `migration/src/api/remove_liquidity.rs:86`

- `migration/src/api/remove_liquidity.rs:90`
- `migration/src/api/add_liquidity.rs:99`
- `migration/src/api/add_liquidity.rs:103`
- `migration/src/api/migrate_tokens.rs:101`
- `migration/src/api/migrate_tokens.rs:105`

This problem is also present in the CosmWasm contracts in at least the following locations:

- `contracts/token-bridge/src/contract.rs:836`
- `contracts/token-bridge/src/contract.rs:899`
- `contracts/token-bridge/src/contract.rs:900`
- `contracts/token-bridge/src/contract.rs:981`
- `contracts/token-bridge/src/contract.rs:1169`
- `contracts/token-bridge/src/contract.rs:1175`
- `contracts/token-bridge/src/contract.rs:1203`
- `contracts/token-bridge/src/contract.rs:1204`

Exploit Scenario

Alice, a Wormhole developer, observes a panic in the Wormhole codebase. Alice ignores the panic, believing that it is caused by user error, but it is actually caused by a bug she introduced.

Recommendations

Short term, reserve the use of panics for programmer errors. Have relevant areas of the code return `Result::Err` on user errors. Adopting such a policy will help to distinguish the two types of errors when they occur.

Long term, consider denying the following Clippy lints:

- `clippy::expect_used`
- `clippy::unwrap_used`
- `clippy::panic`

Doing so will not prevent all panics, but it will prevent many of them.

5. No general protection against type cosplay in bridge	
Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-WORM-5
Target: `solana/bridge`	

Description

Type cosplay is when a contract account of one type is passed in as another type, resulting in type confusion. The bridge contract protects against this issue in certain special cases but has no general protection.

The bridge contract uses the first three bytes of an account to distinguish between `PostedVAAData`, `PostedMessageData`, and `PostedMessageUnreliableData` accounts (e.g., figure 5.1).

```
impl BorshSerialize for PostedVAAData {
    fn serialize<W: Write>(&self, writer: &mut W) -> std::io::Result<()> {
        writer.write_all(b"vaa")?;
        BorshSerialize::serialize(&self.message, writer)
    }
}
```

Figure 5.1: solana/bridge/program/src/accounts/posted_vaa.rs#L43-L48

The bridge contract also uses derived addresses to distinguish between “bridge” and “fee collector” accounts:

```
pub type Bridge<'a, const State: AccountState> = Derive<Data<'a, BridgeData, { State
}>, "Bridge">;
```

Figure 5.2: solana/bridge/program/src/accounts/bridge.rs#L20

However, other types of accounts (e.g., `Claim`, `GuardianSet`, etc.) do not seem to have any form of protection against type cosplay. One of these accounts could be passed where another type was expected, resulting in type confusion.

Exploit Scenario

Eve notices that two types of bridge-owned accounts could serialize to the same sequence of bytes. Eve uses this fact to trick the bridge into accepting an account that it should not.

Recommendations

Short term, prefix all bridge-owned accounts with a fixed length “discriminant” (i.e., a byte sequence that identifies the type of account). This will help protect the bridge from type cosplay attacks.

Long term, if additional accounts must be added to the bridge, ensure they are assigned a new byte sequence that is not already assigned to an existing account type. This will help ensure that adding new account types does not introduce a type cosplay attack vector.

References

- pencilflip.sol on type cosplay

6. Insufficient documentation of Solana contracts	
Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-WORM-6
Target: Various	

Description
The Solana contracts' documentation is insufficient, lacking high-level descriptions and examples. Insufficient documentation makes the code difficult to review and increases the risk that developers attempting to integrate their projects with Wormhole will make mistakes.

The "Solana signature verification: A deep dive" document describes the inner workings of the bridge contract well. However, the following should also be documented:

- The purpose of each crate, including which crates contain legacy code (e.g., `migrations`) and which crates are used only for testing (e.g., `program_stub`)
- The checks that a program using the Wormhole messaging protocol must perform (e.g., to avoid processing a VAA twice)

With regard to the token and NFT bridges, the following should be documented:

- The way the contracts perform the checks described in the previous bullet point
- The purpose of each of the contracts' instructions
- The signer, ownership, address derivation, and account initialization checks that each instruction performs

Exploit Scenario
Bob, a Solana developer, makes mistakes in integrating Wormhole for cross-chain messaging because of insufficient documentation. Bob suffers financial loss as a result.

Recommendations
Short term, document the aspects of the Solana contracts described above. Doing so will assist future auditors and will reduce the risk that developers integrating their projects with Wormhole will do so incorrectly.

Long term, regularly review the documentation to ensure it is accurate. Documentation must be kept up to date to be beneficial.

7. Outdated dependencies

Severity: **Informational**	Difficulty: **Undetermined**
Type: Patching	Finding ID: TOB-WORM-7
Target: Various	

Description

Trail of Bits used `cargo outdated` to detect outdated dependencies in the Wormhole codebase. The tool detected a number of outdated packages that are referenced by the `Cargo.toml` files.

Dependency	Version currently in use	Latest version available
`borsh`	0.9.1	0.9.3
`clap`	2.34.0	3.2.15
`libsecp256k1`	0.6.0	0.7.1
`primitive-types`	0.9.1	0.11.1
`rand`	0.7.3	0.8.5
`sha3`	0.9.1	0.10.1
`solana-clap-utils`	1.9.4	1.11.4
`solana-cli-config`	1.9.4	1.11.4
`solana-client`	1.9.4	1.11.4
`solana-program`	1.9.4	1.11.4
`solana-program-test`	1.9.4	1.11.4
`solana-sdk`	1.9.4	1.11.4
`spl-associated-token-account`	1.0.3	1.0.5
`spl-token`	3.2.0	3.3.0

Exploit Scenario

Alice, the developer of a dependency used by Wormhole, notices a critical bug in the dependency. She fixes it in the latest major version but does not backport the fix to earlier major versions. Because Wormhole does not use the latest major version of the dependency, Wormhole does not receive the fix.

Recommendations

Short term, update the build process dependencies to their latest versions wherever possible. Use tools such as `cargo outdated` and `cargo upgrade` to confirm that no outdated dependencies remain.

Long term, implement outdated dependency checks as part of the CI/CD pipeline of application development. Do not allow builds to continue with any outdated dependencies.

8. AccountMeta::new constructor used for non-writable accounts	
Severity: **Low**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-WORM-8
Target: Various	

Description
Accounts that are not modified by the Solana contracts are specified as writable by the `AccountMeta::new` constructor. Since the accounts are not modified, `AccountMeta::new_readonly` could be used instead.

The Solana documentation states the following:

> Note that because the Solana runtime schedules parallel transaction execution around which accounts are writable, care should be taken that only accounts which actually may be mutated are specified as writable. As the default AccountMeta::new constructor creates writable accounts, this is a minor hazard: use AccountMeta::new_readonly to specify that an account is not writable.

Figure 8.1 shows an example of a discrepancy between constructors for the same type of account.

```
287 AccountMeta::new(solana_program::sysvar::rent::id(), false),
// (...)
369 AccountMeta::new_readonly(solana_program::sysvar::rent::id(), false),
```

Figure 8.1: Metadata construction for the `solana_program::sysvar::rent::id()` accounts (`solana/modules/nft_bridge/program/src/instructions.rs#287–369`)

Exploit Scenario
An attacker finds a bug in the Solana code that affects parallel transaction execution because of a lack of `new_readonly` constructor usage. The bug leads to excessive resource consumption and makes the codebase vulnerable to denial-of-service attacks.

Recommendations
Short term, change the constructors for immutable accounts to `AccountMeta::new_readonly`. Doing so will help prevent unnecessary resource consumption.

Long term, periodically check whether the proper methods are used for non-mutable accounts. Doing so will help to ensure that a code change does not introduce a denial-of-service vector.

9. cw20-wrapped tests do not build

Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-WORM-9
Target: `cosmwasm/contracts/cw20-wrapped/src/contract.rs`, `cosmwasm/Makefile`	

Description

The `cw20-wrapped` package's tests do not build.

The issue appears to be a change to the `mock_dependencies` function in the `cosmwasm_std` library. The existing tests try to pass one argument to this function (e.g., figure 9.1). However, beginning with `cosmwasm_std` 1.0.0, the function takes no arguments (figure 9.2). Since the tests rely on `cosmwasm_std` 1.0.0, they do not build.

```
#[test]
fn can_mint_by_minter() {
    let mut deps = mock_dependencies(&[]);
    let minter = HumanAddr::from("minter");
    let recipient = HumanAddr::from("recipient");
    let amount = Uint128::new(222_222_222);
    do_init_and_mint(deps.as_mut(), &minter, &recipient, amount);
}
```

Figure 9.1: cosmwasm/contracts/cw20-wrapped/src/contract.rs#L316-L323

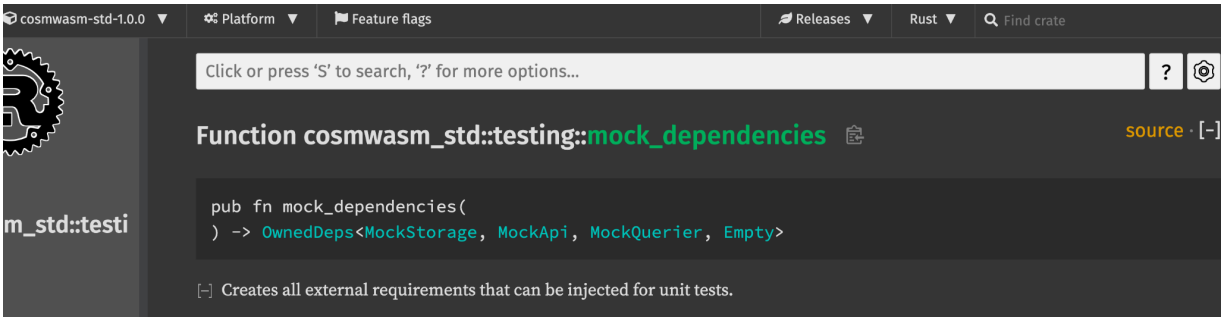

Figure 9.2: docs.rs/cosmwasm-std/1.0.0/cosmwasm_std/testing/fn.mock_dependencies.html

Note that the makefile's `test` target tests the `wormhole` and `token-bridge` packages but not the `cw20-wrapped` package (figure 9.3). This fact is likely in part why the bug was not noticed earlier:

```
.PHONY: unit-test
## Run unit tests
unit-test:
   cargo test -p wormhole-bridge-terra-2
   cargo test -p token-bridge-terra-2

.PHONY: test
## Run unit and integration tests
test: artifacts test/node_modules LocalTerra unit-test
   @if pgrep terrad; then echo "Error: terrad already running. Stop it before
running tests"; exit 1; fi
   cd LocalTerra && docker compose up --detach
   sleep 5
   cd test && npm run test || (cd ../LocalTerra && docker compose down && exit 1)
   cd LocalTerra && docker compose down
```

Figure 9.3: cosmwasm/Makefile#L57-L70

Exploit Scenario

Alice, a Wormhole developer, introduces a bug into the `cw20-wrapped` contract. The bug goes unnoticed because the contract's tests do not build.

Recommendations

Short term, take the following steps:

- Review the changes to the `mock_dependencies` function and remove its argument if it is unneeded. Doing so will enable the tests to build and run.
- Update the makefile's `test` target so that it runs the `cw20-wrapped` package's tests.

Long term, regularly build all Rust code with `--all-targets`. Doing so will help ensure that all targets (including tests) can be built.

References

- The Cargo Book: Target Selection

10. CosmWasm token bridge does not distinguish between loading errors

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-WORM-10
Target: `cosmwasm/contracts/token-bridge/src/contract.rs`	

Description

In several places, such as the examples in figures 10.1 and 10.2, the `cosmwasm` token bridge checks for the absence of a key in storage. If an error is returned, the contract implicitly assumes that it is a `NotFound` error. However, the `load` method could also return `ParseErr` (figure 10.3). Thus, the current practice could allow such errors to go unnoticed.

```
    let result = bucket.load(&token_address.serialize()).ok();
    if result.is_some() {
        return ContractError::AssetAlreadyRegistered.std_err();
    }
```

Figure 10.1: cosmwasm/contracts/token-bridge/src/contract.rs#L378-L381

```
    let existing = bridge_contracts_read(deps.storage).load(&chain_id.to_be_bytes());
    if existing.is_ok() {
        return Err(StdError::generic_err(
            "bridge contract already exists for this chain",
        ));
    }
```

Figure 10.2: cosmwasm/contracts/token-bridge/src/contract.rs#L732-L737

```
    /// load will return an error if no data is set at the given key, or on parse
error
    pub fn load(&self, key: &[u8]) -> StdResult<T> {
        let value = get_with_prefix(self.storage, &self.prefix, key);
        must_deserialize(&value)
    }
```

Figure 10.3: cosmwasm/packages/storage/src/bucket.rs#L143-L147

Based on our understanding, a `ParseErr` error arises only if a piece of data is serialized as one type and later deserialized as another type. Thus, such errors may not be currently realizable. However, a future code change could introduce such a serialization-deserialization inconsistency. Modifying the checks so that they catch `ParseErr` errors will help to protect against this possibility.

Exploit Scenario

Alice, a Wormhole developer, introduces a bug in the `cosmwasm` contracts that causes a piece of data to be serialized as one type but deserialized as another. The bug goes unnoticed because the errors returned by the relevant `load` calls are not checked.

Recommendations

Short term, if `NotFound` errors are expected to be returned from `load` calls, add a check to verify that errors are actually `NotFound`. Doing so will help to ensure that `ParseErr` errors do not go unnoticed.

Long term, whenever an error is expected, check which error variant was returned. Rarely does an error type include just one variant. Assuming the variant returned is the one expected could allow unanticipated errors to go unnoticed.

References

- `cosmwasm_std::StdError`

11. Quorum calculation can be simplified and made more robust

Severity: **Informational**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-WORM-11
Target: `cosmwasm/contracts/wormhole/src/state.rs`, `solana/bridge/program/src/api/post_vaa.rs`	

Description

The Solana and CosmWasm Wormhole contracts perform fixed point integer division with one decimal of precision (scaling factor of 10) to calculate a two-thirds quorum.

```
let required_consensus_count = {
    let len = accs.guardian_set.keys.len();
    // Fixed point number transformation with one decimal to deal with rounding.
    let len = (len * 10) / 3;
    // Multiplication by two to get a 2/3 quorum.
    let len = len * 2;
    // Division to bring number back into range.
    len / 10 + 1
};
```

Figure 11.1: The Solana Wormhole quorum calculation

```
pub fn quorum(&self) -> usize {
    // allow quorum of 0 for testing purposes...
    if self.addresses.is_empty() {
        return 0;
    }
    ((self.addresses.len() * 10 / 3) * 2) / 10 + 1
}
```

Figure 11.2: The CosmWasm Wormhole quorum calculation

However, the scaling factor is not necessary if the order of operations is changed to multiply-before-divide, as opposed to divide-before-multiply. The following implementation is equivalent (see figure 11.4) and more readable:

```
quorum = no_of_guardians * 2 / 3 + 1
```

Figure 11.3: A simplified two-thirds quorum calculation

```
from z3 import Solver, Int
s = Solver()
s.add(Int('x') * 2 / 3 + 1 != Int('x') * 10 / 3 * 2 / 10 + 1)
```

```
assert str(s.check()) == "unsat"
```

Figure 11.4: The SMT solver provides no falsifying counterexample.

In addition, the quorum calculation could be used by both the Solana and CosmWasm contracts, guaranteeing that both Wormhole implementations require the same number of signatures.

Exploit Scenario

Alice, a Wormhole developer, modifies the `cosmwasm` contracts' quorum calculation; the calculation now deviates from that of the Solana program implementation. A VAA that would be considered invalid on Solana due to insufficient signers is accepted by the `cosmwasm` contracts.

Recommendations

Short term, modify the calculation so that it multiplies before dividing to avoid the need for a fixed point scaling factor.

Long term, limit code duplication by sharing implementations between contracts of the same language, namely, Rust. Run differential fuzzing tests between implementations across languages, Solidity and Rust, to verify equivalence.

12. Bridge contract addresses cannot be updated

Severity: **Low**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-WORM-12
Target: `cosmwasm/contracts/token-bridge/src/contract.rs`	

Description
CosmWasm token bridges store the addresses of bridges on other chains. However, once an address is set, it can never be changed. This could be problematic if control of an external bridge is ever lost.

The relevant part of the `handle_register_chain` function appears in figure 12.1. The function checks whether a bridge address was previously registered for `chain_id`. If so, an error is returned. Thus, there is no way to update the address in this case.

```
let existing = bridge_contracts_read(deps.storage).load(&chain_id.to_be_bytes());
if existing.is_ok() {
    return Err(StdError::generic_err(
        "bridge contract already exists for this chain",
    ));
}
```

Figure 12.1: cosmwasm/contracts/token-bridge/src/contract.rs#L732-L737

Exploit Scenario
Mallory gains control over a CosmWasm token bridge. The other CosmWasm token bridges cannot change the address of the token bridge that Mallory controls. A contract upgrade is required to invalidate the rogue token bridge.

Recommendations
Short term, add a function to the CosmWasm token bridge to update the addresses of existing, registered bridges. Add tests to verify that the new function works correctly. Having this functionality will help if control over a bridge is ever lost.

Long term, advise the Wormhole guardians to develop an incident response plan to inform their decisions if a bridge is lost. Doing so will help to prepare them for such a situation.

13. Wormhole archive is checked unnecessarily

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-WORM-13
Target: `cosmwasm/contracts/wormhole/src/contract.rs`	

Description

The CosmWasm core contract exposes a function, `parse_and_verify_vaa`, to verify the structure of a given VAA. However, the function also checks that the VAA was not previously processed by the core contract. This behavior could be problematic should a contract ever want to verify the structure of a VAA previously processed by the core contract.

The relevant part of the `parse_and_verify_vaa` function appears in figure 13.1. As part of the verification, the function performs an "archive check," whether the message was previously processed by the core contract. If so, the function returns an error.

```
/// Parses raw VAA data into a struct and verifies whether it contains sufficient
signatures of an
/// active guardian set i.e. is valid according to Wormhole consensus rules
fn parse_and_verify_vaa(
    storage: &dyn Storage,
    data: &[u8],
    block_time: u64,
) -> StdResult<ParsedVAA> {
    let vaa = ParsedVAA::deserialize(data)?;

    if vaa.version != 1 {
        return ContractError::InvalidVersion.std_err();
    }

    // Check if VAA with this hash was already accepted
    if vaa_archive_check(storage, vaa.hash.as_slice()) {
        return ContractError::VaaAlreadyExecuted.std_err();
    }
```

Figure 13.1: cosmwasm/contracts/wormhole/src/contract.rs#L177-L193

There are at least two downsides to this behavior:

- There could be legitimate reasons for verifying the structure of a VAA that was previously processed by the core contract. However, the current API provides no way to verify such VAAs.

- If there were overlap between the VAAs used by some other contract and the core contract, then this behavior could present a denial-of-service vector. That is, an attacker could submit a VAA to the core contract to cause the VAA to be considered invalid thereafter.

Exploit Scenario
Alice develops a CosmWasm contract whose purpose is to record all valid VAAs. However, Alice's contract has no way to verify the structure of a VAA that was previously processed by the core contract.

Recommendations
Short term, move the "archive check" out of `parse_and_verify_vaa` and into the function that handles VAA submission. Doing so will produce a more intuitive API and eliminate a potential denial-of-service vector.

Long term, develop a means for monitoring the VAAs used in practice and regularly review them. Doing so could help to identify cases in which one application's VAAs could be used adversely against another application.

14. Overflow of native transfer counters may cause reverts

Severity: **Informational**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-WORM-14
Target: `cosmwasm/contracts/token-bridge/src/state.rs`	

Description

The `send_native` function performs unchecked math that may overflow and cause a `0` (or a sufficiently small value) to be stored in the `counter_bucket` for a given asset address. Then, if `receive_native` loads this storage value and subtracts a greater amount, the operation will revert due to the use of `checked_sub`, which guards against overflows. Transfers of native tokens may be disrupted until a contract upgrade can mitigate this issue.

```
pub fn send_native(
    storage: &mut dyn Storage,
    asset_address: &ExternalTokenId,
    amount: Uint128,
) -> StdResult<()> {
    let mut counter_bucket = bucket(storage, NATIVE_COUNTER);
    let new_total = amount
        + counter_bucket
            .load(asset_address.serialize().as_slice())
            .unwrap_or(Uint128::zero());
    if new_total > Uint128::new(u64::MAX as u128) {
        return Err(StdError::generic_err(
            "transfer exceeds max outstanding bridged token amount",
        ));
    }
    counter_bucket.save(asset_address.serialize().as_slice(), &new_total)
}

pub fn receive_native(
    storage: &mut dyn Storage,
    asset_address: &ExternalTokenId,
    amount: Uint128,
) -> StdResult<()> {
    let mut counter_bucket = bucket(storage, NATIVE_COUNTER);
    let total: Uint128 = counter_bucket.load(asset_address.serialize().as_slice())?;
    let result = total.checked_sub(amount)?;
    counter_bucket.save(asset_address.serialize().as_slice(), &result)
}
```

Figure 14.1: cosmwasm/contracts/token-bridge/src/state.rs#L135-L162

Exploit Scenario

Alice, a user, initiates a native transfer that causes an overflow, and the value `0` is stored in the `counter_bucket`. The `receive_native` operation will not succeed for Bob, preventing his funds from being processed.

Recommendations

Short term, use checked math in `send_native` and write unit tests that cover this scenario.

Long term, with a fuzzer, perform multiple operations in sequence to identify cases in which one operation could cause the failure of another. Doing so could help to identify bugs like this one.

15. Inconsistent use of checked math

Severity: **Informational**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-WORM-15
Target: `cosmwasm/contracts/wormhole/src/contract.rs`	

Description

The CosmWasm contracts use checked math fairly consistently; however, there are several instances in which unchecked math is used without explanation. While only one instance was identified as a vulnerability (TOB-WORM-14), the other instances of unchecked math should use their respective checked math operation unless a justification is documented. This will eliminate any sources of undefined behavior in the code.

The following are some of the uses of unchecked math in the CosmWasm contracts:

```
let multiplier = 10u128.pow((max(decimals, 8u8) - 8u8) as u32);
```

Figure 15.1: cosmwasm/contracts/token-bridge/src/contract.rs#L898

```
receive_native(deps.storage, &external_id, Uint128::new(amount + fee))?;
```

Figure 15.2: cosmwasm/contracts/token-bridge/src/contract.rs#L888

```
let real_amount = new_balance.balance - Uint128::from_str(&state.previous_balance)?;
let real_amount = real_amount / multiplier;
```

Figure 15.3: cosmwasm/contracts/token-bridge/src/contract.rs#L196-L197

Exploit Scenario

Bob, a user, sends a payload that causes undefined behavior, and he receives fewer tokens than he anticipated due to arithmetic errors.

Alternatively, an attacker crafts a payload that exploits the arithmetic errors. In turn, she receives excess tokens.

Recommendations

Short term, use checked math wherever unchecked math is currently used. Alternatively, if overflow is desired, use wrapping/saturating arithmetic APIs.

Long term, avoid allowing undefined behavior such as overflows to occur, and document how edge cases are handled in the codebase. Consider setting Clippy's `integer-arithmetic` lint to deny to encourage the use of checked arithmetic.

16. Insufficient safeguards against destruction in the proxy

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-WORM-16
Target: `ethereum/contracts/Governance.sol`	

Description

Following a bug report of an uninitialized implementation contract that could be destroyed, a new Wormhole implementation was deployed that deleted the function body of `initialize`. While deleting the body of `initialize` mitigated the vulnerability, more robust safeguards against contract destruction should be implemented. The current protections are not documented, and they are not standard practice.

```
function upgradeImplementation(address newImplementation) internal {
    address currentImplementation = _getImplementation();

    _upgradeTo(newImplementation);

    // Call initialize function of the new implementation
    (bool success, bytes memory reason) =
newImplementation.delegatecall(abi.encodeWithSignature("initialize()"));

    require(success, string(reason));

    emit ContractUpgraded(currentImplementation, newImplementation);
}
```

Figure 16.1: ethereum/contracts/Governance.sol#L144-L155

The following condition prevents a valid, verified message from being provided because the `guardianSet` is uninitialized in the implementation contract and, thus, its length equals zero.

```
if(guardianSet.keys.length == 0){
    return (false, "invalid guardian set");
}
```

Figure 16.2: ethereum/contracts/Messages.sol#L39-L41

The following modifier, `onlyProxy`, can be used to prevent direct function calls to the implementation contract, instead requiring that a proxy `delegatecall`s the given function. This can be used to prevent reachable `delegatecall`s in an implementation contract from executing in their own context.

```
modifier onlyProxy() {
    require(address(this) != __self, "Function must be called through
delegatecall");
    require(_getImplementation() == __self, "Function must be called through active
proxy");
    _;
}
```

Figure 16.3: contracts/proxy/utils/UUPSUpgradeable.sol#L31-L35

Exploit Scenario

Alice, a Wormhole developer, does not realize that her modifications to the contract allow a code path with `delegatecall`—`submitContractUpgrade`—to be reached on the implementation contract, and an attacker exploits the vulnerability.

Recommendations

Short term, implement an `onlyProxy` modifier and add it to the functions in the implementation contract that should not be called directly, namely, those that perform `delegatecalls`, such as `submitContractUpgrade`.

Long term, stay up to date with best practices and keep dependencies current.

Summary of Recommendations

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

- Improve the token bridge documentation for each of the chains on which Wormhole is deployed. Develop detailed documentation similar to the “Solana signature verification: A deep dive” document that was shared with us (TOB-WORM-6).
- Make the unit tests easier to run. Ideally, a developer should be able to run a contract’s unit tests with just one or two commands and without the need of privileged software, such as Docker (TOB-WORM-1).
- Develop a process for keeping dependencies up to date (TOB-WORM-7).
- Use static analysis tools such as Clippy to enforce that checked arithmetic is used and is used correctly (TOB-WORM-4, TOB-WORM-15).
- Incorporate more advanced testing techniques, such as property testing and fuzz testing, to help reveal bugs like TOB-WORM-14.

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.

- In the following line, “Womrhole” should be “Wormhole”:

  `After you have the dependencies installed and the chains running, you can run Womrhole.`

- In the following lines, sh should be bash, since the scripts use Bash-specific features (e.g., `set -euo pipefail`):

```
        "wormhole": "npm run setup && sh wormhole.sh",
        "evm": "npm run setup && sh evm.sh",
        "solana": "npm run setup && sh solana.sh",
```

- There is considerable code duplication between the definitions of `check_integrity` in `bridge/program/src/api/post_vaa.rs` and `serialize_vaa` in `bridge/program/src/instruction.rs`. We recommend that the common code be consolidated:

```
fn check_integrity<'r>(
    vaa: &PostVAAData,
    signatures: &SignatureSet<'r, { AccountState::Initialized }>,
) -> Result<()> {
    // Serialize the VAA body into an array of bytes.
    let body = {
        let mut v = Cursor::new(Vec::new());
        v.write_u32::<BigEndian>(vaa.timestamp)?;
        v.write_u32::<BigEndian>(vaa.nonce)?;
        v.write_u16::<BigEndian>(vaa.emitter_chain)?;
        v.write_all(&vaa.emitter_address)?;
        v.write_u64::<BigEndian>(vaa.sequence)?;
        v.write_u8(vaa.consistency_level)?;
        v.write_all(&vaa.payload)?;
        v.into_inner()
    };
...
pub fn serialize_vaa(vaa: &PostVAAData) -> Vec<u8> {
    let mut v = Cursor::new(Vec::new());
    v.write_u32::<BigEndian>(vaa.timestamp).unwrap();
    v.write_u32::<BigEndian>(vaa.nonce).unwrap();
    v.write_u16::<BigEndian>(vaa.emitter_chain).unwrap();
    v.write_all(&vaa.emitter_address).unwrap();
    v.write_u64::<BigEndian>(vaa.sequence).unwrap();
    v.write_u8(vaa.consistency_level).unwrap();
    v.write_all(&vaa.payload).unwrap();
    v.into_inner()
```

```
}
```

- The following comment appears twice in `peel.rs`. Since there are multiple base cases, "This is *a* structural recursion base case..." would be more clear:

```
/// This is our structural recursion base case, the trait system will stop
generating new nested
/// calls here.
```

- Adding a newline before `#[cfg(not(feature = "cpi"))]` will make the formatting consistent with other parts of `posted_vaa.rs`:

```
}
#[cfg(not(feature = "cpi"))]
```

- `complete_native` and `complete_native_with_payload` should use the named constant `CHAIN_ID_SOLANA` rather than comparing `vaa.token_chain` to 1:

```
if accs.vaa.token_chain != 1 {
    return Err(InvalidChain.into());
}
```

- The CosmWasm code does not appear to be formatted with `rustfmt`, which we recommend. The code does have a `rustfmt.toml` file; however, it appears to use unstable features, and we could not identify a compiler version that produces the code in its current format.

- In the following line, "querdian" should be "guardian":

```
pub addresses: Vec<GuardianAddress>, // List of querdian addresses
```

- In two places in `cosmwasm/contracts/wormhole/src/state.rs`, and in one place in `cosmwasm/contracts/wormhole/src/msg.rs`, "expirity" should be "expiry":

```
pub guardian_set_expirity: u64,
```

- In the following two locations in `cosmwasm/contracts/wormhole/src/state.rs`, the code could be rewritten to use `unwrap_or`:

```
    .or::<u64>(Ok(0))
    .unwrap()

    .or::<bool>(Ok(false))
    .unwrap()
```

- In the following line, "arbitarily" should be "arbitrarily" and "byes" should be "bytes":

```
/// Since denom names can be arbitarily long, and CW20 addresses are 32 byes,
we
```

- In the following line, “layout of is” should be “layout is”:

  ```
  /// In the first case (native tokens), the layout of is the following:
  ```

- Adding a newline before `WithdrawTokens` will make the formatting consistent with other parts of the `ExecuteMsg` declaration:

  ```
     DepositTokens {},
     WithdrawTokens {
         asset: AssetInfo,
     },
  ```

- The term “native” is used inconsistently in the CosmWasm token bridge source code. The following comment in `token_address.rs` suggests that "native" tokens are "token[s] managed by the Bank cosmos module" and "CW20 token[s]":

  ```
  /// Represent the external view of a token address.
  /// This is the value that goes into the VAA.
  ///
  /// When given an external 32 byte address, there are 3 options:
  /// I. This is a token native to this chain
  ///     a. it's a token managed by the Bank cosmos module
  ///     (e.g. the staking denom "uluna" on Terra)
  ///     b. it's a CW20 token
  /// II. This is a token address from another chain
  ...
  /// In the first case (native tokens), the layout of is the following:
  ```

  However, within `contract.rs`, `handle_complete_transfer_token_native` handles Bank-managed tokens, and `handle_complete_transfer_token` handles `ContractId::ForeignToken` and `ContractId::NativeCW20`.

D. Investigation of TOB-WORM-5

`test-fuzz` is a collection of Rust macros and a Cargo subcommand that automates certain fuzzing-related tasks, such as generating a fuzzing corpus and implementing a fuzzing harness. This appendix explains how we used `test-fuzz` to reveal serialization collisions between the Wormhole Solana account types (i.e., to aid in understanding the impact of finding TOB-WORM-5).

Approach

We implemented a function called `deserialize` (figure D.2) to check whether a given byte sequence deserializes as multiple Wormhole Solana account types. We used `test-fuzz` to automatically generate a fuzz harness for this function.

`test-fuzz` can record data passed to a function during tests. Hence, we tested the `deserialize` function by serializing the default value of each Wormhole Solana account type and passing the resulting bytes to the `deserialize` function. In this way, we generated a fuzzing corpus for `deserialize`.

Finally, we fuzzed `deserialize` for several hours on a single CPU. The results are described in the next section.

Results

Our fuzzing efforts revealed collisions between `PoolData` and `SignatureSetData` and between `GuardianSetData` and `PostedVAAData`.

In each of the observed collisions between `PoolData` and `SignatureSetData`, the signature set size was 24. Our understanding is that the signature set size is 19 in practice. Hence, we added this condition to the fuzzer. After requiring that the signature set size be 19, we no longer observed collisions between `PoolData` and `SignatureSetData`.

The collisions between `GuardianSetData` and `PostedVAAData` are more concerning because `PostedVAAData` is largely user controlled. Our fuzzer revealed two collisions. For those cases, we included the account contents below. Understanding whether such a collision could occur in practice requires further investigation.

Collision 1

- `GuardianSetData { index: 6381942, keys: [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 232, 3], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 226, 255, 255], [255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], creation_time: 16777216, expiration_time: 0 }`

- PostedVAAData { message: MessageData { vaa_version: 0, consistency_level: 4, vaa_time: 0, vaa_signature_account: 11111111111111111111111111111111, submission_time: 0, nonce: 256000, sequence: 0, emitter_chain: 0, emitter_address: [0, 0, 0, 0, 0, 0, 226, 255, 255, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], payload: [0] } }

- Bytes: [118, 97, 97, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 232, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 226, 255, 255, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]

Collision 2

- GuardianSetData { index: 6381942, keys: [[0, 0, 118, 97, 97, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 226, 0, 0, 0, 0, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 226, 128, 155, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0], [0, 60, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]], creation_time: 16843009, expiration_time: 0 }

- PostedVAAData { message: MessageData { vaa_version: 0, consistency_level: 4, vaa_time: 0, vaa_signature_account: 12ohP4fk79DwV8BNuMeH3wwcCHuQR3FaCKYkB7XYoxyu, submission_time: 0, nonce: 0, sequence: 71947096910987264, emitter_chain: 0, emitter_address: [0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 60, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1], payload: [0] } }

- Bytes: [118, 97, 97, 0, 4, 0, 0, 0, 0, 0, 118, 97, 97, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 226, 0, 0, 0, 0, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 226, 128, 155, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 60, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0]

Usage

To use the files in figures D.1 and D.2, do the following:

1. Put the files in a `solana/fuzzing` directory and add the directory to Wormhole's Solana workspace.

2. Add `Debug` as a derived trait to all of the account types referenced in figure D.2.

3. In the `solana/fuzzing` directory, run this command:

```
BRIDGE_ADDRESS='' EMITTER_ADDRESS='' TOKEN_BRIDGE_ADDRESS='' cargo
test
```

4. In the `solana/fuzzing` directory, run this command:

```
BRIDGE_ADDRESS='' EMITTER_ADDRESS='' TOKEN_BRIDGE_ADDRESS='' cargo
test-fuzz
```

As mentioned above, after running for a few hours, the fuzzer should reveal (at least) a collision between `GuardianSetData` and `PostedVAAData`.

```
[package]
name = "fuzzing"
version = "0.1.0"
edition = "2021"

[dependencies]
borsh = "=0.9.1"
paste = "1.0.8"
serde = "1.0.143"
test-fuzz = "3.0.2"

nft-bridge = { path = "../modules/nft_bridge/program", features = ["no-entrypoint",
"cpi"] }
token-bridge = { path = "../modules/token_bridge/program", features =
["no-entrypoint", "cpi"] }
wormhole-bridge-solana = { path = "../bridge/program", features = ["no-entrypoint",
"cpi"] }
wormhole-migration = { path = "../migration", features = ["no-entrypoint", "cpi"] }
```

Figure D.1: `fuzzing/Cargo.toml`

```
use borsh::de::BorshDeserialize;
use paste::paste;

use bridge::accounts::{
    BridgeData,
    ClaimData,
    GuardianSetData,
    PostedVAAData,
    SequenceTracker,
    SignatureSetData,
};
use token_bridge::types::{
    Config,
    EndpointRegistration,
    WrappedMeta,
};
use wormhole_migration::types::{
    PoolData,
    SplAccount,
```

```
    SplMint,
};

macro_rules! test_deserialize {
    ($ty:ty) => {
        paste! {
            #[test]
            fn [< deserialize_ $ty:snake >]() {
                deserialize(&borsh::to_vec(&<$ty>::default()).unwrap());
            }
        }
    };
}

test_deserialize! { BridgeData }
test_deserialize! { ClaimData }
test_deserialize! { GuardianSetData }
test_deserialize! { PostedVAAData }
test_deserialize! { SequenceTracker }
test_deserialize! { SignatureSetData }
test_deserialize! { Config }
test_deserialize! { EndpointRegistration }
test_deserialize! { WrappedMeta }
test_deserialize! { PoolData }
test_deserialize! { SplAccount }
test_deserialize! { SplMint }

macro_rules! try_deserialize {
    ($n:expr, $data:expr, $ty:ty) => {
        if let Ok(account) = <$ty>::try_from_slice($data) {
            eprintln!("{:?}", account);
            $n += 1
        };
    };
}

#[allow(unused)]
#[test_fuzz::test_fuzz]
fn deserialize(data: &[u8]) {
    // smoelius: Without the next condition, one gets a collision between `PoolData`
    // and `SignatureSetData`.
    if let Ok(signature_set) = SignatureSetData::try_from_slice(data) {
        if signature_set.signatures.len() != 19 {
            return;
        }
    }
    let mut n = 0;
    try_deserialize!(n, data, BridgeData);
    try_deserialize!(n, data, ClaimData);
    try_deserialize!(n, data, GuardianSetData);
    try_deserialize!(n, data, PostedVAAData);
    try_deserialize!(n, data, SequenceTracker);
    try_deserialize!(n, data, SignatureSetData);
```

```
    try_deserialize!(n, data, Config);
    try_deserialize!(n, data, EndpointRegistration);
    try_deserialize!(n, data, WrappedMeta);
    try_deserialize!(n, data, PoolData);
    try_deserialize!(n, data, SplAccount);
    try_deserialize!(n, data, SplMint);
    assert!(n <= 1, "{:?}", data);
}
```

Figure D.2: `fuzzing/src/lib.rs`

E. Responses and Links to Publicly Filed Issues

Wormhole has elected not to use its optional fix review from the project statement of work and will be instead sourcing fixes from its community via public GitHub issues. The following are Wormhole Foundation's responses and links to publicly filed issues for the findings in this report:

TOB-WORM-1: wormhole/issues/1528 and wormhole/issues/1529

TOB-WORM-2: wormhole/issues/1530

TOB-WORM-3: wormhole/issues/1531

TOB-WORM-4: wormhole/issues/1532

TOB-WORM-8: wormhole/issues/1533

TOB-WORM-9: wormhole/issues/1534

TOB-WORM-10: wormhole/issues/1535

TOB-WORM-11: *Improved in* wormhole/pull/1536

TOB-WORM-12: *We want to avoid the risk from the complexity of added security-critical functionality to the smart contract for upgrading bridge addresses. In case of a need to update bridge addresses, this can be accomplished through a contract upgrade in a governance action.*

TOB-WORM-13: *We acknowledge the finding, and will consider altering the control flow to accommodate the use case where a user wants to verify an already consumed VAA. Since this is heavily load-bearing code, we do not feel the additional use case justifies the risk at this time.*

TOB-WORM-14: wormhole/issues/1541

TOB-WORM-16: wormhole/issues/1539