Zellic

Wormhole Aptos

Smart Contract Security Assessment

November 29, 2022

Prepared for:

Wormhole Foundation

Prepared by:

Aaron Esau and Varun Verma

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted an audit for Wormhole Foundation from November 7th to November 29th, 2022.

Our general overview of the code is that it was very well-organized and structured. The code coverage is high, and tests are included for the majority of the functions. The documentation was good, although there were some small areas for improvement. The code was easy to comprehend, and in most cases, intuitive.

We applaud Wormhole Foundation for their attention to detail and diligence in maintaining incredibly high code quality standards in the development of Wormhole Aptos.

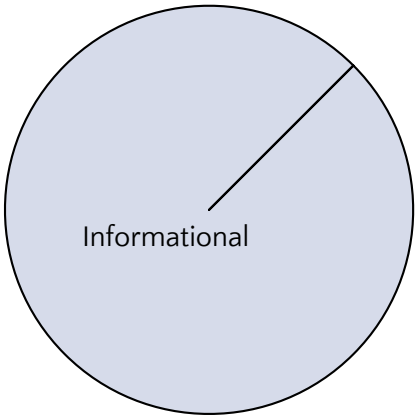
Zellic thoroughly reviewed the Wormhole Aptos codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

Specifically, taking into account Wormhole Aptos's threat model, we focused heavily on issues that would break core invariants. The issues focused upon were guardian signatures being the source of message authorization, the decentralized governance mechanism being the sole authority over the network and accurate reflections of token transfer amounts.

During our assessment on the scoped Wormhole Aptos contracts, we came across several discussion points. Fortunately, no critical issues were found.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	6



2 Introduction

2.1 About Wormhole Aptos

Wormhole is a communication bridge between Solana and other top decentralized finance (DeFi) networks. Existing projects, platforms, and communities are able to move tokenized assets seamlessly across blockchains and benefit from Solana's high speed and low cost.

Wormhole Aptos refers to the modules created to support communication with Aptos blockchains.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of open-source tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these shallow bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so forth as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, and so forth.

Complex integration risks. Several high-profile exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions and summarize the asso-

ciated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so forth.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so forth.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document rather than being ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their importance may differ. This varies based on numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Wormhole Aptos Modules

Repository https://github.com/wormhole-foundation/wormhole/tree/dev.v2/aptos

Versions `a3c9de7d2d3aedff7ecd95a9bce465fd6124ad1c`

Programs

- deployer::deployer
- wormhole::u32
- wormhole::u16
- wormhole::serialize
- wormhole::test_serialize
- wormhole::external_address
- wormhole::external_address_test
- wormhole::keccak256
- wormhole::vaa
- wormhole::vaa_test
- wormhole::guardian_set_upgrade
- wormhole::guardian_set_upgrade_test
- wormhole::cursor
- wormhole::state
- wormhole::guardian_pubkey
- wormhole::guardian_pubkey_test
- wormhole::contract_upgrade
- wormhole::contract_upgrade_test
- wormhole::emitter
- wormhole::emitter_test
- wormhole::set
- wormhole::wormhole
- wormhole::wormhole_test
- wormhole::deserialize
- wormhole::deserialize_test
- wormhole::u256
- wormhole::structs
- 0xf4f53cc591e5190eddbc43940746e2b5deea6e0e1562b2bba765d488504842c7::coin
- core_messages::sender
- core_messages::sender_test
- token_bridge::deploy_coin
- token_bridge::complete_transfer_with_payload
- token_bridge::complete_transfer_with_payload_test
- token_bridge::register_chain
- token_bridge::register_chain_test

Modules continued

- token_bridge::wrapped
- token_bridge::wrapped_test
- token_bridge::attest_token
- token_bridge::attest_token_test
- token_bridge::token_hash
- token_bridge::token_hash_test
- token_bridge::token_bridge
- token_bridge::vaa
- token_bridge::vaa_test
- token_bridge::state
- token_bridge::transfer_result
- token_bridge::transfer_with_payload
- token_bridge::asset_meta
- token_bridge::transfer
- token_bridge::transfer_test
- token_bridge::contract_upgrade
- token_bridge::contract_upgrade_test
- token_bridge::transfer_tokens
- token_bridge::transfer_tokens_test
- token_bridge::normalized_amount
- token_bridge::normalized_amount_test
- token_bridge::string32
- token_bridge::string32_test
- token_bridge::complete_transfer
- token_bridge::complete_transfer_test

Type Move

Platform Aptos

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of five person-weeks. The assessment was conducted over the course of four calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Aaron Esau, Engineer
aaron@zellic.io

Varun Verma, Engineer
varun@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

November 3, 2022	Start of primary review period
November 29, 2022	End of primary review period

3 Discussion

3.1 Transfers can be stuck indefinitely

As the Wormhole Foundation noted in their documentation:

> There is no guarantee for completion of transfers. If a user initiates a transfer and doesn't call `completeTransfer` on the target chain, a transfer might not be completed. In case a guardian set change happens in-between and the original signer guardian set expires, the transfer will be stuck indefinitely.

3.2 Wrapped CoinType can be created using Aptos VAA

Note that anyone may use a VAA emitted from an Aptos source chain to create a wrapped CoinType using the `token_bridge::wrapped::create_wrapped_coin_type` function.

However, there is no security impact to this because users cannot then register the `CoinType` using the `token_bridge::wrapped::create_wrapped_coin` function; it calls `token_bridge::wrapped::init_wrapped_coin` which calls `token_bridge::state::setup_wrapped`, which has the following assertion preventing the VAA source chain from being the current chain:

```
assert!(origin_info.token_chain ≠ state::get_chain_id(),
    E_WRAPPING_NATIVE_COIN);
```

As a side note, we verified that the VAAs cannot be replayed in `create_wrapped_coin_type` using the following test:

```
#[test(deployer=@deployer)]
fun test_create_wrapped_coin_2(deployer: &signer) {
    setup(deployer);
    register_chain::submit_vaa(ETHEREUM_TOKEN_REG);
    wrapped::create_wrapped_coin_type(ATTESTATION_VAA);
    wrapped::create_wrapped_coin_type(ATTESTATION_VAA);
}
```

The above test aborts with the code `ERESOURCE_ACCCOUNT_EXISTS`.

3.3 Initial guardian's length not checked

The protocol's initial guardians are set within the `init_internal` function, however there is no check that the length of this vector is 19, the total amount of guardians within the network.

Within the `initial_guardians` function is this function call

```
state::store_guardian_set(
    create_guardian_set(
        u32::from_u64(0),
        initial_guardians
    )
);
```

that stores the initial guardians. A check that the length of `initial_guardians` is 19 could be stronger from a security standpoint to protect against any configuration errors.

3.4 Use of custom serialization libraries

Aptos has a library built-in for serializing and deserializing many integer types. We recommend using `std::bcs::to_bytes<T>( ... )` and `aptos_std::from_bcs::from_bytes<T>( ... )` when possible.

A custom serialization and deserialization library was used because Wormhole's number representations are big endian, and bcs, the serialization format of Aptos, is little endian.

To further confidence in the serialization & deserialization used within the code, we successfully ran pseudo-fuzz tests over the `ExternalAddress`, `U16`, `U32`, & `U256` datatypes. One example is shown below.

```
#[test]
public fun test_address_deserialization() {
    let serialized_address_vec: vector<u8> = vector::empty();
    let i = 0u64;
    let randomization_seed = 0x47447;
```

```
    while (i < 1000) {
        // already 32 bytes
        let random_vec =
    hash::sha3_256(std::bcs::to_bytes<u64>(&(i+randomization_seed)));
        // of type ExternalAddress
        let external_address = left_pad(&random_vec);
        serialize(&mut serialized_address_vec, external_address);
        let cursor_vec = cursor::init<u8>(serialized_address_vec);
        // of type ExternalAddress
        let deserialized_external_address = deserialize(&mut cursor_vec);
        // confirm the initial generated external address
        assert!(deserialized_external_address == external_address, 1);
        //
    debug::print<vector<u8>>(&get_bytes(&deserialized_external_address));
        // debug::print<vector<u8>>(&get_bytes(&external_address));
        i = i + 1;
        // destroy cursor since it does not have drop ability
        cursor::rest(cursor_vec);
        serialized_address_vec = vector::empty();
    };
}
```

3.5 Integrate native data-types when released

Wormhole provides custom implementations for the U16, U32 and U256 data-types. However, these data types have been integrated into the move language but not into Aptos yet. When support is added for Aptos, we suggest migrating to them instead.

3.6 Misleading name of `get_external_address` function

The `emitter::get_external_address` function's name is misleading; it does not return an address, but rather, a unique ID encoded as a u64.

4 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm. Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

4.1 Module: `wormhole::vaa`

Function: `parse_and_verify()`

Intended behavior:

Parses and verifies the signatures of a VAA. Because this is the only public function that returns a VAA, this ensures that if an external module receives a VAA, it has been verified

Branches and code coverage:

Intended branches:

- VAA has been signed by the proper amount of guardians
  - ☑ Test coverage

Negative behavior:

- VAA has been signed, but with a wrong signature
  - ☑ Negative test?
- VAA has been double signed, therefore should not pass
  - ☑ Negative test?
- VAA has not been signed by the proper amount of guardians
  - ☑ Negative test?

Preconditions:

- *WormholeState* is initialized

Inputs:

- `bytes: vector<u8>`:

- **Control**: None
- **Checks**: Guardian signatures exist
- **Impact**: Only bytes which have been cryptographically verified to have been signed by the guardian set are acceptable inputs

Function call analysis

- `verify()`
  - **What is controllable?** The VAA is controllable, assuming its been signed
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, VAA cannot be verified, and therefore the message cannot be received
- `state::get_guardian_set()`
  - **What is controllable?** Nothing, function param comes from VAA
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, VAA's cannot be verified
- `parse()`
  - **What is controllable?** The bytes to parse
  - **If return value controllable, how is it used and how can it go wrong?** The return value is used to be the VAA, and it can go wrong if somehow a VAA is parsed & returned despite it not having enough signatures
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, nothing of consequence happens. It while loops through the number of signatures which could be problematic if the number of signatures is somehow incorrectly serialized to a super high number, in which case an out of gas error would occur.

4.2 Module: `wormhole::contract_upgrade`

Function: `upgrade()`

Intended behavior:

Upgrades the package metadata & code of the address retrieved by state::wormhole_signer(), which is *@wormhole*

Branches and code coverage:

Intended branches:

- Upgrade code upon proper authorization
  - ☐ Test coverage

Negative behavior:

- Reject code upgrade upon improper authorization
  - ☐ Negative test?

Preconditions:

- UpgradeAuthorized exists in global storage
- The code & metadata serialized corresponds to the upgrade hash from `move_from<UpgradeAuthorized>(@wormhole)` so that the code upgrade is authorized

Inputs:

- `code`:
  - **Control**: None
  - **Checks**: keccak256 hash of metadata_serialized & keccak256 of code equals the hash retrieved from global_storage
  - **Impact**: The code supplied to be used for the upgrade is authorized
- `metadata_serialized`:
  - **Control**: None
  - **Checks**: keccak256 hash of metadata_serialized & keccak256 of code equals the hash retrieved from global_storage
  - **Impact**: The metadata_serialized supplied to be used for the upgrade is authorized

Function call analysis

- `publish_package_txn()`
  - **What is controllable?** metadata_serialized, code
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Code is not upgraded

4.3 Module: `wormhole::guardian_set_upgrade`

Function: `submit_vaa_entry()`

Intended behavior:

Submit a VAA that authorizes a guardian set upgrade

Branches and code coverage:

Intended branches:

- VAA is valid & guardian set is upgraded
  - ☑ Test coverage

Negative behavior:

- The VAA is replayed
  - ☐ Negative test?
- The VAA emitter address and chain is not the governance address and therefore ignored
  - ☐ Negative test?

Preconditions:

- Guardian set is active/declared
- *WormholeState* is initialized

Inputs:

- `vaa`:
  - **Control**: None
  - **Checks**: The VAA has been signed by the current guardian set, it is targeting the latest guardian set index, emitter chain & address and is not being replayed
  - **Impact**: Only authorized changes can be made to the guardian set

Function call analysis

- `submit_vaa()`
  - **What is controllable?** Nothing, just passes the value of the function param passed to the entry function
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Guardian Set Upgrade does not happen under the revert case, no weird

control flow

- `parse_and_verify()`
  - **What is controllable?** Nothing, just passes the value of the function param passed to the entry function
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable as the VAA must have guardian signatures indicating its valid, and this could go wrong if it was possible to obtain guardian signatures without the guardian themselves providing them
  - **What happens if it reverts, reenters, or does other unusual control flow?** Guardian Set Upgrade does not happen under the revert case, no weird control flow
- `parse_payload()`
  - **What is controllable?** Not controllable, payload comes from authorized VAA
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable. Payload is used to provide information regarding the guardian set upgrade and can go wrong if it parsed incorrectly, or was somehow able to get through without the proper amount of guardian signatures
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, guardian set is not upgraded, cannot be reentered & no unusual control flow
- `replay_protect()`
  - **What is controllable?** VAA, which has undergone authorization
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts (hash already been used), no guardian set upgrade takes place, reentrancy not possible and no unusual control
- `assert_governance()`
  - **What is controllable?** VAA, which has undergone authorization
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, guardian set is not upgraded, reentrancy not possible & no unusual control flow
- `do_upgrade()`
  - **What is controllable?** Payload of the VAA, which has undergone authorization
  - **If return value controllable, how is it used and how can it go wrong?** No return value

- **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, guardian set is not upgraded, reentrancy not possible and no unusual control flow

4.4 Module: `wormhole::wormhole`

Function: `init()`

Intended behavior:

Initializes the contract. Note that this function takes additional arguments, so the native init_module function (which takes no arguments) cannot be used. Can only be called by the deployer (checked by the `deployer::claim_signer_capability` function).

Branches and code coverage:

Intended branches:

- Initializes the contract
  - ☑ Test coverage

Negative behavior:

- Cannot be called by someone who is not the dictated by the deployer designated in deployer.move
  - ☐ Negative test?
- The VAA emitter address and chain is not the governance address and therefore ignored
  - ☐ Negative test?

Preconditions:

- Has not been called before, otherwise resources will already exist at the signer's address

Inputs:

- `initial_guardians`:
  - **Control**: None
  - **Checks**: None
  - **Impact**: Any initial_guardians can be supplied, potential impact be could be wrong configuration
- `governance_contract`:

  - **Control**: None
  - **Checks**: None
  - **Impact**: Any governance_contract can be inputted, potential impact could be wrong configuration
- `deployer`:
  - **Control**: None
  - **Checks**: caller_addr == deployer || caller_addr == resource,
  - **Impact**: Only authorized individuals can initialize the contract
- `governance_chainid`:
  - **Control**: None
  - **Checks**: None
  - **Impact**: Any possible governance_chainid of type U16 can be inputted. Potential impact could be wrong configuration
- `chain_id`:
  - **Control**: None
  - **Checks**: None
  - **Impact**: Any possible chain_id of type U16 can be inputted. Potential impact could be wrong configuration

Function call analysis

- `init_internal()`
  - **What is controllable?** chain_id: u64, governance_chain_id: u64, governance_contract, initial_guardians
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Contract is not initialized if it reverts, no unusual control flow
- `init_wormhole_state()`
  - **What is controllable?**chain_id: u64, governance_chain_id: u64, governance_contract, initial_guardians, message_fee
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Contract is not initialized if it reverts, no unusual control flow
- `init_message_handles()`
  - **What is controllable?** Nothing, signer is retrieved from a capability, which enforces authorization
  - **If return value controllable, how is it used and how can it go wrong?** No return value

  - **What happens if it reverts, reenters, or does other unusual control flow?** Contract is not initialized if it reverts, no unusual control flow. Would only revert if the resources already exist within the account
- `store_guardian_set()`
  - **What is controllable?** The initial guardians
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Contract is not initialized if it reverts. No unusual control flow. Would only revert if the guardians already exist at the 0th index, an improbable scenario to occur

4.5 Module: `wormhole::wormhole`

Function: `publish_message()`

Intended behavior:

Publishes a message that emits an event that signals to the guardian chain that an action has occurred on the Aptos chain

Branches and code coverage:

Intended branches:

- State publishes the event
  - ☑ Test coverage: (init_internal tested but not init, so some logic missing testing)

Negative behavior:

- Insufficient fee, message not published
  - ☑ Negative test?

Preconditions:

- WormholeState is initialized for the @wormhole address

Inputs:

- `emitter_cap`:
  - **Control**: None, anyone can state::register_emitter() to get an emitter_cap
  - **Checks**: None

  - **Impact**: Anyone can publish a message
- `nonce`:
  - **Control**: None
  - **Checks**: None
  - **Impact**: Any nonce can be supplied, is nonce of any importance?
- `payload`:
  - **Control**: None
  - **Checks**: None,
  - **Impact**: Any payload can be sent cross-chain
- `message_fee`:
  - **Control**: greater than expected fee
  - **Checks**: None
  - **Impact**: Users have to pay for cross chain messaging

Function call analysis

- `state::get_message_fee()`
  - **What is controllable?** n/a
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable, could only go wrong if message fee on Wormhole state was somehow excessively low or excessively high
  - **What happens if it reverts, reenters, or does other unusual control flow?** Should never revert as long as WormholeState is initialized
- `emitter::use_sequence()`
  - **What is controllable?** The emitter capability
  - **If return value controllable, how is it used and how can it go wrong?** Used to denote the sequence number, no apparent case in which it can go wrong
  - **What happens if it reverts, reenters, or does other unusual control flow?** Should not revert under any circumstance as it simply increases the sequence number by 1
- `state::publish_event()`
  - **What is controllable?** Sequence, nonce, payload and emitter capability
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Message not sent under revert scenario, but function should never revert as long as the WormholeMessageHandle state exists on @wormhole

4.6 Module: `tokenbridge::attest_token`

Function: `attest_token_entry<CoinType>`

Intended behavior:

Produce a `AssetMeta` message for a given token

Branches and code coverage:

Intended branches:

- Message is published after token is attested
  ☑ Test coverage

Negative behavior:

- Message is not published if token is not a wrapped asset or coin is not initialized
  ☑ Negative test?

Preconditions:

- Asset is not wrapped

Inputs:

- `signer`:
  - **Control**: Can't mimic signer, User owns private key corresponding to signer's address
  - **Authorization**: N/A
  - **Impact**: Any user can attest a token

External call analysis

- `attest_token_internal`
  - **What is controllable?** CoinType generic
  - **If return value controllable, how is it used and how can it go wrong?** It's used to denote information about the CoinType, AssetMeta, and could possibly go wrong if CoinType information was retrieved incorrectly, however that does not appear the case.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Token is not attested under revert condition.
- `state::publish_message()`
  - **What is controllable?** Nonce, payload, fee coins

  - **If return value controllable, how is it used and how can it go wrong?** N/A, no return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Message is not published under revert condition, therefore token is not attested
- `asset_meta::encode()`
  - **What is controllable?** asset_meta, via the CoinType generic
  - **If return value controllable, how is it used and how can it go wrong?** Can go wrong if this information is encoded incorrectly, for example an error in the String32 or token_hash library, however that does not appear to be the case.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Token is not attested under revert condition.

4.7 Module: `tokenbridge::transfer_tokens`

Function: `transfer_tokens<CoinType>`

Intended behavior:

Transfer coins from Aptos to produce a wrapped version on another chain to the recipient address

Branches and code coverage:

Intended branches:

- The message to transfer coins cross chain is published
  ☑ Test coverage

Negative behavior:

- Message is not published because the relayer fee is too high
  ☑ Negative test?
- Message is not transferred because the recipient chain or address is not correctly supplied
  ☐ Negative test?
- Message is not published because the fee provided isn't sufficient
  ☐ Negative test?
- Message not published because owner doesn't have enough of the coin to transfer
  ☐ Negative test?

Preconditions:

- Coin of CoinType is initialized
- init_token_bridge_state has been called

Inputs:

- `nonce`:
  - **Control**: None
  - **Authorization**: None
  - **Impact**: Any nonce can be supplied, which will be emitted with a WormholeMessage event
- `relayer_fee`:
  - **Control**: · coin::value<CoinType>(&coins)
  - **Authorization**: None
  - **Impact**: Excessively high fee will not be accepted by the protocol, good protection
- `recipient`:
  - **Control**: None
  - **Authorization**: None
  - **Impact**: Whatever recipient supplied will receive the coins
- `recipient_chain`:
  - **Control**: None
  - **Authorization**: None
  - **Impact**: Non valid recipient chain can be provided
- `coins`:
  - **Control**: None
  - **Authorization**: None
  - **Impact**: Any coin that user has registered for can be used on the bridge
- `wormhole_fee_coins`:
  - **Control**: · state::get_message_fee()
  - **Authorization**: None
  - **Impact**: Sufficient fee must be supplied

External call analysis

- `transfer_tokens_internal()`
  - **What is controllable?** The coins and relayer fee supplied
  - **If return value controllable, how is it used and how can it go wrong?** Its used to encode information about token_chain, token_address, normal-

ized_amount and it can go wrong if the information encoded is incorrect. The information encoded would be incorrect if token_bridge::state methods retrieved values incorrectly,
    - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts a transfer message is not published
- `transfer::create()`
    - **What is controllable?** nor

normalized_amount, token_address, token_chain, recipient, recipient_chain, normalized_relayer_fee

- **If return value controllable, how is it used and how can it go wrong?** Its used to create the transfer information and could go wrong with a transfer with different values then supplied
    - **What happens if it reverts, reenters, or does other unusual control flow?** Message does not get published, no unusual control flow
- `state::publish_message()`
    - **What is controllable?** The nonce, the encoded transfer, and the fee
    - **If return value controllable, how is it used and how can it go wrong?** No return value
    - **What happens if it reverts, reenters, or does other unusual control flow?** Message is not published under a revert condition, no unusual control flow

4.8 Module: `tokenbridge::wrapped`

Function: `create_wrapped_coin<CoinType>`

Intended behavior:

Created a wrapped coin on Aptos that represents the native asset on another chain

Branches and code coverage:

Intended branches:

- VAA is valid and wrapped coin is created
    - ☑ Test coverage

Negative behavior:

- Wrapped coin already exists
    - ☐ Negative test?

Preconditions:

- Chain ID exists as a registered emitter
- VAA can be parsed according to standard structure

Inputs:

- `vaa`:
  - **Control**: N/A
  - **Authorization**: Signed by guardians and not been used already
  - **Impact**: Only valid VAAs can create tokens

External call analysis

- `init_wrapped_coin()`
  - **What is controllable?** asset_meta, maybe coin signer?
  - **If return value controllable, how is it used and how can it go wrong?** N/A
  - **What happens if it reverts, reenters, or does other unusual control flow?** Wrapped coin is not created under revert, no unusual control flow
- `state::get_wrapped_asset_signer()`
  - **What is controllable?** Origin_info
  - **If return value controllable, how is it used and how can it go wrong?** It returns a signer and is not problematic because it passed to init_wrapped_coin in which the user cannot do anything malicious with it
  - **What happens if it reverts, reenters, or does other unusual control flow?** Wrapped coin is not created, no unusual control flow
- `state::create_origin_info()`
  - **What is controllable?** Asset meta information extracted from VAA payload
  - **If return value controllable, how is it used and how can it go wrong?** Used to denote origin info
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A
- `token_bridge::parse_verify_and_replay_protect()`
  - **What is controllable?** VAA
  - **If return value controllable, how is it used and how can it go wrong?** Used to denote the VAA payload, can go wrong if VAA isn't signed by the valid guardians
  - **What happens if it reverts, reenters, or does other unusual control flow?** No unusual control flow, wrapped coin is not created under a revert scenario
- `asset_meta:parse()`

- **What is controllable?** A signed, valid VAA
- **If return value controllable, how is it used and how can it go wrong?** It's used to extract token information from the VAA and can go wrong if this information is parsed incorrectly
- **What happens if it reverts, reenters, or does other unusual control flow?** Wrapped coin is not created if revert occurs

4.9 Module: `tokenbridge::token_hash`

Function: `derive<CoinType>()`

Intended behavior:

Get the 32 token address of an arbitrary CoinType

Branches and code coverage:

Intended branches:

- Creates a token hash
  ☑ Test coverage

Negative behavior:

- Wrapped coin already exists
  ☑ N/A

Preconditions:

- CoinType is a registered CoinType.. but this is not checked

Inputs:

External call analysis

- `type_info::type_name<CoinType>()`
  - **What is controllable?** The generic, and is not restricted to a CoinType generic in specific
  - **If return value controllable, how is it used and how can it go wrong?** N/A
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A
- `hash::sha3_256()`
  - **What is controllable?** Origin_info

- **If return value controllable, how is it used and how can it go wrong?** It returns a signer, but cannot go wrong as its not used in a context where a user can obtain it
- **What happens if it reverts, reenters, or does other unusual control flow?** Token address is not returned under revert condition and no unusual control flow

4.10 Module: `tokenbridge::register_chain`

Function: `submit_vaa()`

Intended behavior:

Register a new chain to send tokens too

Branches and code coverage:

Intended branches:

- Chain is registered
  - ☑ Test coverage

Negative behavior:

- Fails if emitter is not the governance chain
  - ☑ Negative test?
- Chain has already been registered
  - ☑ Negative test?
- VAA is replayed and therefore rejected
  - ☑ Negative Test

Preconditions:

- State for @tokenbridge address is initialized

Inputs:

- `vaa`:
  - **Control**: N/A
  - **Authorization**: Signed by the guardians, sent by the governance, not a replay
  - **Impact**: Only authorized VAAs can register chains

External call analysis

- `state::set_registered_emitter()`
  - **What is controllable?** Emitter chain id, emitter chain address
  - **If return value controllable, how is it used and how can it go wrong?** N/A
  - **What happens if it reverts, reenters, or does other unusual control flow?** Only possible revert scenario is if state does not exist on @tokenbridge
- `parse_payload()`
  - **What is controllable?** The VAA
  - **If return value controllable, how is it used and how can it go wrong?** Used to denote information about the emitter_chain_id and the emitter_address
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts under the condition that the action is incorrect or the target chain is not 0, no unusual control flow
- `vaa::parse_and_verify()`
  - **What is controllable?** The VAA
  - **If return value controllable, how is it used and how can it go wrong?** Used to extract information from the VAA and can go wrong if the information is parsed incorrectly
  - **What happens if it reverts, reenters, or does other unusual control flow?** Chain is not registered
- `vaa::assert_governance()`
  - **What is controllable?** The VAA
  - **If return value controllable, how is it used and how can it go wrong?** N/A
  - **What happens if it reverts, reenters, or does other unusual control flow?** chain is not registered under revert condition, no unusual control flow

4.11 Module: `tokenbridge::vaa.move`

Function: `parse_and_verify()`

Intended behavior:

Parses & verifies a token bridge VAA and Aborts if the VAA is not from a known token bridge emitter

Branches and code coverage:

Intended branches:

- VAA is legitimate and is therefore returned by the function

- ☑ Test coverage

Negative behavior:

- Guardian set is the most current most guardian set
  - ☐ Negative test?
- VAA is a replay
  - ☑ Negative test?
- Signatures are incorrect
  - ☑ Negative test?
- Emitter is unknown
  - ☑ Negative Test

Preconditions:

- State for @tokenbridge address is initialized and so is *wormholestate* for @wormhole

Inputs:

- `bytes`:
  - **Control**: N/A
  - **Authorization**: Signed by the guardian set
  - **Impact**: Only authorized VAA's are returned

External call analysis

- `vaa:parse_and_verify()`
  - **What is controllable?** bytes
  - **If return value controllable, how is it used and how can it go wrong?** It's used to denote information about the VAA and can go wrong if the deserialization incorrectly parses information or an unauthorized VAA is somehow verified as legitimate.
  - **What happens if it reverts, reenters, or does other unusual control flow?** VAA is not verified under revert scenario, and no unusual control flow. Note that if that length of guardian signatures every increases to greater than 19, DoS may be of an issue
- `assert_known_emitter()`
  - **What is controllable?** The VAA
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?**

No unusual control flow, VAA is not authorized under revert case

4.12 Module: `tokenbridge::transfer_tokens_with_payload.move`

Function: `transfer_tokens_with_payload<CoinType>()`

Intended behavior:

Perform a cross chain a token transfer with the ability to send additional payload information as well

Branches and code coverage:

Intended branches:

- Message is published
  ☑ Test coverage

Negative behavior:

- Insufficient fees applied
  ☑ Negative test?

Preconditions:

- State for @tokenbridge address is initialized

Inputs:

- `wormhole_fee_coins`:
  - **Control**: Fee is greater or equal than the expected fee
  - **Authorization**: N/A
  - **Impact**: Adequate fee must be supplied to transmit message cross chain
- `coins`:
  - **Control**: Must fall into category of being either a wrapped or native asset
  - **Authorization**: N/A
  - **Impact**: Native assets are deposited into the bridge, wrapped assets are burned (and presumably released on the other side)
- `payload`:
  - **Control**: N/A
  - **Authorization**: N/A

  - **Impact**: Any payload can be sent cross chain, though maybe an upper bound on the size could be beneficial
- `nonce`:
  - **Control**: N/A
  - **Authorization**: N/A
  - **Impact**: Any nonce can be supplied
- `external_address`:
  - **Control**: · 32 bytes, as required by the module that instantiates an external address
  - **Authorization**: None
  - **Impact**: Funds can be sent to any external address requested
- `recipient_chain`:
  - **Control**: Any chain within range of a U16 can be used
  - **Authorization**: N/A
  - **Impact**: An unsupported chain could be requested
- `emitter_capability`:
  - **Control**: None, emitter capability can be acquired by anyone
  - **Authorization**: N/A
  - **Impact**: Must acquire emitter from wormhole

External call analysis

- `state::publish_message()`
  - **What is controllable?** Nonce, payload and fee
  - **If return value controllable, how is it used and how can it go wrong?** N/A, no return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Message is not delivered under revert condition, no unusual control flow
- `transfer_with_payload::encode()`
  - **What is controllable?** N/A, comes from the transfer result of transfer_tokens_internal
  - **If return value controllable, how is it used and how can it go wrong?** Can go wrong if the information was encoded incorrectly, which can occur if there is an error in the serialization functions, though that does not seem to be the case.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Message not delivered under revert condition, no unusual control flow
- `transfer_with_payload::create()`
  - **What is controllable?** N/A, uses the information extracted from transfer_tokens_internal
  - **If return value controllable, how is it used and how can it go wrong?** Used

to denote information about the transfer and can go wrong if it encodes incorrect information about the transfer
  - **What happens if it reverts, reenters, or does other unusual control flow?** Message not delivered under revert case, no unusual control flow. Though, function should never revert.
- `transfer_result::destroy()`
  - **What is controllable?** N/A, comes from the transfer result of transfer_tokens_internal
  - **If return value controllable, how is it used and how can it go wrong?** It's used to denote information about the transfer and can go wrong if the information was extracted incorrectly
  - **What happens if it reverts, reenters, or does other unusual control flow?** Message not delivered, no unusual control flow. Though, function should never revert.
- `transfer_tokens_internal()`
  - **What is controllable?** The coin to send
  - **If return value controllable, how is it used and how can it go wrong?** It's used to denote information about the transfer result and can go wrong if the information pertaining to the transfer result is not respective of the CoinType and amount of coins sent
  - **What happens if it reverts, reenters, or does other unusual control flow?** Cross chain token message is not delivered under revert scenario and no unusual control flow

4.13 Module: `tokenbridge::state.move`

Function: `set_vaa_consumed()`

Intended behavior:

Denote that a hash has been used to prevent replay attacks

Branches and code coverage:

Intended branches:

- Hash has not been replayed
  - ☐ Test coverage

Negative behavior:

- Hash has been replayed
  - ☐ Negative test?

Preconditions:

- State for @tokenbridge address is initialized

Inputs:

- `hash`:
  - **Control**: Not been replayed
  - **Authorization**: Signed by guardians
  - **Impact**: Duplicate messages cannot go through

External call analysis

- `set::add()`
  - **What is controllable?** Hash
  - **If return value controllable, how is it used and how can it go wrong?** N/A, no return value, but can only go wrong if somehow the state.consumed_vaas does not detect a replayed hash which can only happen if somehow the state resets.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No unusual control flow, vaa fails to be authorized under revert condition

4.14 Module: `tokenbridge::complete_transfer.move`

Function: `submit_vaa<CoinType>()`

Intended behavior:

Complete a cross-chain transfer, which in turn mints wrapped coins, withdraws native coins from the bridge & delivers the coins to the recipient on Aptos

Branches and code coverage:

Intended branches:

- Coins of varying degrees of decimals are supported
  ☑ Test coverage
- VAA is legitimate and the recipient receives their coins
  ☑ Test coverage

Negative behavior:

- VAA not signed by proper amount of guardians
  ☑ Negative test?

- VAA is a replay
  - ☑ Negative test?

Preconditions:

- State for @tokenbridge address is initialized

Inputs:

- `fee_recipient`:
  - **Control**: None
  - **Authorization**: None
  - **Impact**: Anyone can be the fee recipient
- `vaa`:
  - **Control**: Not been replayed
  - **Authorization**: Signed by guardians
  - **Impact**: Duplicate messages cannot go through, nor unauthorized ones

External call analysis

- `complete_transfer<CoinType>()`
  - **What is controllable?** CoinType and the parsed transfer information, to a degree
  - **If return value controllable, how is it used and how can it go wrong?** N/A, no return value, can only go wrong if the information is extracted incorrectly from the transfer event, which does not appear to be the case.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the cross-chain message is not received on Aptos. Should revert if the to_chain is not Aptos, a valid check.
- `transfer::parse()`
  - **What is controllable?** The VAA, though it must have been signed by the guardians
  - **If return value controllable, how is it used and how can it go wrong?** It's used to even further extract information about the transfer event and can go wrong if the deserialization is erroneous, however that does not appear the case.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the cross-chain message is not received on Aptos. Should revert if action is not 1, a valid check.
- `token_bridge::vaa::parse_verify_and_replay_protect()`
  - **What is controllable?** VAA

- **If return value controllable, how is it used and how can it go wrong?** VAA and its used to denote information about the transfer event. It can wrong if the information is parsed incorrectly, but the parsing looks correct.
- **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the cross-chain message is not received on Aptos. However, legitimate messages should not revert

—

4.15 Module: `tokenbridge::structs/transfer.move`

Function: `encode<CoinType>()`

Intended behavior:

Serialize a transfer struct to encode data about the transfer event, such as the amount, to_chain, fee etc.

Branches and code coverage:

Intended branches:

- Legitimate transfer struct is encoded
  ☑ Test coverage

Negative behavior:

- Illegitimate transfer struct aborts
  ☐ Negative test?

Preconditions:

- N/A

Inputs:

- `fee_recipient`:
  - **Control**: None
  - **Authorization**: None
  - **Impact**: Anyone can be the fee recipient
- `vaa`:
  - **Control**: Not been replayed
  - **Authorization**: Signed by guardians
  - **Impact**: Duplicate messages cannot go through, nor unauthorized ones

External call analysis

- `external_address:serialize()`
  - **What is controllable?** N/A
  - **If return value controllable, how is it used and how can it go wrong?** N/A cannot go wrong because all addresses are restricted to being · 32 bytes
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A
- `normalized_amount::serialize()`
  - **What is controllable?** N/A
  - **If return value controllable, how is it used and how can it go wrong?** N/A, cannot go wrong because normalized amount supports serialization of type u64
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A
- `serialize_u16()`
  - **What is controllable?** N/A
  - **If return value controllable, how is it used and how can it go wrong?** N/A cannot go wrong because type U16 is verified to be a uint · 65536
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A
- `serialize_u8()`
  - **What is controllable?** N/A
  - **If return value controllable, how is it used and how can it go wrong?** N/A, can't go wrong just does a vector::push_back on the u8
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A

4.16 Module: `tokenbridge::register_chain.move`

Function: `parse_payload()`

Intended behavior:

Parse a VAA payload to create RegisterChain struct that describes the emitter chain and emitter address

Branches and code coverage:

Intended branches:

- Payload is legitimate and RegisterChain struct is created
  ☑ Test coverage

Negative behavior:

- target module is not the token bridge
  ☐ Negative test?
- payload is not targeting chain 0
  ☐ Negative test?
- Payload is of the wrong action type (not 1)
  ☐ Negative test?

Preconditions:

- N/A

Inputs:

- `payload`:
  - **Control**: action information encoded in payload is 1, target chain is 0, target module is TOKEN_BRIDGE
  - **Authorization**: Signed by the guardians
  - **Impact**: Only VAAs that target registering chains are valid

External call analysis

- `deserialize_vector()`
  - **What is controllable?** the payload, though still has to be signed by guardians.
  - **If return value controllable, how is it used and how can it go wrong?** Return value indicates the target module, cannot go wrong because vector length is restricted to 32 bytes, and addresses are · 32 bytes
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A, vaa is not submitted
- `deserialize_u16()`
  - **What is controllable?** the payload, though still has to be signed by guardians.
  - **If return value controllable, how is it used and how can it go wrong?** Return value indicates the target chain, cannot go wrong because if the type isn't of type u16, the cursor::destroy_empty will fail as cur wont be empty
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A
- `external_address::deserialize()`
  - **What is controllable?** the payload, though still has to be signed by guardians.

- **If return value controllable, how is it used and how can it go wrong?** Return value indicates emitter address, cannot go wrong because the cursor::destroy_empty would abort if bytes were left over. Possible fail case would be if the external address was serialized incorrectly, however that does not appear to be the case.
- **What happens if it reverts, reenters, or does other unusual control flow?** N/A, the submission of the VAA to register a chain would not go through

4.17 Module: `deployer::deployer`

Function: `claim_signer_capability()`

Intended behavior:

Allow an authorized signer to acquire a *DeployingSignerCapability*

Branches and code coverage:

Intended branches:

- Signer capability is retrieved
  - ☐ Test coverage

Negative behavior:

- Caller address is not the resource or deployer
  - ☐ Negative test?

Preconditions:

- DeployingSignerCapability exists the address of resource

Inputs:

- `resource`:
  - **Control**: N/A
  - **Authorization**: Address corresponding to the signer is the resource or the deployer
  - **Impact**: Only authorized users can retrieve a DeployingSignerCapability
- `caller`:
  - **Control**: Can't mimic signer, User owns private key corresponding to signer's address
  - **Authorization**: Signer's address is the deployer or the resource

- **Impact**: Only authorized users can retrieve a DeployingSignerCapability

5 Audit Results

At the time of our audit, the code was deployed to mainnet.

During our audit, we came across six discussion point findings. Wormhole Foundation acknowledged all findings.

5.1 Disclaimers

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any additional code added to the assessed project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.