Wormhole Near

Audit

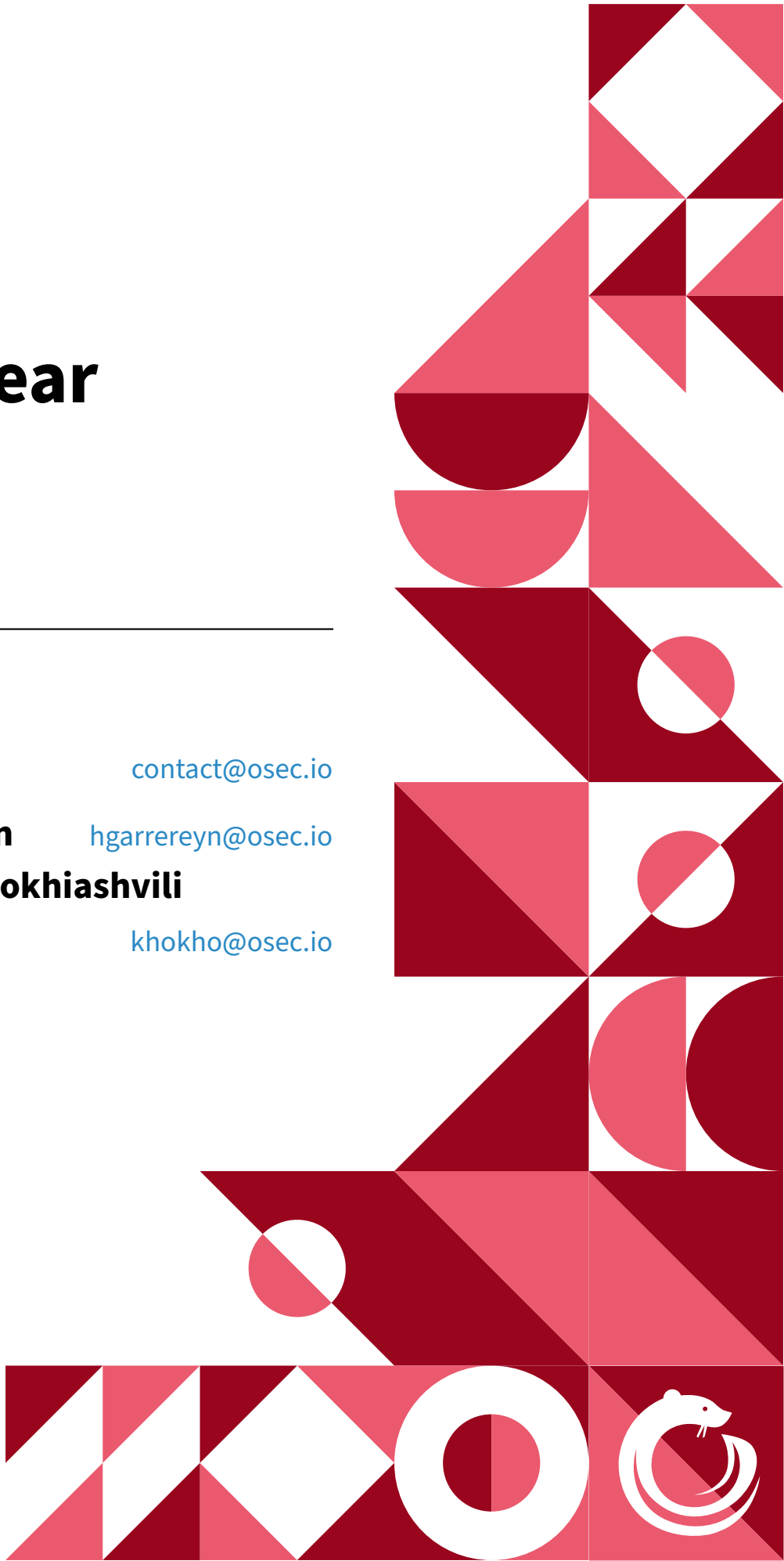
Presented by:

OtterSec contact@osec.io

Harrison Green hgarrereyn@osec.io

Aleksandre Khokhiashvili

khokho@osec.io



Contents

Appendices

01 | Executive Summary

Overview

Wormhole engaged OtterSec to perform an assessment of Wormhole Bridge for NEAR blockchain. This assessment was conducted between August 1st and August 26th, 2022.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. When delivering our audit report, we worked closely with the team over to streamline patches and confirm remediation.

We delivered final confirmation of the patches September 15th, 2022.

Key Findings

The following is a summary of the major findings in this audit.

- 14 findings total
- 2 vulnerabilities which could lead to loss of funds
  - OS-WHN-ADV-00: Taking wrapped nft token contract
  - OS-WHN-ADV-01: Marking `token-bridge` transfer event as completed incorrectly

As part of this audit, we also provided proofs of concept for each vulnerability to prove their exploitability. These scripts can be found at osec.io/pocs/wormhole-near. For a full list, see Appendix B.

02 | Scope

The source code was delivered to us in a git repository at github.com/certusone/wormhole. This audit was performed against commit `178a21d`.

There were a total of 6 programs included in this audit. A brief description of the programs is presented on the following page. A full list of program files and hashes can be found in Appendix A.

Name	Description
`wormhole`	The `wormhole` contract is used for publishing and verifying cross-chain messages(VAAs) made by contracts on different blockchains. Its functionality is fundamental and minimal: • Accept messages from apps deployed on NEAR blockchain for guardian network to observe and publish • Verify integrity of cross-chain messages submitted to the apps on NEAR blockchain • Execute administrative actions by means of guardian produced VAAs(e.g. upgrade, change fee...)
`token-bridge`	The `token-bridge` contract is used bridging tokens between different blockchains and NEAR. Tokens are transferred from one chain to another using a lockup/mint and burn/unlock mechanism. Functionality of the bridge on NEAR includes: • Lock NEAR or NEP141 tokens to publish VAA which can be used to mint wrapped tokens on target chain • Create new fungible token contract for each cross-chain asset. AKA wormhole token • Receive lock up VAA from bridge running on other blockchain and mint them in the form of wrapped NEP141 token • Burn wrapped wormhole tokens and publish it as a message for target chain • Verify burn VAAs for NEP141 and native tokens and release associated locked funds
`ft`	`ft` contract is the wrapped NEP141 form of token from some other blockchain. It gets deployed by the `token-bridge` per each cross-chain wormhole token. • Implement methods required by NEP141&NEP145 standard • Allow `token-bridge` to mint/burn wrapped tokens

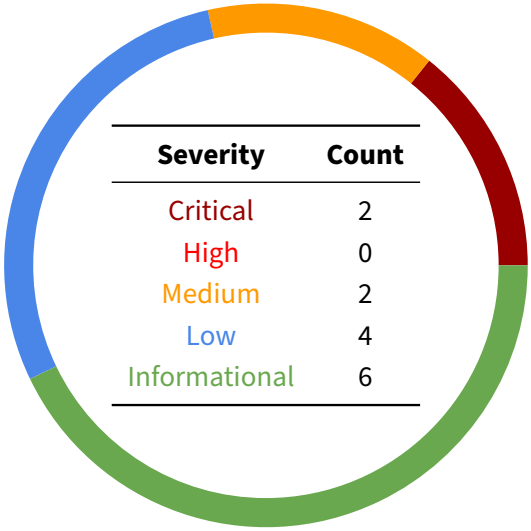
Name	Description
`nft-bridge`	Similar to `token-bridge` but for non fungible tokens & NEP-171. We didn't finish reviewing this contract since wormhole team asked us to remove it from scope during the audit.
`nft-wrapped`	Similar to `ft` but for non fungible tokens & NEP-171. We didn't finish reviewing this contract since wormhole team asked us to remove it from scope during the audit.
`watcher`	Off-chain program used by guardians to fetch published messages from NEAR blockchain. • Fetch all the VAAs emmited by `wormhole` contract • Ensure that all VAAs are final(i.e. can't be reverted)

03 | Findings

Overall, we report 14 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.

The below chart displays the findings by severity.



Severity	Count
Critical	2
High	0
Medium	2
Low	4
Informational	6

Proofs of Concept

For critical vulnerabilities we created a proof of concept to verity the attack ideas. The proof of concept directory structure can be found in Appendix B.

A GitHub repository containing these proofs of concept can be found at osec.io/pocs/wormhole-near.

To run a POC:

```sh
./run.sh <directory name>
```

For example,

```sh
./run.sh os-whn-adv-00
```

Each proof of concept comes with its own patch file which modifies the existing test framework to demonstrate the relevant vulnerability. We also recommend integrating these patches into the test suite to prevent regressions.

04 | Vulnerabilities

Here we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have **immediate** security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix E.

ID	Severity	Status	Description
OS-WHN-ADV-00	Critical	Resolved	When deploying `wrapped-nft` contract async race-condition leads to the ability to arbitrarily mind wrapped NFTs
OS-WHN-ADV-01	Critical	Resolved	token-bridge VAA can be marked as used without its intended action executing
OS-WHN-ADV-02	Medium	Resolved	It's possible to reset sequence number for any wormhole message emitter
OS-WHN-ADV-03	Medium	Resolved	Several occurrences of incorrect parsing for VAA payloads
OS-WHN-ADV-04	Low	Resolved	Missing storage deposit checks for contract update flow
OS-WHN-ADV-05	Low	Resolved	Rollback callback not checking success of correct promise
OS-WHN-ADV-06	Low	Resolved	In token-bridge `vaa_transfer`, the deposit is not refunded when `nfee == 0` and the VAA is handling native NEAR.
OS-WHN-ADV-07	Low	Resolved	In token-bridge `send_transfer_near`, fractional NEAR deposits are truncated and not refunded to the user.

OS-WHN-ADV-00 [crit] [resolved] | Race-Condition when deploying wrapped-nft contract

Description

We found that when `nft-bridge` contract is deploying `nft-wrapped` subcontract, the attacker can set their account as the authority for the newly created subcontract. This is possible because deploying and initialization is not done atomically.

In NEAR protocol, each promise can carry multiple actions and they will all be executed atomically. But if you have two promises chained using `.then(...)` function, those two promises will execute separately. In `nft-bridge`, `deploy_contract` happens in one `Promise` and new function call happens in another. Due to this, the attacker can call the new function during the small time period between `deploy_contract` and new receipts.

Since `nft-wrapped` uses `owner_id` passed in new call as the authority for mint and burn functions, this means that the attacker would be able to arbitrarily mint new wrapped tokens for the NFT subcontracts which they manage to take over.

Below is the promise chain used for deploying the contract. Atomicity breaks between the two highlighted lines.

nft-bridge/src/lib.rs RUST

```rust
Promise::new(bridge_token_account_id.clone())
    .create_account()
    .transfer(cost)
    .add_full_access_key(storage.owner_pk.clone())
    .deploy_contract(BRIDGE_NFT_BINARY.to_vec())
    // Lets initialize it with useful stuff
    .then(ext_nft_contract::ext(bridge_token_account_id.clone()).new(
        env::current_account_id(),
        ft,
        vaa.sequence,
    ))
    .then(
        ext_nft_contract::ext(bridge_token_account_id)
            .with_attached_deposit(dep)
            .nft_mint(token_id.clone(), recipient_account, md,
    ↪   refund_to.clone()),
    )
```

Remediation

Correct implementation of deploying and initialization can be found inside `token-bridge`:

nft-bridge/src/lib.rs RUST

```rust
Promise::new(asset_token_account.clone())
    .create_account()
    .transfer(cost)
    .add_full_access_key(storage.owner_pk.clone())
    .deploy_contract(BRIDGE_TOKEN_BINARY.to_vec())
    .function_call(
        "new".to_string(),
        serde_json::to_string(&new_args)
            .unwrap()
            .as_bytes()
            .to_vec(),
        0,
        Gas(10_000_000_000_000),
    )
```

`function_call` just ends up adding another action to already existing `ActionReceipt`. Since smallest unit of execution in NEAR protocol is receipt, this entire promise ends up being atomic.

Patch

The Wormhole team is aware of the issue and will fix it by doing initialization in an identical way to `token-bridge`, described above. `nft-bridge` has been taken out of scope soon after discovering this vulnerability so we've not verified the patches yet.

OS-WHN-ADV-01 [crit] [resolved] | Marking token-bridge VAA as used incorrectly

Description

In the `token-bridge` contract we found that an attacker is able to mark a VAA as used without executing its intended action supplied inside the VAA payload. One example of such an action is cross-chain token transfer. Due to the wormhole bridge design, anyone is able to relay signed VAAs from guardians to the `token-bridgen` contract. This means that by using this method, an attacker can invalidate other users' VAAs.

Due to NEAR's asynchronous runtime, when a VAA is submitted for execution it ends up being marked as used inside an `ActionReceipt` different from the one where the actual token transfer happens. You can see how a VAA's hash is marked as used before the promise is created inside the `submit_vaa_work` call.

token-bridge/src/lib.rs RUST

```
self.dups.insert(&pvaa.hash, &true);
self.submit_vaa_work(&pvaa, refund_to.unwrap())
```

token-bridge/src/lib.rs RUST

```
prom = ext_ft_contract::ext(account)
    .with_attached_deposit(1)
    .ft_transfer(mr, U128::from(namount), None);
```

Since these two are in different receipts it means that they don't execute atomically. As a result of this, **the second receipt can fail even if the first receipt was successful**. This was supposedly protected by the gas check at the beginning of `submit_vaa`.

token-bridge/src/lib.rs RUST

```
#[payable]
pub fn submit_vaa(
    &mut self,
    vaa: String,
    mut refund_to: Option<AccountId>,
) -> PromiseOrValue<bool> {
    if refund_to == None {
        refund_to = Some(env::predecessor_account_id());
    }
```

```
    if env::prepaid_gas() < Gas(150_000_000_000_000) {
        env::panic_str("NotEnoughGas");
    }
```

Under normal circumstances, this gas check is enough to make sure all receipts have enough gas. But we found that attacker is able to *inflate gas usage* of the first receipt, leaving very little for the following promises. This is possible to do even if attacker is submitting well-formed VAA generated by the normal use of wormhole.

Impact of this vulnerability is that attacker can take other users VAA which was supposed to transfer cross-chain tokens to some NEAR account and submit it in the malicious way described above. Since the VAA is going to be marked as used without the transfer succeeding this means that user won't be able to receive their tokens anymore.

Proof of Concept

We made a proof of concept which shows that such an attack is possible. For clarity, we created a small contract vulnerable with an identical vulnerability. You can see the core part of this contract below:

gas-limit-poc/src/lib.rs RUST

```rust
#[near_bindgen]
#[derive(Default, BorshDeserialize, BorshSerialize)]
pub struct Counter {
    val: bool,
}


#[ext_contract(ext_counter_contract)]
pub trait ExtCounter {
    fn bar(&mut self);
}

#[near_bindgen]
impl Counter {
    // Should be false if no promise is in flight!
    pub fn get_cur(&self) -> bool {
        self.val
    }

    pub fn foo(&mut self, vaa: String) {
        if env::prepaid_gas() < Gas(300_000_000_000_000) {
```

```
            env::panic_str("NotEnoughGas");
        }
        // Actual arguments are not allowed to be huge
        assert!(vaa.len() == 3);
        self.val = true;


        ext_counter_contract::ext(env::current_account_id()).bar();

        // Equivalent:
        // Promise::new(env::current_account_id())
        //     .function_call_weight(String::from("bar"), Vec::new(), 0,
    ↪  Gas(0), GasWeight::default());
    }

    #[private]
    pub fn bar(&mut self) {
        self.val=false;
    }
}
```

Our Proof of Concept is able to get the above contract into the state where `self.val==true`. This should not be possible because `bar` function is always scheduled after `foo` call. And we also have a check verifying that *maximum amount of gas is attached*.

Our attack relies on the fact that arguments for NEAR functions are serialized as JSON. Correct JSON for `foo` would look like this:

JSON

```
{
    "vaa": "abc"
}
```

We found that any fields which don't match any argument names are ignored by the near-sdk wrappers. But processing of those additional fields still happens! Which uses up a lot of gas if the JSON is huge and filled with ignored fields. For example `"xyz"` fields below would be ignored.

JSON

```
{
    "xyz": "abc",
    "xyz": "abc",
```

```
    "xyz": "abc",
    "vaa": "abc"
}
```

Our POC does binary search to try different sized JSONs as arguments for the `foo` function call. After small amount of tries it is able to find JSON which uses up just enough amount of gas inside the JSON parsing code such that call to `foo` succeeds while the Promise for `bar` call fails due to little amount of gas left.

Remediation

To fix this issue we recommend setting static amount of gas for each promise which is critical to execute(e.g. `ft_transfer` call inside `vaa_transfer`). For calls that use `near-sdk`'s `ext_contract` syntax it's possible to attach gas in the following way:

token-bridge/src/lib.rs RUST

```
prom = ext_ft_contract::ext(account)
    .with_attached_deposit(1)
    .with_static_gas(GAS_FOR_FT_TRANSFER)
    .ft_transfer(mr, U128::from(namount), None);
```

For `Promise` api function_call or function_call_weight can be used to attach gas direcly.

Patch

All the important promises now have more than enough static gas attached to them. Additionally gas checks have been replaced from just checking `env::prepaid_gas()` to `env::prepaid_gas()` - `env::used_gas()` which prevents the attack described in our POC. Fixed in commit a752e13 & 4589e89.

OS-WHN-ADV-02 [med] [resolved] | Reset VAA sequence number for any emitter

Description

We found that `register_emitter` function allows anyone to reset their supposedly monotonically increasing sequence number. This means that `seq` field inside VAA can be reset for any message publisher(on NEAR) by anyone. This breaks the purpose of the `seq` field inside the published VAAs.

This snippet shows the counter getting reset even if the emitter was already present.

wormhole/src/lib.rs RUST

```rust
    #[payable]
    pub fn register_emitter(&mut self, emitter: String) ->
    ↪  PromiseOrValue<bool> {
        let storage_used = env::storage_usage();

        self.emitters.insert(&emitter, &1);
```

We've not been able to identify any direct impact which could be caused by this broken invariant. But it could've caused problems for downstream cross-chain applications built on top of wormhole, which might rely on this field being monotonic.

Remediation

Check whether `emitter` is already present inside the `self.emitters` LookupMap.

Patch

It is now checked whether `emitter` is already present inside `self.emitters`. Fixed in a752e13.

wormhole/src/lib.rs DIFF

```
    #[payable]
    pub fn register_emitter(&mut self, emitter: String) ->
    ↪  PromiseOrValue<bool> {
+       if self.emitters.contains_key(&emitter) {
+           env::panic_str("AlreadyRegistered");
+       }

        let storage_used = env::storage_usage();
```

```
        self.emitters.insert(&emitter, &1);
```

OS-WHN-ADV-03 [med] [resolved] | Incorrect parsing of VAA payload

Description

VAA payloads have specific formats which need to be strictly followed by all the contract implementations on different blockchains. We found that the NEAR contract had several places where it was not correctly following the payload specifications.

1. When ContractUpgrade governance payload is parsed, `chain` variable inside `fn vaa_update_contract` ends up being fetched from offset `35+33=68` inside `payload`, instead of 33.

You can see in the code below that `data` argument passed in `vaa_update_contract` is slice of payload. Offset 33 would be correct if `data` was the entire payload, not part of it.

wormhole/src/lib.rs RUST

```
let data: &[u8] = &vaa.payload;

if data[0..32]
    != hex::decode("000[...]000436f7265")
        .unwrap()
{
    env::panic_str("InvalidGovernanceModule");
}

let chain = data.get_u16(33);
let action = data.get_u8(32);

if !((action == 2 && chain == 0) || chain == CHAIN_ID_NEAR) {
    env::panic_str("InvalidGovernanceChain");
}

let payload = &data[35..];

match action {
    1u8 => vaa_update_contract(self, &vaa, payload, deposit,
    ↪  refund_to.clone()),
```

wormhole/src/lib.rs RUST

```
fn vaa_update_contract(
    storage: &mut Wormhole,
    _vaa: &state::ParsedVAA,
    data: &[u8],
```

```
    deposit: Balance,
    refund_to: AccountId,
) -> PromiseOrValue<bool> {
    let chain = data.get_u16(33);
    if chain != CHAIN_ID_NEAR {
        env::panic_str("InvalidContractUpgradeChain");
    }
```

2. Inside `token-bridge` we found that upgrade hash is fetched from offset 0 inside the payload, instead of 35.

token-bridge/src/lib.rs RUST

```
fn vaa_upgrade_contract(storage: &mut TokenBridge, vaa: &state::ParsedVAA,
    ↪ deposit: Balance) -> Balance {
    let data: &[u8] = &vaa.payload;
    let chain = data.get_u16(33);
    if chain != CHAIN_ID_NEAR {
        env::panic_str("InvalidContractUpgradeChain");
    }

    let uh = data.get_bytes32(0);
    env::log_str(&format!(
        "token-bridge/{}#{}: vaa_update_contract: {}",
        file!(),
        line!(),
        hex::encode(&uh)
    ));
    storage.upgrade_hash = uh.to_vec(); // Too lazy to do proper
    ↪ accounting here...
    deposit
}
```

We followed ground truth formats from wormhole message format docs.

Remediation

Fix the occurrences of incorrect parsing. We also strongly recommend refactoring parsing code(look at OS-WHN-SUG-01).

Patch

Correct offset are being used now. Fixed in commit a752e13.

OS-WHN-ADV-04 [low] [resolved] | No storage deposit checking when upgrading contract

Description

`update_contract_work` does not check storage deposit correctly. Even though the required cost is calculated, the fact that caller did not attach enough deposit is ignored.

Remediation

Check that `update_contract` has enough deposit attached for upgraded contract size. Refund for extra deposit is already implemented inside `update_contract_done`.

Patch

Deposit is checked to be enough for contract deployment. Fixed in commit a752e13.

token-bridge/src/lib.rs RUST

```rust
    fn update_contract_work(&mut self, v: Vec<u8>) -> Promise {
        let s = env::sha256(&v);

        env::log_str(&format!(
            "token-bridge/{}#{}: update_contract: {}",
            file!(),
            line!(),
            hex::encode(&s)
        ));

        if s.to_vec() != self.upgrade_hash {
            env::panic_str("invalidUpgradeContract");
        }

        let storage_cost = ((v.len() + 32) as Balance) *
    ↪   env::storage_byte_cost();
        assert!(
            env::attached_deposit() >= storage_cost,
            "DepositUnderFlow:{}",
            storage_cost
        );

        Promise::new(env::current_account_id())
            .deploy_contract(v.to_vec())
```

```
    ↪  .then(Self::ext(env::current_account_id()).update_contract_done(
                env::predecessor_account_id(),
                env::storage_usage(),
                env::attached_deposit(),
            ))
    }
```

OS-WHN-ADV-05 [low] [resolved] | Rollback not checking failure of correct promise

Description

We found that `finish_deploy` callback which was supposed to be checking whether new `ft` contract was deployed successfully, was actually checking the status of deposit refund `Promise`.

token-bridge/src/lib.rs RUST

```
if deposit > 0 {
    env::log_str(&format!(
        "token-bridge/{}#{}: refund {} to {}",
        file!(),
        line!(),
        deposit,
        env::predecessor_account_id()
    ));
    p = p.then(Promise::new(refund_to).transfer(deposit));
}

PromiseOrValue::Promise(
  p.then(ext_token_bridge::ext(env::current_account_id()).finish_deploy(
        asset_token_account.clone(),
        tkey,
        fresh,
  )),
)
```

This means that the caller, which could be malicious, has the ability to make `finish_deploy` run as if the contract was deployed successfully even if it failed. Or, the other way around: even if the subcontract was deployed successfully, making `finish_deploy` cleanup metadata about it. Refund promise can be failed by removing `refund_to` account before `Refund` promise is executed.

This means that, VAA could also be marked as used even if the deployment was unsuccessful due to solely attackers actions. This is not severe since attestation VAAs can be regenerated easily.

Remediation

Make sure that `finish_deploy` promise is always placed right after `Promise` which contains `deploy_contract` action.

Patch

Deposit refund `Promise` is now placed *after* `finish_deploy` `Promise`. This means `finish_deploy` will always run after `deploy_contract` `Promise`. Fixed in commit a752e13.

OS-WHN-ADV-06 [low] [resolved] | Deposit is not Refunded in Certain Cases in token-bridge `vaa_transfer`

Description

In token-bridge `vaa_transfer`, when handling native NEAR and `nfee == 0`, the contract fails to refund the user the deposit:

token-bridge/src/lib.rs RUST

```rust
if nfee == 0 {
    env::log_str(&format!(
        "token-bridge/{}#{}: vaa_transfer:  sending {} NEAR to {}",
        file!(),
        line!(),
        namount,
        mr
    ));
    prom = Promise::new(mr).transfer(namount);
} else {
    env::log_str(&format!(
        "token-bridge/{}#{}: vaa_transfer:  sending {} NEAR to {}",
        file!(),
        line!(),
        namount - nfee,
        mr
    ));
    env::log_str(&format!(
        "token-bridge/{}#{}: vaa_transfer:  sending {} NEAR to {}",
        file!(),
        line!(),
        nfee,
        env::signer_account_id()
    ));

    prom = Promise::new(mr)
        .transfer(namount - nfee)
        .then(Promise::new(refund_to).transfer(nfee + deposit));
}
```

Remediation

Fix the logical error so a user is refunded in all cases.

Patch

Fixed in commit 7d0aa7a0.

OS-WHN-ADV-07 [low] [resolved] | Fractional NEAR is not Refunded in `send_transfer_near`

Description

In token-bridge `send_transfer_near`, NEAR values are truncated to 8 decimals and remaining dust amounts are retained by the token-bridge contract instead of being refunded.

Remediation

Refund dust amounts to the user.

Patch

Fixed in commit 62f1bf97.

05 | General Findings

Here we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they do represent anti-patterns and could introduce a vulnerability in the future.

ID	Status	Description
OS-WHN-SUG-00	TODO	Parsing and packing code for VAA payloads is spread around the codebase
OS-WHN-SUG-01	Resolved	Users can pass in unchecked AccountIds
OS-WHN-SUG-02	Resolved	Important functions inside `ft` contract are callable using `FunctionCall` keys
OS-WHN-SUG-03	TODO	The `get_string_from_32` utility function does not ensure that input or output strings are limited to 32 bytes.
OS-WHN-SUG-04	TODO	The `controller_or_self` function is unused in the Ft contract but could be used in places where the controller authority is verified.
OS-WHN-SUG-05	Resolved	VAA header metadata such as the guardian set index and the list of signatures is not included in the computed VAA hash and therefore systems that rely on the hash as a reference will not properly verify this metadata.

OS-WHN-SUG-00 | Refactor VAA payload parsing and packing code

Description

We recommend creating rust structs for each kind of VAA payload and writing serialization and deserialization functions for those structs. Encapsulating parsing and packing code into a single place would make it easier to verify and use rather than using offsets all around the codebase. Refactoring would allow developers working on business logic to not have to think about payload offsets.

The necessity for this can be shown by OS-WHN-ADV-03, which might've been prevented if parsing was not done inside business logic code.

Remediation

Create rust types for each of the different VAA payload types handled and write two functions for each: one for serialization and one for deserialization.

OS-WHN-SUG-01 [resolved] | Unchecked Account Ids

Description

We found that function `register_account` stores `AccountIds` created from arbitrary strings using `AccountId::new_unchecked`. This function does not check whether a given string is valid `AccountId` or not. Even though most unchecked `AccountId` uses would result in `panics`, we believe it's better to make checking explicit.

token-bridge/src/lib.rs RUST

```rust
    pub fn register_account(&mut self, account: String) -> String {
        let storage_used = env::storage_usage();
        let refund_to = env::predecessor_account_id();

        let account_hash = env::sha256(account.as_bytes());
        let ret = hex::encode(&account_hash);

        if self.hash_map.contains_key(&account_hash) {
            Promise::new(refund_to).transfer(env::attached_deposit());
            return ret;
        }
        let a = AccountId::new_unchecked(account);
        self.hash_map.insert(&account_hash, &a);
```

Remediation

Use checked API for creating `AccountId`.

Patch

Checked API is being used now. Fixed in commit 4589e89.

token-bridge/src/lib.rs DIFF

```
@@ -802,7 +804,7 @@ impl TokenBridge {
             Promise::new(refund_to).transfer(env::attached_deposit());
             return ret;
         }
-        let a = AccountId::new_unchecked(account);
+        let a = AccountId::try_from(account).unwrap();
         self.hash_map.insert(&account_hash, &a);
```

```
        if env::storage_usage() < storage_used {
```

OS-WHN-SUG-02 [resolved] | Important functions callable with FunctionCall keys

Description

It is a common pattern for NEAR contracts to prevent calling important functions with just `FunctionCall` keys. For example, such functions include ones that might end up transferring users NFTs or fungible tokens to someone else. To enforce that transaction is signed using Full Access keys, contracts verify that non-zero deposit was attached.

In `ft` contract, `vaa_transfer` and `vaa_withdraw` functions can mint and burn wrapped tokens for any user. If someone was able to obtain `FunctionCall` key for wormhole AccountId they'd be able to mint any amount of wrapped tokens. So requiring full access keys is a good idea.

Remediation

Check that attached deposit is non-zero. This enforces that transaction was signed with full access key.

Patch

Both functions checks that non-zero deposit has been attached. Fixed in a752e13.

ft/src/lib.rs DIFF

```
@@ -113,6 +115,8 @@ impl FTContract {
         fee: u128,
         payload: String,
     ) -> String {
+        assert_one_yocto();
+
         if env::predecessor_account_id() != self.controller {
             env::panic_str("CrossContractInvalidCaller");
         }
@@ -190,6 +194,10 @@ impl FTContract {

         let mut deposit: Balance = env::attached_deposit();

+        if deposit == 0 {
+            env::panic_str("ZeroDepositNotAllowed");
+        }
+
         if !self.token.accounts.contains_key(&account_id) {
             let min_balance = self.storage_balance_bounds().min.0;
```

```
            if deposit < min_balance {
```

OS-WHN-SUG-03 | Byte Util Functions are Improperly Named

In the `byte_utils` shared utility code, `get_string_from_32` is improperly named and does not enforce a string length of 32 on either the input or output:

byte_utils.rs RUST

```rust
pub fn get_string_from_32(v: &[u8]) -> String {
    let s = String::from_utf8_lossy(v);
    s.chars().filter(|c| c != &'\0').collect()
}
```

Remediation

Rename function to something more accurate like `get_string_from_utf8`.

OS-WHN-SUG-04 | Dead Code in Ft Contract

Description

There is a `controller_or_self` function in the Ft template contract:

ft/src/lib.rs RUST

```rust
/// Return true if the caller is either controller or self
pub fn controller_or_self(&self) -> bool {
    let caller = env::predecessor_account_id();
    caller == self.controller || caller == env::current_account_id()
}
```

This function is unused in the rest of the code. However, it could be used in cases where the controller authority is verified. For example:

ft/src/lib.rs RUST

```rust
if env::predecessor_account_id() != self.controller {
    env::panic_str("CrossContractInvalidCaller");
}
```

Remediation

Use this function for controller verification or remove it.

OS-WHN-SUG-05 [resolved] | VAA Header Data can be Forged

Description

In the token-bridge contract, VAA execution happens in two phases via two calls to `submit_vaa`:

1. In the first step, the VAA is validated and a marker is set in `dups` referenced by the VAA hash.
2. In the second step, this marker is retrieved by the VAA hash and then the VAA is processed in `submit_vaa_work`.

These checks ensure that the VAA which is processed in the second phase is the same one which was validated in the first phase. However, this check only holds for content which was included in the VAA hash. Importantly, any part of the VAA structure outside of the hashed content (such as the guardian set index) is not included in the hash and therefore not verified in the second step.

As a result of this bug, a user can submit an old governance VAA packet by forging the `governance_set_index` in the second call to `submit_vaa`.

Remediation

There are several ways to mitigate this vulnerability:

1. Invoke `verify_vaa` in the same call as `submit_vaa`.
2. Duplicate important values from the VAA header into the body so that they can be included in the hash.

Patch

Wormhole team was aware that guardian set index, passed inside VAA, can be changed by anyone and told us that's by design. It's not considered part of VAA header. All the downstream applications have to be aware of this design choice.

In case of governance messages it is made sure that new governance VAA is created only after all the previous VAAs have been successfully submitted. This makes submitting governance VAAs in incorrect order impossible.

A | Program Files

Below are the files in scope for this audit and their corresponding SHA256 hashes.

```
near
  contracts
    ft
      Cargo.lock                a36f79d1b6707f83b6133c7b3feaaef4b3d66c6ca78e183fbe0090249c04d00f
      Cargo.toml                fb33cfb682b02b09aad5ba769b2dab502eca2587fb55849e74c06ad5d2606a62
      src
        Makefile                58b58a8684fc2dcf315720adfd5a94844a5459882af506288017908322f24543
        lib.rs                  a47fa0b0d7986b105e4c2ce12d343193062877bfb0559189c344d6264db7e69a
    nft-bridge
      Cargo.lock                407aa6fa901f3d8bc803e8fd2fdb54c71a790d28122436f698ab87102490a9ff
      Cargo.toml                20570a3bee5076b79f3404f0c8c8caf8ac06a80d054b963d31c50394bd823fe8
      src
        Makefile                0dfa570bc0ba60048805169cf259e74a39ee126aaada6fd51b4b639d58cbcee8
        byte_utils.rs           c6ccaa40058d356ef3c2aaa44e4cd8432f87a92e1b7b65821607071d86869bbe
        lib.rs                  0e80d3ab6707f106f40e8868a0898f8d881031f8e84be8a6c53481caab019347
        state.rs                78d49ce24e20c5205beecae03dd12e4abbcf60d07930cf57091e975e68f46fa1
    nft-wrapped
      Cargo.lock                3beef810586d66244f29f3a403d9e856dee14f60d9f6383ce5b06888402b8940
      Cargo.toml                61a6840ad9cd49c6aecbc47a9a0dbbedbfb9d814542abf0bc04bdd301c504fa5
      src
        Makefile                64d7696bfa3d382492d0ed085bc1f91c4345ce0561aa1f14f35f98c9520ba437
        lib.rs                  ad2921405da7cefaebe256b04cb9ac8d2d51886da36d45cdb03a3f7893fc6927
    token-bridge
      Cargo.lock                4da0ac0b5245259394d58be7093c5f59c405b025834bff06d6877a61a9d2784b
      Cargo.toml                ed680df192c4faef8dd924e362d93d45a4aba41a4f8d1333c362a7071b906718
      src
        Makefile                5b110bc13c161b2cb7ad484107b143dcdc7c70f831cc6e28e694f315a490d595
        byte_utils.rs           c6ccaa40058d356ef3c2aaa44e4cd8432f87a92e1b7b65821607071d86869bbe
        lib.rs                  44e8e61f9b7bb52df6ca6a4bffc02790d2c7daa0accadaa361149a6489fb6e5f
        state.rs                78d49ce24e20c5205beecae03dd12e4abbcf60d07930cf57091e975e68f46fa1
    wormhole
      Cargo.lock                85fcfa186dde8a852e493382fba733a0f5eeb9167c51ec51af6b4f718eb4cc18
      Cargo.toml                39a17b8f01e664ae91a9c3457ec9cda0ddf537c187511932baea85ac04ff549c
      src
        Makefile                45b35bef0599e1ac57f29bc1440842dceb92e1ca58cb6d59cd4b02270c1a9708
        byte_utils.rs           c6ccaa40058d356ef3c2aaa44e4cd8432f87a92e1b7b65821607071d86869bbe
        lib.rs                  fa9a67150d1deec43b14ff7514bd3b3c3d2b0ba0a16ad1585408c8fdd15738d8
        state.rs                78d49ce24e20c5205beecae03dd12e4abbcf60d07930cf57091e975e68f46fa1
node
  pkg
    near
      watcher.go                8df960cb0368abfee73a1e71be9045f75db58b9452bb66673f7c1bbe7fdf2f2a
```

B | Proofs of Concept

Below are the provided proof of concept files and their corresponding SHA256 hashes.

```
README.md                                   fca85380ef24a6eb12f95becf5314c1bcdca0c753a6cb75d1c879646980229e1
reset-near.sh                               c82ea4bd978ff23bb7b88d22efc881a6af50d01be55fd0cb51b4924036072914
run.sh                                      7231b4ce89310f64419c8a9a4eec940a2953ea54d4dab749418f886980e1a9a7
os-whn-adv-00
  run.sh                                    a699c7ab04fb49cbddc70f740983934d0b25150212fd5311594aa005083df299
  child
    Cargo.lock                              1ba271a9a543b428dea32925c258b9c070764e7f290c2939f58a03afa699cec0
    Cargo.toml                              28807f0446f51fd2f1cea68e9878ecb24c7da8b7cd5bf4719218bcb560859f43
    src
      lib.rs                                3a23c9081e514c284d1328ff6d771e6f285bcaa6e931c0d4f91d3a9231e43209
  parent
    Cargo.lock                              da3ae1bd83dc2ee52b846c19baae8a2408548cca8c8143a20161e8781138adc0
    Cargo.toml                              f778e0960548669349062fe15ddfe386f46028cfd62e65f0205e4a6a3d8758ad
    README.md                               9da152737d2902daee07f0fc534c4c19c487b5d4278bcec907fbf03ee4fcfeee
    src
      lib.rs                                6031aec9850d02fe8a199f8f8fc9ad502e761d9ca58e55581468804c44daafa7
  poc
    Cargo.lock                              585cd0138474720a2fd6eb37f17d5a2def463b5b762855449a0a4b48cdbbf0d9
    Cargo.toml                              7da8cea9beecc0c150fd2c3c741d24dcc6db38ad02029c8cb8eccd4d2dacf2d7
    src
      main.rs                               fb59ac092193a5d195180dc2b224c07bdfc7091dd7c4196f3472d99f1d3e4048
os-whn-adv-01
  run.sh                                    1540f1e56f80ea5f75fa8bdf4bce693f89da2645f7512455492a632a5ac8e816
  flipper
    Cargo.lock                              4611dc2889da1622fb09fca5d4c167b0e306a1c7101144f40c274776ab2bd54e
    Cargo.toml                              c7933ad701e9b34f32edff718cd2ca4278c48eac86ad1b23c7160c8bd4c4a359
    README.md                               41bc017208ccc89bc6cbbd72b68aabb4b8ca934760264f55425dbf95dbed3441
    src
      lib.rs                                466ba76d4f62504546e5a52045143e90603e8f9c3c7d5708bdebe420d3d20af3
  poc
    Cargo.lock                              7eb8f4d755b1bfd1d2182883d96cc212f127a15d05f0e609212fd456d13c02d8
    Cargo.toml                              7da8cea9beecc0c150fd2c3c741d24dcc6db38ad02029c8cb8eccd4d2dacf2d7
    src
      main.rs                               1d98967116135a31e91a1455e157d78dae2516748acd6dc3e0375bcae2329c1b
```

C | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an onchain program. In other words, there is no way to steal tokens or deny service. An example of a design vulnerability would be an onchain oracle which could be manipulated by flash loans or large deposits.

On the other hand, auditing the implementation of the program requires a deep understanding of NEAR's execution model. Some common implementation vulnerabilities include arithmetic overflows and rounding bugs. For a non-exhaustive list of security issues we check for, see Appendix D.

One such issue that was identified here was caused by NEAR's asynchronous runtime. Attacker was able to cancel important callbacks by inflating functions usage of NEAR gas. This kind of attack can unique to NEAR blockchain. see OS-WHN-ADV-01.

Implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program. As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach any target in a team of two. This allows us to share thoughts and collaborate, picking up on details that the other missed. While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

D | Implementation Security Checklist

Unsafe arithmetic

Integer underflows or overflows	Unconstrained input sizes could lead to integer over or underflows, causing potentially unexpected behavior. Ensure that for unchecked arithmetic, all integers are properly bounded.
Rounding	Rounding should always be done against the user to avoid potentially exploitable off-by-one vulnerabilities.
Conversions	Rust as conversions can cause truncation if the source value does not fit into the destination type. While this is not undefined behavior, such truncation could still lead to unexpected behavior by the program.

Input Validation

Timestamps	Timestamp inputs should be properly validated against the current clock time. Timestamps which are meant to be in the future should be explicitly validated so.
Numbers	Reasonable limits should be put on numerical input data to mitigate the risk of unexpected over and underflows. Input data should be constrained to the smallest size type possible, and upcasted for unchecked arithmetic.
Strings	Strings should have reasonable size restrictions to prevent denial of service conditions.

Miscellaneous

Libraries	Out of date libraries should not include any publicly disclosed vulnerabilities
Clippy	cargo clippy is an effective linter to detect potential anti-patterns.

E | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities which immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority/token account validation • Rounding errors on token transfers
High	Vulnerabilities which could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities which could lead to denial of service scenarios or degraded usability. Examples: • Malicious input cause computation limit exhaustion • Forced exceptions preventing normal use
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation • Uncaught Rust errors (vector out of bounds indexing)