Zellic

Prepared for
Thomas Kevers
Mina Foundation

Prepared by
Junghoon Cho
Seunghyeon Kim
Zellic

December 20, 2024

Mina Token Bridge

Smart Contract Security Assessment

Contents

About Zellic **4**

1. **Overview** **4**

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 5

2. **Introduction** **6**

2.1. About Mina Token Bridge 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 10

2.5. Project Timeline 10

3. **Detailed Findings** **10**

3.1. Return value of `transferFrom` is not checked 11

3.2. Signatures can be replayed 13

3.3. Validator public key is not checked 14

3.4. Function `getValidatorIndex` returns index of first validator for every public key 16

3.5. The protocol owner can withdraw all funds from the bridge 18

3.6. Validators cannot be removed 20

3.7. No domain separation for validator signatures 21

3.8. No token whitelist 23

3.9. Lack of parameter validation in initialization 25

3.10. Fixed number of validators 27

3.11. Invalid threshold can halt the protocol 30

4. Discussion 30

4.1. Upgradability of Mina contracts 31

4.2. Simplifications in `validateValidator` 31

4.3. Misleading function name 32

5. Threat Model 32

5.1. Module: Bridge.sol 33

6. Assessment Results 35

6.1. Disclaimer 36

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Mina Foundation from December 9th to December 11th, 2024. During this engagement, Zellic reviewed Sotatek's development of Mina Token Bridge's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are the bridging functions implemented as intended?
- Could an attacker drain the tokens from the bridge?
- Is the bridge implemented to prevent signature-replay attacks?
- Does the bridge have proper access control for its functions?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

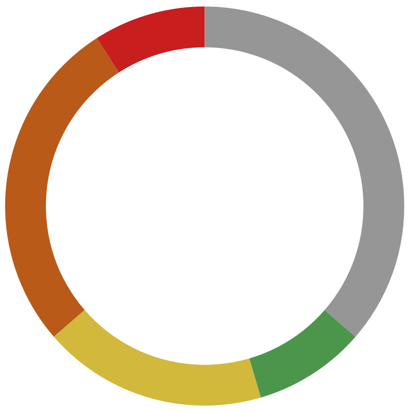
1.4. Results

During our assessment on the scoped Mina Token Bridge contracts, we discovered 11 findings. One critical issue was found. Three were of high impact, two were of medium impact, one was of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Mina Foundation in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	1
🟧 High	3
🟨 Medium	2
🟩 Low	1
⬜ Informational	4



2. Introduction

2.1. About Mina Token Bridge

Mina Foundation contributed the following description of Mina Token Bridge:

> The Mina Token Bridge enables seamless, secure, and efficient asset transfers between EVM blockchain and the Mina Chain.

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 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. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on 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 examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known 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. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential 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, upgradability weaknesses, centralization risks, and so on.

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. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and

Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly 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 various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Mina Token Bridge Contracts

Types	Solidity, TypeScript
Platform	EVM-compatible

Target	contract-evm for commit 6071594f4e12717b8a6636d7b06186124a9bd744
Repository	https://github.com/sotatek-dev/multichain-bridge-smart-contract-evm ↗
Version	`6071594f4e12717b8a6636d7b06186124a9bd744`
Programs	`Bridge`

Target	contract-mina for commit a572ed7a6802d68d69ced8de9720dc5626b9b68c
Repository	https://github.com/sotatek-dev/multichain-bridge-smart-contract-mina ↗
Version	`a572ed7a6802d68d69ced8de9720dc5626b9b68c`

Programs	`Bridge.ts`
	`BridgeToken.ts`
	`Manager.ts`
	`ValidatorManager.ts`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of five person-days. The assessment was conducted by two consultants over the course of three calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Junghoon Cho
Engineer
junghoon@zellic.io

Seunghyeon Kim
Engineer
seunghyeon@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

December 9, 2024	Start of primary review period
December 11, 2024	End of primary review period

3. Detailed Findings

3.1. Return value of transferFrom is not checked

Target	Bridge.sol		
Category	Coding Mistakes	Severity	Critical
Likelihood	Medium	Impact	Critical

Description

When the user calls the lock function to transfer ERC-20 tokens to the bridge on the Ethereum network, a Lock event is emitted. The backend observes this event or transaction to allow the user to receive funds on the Mina network.

In the current implementation, the return value of the ERC-20 transferFrom call is not checked.

```
function lock(address token, string memory receipt, uint256 amount)
    public payable {
    require(whitelistTokens[token], "Bridge: token must be in whitelist");
    require(amount <= maxAmount && amount >= minAmount, "Bridge: invalid
    amount");
    string memory name = "ETH";
    if (token == address(0)) {
        require(msg.value == amount, "Bridge: invalid amount");
    } else {
        IERC20(token).transferFrom(msg.sender, address(this), amount);
        name = IERC20(token).name();
    }
    emit Lock(msg.sender, receipt, token, amount, name);
}
```

Impact

There are tokens that comply with the ERC-20 standard but are implemented in a way that does not revert when a transfer fails. In such cases, the bridge may not receive the tokens, but the Lock event is still emitted. If a malicious user exploits this, they could inflate their funds on the Mina network and potentially drain the protocol.

Recommendations

Use the safeTransferFrom function from OpenZepplin's SafeERC20 library to ensure that the transfer is successful and handles potential errors properly.

Remediation

This issue has been acknowledged by Sotatek, and a fix was implemented in commit 17162c1a ↗.

3.2. Signatures can be replayed

Target	Bridge.ts		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Medium	**Impact**	High

Description

A minter can submit up to three signatures, and if the signatures are validated, they can receive a certain amount of tokens. However, there is no logic to check whether the signatures have already been submitted.

```
this.validateSig(msg, sig1, validator1, useSig1);
this.validateSig(msg, sig2, validator2, useSig2);
this.validateSig(msg, sig3, validator3, useSig3);
const token = new FungibleToken(tokenAddr)
await token.mint(receiver, amount)
this.emitEvent("Unlock", new UnlockEvent(receiver, tokenAddr, amount, id));
```

Impact

If a minter once obtains valid signatures that meet or exceed the threshold defined in the contract, they can replay those signatures to drain all the funds from the contract.

Recommendations

Once a signature is validated, record in a state variable that the signature has already been used, ensuring it cannot be replayed in the future.

Remediation

Sotatek acknowledged the finding and provided the following comment:

> Mina Bridge is currently operating centrally. The owner is responsible for managing minters, senders, and admins. Therefore, the impact of this issue cannot be externally affected, unless the owner discloses the information. Beside it, we need to fix both the SC and BE, and we need to go to production soon, we will temporarily not fix it.

3.3. Validator public key is not checked

Target	Bridge.ts		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Medium	**Impact**	High

Description

To unlock funds from the bridge, a number of signatures by validators need to be submitted. On the Mina side, validation of the validator's public keys is performed by the `validateValidator` function, which is implemented as follows:

```
public async validateValidator(
  useSig1: Bool,
  validator1: PublicKey,
  useSig2: Bool,
  validator2: PublicKey,
  useSig3: Bool,
  validator3: PublicKey,
) {

  let count = UInt64.from(0);
  const zero = Field.from(0);
  const falseB = Bool(false);
  const trueB = Bool(true);
  const validatorManager = new
    ValidatorManager(this.validatorManager.getAndRequireEquals());
  const validateIndex = async (validator: PublicKey, useSig: Bool) => {
    const index = await validatorManager.getValidatorIndex(validator);
    const isGreaterThanZero = index.greaterThan(zero);
    let isOk = Provable.if(useSig, Provable.if(isGreaterThanZero, trueB,
    falseB), trueB);
    isOk.assertTrue("Public key not found in validators");
  };

  const notDupValidator12 = Provable.if(useSig1.and(useSig2),
    Provable.if(validator1.equals(validator2), falseB, trueB), trueB);
  const notDupValidator13 = Provable.if(useSig1.and(useSig3),
    Provable.if(validator1.equals(validator3), falseB, trueB),trueB);
  const notDupValidator23 = Provable.if(useSig2.and(useSig3),
    Provable.if(validator2.equals(validator3), falseB, trueB), trueB);
  const isDuplicate = Provable.if(
    notDupValidator12.and(notDupValidator13).and(notDupValidator23),
    falseB,
    trueB,
  );

  isDuplicate.assertFalse("Duplicate validator keys");

  count = Provable.if(useSig1, count.add(1), count);
  count = Provable.if(useSig2, count.add(1), count);
  count = Provable.if(useSig3, count.add(1), count);
  count.assertGreaterThanOrEqual(this.threshold.getAndRequireEquals(), "Not
    reached threshold");

}
```

The second half of this function performs two checks: it checks that the validators whose signatures are used are all different, and it checks that the total number of used signatures is high enough.

The `validateValidator` function also defines a function `validateIndex`, which uses the Validator-Manager contract to check that the public keys are indeed the public keys of approved validators. However, `validateIndex` is never called on the three public keys. This check is hence not performed by `validateValidator`.

Impact

Any key may be used to sign unlocking messages. The additional security of using a quorum of validators for unlocking over fully trusting the minter is thus lost, as the minter can drain the contract by creating three different keys and signing the unlocking message with those three keys.

Recommendations

We recommend calling `validateIndex` on the three pairs of the public key and Boolean.

Remediation

This issue has been acknowledged by Sotatek, and a fix was implemented in commit `3e87c871` ↗.

3.4. Function `getValidatorIndex` returns index of first validator for every public key

Target	ValidatorManager.ts		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Medium	**Impact**	High

Description

The `getValidatorIndex` of the ValidatorManager Mina contract is intended to be used to check whether the passed public key is the public key of one of the three configured legitimate validators, and if so, which one. It returns the index of the legitimate validator, starting from `1`, or `0` if the public key is not one of the three legitimate validators. The function is implemented as follows:

```
public getValidatorIndex(p: PublicKey): Field {
    if (this.compareValidators(p, this.validator1.getAndRequireEquals()))
    return Field.from(1);
  if (this.compareValidators(p, this.validator2.getAndRequireEquals()))
    return Field.from(2);
  if (this.compareValidators(p, this.validator3.getAndRequireEquals()))
    return Field.from(3);
  return Field.from(0);
}
```

The result of `this.compareValidators(p, this.validator1.getAndRequireEquals())` is a `Bool`, a circuit variable that holds a Boolean value, not a TypeScript Boolean. Such a value is always truthy for TypeScript (see this section ↗ of the Mina documentation), so the function will return `Field.from(1)`, independently of the public key passed as an argument.

Impact

The function `getValidatorIndex` will, for every public key, return a value that indicates that this public key is the first legitimate validator. Thus, any key may be used and passed off as a validator, bypassing any checks involving validator signatures.

In the current version of the Bridge contract, `getValidatorIndex` is never called. This is however only due to another bug; see Finding 3.3. ↗. To avoid the impact discussed in Finding 3.3. ↗, both that bug as well as the one discussed in this finding will need to be fixed.

Recommendations

TypeScript `if` is not compatible with constructing proof circuits.

One option would be to make the `getValidatorIndex` function return (pseudocode),

```
comp1 * 1 + comp2 * 2 + comp3 * 3
```

where `comp1 = this.compareValidators(p, this.validator1.getAndRequireEquals()`, and similarly for `comp2` and `comp3`. This would work as intended as long as the three validator public keys are pairwise distinct, as this assumption implies that at most one of the three Booleans can be nonzero.

Remediation

This issue has been acknowledged by Sotatek, and a fix was implemented in commit `3e87c871` ↗.

3.5. The protocol owner can withdraw all funds from the bridge

Target	Bridge.sol		
Category	Protocol Risks	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

Users can transfer funds across chains through Bridges installed on Ethereum, EVM-compatible chains, and Mina. When a user transfers funds to a bridge on one chain using the `lock` function, a backend system provides a message signed by a validator containing information about the transferred fund's amount, recipient, and fee. Using this message, the bridge on the other chain allows the user to receive the funds through the `unlock` function.

Fee deduction occurs during the `unlock` process on each bridge. For example, when transferring funds from Mina to Ethereum, the fee is sent to the owner when the user calls the `unlock` function on the Ethereum bridge, and the user receives the remaining amount. However, in the case of transferring funds from Ethereum to Mina, there is no fee deduction logic in place. This design decision was made by Sotatek to avoid the protocol owner incurring costs when transferring fees collected on the Mina bridge to Ethereum. Instead of deducting fees within the Mina bridge contract, the backend pre-deducts the fee amount, ensuring the user can only unlock the net amount. The owner can then withdraw the fees recorded by the backend using the `withdrawETH` function.

```
function withdrawETH(uint256 amount) public onlyOwner {
    require(address(this).balance >= amount, "Bridge: insufficient balance");
    payable(msg.sender).transfer(amount);
}
```

However, since the fee amounts accumulated from Ethereum-to-Mina bridging are stored only in the backend and not on-chain, the `withdrawETH` function does not have a mechanism to limit the amount of ether an admin can withdraw.

Impact

The owner can theoretically withdraw the entire balance, including funds that users are supposed to claim from the bridge. This requires unnecessary extra trust on the protocol from users. Furthermore, if a security incident such as a key compromise occurs, the protocol could be drained.

Recommendations

Define the fee rate for Ethereum-to-Mina bridging within the contract, and modify the `lock` function to allow users to pass the fee amount as an argument when calling the function. The `lock` function should then verify whether the user has set a sufficient fee. If the fee is adequate, the specified amount of fee can be directly transferred to the admin. This ensures that the admin receives the exact fee amount without needing to manually withdraw the fees limitlessly.

Remediation

Sotatek acknowledged the finding and provided the following comment:

> Since this is a business requirement for us, we will keep it in this mainnet version for now. Will remove and replace it with the next version.

3.6. Validators cannot be removed

Target	Bridge.sol		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

A minter can pass multiple signatures as arguments to the `unlock` function, signed by various validators, to claim Ether or tokens. Each signature must be signed by a validator that the owner has added to the whitelist. Validators can be added using the `addListValidator` function; however, there is no function implemented to remove validators from the whitelist.

```
function _addListValidator(address[] memory _validators) internal {
    require(_validators.length > 0, "Invalid length");
    for (uint256 i = 0; i < _validators.length; i++) {
        validators[_validators[i]] = true;
    }
}
```

Impact

The inability to remove validators from the whitelist poses a protocol risk if a validator becomes untrustworthy or malicious. For example, if a validator's private key is compromised, the protocol owner may find it difficult to take appropriate security measures, further limiting operational flexibility.

Recommendations

Add a function that allows an owner to remove the registered validator.

Remediation

This issue has been acknowledged by Sotatek, and a fix was implemented in commit 17162c1a ↗.

3.7. No domain separation for validator signatures

Target	Bridge.ts		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Low

Description

In both directions of the bridge, validator signatures are needed to unlock funds. On the Mina side, the message signed by the validator consists of the receiver address (two field elements), the amount (one field element), and the token address (two field elements):

```
@method async unlock(
  amount: UInt64,
  receiver: PublicKey,
  id: UInt64,
  tokenAddr: PublicKey,
  useSig1: Bool,
  validator1: PublicKey,
  sig1: Signature,
  useSig2: Bool,
  validator2: PublicKey,
  sig2: Signature,
  useSig3: Bool,
  validator3: PublicKey,
  sig3: Signature,
) {
  // ...

  const msg = [
    ...receiver.toFields(),
    ...amount.toFields(),
    ...tokenAddr.toFields(),
  ]

  // ...

  this.validateSig(msg, sig1, validator1, useSig1);
  this.validateSig(msg, sig2, validator2, useSig2);
  this.validateSig(msg, sig3, validator3, useSig3);

  // ...
}

  public async validateSig(msg: Field[], signature: Signature, validator:
    PublicKey, useSig: Bool) {
    let isValidSig = signature.verify(validator, msg);
    const isValid = Provable.if(useSig, isValidSig, Bool(true));
    isValid.assertTrue("Invalid signature");
  }
```

This message has no domain separation.

Impact

The risk with no domain separation is that validators may produce signatures that are not intended for the bridge contract but can nevertheless be used there.

Concretely, another contract on Mina may exist, also not using any domain separation, which also asks for signatures of messages consisting of five field elements. If validators use the same key to sign messages for both contracts, it may happen that the same signed message is usable for both contracts. In this case, a validator may sign a message intended for the other contract that can also be used for the bridge contract.

Note that on the Ethereum side, proper domain separation is used, following the EIP-712 standard ↗.

Recommendations

We recommend prefixing the signed message by a field element used for domain separation. For example, such a field element could be obtained from a string such as `"MinaBridge"` as well as possibly the chain ID and the address of the bridge contract.

Remediation

Sotatek acknowledged the finding and provided the following comment:

> Mina Bridge is currently operating centrally. The owner is responsible for managing minters, senders, and admins. Therefore, the impact of this issue cannot be externally affected, unless the owner discloses the information. Beside it, we need to fix both the SC and BE, and we need to go to production soon, we will temporarily not fix it.

3.8. No token whitelist

Target	Bridge.ts		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The bridge is intended to have some whitelisted tokens, but the Mina contract has no whitelist for the tokens.

```
@method async lock(amount: UInt64, address: Field, tokenAddr: PublicKey) {
  // Check if the amount is within the allowed range
  const minAmount = this.minAmount.getAndRequireEquals();
  const maxAmount = this.maxAmount.getAndRequireEquals();

  amount.assertGreaterThanOrEqual(minAmount, "Amount is less than minimum
    allowed");
  amount.assertLessThanOrEqual(maxAmount, "Amount exceeds maximum allowed");
  const token = new FungibleToken(tokenAddr);
  await token.burn(this.sender.getAndRequireSignature(), amount);
  this.emitEvent("Lock", new LockEvent(this.sender.getAndRequireSignature(),
    address, amount, tokenAddr));

}
```

```
@method async unlock(
  amount: UInt64,
  receiver: PublicKey,
  id: UInt64,
  tokenAddr: PublicKey,
  useSig1: Bool,
  validator1: PublicKey,
  sig1: Signature,
  useSig2: Bool,
  validator2: PublicKey,
  sig2: Signature,
  useSig3: Bool,
  validator3: PublicKey,
  sig3: Signature,
) {
  const managerZkapp = new Manager(this.manager.getAndRequireEquals());
  managerZkapp.isMinter(this.sender.getAndRequireSignature());
  const msg = [
    ...receiver.toFields(),
    ...amount.toFields(),
    ...tokenAddr.toFields(),
  ]
  this.validateValidator(
    useSig1,
    validator1,
    useSig2,
    validator2,
    useSig3,
    validator3,
  );

  this.validateSig(msg, sig1, validator1, useSig1);
  this.validateSig(msg, sig2, validator2, useSig2);
  this.validateSig(msg, sig3, validator3, useSig3);
  const token = new FungibleToken(tokenAddr)
  await token.mint(receiver, amount)
  this.emitEvent("Unlock", new UnlockEvent(receiver, tokenAddr, amount, id));
}
```

Impact

The function caller can lock/unlock arbitrary tokens.

Recommendations

We recommend adding a whitelist on the functions.

Remediation

Sotatek acknowledged the finding and provided the following comment:

> Because Mina contracts have storage limitations, we can only store 8 slots in global storage. Therefore, we are hardcoding the validators and don't have enough space for whitelisted tokens. However, we are handling the filtering at the backend level.

3.9. Lack of parameter validation in initialization

Target	Bridge.sol		
Category	Coding Mistakes	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

The `initialize` function initializes several crucial state variables of the Bridge contract, such as `threshold`, `minAmount`, and `maxAmount`. The `minAmount` and `maxAmount` variables define the minimum and maximum values of funds that can be transferred to another chain via the bridge. However, the `initialize` function does not check whether `maxAmount` is greater than or equal to `minAmount`.

```
[...]
minAmount = _minAmount;
maxAmount = _maxAmount;
for (uint256 i = 0; i < _validators.length; ++i) {
    validators[_validators[i]] = true;
}
threshold = _threshold;
whitelistTokens[address(0)] = true;
[...]
```

Impact

These state variables can be modified later through setter functions. However, if the contract is initialized with incorrect values, it may be subtle and difficult to notice the issue, and the `lock` function will remain unusable until the values are corrected.

Recommendations

Add the following check to the `initialize` function.

```
require(_minAmount <= _maxAmount, "Invalid minAmount");
```

Remediation

This issue has been acknowledged by Sotatek, and a fix was implemented in commit 17162c1a ↗.

3.10. Fixed number of validators

Target	Bridge.ts		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The bridge is expected to have some validators, but the Mina contract has a fixed number of validators on the functions.

```
@method async unlock(
  amount: UInt64,
  receiver: PublicKey,
  id: UInt64,
  tokenAddr: PublicKey,
  useSig1: Bool,
  validator1: PublicKey,
  sig1: Signature,
  useSig2: Bool,
  validator2: PublicKey,
  sig2: Signature,
  useSig3: Bool,
  validator3: PublicKey,
  sig3: Signature,
) {
  const managerZkapp = new Manager(this.manager.getAndRequireEquals());
  managerZkapp.isMinter(this.sender.getAndRequireSignature());
  const msg = [
    ...receiver.toFields(),
    ...amount.toFields(),
    ...tokenAddr.toFields(),
  ]
  this.validateValidator(
    useSig1,
    validator1,
    useSig2,
    validator2,
    useSig3,
    validator3,
  );
  this.validateSig(msg, sig1, validator1, useSig1);
  this.validateSig(msg, sig2, validator2, useSig2);
  this.validateSig(msg, sig3, validator3, useSig3);
  const token = new FungibleToken(tokenAddr)
  await token.mint(receiver, amount)
  this.emitEvent("Unlock", new UnlockEvent(receiver, tokenAddr, amount, id));
}
```

```
public async validateValidator(
  useSig1: Bool,
  validator1: PublicKey,
  useSig2: Bool,
  validator2: PublicKey,
  useSig3: Bool,
  validator3: PublicKey,
) {

  let count = UInt64.from(0);
  const zero = Field.from(0);
  const falseB = Bool(false);
  const trueB = Bool(true);
  const validatorManager = new
    ValidatorManager(this.validatorManager.getAndRequireEquals());
  const validateIndex = async (validator: PublicKey, useSig: Bool) => {
    const index = await validatorManager.getValidatorIndex(validator);
    const isGreaterThanZero = index.greaterThan(zero);
    let isOk = Provable.if(useSig, Provable.if(isGreaterThanZero, trueB,
    falseB), trueB);
    isOk.assertTrue("Public key not found in validators");
  };

  const notDupValidator12 = Provable.if(useSig1.and(useSig2),
    Provable.if(validator1.equals(validator2), falseB, trueB), trueB);
  const notDupValidator13 = Provable.if(useSig1.and(useSig3),
    Provable.if(validator1.equals(validator3), falseB, trueB),trueB);
  const notDupValidator23 = Provable.if(useSig2.and(useSig3),
    Provable.if(validator2.equals(validator3), falseB, trueB), trueB);

  const isDuplicate = Provable.if(
    notDupValidator12.and(notDupValidator13).and(notDupValidator23),
    falseB,
    trueB,
  );

  isDuplicate.assertFalse("Duplicate validator keys");
```

```
    count = Provable.if(useSig1, count.add(1), count);
    count = Provable.if(useSig2, count.add(1), count);
    count = Provable.if(useSig3, count.add(1), count);
    count.assertGreaterThanOrEqual(this.threshold.getAndRequireEquals(), "Not
      reached threshold");

  }
```

Impact

The fixed number of validators could cause failure in reaching the expected threshold.

Recommendations

We recommend adding a dynamic array for the validators.

Remediation

Sotatek acknowledged the finding and provided the following comment:

> Because Mina contracts have storage limitations, we can only store 8 slots in global storage. Therefore, we are hardcoding the validators.

3.11. Invalid threshold can halt the protocol

Target	Bridge.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The argument `_newThreshold` for the function `changeThreshold` must be less than or equal to the number of validators.

```
function changeThreshold(uint256 _newThreshold) external onlyOwner() {
    threshold = _newThreshold;
    emit ChangeThreshold(_newThreshold);
}
```

Impact

If the owner sets `threshold` as a high value accidentally, the functions using the `threshold` variable will be denied of service.

Recommendations

We recommend adding logic for the `_newThreshold`.

Remediation

This issue has been acknowledged by Sotatek, and a fix was implemented in commit 17162c1a ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Upgradability of Mina contracts

The Bridge, Manager, and ValidatorManager Mina contracts use default permissions, which include `Permission.VerificationKey.signature()` for `setVerificationKey`. This makes it possible for the deployer of these contracts to change the contract's verification key and thus change the behavior of the contract.

This introduces centralization risks that users should be aware of, as it grants a single point of control over the system. We recommend that this be clearly documented for users so that they are aware of the extent of the owner's control over the contract. This can help users make informed decisions about their participation in the project. Additionally, clear communication about the circumstances in which the owner may exercise these powers can help build trust and transparency with users.

4.2. Simplifications in `validateValidator`

In the `validateValidator` function in the Mina Bridge contract, several operations could be simplified. We collect them here.

Multiple lines of the following occur:

```
const notDupValidator12 = Provable.if(useSig1.and(useSig2),
    Provable.if(validator1.equals(validator2), falseB, trueB), trueB);
```

The part `Provable.if(validator1.equals(validator2), falseB, trueB)` would be clearer as `validator1.equals(validator2).not()`, with the same functionality.

Then, instead of

```
const isDuplicate = Provable.if(
  notDupValidator12.and(notDupValidator13).and(notDupValidator23),
  falseB,
  trueB,
);

isDuplicate.assertFalse("Duplicate validator keys");
```

one could also use the following.

```
const isNotDuplicate =
    notDupValidator12.and(notDupValidator13).and(notDupValidator23);

isNotDuplicate.assertTrue("Duplicate validator keys");
```

Similarly, `Provable.if(isGreaterThanZero, trueB, falseB)`, can be just `isGreaterThanZero` in `validateIndex`.

4.3. Misleading function name

The Mina Bridge contract contains a function as follows:

```
public async verifyMsg(publicKey: PublicKey, msg: Field[], sig: Signature) {
  const isOk = await sig.verify(publicKey, msg);
  Provable.log("isOk", isOk.toString());
}
```

The name may be interpreted that this function ensures that the signature is correct. However, the function only prints whether or not the signature is correct. While this function is not used in the within-scope code, misleading function names can result in incorrect usage by developers at a later time, so we recommend to rename this function to better reflect its functionality.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the 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.

5.1. Module: Bridge.sol

Function: `lock(address token, string receipt, uint256 amount)`

This function locks the tokens for bridging the tokens.

Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be whitelisted.
  - **Impact**: Address of the token to lock.
- `receipt`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: String to be used for the `Lock` event.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be less than or equal to `maxAmount` and greater than or equal to `minAmount`.
  - **Impact**: Amount to lock.

Branches and code coverage

Intended branches

- Transfer the token to this bridge.
  - ☑ Test coverage

Negative behavior

- Revert if the token is not whitelisted.
  - ☑ Negative test

Function: `unlock(address token, uint256 amount, address user, string hash, uint256 fee, bytes[] _signatures)`

This function unlocks the tokens.

Inputs

- `token`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be whitelisted.
    - **Impact**: Address of the token to unlock.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be less than or equal to `maxAmount` and greater than or equal to `minAmount`.
    - **Impact**: Amount to unlock.
- `user`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be signed with the ECDSA.
    - **Impact**: Address to transfer to.
- `hash`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be signed with the ECDSA.
    - **Impact**: String for the `unlockHash`.
- `fee`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be signed with the ECDSA.
    - **Impact**: Amount to take as a fee.
- `_signatures`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be a valid signature.
    - **Impact**: Signature to check the signer.

Branches and code coverage

Intended branches

- Check the signer.
    - ☑ Test coverage
- Transfer the token or Ether to a given user address.
    - ☑ Test coverage

Negative behavior

- Revert when the hash has been already used.
  - ✓ Negative test
- Revert when the signer is not the validator.
  - ✓ Negative test
- Revert when the token balance is not enough.
  - ✓ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to Ethereum Mainnet.

During our assessment on the scoped Mina Token Bridge contracts, we discovered 11 findings. One critical issue was found. Three were of high impact, two were of medium impact, one was of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

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 code added to the project after the version reviewed during 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 samples 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.