CD SECURITY

AUDIT REPORT

ZKTsunami
June 2023

Introduction

A time-boxed security review of the **ZKTsunami** protocol was done by **ddimitrov22** and **chrisdior4**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **ZKTsunami**

ZkTsunami is a decentralized financial payment network that rebuilds the traditional payment stack on the blockchain. It utilizes a basket of fiat-pegged stablecoins, algorithmically stabilized by its reserve currency :ZKT:, to facilitate programmable payments and open financial infrastructure development. Their proposed ZK-AnonSNARK scheme also attains the optimal balance between performance and security, i.e., almost constant proof size and efficient proof generation and verification. This is done to enable trustless, anonymous payment for smart contract platforms.

Fund: Allows users to create an account and convert ETH or ERC-20 tokens to anonymized zkt tokens using the Fund smart contract. The native tokens are stored in the contract, and users can access it via client-side algorithms.

Transfer: Protects payment privacy by using the ZK-ConSNARK scheme to hide sender and receiver identities while encrypting transferred amounts. Essential for integrating the protocol into smart contract platforms.

Withdraw: Enables users to convert zkt tokens back to native ETH or ERC-20 tokens at their convenience using the CreateBurnTx algorithm, which takes the secret key as input.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - eb265362ad33c28d2269176e97e0373824a21fba**

Scope

The following smart contracts were in scope of the audit:

- `TransferVerifier.sol`
- `ZKTBase.sol`
- `Tsunami.sol`
- `InnerProductVerifier.sol`
- `Utils.sol`
- `BurnVerifier.sol`
- `ZKTLog.sol`
- `CheckZKT.sol`
- `ZKTERC20.sol`
- `ZKTBank.sol`
- `ZKTETH.sol`
- `ZKTFactory.sol`
- `Migrations.sol`
- `TestERC20Token.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 2 issues
- Medium: 2 issues
- Low: 4 issues
- Informational: 3 issues

Findings Summary

ID	Title	Severity
[C-01]	The ZKT Native token can be easily compromised	Critical
[C-02]	An attacker can add new ZKT tokens and steal all of the fees from them	Critical
[M-01]	Important setter functions are not constrained	Medium
[M-02]	Use `call` with value instead of `transfer`	Medium
[L-01]	Use a two-step ownership transfer approach	Low
[L-02]	Division before multiplication can lead to rounding down	Low
[L-03]	Transfers are logged with wrong values	Low

ID	Title	Severity
[L-04]	Protocol will not be able to work with certain tokens	Low
[I-01]	Redundant code	Informational
[I-02]	Incomplete NatSpec docs	Informational
[I-03]	Usage of pragma experimental ABIEncoderV2 and floating pragma statement	Informational

Detailed Findings

[C-01] The ZKT Native token can be easily compromised

Severity

Impact: High, because the protocol will lose revenue and a vital part of it will be compromised

Likelihood: High, because anyone can call the critical function directly without access control

Description

The native token is a crucial part of the protocol, the most revenue will be earned through the fees collected from it. The native ZKT Token is set in the Tsunami contract constructor:

```
    constructor (address _nativeFactory, address _erc20Factory) {
        admin = msg.sender;

        nativeFactory = ZKTNativeFactory(_nativeFactory);
        erc20Factory = ZKTERC20Factory(_erc20Factory);

        address native = nativeFactory.newZKTNative(address(this));
        zkts.set(uint256(bytes32(bytes(nativeFactory.nativeSymbol()))),
native);
        ZKTBase(native).setUnit(10000000000000000);
        ZKTBase(native).setAgency(payable(msg.sender));
        ZKTBase(native).setAdmin(msg.sender);
    }
```

However, the newZKTNative function which is called from the ZKTNativeFactory contract is public and is missing any access control:

```
    function newZKTNative (address admin) public returns (address) {
        ZKTETH zktETH = new ZKTETH(transfer, burn);
        zktETH.setAdmin(admin);
```

```
        return address(zktETH);
    }
```

This allows anyone to call it directly from the Factory contract and pass any address which will be set as admin. The admin can change important parameters inside ZKTBase but the worst thing is that a malicious user can change the agency address where all of the fees are sent. Thus, most of the revenue from the protocol will be lost.

Recommendations

Add access control to the newZKTNative function so that only the `admin` can call it.

[C-02] An attacker can add new ZKT tokens and steal all of the fees from them

Impact: High, because the protocol will lose revenue

Likelihood: High, because a critical function is missing access control

Description

Inside the Tsunami contract we have a function to add new ZKT ERC20 tokens which has the onlyAdmin modifier that makes sure only the admin of the protocol can call it.

```
    function addZKT(string calldata symbol, address
token_contract_address) public onlyAdmin {
        bytes32 zktHash = keccak256(abi.encode(symbol));
        uint256 zktId = uint256(zktHash);

        bool zktExists = zkts.contains(zktId);
        if (zktExists) {
            revert("ZKT already exists for this token.");
        }

        address erc20 = erc20Factory.newZKTERC20(address(this),
token_contract_address);
        zkts.set(uint256(bytes32(bytes(symbol))), erc20);
        ZKTBase(erc20).setUnit(10000000000000000);
        ZKTBase(erc20).setAgency(payable(msg.sender));
        ZKTBase(erc20).setAdmin(msg.sender);
    }
```

It checks if the token is already added and then calls newZKTERC20 from the ZKTERC20Factory contract. The problem comes from the fact that newZKTERC20 is a public function without access control and anyone can call it:

```
    function newZKTERC20 (address admin, address _token) public returns
(address) {
        ZKTERC20 zktERC20 = new ZKTERC20(_token, transfer, burn);
        zktERC20.setAdmin(admin);
        return address(zktERC20);
    }
```

Even worse is the fact that an attacker can pass any address that will be set as an admin. The admin has access to critical functions for the protocol.

Let's take a look at the following scenario:

1. An attacker calls newZKTERC20 and adds a new token.
2. The attacker passes his own address and is set to admin.
3. Then, he calls setAgency with his own address again which sets that all of the fees will be sent to him.

Another problem newZKTERC20 is missing any checks and an attacker can add already existing tokens and just update the agency address so he can earn even more fees.

Recommendations

Add an access control modifier that makes sure only the deployer can call newZKTERC20. Also, consider changing the visibility to internal, so that the function cannot be called on its own and all of the checks are required.

[M-01] Important setter functions are not constrained

Impact: High, as this can result in messed up calculations for important functions

Likelihood: Low, as it requires a malicious or a compromised owner, or a big mistake on the owner side

Description

There are couple of methods in ZKTBase.sol that don't have input validations, checking if the arguments value are too big or too small. A malicious/compromised admin, or one that does a "fat-finger", can input a huge number as those methods' argument, which will messed up the protocol's logic as some of these functions are doing important maths regarding fee calculation or wrong calculation of the conversion from unitAmount to nativeAmount for example. These functions are:

- `setBurnFeeStrategy()`
- `setTransferFeeStrategy()`
- `setEpochBase()`
- `setEpochLength()`
- `setUnit()`

Recommendations

Add reasonable constrains for these methods

[M-02] Use `call` with value instead of `transfer`

Impact: Medium, because if the recipient is a smart contract or a specific multisig, the transaction may fail

Likelihood: Medium, because the `transfer` method might be deprecated in the future

Description

The problem in the below snippet of code in `burn()`:

```
payable(tx.origin).transfer(nativeAmount - fee);
```

is that `tx.origin` won't work if the receiver of the rest of the native tokens is a multi-sig wallet, because `tx.origin` works only with externally owned accounts (EOAs) and multi-sig wallets are smart contracts.

Recommendations

Instead, if transferring native assets to multi-sig wallet is likely to be expected, `msg.sender` should be used instead of `tx.origin` because `msg.sender` will return the caller of the function whether it is a smart contract or a EOA. Also if you decide to use `msg.sender` here, you should also change the ether transferring method from .transfer to .call with value and check if the result is a success like this:

```
(bool success, bytes memory data) = payable(msg.sender).call{value:
msg.value}("");
require(success, "Transfer failed.");
```

The use of `.call` is encouraged because if the receiver's address is a smart contract that has a receive or fallback function that takes up more than the 2300 gas which is the limit of `transfer`.

Also the `.call` method should be used instead of `transfer` in `burnTo()` for the same reasons:

```
if (fee > 0) {
            bank.agency.transfer(fee);
            bank.totalBurnFee += fee;
        }

        // Transfer the remaining native tokens to the specified address
        payable(sink).transfer(nativeAmount - fee);
```

The `bank.agency` and `sink` can be smart contracts or multi-sig wallets that requires more than 2300 gas.

[L-01] Use a two-step ownership transfer approach

There are 10 methods with the `onlyAdmin` modifier which shows that the admin role is an important one. Single-step ownership transfer means that if a wrong address was passed when transferring ownership or admin rights it can mean that role is lost forever. The ownership pattern implementation for the protocol is in `setAdmin()` where a single-step transfer is implemented.

It is a best practice to use two-step ownership transfer pattern, meaning ownership transfer gets to a "pending" state and the new owner should claim his new rights, otherwise the old owner still has control of the contract. Consider using OpenZeppelin's `Ownable2Step` contract.

[L-02] Division before multiplication can lead to rounding down

Generally, in Solidity, it is a best practice to place multiplication before division. This is because Solidity doesn't support floating point and it round down the numbers. In the code snippet below we can see an example of that:

File: `ZKTBase.sol`

```
 uint256 fee = (usedGas * bank.TRANSFER_FEE_MULTIPLIER /
bank.TRANSFER_FEE_DIVIDEND) * tx.gasprice;
```

In case that `usedGas * bank.TRANSFER_FEE_MULTIPLIER` is less than `bank.TRANSFER_FEE_DIVIDEND`, the fee will be rounded down to zero.

Change it to:

```
-uint256 fee = (usedGas * bank.TRANSFER_FEE_MULTIPLIER /
bank.TRANSFER_FEE_DIVIDEND) * tx.gasprice;
+uint256 fee = (usedGas * bank.TRANSFER_FEE_MULTIPLIER * tx.gasprice) /
bank.TRANSFER_FEE_DIVIDEND;
```

[L-03] Transfers are logged with wrong values

When transfer is called in Tsunami contract it is logger with 0 amount:

```
ZKTLog.Item memory item = ZKTLog.Item({
    symbol: symbol,
    activity: ZKTLog.Activity.Transfer,
    addr1: zktAddr,
    addr2: msg.sender,
```

```
        amount: 0,
        timestamp: block.timestamp
        ...
    })
```

This could be problematic for any off-chain activities and monitoring.

Change it to:

```
ZKTLog.Item memory item = ZKTLog.Item({
                    symbol: symbol,
                    activity: ZKTLog.Activity.Transfer,
                    addr1: zktAddr,
                    addr2: msg.sender,
-                    amount: 0,
+                    amount: msg.value,
                    timestamp: b
                    ...
}
```

[L-04] Protocol will not be able to work with certain tokens

We can observe numerous require statements which goal is to successfully call `bank.token` transfer function as you can see from the below example snippet from `ZKTERC20.sol`:

```
if (fee > 0) {
            require(
                bank.token.transfer(bank.agency, fee), "Fee charging
failed."
            );
}
```

Now if the `bank.token` is set by the admin to be a token such as `USDT` or `ZRX`, the require statement for these particular `bank.tokens` will always revert and they are not usable because the first one (USDT) doesn't return a bool on transfer and the second (ZRX) return false.

Recommendation:

If you want to integrate such tokens, you should use OpenZeppelin's SafeERC20 library and its safe methods for ERC20 transfers.

[I-01] Redundant code

In `ZKTETH.sol` and `ZKTERC20.sol` we have `"./Utils.sol"` imported but it is not used anywhere in these contracts. Consider removing it from both contracts since it is not needed.

[I-02] Incomplete NatSpec docs

The NatSpec docs on the external methods are incomplete in all contracts - missing `@param`, `@return` and other descriptive documentation about the protocol functionality. Make sure to write descriptive docs for each method and contract which will help users, developers and auditors.

[I-03] Usage of pragma experimental ABIEncoderV2 and floating pragma statement

We can observe a usage of `pragma experimental ABIEncoderV2` in almost all of the contracts. ABIEncoderV2 is not considered experimental anymore since Solidity 0.7.4; and ABIEncoderV2 is turned on by default since Solidity 0.8.0.

Also always use a stable pragma to be certain that you deterministically compile the Solidity code to the same bytecode every time. The project is currently using a floatable version. Furthermore, consider using the latest available version which at the moment is 0.8.20.