ZKTsunami Audit Report

Prepared by bytes032

Contents

1 About bytes032

bytes032 is an independent smart contract security researcher.

His knack for identifying smart contract security vulnerabilities in a range of protocols is more than a skill; it's a passion. Committed to enhancing the blockchain ecosystem, he invests his time and energy into thorough security research and reviews. Want to know more or collaborate? bytes's always up for a chat about all things security.

Feel free to reach out on X.

2 Protocol Summary

ZKTsunami is a ZKSnark who claims to provide ZK-ANONSNARK powered transactional anonymity at your fingertips. Zcash was the first to implement and apply ZK-SNARK in the decentralized cryptocurrency. The relatively costly proof generation further reduces the likelihood of its adoption in practice.

ZKTsunami implements and integrates the state-of-the-art setup-free zero-knowledge proof protocol to enable trustless anonymous payment for smart contract platforms.

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.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of {X} according to the specific commit. Any modifications to the code will require a new security review.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)

- Medium - Should fix
- Low - Could fix

4 Executive Summary

Overview

Project	ZKTsunami
Repository	unore-zkt-audit
Commit	eb265362ad33...
Date	June 2023

Issues Found

Severity	Count
Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	2
Informational	0
Gas Optimizations	0
Total Issues	**4**

Summary of Findings

Title	Status
[M-1] Inconsistency in ZKT existence check can lead to overwritting existing ZKTs	Resolved
[M-2] Absence of toTokenAmount function hinders usability of ZKETH fund(...)	Resolved
[L-1] Unaddressed function renaming in Utils.sol breaks all the contracts that depend on it	Resolved
[L-2] Misalignment of variable names in the register() function	Resolved

5 Findings

5.1 Medium Risk

[M-1] Inconsistency in ZKT existence check can lead to overwritting existing ZKTs

Context: Tsunami.sol

Impact: This issue is of critical severity, because it allows admins to replace already existing ZKT's in the Tsunami contract, thereby potentially causing serious disruption and violating the integrity of the protocol.

Description: Within the Tsunami.sol smart contract, there is an inconsistency in how ZKT existence is checked and set. Specifically, the `addZKT` function checks if the uint256 representation of `keccak256(abi.encode(symbol))` exists, but then sets it as `uint256(bytes32(bytes(symbol))`.

The problem arises from the inequality of `uint256(keccak256(abi.encode(symbol)))` and `uint256(bytes32(bytes(symbol)))`. As a result, the existence check will always fail.

This issue is present in the following code block:

```
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);
    }
```

Recommendation: To rectify this critical issue, it is suggested to use the same `uint256(bytes32(bytes(symbol)))` for both the existence check and the set operation. This consistency will ensure accurate ZKT existence verification.

```
 function addZKT(string calldata symbol, address token_contract_address) public onlyAdmin {
+       uint256 zktId = uint256(bytes32(bytes(symbol))

        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);
    }
```

[M-2] Absence of toTokenAmount function hinders usability of ZKETH fund(...)

Context: ZKETH.sol

Impact: This issue renders the `fund` function in the ZKETH contract practically unusable, thus potentially preventing users from depositing native tokens.

In the ZKETH contract, the `fund` function is trying to use a function called `toTokenAmount` for the validation of the amount that is to be processed. However, upon reviewing the provided code and available functions in the codebase, it's clear that there is no such function named `toTokenAmount`.

The code snippet for the `fund` function is as follows:

```
// Fund function to deposit native tokens
function fund(bytes32[2] calldata y, uint256 unitAmount, bytes calldata encGuess) override external payable {
    uint256 tokenAmount = toTokenAmount(msg.value);
```

In the codebase, there is a function named `toUnitAmount` present in the Utils contract, but `toTokenAmount` is nowhere to be found.

Here's the code snippet for `toUnitAmount`:

https://github.com/Uno-Re/unore-zkt-audit/blob/8c58c3063e577e8a85612e954dfb6cc93f7ac3e2/ZKTBase.sol#L66-L71

```
function toUnitAmount(uint256 nativeAmount) internal view returns (uint256) {
    require(nativeAmount % bank.unit == 0, "error: invalid nativeAmount.");
    uint256 amount = nativeAmount / bank.unit;
    require(0 <= amount && amount <= bank.MAX, "toUnitAmount: out of range.");
    return amount;
}
```

This practically means the `fund` function is unuseable as is.

Recommendation: To rectify this issue, it's recommended to replace the non-existent `toTokenAmount` with `toUnitAmount` in the `fund` function, assuming `toUnitAmount` provides the necessary functionality required by `fund`. This will ensure that the `fund` function will perform as intended.

5.2 Low Risk

[L-1] Unaddressed function renaming in Utils.sol breaks all the contracts that depend on it

Context: TransferVerifier.sol, BurnVerifier.sol, InnerProductVerifier.sol

Impact: This issue renders any contract that relies on Utils.sol unusable. The renaming of key functions in Utils.sol without accounting for these changes throughout the rest of the codebase has resulted in a breakdown of the contract execution and interdependencies.

Description

The Utils.sol contract is a heavily modified version of the Suterusu Utils.sol contract. A range of functions have been renamed in the ZKT's implementation:

1. pAdd -> pointAdd
2. pMul -> pointMul
3. pNeg -> pointNeg
4. pEqual -> pointEqual
5. g -> generator
6. h -> cofactor
7. mapInto -> mapToCurve
8. slice -> extractSlice
9. uint2str -> uintToString

Despite these significant changes, the rest of the codebase does not reflect the new function names. This oversight has caused a cascading effect, where contracts such as TransferVerifier.sol, BurnVerifier.sol, and InnerProductVerifier.sol, which rely on Utils.sol, cannot execute properly due to unrecognized function calls. Essentially, these contracts have been "bricked" due to the changes in Utils.sol.

Recommendation: To rectify this issue, it is imperative to reflect the name changes throughout the entire codebase. Every contract that relies on Utils.sol needs to be updated to call the correctly renamed functions. This will ensure proper interaction between contracts and restore the intended functionality.

Alternatively, consider reverting the function names to their original form if the renaming is not crucial. This option would also require a thorough check of the codebase to ensure the integrity and functionality of the contracts.

[L-2] Misalignment of variable names in the register() function

Context: Tsunami.sol

Description: The register function in the smart contract begins with an attempt to create a Utils.ECPoint struct. However, there is a discrepancy in the variable names used. The function attempts to assign values to X and Y, but the ECPoint struct expects the lowercase versions x and y.

Here is the code snippet in question:

```
function register(bytes32[2] calldata y_tuple, uint256 c, uint256 s) external {
    // Calculate y
    Utils.ECPoint memory y = Utils.ECPoint({
        X: y_tuple[0],
        Y: y_tuple[1]
    });
```

In contrast, the ECPoint struct is defined as follows:

```
struct ECPoint {
    bytes32 x;
    bytes32 y;
}
```

Recommendation: To resolve this issue, the variables `X` and `Y` should be replaced with `x` and `y` in the `register` function to align with the `ECPoint` struct. This modification ensures the contract will compile and run as expected:

```
function register(bytes32[2] calldata y_tuple, uint256 c, uint256 s) external {
    // Calculate y
    Utils.ECPoint memory y = Utils.ECPoint({
+           x: y_tuple[0],
+           y: y_tuple[1]
-           X: y_tuple[0],
-           Y: y_tuple[1]
    });
```