Security Audit Report for Firn Protocol

Date: August 7, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by
BlockSec Audit Team
Date: 2023.08.07 10:48:42
+08'00'

Contents

Report Manifest

Item	Description
Client	Firn
Target	Firn Protocol

Version History

Version	Date	Description
1.0	August 7, 2023	First Release

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repo of Firn Protocol [1], a zero-knowledge privacy protocol that enables private payments. This account-based protocol employs elliptic curve cryptography to ensure the confidentiality of account balances. For this audit, we operate under the assumption that the underlying cryptography and its implementation are correct and secure.

In this protocol, the core `Firn` contract is responsible for managing key operations including registration, deposits, transfers, and withdrawals. It maintains an encrypted account balance mapping `_acc` to track each account's current balance. For registration, a user provides a deposit and a signature, which the contract validates before account registration.

Once registered, an account can engage in anonymous sets (the `FirnReader` contract implements the sampling process for selecting accounts for these sets). For transactions such as deposits, transfers, and withdrawals, the user submits a zero-knowledge proof and a statement about the transaction. The `Firn` contract updates pending balances in advance but delegates the actual verification to specific verifier contracts for each transaction type: `DepositVerifier`, `TransferVerifier`, and `WithdrawalVerifier`. Along with the `InnerProductVerifier`, these contracts execute the validation logic for their respective transaction types.

Furthermore, the `Treasury` contract collects withdrawal fees from the `Firn` contract, and its universally accessible `payout` function distributes these fees among FIRN token holders. The `ERC20` contract functions as the token contract for FIRN, inheriting from `BalanceTree` to organize token holders based on their balances in a red-black tree structure.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Firn Protocol	`Version 1`	`86b76b0434f65c8146b9468aba266fe8a9d48c29`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics

[1] https://github.com/firnprotocol/contracts

of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management

- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we find **one** potential issue. Besides, we have **two** recommendations and **two** notes.

- Low Risk: 1
- Recommendation: 2
- Note: 2

ID	Severity	Description	Category	Status
1	Low	Potential DoS in repeated registrations	Software Security	Confirmed
2	-	Remove unused variable	Recommendation	Confirmed
3	-	Remove unused inheriance	Recommendation	Confirmed
4	-	The design of skipping accounts during fee distribution	Note	-
5	-	Repeated registrations	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential DoS in repeated registrations

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The `Firn` contract implements a pending mechanism. All incoming transfers are initially held in a pending state (stored in `_pending`). Once the current epoch ends, these pending amounts can be rolled over to the respective account balances. `_lastRollOver` tracks the epoch of the last rollover for each account. During deposit, transfer, and withdrawal operations, the `rollOver` function is called to perform the rollover process for each input account Y. If `_lastRollOver[Y]` predates the current epoch (line 82), Y's pending balance is added to its balance, its `_pending` reset, and `_lastRollOver` updated.

```
81    function rollOver(bytes32 Y, uint64 epoch) internal {
82        if (_lastRollOver[Y] < epoch) {
83            _acc[Y][0] = _acc[Y][0].add(_pending[Y][0]);
84            _acc[Y][1] = _acc[Y][1].add(_pending[Y][1]);
85            delete _pending[Y]; // pending[Y] = [Utils.G1Point(0, 0), Utils.G1Point(0, 0)];
86            _lastRollOver[Y] = epoch;
87        }
88    }
```

Listing 2.1: Firn.sol

When an account is registered for the first time, its `_lastRollOver` is zero. Based on the pending design, a rollover should be executed when the next transaction involving this account arrives. However, `rollOver` is not invoked in the `register` function.

```
127   function register(bytes32 Y, bytes32[2] calldata signature) external payable {
128       require(msg.value >= 1e16, "Must be at least 0.010 ETH.");
```

```
129         require(msg.value % 1e15 == 0, "Must be a multiple of 0.001 ETH.");
130
131         uint64 epoch = uint64(block.timestamp / EPOCH_LENGTH);
132
133         require(address(this).balance <= 1e15 * 0xFFFFFFFF, "Escrow pool now too large.");
134         uint32 credit = uint32(msg.value / 1e15); // >= 10.
135         _pending[Y][0] = _pending[Y][0].add(g().mul(credit)); // convert to uint256?
136
137         Utils.Point memory pub = Utils.decompress(Y);
138         Utils.Point memory K = g().mul(uint256(signature[1])).add(pub.mul(uint256(signature[0]).neg
                ()));
139         uint256 c = uint256(keccak256(abi.encode("Welcome to Firn.", address(this), Y, K))).mod();
140         require(bytes32(c) == signature[0], "Signature failed to verify.");
141         touch(Y, credit, epoch);
142
143         emit RegisterOccurred(msg.sender, Y, credit);
144     }
```

Listing 2.2: Firn.sol

This can lead to a potential DoS vulnerability if a user registers a new account, then calls `register` again for the same account. In this case, the account balance is not updated in time between the repeated registrations.

Suppose after the first registration, another user observes the transaction and includes the account in their own anonymous set. They simulate the account balance after rollover using `simulateAccounts` and generate a proof. If the second registration arrives before their transaction, their generated proof will become invalid since it does not account for the second deposit amount. As a result, their transaction will revert and they will lose the transaction fees.

Impact N/A

Suggestion Include `rollOver` inside the `register` function.

2.2 Additional Recommendation

2.2.1 Remove unused variable

Status Confirmed

Introduced by `Version 1`

Description The `_status` variable in the `Treasury` contract is currently unused and serves no purpose. It is recommended to remove this unused variable to improve code clarity and efficiency.

Impact N/A

Suggestion Remove the unused variable.

2.2.2 Remove unused inheriance

Status Confirmed

Introduced by `Version 1`

Description The inheritance of the Beacon Proxy in the `ERC1967Proxy` contract is currently not utilized or referenced in the code. It is recommended to remove the unused inheritance. This will help improve code clarity and eliminate any confusion for future developers working on the codebase.

Impact N/A

Suggestion Remove the unused inheriance.

2.3 Note

2.3.1 The design of skipping accounts during fee distribution

Description The `traverse` function in the `Treasury` contract distributes fee shares to FIRN token holders. It traverses holders in descending order of balance, checking the `_skip` flag to potentially skip allocation for certain holders (line 58). However, this conditional skipping can lead to unfair distribution. Specifically, the order of skipped holders in the BalanceTree determines the actual allocation. Holders with identical balances may receive different allocations depending on the position of skipped holders in the tree.

```
48    function traverse(address cursor) internal {
49        (,address left,address right,) = _erc20.nodes(cursor);
50
51        if (right != address(0)) {
52            traverse(right);
53        }
54        if (gasleft() < _endGas) {
55            return;
56        }
57        uint256 firnBalance = _erc20.balanceOf(cursor);
58        if (!_skip[cursor]) {
59            uint256 amount = address(this).balance * firnBalance / _firnSupply;
60            bool success = payable(cursor).send(amount);
61            if (success) {
62                emit Payout(cursor, amount);
63            }
64        }
65        // there is a further attack where someone could try to transfer their own firn balance
              within their ‘receive‘.
66        // the effect of this would be to get paid essentially twice for the same firn (there are
              other variants of this).
67        // to prevent this, we’re assuming that 2,300 gas isn’t enough to do a FIRN ERC20 transfer.
68        _firnSupply -= firnBalance;
69        if (left != address(0)) {
70            traverse(left);
71        }
72    }
```

Listing 2.3: Treasury.sol

Feedback from the Project Skipping is mainly used for very large holders, who are at the top of the list. Note that this doesn’t happen if the skipped accounts are above all others in the ranking. By the way, a similar thing can also happen if `send` fails (say, the holder is a contract with no `receive` ether function). But as mentioned, I am aware of this and think it’s ok.

2.3.2 Repeated registrations

Description The `register` function in the `Firn` contract allows users to register the same account multiple times. Registering the same account multiple times has a similar effect to making multiple deposits to that account. The key difference is that with multiple registrations, the deposit amount for each registration will be publicly visible, reducing privacy.

Feedback from the Project This is not the intended use of `register`. It's designed to be used only the very first time you set up your account. Later, the UI will use `depsosit`, not `register`. But use of `register` in this way is not prevented by the smart contract.

BTW, note that repeatedly registering is not strictly speaking a problem, except that `rollOver` is missing from `register`, and front-running could happen. If somebody wants to deposit repeatedly by "registering" even though they've already registered, we don't have much reason to stop them. In fact, if they really know what they're doing, this might be a gas-efficient alternative to deposit, which only really needs to be used when you're using a fresh Ethereum account to deposit to the same Firn account.