BENQI Ignite

Smart Contract Security Assessment

March 28, 2023



ABSTRACT

Dedaub was commissioned to perform a security audit of the BENQI Ignite protocol.

Ignite is the new base layer protocol of BENQI for decentralized staking. The protocol allows developers and stakers to deploy Avalanche validators at a fraction of the cost. To become an Avalanche validator, a minimum of `2000 AVAX` must be staked. With Ignite, BENQI provides all `2000 AVAX` needed to spin up a validator node in exchange for a user posting a mere `200 AVAX` worth of `QI` as collateral. The `QI` token is a native asset on Avalanche, which oversees the entire BENQI protocol ecosystem, including the BENQI Liquidity Market and BENQI Liquid Staking. BENQI will support the mission of the Ignite protocol with `1M AVAX` coming from its Liquid Staking protocol.

This audit report covers the contracts of the at-the-time private repository BENQI-fi/ignite-contracts of the BENQI Ignite protocol at commit `498242b800b07230e81cacb6932c217ba3d07d05`. As part of the overall understanding of the protocol and its interactions, the auditors reviewed parts of specific contracts of the at-the-time private repository BENQI-fi/veqi at commit `1b108ea24f65790b279c1b843056611a1d432965`. More specifically, the auditors examined the `deposit()` and `withdraw()` functions of the `VeQi.sol` contract with which the Ignite protocol interacts.

Two auditors worked on the codebase for 3 days on the following contracts:

```
src/
├─ IIgniteVeQiDepositProxy.sol
├─ IPriceFeed.sol
├─ IVeQi.sol
├─ Ignite.sol
├─ IgniteStorage.sol
├─ IgniteVeQiDepositProxy.sol
├─ IgniteVeQiDepositProxyImplementation.sol
├─ IgniteVeQiDepositProxyImplementationRegister.sol
└─ IgniteVeQiDepositProxyStorage.sol
```

SETTING & CAVEATS

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems

	misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes “dismissed” or “acknowledged” but no action taken, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	Missing data structure update in `_deleteRegistration` leaves records in an unusable state	**RESOLVED**
The `Ignite` contract has a function called `_deleteRegistration()` which removes an expired validator node registration or a non-tokenized node registration of a privileged account.		

When a node is deleted, the last record is transferred to the index of the to-be-deleted node and the last node is popped from the array. However, the `registrationIndicesByNodeIdHash` mapping (which holds the registrations array index of each node's ID hash) for the transferred last record isn't updated.

Ignite::_deleteRegistration()

```
function _deleteRegistration(
    string memory nodeId
) internal {
    bytes32 nodeIdHash = keccak256(abi.encodePacked(nodeId));
    uint registrationIndex =
        registrationIndicesByNodeIdHash[nodeIdHash];
    ...
    uint totalRegistrations = registrations.length;

    // Dedaub: The registrationIndicesByNodeIdHash doesn't get updated
    //         when moving the records resulting in a conflict in
    //         the two arrays
    if (totalRegistrations != 1) {
        registrations[registrationIndex] =
            registrations[totalRegistrations - 1];
    }
    registrations.pop();
    delete registrationIndicesByNodeIdHash[nodeIdHash];

    emit RegistrationDeleted(nodeId);
}
```

As a result, all calls for the transferred record will fail, causing a loss of funds and generally breaking the entire functionality of the protocol.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	No staleness check on Oracles returned values	**RESOLVED**

The `register()` function of the `Ignite` contract fetches the AVAX and the QI price each time a new validator node registration is requested.

`Ignite::register()`

```
function register(
    string memory nodeId,
    uint validationDuration
) external payable {
    ...
    // Dedaub: Here the function fetches the current AVAX and QI prices
    //         from the Chainlink Oracles
    (, int256 avaxPrice, , ,) = avaxPriceFeed.latestRoundData();
    (, int256 qiPrice, , ,) = qiPriceFeed.latestRoundData();
    ...
}
```

However, the Chainlink Oracles may be down or unresponsive for a period of time, resulting in outdated prices being fetched.

The Oracles return the time they were last updated along with the asset's price. Adding a check for this value could therefore prevent outdated prices from being used.

LOW SEVERITY:

ID	Description	STATUS
L1	Lack of reentrancy guard in `Ignite::register()` can lead to issues in future versions of the contract	**RESOLVED**

The `Ignite` contract has a function called `register()` which allows a user to register a new validator node by locking up `QI` tokens and optionally `AVAX`.

To ensure that the sender is able to get `AVAX` back when the validation period expires, the function makes a test call to the sender.

`Ignite::register()`

```
function register(
    string memory nodeId,
    uint validationDuration
) external payable {

    require(msg.value == 0 || (msg.value >= minimumAvaxDeposit &&
                               msg.value <= maximumAvaxDeposit &&
                               msg.value % 1e9 == 0),
            "Invalid value");

    // Dedaub: Here a test call is made to ensure that the caller can
    //         receive AVAX back passing the control to the caller
    (bool success, ) = msg.sender.call("");
    require(success);

    ...
}
```

However, this transfers the control back to the user, who can re-enter the function, as there are no re-entrancy guards in place.

Although we do not believe that a re-entrancy can be concretely exploited in the current version of the contract, it is not unlikely that future modifications of the code might introduce such a vulnerability. As a consequence, we recommend adding a guard or using a checks-effects-interactions pattern (with proper documentation to ensure that the pattern is followed in the future).

L2	Initializers of the inherited contracts are not called	**RESOLVED**

The `Ignite` contract inherits from `AccessControlUpgradeable`, `ReentrancyGuardUpgradeable` and `PausableUpgradeable`. However, the `initializer()` of the contract doesn't call their initializers.

`Ignite::initialize()`

```
function initialize(
    address _qi, address _veQi,
    address _veQiDepositProxyImplementationRegister,
    address _slashedTokenRecipient,
    address _avaxPriceFeed, address _qiPriceFeed,
    uint _minimumAvaxDeposit, uint _maximumAvaxDeposit
) initializer public {

    require(_slashedTokenRecipient != address(0), "Zero address");

    _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);

    qi = IERC20Upgradeable(_qi);
    veQi = _veQi;
    veQiDepositProxyImplementationRegister =
        _veQiDepositProxyImplementationRegister;
    slashedTokenRecipient = _slashedTokenRecipient;
```

```
    avaxPriceFeed = IPriceFeed(_avaxPriceFeed);
    qiPriceFeed = IPriceFeed(_qiPriceFeed);
    minimumAvaxDeposit = _minimumAvaxDeposit;
    maximumAvaxDeposit = _maximumAvaxDeposit;

    qiSlashPercentage = 5_000;
    maximumSubsidisationAmount = 500_000e18;
}
```

Some of the initializers may currently have an empty implementation, but this could change in a future version of the OpenZeppelin code, which could be used by an upgraded version of the aforementioned contract. This omission of the call could lead to a serious bug if such a change went unnoticed or was not followed by the introduction of a call to the initializers.

Furthermore, the initializers of the `ReentrancyGuardUpgradeable` and `PausableUpgradeable` already have an implementation that is never executed. In particular, this may not cause any problems in the current state of the code, but it could lead to bugs, as mentioned above, in future versions if new functionality is added.

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Code simplification/gas savings by starting registrations at index 1	**RESOLVED**

At several places in the code, a check is performed to ensure that the element at the selected index has the correct `nodeId`. For example:

`Ignite::redeemAfterExpiry()`

```
function redeemAfterExpiry(
    string memory nodeId
) external nonReentrant whenNotPaused {
    bytes32 nodeIdHash = keccak256(abi.encodePacked(nodeId));
    Registration storage registration =
        registrations[registrationIndicesByNodeIdHash[nodeIdHash]];

    require(
        keccak256(abi.encodePacked(registration.nodeId)) == nodeIdHash,
        "Registration not found");
    ...
}
```

Note that it is not sufficient to check that `nodeIdHash` exists in `registrationIndicesByNodeIdHash`, since we cannot distinguish a non-existent `nodeId` from one that happens to exist at index `0`.

The code could be significantly simplified by simply adding a dummy unused element in the `registrations` array and starting the real elements from index `1`. This way, a simple check `registrationIndicesByNodeIdHash[nodeIdHash] != 0` is enough, the element at that index is guaranteed to have the correct `nodeId`.

In fact, there is no reason to hash the `nodeIds` at all. We could simply have a mapping:

```
mapping(string => uint) public registrationIndicesByNodeId;
```

that directly maps `nodeIds` to indices, saving the gas of hashing.

A2	Possible gas optimization	RESOLVED

The `Ignite` contract has a function called `releaseLockedTokens()` which is called when the validation period has expired for releasing the deposited tokens for withdrawal.

However, when a user hasn't deposited AVAX upon registering his validator node and he has been slashed in the meantime, then some redundant calls are made that could be avoided for saving some gas.

`Ignite::releaseLockedTokens()`

```
function releaseLockedTokens(
    string memory nodeId,
    uint rewards,
    bool slash
) external payable whenNotPaused {
    require(hasRole(ROLE_RELEASE_LOCKED_TOKENS, msg.sender),
        "ROLE_RELEASE_LOCKED_TOKENS");
    require(!slash || rewards == 0,
        "Cannot reward and slash the same node");
    ...
    require(registration.avaxDepositAmount == 0 || slash || rewards > 0,
        "Non-slashed node with deposited AVAX must be rewarded");
    ...

    if (rewards > 0) {
        ...
    } else if (slash) {
        require(msg.value == registration.avaxDepositAmount,
            "Invalid value");

        registration.slashed = true;

        uint avaxSlashAmount = registration.avaxDepositAmount *
```

```
            registration.avaxSlashPercentage / 10_000;

        uint qiSlashAmount = registration.qiDepositAmount *
            registration.qiSlashPercentage / 10_000;

        // Dedaub: Here avaxSlashAmount could be checked if it's 0 to
        //         avoid a redundant storage update and a zero-value
        //         transfer
        minimumContractBalance += msg.value - avaxSlashAmount;
        (bool success, ) = slashedTokenRecipient.call{
            value: avaxSlashAmount }("");
        require(success);
        ...
    } else {
        require(msg.value == 0, "Invalid value");
    }
}
```

A3	There is no check to ensure the correctness of the new AVAX deposit range	**RESOLVED**

The `Ignite` contract has a function called `setAvaxDepositRange()` which changes the range of the AVAX a user can send upon registering his validator node.

However, there are no checks to ensure that the new `minimumAvaxDeposit` is less than or equal to the `maximumAvaxDeposit`. Even though this can easily be resolved it would make sense to add such check for avoiding any unwanted interruptions from the proper execution of the protocol.

`Ignite::setAvaxDepositRange()`

```
function setAvaxDepositRange(
    uint newMinimumAvaxDeposit,
```

```
    uint newMaximumAvaxDeposit
) external {
    require(hasRole(DEFAULT_ADMIN_ROLE, msg.sender),
        "DEFAULT_ADMIN_ROLE");
    uint oldMinimumAvaxDeposit = minimumAvaxDeposit;
    uint oldMaximumAvaxDeposit = maximumAvaxDeposit;

    // Dedaub: No check whether the
    //         newMinimumAvaxDeposit <= newMaximumAvaxDeposit
    minimumAvaxDeposit = newMinimumAvaxDeposit;
    maximumAvaxDeposit = newMaximumAvaxDeposit;

    emit AvaxDepositRangeUpdated(
        oldMinimumAvaxDeposit,
        minimumAvaxDeposit,
        oldMaximumAvaxDeposit,
        maximumAvaxDeposit
    );
}
```

A4	Compiler bugs	**INFO**

The code can be compiled with Solidity 0.8.0 or higher. For deployment, we recommend no floating pragmas, but a specific version, to be confident about the baseline guarantees offered by the compiler. Version 0.8.0, in particular, has some known bugs, which we do not believe affect the correctness of the contracts.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.