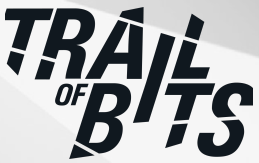

Rocket Pool

Security Assessment

September 9, 2021

Prepared for:

Darren Langley
Rocket Pool Pty Ltd.

David Rugendyke
Rocket Pool Pty Ltd.

Prepared by:

Dominik Teiml, Maximilian Krüger, and Devashish Tomar

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include CircleCI, HashiCorp, Google, Microsoft, SanDisk, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com/
info@trailofbits.com

Notices and Remarks

Classification and Copyright

This report is confidential and intended for the sole internal use of Rocket Pool Pty Ltd.

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and mutually agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or partners. As such, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Table of Contents

Executive Summary

Overview

Rocket Pool engaged Trail of Bits to review the security of its smart contracts. From August 9 to August 20, 2021, a team of three consultants conducted a security review of the client-provided source code, with five person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

We focused our testing efforts on the identification of flaws that could result in a compromise or lapse of confidentiality, integrity, or availability of the target system. We performed automated testing and a manual review of the code, in addition to running system elements.

Summary of Findings

Our review resulted in five high-severity and three informational-severity issues. All issues but one have a difficulty level of high, which means that an attacker would have to overcome nontrivial obstacles to exploit them.

EXPOSURE ANALYSIS

Severity	*Count*
High	5
Medium	0
Low	0
Informational	3
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	4
Data Validation	3
Undefined Behavior	1

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dguido@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

Dominik Teiml, Consultant
dominik.teiml@trailofbits.com

Max Krüger, Consultant
max.kruger@trailofbits.com

Devashish Tomar, Consultant
devashish.tomar@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
August 4, 2021	Project kickoff call
August 13, 2021	Delivery of weekly report
August 24, 2021	Delivery of report draft
August 24, 2021	Report readout meeting
September 8, 2021	Fix Log added (Appendix D)

Project Targets

The engagement involved a review and testing of the target listed below.

Rocket Pool

Repository	https://github.com/rocket-pool/rocketpool
Version	`a65b203cf99c7a991c2d85a7468a97bfb5dbba31`
Type	Solidity
Platform	Ethereum

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

`RocketBase`. Most system contracts are derived from `RocketBase`. We checked the access controls on these contracts, verified that the getters and setters are implemented correctly, and checked the correctness of the modifiers, which define the access controls for the rest of the system.

`RocketStorage`. This contract stores all system storage data. It is one of the few contracts that do not inherit from `RocketBase`. Instead, it defines its own access control modifier; we checked that it is correct. In addition to this modifier, the contract contains direct getters and setters for various types. We assessed the impact of these functions' direct manipulation of storage values and reviewed the guardian mechanism and the mechanism for changing a node's withdrawal address.

`RocketVault`. This contract holds the system's ether and token balances. It also provides an API through which other contracts can deposit and withdraw ether and tokens, transfer tokens, and burn tokens. We checked that these functionalities are implemented correctly and looked for any common pitfalls of token and ether transfers.

`RocketDepositPool`. This contract handles users' ether deposits. We checked that the deposit flow has the correct preconditions, including minimum and maximum values, that rETH tokens are minted in the correct amount, that the ether flow is correct, and that minipools are properly created and assigned to node operators when they make deposits.

`RocketNodeManager`. This contract handles the registration of new decentralized nodes, updates to their time zones, and the aggregation of storage values for other contracts. We checked the node-registration process, the contract's handling of time zones, and the implementation of view methods.

`RocketNodeDeposit`. This contract has one public entry point, which allows nodes to deposit ether stakes. We checked the ether flow and the contract's process of creating minipools and assigning user deposits to them.

`RocketNodeStaking`. This contract enables nodes to stake RPL tokens, increasing the protocol's confidence in the nodes. We verified the staking process, the handling of arithmetic, and the stake withdrawal and slashing processes.

Minipool contracts. Each of these contracts represents a minipool—that is, a pool holding 32 ether that serves as a bridge between the Rocket Pool network and the Eth2 deposit contract. We checked that `RocketMinipoolQueue` correctly interfaces with `AddressQueueStorage` and properly handles the three different deposit types. We also assessed the `RocketMinipoolManager` contract's 16 view methods and creation and destruction of minipools. When reviewing `RocketMinipool`, we focused on the delegation mechanism, checking whether any of the functions in the current delegate shadow those in the proxy; we also verified that the node operator's control over delegate selection is implemented correctly. For the current implementation of `RocketMinipoolDelegate`, we checked that the functions have appropriate access controls, model the correct state transitions, and have the right side effects.

`AddressQueueStorage`. This utility contract defines a priority queue data structure. Using numerous test properties and test sequences, we fuzzed the contract to assess whether the API it exposes is consistent with a typical priority queue API.

`RocketTokenRETH`. The RETH token represents the yield realized by Rocket Pool network nodes. We checked that the methods it uses to calculate exchange rates are correct and that there are appropriate access controls on authorized methods.

`RocketTokenRPL`. The RPL token is inflationary and is the primary network token. The `RocketTokenRPL` contract exposes a public entry point that allows users to mint new RPL tokens distributed by the `RocketRewardsPool`. We checked that tokens can be minted only once per period, that the appropriate number of tokens is minted on each call, and that new RPL tokens are correctly deposited into the rewards pool.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- `RocketRewardsPool`
- `AddressSetStorage`
- `RocketClaim` contracts
- `RocketNetwork` contracts
- `Rocket DAO` contracts
- `RocketAuctionManager`

Automated Testing Results

Trail of Bits has developed three unique tools for testing smart contracts. Descriptions of these tools and their use in this project are provided below.

- Slither is a static analysis framework that can statically verify algebraic relationships between Solidity variables. We used both open- and closed-source Slither detectors.
- Echidna is a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation. We used Echidna to test the implementation of the `AddressQueueStorage` contract.
- Manticore is a symbolic execution framework. Manticore can exhaustively test security properties via symbolic execution. Time constraints prevented us from using Manticore in this assessment.

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations: Slither may identify security properties that fail to hold when Solidity is compiled to EVM bytecode, Echidna may not randomly generate an edge case that violates a property, and Manticore may fail to complete its analysis.

We take a consistent approach to evaluating security properties. When using Echidna, we generate 10,000 test cases per property; when testing with Manticore, we run the tool for a minimum of one hour. In both cases, we then manually review all results.

Our automated testing and verification focused on the following system property:

AddressQueueStorage. When a registered node deposits ether into the system, the `RocketMinipoolManager` contract creates a new minipool. This contract manages each minipool and calls `RocketMinipoolQueue.enqueueMinipool`, which calls the utility contract `AddressQueueStorage.enqueueItem`. We focused on the implementation of this last contract and tested it through differential fuzzing, using an implementation with a linked list. We tested the following properties:

Property	Approach	Result
`enqueueItem(queue, element)` reverts if the element is already present in the queue.	Echidna	**Passed**

`enqueueItem(queue, element)` does not revert if the element is not present. Additionally, that element will be preceded by all elements enqueued before it and will have precedence over all elements that follow it.	Echidna	**Passed**
`dequeueItem(queue)` reverts if the length of the queue is zero.	Echidna	**Passed**
If the queue is not empty, `dequeueItem(queue)` removes and returns the first element and does not revert.	Echidna	**Passed**
`removeItem(queue, element)` reverts if the element is not present in the queue.	Echidna	**Passed**
`removeItem(queue, element)` does not revert if the element is included in the queue. The removal of the element does not affect the order of the other elements.	Echidna	**Passed**
`getItem(queue, index)` is the inverse of `getIndexOf(queue, element)`.	Echidna	**Passed**
`getLength(queue)` returns the correct length.	Echidna	**Passed**

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Access Controls	The system is composed of many modules, which makes it challenging to create suitable access controls. We identified four issues related to access controls (TOB-ROCKET-001, TOB-ROCKET-004, TOB-ROCKET-006, TOB-ROCKET-008). The system would also benefit from clearer documentation on all privileged roles and operations, especially the temporary guardian role.	**Moderate**
Arithmetic	The codebase primarily uses straightforward arithmetic. The codebase uses unprotected arithmetic only in the calculation of constants during compile time; it uses `SafeMath` in all nontrivial runtime arithmetic.	**Satisfactory**
Assembly Use/Low-Level Calls	The codebase does not use bit manipulation. It uses assembly and low-level calls sparingly. However, we found one issue stemming from the use of assembly (TOB-ROCKET-002) and another from the use of low-level calls (TOB-ROCKET-005).	**Moderate**
Centralization	The system includes a guardian role that is temporary until the DAO has been deployed. This role has few privileges beyond the ability to initialize variables of the `RocketStorage` contract. However, the system would benefit from clearer documentation on the guardian role.	**Satisfactory**
Upgradeability	Most of the system contracts can be upgraded. Although these contracts avoid the worst-case scenarios that can arise from upgrades, we found two issues related to	**Moderate**

	upgradeability (TOB-ROCKET-004, TOB-ROCKET-005). We recommend using slither-check-upgradeability to perform automated testing of all future upgrades.	
Function Composition	Overall, the codebase is well structured. The logic is divided well, and the control flow is handled appropriately. However, the eternal storage pattern introduces a significant amount of boilerplate code, which hinders readability.	**Satisfactory**
Front-Running	Further investigation is required.	**Further Investigation Required**
Monitoring	All of the functions emit events where appropriate. The events emitted by the code are capable of effectively monitoring on-chain activity.	**Satisfactory**
Specification	Rocket Pool has developed an explainer series and a whitepaper that serve as an introduction to the system. Additionally, the codebase contains comments that help to explain the system's intended behavior. However, we recommend creating technical documentation that includes architecture diagrams and a list of privileged roles and the related semantics.	**Moderate**
Testing and Verification	The codebase contains unit and integration tests for the most common operations. However, we recommend introducing additional edge cases for both parameter values and complex behavior patterns as well as fuzz tests.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Any network contract can change any node's withdrawal address	Access Controls	High
2	Current storage pattern fails to ensure type safety	Data Validation	High
3	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
4	Upgradeable contracts can block minipool withdrawals	Access Controls	High
5	Lack of contract existence check on delegatecall will result in unexpected behavior	Data Validation	High
6	tx.origin in RocketStorage authentication may be an attack vector	Access Controls	High
7	Duplicated storage-slot computation can silently introduce errors	Data Validation	Informational
8	Potential collisions between eternal storage and Solidity mapping storage slots	Access Controls	Informational

Detailed Findings

1. Any network contract can change any node's withdrawal address	
Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-ROCKET-001
Target: `RocketStorage.sol`	

Description

The `RocketStorage` contract uses the eternal storage pattern. The contract is a key-value store that all protocol contracts can write to and read. However, `RocketStorage` has a special protected storage area that should not be writable by network contracts (figure 1.1); it should be writable only by node operators under specific conditions. This area stores data related to node operators' withdrawal addresses and is critical to the security of their assets.

```
    // Protected storage (not accessible by network contracts)
    mapping(address => address)    private withdrawalAddresses;
    mapping(address => address)    private pendingWithdrawalAddresses;
```

Figure 1.1: Protected storage in the `RocketStorage` contract (`RocketStorage.sol#L24-L26`)

`RocketStorage` also has a number of setters for types that fit in to a single storage slot. These setters are implemented by the raw `sstore` opcode (figure 1.2) and can be used to set any storage slot to any value. They can be called by any network contract, and the caller will have full control of storage slots and values.

```
    function setUint(bytes32 _key, uint _value) onlyLatestRocketNetworkContract override
external {
        assembly {
            sstore (_key, _value)
        }
    }
```

Figure 1.2: An example of a setter that uses `sstore` in the `RocketStorage` contract (`RocketStorage.sol#L205-209`)

As a result, all network contracts can write to all storage slots in the `RocketStorage` contract, including those in the "protected" area.

There are three setters that can set any storage slot to any value under any condition: `setUint`, `setInt`, and `setBytes32`. The `addUint` setter can be used if the unsigned integer representation of the value is larger than the current value; `subUint` can be used if it is smaller. Other setters such as `setAddress` and `setBool` can be used to set a portion of a storage slot to a value; the rest of the storage slot is zeroed out. However, they can still be used to delete any storage slot.

In addition to undermining the security of the protected storage areas, these direct storage-slot setters make the code vulnerable to accidental storage-slot clashes. The burden of ensuring security is placed on the caller, who must pass in a properly hashed key. A bug could easily lead to the overwriting of the guardian, for example.

Exploit Scenario

Alice, a node operator, trusts Rocket Pool's guarantee that her deposit will be protected even if other parts of the protocol are compromised. Attacker Charlie upgrades a contract that has write access to `RocketStorage` to a malicious version. Charlie then computes the storage slot of each node operator's withdrawal address, including Alice's, and calls `rocketStorage.setUint(slot, charliesAddress)` from the malicious contract. He can then trigger withdrawals and steal node operators' funds.

Recommendations

Short term, remove all `sstore` operations from the `RocketStorage` contract. Use mappings, which are already used for strings and bytes, for all types. When using mappings, each value is stored in a slot that is computed from the hash of the mapping slot and the key, making it impossible for a user to write from one mapping into another unless that user finds a hash collision. Mappings will ensure proper separation of the protected storage areas.

Strongly consider moving the protected storage areas and related operations into a separate immutable contract. This would make it much easier to check the access controls on the protected storage areas.

Long term, avoid using assembly whenever possible. Ensure that assembly operations such as `sstore` do not enable the circumvention of access controls.

2. Current storage pattern fails to ensure type safety

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ROCKET-002
Target: `RocketStorage.sol`	

Description

As mentioned in TOB-ROCKET-001, the `RocketStorage` contract uses the eternal storage pattern. This pattern uses assembly to read and write to raw storage slots. Most of the system's data is stored in this manner, which is shown in figures 2.1 and 2.2.

```
    function setInt(bytes32 _key, int _value) onlyLatestRocketNetworkContract override
external {
        assembly {
            sstore (_key, _value)
        }
    }
```

Figure 2.1: `RocketStorage.sol#L229-L233`

```
    function getUint(bytes32 _key) override external view returns (uint256 r) {
        assembly {
            r := sload (_key)
        }
    }
```

Figure 2.2: RocketStorage.sol#L159-L163

If the same storage slot were used to write a value of type *T* and then to read a value of type *U* from the same slot, the value of *U* could be unexpected. Since storage is untyped, Solidity's type checker would be unable to catch this type mismatch, and the bug would go unnoticed.

Exploit Scenario

A codebase update causes one storage slot, *S*, to be used with two different data types. The compiler does not throw any errors, and the code is deployed. During transaction processing, an integer, -1, is written to *S*. Later, *S* is read and interpreted as an unsigned integer. Subsequent calculations use the maximum `uint` value, causing users to lose funds.

Recommendations

Short term, remove the assembly code and raw storage mapping from the codebase. Use a mapping for each type to ensure that each slot of the mapping stores values of the same type.

Long term, avoid using assembly whenever possible. Use Solidity as a high-level language so that its built-in type checker will detect type errors.

3. Solidity compiler optimizations can be problematic

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-ROCKET-003
Target: `truffle.js`	

Description
Rocket Pool has enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to solc-js—causes a security vulnerability in the Rocket Pool contracts.

Recommendations
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

4. Upgradeable contracts can block minipool withdrawals

Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-ROCKET-004
Target: `RocketMinipoolDelegate.sol`	

Description

At the beginning of this audit, the Rocket Pool team mentioned an important invariant: if a node operator is allowed to withdraw funds from a minipool, the withdrawal should always succeed. This invariant is meant to assure node operators that they will be able to withdraw their funds even if the system's governance upgrades network contracts to malicious versions.

To withdraw funds from a minipool, a node operator calls the `close` or `refund` function, depending on the state of the minipool. The `close` function calls `rocketMinipoolManager.destroyMinipool`. The `rocketMinipoolManager` contract can be upgraded by governance, which could replace it with a version in which `destroyMinipool` reverts. This would cause withdrawals to revert, breaking the guarantee mentioned above.

The `refund` function does not call any network contracts. However, the `refund` function cannot be used to retrieve all of the funds that `close` can retrieve.

Governance could also tamper with the withdrawal process by altering node operators' withdrawal addresses. (See TOB-ROCKET-001 for more details.)

Exploit Scenario

Alice, a node operator, owns a dissolved minipool and decides to withdraw her funds. However, before Alice calls `close()` on her minipool to withdraw her funds, governance upgrades the `RocketMinipoolManager` contract to a version in which calls to `destroyMinipool` fail. As a result, the `close()` function's call to `RocketMinipoolManager.destroyMinipool` fails, and Alice is unable to withdraw her funds.

Recommendations

Short term, use Solidity's `try catch` statement to ensure that withdrawal functions that should always succeed are not affected by function failures in other network contracts. Additionally, ensure that no important data validation occurs in functions whose failures are ignored.

Long term, carefully examine the process through which node operators execute withdrawals and ensure that their withdrawals cannot be blocked by other network contracts.

5. Lack of contract existence check on delegatecall will result in unexpected behavior

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ROCKET-005
Target: `RocketMinipool.sol`	

Description

The `RocketMinipool` contract uses the `delegatecall` proxy pattern. If the implementation contract is incorrectly set or is self-destructed, the contract may not detect failed executions.

The `RocketMinipool` contract implements a payable fallback function that is invoked when contract calls are executed. This function does not have a contract existence check:

```
    fallback(bytes calldata _input) external payable returns (bytes memory) {
        // If useLatestDelegate is set, use the latest delegate contract
        address delegateContract = useLatestDelegate ?
            getContractAddress("rocketMinipoolDelegate") : rocketMinipoolDelegate;
        (bool success, bytes memory data) = delegateContract.delegatecall(_input);
        if (!success) { revert(getRevertMessage(data)); }
        return data;
    }
```

Figure 5.1: RocketMinipool.sol#L102-L108

The constructor of the `RocketMinipool` contract also uses the `delegatecall` function without performing a contract existence check:

```
        constructor(RocketStorageInterface _rocketStorageAddress, address _nodeAddress,
MinipoolDeposit _depositType) {
        [...]
        (bool success, bytes memory data) =
getContractAddress("rocketMinipoolDelegate").delegatecall(abi.encodeWithSignature('initialis
e(address,uint8)', _nodeAddress, uint8(_depositType)));
        if (!success) { revert(getRevertMessage(data)); }
    }
```

Figure 5.2: RocketMinipool.sol#L30-L43

A `delegatecall` to a destructed contract will return success as part of the EVM specification. The Solidity documentation includes the following warning:

> *The low-level functions call, delegatecall and staticcall return true as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.*

Figure 5.3: A snippet of the Solidity documentation detailing unexpected behavior related to `delegatecall`

The contract will not throw an error if its implementation is incorrectly set or self-destructed. It will instead return success even though no code was executed.

Exploit Scenario

Eve upgrades the `RocketMinipool` contract to point to an incorrect new implementation. As a result, each `delegatecall` returns success without changing the state or executing code. Eve uses this failing to scam users.

Recommendations

Short term, implement a contract existence check before a `delegatecall`. Document the fact that `suicide` and `selfdestruct` can lead to unexpected behavior, and prevent future upgrades from introducing these functions.

Long term, carefully review the Solidity documentation, especially the "Warnings" section, and the pitfalls of using the `delegatecall` proxy pattern.

References

- Contract Upgrade Anti-Patterns
- Breaking Aave Upgradeability

6. tx.origin in RocketStorage authentication may be an attack vector

Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-ROCKET-006
Target: `RocketStorage.sol`	

Description

The `RocketStorage` contract contains all system storage values and the functions through which other contracts write to them. To prevent unauthorized calls, these functions are protected by the `onlyLatestRocketNetworkContract` modifier.

```
    function setUint(bytes32 _key, uint _value) onlyLatestRocketNetworkContract override
external {
        assembly {
            sstore (_key, _value)
        }
    }
```

Figure 6.1: RocketStorage.sol#L205-209

The contract also contains a `storageInit` flag that is set to `true` when the system values have been initialized.

```
    function setDeployedStatus() external {
        // Only guardian can lock this down
        require(msg.sender == guardian, "Is not guardian account");
        // Set it now
        storageInit = true;
    }
```

Figure 6.2: RocketStorage.sol#L89-L94

The `onlyLatestRocketNetworkContract` modifier has a switch and is disabled when the system is in the initialization phase.

```
    modifier onlyLatestRocketNetworkContract() {
        if (storageInit == true) {
            // Make sure the access is permitted to only contracts in our Dapp
```

```
            require(_getBool(keccak256(abi.encodePacked("contract.exists", msg.sender))),
"Invalid or outdated network contract");
        } else {
            // Only Dapp and the guardian account are allowed access during initialisation.
            // tx.origin is only safe to use in this case for deployment since no external
contracts are interacted with
            require((
                _getBool(keccak256(abi.encodePacked("contract.exists", msg.sender))) ||
tx.origin == guardian
            ), "Invalid or outdated network contract attempting access during deployment");
        }
        _;
    }
```

Figure 6.3: `RocketStorage.sol#L36-L48`

If the system is still in the initialization phase, any call that originates from the guardian account will be trusted.

Exploit Scenario

Eve creates a malicious airdrop contract, and Alice, the Rocket Pool system's guardian, calls it. The contract then calls `RocketStorage` and makes a critical storage update. After the updated value has been initialized, Alice sets `storageInit` to `true`, but the storage value set in the update persists, increasing the risk of a critical vulnerability.

Recommendations

Short term, clearly document the fact that during the initialization period, the guardian may not call any external contracts; nor may any system contract that the guardian calls make calls to untrusted parties.

Long term, document all of the system's assumptions, both in the portions of code in which they are realized and in all places in which they affect stakeholders' operations.

7. Duplicated storage-slot computation can silently introduce errors

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ROCKET-007
Target: `Throughout`	

Description

Many parts of the Rocket Pool codebase that access its eternal storage compute storage locations inline, which means that these computations are duplicated throughout the codebase. Many string constants appear in the codebase several times; these include "`minipool.exists`" (shown in figure 7.1), which appears four times. Duplication of the same piece of information in many parts of a codebase increases the risk of inconsistencies. Furthermore, because the code lacks existence and type checks for these strings, inconsistencies introduced into a contract by developer error may not be detected unless the contract starts behaving in unexpected ways.

```
setBool(keccak256(abi.encodePacked("minipool.exists", contractAddress)), true);
```

Figure 7.1: `RocketMinipoolManager.sol#L216`

Many storage-slot computations take parameters. However, there are no checks on the types or number of the parameters that they take, and incorrect parameter values will not be caught by the Solidity compiler.

Exploit Scenario

Bob, a developer, adds a functionality that sets the `network.prices.submitted.node.key` string constant. He ABI-encodes the node address, block, and RPL price arguments but forgets to ABI-encode the effective RPL stake amount. The code then sets an entirely new storage slot that is not read anywhere else. As a result, the write operation is a no-op with undefined consequences.

Recommendations

Short term, extract the computation of storage slots into helper functions (like that shown in 7.2). This will ensure that each string constant exists only in a single place, removing the potential for inconsistencies. These functions can also check the types of the parameters used in storage-slot computations.

```
function contractExistsSlot(address contract) external pure returns (bytes32) {
  return keccak256(abi.encodePacked("contract.exists", contract);
}

// _getBool(keccak256(abi.encodePacked("contract.exists", msg.sender))
_getBool(contractExistsSlot(msg.sender))

// setBool(keccak256(abi.encodePacked("contract.exists", _contractAddress)), true)
setBool(contractExistsSlot(_contractAddress), true)
```

Figure 7.2: An example of a helper function

Long term, replace the raw setters and getters in `RocketBase` (e.g., `setAddress`) with setters and getters for specific values (e.g., the `setContractExists` setter) and restrict `RocketStorage` access to these setters.

8. Potential collisions between eternal storage and Solidity mapping storage slots

Severity: **Informational**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-ROCKET-008
Target: `Throughout`	

Description

The Rocket Pool code uses eternal storage to store many named mappings. A named mapping is one that is identified by a string (such as "`minipool.exists`") and maps a key (like `contractAddress` in figure 8.1) to a value.

```
        setBool(keccak256(abi.encodePacked("minipool.exists", contractAddress)), true);
```

Figure 8.1: `RocketMinipoolManager.sol#L216`

Given a mapping whose state variable appears at index `N` in the code, Solidity stores the value associated with `key` at a `slot` that is computed as follows:

```
h = type(key) == string || type(key) == bytes ? keccak256 : left_pad_to_32_bytes
slot = keccak256(abi.encodePacked(h(key), N))
```

Figure 8.2: Pseudocode of the Solidity computation of a mapping's storage slot

The first item in a Rocket Pool mapping is the identifier, which could enable an attacker to write values into a mapping that should be inaccessible to the attacker. We set the severity of this issue to informational because such an attack does not currently appear to be possible.

Exploit Scenario

Mapping A stores its state variable at slot n. Rocket Pool developers introduce new code, making it possible for an attacker to change the second argument to `abi.encodePacked` in the `setBool` setter (shown in figure 8.1). The attacker passes in a first argument of 32 bytes and can then pass in n as the second argument and set an entry in Mapping A.

Recommendations

Short term, switch the order of arguments such that a mapping's identifier is the last argument and the key (or keys) is the first (as in `keccak256(key, unique_identifier_of_mapping)`).

Long term, carefully examine all raw storage operations and ensure that they cannot be used by attackers to access storage locations that should be inaccessible to them.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization of users or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	Breach of the privacy or integrity of data
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	System failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions, locking, or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices or defense in depth.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is relatively small or is not a risk the customer has indicated is important.
Medium	Individual users' information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client.
High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is commonly exploited; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of a complex system.
High	An attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Categories	**Description**
Access Controls	The authentication and authorization of components
Arithmetic	The proper use of mathematical operations and semantics
Assembly Use	The use of inline assembly
Centralization	The existence of a single point of failure
Upgradeability	Contract upgradeability
Function Composition	The separation of the logic into functions with clear purposes
Front-Running	Resistance to front-running
Key Management	The existence of proper procedures for key generation, distribution, and access
Monitoring	The use of events and monitoring procedures
Specification	The comprehensiveness and readability of codebase documentation and specification
Testing and Verification	The use of testing techniques (e.g., unit tests and fuzzing)

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed, and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.
Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

`RocketStorage.sol`

- **The comment above the `getDeployedStatus` function does not accurately reflect the function's purpose.** Consider fixing the comment.
- **The comment in line 104 actually pertains to line 106**. Consider moving the comment to line 105.

`RocketBase.sol`

- **The assembly in lines 119–122 is unnecessary because array slices would serve the same purpose.** Consider replacing line 123 with the following code:

```
abi.decode(_returnData[4:], (string));
```

`RocketVault.sol`

- **In the `withdrawEther` function, there is an explicit check of the withdrawer's ether balance; however, `withdrawToken` lacks an explicit check verifying that the withdrawer has enough tokens.** The check is implicit, because otherwise, `tokenBalances[contractKey].sub(_amount)` would fail. Consider making the check explicit, making the code more robust, and providing a clearer error message.
- **The `mapping(bytes32 => uint256) tokenBalances` is accessed through `tokenBalances[keccak256(abi.encodePacked(_networkContractName, _tokenAddress))]`, which is error-prone and hard to read.** Consider changing it to `mapping(string => mapping(address => uint256)) tokenBalances`, which is safer and can be accessed more easily via `tokenBalances[_networkContractName][_tokenAddress]`.

`RocketMinipool.sol`

- **The assembly in lines 120–122 is unnecessary because array slices would serve the same purpose.** Consider replacing line 123 with the following:

```
abi.decode(_returnData[4:], (string));
```

- **Line 104 contains a duplicate inline implementation of `getEffectiveDelegate`.** Consider calling `getEffectiveDelegate` to reduce the amount of duplicated code.
- **The function `getDelegate` returns not the delegate that is used but the delegate that is stored.** Consider changing the function's name to `getStoredDelegate` to make its purpose clearer.
- **Line 41 uses `abi.encodeWithSignature('initialise(address,uint8)'`.** Consider replacing it with `abi.encodeWithSelector(RocketMinipoolDelegate.initialise.selector`, which is functionally equivalent but resistant to typos.
- **Line 41 contains a redundant call to `getContractAddress("rocketMinipoolDelegate")`.** Consider replacing it with the variable `rocketMinipoolDelegate`, which is assigned the value returned by **`getContractAddress("rocketMinipoolDelegate")`** in a prior line.

`RocketMinipoolDelegate.sol`

- **Line 142 contains `status >= MinipoolStatus.Initialized && status <= MinipoolStatus.Staking`, which relies on the order of the `MinipoolStatus` enum and is unclear, as one must look up the enum declaration to understand it.** Consider explicitly listing all allowed statuses.
- **The `stake` function contains a redundant call to `setStatus(MinipoolStatus.Staking)`.** Consider removing the second call to **`setStatus`**.
- **`RocketMinipoolDelegate` has a storage status and a minipool status.** It is not clear that the `onlyInitialized` and `onlyUninitialized` functions reference the former status. To make that clearer, consider renaming the functions to `onlyStorageInitialized` and `onlyStorageUninitialized`, respectively.

`RocketMinipoolStorageLayout.sol`

- **Listing the state variables `nodeAddress` and `nodeDepositAssigned` consecutively would allow the Solidity compiler to pack them into a single slot.** Consider listing state variables that fit in a single slot in consecutive order to reduce gas costs.

D. Fix Log

On September 8, 2021, Trail of Bits reviewed the fixes and mitigations implemented by the Rocket Pool team for the issues identified in this report. The Rocket Pool team fixed four of the issues reported in the original assessment and did not fix the other four. We reviewed each of the fixes to ensure that the proposed remediation would be effective. For additional information, please refer to the Detailed Fix Log.

ID	Title	Severity	Fix Status
1	Any network contract can change any node's withdrawal address	High	Fixed (211fddda)
2	Current storage pattern fails to ensure type safety	High	Fixed (211fddda)
3	Solidity compiler optimizations can be problematic	Informational	Not fixed
4	Upgradeable contracts can block minipool withdrawals	High	Not fixed
5	Lack of contract existence check on delegatecall will result in unexpected behavior	High	Fixed (52d948d1)
6	tx.origin in RocketStorage authentication may be an attack vector	High	Not fixed
7	Duplicated storage-slot computation can silently introduce errors	Informational	Not fixed
8	Potential collisions between eternal storage and Solidity mapping storage slots	Informational	Fixed (211fddda)

Detailed Fix Log

TOB-ROCKET-001: Any network contract can change any node's withdrawal address
Fixed. The Rocket Pool team removed the raw `sstore` and `sload` operations from the `RocketStorage` contract. In their place, the code now uses mappings, which do not allow network contracts to access arbitrary storage slots. (211fddda)

TOB-ROCKET-002: Current storage pattern fails to ensure type safety
Fixed. `RocketStorage` now uses a separate mapping for each type, making it impossible for the code to read a value that was written by a setter for another type. (211fddda)

TOB-ROCKET-003: Solidity compiler optimizations can be problematic
Not fixed. The Rocket Pool team responded to this finding as follows: "We are still discussing but acknowledge your concern."

TOB-ROCKET-004: Upgradeable contracts can block minipool withdrawals
Not fixed. The Rocket Pool team responded as follows: "We acknowledge your feedback on refund and close. As long as distributeBalance can always be called, that meets our requirements."

TOB-ROCKET-005: Lack of contract existence check on delegatecall will result in unexpected behavior
Fixed. The Rocket Pool team added a contract existence check before the `delegatecall`. (52d948d1)

TOB-ROCKET-006: tx.origin in RocketStorage authentication may be an attack vector
Not fixed. The Rocket Pool team responded as follows: "We acknowledge the potential issue here and believe our automated build and deployment process mitigates these concern[s]."

TOB-ROCKET-007: Duplicated storage-slot computation can silently introduce errors
Not fixed. The Rocket Pool team responded as follows: "We acknowledge the potential issue. We do not wish to introduce such broad changes at this time as it would introduce a lot of unaudited changes but will endeavour to improve this aspect as we upgrade contracts moving forward."

TOB-ROCKET-008: Potential collisions between eternal storage and Solidity mapping storage slots
Fixed. `RocketStorage` now uses mappings instead of the `sstore` and `sload` raw storage operations, removing the potential for storage-slot conflicts between the mappings. (211fddda)