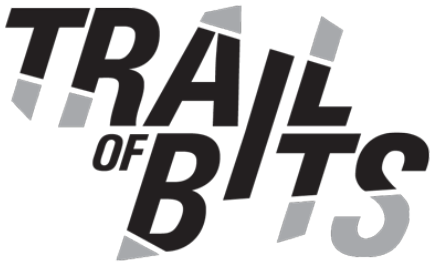

Flexa

Security Assessment

September 20th, 2019

Prepared For:
Flexa Engineering
dev@flexa.network

Prepared By:
Josselin Feist | *Trail of Bits*
josselin.feist@trailofbits.com

Robert Tonic | *Trail of Bits*
robert.tonic@trailofbits.com

Changelog:
September 20th, 2019: Initial report delivered
September 26th, 2019: Added Appendix C with retest results

Executive Summary

From September 16th through September 20th, 2019, Flexa engaged Trail of Bits to review the security of the Flexa staking system. Trail of Bits conducted this assessment over the course of two person-weeks with two engineers working from commit `6128...b54`.

The review focused on the Flexa staking system's Merkle tree operations, where off-chain and on-chain operations complement each other. Situations where system operations and state transitions could lead to lost, trapped, or stolen funds were of particular interest during review. The codebase was reviewed using Slither, a Solidity static analysis tool, and manual review.

In total, seven findings were identified, ranging from high to informational severity. Notably, we found that an integer overflow will cause the withdrawal period to be activated upon the contract's deployment, and that a race condition can allow a user to perform one additional withdrawal if the system is unstable.

To help prevent problems in the future, code quality recommendations have been detailed in Appendix B. These recommendations are oriented towards enhancing code readability and preventing future vulnerabilities from being introduced.

Overall, the staking contract was written with an understanding of common smart contract pitfalls. No critical flaws were found, and most issues were related to improper configuration or abuse of off-chain resources. Improvements could be made in the documentation, particularly the description of the withdrawal procedure and assumptions for nonces.

Flexa should address these findings before production deployment, and consider adding an incident response plan to the system documentation. Trail of Bits recommends an assessment of the off-chain code before deploying the system.

Update: On September 26th, 2019, Trail of Bits reviewed fixes proposed by Flexa for the issues presented in this report. See a detailed review of the current status of each issue in Appendix C.

Project Dashboard

Application Summary

Name	Flexa
Version	`6128af62da3923d007e44637bfb2676c23e7ab54`
Type	Solidity, JavaScript
Platforms	Ethereum

Engagement Summary

Dates	September 16th to September 20th, 2019
Method	Whitebox
Consultants Engaged	2
Level of Effort	2 person-weeks

Vulnerability Summary

Total High-Severity Issues	2	■■
Total Medium-Severity Issues	1	■
Total Low-Severity Issues	1	■
Total Informational-Severity Issues	3	■■■
Total	7	

Category Breakdown

Access Controls	1	■
Auditing and Logging	4	■■■
Data Validation	1	■
Timing	1	■
Total	7	

Engagement Goals

The engagement was scoped to provide a security assessment of the Flexa staking system's Ethereum smart contracts. While the off-chain components of the system were not within the scope of this engagement, the effect of malicious on-chain operations on off-chain components was considered during review.

Specifically, we sought to answer the following questions:

- Can funds become trapped or non-withdrawable?
- Can funds be stolen by a malicious user withdrawing funds?
- Is there any way to cause a Denial of Service for off-chain infrastructure dependent on on-chain infrastructure?
- Can a malicious user exploit the on-chain Merkle tree operations?
- Can the fallback root operations be manipulated maliciously?
- Can the withdrawal root operations be manipulated maliciously?

Coverage

Merkle tree. We looked for flaws in the Merkle tree computation, including incorrect proofs, validations, and potential out-of-gas exceptions.

Withdrawal. We reviewed the nonce usage to ensure that a withdrawal could not take place multiple times. We took into consideration front-running and replay attacks.

Withdrawal fallback. We checked that a deposit could not be withdrawn if accepted by the Merkle tree, and we looked for ways to bypass the fallback mechanism restrictions.

Only the smart contract was reviewed. We did not assess the off-chain code, and assumed the correct Merkle tree generation and reporting. We assumed that deposits are included in the fallback root before their amount are included in the standard withdrawal tree.

Additionally, Flexa was aware of the loss or benefit of funds due to the off-chain/on-chain balance's desynchronisation reached after the withdrawal period.

Recommendations Summary

This section combines all the recommendations made during the engagement. Short-term recommendations address the immediate issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Change the initial value of `_fallbackSetDate` (Staking.sol#L41) to prevent overflow with `_fallbackWithdrawalDelaySeconds` (Staking.sol#L25).** The overflow will trigger the withdrawal period upon the contract's deployment.

❑ **Prevent overflow on the withdrawal period verification in `withdrawFallback` and `refundPendingDeposit`.** The overflow will trigger the withdrawal period upon the contract's deployment.

❑ **Document how to update the fallback root if the withdrawal period is active.** To prevent additional withdrawals, the update should be split into two calls:

- the first call setting to the previous root, and
- the second to the updated one.

❑ **Add the nonce to the `PendingDepositRefund` event. Ensure the off-chain code uses it.** The lack of nonce information can mislead off-chain code to think another deposit was removed.

❑ **Ensure documentation is provided on the expected functionality of the withdrawal root–related functions.** This should include the potential for duplicate roots to be posted.

❑ **Ensure all inputs are appropriately validated for their zero values, including:**

- `newOwnerAddress != 0` in `setOwner` (Staking.sol#L362-370)
- `newFallbackDelaySeconds != 0` in `setFallbackWithdrawalDelay` (Staking.sol#L442-L451)
- `root != 0` in `addWithdrawalRoot` (Staking.sol#L460-L488) and `setFallbackRoot` (Staking.sol#L517-L540)

The lack of a zero-value check might easily lead to incorrect configuration of the contract.

❑ **Ensure in the off-chain code that the `newOwnerAddress` is always under control before invocation of `setOwner`.** `setOwner` will set the new owner without any verification of the destination. Calling `setOwner` with an incorrect value may be an irrevocable error.

❑ **Use the Checks-Effects-Interactions pattern for all the token interactions (`deposit`, `withdraw`, `withdrawFallback`, `refundPendingDeposit`).** The current operations order might allow reentrancies.

Long Term

❑ **Use `SafeMath` for all arithmetic operations.** It is a best practice to use `SafeMath` to prevent arithmetic issues.

❑ **Write an incident response plan that includes strategies to handle compromise and network congestion.** Having a procedure to follow will prevent errors when reacting quickly to a compromise.

❑ **Document every field that is needed for the off-chain code and ensure their information is correctly emitted to the on-chain code.** Any assumption on the off-chain <-> on-chain interactions must be thoroughly documented to be properly verified.

❑ **To prevent a deleted root from being added again, consider:**

- **Adding a method of validating whether a root has been previously encountered, or**
- **Adding the nonce as a leaf of the root.**

Adding a previously deleted root might lead to unexpected behavior for third parties.

❑ **Expand testing to include zero-value tests and ensure validation is appropriate even for unexpected inputs. Document the expected inputs for each function.** Lack of input validation can easily lead to incorrect configuration of the contract.

❑ **Use a two-step process of `transferOwnership` and `acceptOwnership` to ensure an address is controllable before confirming ownership transfer.** `setOwner` will set the new owner without any verification of the destination. Calling `setOwner` with an incorrect value may be an irrevocable error.

❑ **Run Slither continuously on the codebase or use crytic.io.** Slither will detect the common Solidity flaws, such as reentrancy (TOB-Flexa-007).

Findings Summary

#	Title	Type	Severity
1	Initial configuration may allow an attacker to refund an unconfirmed deposit early on	Data Validation	Low
2	Front-running fallback root update might lead to additional withdrawal	Timing	High
3	Missing nonce on PendingDepositRefund event might lead to a double spend	Auditing and Logging	High
4	A withdrawal root could be added again after removal	Data Validation	Informational
5	Missing validations on administration functions	Data Validation	Medium
6	setOwner should be split into two separate functions	Access Controls	Informational
7	Reentrancy could cause incorrect information to be emitted	Data Validation	Informational

1. Initial configuration may allow an attacker to refund an unconfirmed deposit early on

Severity: Low
Type: Data Validation
Target: Staking.sol

Difficulty: Medium
Finding ID: TOB-Flexa-001

Description
Integer overflow allows users to refund deposits during the initial `Staking` configuration.

To refund a deposit, the withdrawal period must be active:

```
function refundPendingDeposit(uint256 depositNonce) external {
    address depositor = _nonceToPendingDeposit[depositNonce].depositor;
    require(
        msg.sender == _owner || msg.sender == depositor,
        "Only the owner or depositor can initiate the refund of a pending deposit"
    );
    require(
        _fallbackSetDate + _fallbackWithdawalDelaySeconds <= block.timestamp,
        "Fallback withdrawal period is not active, so refunds are not permitted"
    );
```

Figure 1.1: refundPendingDeposit (Staking.sol#L327-L336).

The withdrawal period check is vulnerable to an integer overflow:

```
_fallbackSetDate + _fallbackWithdawalDelaySeconds <= block.timestam
```

The initial value of `_fallbackSetDate` is 2**(256-1) and `_fallbackWithdawalDelaySeconds` is `1 weeks :`

```
uint256 public _fallbackSetDate = 2^256-1;
```

Figure 1.2: _fallbackSetDate initial value (Staking.sol#L41.)

```
uint256 public _fallbackWithdawalDelaySeconds = 1 weeks;
```

Figure 1.3: _fallbackWithdawalDelaySeconds initial value (Staking.sol#L25).

These values will trigger the integer overflow. As a result, the withdrawal period is enabled upon the contract's deployment. Additionally, the overflow can be reached by an incorrect configuration.

Exploit Scenario

Bob deploys the contract. Bob plans to wait a few days before publishing the fallback withdrawal root. Eve spams the network with deposits that she withdraws immediately.

Recommendation
Short term, change the initial value of `_fallbackSetDate` to prevent the overflow. Prevent overflow on the withdrawal period verification in `withdrawFallback` and `refundPendingDeposit`.

Long term, use `SafeMath` for all arithmetic operations.

2. Front-running fallback root update might lead to additional withdrawal

Severity: High
Type: Timing
Target: Staking.sol

Difficulty: High
Finding ID: TOB-Flexa-002

Description
A front-running attack might allow an attacker to withdraw a deposit one additional time if the system is not updated before the fallback withdrawal period is reached.

The fallback withdrawal mechanism allows users to withdraw their deposits if the system is not updated after a given period. If the fallback root is updated at the same time that the fallback withdrawal period is active, an attacker can front-run the update and withdraw deposits that will be included by the update. If the fallback tree is not updated again, and the withdrawal period is activated a second time, the attacker will be able to withdraw the funds once more.

Exploit Scenario
- The fallback withdrawal period is reached.
- Eve creates a deposit of $10,000.
- Bob updates the fallback root, and include Eve's deposit.
- Eve front runs Bob's transaction, and withdraws her deposit.
- Bob is not able to update the fallback root. The fallback withdrawal period is reached again.
- Eve withdraws her funds once more.

Recommendation
Short term, we recommend updating the fallback root in two calls if the withdrawal period is active:
- The first call will set a tree that does not contain any deposits that were not already included.
- The second call will set the tree that contains all the deposits that were validated on the block that contained the first call.

Long term, write an incident response plan that includes strategies to handle compromise and network congestion.

3. Missing nonce on PendingDepositRefund event might lead to a double spend

Severity: High
Type: Auditing and Logging
Target: `Staking.sol`

Difficulty: Undetermined
Finding ID: TOB-Flexa-003

Description
A lack of nonce information on the deposit refund event might allow a third-party tool to confound which deposit was refunded.

A deposit is identified by its unique nonce:

```
depositNonce = ++_depositNonce;
_nonceToPendingDeposit[depositNonce].depositor = msg.sender;
_nonceToPendingDeposit[depositNonce].amount = amount;
```

Figure 3.1: Deposit creation (Staking.sol#L160-L162).

Upon deposit refund, `PendingDepositRefund` is emitted:

```
emit PendingDepositRefund(depositor, amount);
```

Figure 3.2: `PendingDepositRefund` *event Staking.sol#L351).*

The event does not include the nonce. As a result, third-party tools watching the event might confound which deposit was refunded.

Trail of Bits did not review the Flexa off-chain code, and could not assess the likelihood of the issue occurring in the deposit off-chain triage code.

Exploit Scenario

- Eve has two pending deposits of $10,000, one with a nonce of 10 (`A`) and one with a nonce of 20 (`B`).
- The withdrawal fallback period is active.
- Eve refunds Deposit `A` by calling `refundPendingDeposit` (Eve has $10,000).
- The off-chain code decodes the event, and thinks `B` was refunded. Deposit `A` is accepted off-chain.
- Eve unlocks the funds from Deposit A and withdraws them (Eve has $20,000).
- The system fails, and the withdraw fallback period is active again.
- Even calls refundPendingDeposit for Deposit B, and receives $10,000.
- Eve ends up with $30,000 for $20,000 invested.

Recommendation

Short term, add the nonce to the `PendingDepositRefund` event. Ensure the off-chain code uses it.

Long term, thoroughly document every field that is needed for the off-chain code and ensure their information is correctly emitted in the on-chain code.

4. A withdrawal root could be added again after removal

Severity: Informational
Type: Data Validation
Target: `Staking.sol`

Difficulty: Low
Finding ID: TOB-Flexa-004

Description
A deleted withdrawal root can be added again, leading to unexpected behavior for users.

Within the `addWithdrawalRoot` function there are validations to prevent a withdrawal root from being added if it is already present. However, in the event of a withdrawal root being removed and added again, there is no method to track whether the root has been previously added.

```
   function addWithdrawalRoot(
        bytes32 root,
        uint256 nonce,
        bytes32[] calldata replacedRoots
    ) external {
        require(
            msg.sender == _owner || msg.sender == _withdrawalPublisher,
            "Only the owner and withdrawal publisher can add and replace withdrawal root
hashes"
        );


        require(
            _maxWithdrawalRootNonce + 1 == nonce,
            "Nonce must be exactly max nonce + 1"
        );


        require(
            _withdrawalRootToNonce[root] == 0,
            "Root already exists and is associated with a different nonce"
        );


        _withdrawalRootToNonce[root] = nonce;
        _maxWithdrawalRootNonce = nonce;


        emit WithdrawalRootHashAddition(root, nonce);


        for (uint256 i = 0; i < replacedRoots.length; i++) {
            deleteWithdrawalRoot(replacedRoots[i]);
        }
    }
```

Figure 4.1: The `addWithdrawalRoot` *function definition.*

Exploit Scenario

- Bob adds the Merkle root AAAA with the nonce 10.
- Bob removes the root, and adds it again, with the nonce 11.
- As a result, users relying on AAAA are confused.

Recommendation
Short term, ensure documentation is provided on the expected functionality of the withdrawal root–related functions, including the potential for duplicate roots to be posted.

Long term, consider either:
- Adding a method of validating whether a root has been previously encountered, or
- Adding the nonce as a leaf of the root.

5. Missing validations on administration functions

Severity: Medium
Type: Data Validation
Target: `Staking.sol`

Difficulty: High
Finding ID: TOB-Flexa-005

Description
Staking relies on correct parametrization from the owner. Several administration functions lack proper input validation, which might lead to a misconfigured system or loss of privileged access.

Within the `setOwner` function (Figure 5.1), there is no check to ensure the `0x0` address is not provided as the `newOwnerAddress` parameter. This could lead to an accidental invocation of `setOwner` with an uninitialized value, resulting in an irrevocable loss of contract ownership.

```
function setOwner(address newOwnerAddress) external {
     require(
         msg.sender == _owner,
         "Only the owner can set the new owner"
     );
     address oldValue = _owner;
     _owner = newOwnerAddress;


     emit OwnerUpdate(oldValue, _owner);
 }
```

Figure 5.1: The `setOwner` function definition.

Additional validations are missing within the `addWithdrawalRoot` (Figure 5.2) and setFallbackRoot (Figure 5.3) functions, where the `root` parameter could potentially be `0` in both, and the `nonce` can potentially be `0`. In the `setFallbackWithdrawalDelay` function (Figure 5.4), the `newFallbackDelaySeconds` could be `0`.

```
   function addWithdrawalRoot(
        bytes32 root,
        uint256 nonce,
        bytes32[] calldata replacedRoots
    ) external {
        require(
            msg.sender == _owner || msg.sender == _withdrawalPublisher,
            "Only the owner and withdrawal publisher can add and replace withdrawal root
hashes"
        );


        require(
            _maxWithdrawalRootNonce + 1 == nonce,
            "Nonce must be exactly max nonce + 1"
```

```
        );


        require(
            _withdrawalRootToNonce[root] == 0,
            "Root already exists and is associated with a different nonce"
        );


        _withdrawalRootToNonce[root] = nonce;
        _maxWithdrawalRootNonce = nonce;


        emit WithdrawalRootHashAddition(root, nonce);


        for (uint256 i = 0; i < replacedRoots.length; i++) {
            deleteWithdrawalRoot(replacedRoots[i]);
        }
    }
```

Figure 5.2: The `addWithdrawalRoot` *function definition.*

```
    function setFallbackRoot(bytes32 root, uint256 maxDepositIncluded) external {
        require(
            msg.sender == _owner || msg.sender == _fallbackPublisher,
            "Only the owner and fallback publisher can set the fallback root hash"
        );
        require(
            maxDepositIncluded >= _fallbackMaxDepositIncluded,
            "Max deposit included must remain the same or increase"
        );
        require(
            maxDepositIncluded <= _depositNonce,
            "Cannot invalidate future deposits"
        );


        _fallbackRoot = root;
        _fallbackMaxDepositIncluded = maxDepositIncluded;
        _fallbackSetDate = block.timestamp;


        emit FallbackRootHashSet(
            root,
            _fallbackMaxDepositIncluded,
            block.timestamp
        );
    }
```

Figure 5.3: The `setFallbackRoot` *function definition.*

```
    function setFallbackWithdrawalDelay(uint256 newFallbackDelaySeconds) external {
        require(
            msg.sender == _owner,
            "Only the owner can set the fallback withdrawal delay"
        );
        uint256 oldDelay = _fallbackWithdawalDelaySeconds;
```

```
        _fallbackWithdawalDelaySeconds = newFallbackDelaySeconds;

        emit FallbackWithdrawalDelayUpdate(oldDelay, newFallbackDelaySeconds);
    }
```

Figure 5.4: The `setFallbackWithdrawalDelay` *function definition.*

Exploit Scenario

- Alice deploys the `Staking` contract.
- She calls `setOwner`, but incorrectly sets the owner to zero.
- As a result, she loses administration access.

Recommendation

Short term, ensure all inputs are appropriately validated for their zero values, including:

- `newOwnerAddress != 0` in `setOwner` (Staking.sol#L362-370).
- `newFallbackDelaySeconds != 0` in `setFallbackWithdrawalDelay` (Staking.sol#L442-L451).
- `root != 0` in `addWithdrawalRoot` (Staking.sol#L460-L488) and `setFallbackRoot` (Staking.sol#L517-L540).

Long term, expand testing to include zero-value tests and ensure validation is appropriate even for unexpected inputs. Document the expected inputs for each function.

6. setOwner should be split into two separate functions

Severity: Informational
Type: Access Controls
Target: `Staking.sol`

Difficulty: High
Finding ID: TOB-FLX-006

Description
`setOwner` changes ownership of the contract in a single transaction. If an incorrect `newOwnerAddress` is provided, ownership may never be recovered. A best practice is to split the ownership into two functions: transfer and accept.

By splitting the functionality of `setOwner` into two functions—`transferOwnership` and `acceptOwnership`—the original owner will retain owner abilities until the new owner calls `acceptOwnership`. This will prevent accidental transfer of ownership to an uncontrolled address.

```
function setOwner(address newOwnerAddress) external {
     require(
         msg.sender == _owner,
         "Only the owner can set the new owner"
     );
     address oldValue = _owner;
     _owner = newOwnerAddress;


     emit OwnerUpdate(oldValue, _owner);
 }
```

Figure 6.1: The `setOwner` *function definition.*

Exploit Scenario
- Alice deploys the `Staking` contract, then decides to change the owner to another address under her control.
- Subsequently, she enters the new address as the `newOwnerAddress`, but mistakenly enters the last hex value of the address incorrectly.
- Upon invocation of `setOwner` with the malformed input, Alice loses all ownership of the contract.

Recommendation
Short term, ensure in the off-chain code that the `newOwnerAddress` is always under control before invocation of `setOwner`.

Long term, use a two-step process of `transferOwnership` and `acceptOwnership` to ensure an address is controllable before confirming ownership transfer.

7. Reentrancy could cause incorrect information to be emitted

Severity: Informational
Type: Data Validation
Target: Staking.sol

Difficulty: Undetermined
Finding ID: TOB-Flexa-007

Description
Reentrancies on token transfer might trigger the incorrect order of events. The reentrancies require a token with external call capabilities, which is not possible with the current token implementation.

After each token transfer, an event is emitted. If the token transfer is made on a contract with external call capabilities (such as ERC223 or ERC777), a reentrancy might cause events to be emitted in the wrong order:

```
        bool transferred = myIERC20(_tokenAddress).transfer(
            toAddress,
            amount
        );


        require(transferred, "Transfer failed");


        emit Withdrawal(
            toAddress,
            amount,
            withdrawalPermissionRootNonce,
            maxAuthorizedAccountNonce
        );
```

Figure 7.1: withdraw (Staking.sol#L251-L263).

```
        bool transferred = myIERC20(_tokenAddress).transfer(
            depositor,
            amount
        );
        require(transferred, "Transfer failed");


        emit PendingDepositRefund(depositor, amount);
```

Figure 7.2: refundPendingDeposit (Staking.sol#L345-L351).

```
        bool transferred = myIERC20(_tokenAddress).transfer(
            toAddress,
```

```
            withdrawalAmount
        );


        require(transferred, "Transfer failed");


        emit FallbackWithdrawal(
            toAddress,
            withdrawalAmount
        );
    }
```

Figure 7.3: withdrawFallback (Staking.sol#L309-L320).

```
        bool transferred = myIERC20(_tokenAddress).transferFrom(
            msg.sender,
            address(this),
            amount
        );
        require(transferred, "Transfer failed");


        depositNonce = ++_depositNonce;
        _nonceToPendingDeposit[depositNonce].depositor = msg.sender;
        _nonceToPendingDeposit[depositNonce].amount = amount;


        emit Deposit(
            msg.sender,
            amount,
            depositNonce
        );
```

Figure 7.4: deposit (Staking.sol#L153-L168).

In addition, the nonce associated with a deposit might not follow the correct order (Figure 7.4).

Exploit Scenario

`Staking` is deployed with an ERC777 token. Eve uses the reentrancy to trigger incorrect events and confuse the staking off-chain monitor.

Recommendation

Short term, use the Checks-Effects-Interactions pattern for all the token interactions (`deposit`, `withdraw`, `withdrawFallback`, `refundPendingDeposit`).

Long term, continuously run Slither on the codebase or use crytic.io.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

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 would be bad for

	client's reputation, moderate financial impact, possible legal implications for client
High	Affects large number of users, exploitation would be very bad for client's reputation, or serious legal or financial implications

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

B. Code Quality

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

Staking.sol

- **Use Modifiers for repetitive checks**. Modifiers should be defined for the common require statements used against `msg.sender`. This will help improve maintenance, and shorten function definitions overall. Furthermore, this will help prevent situations where double requisites are performed, such as in the `withdraw` function, where the same require seems to have been repeated.
- **Use `SafeMath`**. There are several areas in which overflows could be prevented, alleviating the potential for undefined behavior after an overflow.
- **Ensure variable naming is clear and concise,** with all documentation referencing the same terminology and verbiage.
- **Remove the fallback function**. Currently, it is unnecessary. If the fallback function is not present, the code will revert and not accept `ETH`.
- **Remove the duplicate check on** `amount <= _immediatelyWithdrawableLimit` (Staking.sol#L225-L228) **within the `withdraw` function.** The second check is unnecessary.
- **Rename IERC20**. IERC20 is used by the `openzeppelin` dependency. A known bug in Truffle prevents the framework to generate correct artifacts in case of contract's name duplicate. This issue prevent the direct use of crytic-compile-based tools (Slither, Echidna, Manticore).

C. Fix Log

On September 26th, 2019, Trail of Bits reviewed the fixes proposed by Flexa regarding the issues presented in this report.

Flexa remediated five issues, mitigated one issue off-chain, and accepted the risk of one issue. Trail of Bits only reviewed code in the smart contract; therefore, mitigations that are implemented off-chain were not reviewed. Each smart contract code change was reviewed to understand the comprehensiveness of the proposed remediation. The original commit under review was `6128...ab54`, and the commit containing the reviewed fixes is `12ab...94cb`.

ID	Title	Severity	Status
01	Initial configuration may allow an attacker to refund an unconfirmed deposit early on	Low	Fixed
02	Front-running fallback root update might lead to additional withdrawal	High	Fixed
03	Missing nonce on PendingDepositRefund event might lead to a double spend	High	Fixed
04	A withdrawal root could be added again after removal	Informational	Risk Accepted
05	Missing validations on administration functions	Medium	Fixed
06	setOwner should be split into two separate functions	Informational	Fixed
07	Reentrancy could cause incorrect information to be emitted	Informational	Fixed

Flexa also addressed several code quality recommendations. Specifically, Flexa:

- Renamed `IERC20` to `ERC20Token` to prevent the known Truffle bug.
- Removed the redundant check in `withdraw(...)`.
- Removed the fallback function.

Detailed Fix Log

Finding 1: Initial configuration may allow an attacker to refund an unconfirmed deposit early on
Fixed. The overflowable operation has been replaced with the `SafeMath` equivalent, which will revert upon overflow.

Finding 2: Front-running fallback root update might lead to additional withdrawal
Fixed. To correct this issue, the `resetFallbackMechanismDate` function has been added, which sets the fallback date to the current block timestamp at the time of invocation. Then, all deposits made prior to the mined timestamp are integrated into the root off-chain and published to chain. This allows both on- and off-chain components to prevent the additional withdrawal.

Finding 3: Missing nonce on PendingDepositRefund event might lead to a double spend
Fixed. The missing nonce has been added to the emitted event.

Finding 4: A withdrawal root could be added again after removal
Risk Accepted. This is a feature reserved for use by Flexa. The Flexa team stated:

> *Anything regarding TOB-Flexa-004, as the ability to publish previous roots is a feature required by Flexa, not a potential bug that should be prevented. For instance:*
>
> 1. *A & B deposit*
> 2. *A requests withdrawal, Merkle root R is published, A's balance is the only balance in withdrawal Merkle tree*
> 3. *B requests withdrawal, Merkle root R' is published, A & B both have balances in the withdrawal Merkle tree*
> 4. *B renounces their withdrawal authorization, Merkle root R is published, A's balance is the only balance in withdrawal Merkle tree*

Finding 5: Missing validations on administration functions
Fixed. The `setOwner` function has been replaced with `authorizeOwnershipTransfer` and `assumeOwnership`, preventing loss of contract ownership due to the lack of validation. Appropriate events have also been added. Validation of the `root` parameter has been added for `addWithdrawalRoot` and `setFallbackRoot`, preventing a `root` value of `0`. Similarly, `setFallbackWithdrawalDelay` now has validations for the `newFallbackDelaySeconds` parameter, and no longer allows a `0` value.

Finding 6: setOwner should be split into two separate functions
Fixed. The `setOwner` function has been replaced with the `authorizeOwnershipTransfer` and `assumeOwnership` functions. This prevents Flexa from transferring ownership to an

address not under their control, and allows ownership to be retained until the authorized address invokes `assumeOwnership`. Appropriate events are also emitted: `OwnershipTransferAuthorization` is emitted when the `authorizeOwnershipTransfer` function is invoked, and `OwnerUpdate` is emitted when `assumeOwnership` is invoked.

Finding 7: Reentrancy could cause incorrect information to be emitted
Fixed. Flexa has adopted the Checks-Effects-Interactions pattern to prevent reentrancy and help ensure correct information will be emitted throughout the contract. Appropriate changes have been made to the codebase that reflect the adoption of this practice.