ackee | blockchain security

Immunefi

Vault

by Ackee Blockchain

28.2.2024

Contents

1. Document Revisions 4
2. Overview 5
    2.1. Ackee Blockchain 5
    2.2. Audit Methodology 5
    2.3. Finding classification 6
    2.4. Review team 8
    2.5. Disclaimer 8
3. Executive Summary 9
    Revision 1.0 9
    Revision 1.1 10
4. Summary of Findings 12
5. Report revision 1.0 15
    5.1. System Overview 15
    5.2. Trust Model 22
    M1: It is possible to exit with funds during ongoing arbitration 24
    L1: Insufficient data validation when composing contracts 26
    L2: Chainlink feed registry will break on other chains than mainnet 28
    L3: Chainlink validation is not sufficient for L2 deployments 29
    L4: Transaction cooldown can overflow 31
    L5: The `canExecuteTransaction` doesn't check if the vault is frozen 32
    W1: Introduced smart contract logic can be still bypassed 33
    W2: Timelock withdrawal delay can be bypassed 34
    W3: Hardcoded slippage protection on queued rewards 36
    W4: Requesting arbitration by vault is not possible when using fee-on-transfer tokens 37
    W5: Old Safe version is used 38

W6: Sending reward can be front-ran and blocked by arbitration 40
I1: Misleading reference type or naming 42
I2: Withdrawal and reward requests stay opened after expiration 43
I3: Use of custom errors 44
I4: Commented out code 45
I5: The `getWithdrawalHash` function must be called prior to the withdrawal request to be functional 46
I6: Incorrect requirement in Natspec for vault request for arbitration 47
I7: Use pre-incrementation in for cycles 48
I8: Unnecessary lookup in for cycles 49
6. Report revision 1.1 50
6.1. System Overview 50
Appendix A: How to cite 51
Appendix B: Glossary of terms 52
Appendix C: Wake outputs 53
C.1. Detectors 53
C.2. Tests 54
Appendix D: Files included in the audit scope 58

1. Document Revisions

0.1	Draft report	15.2.2024
1.0	Final report	28.2.2024
1.1	Fix review	28.2.2024

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, RockawayX.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Wake is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzz testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzz tests.

2.3. Finding classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**-**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review team

Member’s Name	Position
Jan Kalivoda	Lead Auditor
Lukas Böhm	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Immunefi is a platform for bug bounties. Newly introduced Vault contracts serve for better transparency and they should increase trust in paying out the rewards from listed projects to whitehats.

Revision 1.0

Immunefi engaged Ackee Blockchain to perform a security review of the Immunefi protocol with a total time donation of 15 engineering days in a period between January 22 and February 15, 2024, with Jan Kalivoda as the lead auditor.

The audit was performed on the commit `30540a0` [1] and the scope was the full repository, excluding mocks (for comprehensive list see Appendix D).

We began our review using static analysis tools, including Wake. We then took a deep dive into the logic of the contracts. For testing and fuzzing, we have involved Wake testing framework (for more outputs see Appendix C). Most of the issues were found with manual review and proof of concepts were implemented with unit tests. Fuzz tests discovered minor issues such as I5: The `getWithdrawalHash` function must be called prior to the withdrawal request to be functional, but overall added a bigger confidence to the audit result. During the review, we paid special attention to:

- ensuring funds in Vaults can not be maliciously stolen through ImmunefiModule,
- ensuring Safe guards are correctly scoped and can not be bypassed,
- exploitations of the arbitration process,
- race conditions in timelocks and correct handling of timelock requests,
- testing the codebase with a variety of unusual ERC-20 tokens,

- ensuring the arithmetic of the system is correct,
- detecting possible reentrancies in the code,
- ensuring access controls are not too relaxed or too strict,
- looking for common issues such as data validation.

Our review resulted in 20 findings, ranging from Info to Medium severity.

The code quality of the protocol is good, however, a lot of warnings and minor issues indicate an unnecessarily complex codebase. The main problem is too centralized power (see Trust Model) and too much flexibility that can be used to circumvent intended behavior (see M1: It is possible to exit with funds during ongoing arbitration).

Ackee Blockchain recommends Immunefi:

- define precisely the purpose of these contracts and be more strict in the contract logic,
- reconsider the need for so many elevated privileges in the contracts,
- address all other reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

The client provided the repository with the updated codebase including fixes on the given commit: `0120746` [2]. The fix review was performed on February 23, 2024.

See the summary of the findings for the current status of issues.

See Revision 1.1 for the review of the updated codebase and additional information we consider essential for the current scope.

[1] full commit hash: `30540a009ecdd98b9a854a8796897d62a3a1ecfe`

[2] full commit hash: `0120746440fc277372800f500ad822be93b540f5`

ackee | blockchain security

4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Fix*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
M1: It is possible to exit with funds during ongoing arbitration	Medium	1.0	Acknowledged
L1: Insufficient data validation when composing contracts	Low	1.0	Acknowledged
L2: Chainlink feed registry will break on other chains than mainnet	Low	1.0	Fixed
L3: Chainlink validation is not sufficient for L2 deployments	Low	1.0	Fixed

	Severity	Reported	Status
L4: Transaction cooldown can overflow	Low	1.0	Fixed
L5: The `canExecuteTransaction` doesn't check if the vault is frozen	Low	1.0	Fixed
W1: Introduced smart contract logic can be still bypassed	Warning	1.0	Acknowledged
W2: Timelock withdrawal delay can be bypassed	Warning	1.0	Acknowledged
W3: Hardcoded slippage protection on queued rewards	Warning	1.0	Acknowledged
W4: Requesting arbitration by vault is not possible when using fee-on-transfer tokens	Warning	1.0	Acknowledged
W5: Old Safe version is used	Warning	1.0	Acknowledged
W6: Sending reward can be front-ran and blocked by arbitration	Warning	1.0	Acknowledged
I1: Misleading reference type or naming	Info	1.0	Fixed
I2: Withdrawal and reward requests stay opened after expiration	Info	1.0	Acknowledged

	Severity	Reported	Status
I3: Use of custom errors	Info	1.0	Acknowledged
I4: Commented out code	Info	1.0	Fixed
I5: The `getWithdrawalHash` function must be called prior to the withdrawal request to be functional	Info	1.0	Fixed
I6: Incorrect requirement in Natspec for vault request for arbitration	Info	1.0	Fixed
I7: Use pre-incrementation in for cycles	Info	1.0	Acknowledged
I8: Unnecessary lookup in for cycles	Info	1.0	Fixed

Table 2. Table of Findings

5. Report revision 1.0

5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

Contracts

Contracts we find important for better understanding are described in the following section.

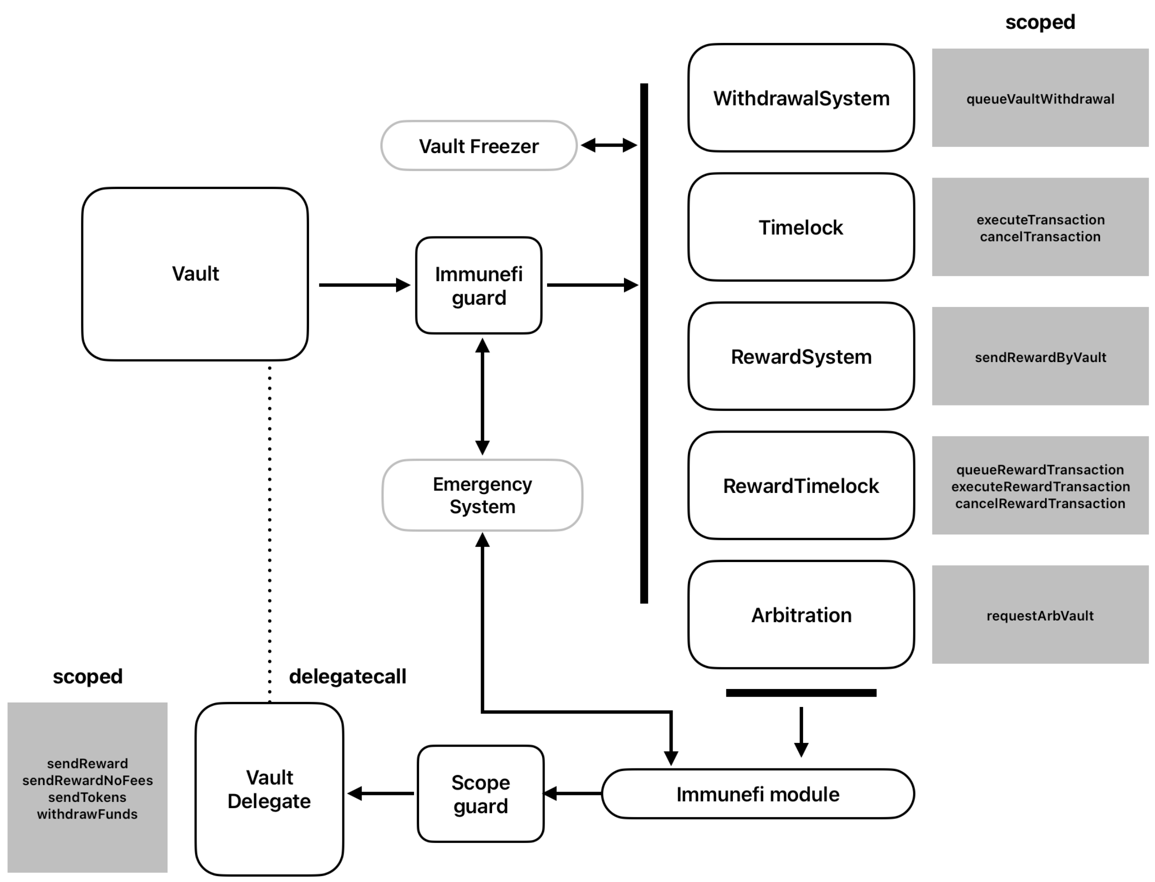


Figure 1. Simplified architecture overview with highlighted scopes

ImmunefiModule

An upgradeable contract that serves as a Safe module with an emergency shutdown feature. For each execution, it checks transactions with ScopeGuard (if set).

ImmunefiGuard

A contract that represents a Safe guard. It validates transactions in the protocol. It inherits from ScopeGuard and adds an emergency shutdown feature and mapping of guard bypassers.

ScopeGuard

This contract is expected to be used as a Safe guard. It allows to specify which addresses and functions are allowed to be called. It is used in two instances. The first is child contract ImmunefiGuard where it allows to call only protocol components. The second instance is a guard of ImmunefiModule where it allows to call only VaultDelegate with delegate call and only some specific functions (see Figure 1).

RewardTimelock

An upgradeable contract that can queue reward transactions. It is used during arbitrations to payout rewards with a delay. Requests are denominated in dollars and thus it is important to choose the correct amount of token for executions.

The calculation is done followingly:

$$X = \frac{A_e \cdot P_e}{10^{18}} + \sum_{i=0}^{n} \frac{A_t(i) \cdot P_t(i)}{D_t(i)}$$

Figure 2. Total dollar amount

where, P_e is a **price for a native token** with 18 decimals, and A_e is an **amount** (18 decimals) for **native token** calculation. Then there is a sum of all chosen tokens up to n , where A_t is an array of the **token amounts** (unknown decimals), P_t is an array of retrieved the **token prices** (18 decimals), and D_t is an array of the **token decimals**.

And the resulting amount (18 decimals) X must hit the following condition:

$$\left(\frac{X_0 \cdot 9900}{10000}\right) \leq X \leq \left(\frac{X_0 \cdot 10100}{10000}\right)$$

Figure 3. Hardcoded slippage protection

where, X_0 is the initial dollar amount from the request (scaled to 18 decimals).

In other words, we can take any number of tokens including the native token to achieve the desired dollar amount.

Timelock

An upgradeable contract that works similarly to RewardTimelock but is more flexible. It can be used for delayed withdrawals (and thus as well as delayed payouts bypassing Immunefi fees).

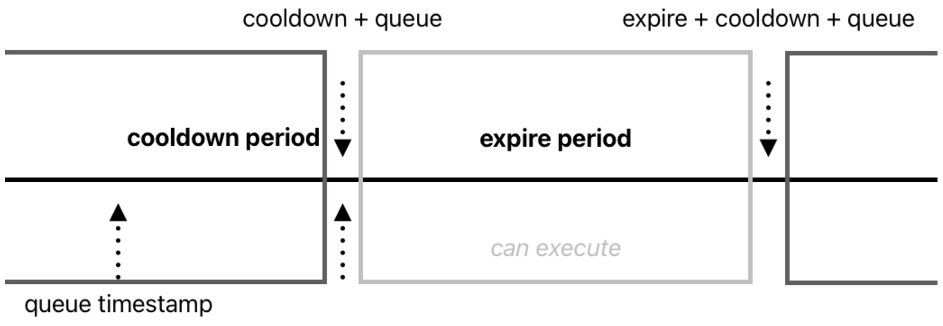


Figure 4. Timelock and RewardTimelock execution window

RewardSystem

An upgradeable contract that manages rewards payouts. These rewards can be initiated by Vault or by Enforcer.

WithdrawalSystem

An upgradeable contract that works as an entrypoint to Timelock. It can queue withdrawals to be executed in Timelock.

ProxyAdminOwnable2Step

Ownable contract that is used as admin for transparent proxies.

Arbitration

An upgradeable contract that is used for arbitrations between whitehats and vaults. Arbitration requests are of two types. The first can be requested by anyone for Whitehat side and the second should be requested by Vault for Vault. The difference is in who pays the arbitration fee. Rewards can be enforced by Arbiter to be sent (with or without fees) to Fee recipient, Whitehat or to Allowed recipient. The contract is governed by Admin. Arbiter can also close arbitration requests.

PriceConsumer

A contract that uses Chainlink oracle (Feed registry) to retrieve prices for tokens in vaults (denominated in USD). It also can use some undefined custom oracle that is expected to match the following interface:

```
IPriceFeed(feed).getUsdPrice() returns (int256 price, uint256 updatedAt)
```

VaultFreezer

An upgradeable contract that can freeze vaults. Freezing doesn't apply to executions triggered by Arbiter during arbitration, otherwise it prevents Vault from executing transactions including withdrawals.

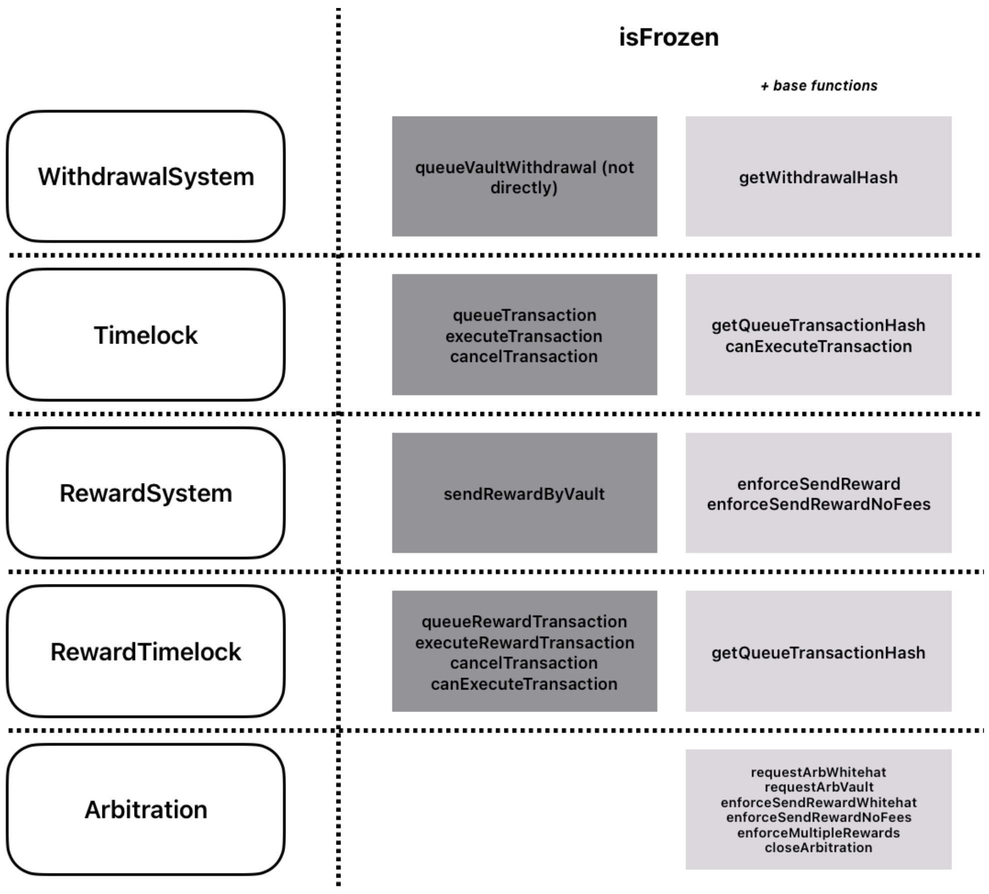


Figure 5. The frozen functions per component

EmergencySystem

A contract that can disable ImmunefiGuard (return behavior) and ImmunefiModule (revert behavior).

VaultSetup

A contract that is used in the Safe setup process to properly enable the Immunefi module and guard.

VaultDelegate

A contract that is meant to be called with delegate call by vaults to provide them additional functionality about transferring tokens out of the vault.

VaultFees

A contract that allows setting custom fees per vault. These fees are paid to Immunefi Fee recipient.

Actors

This part describes actors of the system, their roles, and permissions.

Admin

The most privileged role in the system offers a lot of functionalities depending on the specific component, governed by Immunefi. Such as granting new roles, setting modules, guards, fee amounts, and a lot of other protocol parameters.

Proxy Admin

The admin of all Transparent proxy contracts. This role is implemented by ProxyAdminOwnable2Step and is meant to be owned by Admin.

Vault

The 1:1 Safe multi-sig contract with enabled Immunefi module and guard. This is the main unprivileged actor in the system.

Vault Owner

The only owner of Vault. This address is governed by the listed project on Immunefi.

Whitehat

The externally owned account (or possibly a contract) that is figuring in the protocol as an unprivileged actor that should (not) receive rewards.

Arbiter

An entity that is responsible for resolving arbitrations (sending rewards, closing) chosen by Immunefi.

Enforcer

In the current scope, it is an exclusive role for the Arbitration contract. The role can send rewards if 5.1.2.3 is in arbitration.

Queuer

The role can add transaction requests with arbitrary parameters to the queue. In the current scope, it is an exclusive role for the WithdrawalSystem contract.

Freezer

The role can freeze vaults with the VaultFreezer contract.

Executor

The role that can execute ImmunefiModule (and thus withdraw any funds from any vault with `execTransactionFromModule`). This role is granted to the following components:

- Timelock,
- RewardSystem,
- RewardTimelock,
- Arbitration.

Fee setter

The role that can set the Immunefi fee on VaultFees. Assigned by Admin.

Fee recipient

The address that receives Immunefi or arbitration fees.

Allowed recipient

Is a special role to whom can be sent funds from the arbitration process. It should be some expert who is paid for his work to resolve arbitration disputes, also assigned by Admin.

5.2. Trust Model

The project's purpose is clear but the implementation's flexibility introduces complexity needing a lot of access-controlled contracts and special actors. These elevated privileges are managed by Admin (Immunefi) and several issues can happen if any of these roles are impersonated or used maliciously. We will enumerate the most important ones:

- Admin deploys Safe for the project (Vault that is going to contain the project's funds) and enables Immunefi guard and Immunefi module on that. Both of these contracts are upgradable and controlled by Admin. That has the following implications:
  - Admin restricts what Vault can do (by adjusting the scope on ScopeGuard).
  - Admin can call anything on behalf of the Vault (by upgrading the module).
- Admin is using VaultFees to implement dynamic fees on per vault basis. This allows contract owner to effectively front-run any reward transaction setting 100% fee on it and take the whole amount.
- Admin is choosing the Arbiter for the project. There is no on-chain consensus between Vault and Whiteheat.

- Admin can freeze a specific Vault anytime without any reason or dependent logic, making it unable to withdraw.
- Admin can choose custom oracles to return arbitrary prices.

Vault has to trust Immunefi that they do not misuse their power over their funds since they have full control over them.

Whitehat has to trust Immunefi that they will be watching Vaults from unexpected withdrawals (see W1: Introduced smart contract logic can be still bypassed) or as mentioned above, if they choose an unbiased arbiter for arbitrations. Moreover, their reward is highly dependent on Immunefi since they are responsible for the correct functioning of Vaults.

M1: It is possible to exit with funds during ongoing arbitration

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	`**/*`	Type:	Logic error

Description

There are no restrictions on withdrawal except freezing the vault. Vault freezing must be done manually by Immunefi to prevent the possibility of exiting with funds during ongoing arbitration.

Exploit scenario

There is an ongoing arbitration and Whitehat A is disputing with Project B, whether he is eligible for a reward. Project B queues full withdrawal and off-chain monitoring doesn't detect it. As a result, Project B exits with all funds and Whitehat A is losing the guarantee of receiving the reward.

Recommendation

Restrict withdrawals during an arbitration process. If there is a restriction on paying rewards, there should be also more stringent restrictions than just a cooldown period that relies on off-chain mechanisms.

Client's response

Acknowledged by the client.

> The withdrawals are allowed during arbitration by design. It is expected that the off-chain agents will freeze the vault if the withdrawal proposal is compromising the arbitration. Again, this

> doesn’t seem like an actual smart contract vulnerability here.
>
> — Immunefi

Go back to Findings Summary

L1: Insufficient data validation when composing contracts

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	`**/*`	Type:	Data validation

Description

The protocol consists of multiple components which makes it suitable to utilize contract ids. The contracts are composed via setter function in `setUp` initialize function so the changes to these setter don't affect only the initial setup but also later behavior (reason for high impact instead of medium). For example the RewardSystem contract is assigning in the setup function the following addresses (components):

- address _module,
- address _vaultDelegate,
- address _arbitration,
- address _vaultFreezer.

However, no validation besides zero-address checking is done.

For this purpose, contract ids can be utilized:

1. Define an id for each contract, eg: `bytes32 public constant CONTRACT_ID = keccak256("immunefi.arbitration")`.
2. When composing contracts, check that the contract id matches:

```
require(
    IBase(_arbitration).CONTRACT_ID() == keccak256("immunefi.arbitration"),
    "Not Immunefi Arbitration contract"
);
```

Exploit scenario

A wrong contract address is passed to the `setUp` function of the `RewardSystem` contract. The contract then uses the wrong contract, which will lead to unintended behavior.

Recommendation

Utilize contract ids since they are a cheap and simple way to eliminate more invalid user inputs than zero-address checks.

Client's response

Acknowledged by the client.

> This is something to be careful about at setup time.
>
> — Immunefi

Go back to Findings Summary

L2: Chainlink feed registry will break on other chains than mainnet

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	PriceConsumer.sol	Type:	Logic error

Description

The current codebase can not be reused for deployment on other chains than mainnet, because according to the documentation, the Chainlink feed registry is deployed only on mainnet. Likelihood is considered medium, since it affects only a smaller part of the protocol.

Exploit scenario

The codebase is deployed on Optimism and some Vault is in arbitration and wants to distribute rewards. However, because feed registry doesn't exist, any attempt to send rewards reverts.

Recommendation

Make sure the codebase for Optimism (and other possible chains) will be adjusted properly or adjust the logic to use aggregators directly.

Fix 1.1

There is a new contract FeedRegistryL2, that should be used for the other chains.

Go back to Findings Summary

L3: Chainlink validation is not sufficient for L2 deployments

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	PriceConsumer.sol	Type:	Data validation

Description

On some L2s (like Optimism), we have an entity called a sequencer. The sequencer is a node that receives the user's transactions and posts them in a batch to the L1. Currently, almost no protocol provides decentralized sequencing, and thus their downtime is relatively possible. Additionally, Chainlink updates the L2 Data Feeds through the sequencer, so if it is down, the prices aren't updated.

The current codebase doesn't check if the sequencer is not down. Chainlink provides a feed to check the sequencer downtime, which should be updated through the L1; see the docs.

Exploit scenario

The sequencer is offline, and there is a price drop/increase for a given token. As a result, some Vaults distributes incorrect rewards.

Recommendation

Implement sequencer uptime checks according to to the codebase for L2 deployment.

Fix 1.1

There is a new contract FeedRegistryL2, that adds support for sequencer uptime checks.

```
(, int256 answer, uint256 startedAt, , ) =
SEQUENCER_UPTIME_FEED.latestRoundData();

bool isSequencerUp = answer == 0;
require(isSequencerUp, "FeedRegistryL2: Sequencer is down");

// Make sure the grace period has passed after thesequencer is back up.
uint256 timeSinceUp = block.timestamp - startedAt;
require(timeSinceUp > GRACE_PERIOD_TIME, "FeedRegistryL2: Grace period not
over");
```

The code is following the best practices, however, the validation could be even more strict. According to the docs, the `startedAt` can return 0 on an invalid round. Such an invalid round will pass successfully in this case (if other values are returned as default ones).

Go back to Findings Summary

L4: Transaction cooldown can overflow

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	WithdrawalSystemBase.sol	Type:	Integer overflow

Description

The setter of transaction cooldown for WithdrawalSystem is susceptible to overflow on type casting from uint256 to uint32.

```
function _setTxCooldown(uint256 cooldown) internal {
    txCooldown = uint32(cooldown);
}
```

While the likelihood is very low that someone would like to set `txCooldown` to the equivalent of more than 100 years, it's still possible to do that and it should not.

Exploit scenario

Admin sets tx cooldown to `4294967296`, due to overflow it will result in `txCooldown` being set to 0.

Recommendation

Add allowed range for `txCooldown` to fit into `uint32`, perform safe casting with overflow check or pass uint32 as a parameter to prevent type casting.

Fix 1.1

Fixed by setting parameter's type to `uint32`.

Go back to Findings Summary

L5: The `canExecuteTransaction` doesn’t check if the vault is frozen

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	Timelock.sol	Type:	Data validation

Description

In the Timelock contract, the view `canExecuteTransaction` function doesn’t check if the vault is frozen. That means if the vault is frozen and time is between cooldown and expiration end, the return value is true, but it should be false.

Recommendation

Add validation if the vault is frozen in the `canExecuteTransaction` function, similarly like it is done in RewardTimelock.

```
if (vaultFreezer.isFrozen(txData.vault)) return false;
```

Fix 1.1

There is an added check for the frozen vault in the `canExecuteTransaction` function.

Go back to Findings Summary

W1: Introduced smart contract logic can be still bypassed

Impact:	Warning	Likelihood:	N/A
Target:	WithdrawalSystem.sol, Timelock.sol	Type:	Logic error

Description

Before Immunefi smart contracts were introduced, project's funds weren't under control from unexpected listing withdrawals or payments to whitehat were possible without Immunefi fee (outside the platform). The current smart-contract design is not changing this behavior anyhow, because it is still possible to do withdrawals or payouts with WithdrawalSystem and Timelock.

One of examples could be that whitehat is looking for a vulnerabilities of a given project (because it has locked funds in vault) and he has already issue to report, but prepares more rigorous writeup. If cooldown for withdrawal is set to 1 day, it is likely that he won't be able to notice it and he will lose the opportunity to report the issue and receive the reward.

Recommendation

The chosen delay for the withdrawal process is a key part of this design decision. There should be implemented a mechanism that notifies users if the vault is going to withdraw funds from the vault. This can be at least implemented on frontend side for registered whitehats. Registered whitehat can follo/watch some projects for these notifications and so he will be notified with email if some of the watched vaults/projects began the withdrawal process.

Go back to Findings Summary

W2: Timelock withdrawal delay can be bypassed

Impact:	Warning	Likelihood:	N/A
Target:	WithdrawalSystem.sol, Timelock.sol	Type:	Logic error

Description

Delayed withdrawal from Vault can be potentially bypassed by impersonating the whitehat with the Vault team for the cost of Immunefi fee (10% default). While this action causes losses to the project, it can be still in some cases feasible, to lie and pay for example $1000 instead of $10000 and play a better reputation game (like than if it will be just done without fee).

For example, legit whitehat A submits a valid critical bug report. Project sees that and uses their whitehat B to pay a bounty to him, claiming he was the first.

This action heavily depends on Immunefi front-end to claim which whitehat was first in reporting the issue since in code there is no evidence of submitted bounties and their status.

Recommendation

Be aware of this case since it doesn't only introduce immediate withdrawals for a fee but also possible edge cases like the above.

Client's response

Acknowledged by the client.

> This is known.
>
> — Immunefi

Go back to Findings Summary

W3: Hardcoded slippage protection on queued rewards

Impact:	Warning	Likelihood:	N/A
Target:	RewardTimelock.sol	Type:	Slippage control

Description

During arbitrations, rewards can be sent with a delay. In this case they shouldn't lose or gain much value before execution. In the protocol, the tolerance is set to 1%.

Since there can be used any types of tokens, there is a possibility of high price fluctuation and it might be desirable to configure the slippage before queuing the request with some reasonable maximal and minimal limit. For example, the maximal limit could be used at the mentioned 1% because in this case we don't want to differ from the price so much and high transaction throughput is not a priority like in some more dynamic environments. This could make the payouts more precise. On the other hand, hardcoded slippage in this case offers more predictable behavior overall.

Recommendation

If the range remains hardcoded, it should be well communicated to the whitehat and vault, so they will be aware of potential losses up to 1%. In the case of configurable slippage, it should be transparent to the whitehat, because the chosen slippage will be a vault decision.

Go back to Findings Summary

W4: Requesting arbitration by vault is not possible when using fee-on-transfer tokens

Impact:	Warning	Likelihood:	N/A
Target:	Arbitration.sol	Type:	Logic error

Description

The end of `requestArbVault` function contains the following require statement:

```
require(
    feeToken.balanceOf(feeRecipient) >= initialBalance + _feeAmount,
    "Arbitration: fee transfer failed"
);
```

That checks if the fee amount that is paid for arbitration is successfully transferred. However, in the case of fee-on-transfer tokens, this requirement won't be met because the amount will be less, depending on the token fee.

Recommendation

Ensure the arbitration contract won't use fee-on-transfer tokens for paying fees.

Go back to Findings Summary

W5: Old Safe version is used

Impact:	Warning	Likelihood:	N/A
Target:	Vaults	Type:	Outdated code

Description

Immunefi Vaults are Safe contracts version 1.3.0. The version was released in 2021 and since that, new releases have been introduced. We did not find any direct threat caused by the outdated Safe version, however, the newer version contains several new invariants, checks and updated logic. As a result of these changes, the newer version is considered more robust and safe.

Examples:

- Safe version 1.4.1 is compatible with ERC-4337 account abstraction.
- Version 1.4.0 checks a target address that supports EIP-165 interface while setting a new guard.
- Version 1.4.0 implements the invariant that `dataHash` is equal to the hash of `data` in the signature verification.

To see a complete list of changes, see GitHub Safe CHANGELOG.

Recommendation

We recommend upgrading the Safe version 1.3.0 used in the Immunefi Vaults to version 1.4.1.

For the upgrade, we recommend using the Safe Migration contract (Safe130To141Migration). The migration works with a standard Safe proxy which stores singleton address at the storage slot 0. The migration contract contains several helper functions to ensure the upgrade will be done correctly and commented instructions to follow.

The reason we do not recommend upgrading to version 1.5.0 in this specific case is that guards are attached to the module transaction in the newest version. It means the current logic of the protocol would need to be adjusted because the guards are now attached to the module transaction outside of the Safe core logic.

Go back to Findings Summary

W6: Sending reward can be front-ran and blocked by arbitration

Impact:	Warning	Likelihood:	N/A
Target:	Arbitration.sol	Type:	Griefing

Description

The usual flow for sending rewards is with the `sendRewardByVault` function. The function expects that the vault is not in arbitration. In the meantime, it is possible to open arbitration by anyone on for any vault. This puts the chosen vault into the arbitration process. This logic decision effectively introduces denial of service that can be triggered anytime.

Exploit scenario

Vault decides to call `sendRewardByVault` to pay whitehat. Alice sees the transaction and opens for the Vault arbitration by calling `requestArbWhitehat`. As a result, the reward can not be sent. The likelihood is set to low since there are expected to be very high fees for arbitration services.

Recommendation

Since the process of arbitration is not expected to be evaluated in a short time, introducing a delay (cooldown period) for the requesting of arbitration could be a sufficient solution how to mitigate front-running while not significantly affecting the usual flow of the system and UX for whitehats.

Client’s response

Acknowledged by the client.

> This is known.

> — Immunefi

Go back to Findings Summary

I1: Misleading reference type or naming

Impact:	Info	Likelihood:	N/A
Target:	RewardTimelock.sol	Type:	Code maturity

Description

The `executeRewardTransaction` function accepts `referenceId` as bytes32 parameter. However, in the protocol is figuring `arbitrationId` as bytes32 parameter and `referenceId` as uint96 parameter.

```
    function executeRewardTransaction(
        bytes32 txHash,
        bytes32 referenceId,
```

The value is only emitted in VaultDelegete so it is not affecting any logic directly, but the naming or type is confusing in the code since it is not clear what it should be.

Recommendation

Decide to use the `referenceId` as uint96 or `arbitrationId` as bytes32 depending on the needs.

Fix 1.1

The `referenceId` parameter is now `uint96` type consistently.

Go back to Findings Summary

I2: Withdrawal and reward requests stay opened after expiration

Impact:	Info	Likelihood:	N/A
Target:	RewardTimelock.sol, Timelock.sol	Type:	Best practice

Description

The timelock contracts disallow to cancel requests after expiration, as a result, there are transactions with an opened state, but that can't be executed in the current context. However, it is possible in future refactoring that this behavior will be omitted and thus there will be potentially executable stale requests.

Recommendation

Be aware of this design decision in future development or adjust the logic to check for expiration in function and if the request is expired, mark it as expired.

Go back to Findings Summary

I3: Use of custom errors

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	Gas optimization

Description

The use of custom errors leads to cheaper deployment and also runtime costs. The runtime costs are only relevant when the revert condition is met.

Recommendation

Use custom errors instead of the revert strings.

Client’s response

Acknowledged by the client.

> Might replace revert strings in V2.
>
> — Immunefi

Go back to Findings Summary

I4: Commented out code

Impact:	Info	Likelihood:	N/A
Target:	ArbitrationBase.sol	Type:	Code maturity

Description

The function `_setFeeAmount` contains commented out require. This is potential source of confusion or issues in the later development.

```
function _setFeeAmount(uint256 newFeeAmount) internal {
    // require(newFeeAmount > 0, "ArbitrationBase: feeAmount cannot be 0");
    feeAmount = newFeeAmount;
}
```

Recommendation

Remove all commented out code in the codebase.

Go back to Findings Summary

I5: The `getWithdrawalHash` function must be called prior to the withdrawal request to be functional

Impact:	Info	Likelihood:	N/A
Target:	WithdrawalSystem.sol	Type:	Documentation

Description

The `getWithdrawalHash` function according to Natspec documentation returns the hash of a withdrawal operation. However, it returns only the hash of the withdrawal operation that is going to be requested in the future. This should be mentioned in the Natspec documentation to avoid confusion because the nonce can't be specified.

Recommendation

Adjust the inlined documentation so it is clear how the function should be used or potentially allow the function to specify nonce to get a specific withdrawal hash.

Fix 1.1

There is an added `nonce` parameter to the `getWithdrawalHash` function.

Go back to Findings Summary

I6: Incorrect requirement in Natspec for vault request for arbitration

Impact:	Info	Likelihood:	N/A
Target:	Arbitration.sol	Type:	Documentation

Description

The `requestArbWhitehat` function according to the documentation requires to be called by vault. However the contract doesn't hold any mapping of vaults and as a result the function can be called by any Safe contract.

```
/**
 * @notice Requests arbitration by Vault
 * @dev Caller NEEDS to be the Vault
 * @param referenceId Reference ID of the request.
 * @param whitehat The whitehat address
 */
function requestArbVault(uint96 referenceId, address whitehat) external {
```

Recommendation

Adjust the inlined documentation from:

```
 * @dev Caller NEEDS to be the Vault
```

to:

```
 * @dev Caller SHOULD be the Vault
```

Go back to Findings Summary

I7: Use pre-incrementation in for cycles

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	Gas optimization

Description

The project contains for cycles with post-incrementation. The pre-incrementation is more gas efficient while not introducing any problems affecting code readability.

Recommendation

Use pre-incrementation in for cycle headers instead of post-incrementation.

Go back to Findings Summary

I8: Unnecessary lookup in for cycles

Impact:	Info	Likelihood:	N/A
Target:	RewardTimelockBase.sol, TimelockBase.sol, VaultDelegate.sol, PriceConsumer.sol	Type:	Gas optimization

Description

The project accesses `.length` for cycles in several places. This is unnecessary and can be optimized by saving the length in a local variable.

Recommendation

Store returned values in local variables to prevent multiple `.length` accesses.

Go back to Findings Summary

6. Report revision 1.1

The client provided a new contract FeedRegistryL2 and fixes to following issues:

- L2: Chainlink feed registry will break on other chains than mainnet,
- L3: Chainlink validation is not sufficient for L2 deployments,
- L4: Transaction cooldown can overflow,
- L5: The `canExecuteTransaction` doesn't check if the vault is frozen,
- I1: Misleading reference type or naming,
- I4: Commented out code,
- I5: The `getWithdrawalHash` function must be called prior to the withdrawal request to be functional,
- I6: Incorrect requirement in Natspec for vault request for arbitration,
- I8: Unnecessary lookup in for cycles,

the rest of the issues were acknowledged.

6.1. System Overview

This section contains an outline of the newly added contracts in this revision.

FeedRegistryL2

The contract is responsible for managing various feeds (specified by the contract owner). When the sequencer uptime feed address is set during the contract construction, it then allows to check (during price requests) if the sequencer is up. This contract was implemented as a response to L3: Chainlink validation is not sufficient for L2 deployments.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Immunefi: Vault, 28.2.2024.

Appendix B: Glossary of terms

The following terms might be used throughout the document:

Superclass/Ancestor of C

A contract that C inherits/derives from.

Subclass/Child of C

A contract that inherits/derives from C.

Syntactic contract

A Solidity contract. May have an inheritance chain, and may be deployed.

Deployed contract

An EVM account with non-zero code. If its source was written in Solidity, it was created through at least one syntactic contract. If that contract had superclasses (parents), it would be composed of multiple syntactic contracts.

Init/initialization function

A non-constructor function that serves as an initializer. Often used in upgradeable contracts.

External entrypoint

A `public` or `external` function.

Public/Publicly-accessible function/entrypoint

An `external` or `public` function that can be successfully executed by any network account.

Mutating function

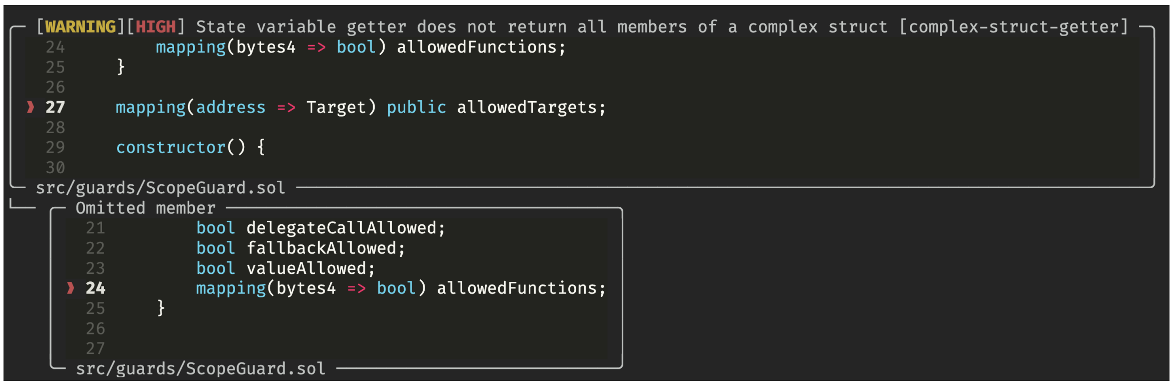
A non-`view` and non-`pure` function.

Appendix C: Wake outputs

This section contains the selected outputs of the Wake tool used during the audit.

C.1. Detectors

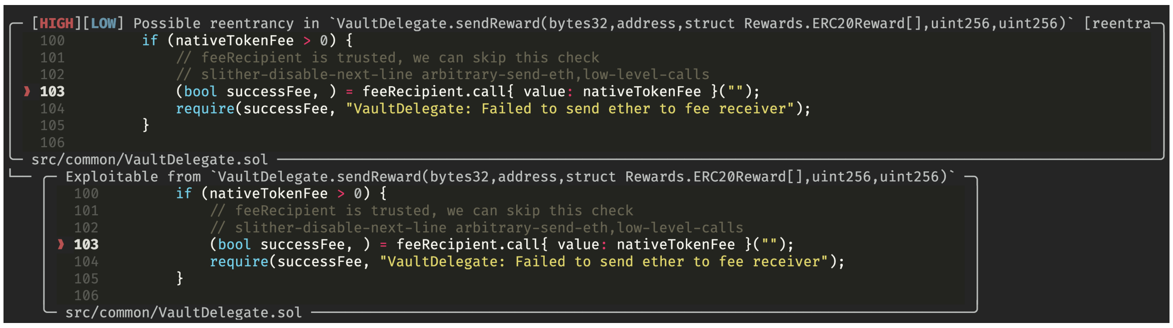
The static analysis discovered for example that `allowedTargets` getter is not returning all elements properly, however, it wasn't found anyhow problematic in the current context.



```
┌ [WARNING][HIGH] State variable getter does not return all members of a complex struct [complex-struct-getter] ┐
│  24          mapping(bytes4 => bool) allowedFunctions;
│  25      }
│  26
│❱ 27      mapping(address => Target) public allowedTargets;
│  28
│  29      constructor() {
│  30
└ src/guards/ScopeGuard.sol ──────────────────────────────────────────────────────────────────────────────────┘
└── ┌ Omitted member ───────────────────────────────────┐
    │  21          bool delegateCallAllowed;
    │  22          bool fallbackAllowed;
    │  23          bool valueAllowed;
    │❱ 24          mapping(bytes4 => bool) allowedFunctions;
    │  25      }
    │  26
    │  27
    └ src/guards/ScopeGuard.sol ────────────────────────┘
```

Figure 6. ScopeGuard detection results

Otherwise, there are a lot of possibilities of reentrancies or unprotected calls, however, all the occurrences are safe because the destination contract is trusted.



```
┌ [HIGH][LOW] Possible reentrancy in `VaultDelegate.sendReward(bytes32,address,struct Rewards.ERC20Reward[],uint256,uint256)` [reentra┐
│  100         if (nativeTokenFee > 0) {
│  101             // feeRecipient is trusted, we can skip this check
│  102             // slither-disable-next-line arbitrary-send-eth,low-level-calls
│❱ 103             (bool successFee, ) = feeRecipient.call{ value: nativeTokenFee }("");
│  104             require(successFee, "VaultDelegate: Failed to send ether to fee receiver");
│  105         }
│  106
└ src/common/VaultDelegate.sol ─────────────────────────────────────────────────────────────────────────────────────────────────────┘
└── ┌ Exploitable from `VaultDelegate.sendReward(bytes32,address,struct Rewards.ERC20Reward[],uint256,uint256)` ┐
    │  100         if (nativeTokenFee > 0) {
    │  101             // feeRecipient is trusted, we can skip this check
    │  102             // slither-disable-next-line arbitrary-send-eth,low-level-calls
    │❱ 103             (bool successFee, ) = feeRecipient.call{ value: nativeTokenFee }("");
    │  104             require(successFee, "VaultDelegate: Failed to send ether to fee receiver");
    │  105         }
    │  106
    └ src/common/VaultDelegate.sol ──────────────────────────────────────────────────────────────────────────┘
```

Figure 7. VaultDelegate detection results

C.2. Tests

The following code snippets show crucial parts of the fuzz test written for this project.[1]

We have modeled the following class representing the protocol.

```
class ImmunefiFuzzTest(FuzzTest):
    arb_token: ERC20
    owner: Account
    fee_recipient: Account
    arbiter: Account
    vaults: Dict[GnosisSafe, Tuple[Account, ERC20]]
    arbitration: Arbitration
    reward_timelock: RewardTimelock
    reward_system: RewardSystem
    timelock: Timelock
    withdrawal_system: WithdrawalSystem
    vault_fees: VaultFees
    vault_delegate: VaultDelegate
    emergency_system: EmergencySystem
    scope_guard: ScopeGuard
    vault_freezer: VaultFreezer
    immunefi_guard: ImmunefiGuard
    immunefi_module: ImmunefiModule
    queue_withdrawal: List[bytes32]
    queue_arbitration: List[bytes32]
    queue_reward: Dict[bytes32, uint256]
```

Then in a pre-sequence part, we deploy the protocol and prepare various existing tokens for testing.

```
def pre_sequence(self) -> None:
    self.arb_token = MyERC20.deploy("Arbitration Token", "ARBT")
    feed_registry = Address("0x47Fb2585D2C56Fe188D0E6ec628a38b74fCeeeDf")
    (
        ...
    ) = deploy_protocol(self.arb_token, feed_registry)
    # we will test some specific tokens for a potentially weird behavior
    testing_tokens = [
        ERC20("0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48"), # usdc
        ERC20("0xdac17f958d2ee523a2206206994597c13d831ec7"), # usdt
        ERC20("0xB8c77482e45F1F44dE1745F52C74426C631bDD52"), # bnb
        ERC20("0x6b175474e89094c44da98b954eedeac495271d0f"), # dai
        ERC20("0xa7DE087329BFcda5639247F96140f9DAbe3DeED1"), # sta
        ERC20("0x80fB784B7eD66730e8b1DBd9820aFD29931aab03"), # lend
        ERC20("0xaba8cac6866b83ae4eec97dd07ed254282f6ad8a"), # yamv2
        ERC20("0x1f9840a85d5af5bf1d1762f925bdaddc4201f984"), # uni
        ...
    ]
    # we will have up to 20 vaults randomly deployed
    self.vaults = {}
    for i in range(random_int(1, 20)):
        a = random_account()
        token_id = random_int(0, len(testing_tokens) - 1)
        self.vaults[deploy_vault(a, self.immunefi_guard,
self.immunefi_module)] = (a, testing_tokens[token_id])
        mint_erc20(
            testing_tokens[token_id],
            list(self.vaults.keys())[i],
            100_000_000 * 10**testing_tokens[token_id].decimals()
        )
    logger.info(f"New sequence initiated - current number of vaults: {len
(self.vaults)}")
    self.queue_withdrawal = []
    self.queue_arbitration = []
    self.arbitration.grantRole(self.arbitration.ARBITER_ROLE(),
self.arbiter)
    self.queue_reward = {}
    self.vault_freezer.grantRole(self.vault_freezer.FREEZER_ROLE(),
self.owner)
```

Then several flows are fuzzed. For example, the following flow shows the usage of the `sendRewardByVault` function.

```
@flow(weight=80, precondition=lambda self: not
self.emergency_system.emergencyShutdownActive())
    def send_reward_directly_to_whitehat(self) -> None:
        vault, (owner, token) = random.choice(list(self.vaults.items()))

        if self.arbitration.vaultIsInArbitration(vault):
            logger.info(f"Vault is in arbitration, cannot send reward")
            return

        amount = random_int(1, 100) * 10**token.decimals()
        native_amount = random_int(0, 3) * 10**18 if native_tokens else 0
        whitehat = random_address()
        rid = random_bytes(32)
        logger.info(f"Sending reward {amount} {token.symbol()} to
whitehat")
        vault_bal_before = token.balanceOf(vault.address)
        whitehat_bal_before = token.balanceOf(whitehat)
        fee_recipient_bal_before = token.balanceOf(
self.fee_recipient.address)

        if self.vault_freezer.isFrozen(vault):
            with must_revert("GS013"):
                simple_safe_tx(
                    vault,
                    owner,
                    Abi.encode_call(
                        self.reward_system.sendRewardByVault,
                        [
                            rid,
                            whitehat,
                            [(token, amount)],
                            native_amount,
                            0
                        ]
                    ),
                    self.reward_system
                )
            assert token.balanceOf(vault.address) == vault_bal_before
            assert token.balanceOf(whitehat) == whitehat_bal_before
            assert token.balanceOf(self.fee_recipient.address) ==
fee_recipient_bal_before
            return


        tx = simple_safe_tx(
            vault,
            owner,
            Abi.encode_call(
                self.reward_system.sendRewardByVault,
                [
                    rid,
                    whitehat,
                    [(token, amount)],
                    native_amount,
                    0
                ]
            ),
            self.reward_system
        )
        fee = self.vault_fees.getFee(vault)[0]
        assert VaultDelegate.RewardSent(
            vault.address, rid, whitehat,
            [Rewards.ERC20Reward(token.address, amount)],
            native_amount, self.fee_recipient.address, fee
            ) in tx.events

        immunefi_fee = (amount * fee) // self.vault_fees.FEE_BASIS()
        assert token.balanceOf(vault.address) == vault_bal_before - amount
- immunefi_fee

        if token.symbol() == "STA":
            amount -= ((amount * 100) // 10_000) # 1% fee
            immunefi_fee -= ((immunefi_fee * 100) // 10_000) # 1% fee

        if whitehat == self.fee_recipient.address:
            amount += immunefi_fee
        else:
            assert token.balanceOf(self.fee_recipient.address) ==
fee_recipient_bal_before + immunefi_fee

        assert token.balanceOf(whitehat) == whitehat_bal_before + amount
```

[1] The whole test code could be found on our GitHub as a repository prefixed with `tests-` if the project allows publishing it.

Appendix D: Files included in the audit scope

- Arbitration.sol
- EmergencySystem.sol
- ImmunefiModule.sol
- RewardSystem.sol
- RewardTimelock.sol
- Timelock.sol
- VaultFreezer.sol
- WithdrawalSystem.sol
- base/AccessControlBaseModule.sol
- base/AccessControlGuardable.sol
- base/ArbitrationBase.sol
- base/RewardSystemBase.sol
- base/RewardTimelockBase.sol
- base/TimelockBase.sol
- base/WithdrawalSystemBase.sol
- common/Rewards.sol
- common/VaultDelegate.sol
- common/VaultFees.sol
- encoders/ArbitrationOperationEncoder.sol
- encoders/BaseEncoder.sol
- encoders/RewardTimelockOperationEncoder.sol

- encoders/TimelockOperationEncoder.sol
- events/IArbitrationEvents.sol
- events/IEmergencySystemEvents.sol
- events/IImmunefiGuardEvents.sol
- events/IRewardSystemEvents.sol
- events/IRewardTimelockEvents.sol
- events/IScopeGuardEvents.sol
- events/ITimelockEvents.sol
- events/IVaultFreezerEvents.sol
- events/IWithdrawalSystemEvents.sol
- guards/ImmunefiGuard.sol
- guards/ScopeGuard.sol
- handlers/VaultSetup.sol
- oracles/IPriceConsumerEvents.sol
- oracles/IPriceFeed.sol
- oracles/PriceConsumer.sol
- oracles/chainlink/Denominations.sol
- oracles/chainlink/FeedRegistryInterface.sol
- proxy/ProxyAdminOwnable2Step.sol

ackee | blockchain security

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://twitter.com/AckeeBlockchain