Internal Audit Report

Jan 15, 2024

This report represents the results of an internal audit report that our team of triagers have conducted on the Vault System, specifically the Arbitration contract, that Immunefi is launching for their customers.

Technical Overview for the Audit

Code repository: https://github.com/immunefi-team/vaults-internal.
Commit for audit:tag v1-internal-audit.
nSLOC: ~1800 (excludes comments and empty lines).
In scope: all files inside the *src* folder, excluding the *mocks* folder.

Upgradeability
Most of the protocol components are designed to work with upgradeability. Contracts are deployed with the proxy as OZ's TransparentUpgradeableProxy. The owner of the proxies is the ProxyAdminOwnable2Step component, a contract inheriting from OZ's ProxyAdmin with the added functionality of OZ's Ownable2Step. Contracts without upgradeability don't have *disableInitialisers* on their constructor.

Admin Permissions

Arbitration.sol

1. *DEFAULT_ADMIN_ROLE* can change settings present in *ArbitrationBase.sol* and grant roles.
2. *ARBITER_ROLE* can enforce rewards and close the arbitration.

RewardSystem.sol

1. *DEFAULT_ADMIN_ROLE* can change settings present in *RewardSystemBase.sol* and grant roles.
2. *ENFORCER_ROLE* can enforce the sending of a reward. The Arbitration contract needs to have this role.

Timelock.sol

1. *DEFAULT_ADMIN_ROLE* can change settings present in *TimelockBase.sol* and grant roles.
2. *QUEUER_ROLE* can queue operations to be timelocked. The WithdrawalSystem component needs to have this role.

WithdrawalSystem.sol

1. The *Owner* can change settings present in *WithdrawalSystemBase.sol*.

EmergencySystem.sol

1. The *Owner* can activate or deactivate the emergency shutdown. They can also transfer ownership (2 steps).

VaultFreezer.sol

1. *DEFAULT_ADMIN_ROLE* can grant roles.
2. *FREEZER_ROLE* can freeze/unfreeze Vault addresses.

ImmunefiModule.sol

1. *DEFAULT_ADMIN_ROLE* can set different guard and grant roles.
2. *EXECUTOR_ROLE* can execute a transaction on a Vault through the module. Components Arbitration, RewardSystem and Timelock need to have this role.

ImmunefiGuard.sol/ScopeGuard.sol

1. The *Owner* can set different scoping parameters.

ProxyAdminOwnable2Step.sol

1. The *Owner* can transfer ownership (2 steps) and can upgrade the implementation of all proxies in the protocol.

Guards Permissions

ImmunefiGuard.sol (used in the Vault)

- Allowed targets:
    - ArbitrationSystem (Scoped)
    - WithdrawalSystem (Scoped)
    - Timelock (Scoped)
    - RewardSystem (Scoped)
    - RewardTimelock (Scoped)

ScopeGuard.sol (used in the ImmunefiModule)

- Allowed targets:
    - VaultDelegate (Scoped)
        - *sendReward* (delegatecall allowed)
        - *sendRewardNoFees* (delegatecall allowed)
        - *sendTokens* (delegatecall allowed)
        - *withdrawFunds* (delegatecall allowed)

Audit Findings:

[Medium] Vault Funds Withdrawal Process Design Flaw Allowing Timelock Bypass

- In the current contract design, there exists a potential issue concerning the withdrawal of funds from the vault. The standard procedure for vault owners to withdraw funds involves utilizing the WithdrawalSystem::queueVaultWithdrawal function, where they must wait for a predefined cooldown period before successfully removing funds from the vault.

- However, a bypass mechanism is identified through the RewardSystem::sendRewardByVault function, enabling vault owners to directly

send the entire vault balance to a specified address without adhering to the withdrawal timelock queue limitations.

- Expected Scenario:

    a. Vault initially holds $500 in funds.
    b. Vault owner decides to withdraw all the funds from the vault and initiates the process by using `WithdrawalSystem::queueVaultWithdrawal`.
    c. The vault owner patiently waits for the cooldown period to elapse.
    d. Once the cooldown period has passed, the timelock executes the transaction hash (txhash), successfully retrieving the $500 funds from the vault.

- Bypass Scenario:

    a. Vault initially holds $500 in funds.
    b. In this scenario, the vault owner chooses to create a fake submission by impersonating a whitehat.
    c. The vault owner leverages the `RewardSystem::sendRewardBy` Vault function to make a bounty payment of $500 to the whitehat using the vault's funds, bypassing the standard withdrawal timelock cooldown procedure entirely.

- This design issue poses a potential risk as it allows vault owners to circumvent the intended withdrawal process, potentially impacting the security and control of the vault's funds. It is essential to evaluate and address this issue to ensure the integrity and functionality of the withdrawal system align with the contract's intended design and security requirements.

Acknowledged: we are aware of this behavior, which is part of the reason why rewards get timelocked once a vault enters into arbitration. While the vault is not in arbitration, using the withdrawal system should be the standard way to withdraw money, not just as a signal to the whitehat community, but also because using sendReward has potential fee charges on the Vault withdrawal.

[Medium] Exploitation of Arbitration Process by Griefing Without Fee Payment

- The Arbitration::requestArbVault function is intended to be invoked by Project to initiate an arbitration process by submitting a fee. However, there is a lack of validation within the function to ensure that only the vault can make this call. Presently, any entity can access this function.

- One plausible rationale for this design decision may be the assumption that only the vault would initiate the call, given that the caller must supply a higher arbitration fee.
- It has been discovered that the Arbitration::requestArbVault function is susceptible to unauthorized calls without requiring any fees to open the arbitration.

- To exploit this vulnerability, an attacker can follow these steps:

    1. Deploy a malicious contract.
    2. Invoke the `requestArbVault` function from the malicious contract.
    3. The ImmunefiModule will trigger the guard to execute certain validations, all of which will succeed.
    4. The ImmunefiModule will then call the msg.sender, which in this case will be the malicious contract.
    5. Implement an `execTransactionFromModule` function in the malicious contract that returns true.

Refer to the PoC for more information:

```
JavaScript
function testReqArbRevertWithoutFee() public {
    vm.startPrank(protocolOwner);
    moduleGuard.setTargetAllowed(address(vaultDelegate), true);
    moduleGuard.setAllowedFunction(address(vaultDelegate),
vaultDelegate.sendTokens.selector, true);
    moduleGuard.setDelegateCallAllowedOnTarget(address(vaultDelegate), true);
    vm.stopPrank();

    MaliciousVault maliciousVault = new MaliciousVault();

    maliciousVault.attack(arbitration);

    // Whitehat trying to open arbitration for same refId
    vm.prank(whitehat);
    bytes memory signature = _signData(
```

```
      whitehatPk,
      arbitration.encodeRequestArbFromWhitehatData("123", address(vault))
    );

    // This will revert
    // vm.startPrank(whitehat);
    // arbitration.requestArbWhitehat("123", address(vault), whitehat,
signature);
    // vm.stopPrank();
  }


contract MaliciousVault {
  function attack(Arbitration arb) public {
    arb.requestArbVault("123", address(this));
  }

  function execTransactionFromModule(
    address to,
    uint256 value,
    bytes memory data,
    Enum.Operation operation
  ) public virtual returns (bool success) {
    success = true;
  }

}
```

Impacts:

1. Any individual can disrupt the whitehat or vault from initiating an arbitration by opening an arbitration with the same `referenceId`. Consequently, when the legitimate whitehat or project attempts to commence an arbitration with the identical `referenceId`, the process will fail and revert due to the following check.

```
JavaScript
require(refIdArb[referenceId].status == ArbitrationStatus.None,
"Arbitration: referenceId already exists");
```

2. If the vault currently has any ongoing arbitration, any attempt to send reward will be unsuccessful. Exploiting this situation, anyone can take advantage to grief the

reward payment by generating fraudulent arbitration through this function without having to pay any fees.

Recommendations:

- The optimal approach is to establish a mapping of vaults (address => bool) and permit only those vaults to invoke this function.
- An alternative, albeit less preferred, method is to verify at the end of the transaction that arbitration fees have been received. This step aims to ensure that potential griefers experience a financial loss, discouraging malicious activities.

Fixed: commits 69e99eda and 66fc348c.

[Medium] Potential Logical Issue in Vault Withdrawal Process Could Lead to Reward Transaction Revert

- In the current system, vault owners can initiate fund withdrawals from the vault using the WithdrawalSystem::queueVaultWithdrawal function. This process involves queuing the withdrawal, waiting for a cooldown period, and then executing the withdrawal by calling timelock.executeTransaction with the corresponding transaction hash (txHash) to retrieve the funds from the vault.

- However, there is a potential logical issue within this system.

    a. Consider a scenario where the protocol has already queued a reward transaction to send a bounty to a whitehat via reward `Timelock.queueRewardTransaction`.

    b. If the vault withdraws its funds before the queued reward transaction is processed, and the vault withdrawal occurs after the `rewardTimelock.txCooldown()` period has passed, a problem arises. The subsequent execution of rewardTimelock.executeRewardTransaction will fail, causing a revert, as the contract will not have sufficient funds to reward the whitehat. Meanwhile, the vault owners will have already withdrawn their funds during this process.

    c. To address this issue, it is crucial to implement a system that checks on the withdrawn funds from the vaults, combined with the remaining funds in

the vault, **that are sufficient to cover any pending queued reward transactions for the whitehat.**

- In the given scenario:

    a. The vault starts with a balance of $500.
    b. The vault queues a $200 reward for whitehat.
    c. Subsequently, the vault decides to withdraw all its funds using the withdrawal queue.
    d. The vault successfully withdraws the entire $500 from the vault.
    e. However, when the vault tries to process the $200 reward for whitehat, it encounters a failure due to insufficient funds, as there is no money left in the vault.

Recommendations:

- To address this issue, a mechanism should be implemented to allow vault owners to withdraw only the remaining funds, which in this case would be $300, leaving the $200 allocated for the reward untouched in the vault during the whole process of the timelock queuing.

Refer to the PoC for more information:

```
JavaScript
function testWithdrawAndRewardTimelock() public {
    uint256 value = 1.1 ether;
    vm.deal(address(vault), value);

    // set right permissions on moduleGuard
    vm.startPrank(protocolOwner);
    moduleGuard.setTargetAllowed(address(vaultDelegate), true);
    moduleGuard.setAllowedFunction(address(vaultDelegate),
vaultDelegate.sendReward.selector, true);
    moduleGuard.setAllowedFunction(address(vaultDelegate),
vaultDelegate.withdrawFunds.selector, true);
    moduleGuard.setDelegateCallAllowedOnTarget(address(vaultDelegate), true);
    vm.stopPrank();

    uint256 nonce = rewardTimelock.vaultTxNonce(address(vault));
    bytes32 txHash = rewardTimelock.getQueueTransactionHash(address(this), 2000,
address(vault), nonce);

    // 1. Queuing the reward transaction
    _sendTxToVault(
      address(rewardTimelock),
```

```
      0,
      abi.encodeCall(rewardTimelock.queueRewardTransaction, (address(this),
2000)),
      Enum.Operation.Call
    );

    Rewards.ERC20Reward[] memory erc20Rewards = new Rewards.ERC20Reward[](0);
    bytes32 txHashWithdrawal = withdrawalSystem.getWithdrawalHash(
      withdrawalReceiver,
      erc20Rewards,
      1 ether,
      address(vault)
    );

    // 2. Queuine the withdraw transaction
    _sendTxToVault(
      address(withdrawalSystem),
      0,
      abi.encodeCall(withdrawalSystem.queueVaultWithdrawal, (withdrawalReceiver,
erc20Rewards, 1 ether)),
      Enum.Operation.Call
    );

    vm.warp(block.timestamp + withdrawalSystem.txCooldown());

    // 3. Executing the withdraw transaction
    _sendTxToVault(
      address(timelock),
      0,
      abi.encodeCall(timelock.executeTransaction, (txHashWithdrawal)),
      Enum.Operation.Call
    );

    vm.mockCall(
      address(priceConsumer),
      abi.encodeCall(priceConsumer.tryGetSaneUsdPrice18Decimals,
(Denominations.ETH)),
      abi.encode(uint256(2000) * 10 ** 18)
    );

    // 4. This transaction would revert
    _sendTxToVault(
      address(rewardTimelock),
      0,
      abi.encodeCall(
        rewardTimelock.executeRewardTransaction,
        (txHash, bytes32(0), erc20Rewards, 1 ether, 50_000)
      ),
      Enum.Operation.Call
```

```
    );
  }
```

Fixed: Though this is technically expected behavior, we've decided to restrict the RewardTimelock usage to arbitration vaults. Commit 2f75e54b.

[Medium] Potential Logical Issue in RewardSystem Process Could Lead to Reward Transaction Revert

- Currently, vaults have the capability to reward whitehats through the rewardTimelock.queueRewardTransaction function. In this process, vaults queue the transaction, undergo a cooldown period, and then utilize rewardTimelock.executeRewardTransaction to transfer funds from the vault to the designated whitehat recipient.

- However, a fundamental issue exists within this system, necessitating the implementation of a more robust mechanism. Specifically, there should be a system in place that performs a balance check at the time of queuing to ensure that the remaining balance in the vault is sufficient to cover the proposed timelock queue transfer.

- Scenario:

    1. The vault initially holds $100 in funds.
    2. The vault queues a reward of $100 for whitehat A.
    3. The vault queues another reward of $100 for whitehat B.
    4. The vault successfully processes the bounty for whitehat A.
    5. However, the vault encounters a failure while attempting to process the bounty for whitehat B.

- This scenario underscores the need for a balance verification mechanism to prevent situations where the vault's balance may not be adequate to fulfill queued rewards, thus enhancing the overall reliability and functionality of the system.

Refer to the PoC for more information:

JavaScript

```
function testDirectRewardAndTimelockReward() public {

    uint256 value = 1.1 ether;
    vm.deal(address(vault), value);

    console.log("Before balance of address this: ", address(this).balance);
    console.log("Before balance of whitehat: ", address(whitehat).balance);
    console.log("Before balance of vault: ", address(vault).balance);

    // set right permissions on moduleGuard
    vm.startPrank(protocolOwner);
    moduleGuard.setTargetAllowed(address(vaultDelegate), true);
    moduleGuard.setAllowedFunction(address(vaultDelegate),
vaultDelegate.sendReward.selector, true);
    moduleGuard.setDelegateCallAllowedOnTarget(address(vaultDelegate), true);
    vm.stopPrank();

    uint256 nonce = rewardTimelock.vaultTxNonce(address(vault));
    bytes32 txHash = rewardTimelock.getQueueTransactionHash(address(this), 2000,
address(vault), nonce);

    // 1. Queue the reward transaction
    _sendTxToVault(
      address(rewardTimelock),
      0,
      abi.encodeCall(rewardTimelock.queueRewardTransaction, (address(this),
2000)),
      Enum.Operation.Call
    );

    Rewards.ERC20Reward[] memory erc20Rewards = new Rewards.ERC20Reward[](0);

    // 2. send reward directly
    _sendTxToVault(
      address(rewardSystem),
      0,
      abi.encodeCall(
        rewardSystem.sendRewardByVault,
        ("0x00", whitehat, erc20Rewards, 1 ether,
vaultDelegate.UNTRUSTED_TARGET_GAS_CAP())
      ),
      Enum.Operation.Call
    );
    console.log("After balance of address this: ", address(this).balance);
    console.log("After balance of whitehat: ", address(whitehat).balance);
    console.log("After balance of vault: ", address(vault).balance);
```

```
    vm.warp(block.timestamp + rewardTimelock.txCooldown());
    assertTrue(rewardTimelock.canExecuteTransaction(txHash));

    // Mock priceConsumer
    vm.mockCall(
      address(priceConsumer),
      abi.encodeCall(priceConsumer.tryGetSaneUsdPrice18Decimals,
(Denominations.ETH)),
      abi.encode(uint256(2000) * 10 ** 18)
    );

    // 3. Execution of Queued transaction would revert
    _sendTxToVault(
      address(rewardTimelock),
      0,
      abi.encodeCall(
        rewardTimelock.executeRewardTransaction,
        (txHash, bytes32(0), erc20Rewards, 1 ether, 50_000)
      ),
      Enum.Operation.Call
    );

  }
```

Acknowledged: I believe there are arguments against performing such a check in the queueing moment. Projects might want to top up their vault only when they are actually going to execute a reward sending. But more than that, the fund availability check cannot be done against a dollar amount, considering the reward timelock has no knowledge about what tokens exist in the Vault. Even if such a knowledge were to be present, slippage could make so that those funds are no longer enough to pay the dollar amount in the moment of execution.

[Medium] Vault in Arbitration would halt other sendRewards transactions for other submissions.

- There is a significant design concern within the contract, particularly related to the arbitration mechanism integrated into the RewardSystem::sendRewardByVault function.

- This function incorporates a validation step that checks whether the vault is currently engaged in any ongoing arbitration process.

- If the vault is found to be in arbitration, the function prohibits any reward payments from being made by the vault with the following code snippet:

```
JavaScript
require(!arbitration.vaultIsInArbitration(msg.sender),
"RewardSystem: vault is in arbitration");
```

- The issue with this design choice lies in the fact that it effectively halts all reward payments originating from the vault, including payments for unrelated submissions, whenever there is an ongoing arbitration process associated with a single submission. This has the potential to result in significant delays, particularly if the arbitration process for one submission becomes protracted.

- Additionally, this design inadvertently opens up the possibility of abuse or malicious actions. For instance, a whitehat or another vault could potentially exploit this situation by initiating arbitration and deliberately prolonging it, knowing that this would disrupt the vault's capability to process subsequent reward payments for other submissions. This could pose significant issues, particularly when the disruption of payments is driven by motives unrelated to legitimate disputes.

Acknowledged: though it is true that the whitehat can grief a reward payment by calling arbitration, it is a very costly grief, which just sends that profit to the fee recipient. The fee recipient could even share or send the entire amount to the vault to compensate for their troubles. Considering the cost and no big impact to a project, we decided not to make the code more complex to avoid this attack vector (hence why it was previously added in the documentation as a known issue). As for vaults being able to grief others, fixing the aforementioned referenceId griefing issue will prevent this from happening.

[Low] Unauthorized Access to RewardTimelock::queueRewardTransaction Function Allows for Redundant Transactions Queues

- RewardTimelock::queueRewardTransaction, it was observed that while the function is intended for use by vaults to queue reward transactions for the whitehat, and includes a verification step to determine if the calling vault is frozen, the current implementation allows it to be accessed by any arbitrary user.

a. This unrestricted access could lead to the `txHashData` mapping being cluttered with unnecessary and redundant transactions. Such clutter could pose challenges, particularly in the context of pagination (e.g., `getVaultTxsPaginated`) at the frontend level.

b. To align with its intended use and enhance security, it is recommended that access to this function be restricted solely to vaults, thereby preventing unauthorized and potentially problematic calls.

Acknowledged: cluttering the txHashData *mapping makes no impact in the smart contract. Spam transaction hashes also provide no challenge to* getVaultTxsPaginated*, since you look at the transaction hashes of specific real vaults. Restricting access to vaults means creating a whitelist of vaults, which further complicates permissions and the protocol itself with no benefit.*

[Low] Fee-on-transfer Tokens May Lead to Reduced Rewards for the Whitehat.

- In the RewardTimelock, the project has the ability to queue a reward transaction, and after the cooldown period, the transaction can be executed by executeRewardTransaction. During the execution, the vault can specify token amounts and native amounts that correspond to the dollar amount.

```
JavaScript
    function executeRewardTransaction(
        bytes32 txHash,
        bytes32 referenceId,
        Rewards.ERC20Reward[] calldata tokenAmounts,
        uint256 nativeTokenAmount,
        uint256 gasToTarget
    ) external
```

- Nevertheless, there are no restrictions on the types of tokens that can be utilized for rewards. In the case of fee-on-transfer tokens, the validation `_checkRewardDollarValue`, which ensures equality between token amounts and dollar value, may pass, but the whitehat could still receive reduced rewards due to fee deductions.

Acknowledged: Supporting fee-on-transfer tokens doesn't seem necessary. The parameters of a reward are all given by the project owning the vault, so they can always pay a smaller reward than what the whitehat is expecting. They shouldn't be using fee-on-transfer tokens to pay the whitehat.

[Low] Missing Validation for Chainlink minPrice hit

- Chainlink aggregators have a built-in circuit breaker if the price of an asset goes outside of a predetermined price band. The result is that if an asset experiences a huge drop in value (i.e. LUNA crash) the price of the oracle will continue to return the minPrice instead of the actual price of the asset.
- This will allow projects to pay reward at wrong price in such situation
- Recommendation:

```
JavaScript
require(price >= minPrice && price <= maxPrice, "invalid price");
```

Acknowledged: There's no clear way of accessing these parameters for each feed, and Chainlink warns most feeds no longer use these. We will refrain from implementing this additional check, since staleness checks are already pretty resilient.

[Low] Insufficient Freezing Controls in VaultFreezer Contract Affecting RewardSystem and Arbitration Contracts

- Potentially there could be a critical oversight in the implementation of the vault freezing functionality, provided through the VaultFreezer contract. This contract enables accounts with the FREEZER_ROLE to freeze and unfreeze vaults. The primary intent of freezing vaults is to prevent them from executing transactions, particularly concerning reward timelocks and general timelocks.

- The current implementation of the vault freezing functionality does not comprehensively restrict all transactional capabilities of the frozen vaults.

    1. RewardSystem Contract Loophole: Despite a vault being frozen, it is still possible to execute transactions through the RewardSystem::sendRewardByVault function. This allows for the

movement of funds, contradicting the intended security measure of freezing.

2. Arbitration Contract: Similarly, the frozen vaults retain the ability to execute certain transactions within the Arbitration contract. This again undermines the purpose of the vault freezing mechanism.

- The primary purpose of freezing vaults is to safeguard against unauthorized movements of funds or tampering with timelock functionalities. The identified loopholes in the RewardSystem and Arbitration contracts create potential vulnerabilities where frozen vaults can still initiate fund transfers, thus compromising the security and integrity of the system.

Fixed: Commit [0f1635a4](0f1635a4).

[Informational] Events are not Emitted during Contracts Initialization

- In the setter function of all contracts, events are emitted first, followed by the invocation of internal setter functions. However, during the initialization of contracts, internal setter functions are directly called, and no events are emitted.

```javascript
function setUp(
  address _owner,
  address _module,
  address _rewardSystem,
  address _vaultDelegate,
  address _feeToken,
  uint256 _feeAmount,
  address _feeRecipient
) public initializer {
  __AccessControl_init();

  require(_owner != address(0), "Arbitration: owner cannot be 0x00");
  // @audit events are not emitted
  _grantRole(DEFAULT_ADMIN_ROLE, _owner);
  _setModule(_module);
  _setRewardSystem(_rewardSystem);
  _setVaultDelegate(_vaultDelegate);
  _setFeeToken(_feeToken);
  _setFeeAmount(_feeAmount);
```

```
    _setFeeRecipient(_feeRecipient);

    emit ArbitrationSetup(msg.sender, _owner);
  }


  function _setModule(address newModule) internal {
    require(newModule != address(0), "ArbitrationBase: module cannot be 0x00");
    immunefiModule = ImmunefiModule(newModule);
  }

  function _setRewardSystem(address newRewardSystem) internal {
    require(newRewardSystem != address(0), "ArbitrationBase: rewardSystem cannot
be 0x00");
    rewardSystem = RewardSystem(newRewardSystem);
  }
```

Arbitration.sol

- It is recommended to emit events within the internal setter functions instead of the public setter functions.

Acknowledged: purposefully didn't emit the typical events in setups to avoid further deployment and initialization costs, but a custom setup event is emitted.

[Informational] Missing Interface Validation For PriceConsumer::setCustomFeed

- In the PriceConsumer smart contract, there is a function setCustomFeed designed to allow the contract owner to specify a custom feed for oracle queries. This functionality is intended as an alternative to using the default Chainlink feed registry. The function is as follows:

```
Unset
function setCustomFeed(address base, address feed) external onlyOwner {
  customFeed[base] = feed;
  emit CustomFeedSet(base, feed);
}
```

- The primary risk identified in this implementation is the lack of validation to ensure that the provided feed address supports the **IPriceFeed** interface. This oversight can lead to a risk that an incompatible contract could be set as the feed.

Acknowledged: Owner needs to make sure the right price feed is being set here.

[Informational] Broad Access Control Concerns in Arbitration Contract's Enforcement Functionality

- We have a concern within the Arbitration contract, specifically pertaining to the enforceSendRewardNoFees(...) function. This function is part of the Arbitration module and is designed to distribute the funds from the vault to the specified recipients.

- The global access granted by the ARBITER_ROLE poses a significant security risk. It concentrates too much power in a single role, which could lead to misuse or mismanagement of funds, especially in cases of compromised or rogue arbiters.

- To address these concerns, it is recommended to split the roles into two distinct responsibilities:

    1. ARBITER_ROLE: Restrict this role solely to managing arbitration-related functions.
        - Responsibilities include paying out rewards to white-hat hackers from the vault and closing arbitrations identified by referenceID.

    2. ENFORCEMENT_RECIPIENT_ROLE:
        - Create this new role to manage the distribution of funds to globally allowed recipient addresses.
        - This role should be strictly limited to handling payments from the vault to these specified recipients.

Acknowledged: There's actually already a role for recipients, and thus the arbiter is not allowed to transfer to addresses without the ALLOWED_RECIPIENT_ROLE *role.*

[Informational] `ScopeGuard.setAllowedFunction` should revert if target is not scoped

- In ScopeGuard, the function setAllowedFunction is available for setting the allowed function that can be called on a scoped target.

```
JavaScript
function setAllowedFunction(address target, bytes4 functionSig, bool
allow) public onlyOwner {
        allowedTargets[target].allowedFunctions[functionSig] = allow;
        emit SetFunctionAllowedOnTarget(target, functionSig,
        allowedTargets[target].allowedFunctions[functionSig]);
}
```

- However, there is no check in place to validate whether the target for which the function sets the allowance is actually scoped and if the target is not scoped, the function should revert.

Acknowledged: No need to revert here, and this follows Zodiac's implementation.

[Informational] Repetitive check for the `emergencyShutdown`

- In ImmunefiGuard.checkTransaction, there is check for the emergencyShutdown

```
JavaScript
if (emergencySystem.emergencyShutdownActive()) {
        return;
  }
```

- The normal function flow is `ImmunefiModule.execute => ImmunefiModule.exec => ImmunefiGuard.checkTransaction`
- And in ImmunefiModule.execute, there is already a check for the same. So, it's a bit redundant

JavaScript

```
    function execute(
        address target,
        address to,
        uint256 value,
        bytes memory data,
        Enum.Operation operation
    ) external onlyRole(EXECUTOR_ROLE) {
        require(target != address(0), "ImmunefiModule: target is
zero address");

        require(!emergencySystem.emergencyShutdownActive(),
"ImmunefiModule: emergency shutdown is active");
```

Acknowledged: Actually, there are not 2 checks, because a transaction through the module doesn't execute the ImmunefiGuard (refer to Gnosis Safe wallet implementation).

[Miscs] `encodeQueueTransactionData` order of variables is different form `QUEUE_TX_TYPEHASH`

- In TimelockOperationEncoder, there's a function for encoding queue transactions, encodeQueueTransactionData.

JavaScript

```
// keccak256("QueueTx(address to,uint256 value,bytes data,uint8
operation,address vault,uint256 nonce)");
    bytes32 private constant QUEUE_TX_TYPEHASH =
0x8c5997cbee0e96cbb74515423607cbefdf7c33a876cfaaec589783883f00637
4;


function encodeQueueTransactionData(
        address to,
        uint256 value,
        bytes calldata data,
        Enum.Operation operation,
```

```
        address vault,
        uint256 nonce
    ) public view returns (bytes memory) {
        bytes32 queueTxHash = keccak256( // @audit-issue order is
different
            abi.encode(QUEUE_TX_TYPEHASH, to, value,
keccak256(data), vault, operation, nonce)
        );
        return _encodeTypedData(queueTxHash);
    }
```

- The order of variables in the hash differs from that specified in QUEUE_TX_TYPEHASH. While QUEUE_TX_TYPEHASH lists the operation first followed by the vault address, the actual encoding reverses this order, placing the vault address before the operation.

Fixed: Commit 8554b8f0.

[Miscs] Remove the duplicate call entries from Tests::Arbitration.t.sol

- The Arbitration tests under testArbSendsRewardWithoutClosingAndThenClose contains the duplicate call entries for the setting up the permissions on the moduleGuard.

Acknowledged: tests are not part of the scope.

[Miscs] Suggestion to implement a whitelist of tokens

- The current protocol permits the transfer of any tokens as a reward, but this design choice presents several issues:
    1. If the token passed as a parameter during reward payment is not supported by Chainlink, it will result in a revert while attempting to fetch its price.
    2. Fee-on-transfer tokens may lead to the receiving party receiving fewer rewards than intended.

3. Some other weird ERC20 tokens might cause unforeseen issues.

- Considering the aforementioned reasons, it is advisable to implement a mechanism for whitelisting tokens. Only tokens on this whitelist should be allowed for reward transactions.

Acknowledged: We might consider this in a future iteration of the protocol. As of now, normal tokens supported by Chainlink should just be used.

[Miscs] Multiple `CloseArb` conditions can be consolidated into a single condition

- In the arbitration.sol contract, there are multiple functions allowing the arbiter to enforce rewards. Within these functions, the arbiter can pass a boolean parameter **closeArb** to optionally close the arbitration. In these functions, the boolean is checked twice – first to set the arbitration status to close, and finally to remove the referenceId from the array.

JavaScript

```
        if (closeArb) {
            refIdArb[referenceId].status =
ArbitrationStatus.Closed;
            emit ArbitrationClosed(referenceId);
        }

        address vault = refIdArb[referenceId].vault;
        address whitehat = refIdArb[referenceId].whitehat;

        if (tokenAmounts.length > 0 || nativeTokenAmount > 0) {
            // enforce reward
            rewardSystem.enforceSendReward(referenceId, whitehat,
tokenAmounts, nativeTokenAmount, vault, gasToTarget);
        }

        if (closeArb) {
            // remove last for enforceSendReward to be executed
            vaultsOngoingArbitrations[vault].remove(referenceId);
        }
```

- One possible rationale for this approach might be to prevent potential reentrancy issues. However, upon our code review, we have not identified any ways in which reentrancy could occur. Therefore, it is suggested to combine these two conditions into a single one for improved code simplicity and readability.

Acknowledged: We will keep it to prevent reentrancy.

[Miscs] The RewardTimelock Fails to Fulfill its Intended Purpose.

- The purpose of implementing RewardTimelock is to prevent the vault from paying the whitehat upfront, particularly if there is an ongoing arbitration for that vault. The rationale behind this approach is to ensure that the vault retains funds in case the ongoing arbitration requires payment in the future. The project can use the RewardTimelock to facilitate payment to the whitehat. However, it has been observed that there are numerous hypothetical yet potentially practical scenarios where payment is still feasible even if the vault is currently undergoing arbitration.

    a) Consider the following scenario.

        1. Vault has 100 tokens, currently no open arbitration.
        2. Project queue reward transaction of 100 through rewardTimelock
        3. Before the execution of that transaction, an arbitration is opened for that vault
        4. Queued transactions get executed.

            - In this scenario, despite the vault being in arbitration, it lacks the necessary funds for arbitration payment.

    b) Another potential scenario is that a reward transaction is queued while arbitration is in progress. It is plausible that this transaction

could be executed after a cooldown period, even before the arbitration concludes.

- It is advisable to implement a role, granted to the protocol owner, that can cancel queued transactions by the vault. This role would be particularly useful in cases where suspicious transactions are timelocked, allowing the protocol owner to cancel those transactions as needed.
- Also, it's suggested that use of RewardTimelock be restricted to Vaults in arbitration only.

Acknowledged: The purpose of a timelock on the reward is to allow off-chain agents to assess whether the queued reward will compromise an ongoing arbitration. If that's the case, agents can ask a project to cancel the transaction, or just freeze the vault.