Security Audit

Immunefi

10/09/2021

PREPARED FOR

Immunefi

CONTACT

audits@arcadiamgroup.com

https://t.me/thearcadiagroup

Table of Contents

Executive Summary

A Representative Party of **IMMUNEFI** (**"CLIENT"**) engaged The Arcadia Group ("Arcadia"), a software development, research, and security company, to conduct a review of the following **IMMUNEFI** smart contracts on the **Immunefi** repo () at Commit **#58f0ce749e01d20097ecfa3acdb433cf58dca56c on tag arcadia_audit_start** .

The audit scope includes the following files:

- arbitration/ArbiterDAO.sol
- arbitration/ArbiterNFT.sol
- arbitration/Arbitration.sol
- governance/Executor.sol
- governance/Governor.sol
- governance/SimpleTimelock.sol
- governance/SimpleTimelockFactory.sol
- governance/Vault.sol
- governance/Governor.sol
- lib/*.sol
- math/*.sol
- utils/*.sol
- vault/*.sol
- vault/specialized/*.sol
- BugReportNotary.sol
- Escrow.sol

Arcadia completed this security review using various methods primarily consisting of dynamic and static analysis. This process included a line-by-line analysis of the in-scope contracts, optimization analysis, analysis of key functionalities and limiters, and reference against intended functionality.

There were X issues found, 1 of which were deemed to be ‘critical’, and 7 of which were rated as ‘high’.

Severity Rating	Number of Original	Number of Remaining

	Occurences	Occurences
Critical	1	0
High	7	0
Medium	6	6
Low	1	0
None/Unknown/Note	3	0

Findings

1. Quorum Minimum

Issue: IMFI-1
Severity: High
Likelihood: Medium
Impact: High

Target: ArbitDAO.sol
Category: Quorum Minimum
Finding Type: Dynamic

Function `quorum` should have a minimum value as if the NFT total supply is small (1 or 2), the owner of the NFTs can create an arbitrary vote (`AdmitVote` and `MigrateVote`). This is because the owner of the single NFT (or 2 NFTs) can vote for any direction of the vote to win.

```
function quorum() public view override returns (uint256) {
   return _NFT.totalSupply() / 2;
 }

function createAdmitVote(uint256 tokenId, address newArbiter) external {
   admissions[newArbiter].summary.status = uint256(VoteStatus.CREATED);
   voteAdmit(tokenId, newArbiter);
 }

 /// Propose the migration of the DAO to a new contract.
 function createMigrateVote(uint256 tokenId, address newDao) external {
   migrations[newDao].summary.status = uint256(VoteStatus.CREATED);
   voteMigrate(tokenId, newDao);
 }
```

Action Recommended: Functions `createAdmitVote` and `createMigrateVote` should be only enabled if the total supply of the NFTs is at a minimum value

```
function createAdmitVote(uint256 tokenId, address newArbiter) external {
  require(_NFT.totalSupply() >= MINIMUM_SUPPLY);
  admissions[newArbiter].summary.status = uint256(VoteStatus.CREATED);
  voteAdmit(tokenId, newArbiter);
}

/// Propose the migration of the DAO to a new contract.
function createMigrateVote(uint256 tokenId, address newDao) external {
  require(_NFT.totalSupply() >= MINIMUM_SUPPLY);
  migrations[newDao].summary.status = uint256(VoteStatus.CREATED);
  voteMigrate(tokenId, newDao);
}
```

Team Resolution: If the quorum falls below the suggested threshold, this would wedge the DAO contract. The DAO would be unable to admit new members to bring the membership above the threshold.

IMO, this is akin to a 51% attack against the DAO. The DAO must be initialized with enough members to keep it independent, and thereafter it is the responsibility of the DAO itself to sagely administer the membership.

A future work may be to allow the `Governor` DAO to admit/remove members of the DAO over the self-governance

2. NFT Burn

Issue: IMFI-2
Severity: High
Likelihood: High
Impact: High

Target: ArbiterNFT.sol
Category: NFT Burn
Finding Type: Dynamic

Function `burn` allows the contract owner to burn any `ArbiterNFT` token. This could create super power for the contract owner.

```
function burn(uint256 tokenId) external override {
   address msgSender = _msgSender();
   require(_isApprovedOrOwner(msgSender, tokenId) || msgSender == owner(),
"ArbiterNFT: not approved");
   _burn(tokenId);
 }
```

Action Recommended: It is recommended that the team should review whether the contract owner has the superpower to burn any ArbiterNFT token, or the contract owner should be a governance or multisignature wallet contract.

Team Resolution: This is intended behavior. This is the mechanism through with the `ArbiterDAO` can eject members for non-participation in arbitration

3. RNG Contract

Issue: IMFI-3
Severity: Medium
Likelihood: note
Impact: Note

Target: Arbitration.sol
Category: External Contract
Finding Type: Dynamic

The `Arbitration` contract uses an external random generator `RandomAuRa`. Random number generation in blockchain is a common topic for discussion regarding the security of generated numbers. The security of this external random generator is out of scope of the audit. The external contract is used in functions `beginDispute`, `impanelJury`, and `vote`.

```
 IRandomAuRa public constant override RNG =
IRandomAuRa(0x5870b0527DeDB1cFBD9534343Feda1a41Ce47766);
```

4. Gas Saving

Issue: IMFI-4
Severity: Low
Likelihood: High
Impact: Informational

Target: Arbitration.sol
Category: Gas Saving
Finding Type: Dynamic

Function `registerReport` computes keccak256 for a constant variable `KEY_REPORTER`. The value of keccak256 for the constant variable can be pre-computed for minimizing gas consumption.

```
} else if (keyHash == keccak256(bytes(KEY_REPORTER))) {
          reporter = abi.decode(reportLeaves[i].value, (address));
        }
```

5. Potential out-of-gas in registerReport

Issue: IMFI-5
Severity: High
Likelihood: Medium
Impact: Medium

Target: Arbitration.sol
Category: Potential out-of-gas
Finding Type: Dynamic

Function `registerReport`, `fullDisclose`, and `fullDiscloseCommentary` might consume lots of gas and could throw out gas exceptions if the report and commentary contain too many leaves.

```
function registerReport(
   bytes32 reportId,
   INotary.LeafData[] calldata reportLeaves,
   INotary.LeafData[] calldata commentaryLeaves
 ) external override {
   NOTARY.fullDisclose(reportId, reportLeaves);
   bytes32 commentaryId = NOTARY.getCanonicalCommentary(reportId);
   NOTARY.fullDiscloseCommentary(commentaryId, commentaryLeaves);

   Report storage report = reports[reportId];
   {
     string memory project;
     address reporter;
     for (uint256 i; i < reportLeaves.length; i++) {
       bytes32 keyHash = keccak256(bytes(reportLeaves[i].key));
       if (keyHash == keccak256(bytes(NOTARY.KEY_PROJECT()))) {
         project = abi.decode(reportLeaves[i].value, (string));
       } else if (keyHash == keccak256(bytes(KEY_REPORTER))) {
         reporter = abi.decode(reportLeaves[i].value, (address));
       }
     }
     require(bytes(project).length != 0, "Arbitration: no project");
     require(reporter != address(0), "Arbitration: no reporter");
     bytes32 projectId = _addProject(project);
     report.projectId = projectId;
     report.reporter = reporter;
     int128 timestamp = int128(uint128(NOTARY.getReport(reportId).timestamp));
     uint256 queueId = _projectQueue[projectId].push(timestamp);
```

```
    _queueIdToReportId[projectId][queueId] = reportId;
  }

  {
    uint256 bundleSize;
    for (uint256 i; i < commentaryLeaves.length; i++) {
      if (keccak256(bytes(commentaryLeaves[i].key)) ==
keccak256(bytes(NOTARY.KEY_BUNDLESIZE()))) {
        bundleSize = abi.decode(commentaryLeaves[i].value, (uint256));
        break;
      }
    }
    IVaultCoordinator.Payout[] memory payout = new
IVaultCoordinator.Payout[](bundleSize);
    for (uint256 i; i < commentaryLeaves.length; i++) {
      INotary.LeafData calldata leafData = commentaryLeaves[i];
      bytes calldata keyBytes = bytes(leafData.key);
      if (keyBytes.length > 7 && bytes7(keyBytes[:7]) == bytes7("bundle ")) {
        payout[bytes(keyBytes[7:]).atoi()] = abi.decode(leafData.value,
(IVaultCoordinator.Payout));
      }
    }
    uint256 length = payout.length;
    for (uint256 i; i < length; i++) {
      if (payout[i].beneficiary == address(0)) {
        IVaultCoordinator.Payout memory replacement;
        while (replacement.beneficiary == address(0) && length > i) {
          length--;
          replacement = payout[length];
        }
        payout[i] = replacement;
      }
    }
    assembly {
      mstore(payout, length)
    }
```

```
    report.payoutId = _addPayout(payout);
  }
 }
```

6. Payout Copy

Issue: IMFI-6
Severity: Critical
Likelihood: High
Impact: High

Target: Arbitration.sol
Category: Array Length
Finding Type: Dynamic

In function `_addPayout`, at line 135, at this statement, `payouts[payoutId]` is still empty, thus `length` is 0, thus no payouts would be copied to `payouts` mapping at `payoutId` key.

```
function _addPayout(IVaultCoordinator.Payout[] memory src) internal returns
(bytes32 payoutId) {
   payoutId = keccak256(abi.encode(src));
   IVaultCoordinator.Payout[] storage dst = payouts[payoutId];
   if (dst.length == 0) {
     // Solidity is unable to copy array-of-structs from memory to storage. So
     // we have to do it ourselves.
     uint256 length = payouts[payoutId].length;
     assembly {
       sstore(dst.slot, length)
     }
     for (uint256 i; i < length; i++) {
       dst[i] = src[i];
     }
```

```
  }
}
```

Action Recommended: Local variable `length` should be set to the length of `src` input parameter array.

```
    uint256 length = src.length;
```

7. Report Status

Issue: IMFI-7
Severity: High
Likelihood: High
Impact: High

Target: Arbitration.sol
Category: Report Status
Finding Type: Dynamic

In function `beginDispute`, the report `status` was never set to DISPUTE in any function while the `DISPUTE` status is checked in function `finishDispute`. This basically makes the function `finishDispute` always fail/revert with exception.

Furthermore, functions `impanelJury` and `vote` never check report `status` for `DISPUTE`.

```
function beginDispute(bytes32 reportId) external override {
  require(RNG.isCommitPhase(), "Arbitration: wait for commit phase");
  Dispute storage dispute = reports[reportId].dispute;
  require(
    block.number > dispute.startBlock + RNG.collectRoundLength() &&
dispute.jurors[0] == 0,
    "Arbitration: dispute already started"
  );
  require(NFT.totalSupply() >= dispute.jurors.length, "Arbitration: not
enough jurors");
```

```
    dispute.startBlock = RNG.nextCommitPhaseStartBlock();
    STAKING.safeTransferFrom(_msgSender(), TREASURY, FEE);
  }
```

Action Recommended: Function `beginDispute` should set report `status` to `DISPUTE`, and functions `impanelJury` and `vote` should check whether the report is in `DISPUTE` status

8. Registered Project Status

Issue: IMFI-8
Severity: High
Likelihood: High
Impact: High

Target: Arbitration.sol
Category: Report Status
Finding Type: Dynamic

The project report `status` was never assigned to `REGISTERED` status, which was checked in function `flush`. This makes the function `flush` always fails if there is any project that has never begun a dispute process, as in this case report status is always `UNSET`. Failure of function `flush` would never make project status PAYABLE, thus no payment would ever be made.

```
function flush(bytes32 projectId, uint256 howMany) external override {
    Heaps.Heap storage queue = _projectQueue[projectId];
    uint64 triageDelay = NOTARY.getProjectTriageDelay(projects[projectId]);
    for (uint256 i; i < howMany; i++) {
      uint256 queueId = queue.pop();
      bytes32 reportId = _queueIdToReportId[projectId][queueId];
      Report storage report = reports[reportId];
```

```
    ReportStatus status = report.status;
    require(
      status == ReportStatus.REGISTERED || status ==
ReportStatus.DISPUTE_FINISHED,
      "Arbitration: wrong status"
    );
    uint64 timestamp = NOTARY.getReport(reportId).timestamp;
    require(
      block.timestamp > timestamp + triageDelay + 1 days + VOTING_BLOCKS *
BLOCK_TIME + 1 days,
      "Arbitration: too early"
    );
    report.status = ReportStatus.PAYABLE;
    emit ReportPayable(reportId);
  }
}
```

Action Recommended: The project report `status` should be set to `REGISTERED` in function `registerProject`. This would make projects flushable if there is no dispute process submitted during `triage delay`.

9. External contract XDAI_ETH_AMB

Issue: IMFI-9
Severity: Medium
Likelihood: Note
Impact: Note

Target: Arbitration.sol
Category: External Contract
Finding Type: Dynamic

The contract uses an external contract XDAI_ETH_AMB to send out payments. This is out of scope of the audit

10. Proposal Execution

Issue: IMFI-10
Severity: High
Likelihood: High
Impact: High

Target: Governor.sol
Category: Proposal Status
Finding Type: Dynamic

In `execute` function, the `status` of the executing proposal becomes closed and no proposal execution is made if one of the dependencies is closed. Is it an intended behavior? Shouldn't the executing proposal be closed only if all dependencies are closed?

```
function execute(
   Call[] calldata calls,
   Call calldata check,
   uint256 gasLimit,
   bytes32[] calldata dependencies,
   uint256 start,
   uint256 end,
   address submitter
 ) external returns (bool success, bytes[] memory returndata) {
   bytes32 callId = _getCallId(calls, check, gasLimit, dependencies, start,
end);
   Proposal storage proposal = proposals[callId];
   _requireOrSim(proposal.summary.vote.status == uint8(VoteStatus.CREATED),
"Governor: invalid proposal");
   _requireOrSim(block.timestamp > proposal.summary.end, "Governor: voting not
finished");

   // Check if the vote passed.
   {
     uint256 votesFor = proposal.summary.vote.votesFor;
     uint256 votesAgainst = proposal.summary.vote.votesAgainst;
```

```
    if (votesFor + votesAgainst <= _quorum || votesAgainst >= votesFor) {
      proposal.summary.vote.status = uint8(VoteStatus.CLOSED);
      emit VoteFailed(callId);
      return (success, returndata);
    }
  }

  // Check if the dependencies are satisfied.
  {
    uint256 length = dependencies.length;
    for (uint256 i; i < length; i++) {
      bytes32 dependency = dependencies[i];
      ProposalStatus status =
ProposalStatus(proposals[dependency].summary.vote.status);
      _requireOrSim(
        status == ProposalStatus.SUCCESS || status == ProposalStatus.CLOSED,
        "Governor: dependency not resolved"
      );
      if (status == ProposalStatus.CLOSED) {
        proposal.summary.vote.status = uint8(ProposalStatus.CLOSED);
        return (success, returndata);
      }
    }
  }

  // Do the call(s)
  (success, returndata) = _execute(callId, calls, check, gasLimit,
submitter);
  // Executor._execute contains protection against reentrancy.
  if (success) {
    proposal.summary.vote.status = uint8(ProposalStatus.SUCCESS);
  } else {
    proposal.summary.vote.status = uint8(ProposalStatus.CLOSED);
  }
 }
```

[redacted]

Team Resolution: This is intended behavior. If any of the dependencies have failed, then the proposal is also failed and gets marked as such.

11. Vesting Contract

Issue: IMFI-11
Severity: Medium
Likelihood: Note
Impact: Note

Target: Governor.sol
Category: External Contract
Finding Type: Dynamic

The contract inherits from the vesting contract `ERC2612VestingNFTFunder` and some functions call the vesting contract functions. However vesting contracts are out of scope of the audit.

12. Escrow Token Whitelist

Issue: IMFI-12
Severity: Medium
Likelihood: High
Impact: Medium

Target: Escrow.sol
Category: Token Payment Whitelist
Finding Type: Dynamic

The contract `Escrow` should have a whitelisted token list for payments.

Function `deposit` and `withdraw` should also have reentrancy guard.

13. Safe Check for Deflationary Token

Issue: IMFI-13
Severity: Medium

Target: Escrow.sol
Category: Deflationary Token

Likelihood: High
Impact: High

Finding Type: Dynamic

Function `deposit` should check whether the token is a deflationary token or token with fees on transfer. If a user deposits a deflationary token amount, the token amount received by the contract will be less than the `amount` input parameter.

```
function deposit(
   bytes32 reportRoot,
   address paymentToken,
   uint256 amount,
   uint256 leafPosition
 ) external payable {
   require(amount > 0, "Escrow: Amount must be larger than zero");
   notary.getReport(reportRoot); // reverts for nonexistent report
   require(notary.getReport(reportRoot).timestamp != 0, "Escrow: Report not
submitted");
   balances[_getBalanceID(reportRoot, leafPosition, paymentToken)] += amount;
   emit Deposit(reportRoot, _msgSender(), paymentToken, leafPosition, amount);
   if (paymentToken == NATIVE_ASSET) {
     require(msg.value == amount, "Escrow: Insufficient funds sent");
   } else {
     require(msg.value == 0, "Escrow: Native asset sent for ERC20 payment");
     IERC20(paymentToken).safeTransferFrom(_msgSender(), address(this),
amount);
   }
 }
```

Action recommended: The function should check the actual received token amount.

```
function deposit(
   bytes32 reportRoot,
   address paymentToken,
   uint256 amount,
```

```
    uint256 leafPosition
  ) external payable {
    require(amount > 0, "Escrow: Amount must be larger than zero");
    notary.getReport(reportRoot); // reverts for nonexistent report
    require(notary.getReport(reportRoot).timestamp != 0, "Escrow: Report not
submitted");

    if (paymentToken == NATIVE_ASSET) {
      require(msg.value == amount, "Escrow: Insufficient funds sent");
      balances[_getBalanceID(reportRoot, leafPosition, paymentToken)] +=
amount;
      emit Deposit(reportRoot, _msgSender(), paymentToken, leafPosition,
amount);
    } else {
      require(msg.value == 0, "Escrow: Native asset sent for ERC20 payment");
      uint256 balBefore = IERC20(paymentToken).balanceOf(address(this));
      IERC20(paymentToken).safeTransferFrom(_msgSender(), address(this),
amount);
      uint256 actualReceived =
IERC20(paymentToken).balanceOf(address(this)).sub(balBefore);

      balances[_getBalanceID(reportRoot, leafPosition, actualReceived)] +=
actualReceived;
      emit Deposit(reportRoot, _msgSender(), paymentToken, leafPosition,
actualReceived);
    }
  }
```

14. PriceFeed external contract

Issue: IMFI-14
Severity: Medium
Likelihood: note
Impact: Note

Target: PriceFeed.sol
Category: External Contract
Finding Type: Dynamic

`PriceFeed` calls external contracts to read asset prices from external oracle contracts in which the auditors have no information about the code and the external oracle code is not in the scope of the audit.

15. Wrong organizer for vaults registration

Issue: IMFI-15
Severity: High
Likelihood: High
Impact: High

Target: VaultCoordinator.sol
Category: Vault Organizer
Finding Type: Dynamic

The function `setOrganizer` code registers old vaults for the same old organizer, which is unexpected.

```
function setOrganizer(string calldata project, address newOrganizer) external
override nonReentrant {
   require(newOrganizer != address(0), "VaultCoordinator: zero address");
   require(
     newOrganizer.supportsInterface(type(IPayoutOrganizer).interfaceId),
     "VaultCoordinator: new organizer does not support interface
IPayoutOrganizer"
   );
   ProjectInfo storage projectInfo = _projects[project];
   require(_msgSender() == projectInfo.admin, "VaultCoordinator: only project
admin can set organizer");
   IPayoutOrganizer oldOrganizer = projectInfo.payoutOrganizer;
   projectInfo.payoutOrganizer = IPayoutOrganizer(newOrganizer);

   // It is possible to "remove" a vault through this mechanism. However,
   // omitting a vault here won't actually prevent it from being used pay
   // bounties. The `fallbackOrder` argument to `collectPayment` can always
```

```
  // name that vault explicitly.
  IBountyVault[] memory oldVaults = oldOrganizer.getVaults();
  for (uint256 i; i < oldVaults.length; i++) {
    oldOrganizer.vaultRegistered(oldVaults[i]);
  }
  emit OrganizerChanged(project, address(oldOrganizer), newOrganizer);
}
```

Action recommended: The function should register the old vaults of the old organizer to the new organizer.

16. Using onlyOwner modifier

Issue: IMFI-16
Severity: None
Likelihood: None
Impact: Informational

Target: ArbiterNFT.sol
Category: Code Readability
Finding Type: Dynamic

Function `mint` should use the modifier `onlyOwner` for better code readability.

17. Unclarified TODO

Issue: IMFI-17
Severity: Unknown
Likelihood: Unknown
Impact: Unknown

Target: Arbitration.sol
Category: Unknown TODO
Finding Type: Dynamic

Function `vote` has unclarified `TODO`: `// TODO: approval voting`

Team Resolution: This was a now-scrapped idea for an improvement to the voting system for arbitration. Instead of using first-past-the-post

https://en.wikipedia.org/wiki/First-past-the-post_voting system for choosing the outcome of arbitration, we could instead use approval voting.
https://en.wikipedia.org/wiki/Approval_voting so that arbiters can vote for multiple acceptable outcomes

18. Unclarified TODO

Issue: IMFI-18
Severity: Unknown
Likelihood: Unknown
Impact: Unknown

Target: BountyVault.sol
Category: Unknown TODO
Finding Type: Dynamic

Function `triggerWithdraw` has unclarified `TODO`.

Conclusion

Arcadia identified issues that occurred at hash **#58f0ce749e01d20097ecfa3acdb433cf58dca56c on tag arcadia_audit_start**.
Arcadia also reviewed the fixes for the issues at this PR https://github.com/immunefi-team/protocol/pull/41.

Disclaimer

While best efforts and precautions have been taken in the preparation of this document, The Arcadia Group and the Authors assume no responsibility for errors, omissions, or damages resulting from the use of the provided information. Additionally, Arcadia would like to emphasize that the use of Arcadia's services does not guarantee the security of a smart contract or set of smart contracts and does not guarantee against attacks. One audit on its own is not enough for a project to be considered secure; that categorization can only be earned through extensive peer review and battle testing over an extended period.