IGRA

Core Smart Contracts

Security Assessment Report

Version: 2.1

February, 2026

Contents

Introduction **2**
- Disclaimer 2
- Document Structure 2
- Overview 2

Security Assessment Summary **3**
- Scope 3
- Approach 3
- Coverage Limitations 4
- Findings Summary 4

Detailed Findings **5**

Summary of Findings **6**
- Inverted Time Check Allows Only Past Schedule Updates 7
- Governance Constructor Parameters Lack Validation 8
- Released Event Emitted On Failed Transfer 10
- Quorum Percentage Is Not Validated 11
- UpdatePoolTime Allows Start To Be Zero 12
- Immutable Minter Reduces Operational Flexibility 13
- Lack Of Governance Guardian Pause And Veto Controls 14
- Lack Of Late Quorum Protection 15
- NonReentrant Initialisation Shifts Gas Cost To First Caller 16
- Lack Of Zero Address Checks In Claimable Token Rescue 17
- Shared Upgrade Authority Across Proxies Due To Immutable Owner 18
- Lack Of Zero Address Check For Owner 20
- Non-standard ERC20 Behaviour Can Break Pre-minted Vesting Transfers 21
- Two Step Wallet Update Is Not Used 23
- Non-standard Return Value For Mint Function 24
- Lack Of Governance Timelock 25
- Fee Contract Cannot Receive Native Token 26
- Token Rescue Assumes VestingPools Is Sole Source Of Token Supply 27
- Vesting Schedule Can Be Modified During Active Vesting 28
- Miscellaneous General Comments 30

A Vulnerability Severity Classification **32**

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Igra components in scope. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Igra components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Igra components in scope.

Overview

Igra Network is an attestation protocol where independent attesters stake the IGRA token and submit attestations for network blocks. Correct participation is rewarded, while invalid or missed attestations can be challenged and penalized via slashing. The protocol also includes the IGRA token (with voting power), onchain governance, and vesting/rewards distribution contracts.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the IgraLabs/igra-core-contracts and IgraLabs/igra-eip4788-modifications repositories.

The scope of this time-boxed review was strictly limited to the following commits and files:

- `f24c22a`:
  - `src/vestingPools/Types.sol`
  - `src/proxy/OwnedUUPSUpgradeable.sol`
  - `src/vestingPools/Constants.sol`
  - `src/Governance.sol`
  - `src/vestingPools/interfaces/IVestingPools.sol`
  - `src/vestingPools/interfaces/IMintable.sol`
  - `src/vestingPools/utils/SafeUints.sol`
  - `src/vestingPools/utils/Claimable.sol`
  - `src/IgraVotingPower.sol`
  - `src/IgraToken.sol`
  - `src/VestingPools.sol`
  - `src/Fee.sol`
- `08533a1`:
  - `src/modified-eip4788-contract.bytecode`

Deployment and initialization of IGRA core contracts are deterministic and cryptographically committed in the genesis file. Constructor parameters are verified off-chain prior to launch. Developers assume this pre-deployment verification as part of the system's trust model.

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity and Assembly.

For the Solidity components, the manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team may use the following automated testing tools:

- Aderyn: `https://github.com/Cyfrin/aderyn`
- Slither: `https://github.com/trailofbits/slither`
- Mythril: `https://github.com/ConsenSys/mythril`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 20 issues during this assessment. Categorised by their severity:

- Medium: 1 issue.
- Low: 3 issues.
- Informational: 16 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Igra components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected components(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
IGRA-01	Inverted Time Check Allows Only Past Schedule Updates	Medium	Resolved
IGRA-02	Governance Constructor Parameters Lack Validation	Low	Resolved
IGRA-03	Released Event Emitted On Failed Transfer	Low	Resolved
IGRA-04	Quorum Percentage Is Not Validated	Low	Resolved
IGRA-05	UpdatePoolTime Allows Start To Be Zero	Informational	Closed
IGRA-06	Immutable Minter Reduces Operational Flexibility	Informational	Closed
IGRA-07	Lack Of Governance Guardian Pause And Veto Controls	Informational	Closed
IGRA-08	Lack Of Late Quorum Protection	Informational	Closed
IGRA-09	NonReentrant Initialisation Shifts Gas Cost To First Caller	Informational	Resolved
IGRA-10	Lack Of Zero Address Checks In Claimable Token Rescue	Informational	Resolved
IGRA-11	Shared Upgrade Authority Across Proxies Due To Immutable Owner	Informational	Resolved
IGRA-12	Lack Of Zero Address Check For Owner	Informational	Resolved
IGRA-13	Non-standard ERC20 Behaviour Can Break Pre-minted Vesting Transfers	Informational	Closed
IGRA-14	Two Step Wallet Update Is Not Used	Informational	Closed
IGRA-15	Non-standard Return Value For Mint Function	Informational	Resolved
IGRA-16	Lack Of Governance Timelock	Informational	Closed
IGRA-17	Fee Contract Cannot Receive Native Token	Informational	Closed
IGRA-18	Token Rescue Assumes VestingPools Is Sole Source Of Token Supply	Informational	Closed
IGRA-19	Vesting Schedule Can Be Modified During Active Vesting	Informational	Closed
IGRA-20	Miscellaneous General Comments	Informational	Resolved

IGRA-01	Inverted Time Check Allows Only Past Schedule Updates		
Assets	`VestingPools.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

A logic error in `updatePoolTime()` inverts the time check, making it impossible to legitimately adjust a pool's vesting schedule and instead only allowing updates after the vesting period has already ended.

This makes legitimate schedule updates impossible while vesting is ongoing, and enables schedule changes that are inconsistent with the stated intent of preventing late updates.

```
function updatePoolTime(uint256 poolId, uint32 start, uint16 vestingDays) external override onlyOwner {
    PoolParams memory pool = _getPool(poolId);

    require(pool.isAdjustable, "VPools: non-adjustable");
    require(uint256(pool.sAllocation) * SCALE > uint256(pool.vested), "VPools: fully vested");
    uint256 end = uint256(start) + uint256(vestingDays) * 1 days;
    // `end` may NOT be in the past, unlike `start` that may be even zero
    require(_timeNow() > end, "VPools: too late updates");
    pool.start = start;
    pool.vestingDays = vestingDays;
    _pools[poolId] = pool;

    emit PoolUpdated(poolId, start, vestingDays);
}
```

Recommendations

Fix the inverted comparison by requiring the current timestamp to be strictly less than the new end time.

Resolution

This issue has been resolved in commit 3d54d72 with the following comment:

> *"We agree with the issue and have fixed it. However, it would not affect Igra in practice because no adjustable pools are used in the genesis configuration; isAdjustable is false for all pools."*

IGRA-02	Governance Constructor Parameters Lack Validation		
Assets	`Governance.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `Governance` constructor accepts several configuration parameters without validating them at deployment, which can lead to misconfiguration or a non-functional governance system:

- `votingPeriod_`: OpenZeppelin `GovernorSettings` enforces `votingPeriod_ != 0`, but does not enforce a minimum voting period beyond that. As a result, a very small `votingPeriod_` can be configured, reducing the time available for participation and review and allowing proposals to complete quickly.
- `votingPower_`: Not checked for `address(0)`. If deployed with a zero-address voting token, vote weight lookups will revert and the governance system will not function.
- `votingDelay_` and `proposalThreshold_`: OpenZeppelin `GovernorSettings` does not enforce minimum or maximum bounds, so extreme values can be set at deployment. These values are only correctable via governance, which may be impractical if the initial configuration prevents proposal creation or meaningful voting.
- `quorumPercentage_`: Not validated in the constructor and covered by the separate quorum percentage bounds finding.

```
constructor(
    string memory name_,
    IVotes votingPower_,
    uint48 votingDelay_,
    uint32 votingPeriod_,
    uint256 proposalThreshold_,
    uint256 quorumPercentage_
) Governor(name_) GovernorSettings(votingDelay_, votingPeriod_, proposalThreshold_) GovernorVotes(votingPower_) {
    quorumPercentage = quorumPercentage_;
}
```

Recommendations

Validate that `votingPower_` is non-zero during deployment.

Consider enforcing reasonable bounds for `votingDelay_`, `votingPeriod_`, and `proposalThreshold_` during deployment to reduce misconfiguration risk, with the exact thresholds chosen to match the intended governance model.

Resolution

The constructor now validates that `votingPower_` is non-zero and enforces `quorumPercentage_` bounds. `votingPeriod_` remains validated by OpenZeppelin `GovernorSettings`, while `votingDelay_` and `proposalThreshold_` are intentionally left unbounded to allow zero values.

This issue has been resolved in commit 3d54d72 with the following comment:

> *"Not applicable in Igra. Constructor parameters are committed in genesis and verified pre-launch, so on-chain validation is not strictly required for this system. Nevertheless, we improved the code to address the issue."*

IGRA-03	Released Event Emitted On Failed Transfer		
Assets	`Fee.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `releaseTo()` function emits `Released(to, amount)` even when `_release()` returns `false`.

This can cause the emitted event to diverge from the actual transfer outcome, since the function explicitly allows silent failure by returning a status value instead of reverting.

```
function releaseTo(address to, uint256 amount) external onlyReleaser returns (bool status) {
    status = _release(to, amount);

    emit Released(to, amount);
}

function _release(address to, uint256 amount) private returns (bool status) {
    require(amount <= address(this).balance, "Insufficient balance");
    require(to != address(0), "Release to zero address");

    (status,) = to.call{value: amount}("");
}
```

Recommendations

Emit `Released` only when `status` is `true`, or include the `status` value in the event so that offchain consumers can distinguish successful and failed transfers.

Resolution

This issue has been resolved in commit 3d54d72.

IGRA-04	Quorum Percentage Is Not Validated		
Assets	`Governance.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

`quorumPercentage` is set in the constructor without validating bounds, which can lead to an insecure or unusable governance configuration:

- If `quorumPercentage_` is set to `0`, then `quorum()` returns `0`. Proposals can satisfy quorum with no participation, and governance outcomes depend only on whichever votes are cast.
- If `quorumPercentage_` is set to a value greater than `100`, then `quorum()` can exceed the total voting power supply at the snapshot. Proposals will never reach quorum, effectively disabling governance.

```
constructor(
    string memory name_,
    IVotes votingPower_,
    uint48 votingDelay_,
    uint32 votingPeriod_,
    uint256 proposalThreshold_,
    uint256 quorumPercentage_
) Governor(name_) GovernorSettings(votingDelay_, votingPeriod_, proposalThreshold_) GovernorVotes(votingPower_) {
    quorumPercentage = quorumPercentage_;
}
```

Recommendations

Validate `quorumPercentage_` during deployment to enforce a sensible range, such as requiring it to be greater than `0` and less than or equal to `100`.

If governance parameter updates are expected post-deployment, consider making the quorum percentage configurable through governance rather than immutable.

Resolution

Added `require(quorumPercentage_ <= 100)`. (0% quorum intentionally allowed.)

This issue has been resolved in commit 3d54d72 with the following comment:

> *"Not applicable in Igra. Constructor parameters are committed in genesis and verified pre-launch, so on-chain validation is not strictly required for this system. Nevertheless, we improved the code to address the issue."*

IGRA-05	UpdatePoolTime Allows Start To Be Zero
Assets	`VestingPools.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

`updatePoolTime()` allows setting `start` to zero.

This behaviour is explicitly documented in the inline comment, but it may be surprising in production configurations because it can materially change the releasable amount calculation and effectively accelerate vesting relative to the original schedule.

```
function updatePoolTime(uint256 poolId, uint32 start, uint16 vestingDays) external override onlyOwner {
    PoolParams memory pool = _getPool(poolId);

    require(pool.isAdjustable, "VPools: non-adjustable");
    require(uint256(pool.sAllocation) * SCALE > uint256(pool.vested), "VPools: fully vested");
    uint256 end = uint256(start) + uint256(vestingDays) * 1 days;
    // `end` may NOT be in the past, unlike `start` that may be even zero
    require(_timeNow() > end, "VPools: too late updates");

    pool.start = start;
    pool.vestingDays = vestingDays;
    _pools[poolId] = pool;

    emit PoolUpdated(poolId, start, vestingDays);
}
```

Recommendations

Consider documenting the ability to set `start` to zero in the external-facing documentation for adjustable pools.

If `start` being zero is not intended for production use, enforce `start != 0` in `updatePoolTime()` and ensure schedule changes are constrained to the intended timing window.

Resolution

The issue was closed by the project team with the following comment:

> *"Intended behavior. Also, unused in Igra because no pools with start time of 0 are committed in genesis. Documented in inline comment."*

IGRA-06	Immutable Minter Reduces Operational Flexibility
Assets	`IgraToken.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

`IgraToken` assigns mint authority to an immutable `MINTER` address at deployment.

This is compatible with setting `MINTER` to the `VestingPools` contract address, since `VestingPools` is designed to call `mint()` during pool creation for pre-minted allocations and during releases for non-pre-minted allocations.

However, because `MINTER` is immutable, mint authority cannot be rotated if the minter contract needs to change, and a token redeployment would be required to recover.

```
contract IgraToken is ERC20, ERC20Permit, ERC20Votes, ERC20Capped {
    uint256 public constant MAX_SUPPLY = 10e27;
    address public immutable MINTER;

    constructor(address minter) ERC20("Igra Token", "IGRA") ERC20Permit("Igra Token") ERC20Capped(MAX_SUPPLY) {
        require(minter != address(0), "Minter address cannot be zero");
        MINTER = minter;
    }
```

Recommendations

Consider making the minter address configurable via a controlled update mechanism, such as an owner or governance-authorised setter with an event emission, to support operational migration without redeploying the token.

Resolution

The issue was closed by the project team with the following comment:

> *"Intentional design choice to improve protocol determinism and security."*

IGRA-07	Lack Of Governance Guardian Pause And Veto Controls
Assets	`Governance.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The governance design executes proposals without a timelock and does not include an emergency control path such as a guardian or multisig pause, veto, or broad cancel mechanism.

The only cancellation path available in the inherited OpenZeppelin `Governor` implementation is limited to the proposal's proposer and is only permitted while the proposal is still in the pending state, before voting begins.

Once voting has started, there is no mechanism in the `Governance` contract to pause governance actions or veto proposals that later appear unsafe.

Recommendations

Consider introducing a timelock-based execution model that allows an emergency actor to cancel queued proposals during the timelock delay.

If a timelock is out of scope by design, consider implementing an explicit governance pause and emergency cancellation mechanism with clearly defined authority and scope, and document this trust assumption in the contract documentation.

Resolution

The issue was closed by the project team with the following comment:

> *"Design choice for Phase 0: governance is intentionally minimal; additional controls are planned to evolve via future DAO proposals."*

IGRA-08	Lack Of Late Quorum Protection
Assets	`Governance.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `Governance` contract does not include a mechanism to extend the voting period when quorum is reached late in the vote.

This allows a large voter to withhold votes until the final block of the voting period, potentially reaching quorum and swaying the outcome when there is little or no time remaining for other participants to react.

```
contract Governance is Governor, GovernorSettings, GovernorCountingSimple, GovernorVotes {
    /// @notice Quorum percentage (e.g., 4 = 4%)
```

OpenZeppelin provides `GovernorPreventLateQuorum` to mitigate this behaviour by extending the proposal deadline when quorum is reached, ensuring a minimum time window remains after quorum is achieved.

Recommendations

Consider integrating `GovernorPreventLateQuorum` and configuring a non-zero late quorum vote extension suitable for the intended voting cadence.

Resolution

The issue was closed by the project team with the following comment:

> *"Design choice for Phase 0: governance is intentionally minimal; additional controls are planned to evolve via future DAO proposals."*

IGRA-09	NonReentrant Initialisation Shifts Gas Cost To First Caller
Assets	`Claimable.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The NatSpec states that this reentrancy guard does not require a constructor call.

While it is functionally correct, `_reentrancyStatus` defaults to `0`, so the first `nonReentrant` call performs a zero-to-non-zero storage write, which is more expensive than the standard pattern that initialises the status to `_NOT_ENTERED` at deployment.

This shifts a small gas cost from the deployer to the first caller, and the NatSpec is misleading if the intention is to match OpenZeppelin's gas profile.

```
/**
 * @title Claimable
 * @notice It withdraws accidentally sent tokens from this contract.
 * @dev It provides reentrancy guard. The code borrowed from openzeppelin-contracts.
 * Unlike original code, this version does not require `constructor` call.
 */
contract Claimable {
    bytes4 private constant SELECTOR_TRANSFER = bytes4(keccak256(bytes("transfer(address,uint256)")));

    uint256 private constant _NOT_ENTERED = 1;
    uint256 private constant _ENTERED = 2;

    uint256 private _reentrancyStatus;
```

Recommendations

Update the NatSpec to reflect that the reentrancy guard relies on default storage initialisation and shifts a small gas cost to the first caller.

Initialise `_reentrancyStatus` to `_NOT_ENTERED` during deployment, either via a constructor or a default assignment, to align with the standard pattern and avoid the initial zero-to-non-zero write.

Consider using transient storage via `tload` and `tstore` for the reentrancy status to reduce persistent storage writes.

Resolution

NatSpec added to document gas behavior.

This issue has been resolved in commit 3d54d72.

IGRA-10	Lack Of Zero Address Checks In Claimable Token Rescue
Assets	`Claimable.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

`_claimErc20()` does not validate `token` or `to` against `address(0)`.

If `token` is `address(0)`, the low-level call can succeed without transferring anything, and the function will treat the operation as successful.

If `to` is `address(0)`, behaviour depends on the token implementation, and may revert or burn funds.

```
/// @dev Withdraws ERC20 tokens from this contract
/// (take care of reentrancy attack risk mitigation)
function _claimErc20(address token, address to, uint256 amount) internal {
    // solhint-disable avoid-low-level-calls
    (bool success, bytes memory data) = token.call(abi.encodeWithSelector(SELECTOR_TRANSFER, to, amount));
    require(success && (data.length == 0 || abi.decode(data, (bool))), "claimErc20: TRANSFER_FAILED");
}
```

Recommendations

Validate `token != address(0)` and `to != address(0)` before performing the transfer.

Resolution

Resolved in commit 3d54d72 with the following comment:

> *"Not applicable in Igra. Constructor parameters are committed in genesis and verified pre-launch, so on-chain validation is not strictly required for this system. Nevertheless, we improved the code to address the issue."*

IGRA-11	Shared Upgrade Authority Across Proxies Due To Immutable Owner
Assets	`OwnedUUPSUpgradeable.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

`OwnedUUPSUpgradeable` stores upgrade authority in an immutable variable.

Immutable values are embedded in the implementation bytecode, meaning every proxy that points to the same implementation shares the same `OWNER`.

This differs from typical UUPS patterns where upgrade authority is stored in proxy storage, allowing distinct owners per proxy and allowing ownership to be updated via an initialiser.

As a result, if multiple proxies reuse the same implementation, they cannot have different upgrade authorities.

If the implementation is deployed with an unintended `OWNER`, upgrade authority cannot be corrected onchain for any proxy using that implementation.

```
import {UUPSUpgradeable} from "@openzeppelin/contracts/proxy/utils/UUPSUpgradeable.sol";

abstract contract OwnedUUPSUpgradeable is UUPSUpgradeable {
    address public immutable OWNER;

    constructor(address owner) {
        OWNER = owner;
    }

    modifier onlyOwner() {
        require(msg.sender == OWNER, "only owner");
        _;
    }

    function _authorizeUpgrade(address newImplementation) internal override onlyOwner {}
}
```

A concrete instance of this pattern exists in `Fee`, where the owner is set in the implementation constructor.

```
contract Fee is OwnedUUPSUpgradeable {
    address public releaser;

    event Released(address indexed receiver, uint256 amount);
    event ReleaserUpdated(address releaser);

    constructor(address owner) OwnedUUPSUpgradeable(owner) {}

    function updateReleaser(address _releaser) external onlyOwner {
        releaser = _releaser;
```

Recommendations

Store upgrade authority in proxy storage rather than an immutable implementation variable.

This can be achieved by using an initialiser to set an owner variable in storage, and by authorising upgrades based on that storage value.

Resolution

This issue has been resolved in commit e2bf619 with the following comment:

> *"Not applicable in Igra. Constructor parameters are committed in genesis and verified pre-launch, so on-chain validation is not strictly required for this system. Nevertheless, we improved the code to address the issue."*

IGRA-12	Lack Of Zero Address Check For Owner
Assets	`OwnedUUPSUpgradeable.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `OwnedUUPSUpgradeable` constructor sets the immutable `OWNER` without validating that it is non-zero.

If the contract is deployed with `owner == address(0)`, any function protected by `onlyOwner` becomes permanently inaccessible, including upgrade authorisation, and any inheriting contract that relies on owner-gated administration will be effectively frozen.

```
abstract contract OwnedUUPSUpgradeable is UUPSUpgradeable {
    address public immutable OWNER;

    constructor(address owner) {
        OWNER = owner;
    }

    modifier onlyOwner() {
        require(msg.sender == OWNER, "only owner");
        _;
    }
}
```

Recommendations

Validate the constructor argument by requiring `owner != address(0)` and revert on invalid deployment configuration.

Resolution

`initialize()` now validates `owner_ != address(0)`.

This issue has been resolved in commit 3d54d72 with the following comment:

> *"Not applicable in Igra. Constructor parameters are committed in genesis and verified pre-launch, so on-chain validation is not strictly required for this system. Nevertheless, we improved the code to address the issue."*

IGRA-13 Non-standard ERC20 Behaviour Can Break Pre-minted Vesting Transfers

Assets	`VestingPools.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The pre-minted vesting path assumes that transferring `released` tokens decreases the contract balance by exactly `released`.

Fee on transfer tokens can debit more than `released` from the contract balance due to transfer fees, and rebasing tokens can change balances independently of transfers.

In both cases, accounting is still updated as if `released` tokens were transferred without side effects, which can lead to a future release reverting due to insufficient contract balance, including for the last claimant.

This issue is only relevant if the vested token can be a fee on transfer or rebasing token, since standard OpenZeppelin ERC20 tokens do not exhibit this behaviour.

```
function _releaseTo(uint256 poolId, address to, uint256 amount) internal returns (uint256 released) {
    PoolParams memory pool = _getPool(poolId);
    _throwUnauthorizedWallet(poolId, msg.sender);

    uint256 releasable = _getReleasable(pool, _timeNow());
    require(releasable >= amount, "VPools: not enough to release");
    released = amount == 0 ? releasable : amount;

    _pools[poolId].vested = _safe96(released + uint256(pool.vested));
    totalVested = _safe96(released + uint256(totalVested));

    // reentrancy impossible (known contract called)
    if (pool.isPreMinted) {
        require(IERC20(_getToken()).transfer(to, released), "VPools:E6");
    } else {
        require(IMintable(_getToken()).mint(to, released), "VPools:E7");
    }
    emit Released(poolId, to, released);
}
```

Recommendations

Document an explicit requirement that the vested token must be a standard ERC20 without fee on transfer or rebasing behaviour.

If supporting non-standard tokens is required, update accounting to rely on balance differentials rather than assuming that `transfer(to, released)` reduces the contract balance by exactly `released`.

Resolution

The issue was closed by the project team with the following comment:

"NatSpec added: vested token must be standard ERC20 (no fee-on-transfer, no rebasing). IGRA token is compliant."

IGRA-14	Two Step Wallet Update Is Not Used
Assets	`VestingPools.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `updatePoolWallet()` function updates a pool wallet in a single step.

If an incorrect address is provided, or if the new wallet is not controlled as intended, the pool may become unrecoverable because only the current wallet is authorised to call `updatePoolWallet()`.

```
function updatePoolWallet(uint256 poolId, address newWallet) external override {
    _throwZeroAddress(newWallet);
    _throwUnauthorizedWallet(poolId, msg.sender);

    _wallets[poolId] = newWallet;
    emit WalletUpdated(poolId, newWallet);
}
```

Recommendations

Consider implementing a two step wallet update flow, where the current wallet nominates a pending wallet and the pending wallet must accept before the change becomes active.

Resolution

The issue was closed by the project team with the following comment:

> *"Design choice: VestingPools manages aggregated pools (not end-user wallets); one-step update is intentioned simplification."*

IGRA-15	Non-standard Return Value For Mint Function
Assets	`IgraToken.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `mint()` function returns a boolean value, despite minting reverting on failure and never returning `false`.

This pattern is not typical for OpenZeppelin-based mint functions, and can create interface inconsistency and integration confusion.

```
function mint(address to, uint256 amount) external onlyOwner returns (bool) {
    _mint(to, amount);
    return true;
}
```

Recommendations

Consider removing the boolean return value from `mint()`.

Resolution

Resolved: `mint()` no longer returns a boolean value. This issue has been resolved in commit 3d54d72.

IGRA-16	Lack Of Governance Timelock
Assets	`Governance.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `Governance` contract is explicitly designed to execute proposals immediately after a successful vote, without a timelock delay.

This design choice departs from common governance best practice, where a timelock is used to provide a buffer between a proposal passing and execution, allowing time for review and operational response.

```
/**
 * @title Governance
 * @notice Simple governance contract for the Igra protocol (no timelock)
 * @dev Uses IgraVotingPower as the voting token, which aggregates:
 *      - IGRA token delegated votes
 *      - Attestation effective stake
 *
 * Note: This governance executes proposals immediately after voting succeeds.
 *       The governance contract itself is the executor (no timelock delay).
 */
contract Governance is Governor, GovernorSettings, GovernorCountingSimple, GovernorVotes {
```

Recommendations

Consider implementing a timelock for governance execution, such as by integrating OpenZeppelin `GovernorTimelockControl` with a `TimelockController`, and routing proposal execution through the timelock with a non-zero delay.

Resolution

The issue was closed by the project team with the following comment:

> *"Design choice for Phase 0: governance is intentionally minimal; additional controls are planned to evolve via future DAO proposals."*

IGRA-17	Fee Contract Cannot Receive Native Token
Assets	`Fee.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `Fee` contract is intended to receive the native token as a transaction fee, but it does not implement a `receive()` function or a payable `fallback()`.

As a result, direct native token transfers to the contract will revert.

This will prevent the fee collection flow from functioning as described in the contract documentation.

```
/**
 * @title
 * @author
 * @notice It receives the iKas native token provided as fee by the transaction.
 * it let the owner to withdraw the ikas native token
 */

contract Fee is OwnedUUPSUpgradeable {
```

Recommendations

Implement a `receive()` function, or a payable `fallback()`, to allow the contract to accept native token transfers as intended.

Resolution

The issue was closed by the project team with the following comment:

> *"Not applicable in Igra. Fee receives iKAS via the consensus layer miner address (not via contract calls), so direct native token transfers are not part of the intended flow. NatSpec was updated to clarify this behavior."*

IGRA-18	Token Rescue Assumes VestingPools Is Sole Source Of Token Supply
Assets	`VestingPools.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `claimErc20()` logic for rescuing the vested token relies on `totalSupply() - totalVested` to determine how many tokens must remain in the contract.

This calculation only holds when the circulating token supply is produced exclusively through this vesting mechanism (i.e. tokens are minted to `VestingPools` up front or minted on release).

If tokens can exist outside `VestingPools` (e.g. minted to a treasury, rewards contract, or other address), then `expected` can be larger than the amount `VestingPools` must reserve for vesting.

In that case, the rescue check can revert even when `VestingPools` holds excess vested tokens that are unrelated to vesting. This does not affect normal vesting releases, but can restrict the owner's ability to recover surplus vested tokens from `VestingPools`.

```
function claimErc20(address claimedToken, address to, uint256 amount) external onlyOwner nonReentrant {
    IERC20 vestedToken = IERC20(_getToken());
    if (claimedToken == address(vestedToken)) {
        uint256 actual = vestedToken.balanceOf(address(this));
        uint256 expected = vestedToken.totalSupply() - totalVested;
        require(actual >= expected + amount, "VPools: too big amount");
    }
    _claimErc20(claimedToken, to, amount);
}
```

Recommendations

Track the amount of vested token that is expected to remain reserved in `VestingPools` based on pool configuration, and use that value in the rescue check instead of `totalSupply() - totalVested`.

Resolution

The issue was closed by the project team with the following comment:

> *"Not applicable in Igra. VestingPools is intentionally the sole minter for IGRA. NatSpec added to document invariant."*

IGRA-19	Vesting Schedule Can Be Modified During Active Vesting
Assets	`VestingPools.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `updatePoolTime()` function allows the owner to modify a pool's `start` and `vestingDays` parameters.

If `updatePoolTime()` is called after vesting has started, the releasable amount calculation will use the updated parameters, which can change how quickly tokens become releasable for current and future claimants.

This can result in earlier claimants receiving more or fewer tokens than intended relative to later claimants, depending on how the schedule is modified.

```
function updatePoolTime(uint256 poolId, uint32 start, uint16 vestingDays) external override onlyOwner {
    PoolParams memory pool = _getPool(poolId);

    require(pool.isAdjustable, "VPools: non-adjustable");
    require(uint256(pool.sAllocation) * SCALE > uint256(pool.vested), "VPools: fully vested");
    uint256 end = uint256(start) + uint256(vestingDays) * 1 days;
    // `end` may NOT be in the past, unlike `start` that may be even zero
    require(_timeNow() > end, "VPools: too late updates");

    pool.start = start;
    pool.vestingDays = vestingDays;
    _pools[poolId] = pool;

    emit PoolUpdated(poolId, start, vestingDays);
}
```

```
function _getReleasable(PoolParams memory pool, uint256 timeNow) internal pure returns (uint256) {
    if (timeNow < pool.start) return 0;

    uint256 allocation = uint256(pool.sAllocation) * SCALE;
    if (pool.vested >= allocation) return 0;

    uint256 releasable = allocation - uint256(pool.vested);
    uint256 duration = uint256(pool.vestingDays) * 1 days;
    uint256 end = uint256(pool.start) + duration;
    if (timeNow < end) {
        uint256 unlocked = uint256(pool.sUnlocked) * SCALE;
        uint256 locked = ((allocation - unlocked) * (end - timeNow)) / duration;

        releasable = locked > releasable ? 0 : releasable - locked;
    }

    return releasable;
}
```

Recommendations

Restrict `updatePoolTime()` to be callable only before vesting starts for the pool, for example by requiring the current timestamp to be strictly less than the current `pool.start`.

Resolution

The issue was closed by the project team with the following comment:

> *"Intended behavior. Also, unused in Igra because no adjustable pools are committed in genesis."*

IGRA-20	Miscellaneous General Comments
Assets	`All contracts`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **NatSpec Mismatch For Withdraw Authority**

   Related Asset(s): Fee.sol

   The contract documentation states that the owner can withdraw the native token, but withdrawals are restricted to the `releaser` role.

   The owner can only update the `releaser` address and cannot directly call `releaseTo()` or `releaseToOrRevert()` unless the owner is also set as the `releaser`.

   This mismatch can lead to incorrect operational assumptions about which account is required to perform withdrawals.

   ```
   /**
    * @title
    * @author
    * @notice It receives the iKas native token provided as fee by the transaction.
    * it let the owner to withdraw the ikas native token
    */
   ```

   Update the NatSpec to reflect that the `releaser` is the authority that performs withdrawals, and clarify the owner's role as being responsible for configuring the `releaser`.

2. **Inconsistent NatSpec For Non Virtual Function**

   Related Asset(s): Fee.sol

   The NatSpec for `releasableAmount()` states that the function can be overridden to implement reserve ratios or vesting logic, but the function is not declared `virtual`.

   This makes the NatSpec misleading, because derived contracts cannot override `releasableAmount()` unless it is marked `virtual`.

   ```
   /// @notice Returns the amount available for release to attestation rewards
   /// @dev Override this to implement reserve ratios or vesting logic
   function releasableAmount() external view returns (uint256) {
       return address(this).balance;
   }
   ```

   Either declare `releasableAmount()` as `virtual` to match the NatSpec, or update the NatSpec to reflect that the function is not intended to be overridden.

3. **Floating Pragma**

   Related Asset(s): Multiple Solidity source files (e.g., SafeUints.sol)

   Several Solidity files in the repository declare floating pragma directives (e.g. `pragma solidity >0.8.0;`).

   This permits compilation with a broad range of compiler versions, which can introduce unintended behaviour changes due to compiler updates and reduces build reproducibility across environments.

```
// SPDX-License-Identifier: MIT
pragma solidity >0.8.0;

/**
 * @title SafeUints
 * @notice Util functions which throws if a uint256 can't fit into smaller uints.
 */
contract SafeUints {
    // @dev Checks if the given uint256 does not overflow uint96
    function _safe96(uint256 n) internal pure returns (uint96) {
        require(n < 2 ** 96, "VPools: Unsafe96");
        // forge-lint: disable-next-line(unsafe-typecast)
        return uint96(n);
    }
}
```

Consider pinning pragmas to the project's chosen compiler version (or a tightly scoped range) to improve reproducibility.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

Resolved in commit 3d54d72.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].