CYBER SECURITY SERVICES

Smart Contract Audit

Pandora Labs

Pandora ERC404 v2.0

February 15, 2024

Banshie ApS
Holmbladsgade 133
2300 Copenhagen S
Denmark
CVR: 37110108

Phone: (+45) 4245 2048
Email: info@banshie.com
Web: www.banshie.com

Table of contents

Executive Summary

Synopsis

Banshie was contracted by Pandora Labs to perform a of Pandora ERC404 v2.0. The purpose of the audit was to ensure that no security related issues or vulnerabilities were present in the source code that could be exploited by an attacker.

This report documents the audit process and the security issues identified. Additionally, it provides detailed description of the identified security issues along with recommendations and remediations.

Scope and Objectives

These are the main objectives that were in focus during the audit; please see Assessment Scope section for a more detailed description:

Source code	
Repository	https://github.com/Pandora-Labs-Org/erc404.git
Commit hash	987280e5feb3817dc6a7efd6d9e68fc49c297751

Fixes and Reaudit

A reaudit to verify remediation of the reported issues was done on the following source code:

Source code	
Repository	https://github.com/Pandora-Labs-Org/erc404.git
Commit hash	32061df3421bc8717035cf2b50a5571ca381ea48

Key Findings and Action Points

The audit identified:

- 1 high severity findings
- 4 medium severity findings
- 1 low severity findings
- 5 informational findings

The findings identified could lead to compromise of confidentiality, integrity, and authenticity.

The overall security posture is based on the observed findings and their criticality and is an estimate of level of security compared to similar code bases reviewed by *Banshie*. The smart contract is categorized with an overall HIGH security posture at the time of the audit.

For further details and a technical description of all identified findings and our recommendations, please refer to the Security Issues section and appendices.

Assessment Scope

Overview

The objective of the assessment was to perform a of Pandora Labs's Pandora ERC404 v2.0.

The smart contract implementation in scope has been downloaded from its Git repository and includes the following files:

```
├── contracts/
│   ├── examples/
│   │   └── ExampleERC404.sol
│   ├── extensions/
│   │   ├── ERC404MerkleClaim.sol
│   │   └── IERC404MerkleClaim.sol
│   ├── interfaces/
│   │   └── IERC404.sol
│   ├── lib/
│   │   ├── interfaces/
│   │   │   └── IERC165.sol
│   │   ├── DoubleEndedQueue.sol
│   │   └── ERC721Receiver.sol
│   └── ERC404.sol
├── hardhat.config.ts
├── package.json
├── pnpm-lock.yaml
├── README.md
└── tsconfig.json
```

The scope of the assessment included, but not limited to, the following high-level test cases:

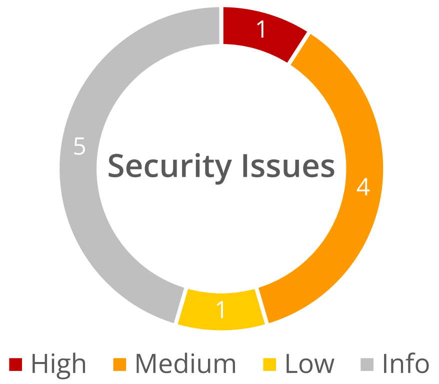
- Reentrancy
- Ownership Takeover
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Style guide violation
- Costly Loop
- ERC20 API violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level
- Deployment Consistency
- Repository Consistency
- Data Consistency
- Business Logics Review
- Functionality Checks
- Access Control & Authorization
- Escrow manipulation
- Token Supply manipulation
- Asset's integrity
- User Balances manipulation
- Kill-Switch Mechanism
- Operation Trails & Event Generation

Out of scope

Third party code and libraries are by default out of scope but will be reviewed as necessary to get a complete picture of the code base.

Security Issues

Security issues are findings that pose a threat and should be addressed according to their risk. Each issue found has been risk assessed according to our Risk Rating Methodology.



ID	Title	Severity
SI-1	Ownership corruption	High
SI-2	Transfer of ERC721 tokens does not respect whitelisting	Medium
SI-3	Self-transfer of ERC20 can produce ERC721	Medium
SI-4	ERC721 tokens can be minted to whitelisted accounts	Medium
SI-5	ERC721 owners should not be whitelisted	Medium
SI-6	ERC721Receivers receive invalid token ids	Low
SI-7	Use setters instead of togglers	Informational
SI-8	Front running approvals	Informational
SI-9	Conditional functionality for ERC20 and ERC721	Informational
SI-10	Use OpenZeppelin interfaces	Informational
SI-11	Floating pragma	Informational

SI-1 | Ownership corruption

Status	Severity
REMEDIATED	HIGH

Summary

In Yul assembly, the syntax for the shift left function `shl(x, y)` results in the value of `y` being shifted left by `x` bits. In Solidity syntax this is equivalent to `y << x`.

In the `ERC404` contract, the arguments for the Yul shift left function have mistakenly been swapped.

Impact

The affected code is used to set and retrieve indexes for ownership.

As the indexes are incorrectly calculated this will result in corruption in the ownership index causing both token loss and invalid ownerships.

Remediation

The easiest fix is the rewrite the left shift operations in pure Solidity which will improve readability.

For performance reasons, the Yul assembly may be kept and fixed by passing the arguments for the Yul assembly `shl()` function in the right order.

For `ERC404._getOwnedIndex()`:

```
function _getOwnedIndex(
  uint256 id_
) internal view virtual returns (uint256 ownedIndex_) {
  uint256 data = _ownedData[id_];

  assembly {
    ownedIndex_ := shl(160, data)
  }
}
```

For `ERC404._setOwnedIndex()`:

```
function _setOwnedIndex(uint256 id_, uint256 index_) internal virtual {
  uint256 data = _ownedData[id_];

  if (index_ > _BITMASK_OWNED_INDEX >> 160) {
    revert OwnedIndexOverflow();
  }

  assembly {
    data := add(
      and(data, _BITMASK_ADDRESS),
      and(shl(160, index_), _BITMASK_OWNED_INDEX)
    )
  }

  _ownedData[id_] = data;
```

```
}
```

Details

There are two invalid uses of the Yul assembly `shl` function `ERC404._getOwnedIndex()` and `ERC404._setOwnedIndex()`.

In `ERC404._setOwnedIndex()`:

erc404.sol

```
function _setOwnedIndex(uint256 id_, uint256 index_) internal virtual {
  uint256 data = _ownedData[id_];

  if (index_ > _BITMASK_OWNED_INDEX >> 160) {
    revert OwnedIndexOverflow();
  }

  assembly {
    data := add(
      and(data, _BITMASK_ADDRESS),
      and(shl(index_, 160), _BITMASK_OWNED_INDEX)
    )
  }

  _ownedData[id_] = data;
}
```

Rewriting the behavior of `_setOwnedIndex()` into pure Solidity will show the resulting behavior:

erc404.sol

```
function _setOwnedIndex(uint256 id_, uint256 index_) internal virtual {
  uint256 data = _ownedData[id_];

  if (index_ > _BITMASK_OWNED_INDEX >> 160) {
    revert OwnedIndexOverflow();
  }

  data = (data & _BITMASK_ADDRESS) + ((160 << index_) & _BITMASK_OWNED_INDEX);
  _ownedData[id_] = data;
}
```

Notice, that `index_ > _BITMASK_OWNED_INDEX >> 160`. As `_BITMASK_OWNED_INDEX = ((1 << 96) - 1) << 160` then `index >= (1 << 96)`. As we are working on an `uint256` and as `index > 256` then shifting `160` left by `index` will just result in zero as it "overflows".

Thus, the index will always be set to zero resulting in `data` being an `uint256` version of the owner's address.

In `ERC404._getOwnedIndex()`:

erc404.sol

```
function _getOwnedIndex(
  uint256 id_
) internal view virtual returns (uint256 ownedIndex_) {
  uint256 data = _ownedData[id_];

  assembly {
    ownedIndex_ := shl(data, 160)
  }
}
```

Again, rewriting the behavior of `_getOwnedIndex()` into pure Solidity shows the resulting behavior:

```
function _getOwnedIndex(
  uint256 id_
) internal view virtual returns (uint256 ownedIndex_) {
  uint256 data = _ownedData[id_];
  ownedIndex_ = 160 << data;
}
```

The 256 bits of `data` is a concatenation of:

- 96 bits for the owned index
- 160 bits for the owner address

Hence, the integer value of `data` is therefore either:

If `data` is unassigned then the numeric value is zero. Then 160 will not be shifted at all resulting in the `ownedIndex = 160`.

If an owner address is assigned to `data` then the numeric value of that address is likely a large number. Unless the most of the leading bits of the address are all zeroes, the numeric value of `data` will be large enough to overflow left shifting of 160 which will result in `ownedIndex = 0`.

In `_transferERC721()` the `_getOwnedIndex()` function is called to rearrange the owned index by moving the last token to the index of the token being transferred from the sender account:

erc404.sol

```
      uint256 updatedId = _owned[from_][_owned[from_].length - 1];

      if (updatedId != id_) {
        uint256 updatedIndex = _getOwnedIndex(id_);
        // update _owned for sender
        _owned[from_][updatedIndex] = updatedId;
        // update index for the moved id
        _setOwnedIndex(updatedId, updatedIndex);
      }

      // pop
      _owned[from_].pop();
```

So, if the sender's address was intentionally assigned then the result from `_getOwnedIndex()` will be zero. The token rearrangement will then replace the first token in the array with the last, keep the token being transferred, and loose the last token.

In other words, the ownership index will be corrupted...

References

- Solidity documentation - Yul

SI-2 | Transfer of ERC721 tokens does not respect whitelisting

Status	Severity
REMEDIATED	MEDIUM

Summary

When transferring ERC721 tokens, whitelisting is not respected.

Impact

Whitelisted accounts are allowed to send and received ERC721 tokens.

But when a whitelisted account sends ERC20 tokens, ERC721 are not transferred because of the whitelisting.

So, after receiving a specific ERC721 token, a whitelisted account can return the ERC20 tokens that followed along without losing the ERC721 token.

This can be repeated to multiply the ERC721 tokens owned by the whitelisted account.

The produced ERC721 tokens cannot be transferred as the whitelisted account does not own the sufficient amount of corresponding ERC20 tokens.

However, if a contract specializes the ERC404 contract and adds functionality based on the users' ERC721 holdings then this can be abused. For example in functionality such as voting with a vote per ERC721 token, these produced tokens can be used maliciously.

Remediation

Do not allow whitelisted accounts to send or receive in `ERC404._transferERC721()`:

```
function _transferERC721(
  address from_,
  address to_,
  uint256 id_
) internal virtual {
  if (from_ != address(0) && !whitelisted[from_]) {
    // ...
  }

  if (to_ != address(0) && !whitelisted[to_]) {
    // ...
  } else {
    delete _ownedData[id_];
  }
}
```

Details

Under normal circumstances, transferring ERC20 tokens will result in a corresponding amount of ERC721 tokens being transferred as well. But if the sender is whitelisted, then the ERC721 tokens are either taken from the bank or minted.

This behavior is handled by `ERC404._transferERC20WithERC721()`:

erc404.sol

```
    } else if (isFromWhitelisted) {
      // Case 2) The sender is whitelisted, but the recipient is not. Contract should not attempt
      //         to transfer ERC-721s from the sender, but the recipient should receive ERC-721s
      //         from the bank/minted for any whole number increase in their balance.
      // Only cares about whole number increments.
      uint256 tokensToRetrieveOrMint = (balanceOf[to_] / units) -
        (erc20BalanceOfReceiverBefore / units);
      for (uint256 i = 0; i < tokensToRetrieveOrMint; i++) {
        _retrieveOrMintERC721(to_);
      }
```

However, if an ERC721 token is transferred by its id in `transferFrom()`, then `_transferERC721()` transfers the ERC721 token without validating if the sender or the receiver is whitelisted:

erc404.sol

```
    if (valueOrId_ <= _minted) {
      // Intention is to transfer as ERC-721 token (id).
      uint256 id = valueOrId_;

      if (from_ != _getOwnerOf(id)) {
        revert Unauthorized();
      }

      // Check that the operator is either the sender or approved for the transfer.
      if (
        msg.sender != from_ &&
        !isApprovedForAll[from_][msg.sender] &&
        msg.sender != getApproved[id]
      ) {
        revert Unauthorized();
      }

      // Transfer 1 * units ERC-20 and 1 ERC-721 token.
      _transferERC20(from_, to_, units);
      _transferERC721(from_, to_, id);
```

As a result, a user in control of a whitelisted account `W` and another account `A` can repeat the following sequence to get ownership of an infinite amount of ERC721 tokens:

1. Let `A` call `transferFrom(A, W, tokenId)` to transfer a specific ERC721 token from account `A` to account `W`.
2. Let `W` call `transferFrom(W, A, units)` or `transfer(A, units)` to returns the ERC20 tokens to `A` along with a fresh ERC721 token.

SI-3 | Self-transfer of ERC20 can produce ERC721

Status	Severity
REMEDIATED	MEDIUM

Summary

When transferring ERC20 tokens from and to the same account, the ERC721 token balances are adjusted to match a fraction of the ERC20 token balance.

However, the calculation of the new ERC721 token balance is based on the assumption that the ERC20 balances actually have changed which is not the case in a self-transfer.

Impact

Miscalculations in the balancing of ERC721 tokens can be used for either burn or mint of ERC721 tokens.

The produced ERC721 tokens cannot be transferred as the whitelisted account does not own the sufficient amount of corresponding ERC20 tokens.

However, if a contract specializes the ERC404 contract and adds functionality based on the users' ERC721 holdings then this can be abused.

Remediation

Use actual balance values for calculating if the fractional amounts of ERC721 tokens in `ERC404._transferERC20WithERC721`:

For burning:

```
uint256 erc20BalanceOfSenderAfter = erc20BalanceOf(from_);
if ((erc20BalanceOfSenderBefore / units) -
 ↪  (erc20BalanceOfSenderAfter / units)) > nftsToTransfer) {
  _withdrawAndStoreERC721(from_);
}
```

For minting:

```
uint256 erc20BalanceOfReceiverAfter = erc20BalanceOf(to_);
if ((erc20BalanceOfReceiverAfter / units) -
 ↪  (erc20BalanceOfReceiverBefore / units)) > nftsToTransfer) {
  _retrieveOrMintERC721(to_);
}
```

Details

In `ERC404._transferERC20WithERC721()` the ERC20 balances of the sender and receiver accounts are initially cached and then ERC20 tokens are transferred:

```
erc404.sol
  function _transferERC20WithERC721(
    address from_,
    address to_,
    uint256 value_
  ) internal virtual returns (bool) {
    uint256 erc20BalanceOfSenderBefore = erc20BalanceOf(from_);
    uint256 erc20BalanceOfReceiverBefore = erc20BalanceOf(to_);

    _transferERC20(from_, to_, value_);
```

In case both sender and receiver are not whitelisted, ERC721 tokens are first transferred:

```
erc404.sol
      uint256 nftsToTransfer = value_ / units;
      for (uint256 i = 0; i < nftsToTransfer; i++) {
        // Pop from sender's ERC-721 stack and transfer them (LIFO)
        uint256 indexOfLastToken = _owned[from_].length - 1;
        uint256 tokenId = _owned[from_][indexOfLastToken];
        _transferERC721(from_, to_, tokenId);
      }
```

The change to the sender's ERC20 balance may require that an additional ERC721 token is removed:

```
erc404.sol
      uint256 fractionalAmount = value_ % units;

      if (
        (erc20BalanceOfSenderBefore - fractionalAmount) / units <
        (erc20BalanceOfSenderBefore / units)
      ) {
        _withdrawAndStoreERC721(from_);
      }
```

Likewise, the change in the receiver's ERC20 balance may require that an additional ERC721 token is added:

```
erc404.sol
      if (
        (erc20BalanceOfReceiverBefore + fractionalAmount) / units >
        (erc20BalanceOfReceiverBefore / units)
      ) {
        _retrieveOrMintERC721(to_);
      }
```

However, these calculations assume that the ERC20 balances are actually changed.

This is not the case during a self-transfer where the sender and the receiver is the same account. Then the call to `_transferERC20()` will not change the balances by `fractionalAmount`. Instead, the balances (or balance) remain the same.

This will allow a user to produce an additional ERC721 token as follows:

1. The account `A` used must have an ERC20 balance where:
   1. `erc20BalanceOf(A) % units > units / 2`.
   2. `erc20BalanceOf(A) > units` should that `erc721BalanceOf(A) > 1`.
2. Call `transfer(A, units / 2)`.

This will result in `_withdrawAndStoreERC721(from_)` is not called and `_retrieveOrMintERC721(from_)`. Thus, an additional ERC721 token has been added to account `A` without losing any ERC20 tokens.

SI-4 | ERC721 tokens can be minted to whitelisted accounts

Status	Severity
REMEDIATED	MEDIUM

Summary

Whitelisted accounts can receive ERC721 tokens through minting.

Impact

Letting whitelisted accounts own ERC721 tokens may introduce issues with the balance between owned ERC20 and ERC721 tokens.

Whitelisted accounts can transfer ERC20 tokens without losing the corresponding amount of ERC721 tokens. A whitelisted account can therefore end up with too few ERC20 tokens to cover the value of owned ERC721 tokens. If such an account is removed from the whitelist, situations will occur where ERC721 tokens cannot be transferred because of the lack of corresponding ERC20 tokens.

Even though the minting functionality is internal, other contracts that specialize the `ERC404` contract may expose the functionality in ways open for abuse or errors.

Remediation

Revert calls to `ERC404._mintERC20()` if `to_` is whitelisted:

```
function _mintERC20(
  address to_,
  uint256 value_,
  bool mintCorrespondingERC721s_
) internal virtual {
  /// You cannot mint ERC721 tokens to whitelisted addresses
  if (mintCorrespondingERC721s_ && whitelist[to_]) {
    revert InvalidRecipient();
  }
```

Details

The `ERC404._mintERC20()` function will transfer ERC721 tokens to any account if `mintCorrespondingERC721s_ = true`:

erc404.sol

```
function _mintERC20(
  address to_,
  uint256 value_,
  bool mintCorrespondingERC721s_
) internal virtual {
  /// You cannot mint to the zero address (you can't mint and
   ↪  immediately burn in the same transfer).
  if (to_ == address(0)) {
    revert InvalidRecipient();
  }

  _transferERC20(address(0), to_, value_);

 // If mintCorrespondingERC721s_ is true, mint the corresponding ERC721s.
  if (mintCorrespondingERC721s_) {
    uint256 nftsToRetrieveOrMint = value_ / units;
    for (uint256 i = 0; i < nftsToRetrieveOrMint; i++) {
      _retrieveOrMintERC721(to_);
    }
  }
}
```

The `_mintERC20()` function is an internal function. But in the `ExampleERC404.airdropMint()` contract the `_mintERC20()` function is called:

examples/exampleerc404.sol

```
function airdropMint(
  bytes32[] memory proof_,
  uint256 value_
) public override whenAirdropIsOpen {
  super.airdropMint(proof_, value_);
  _mintERC20(msg.sender, value_, true);
}
```

Even though whitelisted accounts can receive ERC721 tokens it is not intended...

SI-5 | ERC721 owners should not be whitelisted

Status	Severity
REMEDIATED	MEDIUM

Summary

If an owner of ERC721 tokens is whitelisted, ERC20 tokens can be transferred without losing the ERC721 tokens.

Impact

Users with authorization to change the whitelist can repeatedly duplicate their owned amount of ERC721 tokens.

The produced ERC721 tokens cannot be transferred as the whitelisted account does not own the sufficient amount of corresponding ERC20 tokens.

Even though the functionality for setting whitelist is internal, other contracts that specialize the `ERC404` contract may expose the functionality in ways open for abuse or errors.

Remediation

Only allow accounts to be whitelisted if they do not own any ERC721 tokens:

```
function _setWhitelist(address target_, bool state_) internal virtual {
  if (state_) {
    if (erc721BalanceOf(target_) > 0) {
      revert CannotAddToWhitelist();
    }
  } else {
    if (erc20BalanceOf(target_) >= units) {
      revert CannotRemoveFromWhitelist();
    }
  }
  whitelist[target_] = state_;
}
```

Or even simpler, never allow accounts to be removed from the whitelist.

Details

All `ERC404` whitelist functionality is handled by the `_transferERC20WithERC721()` function.

In the case that the sender `from_` is whitelisted, `_retrieveOrMintERC721(to_)` is called instead of transferring the ERC721 tokens from the sender:

erc404.sol

```
    } else if (isFromWhitelisted) {
      // Case 2) The sender is whitelisted, but the recipient is not. Contract should not attempt
      //         to transfer ERC-721s from the sender, but the recipient should receive ERC-721s
      //         from the bank/minted for any whole number increase in their balance.
      // Only cares about whole number increments.
      uint256 tokensToRetrieveOrMint = (balanceOf[to_] / units) -
        (erc20BalanceOfReceiverBefore / units);
      for (uint256 i = 0; i < tokensToRetrieveOrMint; i++) {
        _retrieveOrMintERC721(to_);
      }
```

The `ERC404._setWhitelist()` function is used to add and remove users to and from the whitelist:

erc404.sol

```
  function _setWhitelist(address target_, bool state_) internal virtual {
    // If the target has at least 1 full ERC-20 token, they should not be removed from the whitelist
    // because if they were and then they attempted to transfer, it would revert as they would not
    // necessarily have ehough ERC-721s to bank.
    if (erc20BalanceOf(target_) >= units && !state_) {
      revert CannotRemoveFromWhitelist();
    }
    whitelist[target_] = state_;
  }
```

A check prevents owners of ERC20 tokens equals to at least one ERC721 token (specified by the exchange rate in `units`) to be removed from the whitelist. However, there are no restrictions on adding users to the whitelist.

The issue occurs, if a user owning at least one ERC721 token is whitelisted. When a whitelisted user transfers an amount higher than `units` ERC20 tokens to another account, the `_transferERC20WithERC721()` function will then let the sender keep the ERC721 tokens because of the whitelisting.

So, if a user is authorized to make changes to the whitelist, that user can duplicate his ERC721 tokens as follows:

1. Two accounts are needed:
   1. Account `A` owning `m` ERC721 tokens and `m * units` ERC20 tokens.
   2. Account `B` which initially does not need any tokens.
2. Call `ERC404._setWhitelist(A, true)`. For example in the `ExampleERC404` contract, the contract owner is allowed to call `setWhitelist(A, true)`.
3. Let `A` call `ERC404.transfer(B, m * units)` to transfers `m * units` ERC20 tokens from `A` to `B`. Notice that `B` will receive ERC721 tokens as well, but `A` gets to keep the `m` ERC721 tokens because of the whitelisting.
4. Call `ERC404._setWhitelist(A, false)` to remove `A` from the whitelist. This is allowed since `A` no longer owns any ERC20 tokens.

5. Let `B` call `ERC404.transfer(A, m * units)` to transfers the `m * units` ERC20 tokens back to `A`.
6. `A` now owns `2 * m` ERC721 tokens but only `m * units` ERC20 tokens.

SI-6 | ERC721Receivers receive invalid token ids

Status	Severity
REMEDIATED	LOW

Summary

The callback performed by the `ERC404.safeTransferFrom()` function will contain invalid token ids if an amount of ERC20 tokens is passed instead of an ERC721 token id.

Impact

If contract that implements the `ERC721Receiver` interface and calls the `safeTransferFrom()` function receives an invalid token id in the callback it may break the logic of that contract.

Remediation

Revert calls to the two `ERC404.safeTransferFrom()` functions if the passed ERC721 token id is invalid:

```
function safeTransferFrom(
  address from_,
  address to_,
  uint256 id_
) public virtual {
  if (valueOrId_ > _minted) {
      revert InvalidId();
  }
  transferFrom(from_, to_, id_);
```

Details

The `ERC404.safeTransferFrom(address, address, uint256)` function simply calls `transferFrom()` and then does a callback if the caller is a contract implementing the `ERC721Receiver` interface:

erc404.sol

```
function safeTransferFrom(
  address from_,
  address to_,
  uint256 id_
) public virtual {
  transferFrom(from_, to_, id_);

  if (
    to_.code.length != 0 &&
    ERC721Receiver(to_).onERC721Received(msg.sender, from_, id_, "") !=
    ERC721Receiver.onERC721Received.selector
  ) {
    revert UnsafeRecipient();
  }
}
```

And the `ERC404.safeTransferFrom(address, address, uint256, bytes calldata)` function has identical behavior:

erc404.sol

```
function safeTransferFrom(
  address from_,
  address to_,
  uint256 id_,
  bytes calldata data_
) public virtual {
  transferFrom(from_, to_, id_);

  if (
    to_.code.length != 0 &&
   ERC721Receiver(to_).onERC721Received(msg.sender, from_, id_, data_) !=
    ERC721Receiver.onERC721Received.selector
  ) {
    revert UnsafeRecipient();
  }
}
```

But the `transferFrom()` function will interpret the `id_` as either an ERC721 token id or an amount of ERC20 tokens
to transfer:

erc404.sol

```
  if (valueOrId_ <= _minted) {
    // Intention is to transfer as ERC-721 token (id).
    uint256 id = valueOrId_;
```

If an amount of ERC20 tokens which is higher than `_minted` is passed then `_transferERC20WithERC721()` will be called to transfer that amount of ERC20 tokens and the corresponding amount of ERC721 tokens:

erc404.sol

```
  } else {
    // Intention is to transfer as ERC-20 token (value).
    uint256 value = valueOrId_;
    uint256 allowed = allowance[from_][msg.sender];

    // Check that the operator has sufficient allowance.
    if (allowed != type(uint256).max) {
      allowance[from_][msg.sender] = allowed - value;
    }

    // Transferring ERC-20s directly requires the _transfer function.
    _transferERC20WithERC721(from_, to_, value);
  }
```

The callback sent via `ERC721Receiver(to_).onERC721Received(msg.sender, from_, id_, data_)` will then contain an `id_` which is not a valid ERC721 token id.

SI-7 | Use setters instead of togglers

Status	Severity
OPEN	INFORMATIONAL

Summary

Opening and closing airdrops in the `ERC404MerkleClaim` contract is controlled by a toggle function instead of a setter function.

Impact

A caller may open an airdrop by accident.

Remediation

Implement the `_toggleAirdropIsOpen()` function as a setter function:

```
function _setAirdropState(bool isOpen_) internal {
  airdropIsOpen = isOpen_;
}
```

Details

Implementing state changes with toggle functions makes it harder for the caller to predict the outcome.

Toggle functions require the caller to perform additional checks to ensure whether the specific feature is enabled or disabled.

Furthermore, accidentally calling a toggle function twice will result in the state not being changed.

On the contrary setter functions take an argument that explicitly defines whether the specific feature should be enabled or disabled. Hence, the caller is never in doubt if a feature is enabled or disabled by a setter function.

The `ERC404MerkleClaim._toggleAirdropIsOpen()` implements a toggle function:

extensions/erc404merkleclaim.sol

```
function _toggleAirdropIsOpen() internal {
  airdropIsOpen = !airdropIsOpen;
}
```

SI-8 | Front running approvals

Status	Severity
OPEN	INFORMATIONAL

Summary

A known design issue of the ERC20 approve function is that it opens for front running attacks.

Impact

If an owner needs to increase the allowance for a spender, the spender can front run the owner's approval and "double spend" the allowance.

Remediation

Do not to repeat the known design issue of the ERC20 approve function.

Instead, implement the approval functionality as an incrementing function and a decrementing function similar to OpenZeppelin's SafeERC20.

Details

Consider an owner who has already approved a spender for an amount of A tokens. Now, the owner needs to increase the approved amount to B and calls `ERC20.approve(spender, B)`.

Notice, that the amount B is the new total amount that the spender will be approved to use.

Remember, that before the new approval the spender was approved to spend A tokens. This allows the spender to front run the new approval and spend all the initially approved A tokens. After this, the spender is approved for B tokens which can also be spent.

In total and unintended by the owner, the spender gets to spend $A + B$ tokens and not just B.

The `ERC404.approve()` function behaves identical to `ERC20.approve()` when `valueOrId_` is higher than the maximum ERC721 token id:

erc404.sol

```
function approve(
  address spender_,
  uint256 valueOrId_
) public virtual returns (bool) {
  // The ERC-721 tokens are 1-indexed, so 0 is not a valid id and
   ↪ indicates that
  // operator is attempting to set the ERC-20 allowance to 0.
  if (valueOrId_ <= _minted && valueOrId_ > 0) {
    // Intention is to approve as ERC-721 token (id).
```

The approved amount is set without considering the existing approved amount:

erc404.sol

```
    } else {
      // Prevent granting 0x0 an ERC-20 allowance.
      if (spender_ == address(0)) {
        revert InvalidSpender();
      }

      // Intention is to approve as ERC-20 token (value).
      uint256 value = valueOrId_;
      allowance[msg.sender][spender_] = value;

      emit ERC20Approval(msg.sender, spender_, value);
    }

    return true;
  }
```

If the spender is approved an amount before the owners call the `approve()` then the spender can front run the owner's call to `approve()` and spend the initially approved amount as well.

References

- SmartDec - ERC20 approve issue in simple words
- GitHub - OpenZeppelin Contracts - SafeERC20

SI-9 | Conditional functionality for ERC20 and ERC721

Status	Severity
REMEDIATED	INFORMATIONAL

Summary

The `transferFrom()`, `safeTransferFrom()`, and `approve()` functions in the `ERC404` contract implement conditional behavior depending on whether the input is a valid ERC721 token id or not.

Impact

Conditional behavior introduces the risk of unintentionally triggering ERC721 functionality instead of ERC20.

Although not currently possible, a caller may want to approve or transfer a small amount of ERC20 tokens. Trying to do this will result in either:

1. Most likely, the transaction will fail as the passed amount is not the id of an ERC721 token that the caller has authority over.
2. Unlikely, the caller has authority of an ERC721 token with that id. If this is the case the caller will lose this ERC721.

Remediation

The functions for `balanceOf` and `totalSupply` have already been split into function pairs for handling ERC20 and ERC721 behavior respectively. These function are named `erc20BalanceOf`, `erc721BalanceOf`, `erc20TotalSupply`, and `erc721TotalSupply`. The naming and splitting of the functions makes the expected behavior more obvious.

Consider implementing such function pairs for ERC20 and ERC721 specific behavior for the following `ERC404` functions:

- `approve(address, uint256)`
- `transferFrom(address, address, uint256)`
- `safeTransferFrom(address, address, uint256)`
- `safeTransferFrom(address, address, uint256, bytes calldata)`

Implementing these specific functions for handling ERC20 and ERC721 tokens will also allow approvals and transfers of even small amounts of ERC20 tokens. Currently, this is not possible as small amounts of ERC20 tokens collides with valid ERC721 token ids.

Details

The `ERC404.approve()` function implements conditional behavior depending on the value of the input `valueOrId_`. If `valueOrId_ <= _minted` and is not zero then `valueOrId_` is a considered as an ERC721 token id which the spender be will be approved for. Otherwise, `valueOrId_` is considered as an amount of ERC20 tokens which the spender will be approved for:

erc404.sol

```
function approve(
  address spender_,
  uint256 valueOrId_
) public virtual returns (bool) {
  // The ERC-721 tokens are 1-indexed, so 0 is not a valid id and indicates that
  // operator is attempting to set the ERC-20 allowance to 0.
  if (valueOrId_ <= _minted && valueOrId_ > 0) {
    // Intention is to approve as ERC-721 token (id).
```

Likewise, The `ERC404.transferFrom()` function implements conditional behavior. If `valueOrId_ <= _minted` then `valueOrId_` is a considered as an ERC721 token id to be transferred. Otherwise, `valueOrId_` is considered as an amount of ERC20 tokens to be transferred:

erc404.sol

```
function transferFrom(
  address from_,
  address to_,
  uint256 valueOrId_
) public virtual returns (bool) {
  // Prevent transferring tokens from 0x0.
  if (from_ == address(0)) {
    revert InvalidSender();
  }

  // Prevent burning tokens to 0x0.
  if (to_ == address(0)) {
    revert InvalidRecipient();
  }

  if (valueOrId_ <= _minted) {
    // Intention is to transfer as ERC-721 token (id).
```

Furthermore, `transferFrom()` is called from the two public `safeTransferFrom()` functions which then inherits the conditional behavior:

erc404.sol

```
function safeTransferFrom(
  address from_,
  address to_,
  uint256 id_
) public virtual {
  transferFrom(from_, to_, id_);
```

and

erc404.sol

```
  function safeTransferFrom(
    address from_,
    address to_,
    uint256 id_,
    bytes calldata data_
  ) public virtual {
    transferFrom(from_, to_, id_);
```

Although this conditional behavior does work, it introduces an unnecessary complexity for clients or third party developers who are calling the contract directly.

SI-10 | Use OpenZeppelin interfaces

Status	Severity
REMEDIATED	INFORMATIONAL

Summary

The project defines interfaces and contract which are already included.

Impact

There is no need to duplicate existing code.

Remediation

To get rid of the duplicated code:

1. In `ERC404.sol` and `interfaces/IERC404.sol`
   replace the imports of `lib/interfaces/IERC165.sol`
   with `@openzeppelin/contracts/interfaces/IERC165.sol`
2. In `ERC404.sol`
   replace the import of `lib/ERC721Receiver.sol`
   with `@openzeppelin/contracts/interfaces/IERC721Receiver.sol`.
3. In `ERC404.sol` uses of `ERC721Receiver` must be changed to `IERC721Receiver`.
4. Delete `lib/interfaces/IERC165.sol`.
5. Delete `lib/ERC721Receiver.sol`.

Details

The provided `IERC165` interface is identical to `IERC165` interface from Open Zeppelin:

lib/interfaces/ierc165.sol

```
interface IERC165 {
  /**
   * @dev Returns true if this contract implements the interface defined by
   * `interfaceId`. See the corresponding
   * https://eips.ethereum.org/EIPS/eip-165#how-interfaces-are-identified[ERC section]
   * to learn more about how these ids are created.
   *
   * This function call must use less than 30 000 gas.
   */
  function supportsInterface(bytes4 interfaceId) external view returns (bool);
}
```

Besides whitespaces and comments, Open Zeppelin's `IERC165` interface is the same:

@openzeppelin/contracts/utils/introspection/ierc165.sol

```
interface IERC165 {
    /**
    * @dev Returns true if this contract implements the interface defined by
     * `interfaceId`. See the corresponding
     * https://eips.ethereum.org/EIPS/eip-165#how-interfaces-are-
     ↪ identified[EIP section]
     * to learn more about how these ids are created.
     *
     * This function call must use less than 30 000 gas.
     */
    function supportsInterface(bytes4 interfaceId) external view returns
     ↪ (bool);
}
```

So, the provided `IERC165` interface can be replaced with OpenZeppelin's.

Also, the abstract contract `ERC721Receiver` is only used for external calls by the `safeTransferFrom()` functions:

erc404.sol

```
      ERC721Receiver(to_).onERC721Received(msg.sender, from_, id_, "") !=
      ERC721Receiver.onERC721Received.selector
```

For doing external calls an interface would be sufficient.

Looking at the signature of the `ERC721Receiver`, it is identical to Open Zeppelin's `IERC721Receiver` interface:

lib/erc721receiver.sol

```
abstract contract ERC721Receiver {
  function onERC721Received(
    address,
    address,
    uint256,
    bytes calldata
  ) external virtual returns (bytes4) {
    return ERC721Receiver.onERC721Received.selector;
  }
}
```

Open Zeppelin's `IERC721Receiver` interface:

@openzeppelin/contracts/token/erc721/ierc721receiver.sol

```
interface IERC721Receiver {
    /**
     * @dev Whenever an {IERC721} `tokenId` token is transferred to this
    ↪ contract via {IERC721-safeTransferFrom}
     * by `operator` from `from`, this function is called.
     *
     * It must return its Solidity selector to confirm the token transfer.
     * If any other value is returned or the interface is not implemented
    ↪ by the recipient, the transfer will be
     * reverted.
     *
     * The selector can be obtained in Solidity with
    ↪ `IERC721Receiver.onERC721Received.selector`.
     */
    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external returns (bytes4);
}
```

So, the provided abstract contract `ERC721Receiver` can be replaced with OpenZeppelin's `IERC721Receiver` interface.

Furthermore, OpenZeppelin's contracts are already included as a dependency in the project:

package.json

```
  "dependencies": {
    "@openzeppelin/contracts": "^5.0.1",
    "solhint": "^4.1.1"
  },
```

So, there is no reason to duplicate already included code.

References

- OpenZeppelin - ERC165
- OpenZeppelin - ERC721Holder
- OpenZeppelin - IERC721Receiver

SI-11 | Floating pragma

Status	Severity
OPEN	INFORMATIONAL

Summary

Floating pragma versions have been used to define the version of the Solidity compiler to use.

Impact

Specifying the compiler version as a floating range will allow any compiler within the range to be used. This also includes nightly builds of the Solidity compiler.

If an unstable compiler is used during deployment, the resulting binaries may be unstable and in worst case buggy or vulnerable.

Remediation

Specify exact pragma versions in the code. For example:

```
pragma solidity 0.8.18;
```

Details

The contracts define solidity versions using floating pragmas:

erc404.sol

```
pragma solidity ^0.8.20;
```

examples/exampleerc404.sol

```
pragma solidity ^0.8.0;
```

extensions/erc404merkleclaim.sol

```
pragma solidity ^0.8.20;
```

extensions/ierc404merkleclaim.sol

```
pragma solidity ^0.8.20;
```

interfaces/ierc404.sol

```
pragma solidity ^0.8.20;
```

lib/doubleendedqueue.sol

```
pragma solidity ^0.8.20;
```

lib/erc721receiver.sol

```
pragma solidity ^0.8.20;
```

lib/interfaces/ierc165.sol#

```
pragma solidity ^0.8.20;
```

References

- Ethereum Smart Contract Best Practices - Locking Pragmas

Gas optimizations

The following sections contains suggestions on improvements for limiting gas expenses.

ID	Title	Improvement
GO-1	Use compiler optimization	Small
GO-2	Unnecessary checked arithmetic in loop	Small

GO-1 | Use compiler optimization

Summary

The Solidity compiler has a built-in optimizer that can be used to produce a more gas efficient binary.

Effect

In general the effect depends on how optimized the code has already been written.

To evaluate the effect of the optimization, gas reports have been generated by running `hardhat test` on the provided test suite with and without the optimization. The following tables sum up the averages from the gas reports and includes the effect on gas reduction.

The first table shows that the optimizer can reduce the gas cost between 0.9 % and 3.5 % when calling functions:

Contract	Method	Non-optimized	Optimized	Effect
ExampleERC404	setApprovalForAll	40335	39899	1.1 %
ExampleERC404	setWhitelist	38678	38234	1.1 %
ExampleERC404	transfer	710853	698468	1.7 %
MinimalERC404	approve	47659	47018	1.3 %
MinimalERC404	mintERC20	1271763	1251528	1.6 %
MinimalERC404	permit	79063	76276	3.5 %
MinimalERC404	setApprovalForAll	46640	46211	0.9 %
MinimalERC404	transfer	119814	116822	2.5 %
MinimalERC404	transferFrom	106601	105139	1.4 %

The second table shows that the optimizer can reduce the size of the compiled binaries and save between 41.8 % and 43.1 % of the deployment costs:

Deployments	Non-optimized	Optimized	Effect
ExampleERC404	4251873	2474051	41.8 %
MinimalERC404	3762934	2140558	43.1 %

Optimization

Enable the Solidity compiler's optimizer in the configuration:

hardhat.config.ts

```
import { HardhatUserConfig } from "hardhat/config"
import "@nomicfoundation/hardhat-toolbox"
import "hardhat-gas-reporter"

const config: HardhatUserConfig = {
  solidity: {
    compilers: [{
      version: "0.4.18"
       settings: {
         optimizer: {
           enabled: true,
           runs: 1000,
         }
       }
    }]
  },
  gasReporter: {
    currency: "USD",
    gasPrice: 21,
    enabled: true,
  },
}

export default config
```

References

- Solidity - The Optimizer
- Hardhat - Compiling your contracts - Configuring the compiler

GO-2 | Unnecessary checked arithmetic in loop

Summary

A `for` loop is mostly used to iterate through a range of integers. In most cases where the upper range is already known, there is no risk of overflowing. Hence, gas can be saved by skipping unnecessary overflow checks during counter incrementation.

Effect

In general the effect depends on how many times a `for` loop iterates and thereby saves gas on not checking for overflows.

To evaluate the effect of the optimization, gas reports have been generated by running `hardhat test` on a custom test with and without the optimization.

To test the effect in both `ERC404._transferERC20WithERC721()` and `_mintERC20()` the custom test does the following:

- Mint 500 * units ERC20 tokens and corresponding 500 ERC721 tokens to trigger `_mintERC20()`
- Transfer 500 * units ERC20 tokens and corresponding 500 ERC721 tokens to trigger `_transferERC20WithERC721()`.

The following table sums up the averages from the gas reports and includes the effect on gas reduction:

Contract	Method	Non-optimized	Optimized	Effect
MinimalERC404	mintERC20	24707666	24645166	0.3 %
MinimalERC404	transfer	17690937	17628437	0.4 %

The table above shows that skipping overflow checks for counter incrementation in `for` loop reduces gas cost by 0.3 % for minting and transfers.

Optimization

Rewrite simple `for` loops to use unchecked counter incrementation at the end of the loop:

```
for (uint256 i = 0; i < n;) {
  // Do something in the loop
  unchecked { ++i; }
}
```

Also notice that prefix increments `++i` are cheaper that postfix increment `i++`.

This optimization can be applied to the following code:

erc404.sol

```
      for (uint256 i = 0; i < tokensToRetrieveOrMint; i++) {
```

erc404.sol

```
      for (uint256 i = 0; i < tokensToWithdrawAndStore; i++) {
```

erc404.sol

```
        for (uint256 i = 0; i < nftsToTransfer; i++) {
```

erc404.sol

```
        for (uint256 i = 0; i < nftsToRetrieveOrMint; i++) {
```

The implicated functions are `ERC404._transferERC20WithERC721()` and `_mintERC20()`

References

- Semgrep - Unnecessary checked arithmetic in loop

Appendix

Methodology

Banshie performs Smart Contract Audits through a combination of manual and automatic analysis for issues and potential vulnerabilities. During the analysis the application architecture, specification and functionality will be audited, if possible, to identify potential security issues.

Smart Contract Audits are performed using a combination of methods to achieve the best combination of coverage and efficiency.

- Static Application Security Testing (SAST) tools
- Property, Differential and Coverage fuzzing
- Compilation checks
- Manual review

All tools and checks are customized for each code base.

Code Safety

Smart Contract Audits are assessed with the following high-level focus:

- General code safety and susceptibility to known issues
- Poor coding practices and unsafe behavior
- Leakage of secrets or other sensitive data through memory mismanagement
- Susceptibility to misuse and system errors
- Error management and logging

This list is a general list and not comprehensive, meant only to give an understanding of the issues targeted in a Smart Contract Audit.

Risk Rating Methodology

Risk	Definition
Critical	**Vulnerabilities that will lead to loss of protected assets** • This is a vulnerability that would lead to immediate loss of protected assets • The complexity to exploit is low • The probability of exploit is high
High	**Vulnerabilities that can lead to loss of protected assets** • All discrepancies found where there is a security claim made in the documentation that cannot be found in the code • All mismatches from the stated and actual functionality • Unprotected key material • Weak encryption of keys • Badly generated key materials • Tx signatures not verified • Spending of funds through logic errors • Calculation errors and over-/underflows
Medium	**Vulnerabilities that disrupt the uptime of the system or can lead to other problems** • Insecure calls to third party libraries • Use of untested or nonstandard or non-peer-reviewed crypto functions • Program crashes leaves core dumps or write sensitive data to log files
Low	**Problems that have a security impact but does not directly impact the protected assets** • Overly complex functions • Unchecked return values from 3rd party libraries that could alter the execution flow
Informational	**General recommendations**