CD SECURITY

AUDIT REPORT

Groge
April 2024

Introduction

A time-boxed security review of the **Groge** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Conducted by: `immeas`, `hals`

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Groge**

Groge is a new ERC420 token which acts as a hybrid between an ERC20 and an ERC1155 token. A user can interact with it either as an ERC20 token or ERC1155 as it has the calls for both.

Groge keeps a representation of a users balance both as ERC20 and ERC1155, where the ERC1155 balance simply is the ERC20 balance without the decimal subdivision. Which for Groge is 18.

In addition to this, Groge also keeps an allow list which bypasses the ERC1155 state for certain administrative accounts.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - `2f3c40c1c52a65c195855472182d5d11987cde2e`**

Scope

The following smart contracts were in scope of the audit:

- `Groge.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 0 issues
- Medium: 4 issues
- Low: 1 issues
- Informational: 7 issues

Findings Summary

ID	Title	Severity
[M-01]	A user can inflate their NFT balance	Medium
[M-02]	`ERC420.setAllowList()` can replicate the previous target state	Medium
[M-03]	`ERC420._safeBatchTransferFrom()` doesn't verify allowList	Medium
[M-04]	Approve race condition in `ERC420.approve()`	Medium
[L-01]	No call to ERC1155 transfer hook	Low
[I-01]	Lack of `memory-safe` annotation on memory safe assembly	Informational
[I-02]	Events emitted when transferring ERC1155 not following EIP-1155	Informational
[I-03]	No event emitted when adding or removing account in allow list	Informational
[I-04]	Unnecessary `msg.sender != address(0)` check	Informational
[I-05]	`ERC420.setAllowList()` can be optimized	Informational
[I-06]	`ERC420.balanceOf()` doesn't check if the id is valid	Informational
[I-07]	Updating baseURI renders previously minted NFT URIs invalid	Informational

Detailed Findings

[M-01] A user can inflate their NFT balance

Severity

Likelihood: High

Impact: Low

Description

When a user is using the ERC20 call of transferring tokens (`transfer` or `transferFrom`) the ERC20 balance changes are synced to ERC1155 balance changes in `_transfer`:

```
File: Groge.sol

        uint256 balanceBeforeSender = _balances[from];
        uint256 balanceBeforeReceiver = _balances[to];

        if (!isInAllowlist(from)) {
            uint256 tokens_to_burn = (balanceBeforeSender / 10 ** _decimals) -
                ((balanceBeforeSender - value) / 10 ** _decimals);
            _nft_burn(from, 0, tokens_to_burn);
        }

        if (!isInAllowlist(to)) {
            uint256 tokens_to_mint = ((balanceBeforeReceiver + value) /
                10 ** _decimals) - (balanceBeforeReceiver / 10 ** _decimals);
            _nft_mint(to, 0, tokens_to_mint);
        }

        _update(from, to, value);
```

The issue above is that the old state of `balanceBeforeReceiver` is used to determine how many nfts should be minted.

If a user does a transfer to themselves that is low enough to not decrease their `tokens_to_burn` but high enough to increase their `tokens_to_mint` they will be able to inflate their nft balance:

Alice has `1.9e18` ERC20 tokens. She transfers `0.2e18` tokens to herself so both `balanceBeforeSender/Receiver` are `1.9e18`.

`tokens_to_burn` will not decrease as `1.9e18/1e18 - 1.7e18/1e18 => 1 - 1 = 0`. But, `tokens_to_mint` will increase as `2.1e18/1e18 - 1.9e18/1e18 => 2 - 1 = 1`

Hence after the transfer, Alice will still have an ERC20 balance of `1.9e18` but 2 nfts. This could be repeated for as many times as you want.

Since all the balance is kept in the ERC20 token balance representation this will not affect anything more than the nft balance representation. If Alice would try to transfer her two nfts it would revert as she hasn't got `2e18` ERC20 tokens to transfer.

This also works in the other direction. A user could artificially decrease their nft balance by doing a transfer to themselves. Do the same scenario as above but Alice starts with `1.1e18` tokens. Then she would end up with no nfts but `1.1e18` tokens after transferring `0.2e18` tokens to herself.

Recommendation

Consider doing the balance changes before the balance sync, then using the changed values to calculate the difference:

```
        uint256 balanceBeforeSender = _balances[from];
        uint256 balanceBeforeReceiver = _balances[to];

+       _update(from, to, value);

        if (!isInAllowlist(from)) {
            uint256 tokens_to_burn = (balanceBeforeSender / 10 **
_decimals) -
-               ((balanceBeforeSender - value) / 10 ** _decimals);
+               (_balances[from] / 10 ** _decimals);
            _nft_burn(from, 0, tokens_to_burn);
        }

        if (!isInAllowlist(to)) {
-           uint256 tokens_to_mint = ((balanceBeforeReceiver + value) /
+           uint256 tokens_to_mint = (_balances[to] /
                10 ** _decimals) - (balanceBeforeReceiver / 10 **
_decimals);
            _nft_mint(to, 0, tokens_to_mint);
        }

-       _update(from, to, value);
```

[M-02] `ERC420.setAllowList()` can replicate the previous target state

Severity

Likelihood: Medium

Impact: Low

Description

- The `ERC420` contract owner can add/remove accounts from/to the allowList, and if an account is added to the the allowList (where the `state == true`); then his NFTs will be burnt, and if an account is removed from the allowList (where the `state == false`) then he will be minted an NFTs equivalent to his balance of tokens (`balance / 1e18`).
- But it was noticed that the `setAllowList()` function misses checking if the new state is different from the currently assigned one, so if the current `state` of the target was set previously to `false` (where this target has previously minted NFTs when changing his state), then if the owner re-sets the

target's state again to `false`; the target will be again minted NFTs equivalent to his current tokens balance.

- Same issue if the target's previous state was `true` (the target is in the allowList), where his tokens can be burnt twice if the owner calls `setAllowList()` without changing the target's state (note: due to another vulnerability; the target can be minted tokens even if he's in the allowList via `safeBatchTransferFrom()` function as it doesn't check for the receiver being allowListed or not before minting him NFTs).

Code Snippet

ERC420.setAllowList function

```
function setAllowList(address target, bool state) public onlyOwner {
        _allowList[target] = state;
        uint256 balance = _balances[target];
        if (state) {
            uint256 tokens_to_burn = balance / 10 ** _decimals;
            _nft_burn(target, 0, tokens_to_burn);
        } else {
            uint256 tokens_to_mint = balance / 10 ** _decimals;
            _nft_mint(target, 0, tokens_to_mint);
        }
    }
```

Recommendation

Update `setAllowList()` to check if the new state is different from the currently assigned one:

```
function setAllowList(address target, bool state) public onlyOwner {
+       require(_allowList[target] != state,"assigning the same state is
not allowed");
        _allowList[target] = state;
        uint256 balance = _balances[target];
        if (state) {
            uint256 tokens_to_burn = balance / 10 ** _decimals;
            _nft_burn(target, 0, tokens_to_burn);
        } else {
            uint256 tokens_to_mint = balance / 10 ** _decimals;
            _nft_mint(target, 0, tokens_to_mint);
        }
    }
```

[M-03] `ERC420._safeBatchTransferFrom()` doesn't verify allowList

Severity

Likelihood: Medium

Impact: Medium

Description

ERC420 contract is designed to work with one type of NFT with id=0, so if a user has 100 full tokens then he has a 100 NFT of id=0 (as having one full token -1e18- will mint you one NFT, and having less than one full token -1e17 for example- will not mint you an NFT). When the user transfers his tokens, an equivalent amount of NFTs will be burnt from his balance **if he is not in the allowList**, and the receiver will be minted an equivalent amount of these transferred tokens **if he is not in the allowList**:

```
function _transfer(address from, address to, uint256 value) internal {
        if (from == address(0)) {
            revert InvalidSender(address(0));
        }
        if (to == address(0)) {
            revert InvalidReceiver(address(0));
        }

        uint256 balanceBeforeSender = _balances[from];
        uint256 balanceBeforeReceiver = _balances[to];

        if (!isInAllowlist(from)) {
            uint256 tokens_to_burn = (balanceBeforeSender / 10 **
_decimals) -
                ((balanceBeforeSender - value) / 10 ** _decimals);
            _nft_burn(from, 0, tokens_to_burn);
        }

        if (!isInAllowlist(to)) {
            uint256 tokens_to_mint = ((balanceBeforeReceiver + value) /
                10 ** _decimals) - (balanceBeforeReceiver / 10 **
_decimals);
            _nft_mint(to, 0, tokens_to_mint);
        }

        _update(from, to, value);
    }
```

so if userA transferred 3 tokens (3e18) to userB, then 3 NFTs will be burnt from userA if he is not in the allowList, and userB will be minted 3 NFTs if he is not in the allowList.

safeBatchTransferFrom function is supposed to transfer a batch of NFTs with different ids from the sender to the receiver, but since the contract is designed to handle only one NFT type of id=0, then this function is considered as a dummy function that has no difference from the transfer function, where it will only handle transferring tokens of id=0:

```
function _safeBatchTransferFrom(
        address from,
        address to,
        uint256[] memory ids,
        uint256[] memory values
    ) internal {
        if (to == address(0)) {
            revert InvalidReceiver(address(0));
        }
        if (from == address(0)) {
            revert InvalidSender(address(0));
        }
        uint value;
        for (uint256 i = 0; i < ids.length; i++) {
            value += values[i];
        }
        _update(from, to, value * 10 ** _decimals);
        _nft_update(from, address(0), ids, values);
        _nft_mint(to, 0, value);
    }
```

But as can be noticed, the `_safeBatchTransferFrom` function burns the NFTs from the sender and mints the receiver **without checking if they are in the allowList or not** (similar check made in the `transfer`, `safeTransferFrom` & `transferFrom` functions).

This will result in an inconsistency in the transfer mechanism across `ERC420` transfer functions, where users in the allowList can be minted NFTs/ burning from their NFTs when transferring tokens via `safeBatchTransferFrom()` function.

Code Snippet

ERC420._safeBatchTransferFrom function

```
function _safeBatchTransferFrom(
        address from,
        address to,
        uint256[] memory ids,
        uint256[] memory values
    ) internal {
        if (to == address(0)) {
            revert InvalidReceiver(address(0));
        }
        if (from == address(0)) {
            revert InvalidSender(address(0));
        }
        uint value;
        for (uint256 i = 0; i < ids.length; i++) {
            value += values[i];
        }
        _update(from, to, value * 10 ** _decimals);
        _nft_update(from, address(0), ids, values);
        _nft_mint(to, 0, value);
    }
```

Recommendation

Since `ERC420` contract is designed to handle only one NFT type (with id=0), the update `_safeBatchTransferFrom` function to check if the sender/receiver are not in the allowList before burning/minting NFTs:

```
function _safeBatchTransferFrom(
        address from,
        address to,
        uint256[] memory ids,
        uint256[] memory values
    ) internal {
        if (to == address(0)) {
            revert InvalidReceiver(address(0));
        }
        if (from == address(0)) {
            revert InvalidSender(address(0));
        }
        uint value;
        for (uint256 i = 0; i < ids.length; i++) {
            value += values[i];
        }
        _update(from, to, value * 10 ** _decimals);
-       _nft_update(from, address(0), ids, values);
-       _nft_mint(to, 0, value);

+       if (!isInAllowlist(from)) {
+       _nft_update(from, address(0), ids, values);
+       }


+       if (!isInAllowlist(to)) {
+       _nft_mint(to, 0, value);
+       }
    }
```

[M-04] Approve race condition in `ERC420.approve()`

Severity

Likelihood: Medium

Impact: Medium

Description

The `ERC420.approve()` doesn't have any protection against the multiple withdrawal attack on the `approve()` , `transferFrom()` & `safetTransferFrom` functions.

This race condition can occur when the owner calls `approve()` function to change the allowance of the spender on his tokens, but then the spender sandwiches the `approve()` transaction where he first frontruns it and calls `transferFrom()` before changing the allowance and the other `transferFrom()` call is made after the `approve()` transaction is made (that has changed the spender allowance); which will lead to the owner's tokens being spent twice by the spender.

Code Snippet

ERC420.approve function

```
    function approve(address spender, uint256 value) external returns
(bool) {
        address owner = msg.sender;
        _approve(owner, spender, value);
        return true;
    }
```

Recommendation

Add functions to increase/decrease allowance.

[L-01] No call to ERC1155 transfer hook

The EIP-1155 standard states that when doing a ERC1155 transfer the `ERC1155TokenReceiver` must be called when the receiver is a contract.

This is not followed in `Groge`. Hence the ERC1155 implementation is not compliant with the EIP-1155 standard. Which as above, can cause problems with external integrations and tooling.

Recommendation

Consider implementing the `ERC1155TokenReceiver` hooks for the receiver to acknowledge the transfer.

[I-01] Lack of `memory-safe` annotation on memory safe assembly

To create lists of only one item the protocol uses a function `_asSingletonArrays` which uses assembly to create the array for gas efficiency.

This is very similar to the same function in `OpenZeppelin`. Here however, they use the `/// @solidity memory-safe-assembly` which hints the compiler that it can make certain assumptions about the code.

This annotation is not used in the `Groge::_asSingletonArrays` which can limit compiler optimizations.

Recommendation

Consider adding `memory-safe` to the assembly:

```diff
-       assembly {
+       assembly ("memory-safe") {
```

Can use `("memory-safe")` here as this contract will be compiled with solidity >0.8.

[I-02] Events emitted when transferring ERC1155 not following EIP-1155

The EIP-1155 standard defines that a transfer between two users should emit an event `TransferSingle` with `_from` and `_to`:

> MUST emit the `TransferSingle` event to reflect the balance change (see “TransferSingle and TransferBatch event rules” section).

and

> `TransferSingle` SHOULD be used to indicate a single balance transfer has occurred between a `_from` and `_to` pair

This is not followed, as when doing a transfer two `TransferSingle` events are emitted, `Groge::_safeTransferFrom`:

```
if (!isInAllowlist(from)) {
     // will emit a `TransferSingle` event: from -> 0, a burn
    _nft_update(from, address(0), ids, values);
}
if (!isInAllowlist(to)) {
    // will emit a `TransferSingle` event: 0 -> to, a mint
    _nft_mint(to, 0, value);
}
```

This can be confusing as no actual event, `from` -> `to` is emitted. It also makes the `Groge` contract non-compliant with EIP1155. Which in turn can make external integrations and tooling not work as expected.

Recommendation

Consider only emitting one event when ERC1155 transfers are used (`safeTransferFrom`/`safeBatchTransferFrom`) calls.

[I-03] No event emitted when adding or removing account in allow list

It's generally considered good practice to emit events for any state changes so that these can be easily tracked off chain. This is however not followed in `setAllowList`. There no event is emitted when an account is added or removed.

Recommendation

Consider emitting an event when adding or removing an account in the allow list.

[I-04] Unnecessary `msg.sender != address(0)` check

When doing a burn there is a check that `msg.sender != address(0)` in `burn`:

```
File: Groge.sol

        address owner = msg.sender;
        if (owner == address(0)) {
            revert InvalidSender(address(0));
        }
```

Here, `msg.sender` is checked not to be `address(0)`, it is very, very unikely that `msg.sender` would be `address(0)` thus this check is unnecessary.

Recommendation

Consider removing the `address(0)` check

[I-05] `ERC420.setAllowList()` can be optimized

`ERC420.setAllowList()` function is meant to burn/mint the targets's NFTs based on the new state, where the number of these burnt/minted tokens is calculated based on the target's current balance:

```
function setAllowList(address target, bool state) public onlyOwner {
        _allowList[target] = state;
        uint256 balance = _balances[target];
        if (state) {
            uint256 tokens_to_burn = balance / 10 ** _decimals;
            _nft_burn(target, 0, tokens_to_burn);
        } else {
            uint256 tokens_to_mint = balance / 10 ** _decimals;
            _nft_mint(target, 0, tokens_to_mint);
        }
    }
```

so as can be noticed, the function will proceed with burning/minting NFTs even if the calculated `tokens_to_burn` or `tokens_to_mint` is zero, which will result in unnecessary gas consumption without updating the balance state.

Recommendation

Update `setAllowList()` function to burn/mint NFTs if the calculated tokens to burn/mint is > 0:

```
function setAllowList(address target, bool state) public onlyOwner {
        _allowList[target] = state;
        uint256 balance = _balances[target];
        if (state) {
            uint256 tokens_to_burn = balance / 10 ** _decimals;
+            if(tokens_to_burn > 0)
            _nft_burn(target, 0, tokens_to_burn);
        } else {
            uint256 tokens_to_mint = balance / 10 ** _decimals;
+            if(tokens_to_mint > 0)
            _nft_mint(target, 0, tokens_to_mint);
        }
    }
```

[I-06] `ERC420.balanceOf()` doesn't check if the id is valid

`ERC420.balanceOf()` function is supposed to return the amount of tokens of a specific id, but the function doesn't check if the id is within the bounds as the maximum id is the `_nft_count` (but the current design always assumes that there's only one type of NFT with id=0).

ERC420.balanceOf function

```
    function balanceOf(
        address account,
        uint256 id
    ) public view returns (uint256) {
        return _nft_balances[id][account];
    }
```

Recommendation

Update `balanceOf()` function to check if the token id is a valid one:

```
    function balanceOf(
        address account,
        uint256 id
    ) public view returns (uint256) {
+       require(id < _nft_count, "invalid token id");
        return _nft_balances[id][account];
    }
```

[I-07] Updating baseURI renders previously minted NFT URIs invalid

The owner of the `Groge` contract can change the `baseURI` of the collection, where this would result in generating wrong/invalid URIs for the previously minted NFTs so they can't be rendered correctly.

Groge.setURI function

```
function setURI(string memory newURI) public onlyOwner {
     baseURI = newURI;
 }
```

Recommendation

Implement a mechanism to cache the `baseURI` for each minted token collection.