SOLIDIFIED

Audit Report for Grease Monkey - June 19, 2023

Summary

Audit Report prepared by Solidified covering the Grease Monkey contracts.

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code below. The final debrief took place on June 2, 2023, and the results are presented here.

Audited Files

The source code has been supplied in the following source code repositories:

Repo: https://github.com/Grease-Monkey-Games/tm-smart-contracts
Commit hash: `4692094cc6e7badd54a93f43e31f415cb42131b6`
Fixes received at commit `aa27ab566e9e633a443dff583c2e7b218098fd03`

```
contracts
└── nissan
    └── cars
        ├── ERC721_GMG_Asset.sol
        └── MintManager.sol
```

Intended Behavior

The audited code base implements GMG assets that can be bought with ERC20 tokens. The mints can be randomized with Chainlink's VRF.

Findings

Smart contract audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of a smart contract system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**.

Note, that high complexity or lower test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than a security audit and vice versa.

Criteria	Status	Comment
Code complexity	Low	-
Code readability and clarity	Medium	-
Level of Documentation	High	-
Test Coverage	High	-

SOLIDIFIED

Audit Report for Grease Monkey - June 19, 2023

Issues Found

Solidified found that the Grease Monkey contracts contain no critical issues, 2 major issues, 7 minor issues, and 6 informational notes.

We recommend issues are amended, while informational notes are up to the team's discretion, as they refer to best practices.

Issue #	Description	Severity	Status
1	Users can cancel Chainlink VRF fulfillments by removing approvals	Major	Fixed
2	Chainlink subscription funds can be drained by spamming invalid requests	Major	Fixed
3	pickRandomTokenType skips token types in some cases	Minor	Fixed
4	createCustomId vulnerable to ID collisions	Minor	Fixed
5	mintBatch can return wrong IDs	Minor	Fixed
6	MINTER_ROLE can be granted to arbitrary addresses	Minor	Acknowledged
7	Batch mints may revert when the current max supply is too low	Minor	Fixed
8	setCheckoutWallet does not update default royalty receiver	Minor	Fixed
9	getVariantRemainingSupply may return wrong values	Minor	Fixed
10	Function mintType() does not have a check for _tokenType	Note	Fixed
11	Floating pragma	Note	Fixed
12	Function withdrawToken() compares the balance with 0 instead of amount to withdraw	Note	Fixed

13	Unnecessary receive for GMG assets	Note	Fixed
14	Improvements for token recovery functions	Note	Fixed
15	Redundant require in function fulfillRandomWords	Note	Fixed

Critical Issues

No critical issues have been found.

Major Issues

1. Users can cancel Chainlink VRF fulfillments by removing approvals

When a user sees a call to `MintManager.fulfillRandomWords` in the mempool, he can calculate the NFT that he will get. If the result is not desired, he can set the ERC20 approval of the payment token for the `MintManager` contract to zero. This will cause the ERC20 transfer within `fulfillRandomWords` to fail and the whole VRF fulfillment to fail because of this reversion.

Recommendation
We recommend locking the funds in the mint manager after a request. The user should then be reimbursed if the mint fails for a reason that he does not control (e.g., by wrapping the mint in an external call to the `MintManager` itself and having a `try` / `catch` block).

2. Chainlink subscription funds can be drained by spamming invalid requests

The function `MintManager.requestRandomNFTs` only does minimum input validation. It is not checked whether the passed variants are valid variants (only if they adhere to the interface, which any contract can) or if the payment token is a valid ERC20 token that is supported by the variant. An attacker can abuse this to create many invalid requests that will fail when they are

fulfilled (i.e., `fulfillRandomWords` will revert). The Chainlink subscription is still charged in such cases, which means that this can be abused to drain it completely.

Recommendation
This issue can be resolved with a similar solution like the previous one. If the funds are locked up after a request and it is furthermore validated that only valid variants (e.g., by whitelisting them) and payment tokens (by validating if they are supported before the fulfillment) are used, the attack is no longer possible.

Minor Issues

3. `pickRandomTokenType` skips token types in some cases

The function `pickRandomTokenType` first checks if the picked random number (modulo the number of token types) is available for minting. If not, all other variants are checked and the first one that is available is returned. However, the algorithm to do so contains an error, which means that not all types are always checked. In general, the `tokenType + index` (where `index` is a consecutive loop variable) is checked. The loop also contains the following line to limit the range of the checked values:
`if(tokenType > tokenTypes.length - 1) tokenType = 1;`
Now imagine that `tokenTypes.length` is 6 and the resulting `randomNumberModded` is 3. When the `else` branch is entered, `tokenType` starts at 5 (with `index = 1`). After one loop iteration (i.e., when `index` is 2), `tokenType` is reset to 1. However, it is then assigned `tokenType + index = 3`. Therefore, the value 2 is skipped and will never be checked.

Note that `MintManager.pickRandomVariant` uses the same algorithm and therefore has the same problem. Additionally, since the `_variants` array indexing starts from `0` (differently from `tokenTypes`), the `for` loop should start from `index = 0` as well.

The issue is demonstrated in the following proof of concept:

```
function test_random_doesnot_traverse_all() external {
    uint _randomNumber = 4;
    uint length = 7;
    uint256 randomNumberModded = _randomNumber % (length - 1);
    console.log(randomNumberModded +1);
    uint256 tokenType = randomNumberModded + 1 + 1;
    for (uint256 index = 1; index < length - 1; index++) {
        if (tokenType > length - 1) tokenType = 1;
        console.log(tokenType);
        tokenType = tokenType + index;
    }
}
```

Recommendation

We recommend increasing `tokenType` by 1 instead of `index` and starting the for loop at the proper index for `MintManager.pickRandomVariant`.

4. `createCustomId` vulnerable to ID collisions

The function `getCustomTokenId` in `ERC721_GMG_Asset.sol` passes different `uint256` values to `GMG_CustomTokenIds_NFT_Asset.createCustomId`. These are combined into one ID using `GMG_CustomTokenIds_Base.addField` with bit-shift and logical OR operations. Every field has a maximum allowed length, for instance, `MODEL_BITS` (6) for the model field. However, it is never validated that this value fits in 6 bits. If it does not, it will set some bits of other fields. This can cause ID collisions and means that the view functions which extract the fields will return wrong values.

Recommendation

We recommend restricting the range of all values to the allowed range. For instance, it should not be possible to create a GMG asset where the model value is larger than `2**6 - 1`.

5. `mintBatch` can return wrong IDs

The function `mintBatch` allocates an array `allTokenIds` of size `_amount` for returning the token IDs. However, the amount that is minted (which is also used in the mint loop) is stored in the variable `_amountToMint`. As long as there are enough tokens available for the mint, these values will be equal. If not, `_amountToMint` is smaller than `_amount`. Because of this, the array will contain trailing zeroes, which may lead to errors in callers of the function.

Recommendation

We recommend allocating an array of size `_amountToMint`.

6. `MINTER_ROLE` can be granted to arbitrary addresses

All mint functions within `ERC721_GMG_Asset` are restricted to the role `MINTER_ROLE`. While this role should usually be only held by the `MintManager` contract, this is not enforced. In principle, the administrator (deployer of the contract) can grant the role to other addresses, which could then mint arbitrarily and control the randomness.

Recommendation

We recommend restricting minting to only one address (the `MintManager` contract) instead of using a role that can be granted to multiple addresses.

7. Batch mints may revert when the current max supply is too low

The batch mint functions in `ERC721_GMG_Asset` contain the following line to choose the amount that is minted:

```
uint256 _amountToMint = maxSupply > 0 && _amount + supply > maxSupply ? maxSupply - supply : _amount;
```

This logic has two problems:

- In `doMint` (that is called later for the actual mint), `totalSupply() < currentMaxSupply` is enforced. Because `currentMaxSupply <= maxSupply` always holds, this is a stricter check and it can result in situations where the batch mint fails, although it would not, if the amount was properly adjusted. For instance, consider a scenario where `maxSupply = 20`, `currentMaxSupply = 18`, `totalSupply() = 17`. A batch mint with amount 3 would fail, although it could succeed with an adjustment of the amount to 1.
- Previously, it was already checked that `getRemainingSupplyOfTokenType(1) >= _amount`. Furthermore, the initializer enforces that `maxSupply` is equal to the max supplies of all token types. Because of this relation, it is mathematically impossible that `_amount + supply > maxSupply`.

Recommendation

We recommend replacing `maxSupply` in the check with `currentMaxSupply`.

8. setCheckoutWallet does not update default royalty receiver

---

In the constructor, the default royalty receiver is set to `_checkoutWallet`. However, when the checkout wallet is later updated with `setCheckoutWallet`, the default royalty receiver is not updated. This may lead to unintended situations where an old wallet still receives royalties.

Recommendation

We recommend updating the default royalty receiver in `setCheckoutWallet`.

9. getVariantRemainingSupply may return wrong values

The function `MintManager.getVariantRemainingSupply` is used in various places to check if it is possible to mint a given amount of a variant. However, the function queries `variant.maxSupply()`, which can be higher than `variant.currentMaxSupply()`. Because the `doMint` function of the variant then uses `currentMaxSupply`, this can lead to unintended behavior. For instance, `pickRandomVariant` may pick a variant that is currently not mintable because of this.

Recommendation

We recommend querying `variant.currentMaxSupply()`.

Informational Notes

10. Function mintType() does not have a check for _tokenType

The function `mintType()` is used to mint NFTs of a particular type. However, it does not check if the type of token provided as an argument (`_tokenType)` exists or not. This would cause the function to fail without any readable error.

Recommendation

Add a check to ensure that the `_tokenType` provided for minting exists.

11. Floating pragma

Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively, or recently released pragma versions may have unknown security vulnerabilities.

Recommendation

Consider locking the pragma version. It is not recommended to use a floating pragma in production. Apart from just locking the pragma version in the code, the sign (^) needs to be removed.

12. Function withdrawToken() compares the balance with 0 instead of amount to withdraw

The function `withdrawToken()` is used for withdrawing tokens from the NFT contract. It compares whether the token balance is more than 0 and transfers the amount passed as an argument to the owner. It is better to compare the token balance with `_amount` and revert with a readable error message if the comparison fails.

Recommendation

Change the check as follows.

```
require(tokenContract.balanceOf(address(this)) >= _amount , "Token balance can't be less than _amount");
```

13. Unnecessary receive for GMG assets

The `ERC721_GMG_Asset` contract has a `receive` function which emits an event with the received ETH / native token. However, the contract should never receive native tokens when a user mints (as minting with native tokens is not supported and all proceeds go to a dedicated fee wallet).

Recommendation

Consider removing the `receive` function.

14. Improvements for token recovery functions

The function `withdrawToken` uses `transfer` directly for `transferring` the ERC20 tokens. To support arbitrary ERC20 tokens, using a library like `SafeERC20` is recommended. Moreover, `withdrawNative` uses `transfer` for transferring native tokens, which is not recommended anymore.

Recommendation

Consider replacing these token recovery functions.

15. Redundant `require` in function `fulfillRandomWords`

In line 214 of the function `fulfillRandomWords()`, the check the `require(getVariantRemainingSupply(randomVariant) >= 1)` is redundant because the function `pickRandomVariant` already takes care of the condition.

Recommendation

Consider removing the check.

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of GREASE MONKEY GAMES PTY. LTD. or its products. This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors of Solidified platform from legal and financial liability.

Oak Security GmbH