Diabolical Machines

Anma

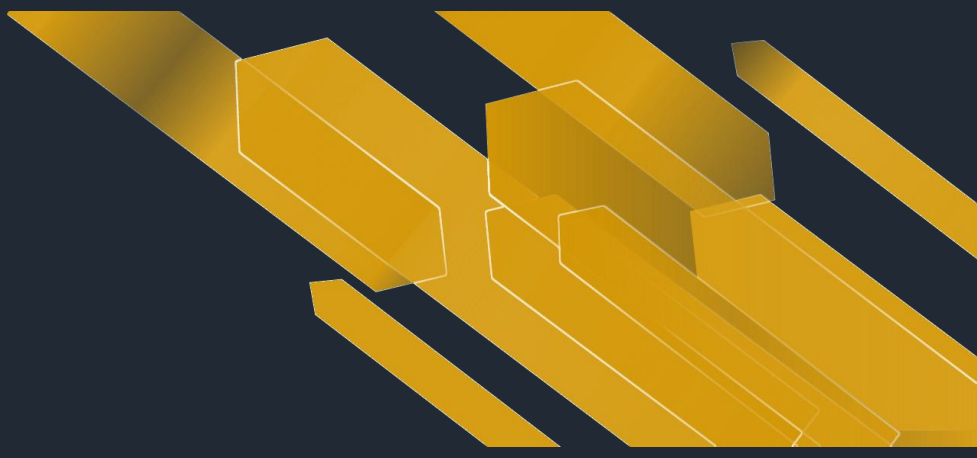
SMART CONTRACT
SECURITY AUDIT

Date:

May 16th 2023

Written by:

Pim Hakkert



DISCLAIMER

The following disclaimer outlines important information related to a Solidity smart contract audit conducted by Pim Hakkert.

Pim performed the audit as a single individual, and his report does not express any opinion or judgment about the client or their project. It is crucial to understand that this audit should not be taken as an endorsement or criticism of the client or their project.

It is important to acknowledge that no smart contract can be declared as "safe" due to the absence of regulations regarding their quality. Pim's responsibility was limited to reviewing the Solidity smart contracts delivered by the client within the scope of the audit.

Despite his best efforts to review the code to the best of his abilities, it cannot be assumed that the smart contracts are entirely safe. Therefore, the client must assess whether additional audits by third parties are necessary to identify any issues with the smart contracts.

Solidity is a continuously evolving language, and Ethereum (hard)forks past the front cover’s date can render parts of the audit review obsolete. As such, it is essential to keep in mind that the audit report reflects the state of the reviewed contracts as of the audit's date.

To mitigate the risks associated with using smart contracts, it is recommended that the client stay up-to-date with any changes to the Solidity language and Ethereum network, conduct regular audits, and engage with professionals experienced in smart contract development and security.

PROCESS

The smart contracts are audited according to a 7-step plan:

Step 1. The client forks the Github repository that contains the smart contracts. This repository contains a valid installation of a framework such as Hardhat or Foundry.

Step 2. The client modifies the repository to prepare it for an audit and creates a readme file that outlines instructions for installing dependencies and running tests. The client also invites Pim's Github account (pimhakkert) as a collaborator.

Step 3. The tests are reviewed and either approved to run at an acceptable level or flagged for fixes before the audit can commence.

Step 4. Pim conducts a multi-day audit of the code and produces an audit report that is shared with the client.

Step 5. The client responds to all questions and findings raised in the audit report, making any necessary fixes, and pushes the updated code to the Github fork.

Step 6. The updated code is reviewed, and a final audit report is prepared and sent to the client.

Step 7. The client approves the final report, and a public version of the audit report is published.

SUMMARY

Details

Project name: Diabolical Machines

Description: Diabolical Machines is an NFT with randomly chosen attributes. An animated SVG is generated with on-chain code with randomisation provided from an oracle. The code includes an English Auction with static incrementing amounts of ether.

Audit details

Methods used: Static analysis, manual review, fuzzing

Audit timeframe: April May 9th – May 16th 2023

Commits:

- `51aaf3a60e82e501434f336cb0bb4766d7e657b9`
- `fdeb6255711e044955cf255a213c97f2f3dd4d95`
- `82d287ee82496ee44fdb6f38d19d786103e82b7b`

Findings

Total findings	5
High risk	0
Medium risk	0
Low risk	3
Informational / Gas	2

Team Interaction

“Anma specialises in generative NFT projects and that expertise is visible within the in-scope contracts. The team has an understanding of how to apply random noise to generate NFTs with values according to their desired distribution.

They were quick to answer any of my questions and concerns, and I appreciated their technical documentation which was provided to me prior to the audit. Theirs is a good example to follow.

“- Pim

Project & team score

Preparation

Communication

Processing speed

Contracts audited

AssetRetriever	src/AssetRetriever.sol
Clifford	src/Clifford.sol
CommonSVG	src/CommonSVG.sol
Environment	src/Environment.sol
GlobalNumbers	src/GlobalNumbers.sol
GlobalSVG	src/GlobalSVG.sol
GridHelper	src/GridHelper.sol
Machine	src/Machine.sol
Metadata	src/Metadata.sol
Noise	src/Noise.sol
Patterns	src/Patterns.sol
TraitBase	src/TraitBase.sol

FINDINGS

Audit findings are organized by their severity, and any affected contracts are identified. Each finding features a title and a unique ID for easy reference. Within the finding, the issue is clearly explained, relevant lines of code are highlighted, a recommendation is presented, and any associated links to support the suggestion are provided.

High risk

There were no high-severity findings found within any of the in-scope smart contracts.

High severity findings are critical issues that pose a significant threat to the smart contract's security, functionality, or user assets.

These issues may result in the loss of funds, unauthorized access, or a complete breakdown of the contract's intended operations. Addressing high severity findings is of utmost importance, and they should be resolved before the contract is deployed or used in production.

Medium risk

There were no medium-severity findings found within any of the in-scope smart contracts.

Medium-severity findings are issues that may not immediately compromise the smart contract's security but can still have a considerable impact on its proper functioning or user experience.

These issues could potentially be exploited by an attacker in certain scenarios or lead to unexpected outcomes. Addressing medium-severity findings is important, and they should be carefully reviewed and fixed as necessary.

Low risk

Three (3) low-severity findings were found while auditing the in-scope smart contracts.

Low-severity findings are minor issues that are unlikely to have a significant impact on the smart contract's overall security or functionality.

These issues may still be worth addressing to improve the contract's robustness, reliability, or user experience, but they are generally not considered urgent.

L1: constrainToHex has incorrect docs and lacks bounds checks

The `GridHelper::constrainToHex()` function is supposed to constrain a value to the range of 0 to 255, but does so only with values higher than `-255` and lower than `511`. Any value higher than `510` will cause an underflow - which wont revert because as it is casted from a `int` to a `uint` - and produce a very high number, way higher than 255.

The developer has noted that the highest value this function will receive is `311`. This, however, is not documented and not checked within the code.

Affected Contracts

- GridHelper

Recommendation

Add documentation to explain that values above `510` will wrap around to an extremely high `uint`. A value lower than `-255` will result in a value higher than the max baseline rarity.

AND

Add a requirement for the input value:

```
require(value >= -255 && value <= 510, "Value out of bounds.");
```

Result

Resolved. The provided “require” statement was added together with extra documentation.

L2: Owner can start auction before intended

While there is a variable `Clifford::cypherClaimStarted` that is set to true by running `Clifford::startCypherClaimPeriod()`, there is nothing preventing the owner from starting the auction. If the owner by accident runs the `Clifford::startAuction()` function, Cypher owners can no longer claim their free NFT, even if they are promised off-chain that they can do so for a set period of time.

Affected Contracts

- Clifford

Recommendation

Add a `Clifford::cypherClaimEnded` variable that is manually set to true by the owner after the claim period has ended.

OR

An even better solution is modifying `Clifford::startCypherClaimPeriod()` by setting a state variable to a timestamp in the future that must be passed before `Clifford::startAuction()` will execute. The start auction function will need to modify the starting requirements:

```
if (!cypherClaimStarted) revert CypherClaimNotStarted();
if (!cypherClaimFinished) revert CypherClaimNotFinishedYet(); //NEW
if (startedAt != 0) revert AuctionAlreadyStarted();
```

Result

Resolved. The team altered the logic so that the auction will start automatically after the claim period has ended. The `Clifford::startAuction()` function was removed.

L3: Bid increment requirement

Line 254 of `Clifford` requires that the user places a bid that matches `bidAmount % BID_INCREMENT == 0`. This means that a user cannot bid a custom amount of ether, like `0.015`, which at the time the audit was performed can mean a difference of $7.

Affected Contracts

- Clifford

Recommendation

Remove the existing requirement and replace it with `bidAmount >= BID_INCREMENT`.

AND

Change `Clifford::getMinimumBid()` so that it wont force the next increment `(2.015 -> 2.02)`, but rather require the increment to be added to the minimum value. `(2.015 → 2.025)`.

OR

Ignore if this is the intended behavior.

Result

Acknowledged. The team commented that this is intended behavior. It reduces the chance of people getting knocked out of the auction, resulting in less NFTs remaining after the auction finished.

Informational/Gas

Two (2) info/gas severity findings were found while auditing the in-scope smart contracts.

Informational severity findings, often associated with gas savings, are observations that do not pose a direct risk to the smart contract's security or functionality but may provide valuable insights for optimization, code readability, or adherence to best practices.

Addressing these issues can lead to a more efficient contract execution and reduced gas costs, improving the overall user experience.

IG1: Duplicate getProductivity function names

The `getProductivity()` function name is declared in multiple contracts, which can cause confusion. The only function worthy of this name is `AssetRetriever::getProductivity()`, which returns a `uint`. The `Machine::getProductivity()` function only returns a string, and is used by `Metadata::getProductivity()`.

Affected Contracts

- Machine
- AssetRetriever
- Metadata

Recommendation

Consider renaming the last two functions to `getProductivityTier()`, as specified in the function docs in `Machine::getProductivity()`.

Result

Resolved. The team renamed the `getProductivity()` functions from `Machine` and `Metadata` to `getProductivityTier()`

IG2: AssetIds must be divisible by 1000 for correct functionality

Functions `AssetRetriever::getAsset()` and `AssetRetriever::getProductivity()` require an `assetID` in the multiple thousands to be able to get the right asset/productivity from the correct TraitBase. If asset IDs are not correctly configured in the `/src/Assets` folder, wrong TraitBases could be used for assets, which may result in the wrong assets being picked, or none at all.

Affected Contracts

- AssetRetriever

Recommendation

Ensure every asset ID can be divided by 1000, as expected in the return statements of `AssetRetriever::getAsset()` and `AssetRetriever::getProductivity()`.

Result

Acknowledged. An internal spreadsheet is present which tracks all assets, to ensure they are divisible by 1000.

CLOSING WORDS

”I enjoyed working together with the team to ensure their contracts are secure. Before this audit I had no idea there were libraries to alter and generate SVG code, but I am glad to have experienced viewing a good use case for it.

Any minor flaws in the auction logic have been rectified and I hope the auction itself goes well. Hopefully we can work together again in the future.

”**- Pim**