Smart Contract Audit Report

p00ls

16th of August 2022

Contents

1. Preface 3

2. Manual Code Review 4

2.1 Severity Categories 4

2.2 Summary 5

2.3 Findings 6

2.4 Gas Optimizations 14

2.5 Informational Notes 15

3. Protocol/Logic Review 18

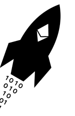
4. Summary 19

Disclaimer

As of the date of publication, the information provided in this report reflects the presently held understanding of the auditor's knowledge of security patterns as they relate to the client's contract(s), assuming that blockchain technologies, in particular, will continue to undergo frequent and ongoing development and therefore introduce unknown technical risks and flaws. The scope of the audit presented here is limited to the issues identified in the preliminary section and discussed in more detail in subsequent sections. The audit report does not address or provide opinions on any security aspects of the Solidity compiler, the tools used in the development of the contracts or the blockchain technologies themselves, or any issues not specifically addressed in this audit report.

The audit report makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, the legal framework for the business model, or any other statements about the suitability of the contracts for a particular purpose, or their bug-free status.

To the full extent permissible by applicable law, the auditors disclaim all warranties, express or implied. The information in this report is provided "as is" without warranty, representation, or guarantee of any kind, including the accuracy of the information provided. The auditors hereby disclaim, and each client or user of this audit report hereby waives, releases and holds all auditors harmless from, any and all liability, damage, expense, or harm (actual, threatened, or claimed) from such use.



1. Preface

The developers of **p00ls** contracted byterocket to conduct a smart contract audit of their creator token suite. p00ls "*enables creators and brands to create the tokens of their worlds. [They] launch and distribute it to their communities*".

The team of byterocket reviewed and audited the above smart contracts in the course of this audit. We started on the 28th of July and finished on the 16th of August 2022.

The audit included the following services:

- *Manual Multi-Pass Code Review*
- *Protocol/Logic Analysis*
- *Automated Code Review*
- *Formal Report*

byterocket gained access to the code via a public GitHub repository. We based the audit on the main branch's state on July 27th, 2022 (*commit hash ada109e6f47aa4f682c3bf4eebba53412d8ad663*). The updated version was provided to us via multiple new commits to the repository, addressing our findings. The last and most recent commit hash that we audited is *8c50460480757b52f377d2ab98430e4b155a4372*.

2. Manual Code Review

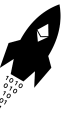
We conducted a manual multi-pass code review of the smart contracts mentioned in section (1). Three different people went through the smart contract independently and compared their results in multiple concluding discussions.

The manual review and analysis were additionally supported by multiple automated reviewing tools, like Slither, GasGauge, Manticore, and different fuzzing tools.

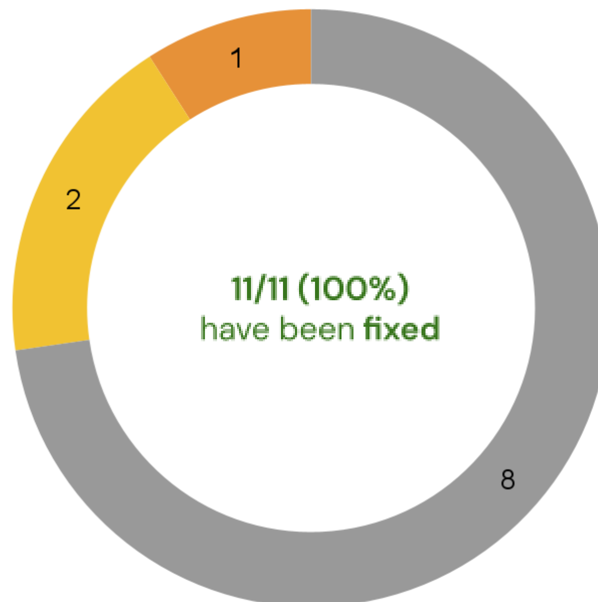
2.1 Severity Categories

We are categorizing our findings into four different levels of severity:

Non Critical	Does not impose immediate risk but is relevant to security best practices. Includes issues with - Code style and clarity - Versioning - Off-chain monitoring
Low Severity	Imposes relatively small risks or could impose risks in the long-term but without assets being at risk in the current implementation. Includes issues with - State handling - Functions being incorrect as to specification - Faulty documentation or in-code comments
Medium Severity	Imposes risks on the function or availability of the protocol or imposes financial risk by leaking value from the protocol if external requirements are met.
High Severity	Imposes catastrophic risk for users and/or the protocol. Includes issues that could result in - Assets being stolen/lost/compromised - Contracts being rendered useless - Contracts being gained control of



2.2 Summary



On the code level, we **found 11 bugs or flaws, with 11 of them being fixed in a subsequent update.** Prior to this, there have been 8 non-critical, 2 of low severity and 1 of medium severity findings. Our automated systems and review tools did **not find any additional ones.** Additionally, we found 3 gas improvements and 6 informational notes.

The contracts are written according to the latest standard used within the Ethereum community and the Solidity community's best practices. The naming of variables is very logical and understandable, which results in the contract being easy to understand. The code is very well documented. The developers provided us with a test suite as well as proper deployment scripts.

2.3 Findings

[FIXED] ~~[MEDIUM SEVERITY]~~ M.1 - Insufficient AccessControl check in hasRole function

Location: POOlsCreatorRegistry.sol - Line 110

Description:
The `hasRole` function is overwritten to use the NFT owner mechanism. However, the `hasRole` function returns `true` for an address that holds the `DEFAULT_ADMIN_ROLE` but does not hold the owner-NFT.
Even if there are no plans to grant multiple addresses the `DEFAULT_ADMIN_ROLE`, maintenance errors can occur and a defensive programming style should be promoted.
The check is insufficient because the function defaults to the original `hasRole` implementation in case the owner check fails.

The check is implemented as follows:

```
return role == DEFAULT_ADMIN_ROLE && owner() == account ||
       super.hasRole(role, account);
```

Let's say the address `account` holds the `DEFAULT_ADMIN_ROLE` but does not hold the corresponding owner-NFT. Therefore, the check evaluates to:

```
role == DEFAULT_ADMIN_ROLE && owner() == account || super.hasRole(role, account)
^^^^^^^^^^^^^^^^^^^^^^^^^^    ^^^^^^^^^^^^^^^^^^    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        true                 &&         false        ||    (not yet evaluated)
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                    false                            ||          true
                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                                        true
```

Recommendation:
There are two possible paths to handle this problem.

1. Make sure that there is only ever one address holding the `DEFAULT_ADMIN_ROLE` and that this address also always holds the owner-NFT. This requires overriding the `transferOwnership` and `_grantRole` functions to forbid transferring the owner-NFT and granting the `DEFAULT_ADMIN_ROLE`. Then implement a `changeAdmin` function that transfers the owner-NFT, grants the `DEFAULT_ADMIN_ROLE`, and revokes the role from the previous owner-NFT holder. This ensures that the `DEFAULT_ADMIN_ROLE` address is also always the owner-NFT holder.

2. Refactor the overridden `hasRole` function to:
```
return role == DEFAULT_ADMIN_ROLE ? owner() == account :
        super.hasRole(role, account);
```
   This ensures that for the `DEFAULT_ADMIN_ROLE` only the owner-NFT is checked.

Furthermore, it should be stated that this behavior should be documented accordingly.

Update on the 18th of August 2022:
The developers have fixed the issue according to our recommendation. Now, only the owner-NFT holder has the `DEFAULT_ADMIN_ROLE`. For all other roles, the regular access control mechanism is being used.

[FIXED] ~~[LOW SEVERITY]~~ L.1 - Unmet requirements in overwritten function

Location: POOlsCreatorRegistry.sol - Line 92 - 99

Description:
The ERC721 function `_isApprovedOrOwner` is expected to fail in case the `tokenId` does not exist (see OpenZeppelin's ERC721 implementation). The function is overridden in various contracts, but does not always fail in case the given `tokenId` does not exist.

The `_isApprovedOrOwner` function is implemented as follows:
```
function _isApprovedOrOwner(address spender, uint256 tokenId)
    internal view override returns (bool)
{
  return addressToUint256(spender) == tokenId ||
         super._isApprovedOrOwner(spender, tokenId);
}
```

The function returns `true` in case the `spender` and `tokenId` are equal. However, this does not indicate that the token actually exists.

The same also applies to `VestingFactory.sol` in line 104 - 111.

Recommendation:
Consider either ensuring that the token actually exists via a `require(super._exists(tokenId))` call prior to the return statement or by switching the two function calls in the return statement. This works because the `_isApprovedOrOwner` function implemented in the OpenZeppelin ERC721 implementation reverts in case the `tokenId` does not exist.

Update on the 18th of August 2022:
The developers have fixed the issue according to our suggestion.

[FIXED/ACK] ~~[LOW SEVERITY]~~ L.2 - Missing vanity checks for fees

Location: FeeManager.sol - Line 110 - 116

Description:
The fee variable can be changed via the `setFee` function. It is, however, not documented which denomination the fee is expected to be submitted in. Additionally, there is no upper limit, which could potentially lead to a fee value of > 100%.

Recommendation:
Consider documenting that the fee's denomination is in ether, i.e. `100% = 1 ether = 1e18`. Additionally, consider ensuring that the fee is less than or equal to 100% and introducing a "`public constant MAX_FEE`" variable to give users certainty about the maximum fee that they can expect.

Update on the 18th of August 2022:
The developers have introduced an upper limit for the fee of 100%. However, they only acknowledge but did not implement a public constant instead of using `1e18` inline. This could increase users' trust as the requirement would be more "accessible and understandable" for them.

[FIXED] ~~[NON CRITICAL]~~ NC.1 - Use of deprecated function

Location: Multiple

Description:
Throughout the codebase `AccessControl`'s `_setupRole` function is used. The function is deprecated in favor of `_grantRole`.

These are the affected occurrences:

- `P00lsCreatorRegistry::initialize`
- `Escrow::constructor`
- `AcutionFactory::constructor`
- `VestedAirdrops::constructor`
- `FeeManager::constructor`
- `UniswapV2Factory::constructor`

Recommendation:
Consider refactoring the `_setupRole` calls to `_grantRole` calls.

Update on the 18th of August 2022:
Each of the calls to `_setupRole` in the contracts within scope has been updated to using `_grantRole` accordingly.

[FIXED] ~~[NON CRITICAL]~~ NC.2 - Unnecessary use of the payable modifier

Location: Auction.sol - Line 108 - 115

Description:
The address argument is labeled as `payable`. As no ETH is sent to the address, the keyword is unnecessary.

Recommendation:
Consider removing the `payable` keyword if there are no plans to make use of it.

Additionally, you could consider sending ETH to the user in case of WETH (if the payment token is WETH). This could improve the UX because the user does not need to manually unwrap the WETH after leaving the auction. In this case, the address would need to be `payable`.

Update on the 18th of August 2022:
The `payable` modifier has been removed.

The developers followed up with us on our recommendation to consider sending ETH back to the user instead of WETH, which would require either adapting the `receive()` function of the contract to cover multiple purposes or using the WETH10 deployment which is not widely used yet. As this decision has no security implications, we fully leave the decision to the developers.

[FIXED] ~~[NON CRITICAL]~~ NC.3 - Incorrect version pragma in library

Location: UniswapV2Library.sol - Line 2

Description:
The `UniswapV2Library` has been refactored with the assumption of Solidity's builtin over/underflow protection. However, the version pragma is still set to `>=0.5.0`, allowing Solidity versions without the necessary protections.

In the case of this specific project, the pragma is fine because the `UniswapV2Factory`, which uses the library, expects a version of at least `0.8.0`. However, the pragma should be adjusted anyway to make the contract correct on its own.

Additionally, the `P00lsDAO.sol` contract uses a version pragma of `^0.8.2` instead of

`^0.8.0` like the rest of the contracts.

Recommendation:
Consider updating the version pragma of the library to `^0.8.0` as well.

Update on the 18th of August 2022:
The version pragma of the two contracts has been updated to `^0.8.0` accordingly.

[ACK] ~~[NON CRITICAL]~~ NC.4 - Unspecific Compiler Version Pragma

Location: Multiple

Description:
Avoid floating pragmas for non-library contracts.

While floating pragmas make sense for libraries to allow them to be included with multiple different versions of applications, it may be a security risk for application implementations. A known vulnerable compiler version may accidentally be selected or security tools might fall back to an older compiler version ending up checking a different EVM compilation that is ultimately deployed on the blockchain.

Recommendation:
Consider pinning a concrete compiler version, as this is the best practice.

Update on the 18th of August 2022:
The developers acknowledge our finding. The developers stated that they are making sure to use the latest version of the compiler. Additionally, we see no immediate security issues.

[FIXED] ~~[NON CRITICAL]~~ NC.5 - Undocumented API change

Location: UniswapV2Library.sol - Line 18 - 26

Description:
The internal API of the forked `UniswapV2Library` has been changed inside the `pairFor` function. While the original function always returns the (*pre-calculable*) pool address, even if the pool does not exist yet, the forked version returns the `zero` address in case the pool does not exist.

```
// Inside the forked UniswapV2Library::pairFor function.
return Address.isContract(predicted) ? predicted : address(0);
```

Due to this change, the `UniswapV2Library::getReserves` function also changes its behavior compared to the original.

```
// Inside the forked UniswapV2Library::getReserves function.
(uint reserve0, uint reserve1,) = IUniswapV2Pair(pairFor(factory,
tokenA, tokenB)).getReserves();
```

In case a token pair does not exist, the original UniswapV2 would fail due to calling the token pair's pre-calculated address, while the forked version fails due to calling the zero address.

This behavior change is further exposed via the external API inside the `UniswapV2Router02` functions.

Recommendation:
The overall aim of the Uniswap fork should be to not introduce unnecessary API changes. It is therefore suggested to change the `UniswapV2Library::pairFor` function to always return the pre-calculated address. Afterward, the `UniswapV2Factory::getPair` function would need to be adjusted to only return the pair address if it exists already, and otherwise return the zero address.

Update on the 18th of August 2022:
The developers have updated the `UniswapV2Library::pairFor` and `UniswapV2Factory::getPair` functions to be functionally equivalent to the original Uniswap implementation.

[FIXED] ~~[NON CRITICAL]~~ NC.6 - Unnecessary require statement

Location: UniswapV2Factory.sol - Line 38 - 46

Description:
The `createPair` function starts with the following three lines:

```
require(tokenA != tokenB, 'UniswapV2: IDENTICAL_ADDRESSES');
(address token0, address token1) = UniswapV2Library.sortTokens(tokenA,
                                    tokenB);
require(token0 != address(0), 'UniswapV2: ZERO_ADDRESS');
```

However, the `UniswapV2Library::sortTokens` function includes both checks already:

```
function sortTokens(address tokenA, address tokenB) internal pure
         returns (address token0, address token1) {
  require(tokenA != tokenB, 'UniswapV2Library: IDENTICAL_ADDRESSES');
  (token0, token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB,
                     tokenA);
  require(token0 != address(0), 'UniswapV2Library: ZERO_ADDRESS');
}
```

Note that while the original version includes the `require` statements, it does not use the library's `sortTokens` function but rather sorts the tokens in-line:

```
require(tokenA != tokenB, 'UniswapV2: IDENTICAL_ADDRESSES');
(address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) :
                                   (tokenB, tokenA);
require(token0 != address(0), 'UniswapV2: ZERO_ADDRESS');
```

The `require` statements inside the `createPair` function can therefore be removed.

Recommendation:
Consider removing the unnecessary `require` statements inside the `createPair` function to reduce gas costs.

Update on the 18th of August 2022:
The developers have removed the `require` statement according to our recommendation.

[ACK] ~~[NON CRITICAL]~~ NC.7 - Functions are susceptible to MEV

Location: Multiple

Description:
The following functions are MEV-able if run through the public mempool and should therefore only be run through a private mempool like Flashbots:

- - `AuctionFactory::finalize`
- - `FeeManager::redistributedFees`

Note: *This finding is just listed as a piece of information. There is no fix required.*

Update on the 18th of August 2022:
The developers have acknowledged this informational finding.

[ACK] ~~[NON CRITICAL]~~ NC.8 - Inconsistent use of safe- vs non-safe variants

Location: AuctionFactory.sol - Line 87 - 88

Description:

OpenZeppelin's `SafeERC20::safeApprove` function should be used whenever the exact ERC20 token to approve is unknown during development time and/or the ERC20 token does not follow the ERC20 specification the same way as most ERC20 tokens do.

The current codebase uses the `SafeERC20::safeApprove` function even if the token is known to work fine with the standard `ERC20::approve` function (e.g. `FeeManager::_liquidateAllLP`) with the exception inside the `AuctionFactory::finalize` function:

```
payment.approve(address(router), type(uint256).max);
token.approve(address(router), type(uint256).max);
```

Recommendation:

For the sake of consistency and increased security during future development, use OpenZeppelin's `SafeERC20::safeApprove` function inside the `Auction::finalize` function.

Update on the 18th of August 2022:

The developers have refactored the non-safe approval calls to their safe counterparts accordingly.

2.4 Gas Optimizations

[Gas Optimization] **GO.1 - Use msg.sender instead of owner() if applicable**
Location: Throughout the project

Description:
If a function verifies that the `msg.sender` is the address returned by `owner()`, it saves gas to use `msg.sender` from there on instead of recalling the `owner()` function.

The affected occurrences are:
- `RegistryOwnable::transferOwnership`
- `RegistryOwnableUpgradeable::transferOwnership`
- `VestingFactory::transferOwnership`

Recommendation:
Consider making use of `msg.sender` instead of calling the `owner()` functions in the applicable cases.

[Gas Optimization] **GO.2 - Optimize loop to be more efficient**
Location: UniswapInterfaceMulticall.sol - Line 29 - 36

Description:
The loop is implemented as follows:

```
for (uint256 i = 0; i < calls.length; i++) { /*...*/ }
```

A more gas optimized version is:

```
uint len = calls.length;
for (uint256 i; i < len; ++i) { /*...*/ }
```

Note: *This optimization is invalid if the new `--via-ir` compiler pipeline is used.*

Recommendation:
Consider optimizing the for-loop to be more efficient in terms of its gas usage.

[Gas Optimization] **GO.3 - Functions can be external instead of public**
Location: POOlsTokenXCreatorV2.sol - Line 66 + 76

Description:
The functions are never called internally in the contract:
- `P00lsTokenXCreatorV2::onEscrowRelease`
- `P00lsTokenXCreatorV2::convertToAssetsAtBlock`

Recommendation:
Consider changing the functions visibility from `public` to `external` in order to save some gas.

2.5 Informational Notes

[Informational] **IN.1 - Inconsistent Code Style in POOlsTokenCreator::claim**
Location: POOlsTokenCreator.sol - Line 43 - 52

Description:
The `claim` function checks that the account did not claim tokens yet via:

```
require(!__claimedBitMap.get(index), "P00lsTokenCreator::claim: drop already claimed");
```

Given the code style used throughout the project, the `P00lsTokenCreator::isClaimed` function should maybe be `public` and then using that function instead of the inlined access via `__claimedBitMap.get(index)` statement.

[Informational] **IN.2 - Inconsistent Usage of uint vs uint256 in UniswapV2Pair**
Location: UniswapV2Pair.sol

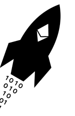
Description:
While the project uses the `uint256` type declaration, the UniswapV2 forked codebase uses `uint`. This is fine, but the variable `UniswapV2Pair::MINIMUM_LIQUIDITY` was changed from `uint` to `uint256`.

In order to stay consistent, it is suggested to change the declaration again to `uint256`.

[Informational] **IN.3 - Files containing more than one contract**
Location: Multiple

Description:
For clarity's sake it would make sense to split the `VestingFactory.sol` and the `RegistyOwnable.sol` files into multiple sub-files to contain a single contract in each of them.

- `VestingFactory.sol` → `VestingFactory.sol` + `VestingTemplate.sol`
- `RegistyOwnable.sol` →
  - → `RegistyOwnable.sol` + `RegistyOwnableUpgradeable.sol`



[Informational] IN.4 - Unnecessary use of the virtual keyword

Location: VestingFactory.sol

Description:

The `owner`, `transferOwnership`, `cliff`, and `delegate` functions are declared `virtual`, although there is no apparent reason to do so.

[Informational] IN.5 - Typing Errors

Location: Multiple

Description:

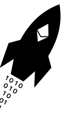
- `AuctionFactory` contract documentation
    - Otherwize → Otherwise
- `VestedAirdrops::release`
    - reverts it proof is invalud → reverts if proof is invalid
- `FeeManager::redistributedFees`
    - there might me more… → there might be more…
- `Escrow::_configure`
    - if previous is schedule is… → if previous schedule is…

[Informational] IN.6 - Unused Modifier

Location: POOlsTokenXCreatorV2.sol - Line 20 - 23

Description:

The `onlyOwner` modifier in the `P00lsTokenXCreatorV2` is never used within the contract and could therefore be removed.



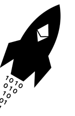
3. Protocol/Logic Review

Part of our audits are also analyses of the protocol and its logic. The byterocket team went through the implementation and documentation of the implemented protocol.

The repository itself contained tests and documentation. We found the provided unit tests that are coming with the repository execute without any issues and cover the most important parts of the protocol.

According to our analysis, the protocol and logic are working as intended, given that any findings with a severity level are fixed. When making use of the Mainnet forking method, we were able to successfully execute the protocol.

We were **not able to discover any additional problems** in the protocol implemented in the smart contract.



4. Summary

During our code review (*which was done manually and automated*), we **found 11 bugs or flaws, with 11 of them being fixed** in a subsequent update**.** Prior to this, there have been 8 non-critical, 2 of low severity and 1 of medium severity findings. Our automated systems and review tools did **not find any additional ones.** Additionally, we found **3 gas improvements** and **6 informational notes**.

The protocol review and analysis did neither uncover any game-theoretical nature problems nor any other functions prone to abuse besides the ones that have been uncovered in our findings.

In general, there are some improvements that can be made, but we are **very happy** with the overall quality of the code and its documentation. The developers have been very responsive and were able to answer any questions that we had.