PUBLIC

Code Assessment

of the V3 Whitelister Smart Contracts

April 03, 2024

Produced for

METHLAB

by

CHAINSECURITY

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 7
4 Terminology 8
5 Findings 9
6 Informational 10
7 Notes 12

1 Executive Summary

Dear MethLab team,

Thank you for trusting us to help you with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of V3 Whitelister according to Scope to support you in forming an opinion on their security risks.

MethLab implements a modified version of Uniswap V3, which introduces a whitelist for LP token minting.

The most critical subjects covered in our audit are functional correctness and access control. Security regarding both subjects is high.

A general subject covered was Event Handling. Event handling is improvable, as the Whitelister does not emit events for some owner operations. See Missing Events for Owner Operations.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the V3 Whitelister repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V3 Whitelister

V	Date	Commit Hash	Note
1	28 Feb 2024	8c228525f262314fc248857424d0e404a99fa6bc	Initial Version

V3 Core Whitelister

V	Date	Commit Hash	Note
1	28 Feb 2024	04946adf494cdd7c605a4d11c39db7d4d7eb5d32	Initial Version

For the solidity smart contracts, UniswapV3Pool uses compiler version `0.7.6` and WhitelisterV3 uses `0.8.19`.

The following files were in scope of this review:

- WhitelisterV3.sol
- UniswapV3Pool.sol (only modifications)

Note that for the UniswapV3Pool, only the modifications made to the original UniswapV3Pool were reviewed, assuming that the original contract was correct.

2.1.1 Excluded from scope

All other files, including all files that were taken from Uniswap unmodified.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

MethLab offers V3 Whitelister, a fork of Uniswap V3 that adds a whitelist for Liquidity providers. Only whitelisted addresses can deposit liquidity. The whitelist can be toggled on or off per pool and is enabled for all pools by default.

2.2.1 WhitelisterV3

The WhitelisterV3 is a contract that allows the owner to add and remove addresses from a whitelist. There is a separate whitelist for every pool.

The `isAllowed()` view function is called by a UniswapV3Pool and returns true when the LP passed in `data` is allowed to mint and the `recipient` of the LP tokens is the `NonFungiblePosManager`.

The owner role can toggle the whitelist on or off for each pool (`setPoolWhitelist()`), or set the `ENABLED` flag to `false` to disable the whitelist for all pools (`setContractState()`). When `ENABLED` is set to `false`, the system will behave exactly like Uniswap V3 without modifications. The owner can add addresses to the whitelist of a pool by calling `setLPWhitelist(address _pool, address _lp, bool _status)`.

When the whitelist is enabled for a pool, it is enforced that all calls to `mint()` in UniswapV3Pool must specify the exact `NonFungiblePosManager` as `recipient` that is set in WhitelisterV3. This means the whitelisted LPs must use this PositionManager, and cannot use a different one.

The WhitelisterV3 is initialized with a `Factory` and a `NonFungiblePosManager` once, and they cannot be changed later.

The ownership of the WhitelisterV3 can be transferred to another address by calling `transferOwnership()`. The ownership can be renounced by calling `renounceOwnership()`, but only if the `ENABLED` flag is set to false. This can be used to assure users that the whitelists will stay disabled forever, once they have been disabled.

2.2.2 UniswapV3Pool

The UniswapV3Pool has been modified from Uniswap with a single code change:

```
 function mint(
    [...]
) external override lock returns (uint256 amount0, uint256 amount1) {
    require(amount > 0);

    // check if the LP is whitelisted or not
    require(IWhitelister(whitelister).isAllowed(recipient, data), '!W');
```

The `mint()` function now calls the WhitelisterV3 (a constant address) and checks the return value of `isAllowed()`. If the LP is not whitelisted, the function will revert. This ensures that only whitelisted LPs can mint.

All other contracts are taken 1-to-1 from Uniswap V3 and are not modified.

2.2.3 Trust model

The owner of the WhitelisterV3 can modify the whitelist and toggle it on or off for each pool. In the worst case, it could allow all addresses to mint to a pool that should be restricted, or restrict addresses from minting that should be allowed.

The addresses on the whitelist can mint LP tokens using the NonFungiblePositionManager. These positions can then be transferred to other addresses, even if they are not on the whitelist. The whitelisted addresses are assumed to only transfer positions to addresses that should be allowed to receive them. The contract does not restrict this.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe our findings. The findings are split into these different categories:

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

6.1 Floating Solc and Dependency Versions

Informational **Version 1** **Acknowledged**

CS-DFWL-001

MethLab uses a floating pragma solidity ^0.8.19, and the solc version is not specified in the configuration file (*foundry.toml*). Contracts should be deployed with the same compiler version and flags that have been used during testing and audit. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

In addition, the dependency (openzeppelin contracts) used is not fixed to a specific commit. With new versions being pushed to the dependency registry, the compiled smart contracts can change. This may lead to incompatibilities with older compiled contracts. If the imported and parent contracts change the storage slot order or change the parameter order, the child contracts might have different storage slots or different interfaces due to inheritance.

Both types of floating versions can lead to issues when trying to recreate the exact bytecode that was deployed.

6.2 Hardcoded Address

Informational **Version 1** **Acknowledged**

CS-DFWL-002

The address of the WhitelisterV3 is hardcoded in the modified UniswapV3Pool contract.

It should be ensured that the address is set to the correct address before deployment.

In particular, it should be noted that the address may differ if the contracts are deployed to a different chain.

6.3 Missing Events for Owner Operations

Informational **Version 1** **Acknowledged**

CS-DFWL-003

No events will be emitted in the following owner operations:

```
function initializeV3Address
function setContractState
function setPoolWhitelist
function setLPWhitelist
```

Events can notify off-chain observers of important state changes on the contract.

6.4 Users Can Donate to NonFungiblePosManager by Interacting With UniswapV3 Pool Directly

Informational **Version 1** **Acknowledged**

CS-DFWL-004

The WhitelisterV3 will ensure that the recipient can only be the NonFungiblePosManager, and that lps are whitelisted.

```
require(_recipient == NonFungiblePosManager, "Invalid Callee");
(, address lp) = abi.decode(_data, (PoolKey, address));
if (!isLPWhitelisted[msg.sender][lp]) return false;
```

However, a user can circumvent the whitelist to deposit into any ticks they want, by directly calling `mint()` on the UniswapV3 pool with NonFungiblePosManager as the recipient and passing any whitelisted address as lp. This is a donation to the NonFungiblePosManager. As the NonFungiblePosManager is not aware of the donation, the donated funds will be locked.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 LP Tokens Can Be Transferred

Note Version 1

The WhitelisterV3 ensures that only whitelisted addresses can mint LP tokens. However, once LP tokens have been minted using the NonFungiblePositionManager, they can also be transferred to other addresses. These transfers are unrestricted.

Any LP that is on the whitelist can mint and then transfer LP tokens to any address, even if that address is not on the whitelist.

Also note that if an LP was on the whitelist once, but is removed, they will still own the LP tokens they minted while they were whitelisted. Only future mints will be restricted in this case.