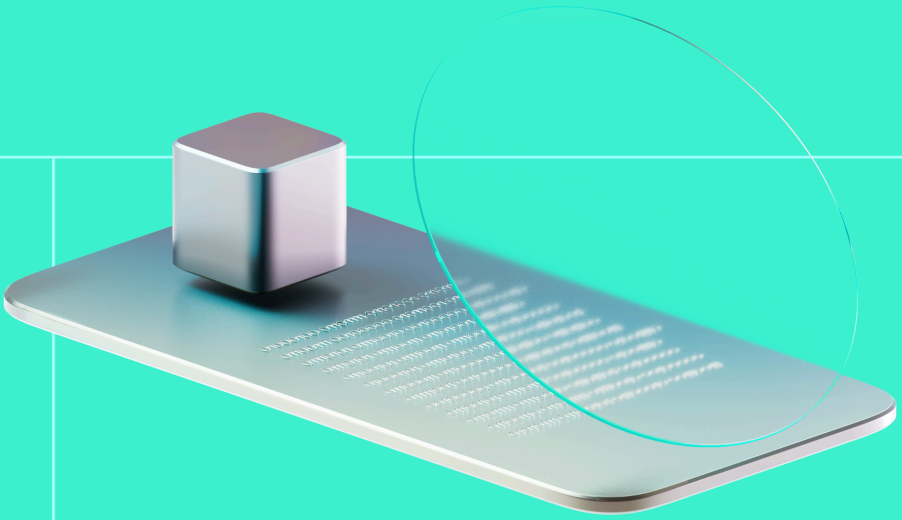
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Ociswap

Date: 22 May, 2024

Hacken OU
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia

We thank Ociswap for allowing us to conduct a Smart Contract Security Assessment. This document outlines our methodology, limitations, and results of the security assessment.

Ociswap is a top-tier solution, combining greatest DEX methodologies into a novel, high-performance system, catalyzing decentralized finance in the Radix ecosystem.

Platform: Radix DLT

Language: Rust, Scrypto

Tags: DEX, Flash Loans, Liquidity Pools

Timeline:

1st review 06.11.2023 - 21.11.2023

2nd review 08.05.2024 - 18.05.2024

Methodology: Link

Last review scope

Repositories	Initial - https://github.com/ociswap/ociswap-blueprints Final - https://github.com/ociswap/precision-pool
Commit	a3f458b

View full scope

Audit Summary

10/10	10/10	100%	10/10
Security score	Code quality score	Test coverage	Documentation quality score

Total: 10.0/10

🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩

The system users should acknowledge all the risks summed up in the risks section of the report.

5	4	1	0
Total Findings	Resolved	Acknowledged	Mitigated

Findings by severity	Findings Number	Resolved	Mitigated	Acknowledged
Critical	0	0	0	0
High	0	0	0	0
Medium	0	2	0	0
Low	0	2	0	1

Document

Name	Smart Contract Code Review and Security Analysis Report for Ociswap
Approved By	Grzegorz Trawiński | SC Audits Expert at Hacken OÜ
Audited By	Jakub Heba | SC Auditor at Hacken OÜ Vladyslav Khomenko | SC Auditor at Hacken OÜ
Website	https://ociswap.com
Changelog	20.11.2023 – Preliminary Report 22.05.2024 - Remediation Check and Second Review results

Introduction

Hacken OÜ (Consultant) was contracted by Ociswap (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

Ociswap Precision Pool - designed with the influence of UniswapV3's AMM algorithm. It lets users provide liquidity in specific price ranges, maximizing liquidity efficiency and thus increasing their rewards. For users that want to swap their tokens it also means less price slippage.

LPs can focus their capital in Ociswap within a custom price range, offering more liquidity at targeted prices. By doing this, LPs create customized price curves that represent their personal capital preferences.

LPs can offer the same level of liquidity depth as Basic Pool within predetermined price ranges with far less money but at greater risk by concentrating their liquidity. For LPs it means increased exposure and therefore greater trading fees.

The creator of the pool is able to specify hooks that are going to be called before and after such events:

- New pool creation
- Swap
- Liquidity being provided/retrieved

Each such event can have multiple hooks. Hooks are methods on a given component that comply with a certain interface. The hook-powered design of the system enables:

- Customized on-chain oracles
- Time-weighted price oracles
- Dynamic fees based on volatility or other factors
- On-chain limit orders

Other parts of the system:

- Registry - developer-owned component, designed to collect the protocol fee.

Privileged roles

- blueprint - role assigned to the component itself, used for metadata setting, as well as executing hook after instantiation.

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements are not finished.
- Technical description is available, as well as Uniswap V3 whitepaper is linked as an extension.
- Code has very rich comments.

Code quality

The total Code Quality score is **10** out of **10**.

- The code is well formatted.

Test coverage

Code coverage of the project is **100%** (functional coverage).

- The code is thoroughly tested.
- Deployment and basic user interactions are covered with tests.

Security score

As a result of the audit, the code contains **2** medium and **4** low severity issues. The security score is **8** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **10.0**. The system users should acknowledge all the risks summed up in the risks section of the report.

Risks

- The system is designed to call external code, which can have an impact on the fundamental pieces of the system: fees and tokens, involved in swap. If a vulnerable or compromised component were to have access, it could pose a risk of mishandling mentioned parts of the system.
- Swap fees that make up LP rewards are dynamic and can be modified by external components (via hooks) during the swap process which is outside of the audit scope.

Findings

■■■■ Critical

No critical severity issues were found.

■■■ High

No high severity issues were found.

Medium

M01. input_fee_rate state variable has insecure upper bounds

Impact	High
Likelihood	Low

One of the variables that determines the pool state is the *input_fee_rate* value. It is used to calculate the fee charged by the pool during a *swap* operation.

```
assert!(
    input_fee_rate.between_zero_and_one(),
    "Input fee rate must be between zero and one!"
);
```

However, this value can be anything between <0.1> range. This means that it is possible to impose a fee of up to 100%, which will result in the loss of a large amount of funds by an unaware user of the pool.

It is good practice to limit the possibility of imposing too high a fee by referring to recognized protocols such as Uniswap. It is worth noting that the *input_fee_rate* value may also change after the hook executes - so the user may not be aware of the actual fee while performing an action in the pool.

Path: ./common/src/math.rs : between_zero_and_one()

Recommendation: It is suggested to adjust the maximum fee cap to a reasonable value that is a fair fee for using the pool. Additionally, make sure that both *BeforeSwap* and *AfterSwap* hooks will not change the fee range to some unexpected value during the *swap* execution.

Found in: d0889a2

Status: Fixed (Revised commit: a3f458b)

Remediation: Maximum fee rate constant (*INPUT_FEE_RATE_MAX*) was implemented, as well as a function (*assert_input_fee_rate_is_valid*) responsible for asserting that the fee rate in the state is between *<0,0.1>* bounds.

M02. Protocol's cut of the fee is unlimited

Impact	High
Likelihood	Low

During each *swap*, the pool charges a certain number of fees for processing the transaction. From this fee, the protocol takes a certain part for itself, via the *sync_registry* operation. The fee mechanism of the protocol is configured in the registers. The *fee_protocol_share* variable specifies how much the pool's fee will be collected as a protocol fee.

The problem occurs if the protocol owner changes this value to a very high one, deducting a larger part of the fee. Such an action may make being a liquidity provider unprofitable.

```
pub fn update_config(
    &mut self,
    fee_protocol_share: Decimal,
    sync_period: u64,
    sync_slots: u64
) {
    self.fee_protocol_share = fee_protocol_share;
    self.sync_period = sync_period;
    self.sync_slots = sync_slots;
}
```

Path: ./ociswap-blueprints/registry/src/registry.rs : update_config()

Recommendation: It is recommended to set a ceiling for *fee_protocol_share* so that the protocol owner cannot increase this value to an unfairly high level.

Found in: d0889a2

Status: Fixed (Revised commit: a3f458b)

Remediation: Maximum fee rate constant (*FEE_PROTOCOL_SHARE_MAX*) was implemented, as well as a function (*assert_fee_rate_within_bounds*) responsible for asserting that the fee rate in the state is between *<0,0.25>* bounds.

Low

L01. Variables with hardcoded values might lead to unexpected side issues

Impact	Low
Likelihood	Low

The *BeforeSwapState* and *AfterSwapState* structures in Precision Pool have among others two variables - *x_active_amount* and *y_active_amount*. While in Basic Pool they store token amounts in both vaults, in Precision Pool they are statically set to zero.

If the *BeforeSwap* or *AfterSwap* hook logic does not take this into account, it can lead to unexpected problems and side issues.

```
fn before_swap_state(&self, swap_type: SwapType) -> BeforeSwapState {
    BeforeSwapState {
        price_sqrt: self.price_sqrt,
        active_liquidity: self.active_liquidity,
        x_active_liquidity: dec!(0), // todo fix
        y_active_liquidity: dec!(0), // todo fix
        swap_type,
        input_fee_rate: self.fee_input_rate,
        fee_protocol_share: self.fee_protocol_share,
    }
}
```

Path: ./ociswap-blueprints/concentrated_pool_v1/src/pool.rs : before_swap_state(), pool_math.rs : from() { AfterSwapState }

Recommendation: The values of the *x_active_amount* and *y_active_amount* variables in the Precision Pool should be adjusted or eliminated from the *BeforeSwapState* and *AfterSwapState* structures, bearing in mind that they will still be present in the contract state.

Found in: d0889a2

Status: Fixed (Revised commit: a3f458b)

Remediation: Both *x_active_amount* and *y_active_amount* are not used in the codebase logic anymore.

L02. Wrong metadata generation

Impact	Low
Likelihood	Low

While analyzing the code of Precision Pool, it was noticed that in the *Instantiation* operation, when calling the *set_lp_manager* function, metadata about the pool is written incorrectly, probably by duplicating and not updating the code coming from the Basic Pool contract.

When pattern matching returns *None*, the string is created without inserting the *pair_symbol* variable, but the string contains "Basic Pool" instead of "Precision Pool".

While this does not pose an immediate risk to the protocol, components that use metadata may behave in inappropriate and unexpected ways. For example, an off chain service which aggregates pools based on its type might be misled if it receives not-well formatted information from the contract metadata.

```
fn set_lp_manager(
    pool_address: ComponentAddress,
    x_address: ResourceAddress,
    y_address: ResourceAddress,
    dapp_definition: ComponentAddress
) -> ResourceManager {
    let x_symbol = token_symbol(x_address);
    let y_symbol = token_symbol(y_address);
    let (name, description) = match
        x_symbol.zip(y_symbol).map(|(x, y)| format!("{}/{}", x, y))
    {
        Some(pair_symbol) => {
            (
                format!("Ociswap LP {}", pair_symbol).to_owned(),
                format!("Ociswap LP token for the {} Concentrated Pool",
pair_symbol).to_owned(),
            )
        }
        None => ("Ociswap LP".to_owned(), "Ociswap LP token for the Basic
Pool".to_owned()),
    };
    let tags = vec![
        "ociswap".to_owned(),
        "liquidity-pool".to_owned(),
        "lp".to_owned(),
        "dex".to_owned(),
        "defi".to_owned()
    ];
    let dapp_definition_global: GlobalAddress = dapp_definition.into();
[..]
```

Path: ./ociswap-blueprints/concentrated_pool_v1/src/pool.rs : set_lp_manager()

Recommendation: It is suggested to adjust the metadata to the values associated with the Precision Pool.

Found in: d0889a2

Status: Fixed (Revised commit: a3f458b)

Remediation: Vulnerable function returns proper pool type now, if *None* branch is executed.

L03. Floating language version

Impact	Low
Likelihood	Low

It is preferable for a production project, especially a smart contract, to have the programming language version pinned explicitly. This results in a stable build output, and guards against unexpected toolchain differences or bugs present in older versions, which could be used to build the project.

The language version could be pinned in automation/CI scripts, as well as proclaimed in README or other kinds of developer documentation. However, in the Rust ecosystem, it can be achieved more ergonomically via a rust-toolchain.toml descriptor (see https://rust-lang.github.io/rustup/overrides.html#the-toolchain-file)

Path: *

Recommendation: It is suggested to set a concrete Rust version.

Found in: d0889a2

Status: Accepted

Remediation: The compiler version is already pinned through the Scrypto toolchain using Deterministic Builder.

Informational

I01. Vulnerable dependencies

Few contracts and libraries use packages with publicly known vulnerabilities, which is considered a deviation from leading security practices. Vulnerable packages may have uncertain impact on implemented functionalities.

```
Crate:      ed25519-dalek
Version:    1.0.1
Title:      Double Public Key Signing Function Oracle Attack on `ed25519-dalek`
Date:       2022-06-11
ID:         RUSTSEC-2022-0093
URL:        https://rustsec.org/advisories/RUSTSEC-2022-0093
Solution:   Upgrade to >=2
Dependency tree:
ed25519-dalek 1.0.1
[..]
```

Path: Multiple Cargo.toml files

Recommendation: It is recommended to verify that none of the vulnerable functions are used in the code, or update the package to a higher, secure version.

Found in: d0889a2

Status: Accepted

Remediation: The vulnerable dependency is used by the SDK and is only used in tests. The contract does not have vulnerable dependencies.

I02. Multiple TODO comments

It was identified that in many places in the code there are comments indicating that the contract logic is not completed at this point - marked as "TODO". While they do not pose an immediate threat, their existence in production code is considered bad security practice because it can help a would-be attacker map interesting attack vectors onto the protocol.

Occurrences throughout the codebases:

- ./concentrated_pool_v1/src/pool_math:rs#248, 249
- ./concentrated_pool_v1/src/pool:rs#699, 700

Recommendation: It is recommended to eliminate all "TODO" comments.

Found in: d0889a2

Status: Fixed (Revised commit: a3f458b)

Remediation: TODO comments were removed from the codebase.

I03. Usage of debugging macros throughout the codebases

When creating code, functions that support developers in returning the values of state variables and comparing them with the expected results are very helpful. Examples of such functions/macros are *debug!()* and *info!()*, which display strings and variable values in the console.

While this is very helpful during code development, in a production version of the solution, these types of calls should be removed. They affect the readability of the code and may be negatively perceived by developers creating solutions based on the protocol code.

Path: *

Recommendation: It is suggested to eliminate the *debug!()* and *info!()* macros in contract and library codes in production code.

Found in: d0889a2

Status: Fixed (Revised commit: a3f458b)

Remediation: All mentioned macros were removed from the codebase.

I04. Missing useful utility functions

The fees in the swap pool are not fixed. In fact, they can be changed at any moment, as stated in M01 and M02. It can be useful for users and liquidity providers to be able to check the fees at a given time before doing swap.

Path: ./common/src/basic_pool.rs

Recommendation: Since the fees are dynamic, we suggest creating utility methods for checking the fees, for example, *check_fee()* to calculate the swap fee before swapping and *check_lp_fee()* for liquidity providers to see their cut of the revenue.

Found in: d0889a2

Status: Fixed (Revised commit: a3f458b)

Remediation: Three getters were introduced, allowing for input fee, protocol share fee, as well as flash loan fee verification.

I11. Unformatted Code

The tool *cargo fmt --check* reports that code is not formatted.

Path: *

Recommendation: It is suggested to format the code using *rustfmt* or an equivalent.

Found in: d0889a2

Status: Fixed (Revised commit: a3f458b)

Remediation: Formatter is not returning any issues now.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	Critical	High	Medium
Medium Likelihood	High	Medium	Low
Low Likelihood	Medium	Low	Low

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope details

Repositories	https://github.com/ociswap/ociswap-blueprints
Commits	Initial - d0889a2 Final - a3f458b
Requirements	Link
Technical Requirements	Link

Contracts in Scope

./concentrated_pool_v1/src/constants.rs
./concentrated_pool_v1/src/lib.rs
./concentrated_pool_v1/src/pool.rs
./concentrated_pool_v1/src/pool_math.rs
./concentrated_pool_v1/src/utils.rs
./common/src/hooks.rs
./common/src/lib.rs
./common/src/math.rs
./common/src/pools.rs
./common/src/time.rs
./registry/src/lib.rs
./registry/src/registry.rs