Audit Report

FortyTwo

v1.1

June 5, 2023

Table of Contents

License

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security has been engaged by Hitchhikers Incorporated to perform a security audit of FortyTwo.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/fortytwomoney/modules
Commit	`283bf670fd48c5732ffa359fb88945f8af058ac8`
Scope	All contracts were in scope.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

FortyTwo is a decentralized multichain yield optimizer for the Cosmos ecosystem, designed to help users earn passive income on their crypto holdings. It employs investment strategies to automatically optimize rewards from liquidity pools, automated market-making projects, and various DeFi yield farming opportunities.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending, Acknowledged**, or **Resolved**.

Note that audits are an important step to improving 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 the 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**. We include a table with these criteria below.

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

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	**Medium-High**	The codebase relies on heavy integration with Abstract's contracts, SDK, and APIs which are out-of-scope of this audit.
Code readability and clarity	**Low-Medium**	There are outstanding TODO comments and debug lines in the codebase, which suggests the codebase is still under development.
Level of documentation	**Medium**	No sufficient documentation was provided during the audit.
Test coverage	**Low-Medium**	-

Summary of Findings

No	Description	Severity	Status
1	Autocompounder is vulnerable to share inflation attack	Critical	Resolved
2	Attackers can cause batch unbondings to fail, preventing users from unstaking liquidity pool tokens	Critical	Resolved
3	Funds in the proxy contract may be unintentionally forwarded to others	Critical	Resolved
4	Deposit and withdrawal fees are not charged	Critical	Resolved
5	The compound function is susceptible to sandwich attack	Major	Resolved
6	Incorrect cooldown condition locks unbonding	Major	Resolved
7	Unbonding cooldowns are not respected	Major	Resolved
8	Compounding will fail for zero performance fees	Minor	Resolved
9	Hard-coded slippage value makes deposits susceptible to sandwich attacks	Minor	Resolved
10	Non-updatable configuration can not reflect changes in Staking contract	Minor	Resolved
11	Minting of zero vault tokens possible	Minor	Resolved
12	Unused variable in codebase	Informational	Resolved
13	Lack of validation upon deposit lead to inefficiencies and locks additional funds	Informational	Resolved
14	Remove TODO comments	Informational	Resolved
15	Unused commented code in codebase	Informational	Resolved
16	Misleading variable name when withdrawing liquidity	Informational	Resolved
17	`calculate_withdrawals` sets expiration to current block height which may become problematic if the function is exposed in a future upgrade	Informational	Resolved

18	Users cannot query fees through smart queries	Informational	Resolved
19	Panicking macros and debugging code	Informational	Resolved
20	Overflow checks not enabled for release profile	Informational	Resolved
21	Additional funds sent to the contract are lost	Informational	Resolved

Detailed Findings

1. Autocompounder is vulnerable to share inflation attack

Severity: Critical

The compute_mint_amount function in contracts/autocompounder/src/handlers/reply.rs:130 is vulnerable to a share inflation attack. A share inflation attack represents a scenario where a malicious actor artificially inflates the supply of tokens, potentially manipulating the token's value and diluting other holders' shares.

Currently, the function computes the mint amount by using integer division. Due to the nature of integer division, results are always floored. This allows an attacker to inflate their current shares while stealing funds from unsuspecting users.

The compute_mint_amount is susceptible to this attack because the denominator of the integer division staked_lp can be manipulated for Astroport, representing the total staked amount of the proxy address.

An example attack scenario:

1. The attacker makes the first deposit of 1 token and thus receives one share.
2. The victim wants to deposit funds into the protocol, but the attacker front-runs it. For simplicity, let's assume that the liquidity pool tokens deposited by the victim will be 10_000.
3. The attacker provides liquidity such that the liquidity pool tokens amount is 5001 ((10_000/2) + 1). To do this, they call the ProvideLiquidity message in the Astroport contract while specifying auto_stake as true and the receiver as the proxy contract's address, causing the generator to stake on behalf of the proxy contract.
4. This causes the staked_lp value from the query_stake function to include the liquidity provided by the attacker, which is 5002 (5001+1).
5. The victim's transaction is processed. When minting the vault shares, the compute_mint_amount function will be evaluated as 1.99 (1 * (10000 / 5002)), which rounds down to 1. Consequently, the victim only receives one share.
6. Finally, the attacker redeems their shares. The computed lp_tokens_withdraw_amount value will be evaluated to 7501 ((1/2) * 15002), earning the attacker an extra 2499 (7501-5002) liquidity pool tokens.

Recommendation

We recommend enforcing a minimum amount that needs to be met in the first deposit. This will greatly increase the cost of orchestrating a share inflation attack. Note that it is common practice to mint “dead shares” to the protocol to increase the cost of the attack further. For more details, please see Astroport’s implementation.

Status: Resolved

2. Attackers can cause batch unbondings to fail, preventing users from unstaking liquidity pool tokens

Severity: Critical

In `contracts/autocompounder/src/handlers/execute.rs:168-173`, all pending claims are iterated without pagination and processed in the `calculate_withdrawals` function. If there are too many entries, the transaction will fail due to an out-of-gas error, which will effectively block any withdrawals.

An attacker can exploit this by creating a parent contract that creates many dummy child contracts to call the `Cw20HookMsg::Redeem` message. This would cause the `PENDING_CLAIMS` storage to store the addresses, as seen in lines `280-291`, eventually causing the loop in lines `432-462` to fail.

Currently, the attacker requires a small number of initial funds to execute the attack. In the future, this cost will decrease as CW20 tokens will allow zero-amount transfers.

Recommendation

We recommend processing batch unbondings in batches to prevent out-of-gas errors.

Status: Resolved

3. Funds in the proxy contract may be unintentionally forwarded to others

Severity: Critical

When processing replies with the `LP_WITHDRAWAL_REPLY_ID` identifier, all available asset balances in the proxy contract are sent to the caller. This happens when users redeem or withdraw claims in `contracts/autocompounder/src/handlers/reply.rs:160-164`.

As the proxy contract is shared between modules, other modules might rely on it temporarily to hold funds. For example, a DCA contract automatically withdraws native funds to purchase assets at a specific interval, or a limit order contract automatically purchases assets at a specific price.

As a result, malicious users can steal excess funds in the proxy contract. For instance, an attacker may notice that the proxy contract has an excess balance of USDC to be used by a DCA contract. The attacker then redeems their vault token to also receive the excess USDC as part of the `lp_withdrawal_reply` function.

This issue is also present in the `fee_swapped_reply` function, where all available funds in the contract are sent as long they have the same denomination as the fees (see lines `331-338`).

Recommendation

We recommend modifying the reply handlers to send only the required amount instead of transferring all available assets. This can be achieved by recording the proxy contract's old balance before the action, and the amount to be sent can be calculated by taking the difference between the old and new balances.

Status: Resolved

4. Deposit and withdrawal fees are not charged

Severity: Critical

In `packages/forty-two/src/autocompounder.rs:142-143`, deposit and withdrawal fees are defined as part of the fee configuration. However, neither is used within the codebase to collect user deposit fees or withdrawal fees to prevent bad actors from exploiting the compound system.

As stated in the documentation, withdrawal fees deter users from depositing right before the compound function is called and immediately withdrawing, thus taking a percentage of gains made by legitimate users.

Assuming no unbonding period is configured, the lack of a withdrawal fee would allow attackers to skim rewards from other users by depositing immediately before the auto-compound is performed, getting additional rewards with a minimum deposit time.

Recommendation

We recommend implementing functionality to charge deposit and withdrawal fees.

Status: Resolved

5. Compound function is susceptible to sandwich attack

Severity: Major

The `swap_rewards_with_reply` function in `contracts/autocompounder/src/handlers/reply.rs:397` has a hard-coded max spread of `50%`. Using such a large max spread may allow for value to be extracted from the transaction through sandwiching, for example by MEV bots.

The `swap_rewards_with_reply` function is called during the contract's compound operation, which is permissionless. This means that an attacker can initiate the compound functionality while manipulating the pools in their favor when they feel it would present the most value, all as part of a bundle, without even having to watch the mempool for the target transaction.

For example, a DEX like Astroport specifies a default spread of `0.5%` and a maximum allowed spread amount of `50%`. We recognize that the spread is most likely set at `50%` to avoid causing an error in the compound functionality, but a value so large presents a large opportunity for attackers.

While some low liquidity pairs may need a higher slippage tolerance, it is best practice to set a low default value. A low max spread may cause the compound function to fail occasionally, but it limits the value to be lost due to sandwiching. The downside to a low spread is that more transactions may fail, so the gas cost over time may increase due to failed executions.

A similar hardcoded maximum spread value also affects the `compound` function in `contracts/autocompounder/src/handlers/reply.rs:229`.

Recommendation

We recommend creating an updatable config parameter representing the max spread to be used in the compound functionality for both instances described above.

Status: Resolved

6. Incorrect cooldown condition locks unbonding

Severity: Major

The `check_unbonding_cooldown` function is intended to enforce that the cooldown period for batch unbonding has passed in `contracts/autocompounder/src/handlers/execute.rs:471-488`. However, there is a missing operand in the condition in line `479` that reverses it. If `latest_unbonding` plus `min_cooldown` is in the past, the function will raise an `UnbondingCooldownNotExpired` error.

Therefore, the `batch_unbond` function is locked when the cooldown has passed and only can only be used before the cooldown has ended.

Recommendation

We recommend adding a negation operand `(!)` so the condition looks like the following:

```
if !latest_unbonding.add(min_cooldown)?.is_expired(&env.block) {
    return Err(AutocompounderError::UnbondingCooldownNotExpired {
        min_cooldown,
        latest_unbonding,
    });
}
```

Status: Resolved

7. Unbonding cooldowns are not respected

Severity: Major

In `contracts/autocompounder/src/handlers/execute.rs:471-488`, the `check_unbonding_cooldown` function attempts to verify the last unbonding time has exceeded the minimum unbonding cooldown time. However, the `LATEST_UNBONDING` storage state is never stored anywhere in the codebase. Consequently, batch unbondings can be repeatedly performed without respecting the configured cooldown period.

This issue also causes the `query_latest_unbonding` function to fail when loading `LATEST_UNBONDING` from the storage.

We classify this issue as major because it affects the correct functioning of the system.

Recommendation

We recommend storing the latest unbonding time after performing a batch unbond.

Status: Resolved

8. Compounding will fail for zero performance fees

Severity: Minor

In `contracts/autocompounder/src/handlers/reply.rs:190-200`, the fees are deducted from rewards and sent to the commission address. However, the swap will fail if the performance fee is zero, preventing the auto-compounding from working successfully.

Please see the `test_zero_performance_fees` test case to reproduce the issue.

We classify this issue as minor because the manager contract can recover to a correct state by updating the performance fees.

Recommendation

We recommend skipping the swap if the performance fee is zero.

Status: Resolved

9. Hard-coded slippage value makes deposits susceptible to sandwich attacks

Severity: Minor

The `deposit` function in `contracts/autocompounder/src/handlers/execute.rs:143` specifies a hard-coded max spread value of `5%`. Depending on the pair that liquidity is being provided for, this may be a too large value, making the deposit functionality vulnerable to a sandwich attack.

Recommendation

We recommend allowing the user to specify the spread depending on which pair they are providing and adding a config parameter that enforces an upper limit `max_spread` to ensure that the user does not supply too large a value.

Status: Resolved

10. Non-updatable configuration can not reflect changes in staking contract

Severity: Minor

In `contracts/autocompounder/src/handlers/instantiate.rs:139-148`, the configuration is defined as taking both `min_unbonding_cooldown` and `unbonding_period` directly from the staking contract. However, there is no entry point to update this data.

In case the affected parameters are modified in the staking contract, this will cause inconsistencies in the unbonding mechanism. The underlying messages directed to the staking contract could fail without the users being able to understand why, as the reported information will not reflect the new limits.

Recommendation

We recommend implementing an additional entry point that queries the staking contract and updates the affected parameters if required.

Status: Resolved

11. Minting of zero vault tokens possible

Severity: Minor

The autocompounder contract allows for the minting of zero shares upon depositing in some edge cases. This is possible as CW20 tokens now allow the minting of zero tokens, and the compute_mint_amount function does not perform further validation on the returned amount of shares to be minted in contracts/autocompounder/src/handlers/reply.rs:135-143.

Users depositing small amounts of funds could receive zero shares in exchange, effectively losing access to those funds.

In addition, this issue causes the Autocompounder is vulnerable to share inflation attack to be even more lucrative for an attacker.

Recommendation

We recommend returning an error if the calculation results in zero minted shares.

Status: Resolved

12. Unused variable in codebase

Severity: Informational

In the deposit function in contracts/autocompounder/src/handlers/execute.rs:98, the _staking_address variable is declared, but is not used.

Recommendation

We recommend removing the unused variable.

Status: Resolved

13. Lack of validation upon deposit lead to inefficiencies

Severity: Informational

The deposit function performs insufficient validation on funds to be forwarded for liquidity provision in contracts/autocompounder/src/handlers/execute.rs:111-121. As the funds variable is not explicitly checked to contain pool assets only, the contract would try to provide liquidity with potentially erroneous assets in lines 140-144.

This will cause the execution of all the code in the function to end up erroring, consuming an unnecessary amount of gas.

Recommendation

We recommend enforcing that the assets contained within the `funds` variable are part of the target pool.

Status: Resolved

14. Remove TODO comments

Severity: Informational

The codebase includes multiple TODO comments. It is best practice to remove all pending TODO items before releasing code to production.

Recommendation

We recommend resolving all TODO items.

Status: Resolved

15. Unused commented code in codebase

Severity: Informational

In `contracts/autocompounder/src/handlers/instantiate.rs:78-82`, there is unused commented code. It is best practice to remove all unused commented code blocks before code is released toproduction.

Recommendation

We recommend removing the unused code noted above.

Status: Resolved

16. Misleading variable name when withdrawing liquidity

Severity: Informational

In `contracts/autocompounder/src/handlers/execute.rs:267` and `382`, the `swap_msg` variable is set when calling `withdraw_liquidity` on the specific DEX. The `swap_msg` variable name is misleading, swaps are performed later on the DEX.

Recommendation

We recommend renaming the variable from `swap_msg` to `withdraw_msg`.

Status: Resolved

17. `calculate_withdrawals` sets expiration to current block height which may become problematic if the function is exposed in a future upgrade

Severity: Informational

In `contracts/autocompounder/src/handlers/execute.rs:411-414`, the unbonding timestamp defaults to the current block height if `config.unbonding_period` is `None`. This is not a security concern in the current implementation, since the `calculate_withdrawals` function can only be called from the `batch_unbond` function, which ensures the unbonding period is `Some(_)`, as seen in lines `161-163`.

If future code was introduced though that allows calling the `calculate_withdrawals` function without going through the `batch_unbond` functions, it might cause withdrawals to be unlocked immediately in the next block.

Recommendation

We recommend making the unbonding period a non-optional parameter, or unwrapping it and returning an error if `None` is passed.

Status: Resolved

18. Users cannot query fees through smart queries

Severity: Informational

In `contracts/autocompounder/src/handlers/query.rs:16-42`, no exposed queries return the fee configurations. Consequently, users cannot query the protocol's configured performance, deposit, and withdrawal fees.

Recommendation

We recommend exposing a query that returns the fee configurations.

Status: Resolved

19. Panicking macros and debugging code

Severity: Informational

The `autocompounder` contract uses Rust panicking macros to handle undesired situations in `contracts/autocompounder/src/handlers/instantiate.rs:96` and `130`. Panicking macros do not report meaningful error messages for users to understand what went wrong.

In addition, an `eprintln!` statement which is typically used for debugging purposes can be found in `contracts/autocompounder/src/handlers/execute.rs:357-360`.

Recommendation

We recommend implementing adequate error handling, providing meaningful error messages, and removing unnecessary lines for debugging purposes.

Status: Resolved

20. Overflow checks not enabled for release profile

Severity: Informational

The following packages and contracts do not enable `overflow-checks` for the release profile:

- `contracts/autocompounder/Cargo.toml`
- `packages/forty-two/Cargo.toml`
- `packages/forty-two-boot/Cargo.toml`

While enabled implicitly through the workspace manifest, a future refactoring might break this assumption.

Recommendation

We recommend enabling overflow checks in all packages, including those that do not currently perform calculations, to prevent unintended consequences if changes are added in future releases or during refactoring. Note that enabling overflow checks in packages other than the workspace manifest will lead to compiler warnings.

Status: Resolved

21. Additional funds sent to the contract are lost

Severity: Informational

The `deposit` function does not check whether additional native tokens are sent along the message in `contracts/autocompounder/src/handlers/execute.rs:89-157`.

Any additional native tokens are not returned to the user, so they will be stuck in the contract forever.

While blockchains generally do not protect users from sending funds to the wrong accounts, reverting extra funds increases the user experience.

Recommendation

We recommend checking that the transaction contains only the expected `Coin` using https://docs.rs/cw-utils/latest/cw_utils/fn.must_pay.html.

Status: Resolved

Appendix A: Test Cases

1. Test case for "Compounding will fail for zero performance fees"

The test case should pass if the issue is patched.

```
#[test]
fn test_zero_performance_fees() -> AResult {
    use forty_two::autocompounder::AutocompounderExecuteMsg;

    let owner = Addr::unchecked(test_utils::OWNER);
    let wyndex_owner = Addr::unchecked(WYNDEX_OWNER);

    // create testing environment
    let (_, mock) = instantiate_default_mock_env(&owner)?;

    // create a vault
    let mut vault = crate::create_vault(mock.clone())?;
    let WynDex {
        eur_token,
        usd_token,
        eur_usd_staking,
        ..
    } = vault.wyndex;

    let eur_asset = AssetEntry::new("eur");
    let usd_asset = AssetEntry::new("usd");

    // give user some funds
    mock.set_balances(&[
        (
            &owner,
            &[
                coin(100_000u128, eur_token.to_string()),
                coin(100_000u128, usd_token.to_string()),
            ],
        ),
        (&wyndex_owner, &[coin(100_000u128, WYND_TOKEN.to_string())]),
    ])?;

    // update performance fees to zero
    let manager_addr = vault.os.manager.address()?;

vault.auto_compounder.call_as(&manager_addr).execute_app(AutocompounderExecuteMs
g::UpdateFeeConfig { performance: Some(Decimal::zero()), deposit: None,
withdrawal: None }, None)?;

    // initial deposit must be > 1000 (of both assets)
    // this is set by WynDex
    vault.auto_compounder.deposit(
        vec![
            AnsAsset::new(eur_asset, 100_000u128),
            AnsAsset::new(usd_asset, 100_000u128),
        ],
        &[coin(100_000u128, EUR), coin(100_000u128, USD)],
    )?;

    // process block -> the AC should have pending rewards at the staking
contract
    mock.next_block()?;

    // distribute rewards
    vault.wyndex.suite.distribute_funds(
        eur_usd_staking,
        wyndex_owner.as_str(),
        &coins(1000, WYND_TOKEN),
    )?;

    // compound rewards
    vault.auto_compounder.compound()?;

    Ok(())
}
```