StakeWith.Us

Security Assessment

November 26th, 2020

For :
StakeWith.Us

By :
Alex Papageorgiou @ CertiK
alex.papageorgiou@certik.org

Georgios Delkos @ CertiK
georgios.delkos@certik.io

Disclaimer

CertiK reports are not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. These reports are not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts CertiK to perform a security review.

CertiK Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

CertiK Reports should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

CertiK Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK’s position is that each company and individual are responsible for their own due diligence and continuous security. CertiK’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

What is a CertiK report?

- A document describing in detail an in depth analysis of a particular piece(s) of source code provided to CertiK by a Client.
- An organized collection of testing results, analysis and inferences made about the structure, implementation and overall best practices of a particular piece of source code.
- Representation that a Client of CertiK has indeed completed a round of auditing with the intention to increase the quality of the company/product's IT infrastructure and or source code.

Overview

Project Summary

Project Name	Unagii Vaults by StakeWith.Us
Description	This is the Unagii vault implementation of StakeWith.Us, an automated harvester containing investment strategies for various solutions in the DeFi space, including but not limited to interfacing with AMMs like Curve and other automated harvesters.
Platform	Ethereum; Solidity, Yul
Codebase	GitHub Repository
Commits	pre-audit: 74c2ebeabd27ad8fae7fcc002aac1ea9d76f922f post-audit: 6d1919cc327cdadd6804ac55dc0109d97310c4d0

Audit Summary

Delivery Date	November 26th, 2020
Method of Audit	Static Analysis, Manual Review
Consultants Engaged	2
Timeline	November 16th, 2020 - November 26th, 2020

Vulnerability Summary

Total Issues	11
Total Critical	0
Total Major	0
Total Medium	2
Total Minor	3
Total Informational	6

Executive Summary

The audit of the codebase was conducted from two different viewpoints; a security perspective and an optimizational perspective, the former taking precedence over the latter. The StakeWith.Us team laid some of the security assumptions they wished validated directly on the codebase which we proceeded to utilize as building blocks of our security analysis of the project.

We observed several security standards and practices applied to the codebase, such as flash-loan protection via `tx.origin` and a `whitelist`, strict role-based access-control akin to OpenZeppelin's implementation, slippage protection for withdrawals via dynamic balance evaluation, re-entrancy protection via mutexes and more.

The contracts of the project interface with many DeFi blocks which were treated as black-boxes during the audit and only analyzed from an interfacing and expected functionality standpoint.

From an optimizational perspective, we observed certain minor security assumptions that were instead expected to be complied to by the administrative caller of the contracts rather than the code itself. We noted that these should instead be assimilated directly in the codebase for peaceof-mind as well as ensuring that security comes first.

With regards to the mathematical operations that are conducted across the codebase, we validated their definitions according to in-line comments and identified no flaws in the way they are carried out. We pinpointed certain `SafeMath` invocations that could be omitted as they are guaranteed by preceding `if` and `require` clauses in favor of gas optimization.

Funds are not meant to remain at rest, which is further indicated by the way investments in strategies are directly utilized in their respective staking methodology. The contract utilizes ephemeral balance calculation to detect how many funds were deposited and withdrawn from a particular strategy as well as DeFi component, ensuring that no assumptions are made with regards to the impact of external DeFi calls.

We should note that the strategies rely on price calculations that are based on-chain, meaning that they would be susceptible to flash-loan attacks by manipulating the price of given pairs to the attacker's benefit. However, such attempts are impossible in the codebase of the Unagii contracts as both the `deposit` and `withdraw` functions are safe-guarded against flash loans. We should note that both functions should remain guarded as it would be possible for a flash-loaner to affect the result of a `deposit` and `withdraw` combination by first depositing real funds and affecting their withdrawal with the flash-loan.

As per the security considerations, direct underlying token deposits to the vault or strategy should not impact their operation as balances are dynamically evaluated at each point of execution. For example, a manual deposit to a `Vault` would affect the number of shares minted by `deposit`, however the output of `withdraw` would also be proportionately affected as the `_getExpectedReturn` function also factors these balances into account. Likewise, direct deposits to a `Strategy` would be accounted for by the `withdraw` and `withdrawAll` functions.

The sole important implementation flaws we identified are the 2 `medium` severity vulnerabilities that involve the 3Curve strategies and in detail their re-investment functionality. The strategies are meant to implement a `_getMostPremiumToken` that retrieves the stablecoin out of the three that contains the least balance and should be re-invested in the pool by swapping acquired CRV tokens for the particular stablecoin. As the implementation is incorrect, we highly urge this is dealt with prior to launch.

One thing that should be noted is that the built-in slippage protection parameters of Pickle Finance, Uniswap and Curve are not utilized directly. Withdrawals are guaranteed to not slip as the `Vault` implementation contains manual slippage protection, however the deposits do not seem to utilize this functionality. This would allow a malicious sand-which attack to occur, leading to a user's deposit resulting in less funds than originally planned. This attack vector can only be utilized maliciously to decrease the value of a user's deposit and can be considered negligible as the slippage of the supported stablecoins of the project is minimal.

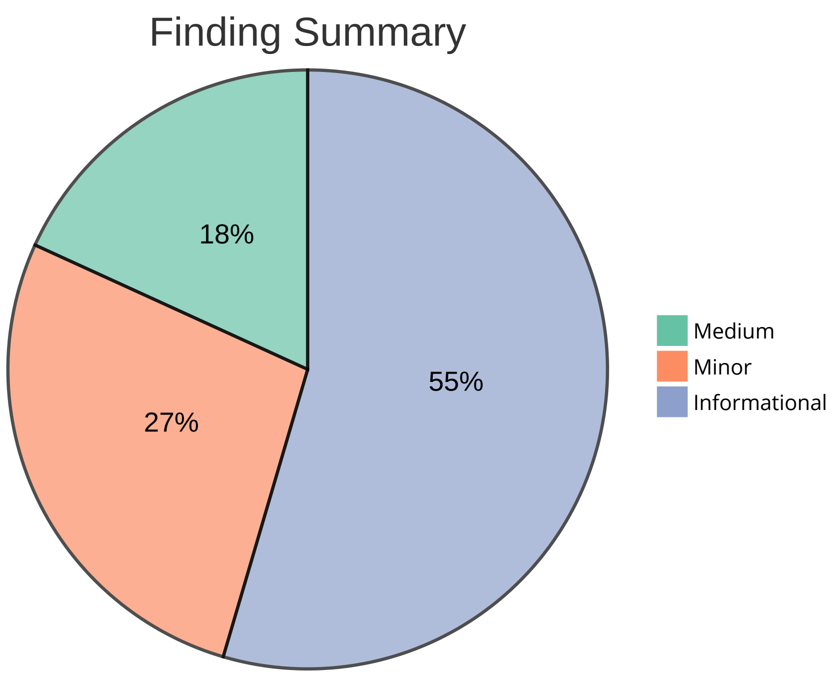
The StakeWith.Us team remediated all the findings outlined in the report apart for `VAU-01` which was extensively discussed with us and concluded to be negligible in the overall security of the project.

Files In Scope

ID	Contract	Location
ACL	AccessControl.sol	contracts/AccessControl.sol
CON	Controller.sol	contracts/Controller.sol
GRR	GasRelayer.sol	contracts/GasRelayer.sol
IOS	IOneSplit.sol	contracts/interfaces/1inch/IOneSplit.sol
DEP	Deposit2.sol	contracts/interfaces/curve/Deposit2.sol
GAU	Gauge.sol	contracts/interfaces/curve/Gauge.sol
MIN	Minter.sol	contracts/interfaces/curve/Minter.sol
SS2	StableSwap2.sol	contracts/interfaces/curve/StableSwap2.sol
SS3	StableSwap3.sol	contracts/interfaces/curve/StableSwap3.sol
GTN	GasToken.sol	contracts/interfaces/GasToken.sol
MCF	MasterChef.sol	contracts/interfaces/pickle/MasterChef.sol
PJR	PickleJar.sol	contracts/interfaces/pickle/PickleJar.sol
UNI	Uniswap.sol	contracts/interfaces/uniswap/Uniswap.sol
ICR	IController.sol	contracts/protocol/IController.sol
ISY	IStrategy.sol	contracts/protocol/IStrategy.sol
ITL	ITimeLock.sol	contracts/protocol/ITimeLock.sol
IVT	IVault.sol	contracts/protocol/IVault.sol
SCV	Strategy3Crv.sol	contracts/strategies/Strategy3Crv.sol
SCD	Strategy3CrvDai.sol	contracts/strategies/Strategy3CrvDai.sol
SCU	Strategy3CrvUsdc.sol	contracts/strategies/Strategy3CrvUsdc.sol
CON	Strategy3CrvUsdt.sol	contracts/strategies/Strategy3CrvUsdt.sol
SCE	StrategyCurve.sol	contracts/strategies/StrategyCurve.sol
SCS	StrategyCusd.sol	contracts/strategies/StrategyCusd.sol
CON	StrategyCusdDai.sol	contracts/strategies/StrategyCusdDai.sol
CON	StrategyCusdUsdc.sol	contracts/strategies/StrategyCusdUsdc.sol
SPC	StrategyP3Crv.sol	contracts/strategies/StrategyP3Crv.sol
SPD	StrategyP3CrvDai.sol	contracts/strategies/StrategyP3CrvDai.sol
SPU	StrategyP3CrvUsdc.sol	contracts/strategies/StrategyP3CrvUsdc.sol
PCU	StrategyP3CrvUsdt.sol	contracts/strategies/StrategyP3CrvUsdt.sol
SBE	StrategyBase.sol	contracts/StrategyBase.sol

ID	Contract	Location
TLK	TimeLock.sol	contracts/TimeLock.sol
UUP	UseUniswap.sol	contracts/UseUniswap.sol
VAU	Vault.sol	contracts/Vault.sol

Findings



ID	Title	Type	Severity	Resolved
CON-01	Security Comments to Code	Logical Issue	Informational	✓
CON-02	Invocation Check	Control Flow	Minor	✓
CON-03	Improper Access Control	Control Flow	Minor	✓
TLK-01	Usage of Setter	Gas Optimization	Informational	✓
VAU-01	Fee Bypass	Mathematical Operation	Minor	✓
VAU-02	Redundant `SafeMath` Utilization	Gas Optimization	Informational	✓
SBE-01	Redundant `SafeMath` Utilization	Gas Optimization	Informational	✓
SCE-01	Redundant `SafeMath` Utilization	Gas Optimization	Informational	✓
SCV-01	Incorrect Premium Token Calculation	Logical Issue	Medium	✓
SCS-01	Redundant Comparison	Gas Optimization	Informational	✓
SPC-01	Incorrect Premium Token Calculation	Logical Issue	Medium	✓

CON-01: Security Comments to Code

Type	Severity	Location
Logical Issue	Informational	Controller.sol L38-L45

Description:

The linked code segment contains comments that warn a call of the `setAdmin` function shuold be preceded by a revocation of the `ADMIN_ROLE` and `HARVESTER_ROLE` for the old administrator and a grant of these for the new administrator.

Recommendation:

We advise that these are instead performed directly on the code to ensure that these security considerations cannot be bypassed and are instead guaranteed by the code.

Alleviation:

The team migrated the restrictions mentioned in the comments to actual statements carried out within the function body thus addressing this issue.

CON-02: Invocation Check

Type	Severity	Location
Control Flow	Minor	Controller.sol L83, L99, L108, L118

Description:

The linked warning comments indicate that the associated functions are able to make "sensitive" calls to `IStrategy` instances that are not associated with a particular `vault`.

Recommendation:

We advise that the warning comments are integrated within the code, either within the definition of a generic strategy (`StrategyBase.sol`) or within `Controller.sol` directly by invoking the getter function of the `vault` within the strategy and ensuring that the `strategy` of the vault is the same as the strategy we are making an invocation to.

Alleviation:

All invocations on an `IStrategy` were adapted to be guared by a `modifier` that ensures the strategy's vault is utilizing it, thus preventing the warning comment that existed earlier from manifesting under any circumstances.

CON-03: Improper Access Control

Type	Severity	Location
Control Flow	Minor	Controller.sol L99-L106, L108-L116

Description:

The function `withdraw` can be invoked by anyone with the `HARVESTER_ROLE` whereas the `withdrawAll` function can only be called by the `ADMIN_ROLE`, as the code dictates.

Recommendation:

As the strategy implementations of `withdrawAll` act like a `withdraw` with the amount set to the maximum, a `HARVESTER_ROLE` is able to replicate a `withdrawAll` invocation by setting proper input parameters for `withdraw`. As such, we advise that the access control for these functions is revised since they do not properly achieve their purpose.

Alleviation:

The `withdrawAll` function was adapted to also utilize the `HARVESTER_ROLE` as it was basically a utility case of the `withdraw` function `HARVESTER_ROLE` was already able to invoke.

TLK-01: Usage of Setter

Type	Severity	Location
Gas Optimization	Informational	TimeLock.sol L43-L44, L47, L63-L73

Description:

The variable `delay` is assigned to during the `constructor` of the contract after passing certain checks that are also imposed by the `setDelay` function.

Recommendation:

We advise that the `constructor` of the contract utilizes the `setDelay` function directly by splitting its implementation to an `internal` that simply conducts the statements and an `external` that applies proper access control.

Alleviation:

The `_setDelay` setter is now properly invoked in the `constructor` leading to a reduction in the contract's generated bytecode.

VAU-01: Fee Bypass

Type	Severity	Location
Mathematical Operation	Minor	Vault.sol L429-L437, L440

Description:

The first code block calculates the `fee` that should be acquired from the `withdrawAmount` of a particular user. The latter code line imposes a `require` check on the amount withdrawn based on the user's `_min` input.

Recommendation:

As the minimum of a withdrawal is imposed by the user, it is possible for the user to invoke `withdraw` repeatedly with a small amount bypassing the `fee` due to the withdrawal fee resulting in `0` because of truncation. We advise that the `require` check of L440 calculates the minimum between `_min` and `FEE_MAX` to ensure that the division does not result in `0`.

Alleviation:

After discussing with the StakeWith.Us team, we concluded that any type of solution to this particular issue would restrain the actions a user would be able to take and as such we collectively agreed that no action should be taken to remediate this exhibit as its impact is minimal and potentially negligible.

VAU-02: Redundant `SafeMath` Utilization

Type	Severity	Location
Gas Optimization	Informational	Vault.sol L230, L378, L426, L430, L435

Description:

The linked mathematical statements utilize numbers wrapped around the `SafeMath` library to ensure operations are carried out safely.

Recommendation:

The usage of `SafeMath` in the linked statements is redundant as their result is guaranteed to be safe based on `if` conditionals or `require` checks that precede them. We advise they are omitted to optimize the gas cost of the contract.

Alleviation:

The linked statements were optimized by removing the utilization of `SafeMath` as they are guaranteed to be safe by surrounding statements.

SBE-01: Redundant `SafeMath` Utilization

Type	Severity	Location
Gas Optimization	Informational	StrategyBase.sol L92, L142, L205

Description:

The linked mathematical statements utilize numbers wrapped around the `SafeMath` library to ensure operations are carried out safely.

Recommendation:

The usage of `SafeMath` in the linked statements is redundant as their result is guaranteed to be safe based on `if` conditionals or `require` checks that precede them. We advise they are omitted to optimize the gas cost of the contract.

Alleviation:

The linked statements were optimized by removing the utilization of `SafeMath` as they are guaranteed to be safe by surrounding statements.

SCE-01: Redundant `SafeMath` Utilization

Type	Severity	Location
Gas Optimization	Informational	StrategyCurve.sol L44, L120

Description:

The linked mathematical statements utilize numbers wrapped around the `SafeMath` library to ensure operations are carried out safely.

Recommendation:

The usage of `SafeMath` in the linked statements is redundant as their result is guaranteed to be safe based on `if` conditionals or `require` checks that precede them. We advise they are omitted to optimize the gas cost of the contract.

Alleviation:

The linked divisions were properly replaced with their raw format as the divisors are literals that will always be different than zero.

SCV-01: Incorrect Premium Token Calculation

Type	Severity	Location
Logical Issue	Medium	Strategy3Crv.sol L48-L70

Description:

The linked `_getMostPremiumToken` implementation compares the balances of the three stablecoins held in the Curve pool by offsetting them to the proper number of decimals and comparing them in sequence using a less-than (`<`) comparator, in the end returning `DAI` as the most premium stablecoin if the previous `if` conditionals fail.

Recommendation:

The implementation is invalid as the default value returned will can be the least-premium token. If the balances of `USDC` and `USDT` are equal but less-than `DAI`, all conditionals will fail and the function will return `DAI` when in-fact the most premium token is either `USDC` or `USDT`. We advise that the `if` conditionals utilize a less-than-or-equal comparison instead and the last `if` clause is omitted to ensure that the event of equal balances is taken into account.

Alleviation:

The premium token calculation was corrected by utilizing a less-than-or-equal comparison instead of a strict less-than comparison, nullifying this exhibit.

SCS-01: Redundant Comparison

Type	Severity	Location
Gas Optimization	Informational	StrategyCusd.sol L62-L65

Description:

The linked code block of `_getMostPremiumToken` checks for the balance difference between `DAI` and `USDC` and returns `DAI` as the most premium token if the balance is less, which is also the default return value of the function.

Recommendation:

As `DAI` is returned by default, the whole `if` clause can be omitted optimizing gas cost.

Alleviation:

The `_getMostPremiumToken` function was optimized to instead conduct a single comparison instead of two, leading to a reduction in gas cost.

SPC-01: Incorrect Premium Token Calculation

Type	Severity	Location
Logical Issue	Medium	StrategyP3Crv.sol L110-L135

Description:

The linked `_getMostPremiumToken` implementation compares the balances of the three stablecoins held in the Curve pool by offsetting them to the proper number of decimals and comparing them in sequence using a less-than (`<`) comparator, in the end returning `DAI` as the most premium stablecoin if the previous `if` conditionals fail.

Recommendation:

The implementation is invalid as the default value returned will can be the least-premium token. If the balances of `USDC` and `USDT` are equal but less-than `DAI`, all conditionals will fail and the function will return `DAI` when in-fact the most premium token is either `USDC` or `USDT`. We advise that the `if` conditionals utilize a less-than-or-equal comparison instead and the last `if` clause is omitted to ensure that the event of equal balances is taken into account.

Alleviation:

The premium token calculation was corrected by utilizing a less-than-or-equal comparison instead of a strict less-than comparison, nullifying this exhibit.

Appendix

Finding Categories

Gas Optimization

Gas Optimization findings refer to exhibits that do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Mathematical Operations

Mathematical Operation exhibits entail findings that relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings are exhibits that detail a fault in the logic of the linked code, such as an incorrect notion on how `block.timestamp` works.

Control Flow

Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Data Flow

Data Flow findings describe faults in the way data is handled at rest and in memory, such as the result of a `struct` assignment operation affecting an in-memory `struct` rather than an in-storage one.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of `private` or `delete`.

Coding Style

Coding Style findings usually do not affect the generated byte-code and comment on how to make the codebase more legible and as a result easily maintainable.

Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a `constructor` assignment imposing different `require` statements on the input variables than a setter function.

Magic Numbers

Magic Number findings refer to numeric literals that are expressed in the codebase in their raw format and should otherwise be specified as `constant` contract variables aiding in their legibility and maintainability.

Compiler Error

Compiler Error findings refer to an error in the structure of the code that renders it impossible to compile using the specified version of the project.

Dead Code

Code that otherwise does not affect the functionality of the codebase and can be safely omitted.