CERTIK

CertiK Audit Report for StakeWith.Us

Contents

Contents 1

Disclaimer 2

About CertiK 2

Executive Summary 3

Testing Summary 4

SECURITY LEVEL 4

Review Notes 5

Introduction 5

Scope of Work 5

Summary 6

Revisions 7

Analysis 8

Audit Findings 14

Exhibit 1 14

Exhibit 2 15

Exhibit 3 16

Exhibit 6 19

Exhibit 7 20

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between CertiK and Unagii.com (the "Company"), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the "Agreement").

About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK's mission of every audit is to apply different approaches and detection methods, ranging from manual, static, and dynamic analysis, to ensure that the project is checked against known attacks and potential vulnerabilities. CertiK leverages a team of seasoned engineers and security auditors to apply testing methodologies and verifications on the project, in turn creating a more secure and robust software system.

CertiK has served more than 100 clients with high quality auditing and consulting services, ranging from stablecoins such as Binance's BGBP and Paxos Gold to decentralized oracles such as Band Protocol and Tellor.

Executive Summary

Unagii.com is a suite of tools that makes it easier to interact with decentralized finance systems. This report has been prepared for them to discover issues and vulnerabilities in the source code of their **smart contract wallet and a set of "recipe's" which provide additional functionality over barebones interactions with defi products,** as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Dynamic Analysis, Static Analysis, and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

After conducting our review, we identified one Minor vulnerability and a few Info-level points that we think the team should be aware of.

Testing Summary

SECURITY LEVEL

TYPE	Smart Contract Audit
SOURCE CODE	https://github.com/stakewithus/smart-contracts/
PLATFORM	Ethereum Virtual Machine
LANGUAGE	Vyper
REQUEST DATE	July 10, 2020
DELIVERY DATE	July 20, 2020
METHODS	A comprehensive examination has been performed using Whitebox Analysis. In detail, Dynamic Analysis, Static Analysis, and Manual Review were utilized.

Review Notes

Introduction

Unagii.com is a platform that allows users to invest stablecoins into lending and liquidity pools starting with Curve.Fi. The audit was conducted from July 10, 2020 to July 15, 2020. Unlike other smart contract based projects on Ethereum blockchain Unagii.com utilizes Vyper which compared to Solidity specializes more on simplicity and security. The goal of this audit was to further review smart contracts' security and ensure that their functionalities adhere to specifications.

Scope of Work

The exact scope of the audit covered the following files:

- gas_relayer.vy
- admin.vy
- smart_contract_wallet.vy
- recipe_curve_busd.vy
- recipe_curve_compound.vy
- recipe_curve_pax.vy
- recipe_curve_susd.vy
- recipe_curve_usdt.vy
- recipe_curve_y.vy

The version of the codebase was that corresponding to the commit:

c9be7616c386a3f2efe21a2e3d69d3277d2b57f9

The post-changes commit was:

3dd91945e3950f62a36d523debe8a4a6221d0638

Summary

After a careful assessment we reached the conclusions that the smart contract wallet is truly non-custodial, namely that only the user, not even smart contract creator, has access to their own fund. Moreover they are the only person authorized to redeem funds that are held by the Recipe.

Furthermore, after additionally reviewing the smart contract gas relayer that uses gas token, which stores gas by creating a new contract during low gas cost period and releases gas by calling the self-destruct opcode during high gas cost period — we believe this optimization makes the smart contract cheaper on gas and more attractive to the users.

While most of the issues pinpointed were of negligible importance and mostly referred to coding standards and inefficiencies, a minor flaw was identified that can be remediated to ensure the contracts of the Unagii.com are of the highest standard and quality.

These inefficiencies and flaws can be swiftly dealt by the development team behind Unagii.com so that a second round of auditing can proceed. We will create and maintain a direct communication channel between us and the Unagii.com team to aid in amending the issues identified in the report.

Revisions

The Client has made changes to the codebase based on our suggestions. The points where no action was taken we do not think give rise to vulnerabilities. Furthermore, the points where changes have been made were correctly implemented, with no new vulnerabilities arising.

Analysis

What follows is a brief analysis of each of the modules.

Modules

Modules	TYPE	PASS
admin.vy	• the owner can set fixed fee's per token • the owner can withdraw tokens held by the contract	✔
Timelock	• Regarding the first point, the contract has a timelock mechanism. The owner can propose new fee's for a token. After 2 hours, they can realize the change. • It should be noted that the latter function (apply_new_fee) requires the deadline to exist, and immediately resets it. If we think of binary strings {0,1}^2 as representing whether these two statements are there[1], then all four combinations would be secure. In particular, 11 is, which represents the approach taken.	✔
Withdrawing	• The withdraw functionality is intended to allow the owner to send any ERC20 token (in particualr those accumulated from the fee mechanism) to an arbitrary address.	✔
gas_relayer.vy	The gas_relayer is also parameterized by the creator (owner), but also by the gastoken, admin, tether and the	✔

[1] e.g. 01 would represent no require but a reset.

	smart contract wallet as a factory model. There are four main functionalities: ● Minting and transferring GST2 tokens ● Access control / whitelisting ● Relaying tx's (executing message calls) ● Creating new wallets	
GST2	The contract allows anyone to call "bank", which mints GST2 token for the current contract using gastoken.mint. The owner can transfer an arbitrary amount of these tokens to an arbitrary address.	✔
Whitelist	The owner is defaultly whitelisted. The owner has the power to whitelist and dewhitelist arbitrary addresses. They can dewhitelist themselves, but since whitelisting is not predicated upon already being whitelisted, they could easily whitelist themselves again.	✔
Executing msg calls	Whitelisted users can execute arbitrary[2] CALL's on behalf of the gas_relayer contract. They can also set a _value != 0, in which case up to _value of GST2 sub-tokens will be freed. Two notes: ● It might be better to set the _value to the balance, if it exceeds it, rather than reverting. ● The GST2 freeUpTo fn should be called with sufficient gas. See gas token[3] for more details.	✔

[2] the to field's length is bounded from above by 6000 bytes. The programming language does not allow fully dynamic byte-sequences. We think this cap is very reasonable and shouldn't be a constraint very often, if ever, for full interaction with the Ethereum network.
[3] <https://github.com/projectchicago/gastoken/blob/da37d16390f3b91ebbb7d8e7744f4bdd16b3d16a/contract/GST2_ETH.sol#L188>

Creation of new wallets	Whitelisted users can also create new smart contract wallets. This is done using self.SCW as a model, and using Vyper's built-in creater_forwarder_to. As can be learned from the Vyper compiler source code, this function creates a proxy with a hard-coded "masterCopy" address equal to the only argument to the fn. While this doesn't allow the destination to be changed, it does lead to one less SLOAD op and therefore to gas savings. Since the new wallet is a proxy, we also call its setup fn with admin and tether. Finally, whitelisted users can also can the change_owner fn on any address on behalf of the gas_relayer.	✔
smart_contract_wallet.vy	The SCW enables construction using either EVM's construction mechanism or using a masterCopy. For this purpose, there is an __init__ fn and a setup fn that differ only in that setup needs logic to ensure it is called only once. In both cases, the owner and creator is set to msg.sender and admin and tether address are parameterized. The wallet has these major functionalities: ● Access control ● Relaying msg calls by owner ● ERC20 transfers	✔
Access control	The contract has three elevated roles. 1. proxy_admin is the fee admin. They are set during construction / setup and cannot be changed later. They are relevant only for ERC20 transfers. There, we get the fixed fee corresponding to that particular token, and finally transfer them precisely that amount. 2. creator is set to msg.sender during construction / setup. They have only one power, and that is to	✔

	change the owner a single time. 3. The owner is also originally set to msg.sender. They can be changed a single time. They have one power which is to sign ”transactions” that anybody can then relay to the smart contract.	
Relaying msg calls by owner	The mechanism works as follows: ● The relayer submits an approve and return coin and amount, a to and data field, and an Ethereum-style signature (v, r, s). ● We concatenate of all the arguments together with the nonce, hash it, add the prefix, hash that and that becomes the signing data hash. ● If the signature matches, the contract approves another address w.r.t. a token and readsits current balance of the return coin. ● It performs the call. ● It checks whether it received at least the return amount. ● The nonce is also incremented to prevent replay attacks. **There is a slight deviation from EIP191[4] that is described in Collated Findings*. ● The contract also supports collating multiple such message calls atomically in one transaction, and the logic is very similar.	✔
ERC20 transfers	The contract also has functionality for transferring ERC20 tokens. We think there is a way to circumvent these fee’s using the above functionality.	✔

[4] <https://eips.ethereum.org/EIPS/eip-191>

Re-entrancy attacks	In theory re-entrancy should not be an issue, since the theoretical framework of a re- entrancy attack is to drain funds from a contract, while we are assuming that if a signature gets validated, it's the owner and therefore they have full access to the funds. Nevertheless, all public, state-modifying functions are marked with a single non-reentrancy mutex, serving to further the security guarantees. The only exceptions are the admin functions, and we do not see a way they could be abused in any way.	✔
recipe_curve_*	These contracts represent the various intermediate algorithms, between the smart contract wallet and CurveFi pools. In particular, the recipe's add two additional functionalities: automatic wrapping and unwrapping of tokens and an additional slippage check. The contracts are parameterized by the file recipe_curve_base.template. All contracts are parameterized by an array of _coins and _underlying_coins, the curve and curve_pool, and finally the owner (initially set to msg.sender). * *We think a few small improvements could be made, listed in the Findings*. ● We have not found any logic vulnerabilities in these contracts. We also verified the parameter values and found them to be correct. ● We also checked for re-entrancy attacks. The only time the contracts interact with a non-pre-approved external address is in _transfer, and there it is the last bit of code. ● Finally, we checked for overflow problems (despite Vyper having built-in protection, it can	✔

	happen that e.g. the accompanying REVERT stalls the system), and have not found any issues.	

Audit Findings

Exhibit 1

TITLE	TYPE	SEVERITY	LOCATION
Graceful error handling	Efficiency	Informational	gas_relayer

[INFORMATIONAL] Description:

There are situations where it is possible to adjust an incorrectly inputted value rather than reverting. One example is relay_tx function in gas_relayer. It is possible to set _value to min(_value, _bal), where _bal is the user's gas token balance.

Client's response:

No action

The value for GST2 to be burnt is provided by the Unagii backend. Having the smart contract revert when there is insufficient value is the desired behaviour.

Exhibit 2

TITLE	TYPE	SEVERITY	LOCATION
DRY code for __init__ and setup	Best practices	Informational	smart_contract_wallet

[INFORMATIONAL] Description:

Since __init__ and setup differ only slightly in smart_contract_wallet, it is possible to extract the common logic to an internal function.

Recommendation:

Refraction of the logics to a helper function, as this makes the code more readable.

Client's response:

No action

There is code duplication, however we believe in the interest of clarity, it is acceptable since there are few lines that are being duplicated.

Exhibit 3

TITLE	TYPE	SEVERITY	LOCATION
EIP191 Compatibility	Standards	Informational	smart_contract_wallet

Description:

According to EIP191[5], pre-signed tx's should have format 0x19 <1 byte version> <version specific data> <data to sign>. The <data to sign> can remain a hash, but we think the <1 byte version> and <version specific data> should be 0 and the address of the current contract, respectively. This will ensure compliance with the standard and most importantly prevent replay attacks for wallets with the same owner.

Client's response:

Added (self) address field during creation of the digest for proxy_action_single, proxy_action_multi and proxy_transfer

The smart_contract_wallet.vy is using the `personal_sign` convention for signature verification which is already making use of the leading `0x19` scheme. However, there is concern about the re-use of signatures for two smart contract wallets sharing the same owner, hence an additional field is added in the calculation of the digest to account for, with additional test cases added.

Check:

We can confirm the address is part of the argument in all relevant functions.

[5] <https://eips.ethereum.org/EIPS/eip-191>

Exhibit 4

TITLE	TYPE	SEVERITY	LOCATION
Transfer fee's can be circumvented	Economics	Minor	smart_contract_wallet

Description:

We think the transfer fee's in SCW can be circumvented. Namely instead of proxy_transfer(_to, _coin, _amount, _v, _r, _s), one could do proxy_action_single(0x0, 0, 0x0, 0, _coin, data, _v2, _r2, _s2) (where data is the abi-encoding of transfer(_to, _amount)).

On a different note, the fee's in the recipe's could also be circumvented, although that would require deploying a new, fee-less contract.

Client's response:

No action

We recognize that the fixed token fee can be bypassed by sending transfer calldata to the proxy_action_single and multi functions. In normal operation, all operations on the smart contract wallet is relayed by the unagii backend with arguments pre-formed.Hence, the only way to bypass the fees would be direct interaction with the smart contract for which the user would have to pay their own gas. In similar fashion, users can fork the smart contract code and leave fees as zero.

Exhibit 5

TITLE	TYPE	SEVERITY	LOCATION
Incorrect natspec	Natspec	Informational	recipe_curve_base

[INFORMATIONAL] Description:

redeem_uneven and __init__ in recipe_curve_base.template have incorrect natspec fields.

Client's response:

Fixed incorrect natspec

Check:

redeem_uneven has correct nastpec.

Exhibit 6

TITLE	TYPE	SEVERITY	LOCATION
redeem_uneven doesn't transfer remainder	Better implementation	Informational	recipe_curve_base

[INFORMATIONAL] Description:

There is an asymmetry between the sets {invest, redeem_even} and {redeem_uneven}. In the first set, setting the min values to 0 does not incur a cost. It is just an expressed preference of conducting a market-order-style transaction rather than a limit order. In the second set however, setting _max_burn_amt to anything above the actual burn amount does incur a material cost. In particular, the entire _max_burn_amt is transferred from the sender, and if anything less is burned, it is not refunded.

Recommendations:

A solution would be to check the balances before and after the CurveFi call, and refund the remainder.

Client's response:

Amended the recipe base template to send unburnt / surplus pool tokens back to the smart contract wallet / sender.

Check:

We have checked that this was implemented correctly and does not introduce any new bugs.

Exhibit 7

TITLE	TYPE	SEVERITY	LOCATION
More efficient arithmetic	Ineffectual Code	Informational	recipe_curve_base

Description:

There is only one place in the recipe's where the _stored_rates fn is used and that is in the line:

redeem_wrapped_amts[i] = (_amounts[i] * PRECISION * prec_mul[i]) / rates[i]

_stored_rates is effectively a switch:

• if the token is NOT a wrapped token, it is equal to PRECISION_MUL * PRECISION

• if the token IS a wrapped token, it is equal to PRECISION_MUL * mul, where mul is parameterized.

Recommendations:

We could change the semantics of _stored_rates by removing the PRECISION_MUL.

Then it wouldn't be necessary to multiply by it in the numerator either, saving gas.

Client's response:

No action

The recipe base template closely mirrors how Curve.fi is doing their calculations for rates , etc, and we do not find it a good reason to deviate since it might affect compatibility for future Curve.fi pools.