WeStarter

Security Assessment

Feb 16th 2021

By:

Boxi Li | CertiK

boxi.li@certik.org

Junhong Chen | CertiK

junhong.chen@certik.org

Jialiang Chang | CertiK

jialiang.chang@certik.org

Summary

This report has been prepared for the WeStarter smart contract, Starter, to discover issues and vulnerabilities in the source code as well as any dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing 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 security experts.

The security assessment resulted in 6 findings that ranged from minor to informational. We recommend to address these findings as potential improvements that can benefit the long run as both smart contracts would lock a significant amount of WAR tokens for a significant amount of time. We have done rounds of communications over the general understanding and the WeStarter team has resolved the questions promptly.

Overall the source code is well written with security practices. The business logic is straightforward and implemented accordingly. Yet we suggest a few recommendations that could better serve the project from the security perspective:

1. Enhance general coding practices for better structures of source codes;
2. Add enough unit tests to cover the possible use cases given they are currently missing in the repository;
3. Provide more comments per each function for readability, especially contracts are verified in public.

Overview

Project Summary

Name	WeStarter
Codebase	https://github.com/we-starter/contracts

Engagement Summary

Delivery Date	Feb 16th, 2021
Methodology	Static analysis, manual review and testnet simulation
Contracts in Scope	1
Contract - STARTER	Starter

Finding Summary

Total	0
Critical	0
Medium	0
Minor	1
Informational	5

Understanding of Core Logics

Starter

Key Functions

The `Starter` contract enables purchase of `underlying` token with `currency` token.

The following quantities are assumed for a formalized description of the key functions:

- d_c: currency token's decimal.
- d_u: `underlying` token's decimal.
- T_c: Total amount of `currency` token that all users deposited at the time of `settle`, in the `currency` token's own unit. T_c: Total amount of currency token that all users deposited at the time of settle, in the currency token's own unit.
- T_u: Total amount of `underlying` token available at the time of `settle`, in the `underlying` token's own unit. T_u: Total amount of underlying token available at the time of settle, in the underlying token's own unit.
- $P_c^{(a)}$: Amount of currency token that user a has deposited (through purchase function) at the time of settle, in the currency token's own unit.
- $P_u^{(a)}$: Amount of `underlying` token that user `a` is able to purchase (through `settle` function) at the time of `settle`, in the `underlying` token's own unit.

constructor

- `currency`: The `currency` token used to purchase the `underlying` token.
- `underlying`: The `underlying` token to be purchased by `currency` token.
- `Price`: $(p \times 10^{d_c}/10^{d_u}) \times 10^{18}$, a predefined expected amount of `currency` token required to purchase a single unit of `underlying` token, with additional precision of 10^{18}.
- `time`: A predefined deadline timestamp of the purchase phase before which users may be able to deposit the `currency` token. After the `time` timestamp users are able to exchange the `underlying` token and retrieve back unused `underlying` token.

purchase

The `purchase` function permits users to deposit `currency` token in exchange of `underlying` token, prior to a pre-defined timestamp `time`.

Assume after the finish of the purchase phase (after timestamp `time`), the amount of `currency` token that user a has purchased is $P_u^{(a)} \times 10^{d_c}$, and all deposited `currency` tokens `totalPurchasedCurrency` is $\sum_{a \in A}(P_c^{(a)} \times 10^{d_c})$ where A is the set of users that have called `purchase` to deposit.

settle

The `settle` function carries out the token exchange process for a user based on its deposited `currency` tokens. The amount of `underlying` token that the user is able to purchase is calculated by the `settleable` function. The `settle` process can only be invoked after the pre-defined timestamp `time` (when the purchase phase defined by the `purchase` function has finished).

The final exchange rate of the `underlying` token (in `currency` token) is fixed at the initial successful call of `settle` function by the amount of total deposited `currency` token ($T_c \times 10^{d_c}$) and the available amount of `underlying` token ($T_u \times 10^{d_u}$) at the call time. The amount of `underlying` token that a user can exchange is also fixed at the same time, based on the amount of `currency` token it has deposited.

Let the total amount of `underlying` token that user `a` is able to purchase (with its deposited `currency` token $P_c^{(a)} \times 10^{d_c}$) be $P_u^{(a)} \times 10^{d_u}$. Then we have:

`settleable`:

- `totalCurrency`: $T_c \times 10^{d_c}$, total amount of deposited `currency` token.
- `totalUnderlying`: $T_u \times 10^{d_u}$, total amount of available `underlying` token for exchange.
- `rate`: The ratio of the actual amount of required `currency` token to the original expected amount of required `currency` token for purchasing a single unit of `underlying` token, with precision of 10^{18}.
  - If $(T_u \times p) < T_c$ (received `currency` token exceeds the expected required amount of `currency` token), then `rate` is $(p \times T_u/T_c) \times 10^{18} < 1 \times 10^{18}$.
  - If $(T_u \times p) \geq T_c$ (received `currency` token is less than the expected required amount of `currency` token), then `rate` is 1×10^{18}.
- `purchasedCurrency`: $P_c^{(a)} \times 10^{d_c}$, the amount of `currency` token that user a has purchased.

- settleAmount: $P_c^{(a)} \times 10^{d_c} \times (T_u \times p / T_c)$, or $P_c^{(a)} \times 10^{d_c}$, the actual amount of currency token used (for purchasing underlying token).
- amount: $P_c^{(a)} \times 10^{d_c} \times [1 - (T_u \times p / T_c)]$, or 0, the amount of unused currency token to be returned to the user.
- volume: $\frac{P_c^{(a)}}{T_c} \times T_u \times 10^{d_u} = P_u^{(a)} \times 10^{d_u}$, amount of underlying token purchased by user a.

settle:

- settleUnderlyingOf: $P_u^{(a)} \times 10^{d_u}$, amount of underlying token purchased by user a.

withdraw

The withdraw function enables the governor (an administrator role) of the Starter contract to withdraw obtained currency token from the purchasers and excessive underlying token that are not sold.

withdrawable:

- amt: $T_c \times 10^{d_c} - \sum_{a \in A \backslash A'} [P_c^{(a)} \times 10^{d_c} \times (1 - p \times \frac{T_u}{T_c})]$, the amount of currency token that the Starter contract's governance role is able to withdraw after the token sale has completed, which amounts to all currency tokens that the Starter contract owns except those belonging to users that have deposited currency tokens but haven't called settle to withdraw the unused portion of the deposited currency.
- vol: $T_u \times 10^{d_u} - \sum_{a \in A \backslash A'} [\frac{P_c^{(a)}}{T_c} \times T_u \times 10^{d_u}]$, the amount of underlying token that the Starter contract's governance role is able to withdraw after the token sale has completed, which amounts to all underlying tokens that the Starter contract owns except those belonging to users that have purchased underlying tokens but haven't called settle to retrieve.

where:

- A: The set of users that have deposited currency tokens through purchase.
- A': The set of users that have deposited currency tokens and have called settle to retrieve the underlying token.

- AA' : The set of users that have deposited `currency` tokens but haven't called `settle` to retrieve the `underlying` token.

`withdraw`:

- `amount`: Actual amount of `currency` token withdrawn by the `Starter` contract's governance role.
- `volume`: Actual amount of `underlying` token withdrawn by the `Starter` contract's governance role.

Key Properties

Time

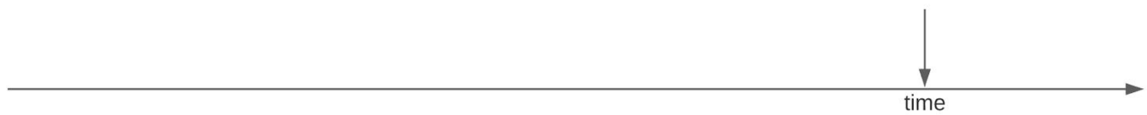
- Users should only be able to make deposits before timestamp `time`.
- Exchange rate (`settleRate`) should not be finalized before timestamp `time`.
- Users should not be able to purchase more `underlying` tokens after token exchange rate (`settleRate`) is finalized (`completed_`).
- Users should not be able to obtain `underlying` tokens before the exchange rate (`settleRate`) is finalized.
- Users should not be able to re-obtain `underlying` tokens after the initial withdrawal.

Permission

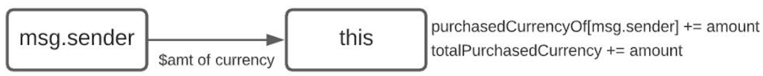
- Non-governance users should not be able to change their governance role.
- Non-governance users should not be able to `withdraw`.

Diagrams based on Timeline

now < time
completed = false



purchase():



settleable():

```
rate = min(
    totalUnderlying * price / totalCurrency,
    1 ether
)
```

$$amount = purchasedCurrencyOf[acct] \times \frac{1-rate}{1\ ether}$$

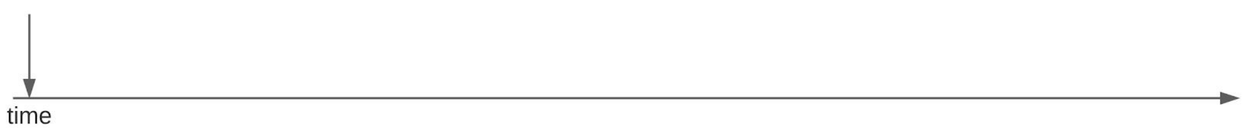
$$volume = purchasedCurrencyOf[acct] \times \frac{rate}{price}$$

settle(): revert()

withdrawable(): return (amt = 0, vol = 0)

withdraw(): revert()

now >= time
completed = true (after settle() is called)



purchase(): revert()

settleable():

settle() is not yet called,
completed = false

```
rate = min(
    totalUnderlying * price / totalCurrency,
    1 ether
)
```

$$amount = purchasedCurrencyOf[acct] \times \frac{1-rate}{1\ ether}$$

$$volume = purchasedCurrencyOf[acct] \times \frac{rate}{price}$$

settleable():

settle() is called,
completed = true

```
rate = settleRate,
    where settleRate is set in the previous call of settle()

if settledUnderlyingOf[acct] > 0:
    return (completed = true, amount = 0, volume = 0, rate = settleRate)
else:
```

$$amount = purchasedCurrencyOf[acct] \times \frac{1-rate}{1\ ether}$$

$$volume = purchasedCurrencyOf[acct] \times \frac{rate}{price}$$

Findings

ID	Title	Severity	Response
CTK-STARTER-1	Layout of Contract	Informational	Pending
CTK-STARTER-2	Pragma Solidity Version	Minor	Pending
CTK-STARTER-3	Unit Consistency	Informational	Pending
CTK-STARTER-4	Condition for withdraw()	Informational	Pending
CTK-STARTER-5	Unnecessary Return Value	Informational	Pending
CTK-STARTER-6	Implicit Status Check	Informational	Pending

CTK-STARTER-1 | Layout of Contract

Type	Severity	Location
Language Specific	Informational	Starter

Description

According to Solidity Documentation, the layout of contract components are recommended to be ordered. To be specific, event declaration could be gathered before functions.

Recommendation

Layout contract elements should following below order:

1. Pragma statements
2. Import statements
3. Interfaces
4. Libraries
5. Contracts

Each contract, library or interface should following below order:

1. Type declarations
2. State variables
3. Events
4. Functions

Functions should be grouped according to their visibility and ordered:

1. constructor
2. fallback function (if exists)
3. external
4. public
5. internal
6. private

CTK-STARTER-2 | Pragma Solidity Version

Type	Severity	Location
Language Specific	Minor	Starter: L335; WAR: L352

Description

For the syntax that setting gas and value in external calls, it is required that the Solidity version is greater than or equal to `0.6.2`. However, current contracts are declaring the compiler version of `pragma solidity ^0.6.0`, where `0.6.0` and `0.6.1` would fail in compiling the contracts.

```
(bool success, ) = recipient.call{ value: amount }("");
```

Recommendation

Recommend explicitly pointing out the version of the compiler, for example, since on the HECO Chain, the three contracts are deployed and tested using `0.6.12`, the statement could be `pragma solidity 0.6.12`. Otherwise, `pragma solidity ^0.6.2` also works.

CTK-STARTER-3 | Unit Consistency

Type	Severity	Location
Coding Style	Informational	Starter

Description

According to Solidity Documentation:

```
assert(1 wei == 1);
assert(1 gwei == 1e9);
assert(1 szabo == 1e12);
assert(1 finney == 1e15);
assert(1 ether == 1e18);
```

Recommendation

Recommend to keep consistency of using `1e18` and `1 ether`.

CTK-STARTER-4 | Condition for withdraw()

Type	Severity	Location
Volatile Code	Informational	Starter: `withdraw()`

Description

Function `withdraw()` requires `completed == true`, where `completed` is only set `true` in `settle()`. In this case, if no external users call `settle()`, the `governor` of the starter contract is not able to perform `withdraw()`.

Recommendation

Our current understanding is that if the fundraising is completed, then users can `settle()` and the governor can `withdraw()`, so probably use `time` to check if `withdraw()` is ready to be performed, just like `settle()`.

CTK-STARTER-5 | Unnecessary Return Value

Type	Severity	Location
Volatile Code	Informational	Starter: `settleable()`

Description

Return value `completed_` is not necessary for function `settleable`.

Recommendation

Recommend removing the use of return value `completed_` in function `settleable` and `settle` for better clarity.

CTK-STARTER-6 | Implicit Status Check

Type	Severity	Location
Volatile Code	Informational	Starter: `withdrawable()`, `settleable()`

Description

The use of `if (!completed) return (0, 0);` in function `withdrawable` results in unnecessary execution and is redundant with the `require(completed, "uncompleted");` check in function `withdraw`.

The use of `if (settledUnderlyingOf[acct] > 0) return (completed_, 0, 0, rate);` in function `settleable` results in unnecessary execution.

Recommendation

Recommend extracting `if (!completed)` check in function `withdrawable` to explicit status check modifier `hasCompleted` and add the modifier to function `withdraw` (and/or `withdrawable`) for better clarity:

```
modifier hasCompleted {
  require(now >= time, "...");  // optional
  require(completed, "...");
  _;
}
```

Recommend extracting `if(settledUnderlyingOf[acct] > 0)` check in function `settleable` to explicit status check modifier `hasNotSettled` and add the modifier to function `settle` (and/or `settleable`) for better clarity:

```
modifier hasNotSettled {
  require(settledUnderlyingOf[acct] == 0, "...");
  _;
}
```

The `require(now < time, 'expired');` check in function `purchase` and `require(now >= time, "It's not time yet");` check in function `settle` can also be optionally extracted as explicit modifiers.

Appendix | Finding Categories

Gas Optimization

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

Refer to exhibits that relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

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

Control Flow

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

Volatile Code

Refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Data Flow

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 instorage one.

Language Specific

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

Coding Style

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

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

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

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.

Business Model

Refer to contract or function logics that are debatable or not clearly implemented according to the design intentions.

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 Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is 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 assessment. This report does 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.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing 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.

About CertiK

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.

CERTIK

Provable Trust For All