CERTIK

Security Assessment

unizen

Jul 13th, 2021

Table of Contents

Summary

This report has been prepared for unizen to discover issues and vulnerabilities in the source code of the unizen project as well as any contract 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 industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases given they are currently missing in the repository;
- Provide more comments per each function for readability, especially contracts are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

Overview

Project Summary

Project Name	unizen
Platform	Ethereum
Language	Solidity
Codebase	https://github.com/unizen-io/unizen-flexible-staking/
Commit	f3039faa51172ff3d61c8e4faeea463be0bd370b

Audit Summary

Delivery Date	Jul 13, 2021
Audit Methodology	Static Analysis, Manual Review
Key Components	

Vulnerability Summary

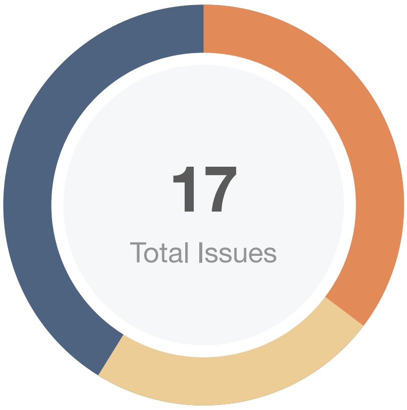
Vulnerability Level	Total	Pending	Partially Resolved	Resolved	Acknowledged	Declined
● Critical	0	0	0	0	0	0
● Major	6	0	0	2	4	0
● Medium	0	0	0	0	0	0
● Minor	4	0	0	4	0	0
● Informational	7	0	0	5	2	0
● Discussion	0	0	0	0	0	0

Audit Scope

ID	file	SHA256 Checksum
FCK	Faucet.sol	7f5c30d32c0bc4be451fe60a5ee3b372daabf9d31b32521c4e92e5cca0211f03
UZF	UZFactoryV1.sol	36a445f2dd5d686c4b29b91028558c89fb5fd9374ad48b0d33fb2c797cad5380
UZR	UZRouterV1.sol	34aa7401ad6fbf5603ad9ebbb9468f2b0ba2b7ca243af8bda9b122672cb12094
UZS	UZStakingV1.sol	378a77b11b8f33aa54420c87a3e85f7b0caebe32bc334aedfae0a25f7a3bf025
ZCX	ZCX.sol	7f01fa7d7c27b66360a5496509dc0579e4145549df84858fdd7ad557d785301a
IUZ	interfaces/IUZDAO.sol	26641e18a483dc7522285bb5f4369d0c75af14caf6374c53287692d9862fee98
IUF	interfaces/IUZFactoryV1.sol	3919546b245166c00eb950152711c867009d75472caabba93e4c1bd3f53d70eb
IUR	interfaces/IUZRewardPoolV1.sol	0963c7146b3667078bd87a322e69737da08742617244c5eb4e24328a7d0ebcb1
IUV	interfaces/IUZRouterV1.sol	9c9368420d04b5a41e3a7e4a085e8b86b2583cce3e131b21466456d2018ed2aa
IUS	interfaces/IUZStakingV1.sol	2c86175c63699ccc1665442ce04bc3aa2ad00cbcedfac1b0ad7e1b5f37cad694
CFC	libraries/CloneFactory.sol	0a280277d55e8e0161d5a84675df798bbc4978043646120e15debf3349ba3748
PCC	libraries/PoolCalculations.sol	0bf1a4e17f2cb4f70fcb43ecab5784a14b696d2c0199c33e6f535c484471dd3d
SDT	libraries/SharedDataTypes.sol	30f436b451f6ec0272154f92d5b5c84200e2d37762f1de630027b2a0a8ae4e63
IER	membership/IERC20Extended.sol	44b0ee9df70c404f6eb5327d3548d5f3b1f5dde68351e03e0bfd35ece52b50ac
IUP	membership/IUZProNFT.sol	0be87ca1e4f8f635d2ec7c4e7a46d2bd110ac09d0247cb1ac08d4f7c9906cc73
UZP	membership/UZProNFT.sol	c16eddce5a9d3b7e41ff00c5e5c26382b1291cd6929656eaa893eeceff06b581
UZC	membership/UZProSale.sol	d306291a5259ad5486bf43a37d9175215ae253f665b83141060496c1e6d2d75f
UZB	pools/UZBasePayablePoolV1.sol	e8e33c8455ed64b2ac52c4cc6b6879f528532c049e57c0f08d791820b9fd949e
UZV	pools/UZBasePoolV1.sol	ad16eed9bdf5feda86d8e6827b2f98bae38f861b87ba2de0db6bea7038f6887f
UZI	pools/UZBasicIncubatorPoolV1.sol	a759b8f8cbf7bb33593f60b2f5d2607f621782c3283f060816ffad9762178d31
UZK	pools/UZBasicRewardPoolV1.sol	9619ab8b52b7df82f5d7019ef1af69a66dec8594a0904a3ed3478784c24582da
UZM	pools/UZMainnetIncubatorPoolV1.sol	4008dae4cecd4c8d2bb04a61bae9f068ff817eb17f6c813eaf3c9f59aa96fedd

ID	file	SHA256 Checksum
UMR	pools/UZMainnetRewardPoolV1.sol	8bc1a429b7bdfcee1f2154bc16e39f2d7df0c44520c8da9bb6f7c8657c6244af
AHC	utils/ArrayHelper.sol	b819b0abaf0b6d7b81ae1e303f3546454f2c9ff42f635f021f005516a97dc299
UZA	utils/UZProAccess.sol	6323109bf7da5da95d49786c77a504c241febf2c77df36fb502cf3d25db16987

Findings



🟥 **Critical**	**0** (0.00%)
🟧 **Major**	**6** (35.29%)
🟨 **Medium**	**0** (0.00%)
🟨 **Minor**	**4** (23.53%)
🟦 **Informational**	**7** (41.18%)
🟩 **Discussion**	**0** (0.00%)

ID	Title	Category	Severity	Status
UZC-01	Centralization Risk	**Centralization / Privilege**	● **Major**	ⓘ **Acknowledged**
UZC-02	Logic Issue In Token ETH Price Calculation	Logical Issue	● Minor	✓ Resolved
UZC-03	Flash Loans Prevention	Logical Issue	● Informational	ⓘ Acknowledged
UZC-04	`SafeMath` Not Used	Mathematical Operations	● Minor	✓ Resolved
UZF-01	Iterate Over Array Upper Bond	Logical Issue	● Major	✓ Resolved
UZF-02	Lack Of Input Validation	Volatile Code	● Informational	✓ Resolved
UZF-03	Centralization Risk	**Centralization / Privilege**	● **Major**	ⓘ **Acknowledged**
UZI-01	Logic Issue In `payRewardPool()`	Logical Issue	● Minor	✓ Resolved
UZK-01	Dead Code	Logical Issue	● Informational	✓ Resolved
UZR-01	Comment Typo	Coding Style	● Informational	✓ Resolved
UZR-02	Proper Usage of `require` And `assert` Functions	Coding Style	● Informational	✓ Resolved
UZR-03	Logic Issue In `payRewardPool()`	Logical Issue	● Minor	✓ Resolved
UZR-04	Centralization Risk	**Centralization / Privilege**	● **Major**	ⓘ **Acknowledged**

ID	Title	Category	Severity	Status
UZV-01	Potential Initialization By Frontrunner	Logical Issue	Major	Resolved
UZV-02	Centralization Risk	**Centralization / Privilege**	**Major**	**Acknowledged**
UZV-03	Lack Of Input Validation	Volatile Code	Informational	Resolved
UZV-04	Lack Of Input Validation	Volatile Code	Informational	Acknowledged

UZC-01 | Centralization Risk

Category	Severity	Location	Status
Centralization / Privilege	● Major	projects/unizen/contracts/membership/UZProSale.sol: 49 9	ⓘ Acknowledged

Description

withdraw In the contract `UZProSale` and `UZBasePoolV1`, the role `owner` has the authority to call `withdrawTokens()` to withdraw any amount of token to any address. Any compromise to the `exchanger` account may allow the hacker to take advantage of this and withdraw all tokens from the contract.

Recommendation

We advise the client to carefully manage the `owner` account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol to be improved via a decentralized mechanism or via smart-contract-based accounts with enhanced security practices, e.g. Multisignature wallets.

Here are some feasible solutions that would also mitigate the potential risk:

- Time-lock with reasonable latency, i.e. 48 hours, for awareness on privileged operations;
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key;
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.

Alleviation

`[unizen]`: multisignature owner wallet will be adopted in the project.

UZC-02 | Logic Issue In Token ETH Price Calculation

Category	Severity	Location	Status
Logical Issue	● Minor	projects/unizen/contracts/membership/UZProSale.sol: 360	✓ Resolved

Description

Per comment in L359, the design of L360 is to calculate the token ETH price, which is not aligned with the implementation.

Recommendation

We advise the client to revise the L360 implementation and consider the following code snippet:

```
360  tokenETHPrice = _getUniswapPrice(_baseToken, _uniswapRouter.WETH());
```

Alleviation

`[unizen]`: We have `ethPrice` in line `uint256 ethPrice = _getUniswapPrice(_baseStableToken, _uniswapRouter.WETH());` where we fetch the current eth price. Then we have `tokenETHPrice` in line `tokenETHPrice = _getUniswapPrice(_baseToken, _baseToken);` where we want to know the price of ZCX in ETH. Their implementation would return the price of 1 ETH in ZCX.

UZC-03 | Flash Loans Prevention

Category	Severity	Location	Status
Logical Issue	● Informational	projects/unizen/contracts/membership/UZProSale.sol: 358, 360	ⓘ Acknowledged

Description

Flash loans are a way to borrow large amounts of money for a certain fee. The requirement is that the loans need to be returned within the same transaction in a block. If not, the transaction will be reverted.

An attacker can use the borrowed money as the initial funds for an exploit to enlarge the profit and/or manipulate the token price in the decentralized exchanges.

We find that the `_getUniswapPrice()` relies 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.

Recommendation

If a project requires price references, it needs to be careful of flash loans that might manipulate token prices. To prevent this from happening, we recommend the following.

1. Use Time-Weighted Average Price (TWAP). The TWAP represents the average price of a token over a specified time frame. If an attacker manipulates the price in one block, it will not affect too much on the average price.
2. If the business model allows, restrict the function caller to be a non-contract/EOA address.
3. Flash loans only allow users to borrow money within a single transaction. If the contract use cases allowed, force critical transactions to span at least two blocks.

Alleviation

`[unizen]`: We don't think it would be worth it, considering that people can only purchase one single membership token per address and we also have a safeguard that allows a specific difference between the current purchase price and the last purchase. So we thought that no one would benefit from a flash loan attack on this as they probably wouldn't gain anything from it and would probably cost more than just purchasing the membership.

UZC-04 | `SafeMath` Not Used

Category	Severity	Location	Status
Mathematical Operations	● Minor	projects/unizen/contracts/membership/UZProSale.sol: 183, 219	✓ Resolved

Description

This expression does not check arithmetic overflow. Such unsafe math operation may cause unexpected behavior if unusual parameters are given.

Recommendation

We advise the client to consider using `SafeMath` library of Openzeppelin library to prevent underflow.

Alleviation

`[unizen]`: Fixed in commit ef414c32ce55715662ec6cace47c850ea6781b7c

UZF-01 | Iterate Over Array Upper Bond

Category	Severity	Location	Status
Logical Issue	● Major	projects/unizen/contracts/UZFactoryV1.sol: 53, 57, 61	✓ Resolved

Description

`_poolCount` is a value to calculate the current amount of existing pools. When the value of `_poolCount` is less than 10, there will be an issue due to the access to the member that exceed max length of the array.

Recommendation

We would like to advise the client to revise the design and prevent any member access that exceeds the max length of the array. Reference: https://blog.soliditylang.org/2020/04/06/memory-creation-overflow-bug/

Alleviation

`[unizen]`: The function creates an in-memory array of the size for the current amount of active pools. We can have a maximum of 10 active pools at the same time, but perhaps they aren't on the correct ordered index, so they could be anywhere from 0-9, so we iterate from 0-9 and check if a pool exists. If it exists we add it to the in-memory array at index _idx, and then we increment it. PoolCount only is changed on creating/removing a pool, so this actually can never get out of sync.

The provided link describes an issue with memory arrays that have user-supplied length, which is not the case. The memory length is bound to the current active pool count which is owner-managed and changes on add/remove of pools. (https://blog.soliditylang.org/2020/04/06/memory-creation-overflow-bug/ explicitly mentions user input on this for the being vulnerable)

UZF-02 | Lack Of Input Validation

Category	Severity	Location	Status
Volatile Code	● Informational	projects/unizen/contracts/UZFactoryV1.sol: 255~259	✓ Resolved

Description

The given input `_startBlock`, and `_endBlock` are missing the sanity checks for ensuring the `_startBlock` is strictly less than `_endBlock`.

Recommendation

We advise the client to add the following input validation in the function `setStakingWindow()`:

```
require(_startBlock < _endBlock, "_startBlock is not strictly less than _endBlock");
```

Alleviation

`[unizen]`: Fixed in commit 1216f64a7ee0ed3dd25219c6c2c0d8c83014f6c2

UZF-03 | Centralization Risk

Category	Severity	Location	Status
Centralization / Privilege	● Major	projects/unizen/contracts/UZFactoryV1.sol: 271~275	ⓘ Acknowledged

Description

In the contract `UZFactoryV1`, the role `owner` has the authority to call function `withdrawTokens()`. Any compromise to the `owner` account may allow the hacker to take advantage of this and withdraw any type/amount of token to `owner`'s address.

Recommendation

We advise the client to carefully manage the `owner` account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol to be improved via a decentralized mechanism or via smart-contract-based accounts with enhanced security practices, e.g. Multisignature wallets.

Here are some feasible solutions that would also mitigate the potential risk:

- Time-lock with reasonable latency, i.e. 48 hours, for awareness on privileged operations;
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key;
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.

Alleviation

`[unizen]`: multisignature owner wallet will be adopted in the project.

UZI-01 | Logic Issue In `payRewardPool()`

Category	Severity	Location	Status
Logical Issue	● Minor	projects/unizen/contracts/pools/UZBasicIncubatorPoolV1.sol: 71	✓ Resolved

Description

Based on logic in L71, the current implementation is to calculate the total balance of current contact, i.e. `address(this)`. `address(0)` is mistakenly used to calculate this value.

Recommendation

We advise the client to consider fixing the issue with the following code snippet:

```
71  uint256 _balance = _token.balanceOf(address(this));
```

Alleviation

`[unizen]`: Fixed in commit 25c7f744e4f439689ed1a3341aa412204661c908

UZK-01 | Dead Code

Category	Severity	Location	Status
Logical Issue	● Informational	projects/unizen/contracts/pools/UZBasicRewardPoolV1.sol: 47	✓ Resolved

Description

Internal function `_safeClaim()` is not used in the contract `UZBasicRewardPoolV1` nor in any child contract as the contract itself is not inherited.

Recommendation

We advise the client to revise function `_safeClaim()` by reducing none essential implementation to save gas

Alleviation

`[unizen]`: `UZBasicRewardPoolV1` is an implementation of `UZBasePoolV1`, which as the function `_safeClaim()` used on the `userUpdate` and `UZBasicRewardPoolV1` overrides this function as is used. The needed implementation for the basic reward pool also differs in the implementation from the base function.

UZR-01 | Comment Typo

Category	Severity	Location	Status
Coding Style	● Informational	projects/unizen/contracts/UZRouterV1.sol: 146	✓ Resolved

Description

The linked comment contains a typo in its statement, namely `aloowance` should be `allowance`.

Recommendation

We advise to address the comment text.

Alleviation

`[unizen]`: Fixed in commit afa8c9c54c27ea1bbac18b28aa38d87326674061

UZR-02 | Proper Usage of `require` And `assert` Functions

Category	Severity	Location	Status
Coding Style	● Informational	projects/unizen/contracts/UZRouterV1.sol: 162, 164, 166, 199	✓ Resolved

Description

The `assert` function should only be used to test for internal errors, and to check invariants. The `require` function should be used to ensure valid conditions, such as inputs, or contract state variables are met, or to validate return values from calls to external contracts.

Recommendation

We advise the client using the `require` function, along with a custom error message when the condition fails, instead of the `assert` function

Alleviation

`[unizen]`: Fixed in commit 19a72d66211b9e69fa092727516a0705aba23ed3

UZR-03 | Logic Issue In `payRewardPool()`

Category	Severity	Location	Status
Logical Issue	● Minor	projects/unizen/contracts/UZRouterV1.sol: 150	✓ Resolved

Description

Based on logic in L150, it is supposed to store the specific user's current total balance. However, the current implementation does not align with the design.

Recommendation

We advise the client to consider fixing the issue with the following code snippet:

```
150  uint256 _currentUserTokenBalance = _paymentToken.balanceOf(_msgSender());
```

Alleviation

`[unizen]`: Fixed in commit 943dd0ac0a934dbf7287ae269833ea08dd8fb3d1

UZR-04 | Centralization Risk

Category	Severity	Location	Status
Centralization / Privilege	● Major	projects/unizen/contracts/UZRouterV1.sol: 275	ⓘ Acknowledged

Description

In the contract `UZRouter`, the role `owner` has the authority to call `emergencyWithdrawTokenFromRouter()` to withdraw any amount of token to any address. Any compromise to the `exchanger` account may allow the hacker to take advantage of this and withdraw all tokens from the contract.

Recommendation

We advise the client to carefully manage the `owner` account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol to be improved via a decentralized mechanism or via smart-contract-based accounts with enhanced security practices, e.g. Multisignature wallets.

Here are some feasible solutions that would also mitigate the potential risk:

- Time-lock with reasonable latency, i.e. 48 hours, for awareness on privileged operations;
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key;
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.

Alleviation

`[unizen]`: multisignature owner wallet will be adopted in the project.

UZV-01 | Potential Initialization By Frontrunner

Category	Severity	Location	Status
Logical Issue	● Major	projects/unizen/contracts/pools/UZBasePoolV1.sol: 52	✓ Resolved

Description

The initialization of pool contracts, including any pool contracts inherited from UZBasePoolV1.sol, can only be done by calling function `init()`. Theoretically, hackers can monitor the transaction of `init()` and front-run with a similar transaction to initialize the pool beforehand.

Recommendation

We advise the client to add an extra guard in the constructor:

```
constructor(address _newRouter, address _accessToken) UZProAccess(_accessToken) {
    _initiated = true;
}
```

As the `clone()` function only clones the runtime code of smart contract, which does not include the creation code in the constructor, this modification will not affect the current pool creation functionality in the contract `UZFactoryV1`

Alleviation

`[unizen]`: Fixed in commit f03fb66a878742979cd26d8886d70246596560af

UZV-02 | Centralization Risk

Category	Severity	Location	Status
Centralization / Privilege	● Major	projects/unizen/contracts/pools/UZBasePoolV1.sol: 590	ⓘ Acknowledged

Description

withdraw In the contract `UZProSale` and `UZBasePoolV1`, the role `owner` has the authority to call `withdrawTokens()` to withdraw any amount of token to any address. Any compromise to the `exchanger` account may allow the hacker to take advantage of this and withdraw all tokens from the contract.

Recommendation

We advise the client to carefully manage the `owner` account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol to be improved via a decentralized mechanism or via smart-contract-based accounts with enhanced security practices, e.g. Multisignature wallets.

Here are some feasible solutions that would also mitigate the potential risk:

- Time-lock with reasonable latency, i.e. 48 hours, for awareness on privileged operations;
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key;
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.

Alleviation

`[unizen]`: multisignature owner wallet will be adopted in the project.

UZV-03 | Lack Of Input Validation

Category	Severity	Location	Status
Volatile Code	● Informational	projects/unizen/contracts/pools/UZBasePoolV1.sol: 486~490	✓ Resolved

Description

The given input `_token`, and `_user` are missing the sanity checks for ensuring non-zero values

Recommendation

We advise the client to add the following input validation in the function `_updateUserData()`:

```
require(_token != address(0), "_token is a zero address");
require(_user != address(0), "_user is a zero address");
```

Alleviation

`[unizen]`: Fixed in commit af99392d81be03b2ac6a5d5ec51649955f289386

UZV-04 | Lack Of Input Validation

Category	Severity	Location	Status
Volatile Code	● Informational	projects/unizen/contracts/pools/UZBasePoolV1.sol: 393~397, 363~367	ⓘ Acknowledged

Description

The given input `_token` is missing the sanity checks for ensuring non-zero value

Recommendation

We advise the client to add the following input validation in the function `_updatePoolData()` and `_syncStakingData()`:

```
require(_token != address(0), "_token is a zero address");
```

Alleviation

`[unizen]`: The suggested input validation is not necessary, as it's also just used to check whether the updated token balance needs to be used or the synced token balance. And address(0) is a valid value to mark only updating from stored balances.

Appendix

Finding Categories

Centralization / Privilege

Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.

Mathematical Operations

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

Logical Issue

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

Volatile Code

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

Coding Style

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

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

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

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