hexens × VICINFT

Aug.23

SECURITY REVIEW REPORT FOR VICICOIN

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

AUDIT LED BY

VAHE KARAPETYAN

Co-founder / CTO | Hexens

Audit Starting Date	Audit Completion Date
21.08.2023	25.08.2023

METHODOLOGY

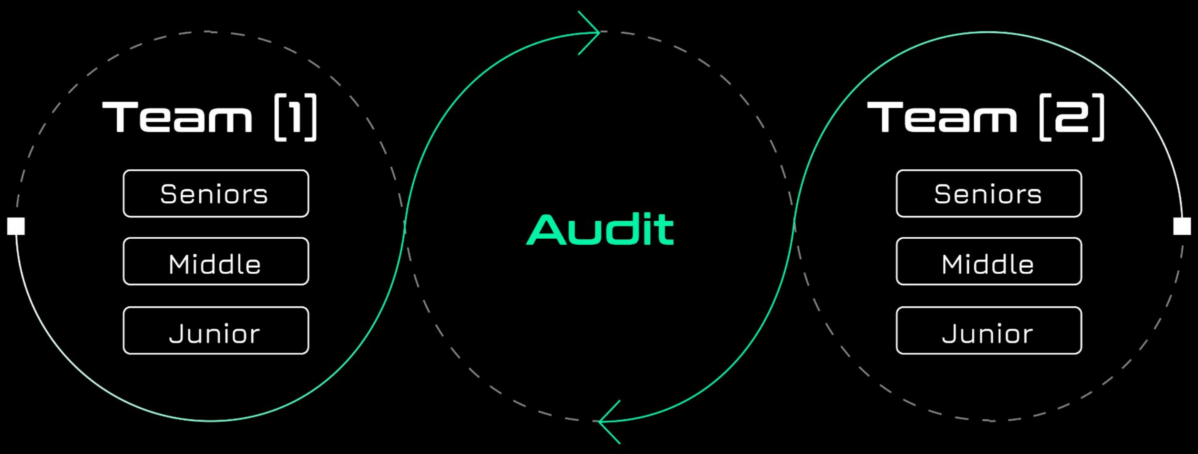
COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

Auditor*
Audit

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

IMPACT	PROBABILITY			
	Rare	Unlikely	Likely	Very Likely
Low / Info	Low / Info	Low / Info	Medium	Medium
Medium	Low / Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of vulnerabilities:

CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of the project. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

SCOPE

The analyzed resources are located on:
https://github.com/ViciNFT/ViciNFT-Token/commit/488e5981025a31cc7c787177d9d8440cf4255570

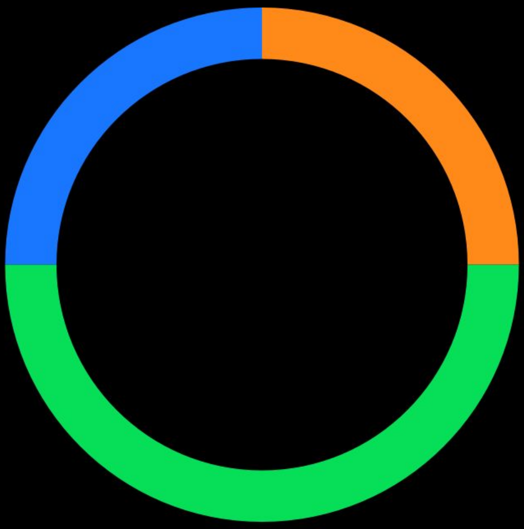
The issues described in this report were fixed in the following commit:
https://github.com/ViciNFT/ViciNFT-Token/commit/98074ec5649a345f1b60b5d21e77209c92336f95

SUMMARY

SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	0
MEDIUM	2
LOW	4
INFORMATIONAL	2

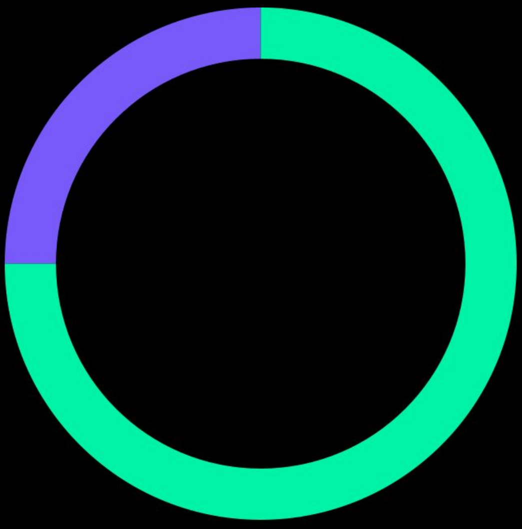
TOTAL: 8

SEVERITY



● Medium ● Low ● Informational

STATUS



● Fixed ● Acknowledged

WEAKNESSES

This section contains the list of discovered weaknesses.

VCNFT-15. CENTRALIZATION RISK FOR THE ACCESSSERVER CONTRACT

SEVERITY: Medium

PATH: AccessServer.sol

REMEDIATION: we strongly recommend fixing the deployment scripts to ensure that the deployer renounces ownership of the AccessServer contract and removes itself from the list of administrators. This will mitigate the security risk posed to the ViciCoin token holders in the event of a compromised deployer's private key

STATUS: fixed

DESCRIPTION:

The **AccessServer** contract is designed to be ownable, meaning that the deployer who calls the **initialize()** function becomes the owner of the contract. As the owner, they have the ability to call the **addAdministrator()** and **setSanctionsList()** functions with the **onlyOwner** modifier. Additionally, after initialization, the owner sets the Chainalysis sanctions oracle and adds the **MultiSigWalletWithSurvivorship** contract as an administrator of the **AccessServer** contract. It is expected that the owner will renounce ownership after these steps are completed.

However, if the owner fails to renounce ownership, there is a significant security risk to the **ViciCoin** token holders in the event that the deployer's private key is compromised. In such a scenario, a threat actor could potentially steal all **ViciCoin** tokens from the holders.

The compromised deployer, being an administrator of the **AccessServer** contract, could assign the **BANNED** global role to the victim. Consequently, the deployer can seize tokens by calling the **recoverSanctionedAssets()** function of the **ViciERC20** contract, as the victim's account is now banned.

Currently, the **AccessServer** contract deployed on the Polygon mainnet at the address **0x54D6970358A6BD193B7e98a92Ef0A1ecCcfBd704** has not renounced ownership from the **ViciCoin** deployer (EOA **0x4c1833Cb42FF2f07Bd332D04A2100ebB570A7112**). Furthermore, the **AccessServer** contract currently has two administrators: the **ViciCoin** deployer and the **MultiSigWalletWithSurvivorship** smart contract.

VCNFT-3. INCONSISTENT USE OF WHENNOTPAUSED MODIFIER

SEVERITY: Medium

PATH: see description

REMEDIATION: remove the whenNotPausedModifier from the transferFrom in ViciERC20 contract in order to allow users to perform operations as expected

STATUS: fixed

DESCRIPTION:

The function **transferFrom** in the ViciERC20 contract (L229) contains the **whenNotPauseModifier**, unlike **transfer** and **transferAndCall** which have very similar functionality and purpose. A user or smart contract might solely rely on **transferFrom** to perform operations depending on the setup they use. For example only keeping gas in one approved main address and using it to control all operations for many different tokens that are owned by difference address via **transferFrom**.

Even if we discount such scenario the logic behind using the **whenNotPauseModifier** on transferFrom is something that could prevent many users to transfer their funds when and how they intended to. This is a centralisation risk that can lead to reputational damage.

```
function transfer(
    address toAddress,
    uint256 amount
) public virtual override returns (bool) {
    tokenData.transfer( //@audit unsafe transfer?
        this,
        ERC20TransferData(_msgSender(), _msgSender(), toAddress, amount)
    );
    _post_transfer_hook(_msgSender(), toAddress, amount);
    return true;
}

/**
 * @dev See {IERC20-transferFrom}.
 * @dev See {safeTransferFrom}.
 *
 * Requirements
 *
 * - Contract MUST NOT be paused.
 * - `fromAddress` and `toAddress` MUST NOT be the zero address.
 * - `toAddress`, `fromAddress`, and calling user MUST NOT be banned.
 * - `_tokenId` MUST belong to `fromAddress`.
 * - Calling user must be the `fromAddress` or be approved by the `fromAddress`.
 * - `_tokenId` must exist
 *
 * @inheritdoc IERC20
 */
function transferFrom(
    address fromAddress,
    address toAddress,
    uint256 amount
) public virtual override whenNotPaused returns (bool) {
    tokenData.transfer(
        this,
        ERC20TransferData(_msgSender(), fromAddress, toAddress, amount)
    );
    _post_transfer_hook(fromAddress, toAddress, amount);
    return true;
}
```

```
/**
 * @inheritdoc IERC677
 */
function transferAndCall(
    address to,
    uint256 value,
    bytes calldata data
) public virtual override returns (bool success) {
    transfer(to, value);
    ERC677ReceiverInterface receiver = ERC677ReceiverInterface(to);
    receiver.onTokenTransfer(_msgSender(), value, data); //@audit-issue reentrancy?
    return true;
}
```

VCNFT-2. LACK OF TWO-STEP OWNERSHIP TRANSFER

SEVERITY: Low

PATH: Ownable.sol

REMEDIATION: implement a two-step transfer of ownership where newOwner has to claim the role after it is set by the current owner in order to finalise the transfer

STATUS: acknowledged

DESCRIPTION:

Despite checking against a zero address, the **transferOwnership()** function does not prevent the transfer of ownership to an inaccessible address.

If a typo is made or **newOwner** keys are lost during the transfer period, all owner functionality will be lost.

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _transferOwnership(newOwner);
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Internal function without access restriction.
 */
function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

VCNFT-5. DUPLICATE FUNCTIONALITY FOR CHECKING IF ADDRESS HAS ROLE

SEVERITY: Low

PATH: see description

REMEDIATION: eliminate the function _hasRole and using hasLocalRole instead when calling hasGlobalRole. This will decrease total bytecode size and gas cost on deployment, as well as making the contract more streamlined and easy to understand.

STATUS: fixed

DESCRIPTION:

The functions **hasLocalRole** (L548) and **_hasRole** (L631) in the **AccessServer** contract are identical. With the only difference of one being **public** and the other being **internal**. Having more than one function with the exact same functionality is gas inefficient and hurts readability from a developing, auditing and user perspective.

```
function hasLocalRole(
    address resource,
    bytes32 role,
    address account
) public view virtual override returns (bool) {
    return managedResources[resource].roles[role].members[account];
}
```

```
function _hasRole(
    address resource,
    bytes32 role,
    address account
) internal view virtual returns (bool) {
    return managedResources[resource].roles[role].members[account];
}
```

VCNFT-13. THE GETTRANSACTIONIDS FUNCTION MIGHT RETURN INCORRECT RESULT

SEVERITY: Low

PATH: MultiSigWalletWithSurvivorship.sol

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The **getTransactionIds()** function calculates incorrectly an array of transaction IDs with a **status** when the **from** parameter is non-zero.

More precisely, when **from=0** and **to=getTransactionCount(all)** the result is correctly returned and includes all transaction IDs.

However, for other cases, an incorrect array of transaction IDs is returned, more precisely:

- when **from=1** and **to=getTransactionCount(all)**, transaction IDs aren't missing in the result array, but the last element is **0**.
- when **from=1** and **to=getTransactionCount(all)**, the last transaction ID is missing and the result contains two **0** elements at the end of the array.

This happens because:

- The first loop contains incorrect condition **i <= transactionCount && count <= maxResults**.

- An incorrect array length is used for **_transactionIds array _transactionIds = new uint256[](count)**.

```
function getTransactionIds(
    uint256 from,
    uint256 to,
    TransactionStatus status
) public view virtual returns (uint256[] memory _transactionIds) {
    uint256[] memory transactionIdsTemp = new uint256[](transactionCount);
    uint256 count = 0;
    uint256 i;
    uint256 maxResults = to - from;
    for (i = 1; i <= transactionCount && count <= maxResults; i++)
        if (
            status == TransactionStatus.EVERY_STATUS ||
            transactions[i].status == status
        ) {
            transactionIdsTemp[count] = i;
            count += 1;
        }
    _transactionIds = new uint256[](count);
    for (i = from; i < count; i++)
        _transactionIds[i - from] = transactionIdsTemp[i];
}
```

The **getTransactionIds()** should return correct result when the **from** parameter is not **0**.

The following code returns the correct array of transaction IDs:

```
function getTransactionIds(
    uint256 from,
    uint256 to,
    TransactionStatus status
) public view virtual returns (uint256[] memory _transactionIds) {
    uint256[] memory transactionIdsTemp = new uint256[](transactionCount);
    uint256 count = 0;
    uint256 i;
    uint256 maxResults = to - from;
    for (i = 1; i <= transactionCount && count <= to; i++)
        if (
            status == TransactionStatus.EVERY_STATUS ||
            transactions[i].status == status
        ) {
            transactionIdsTemp[count] = i;
            count += 1;
        }
    _transactionIds = new uint256[](maxResults);
    for (i = from; i < count; i++)
        _transactionIds[i - from] = transactionIdsTemp[i];
}
```

VCNFT-14. ADD DEFAULT CONSTRUCTOR THAT CALLS _DISABLEINITIALIZERS()

SEVERITY: Low

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The protocol uses upgradable contracts. There is a danger of calling the **initialize()** function directly on the implementation contract behind proxy. In such case if the implementation calls self-destruct or performs delegate calls it's possible to delete the implementation leaving contract bricked.

Contracts should include a default constructor **calling_disableInitializers()** function of **Initializable.sol**.

VCNFT-9. DEFAULT VALUE INITIALIZATION

SEVERITY: Informational

PATH: MultiSigWalletWithSurvivorship.sol

REMEDIATION: see description

STATUS: acknowledged

DESCRIPTION:

We found that at the following locations in the code, there are variables initialized to their default value. This is already the default behavior of Solidity and it becomes therefore redundant and a waste of gas.

1. **count** L728, L806, 857;
2. **inactiveCutoff** L729, L753.

```
uint256 count = 0;
uint256 inactiveCutoff = 0;
```

Remove the initialization to a default value in favor of saving gas.

For example:

```
uint256 count;
uint256 inactiveCutoff;
```

VCNFT-7. CONSTANT VARIABLE SHOULD BE MARKED AS PRIVATE

SEVERITY: Informational

PATH: MultiSigWalletWithSurvivorship.sol

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The **MAX_OWNER_COUNT** parameter on line 175 should also be **private**. Setting constant to private will save deployment gas. This is because the compiler won't have to create non-payable getter functions for deployment calldata, won't need to store the bytes of the value outside of where it's used, and won't add another entry to the method ID table. If necessary, the value can still be read from the verified contract source code.

```
uint256 public constant MAX_OWNER_COUNT = 50;
```