Quantstamp Security Assessment Certificate

May 25th 2022 — Quantstamp Verified

CapsuleNFT

This audit report was prepared by Quantstamp, the leader in blockchain security.

Executive Summary

Type	NFT
Auditors	Cristiano Silva, Research Engineer Rabib Islam, Research Engineer Guillermo Escobero, Security Auditor
Timeline	2022-04-04 through 2022-04-18
EVM	London
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	User facing documentation Development documentation
Documentation Quality	Medium
Test Quality	Low

Source Code

Repository	Commit
capsule	67caf73

Goals

- Find logical bugs
- Match code against specification
- Find possible exploits

Total Issues	**16** (9 Resolved)
High Risk Issues	**3** (1 Resolved)
Medium Risk Issues	0 (0 Resolved)
Low Risk Issues	0 (0 Resolved)
Informational Risk Issues	**13** (8 Resolved)
Undetermined Risk Issues	0 (0 Resolved)

FAST RESPONSE TIMES

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances.
Informational	The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
? Undetermined	The impact of the issue is uncertain.

Unresolved	Acknowledged the existence of the risk, and decided to accept it without engaging in special efforts to control it.
Acknowledged	The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).
Fixed	Adjusted program implementation, requirements or constraints to eliminate the risk.
Mitigated	Implemented actions to minimize the impact or likelihood of the risk.

Summary of Findings

We raised a total of 16 issues, with two of them rated as high severity ones. During the reaudit phase, the CapsuleNFT has demonstrated that 3 issues were false-positives. In general, the issues are related to validation of input/output parameters, and some best practices (such as emitting events and implementation of interfaces). The code is easy to follow, which shows a good programming style. The `blacklist` feature is still not used by the application. The CapsuleNFT provided fast response in addressing the issues (or acknowledging them, when the issue has no practical solution or it is the intended behavior of the application). In terms of tests, we recommend code coverage above 90%.

ID	Description	Severity	Status
QSP-1	Access control vulnerability in `CapsuleFactory.updateCapsuleCollectionOwner(...)`	High	Fixed
QSP-2	`CapsuleMinter` does not check whether pre- and post-deposit ERC20 amounts are the same	High	Acknowledged
QSP-3	Anyone can call `initialize`	High	Acknowledged
QSP-4	Collection tax and mint tax can be set to any value	Informational	Fixed
QSP-5	Consider the use of `_safeMint(...)` instead of `_mint(...)`	Informational	Fixed
QSP-6	Ownership can be renounced	Informational	Fixed
QSP-7	Some functions not sanitizing input addresses	Informational	Fixed
QSP-8	Missing the emission of events for important state transitions	Informational	Fixed
QSP-9	Arbitrary number of tokens in a single collection	Informational	Fixed
QSP-10	Incorrect inheritance	Informational	Fixed
QSP-11	Clone-and-Own	Informational	Acknowledged
QSP-12	Unused `blacklist` functionality	Informational	Acknowledged
QSP-13	Privileged roles and ownership	Informational	Acknowledged
QSP-14	Possibility of minting more than one NFT having the same `URI`	Informational	Acknowledged
QSP-15	The same address can be at the `whitelist` and the `blacklist` simultaneously	Informational	Fixed
QSP-16	Users in minting whitelist can send Ether	Informational	Acknowledged

Quantstamp Audit Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

DISCLAIMER: The scope of the audit is the files: a) CapsuleFactory.sol; b) CapsuleFactoryStorage.sol; c) Errors.sol; d) Capsule.sol; e) CapsuleMinter.sol; f) CapsuleMinterStorage.sol.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

The Quantstamp auditing process follows a routine series of steps:

1. Code review that includes the following
   i. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
   ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
   i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
   ii. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Toolset

The notes below outline the setup and steps performed in the process of this audit.

Setup

Tool Setup:

- Slither v0.8.2

Steps taken to run the tools:

1. Installed the Slither tool: `pip install slither-analyzer`
2. Run Slither from the project directory: `slither .`

Findings

QSP-1 Access control vulnerability in `CapsuleFactory.updateCapsuleCollectionOwner(...)`

Severity: *High Risk*

Status: Fixed

File(s) affected: `contracts/CapsuleFactory.sol`

Description: Although the inline comment states that the function `CapsuleFactory.updateCapsuleCollectionOwner(...)` can be called only by the `Capsule Collection` as part of the `transferOwnership(...)`, we notice that this function has `external` visibility and, apparently, no access control. This function transfers the ownership from `_previousOwner` to the `_newOwner` without any consentiment from any of the parties. The function is reproduced below.

```
    function updateCapsuleCollectionOwner(address _previousOwner, address _newOwner) external {
        address _capsule = _msgSender();
        require(isCapsule[_capsule], Errors.NOT_CAPSULE);
        require(capsulesOf[_previousOwner].remove(_capsule), Errors.ADDRESS_DOES_NOT_EXIST);
        require(capsulesOf[_newOwner].add(_capsule), Errors.ADDRESS_ALREADY_EXIST);
        emit CapsuleOwnerUpdated(_capsule, _previousOwner, _newOwner);
    }
```

Exploit Scenario: In its actual form, the function allows the `owner` of Capsule A to change the `owner`of Capsule B, even when Capsule B is not owned by the `owner`of Capsule A. In other words, in its actual form, the function allows that the `owner` of any given Capsule transfers the ownership of all the other Capsules to him/herself. This is a serious issue that must be addressed by the Capsule NFT team.

Recommendation: Double-check the function and consider to:

1. Implement an access control strategy in order to guarantee that the `_previousOwner` (and the `_newOwner`) acknowledge the transfer;
2. Guarantee that only the `owner` of Capsule A can call this function to change the owner of Capsule A, and any other Capsule;
3. Check if the input parameters are different from `address(0x0)`;
4. Clear specify this use case. Is it necessary to validate if the `_newOwner` is a whitelisted or blacklisted address before transfering the ownership?

Update: Response from the cliente: From our review (and tests added in commit 0f35e1d38d2170781716f37136900aca92e691af), we do not see an issue in this comment, even considering the method scope is external. Note the first two lines: the first which assigns the value of the _msgSender(). The second then checks to see if the _msgSender() is a Capsule himself (using the isCapsule mapping). The owner of Capsule A would not be able to change. the owner of Capsule B because he would only be able to call the method through the Capsule A contract (using method transferOwnership). Unless he was the owner of both Capsule A and Capsule B, he would not be able to change ownership of either singularly.
Given the clarifications made by the CapsuleNFT, we consider this issue as false-positive.

QSP-2 `CapsuleMinter` does not check whether pre- and post-deposit ERC20 amounts are the same

Severity: *High Risk*

Status: Acknowledged

File(s) affected: `CapsuleMinter.sol`

Description: In functions `mintSingleERC20Capsule(...)` and `mintMultiERC20Capsule(...)`, there are no checks to determine whether the pre- and post-deposit ERC20 amounts are the same. This can be an issue when special, deflationary ERC20 contracts burn some percentage of the tokens in every token transfer. Such behaviour in `CapsuleMinter` is contrary to the intent in the whitepaper, which states "the Capsule Protocol ensures that the exact same amount of token deposited is found at the contract post-mint".
The code of the function `mintMultiERC20Capsule(...)` is presented below:

```
function mintMultiERC20Capsule(
       address _capsule, address[] memory _tokens, uint256[] memory _amounts, string memory _uri, address _receiver
   ) external payable nonReentrant onlyCollectionMinter(_capsule) checkStatus {
       uint256 tokensLength = _tokens.length;
       uint256 amountsLength = _amounts.length;

       require(tokensLength <= TOKEN_TYPE_LIMIT, Errors.INVALID_TOKEN_ARRAY_LENGTH);
       require(amountsLength <= TOKEN_TYPE_LIMIT, Errors.INVALID_AMOUNT_ARRAY_LENGTH);
       require(tokensLength == amountsLength, Errors.LENGTH_MISMATCH);

       // get the current top counter
       uint256 _id = ICapsule(_capsule).counter();
       // Some tokens, like USDT, may have a transfer fee, so we want to record actual transfer amount
       uint256[] memory _actualAmounts = new uint256[](amountsLength);
       // loop assumes that the token address and amount is mapped to the same index in both arrays
       // meaning: the user is sending _amounts[0] of _tokens[0]
       for (uint256 i; i < tokensLength; i++) {
           address _token = _tokens[i];
           uint256 _amount = _amounts[i];

           require(_amount != 0, Errors.INVALID_TOKEN_AMOUNT);
           require(_token != address(0), Errors.INVALID_TOKEN_ADDRESS);

           // transfer tokens from caller to contract
           _actualAmounts[i] = _depositToken(IERC20(_token), _msgSender(), _amount);
       }

       // then add user data into the contract (tie Capsule NFT to input):
       multiERC20Capsule[_capsule][_id].tokenAddresses = _tokens;
       multiERC20Capsule[_capsule][_id].tokenAmounts = _actualAmounts;

       // lastly, mint the Capsule NFT
       ICapsule(_capsule).mint(_receiver, _uri);

       emit MultiERC20CapsuleMinted(_receiver, _capsule, _tokens, _amounts, _uri);
   }
```

The code of the function `mintSingleERC20Capsule(...)` (without inline comments) is presented below:

```
function mintSingleERC20Capsule(
        address _capsule, address _token, uint256 _amount, string memory _uri, address _receiver
    ) external payable nonReentrant onlyCollectionMinter(_capsule) checkStatus {
        require(_amount != 0, Errors.INVALID_TOKEN_AMOUNT);
        require(_token != address(0), Errors.INVALID_TOKEN_ADDRESS);

        // get the current top counter
        uint256 id = ICapsule(_capsule).counter();

        // transfer tokens from caller to contract
        uint256 _actualAmount = _depositToken(IERC20(_token), _msgSender(), _amount);

        // then, add user data into the contract (tie NFT to value):
        // - set the ID of the Capsule NFT at counter to map to the passed in tokenAddress
        // - set the ID of the Capsule NFT at counter to map to the passed in tokenAmount
        singleERC20Capsule[_capsule][id].tokenAddress = _token;
        singleERC20Capsule[_capsule][id].tokenAmount = _actualAmount;
        // lastly, mint the Capsule NFT (minted at the current counter (obtained above as id))
        ICapsule(_capsule).mint(_receiver, _uri);

        emit SingleERC20CapsuleMinted(_receiver, _capsule, _token, _amount, _uri);
    }
```

Recommendation: Conclude as to whether it should be possible to mint capsule NFTs with deflationary ERC20 tokens. If not, add checks to ensure that the pre- and post-deposit token amounts are the same.

Update: Response from the client: This issue stems from an older version of documentation presented. The old whitepaper, which stated: “the Capsule Protocol ensures that the exact same amount of token deposited is found at the contract post-mint”, is no longer true. The CapsuleMinter should be able to support deflationary ERC20 tokens (or tokens, such as USDT, which can potentially enact a ‘fee’ on transfer). Our internal method, _depositToken, allows for deflationary/fee tokens to be properly inserted into Capsules.

QSP-3 Anyone can call `initialize`

Severity: *High Risk*

Status: Acknowledged

File(s) affected: `CapsuleMinter.sol`, `CapsuleFactory.sol`

Description: Functions dedicated to initializing contracts can be initialized by any address, leaving them open to exploitation. Below we reproduce the code from `CapsuleMinter.initialize(...)`, where we notice that the only ckeck is if `_factory != address(0)`. As long as we call this function with the input parameter `_factory != 0` we can make the function run without problems.

```
    function initialize(address _factory) external initializer {
        require(_factory != address(0), Errors.ZERO_ADDRESS);
        _initializeOwnable();
        factory = ICapsuleFactory(_factory);
        capsuleMintTax = 0.001 ether;
    }
```

Now, we present the code of the function `CapsuleFactory.initialize(...)`, which also does not present any access control, and can be called by any address.

```
    function initialize() external initializer {
        _initializeOwnable();
        capsuleCollectionTax = 0.025 ether;
        taxCollector = _msgSender();
    }
```

Recommendation: Wherever possible, use a constructor to initialize contracts. In cases where this is not possible, deploy and initialize contracts using deployer contracts.

Update: Response from the client: It is correct that any address may call the initialize function as mentioned. It would be possible to create a helper contract which would deploy the contract and call the initialize() function straight after. We opt not to deploy an extra contract to do so, simply because of the extra incurred gas cost and the extremely small chance anyone would call the initialize function before us, after deployment. Our setup, viewable in deploy/CapsuleMinter.ts (as an example), deploys the proxy contract and immediately after will call the initialize function. Even in the unlikely case someone else calls the initialize function before us, we would spot the error (our method call would fail) and we would simply redeploy the contract again, re-calling the method on the newly deployed contract.

QSP-4 Collection tax and mint tax can be set to any value

Severity: *Informational*

Status: Fixed

File(s) affected: `contracts/CapsuleFactory.sol`, `contracts/CapsuleMinter.sol`

Description: The function `CapsuleFactory.updateCapsuleCollectionTax(...)` can set arbitrary values for the collection tax. Similarly, the function `CapsuleMinter.updateCapsuleMintTax(...)` can also set arbitrary values for the mint tax. That means that there is no upper bound for the taxes, which can lead to reduced gain for users, or even fund losses. It is important to clearly advertise to users the taxes that they will have to face, and also impose thresholds in order to keep the application financially healthy, while guaranteeing the interest of users by providing interesting and competitive gains for users. The function `updateCapsuleCollectionTax(...)` is shown below:

```
    function updateCapsuleCollectionTax(uint256 _newTax) external onlyOwner {
        require(_newTax != capsuleCollectionTax, Errors.SAME_AS_EXISTING);
        emit CapsuleCollectionTaxUpdated(capsuleCollectionTax, _newTax);
        capsuleCollectionTax = _newTax;
    }
```

The function `updateCapsuleCollectionTax(...)` is shown below:

```
    function updateCapsuleMintTax(uint256 _newTax) external onlyOwner {
        require(_newTax != capsuleMintTax, Errors.SAME_AS_EXISTING);
        emit CapsuleMintTaxUpdated(capsuleMintTax, _newTax);
        capsuleMintTax = _newTax;
    }
```

Exploit Scenario: Setting very high taxes can consume the whole investment of users during the mint of the first NFT. Such can be achieved by imposing a mint tax of 100% of the asset.

Recommendation: Impose lower and upper bound for these taxes. In the future, consider creating a mechanism that balances the financial health of the application and the users financial gains, which can be performed off-chain for reducing the operational cost of the application.

Update: Response from the client: fixed in commit (67caf73be8026d986f516786a8a7cb4e64475e8b).

QSP-5 Consider the use of `_safeMint(...)` instead of `_mint(...)`

Severity: *Informational*

Status: Fixed

File(s) affected: `contracts/Capsule.sol`

Description: The function `Capsule.mint(...)` calls `IERC721._mint(...)` instead of `IERC721._safeMint(...)`. However, OpenZeppelin discourages the use of `IERC721._mint(...)`. The `_safeMint(...)` flavor of minting causes the recipient of the tokens, if it is a smart contract, to react upon receipt of the tokens. Here is a general consideration to help you decide which one to use:

1. If the application is paying for the minting of tokens, consider use `_mint(...)`. The `_safeMint(...)` might cost you an arbitrary amount of money because of choices made by the recipient of the tokens. This is enough to deter you from considering it;
2. If the other party is paying for the minting of tokens and you expect buyers to be composing functionality with smart contracts, use `_safeMint(...)`. There is marginal benefit in allowing the extra features with certain smart contracts;
3. If the other party is paying and you expect EOA to buy tokens, then use `_mint(...)`. The extra features in `_safeMint(...)` are not expected. The extra cost from using `_safeMint(...)` is non-zero.

The function `Capsule.mint(...)` is shown below:

```
    function mint(address _account, string memory _uri) external onlyMinter {
        require(!isCollectionLocked(), 'collection is locked');
        _mint(_account, counter);
        _setTokenURI(counter, _uri);
        counter++;
    }
```

Recommendation: Use `IERC721._safeMint(...)` in case you are interacting with smart contracts and the cost of minting belongs to the other party.

Update: Response from the client: fixed in commit (67caf73be8026d986f516786a8a7cb4e64475e8b).

QSP-6 Ownership can be renounced

Severity: *Informational*

Status: Fixed

File(s) affected: `all ownable contracts`

Description: The `Ownable.sol` has the function `renounceOwnership()`. Although it can only be called by the `owner`, such a function leaves the contract without an owner, which surely compromises any ability to manage the contract. Several critical operations can only be performed by the `owner`. In case the `owner` is accidentally removed, the operations of a) adding and removing users in the whitelist and blacklist are blocked, b) the creation of new Capsules, c) the definition of entities responsible for minting Capsules, d) the collection of taxes/fees, e) the minting of new tokens, and f) the operations of lock collections, all this considering only the files that are in the scope of this audit. In practical terms, it would not be an exaggeration to say that the application would become unusable.

Recommendation: Overwrite this function in order to have zero risk of leaving the contract without an owner and thereby possibly losing all funds vested in the contract.

Update: Response from the client: fixed by using new Governable contract, can be found at access/Governable.sol. Commit hash e53ae79b8a287d0b7791d9419639968b76221520.

QSP-7 Some functions not sanitizing input addresses

Severity: *Informational*

Status: Fixed

File(s) affected: `contracts/CapsuleFactory.sol`

Description: Some functions receiving input addresses do not check if they are different from `address(0x0)`, nor that the given addresses refer to a contract (when applicable). Setting addresses incorrectly can lead to incorrect setups that will prevent contracts from working as expected. A non-exhaustive list of functions not sanitizing input addresses is presented below:

```
- CapsuleFactory.addToWhitelist(...)
- CapsuleFactory.addToBlacklist(...) (only useful for reducing the storage space)
- CapsuleFactory.updateCapsuleCollectionOwner(...)
```

Exploit Scenario:

1. When considering the functions `CapsuleFactory.addToWhitelist(...)` and `CapsuleFactory.addToBlacklist(...)`, the contract can waste the expensive storage space for

keeping `address(0x0)` in the `whitelist` and/or `blacklist`.

2. When considering the function `CapsuleFactory.updateCapsuleCollectionOwner(address _previousOwner, address _newOwner)`, the owner of one Capsule can set the address of another Capsule to `address(0x0)`, making this Capsule non-operational, especially if we consider that there is no check that the caller is the real owner of the Capsule that he/she is updating the ownership. The code of the function is presented below, where we note that there is no check that the `_previousOwner` really owns that specific Capsule, and also that there is no check the `_newOwner` is different from `address(0x0)`, or even if the `_newOwner` is blacklisted.

Recommendation: Unless input addresses are known to be correct (e.g., controlled by deployment scripts), we suggest verifying that:

1. They are different than ` address(0x0);
2. They are indeed contracts (when applicable);
3. When considering the function `CapsuleFactory.updateCapsuleCollectionOwner(address _previousOwner, address _newOwner)`: a) Check if the `_previousOwner` is the owner of the target Capsule; b) Check if the `_newOwner` is different from `address(0x0)`; c) Check if the `_newOwner` is not blacklisted; d) Depending upon the business logic, also check if the `_previousOwner` is not blacklisted.

Update: Response from the client: Fixed in commit (67caf73be8026d986f516786a8a7cb4e64475e8b). updateCapsuleCollectionOwner can only be called by transferOwnership method of a valid Capsule. OZ transferOwnership has non-zero address check, so no update there.

QSP-8 Missing the emission of events for important state transitions

Severity: *Informational*

Status: Fixed

File(s) affected: `contracts/CapsuleFactory.sol`, `contracts/Capsule.sol`, `contracts/CapsuleMinter.sol`

Description: Ideally, every important state change should be recorded by emitting events describing the past and the new state. A non-exhaustive list of functions that could emit events for improving post-deployment contracts monitoring is presented below:

```
- CapsuleFactory.initialize()
- CapsuleFactory.addToWhitelist(...)
- CapsuleFactory.removeFromWhitelist(...)
- CapsuleFactory.addToBlacklist(...)
- CapsuleFactory.removeFromBlacklist(...)
- CapsuleFactory.flushTaxAmount(...)
- CapsuleFactory.updateCapsuleCollectionTax(...) not showing the past fee
- Capsule.lockCollectionCount(...)
- CapsuleMinter.initialize(...)
- CapsuleMinter.addToWhitelist(...)
- CapsuleMinter.removeFromWhitelist(...)
- CapsuleMinter.flushTaxAmount()
- CapsuleMinter.updateCapsuleMintTax(...) not showing the past mint tax
```

Exploit Scenario: If a hack, an operation error, or even a malicious activity occurs, the application management team will not have enough information to quickly react to the situation and recover the system to its original state. Being in such a situation can be quite uncomfortable, lead to loss of funds by users, and ultimately result in collapse of the credibility.

Recommendation: Consider emitting events for recording any state transitions.

Update: Response from the client: fixed in commit (67caf73be8026d986f516786a8a7cb4e64475e8b).

QSP-9 Arbitrary number of tokens in a single collection

Severity: *Informational*

Status: Fixed

File(s) affected: `contracts/Capsule.sol`

Description: The function `Capsule.lockCollectionCount(...)` does not provide an upper bound on the number of tokens that each collection can have. This can lead to gas concerns in use cases where the application has to go over the entire collection. We could not find any loop over the entire collection in the piece of code audited. However, it is safer to keep some limits in order to avoid Denial-of-Service attacks.

Exploit Scenario: This vulnerability can be exploit to implement some sort of Denial-of-Service attack in the application.

Recommendation: Consider implementing an upper bound on the number of distinct tokens. Otherwise, just be careful when looping over the entire collection.

Update: Response from the client: This is not an issue as lockCollectionCount locks the collection at a given collection count. And there is no need to iterate over these collections for any reason. We do maintain user, his/her NFT and token count in data structures separate.
Given the clarification made by the CapsuleNFT, we consider this issue as false-positive.

QSP-10 Incorrect inheritance

Severity: *Informational*

Status: Fixed

File(s) affected: `contracts/CapsuleFactory.sol`, `contracts/CapsuleFactoryStorage.sol`, `contracts/CapsuleMinter.sol`, `contracts/CapsuleMinterStorage.sol`

Description: - `CapsuleFactoryStorage` inherits from `ICapsuleFactory` but does not implement any function.

- `CapsuleMinterStorage` inherits from `ICapsuleMinter` but does not implement any function.

This can lead to calls to not-implemented functions.

Recommendation: Consider that `CapsuleFactory` should inherit from `ICapsuleFactory` and `CapsuleFactoryStorage`. `CapsuleFactoryStorage` should remain independent and not inherit from `ICapsuleFactory`. Same idea is applicable to `ICapsuleMinter`, `CapsuleMinter` and `CapsuleMinterStorage`.

Update: Response from the client: We couldn't find any unimplemented function from interface. We also acknowledge that interfaces were not complete and has been updated with all the functions. Also recommended inheritance arrangement is not possible hence no update on the existing inheritance tree.
Given the clarification made by the CapsuleNFT team, we consider this issue as false-positive.

QSP-11 Clone-and-Own

Severity: *Informational*

Status: Acknowledged

File(s) affected: `./openzeppelin/*`

Description: The clone-and-own approach involves copying and adjusting open source code at one's own discretion. From the development perspective, it is initially beneficial as it reduces the amount of effort. However, from the security perspective, it involves some risks as the code may not follow the best practices, may contain a security vulnerability, or may include intentionally or unintentionally modified upstream libraries. Rather than the clone-and-own approach, a good industry practice is to use the Truffle framework for managing library dependencies. This eliminates the clone-and-own risks yet allows for following best practices, such as, using libraries.

Recommendation: Rather than the clone-and-own approach, a good development practice is to use the Truffle or Hardhat framework for managing library dependencies. This eliminates the clone-and-own risks yet allows for following best practices, such as, using libraries.

Update: Response from the client: as Capsule project is using proxy and we do not want any new updated from upstream as updates may corrupt proxy storage.

QSP-12 Unused `blacklist` functionality

Severity: *Informational*

Status: Acknowledged

File(s) affected: `contracts/CapsuleFactory.sol`, `contracts/CapsuleFactoryStorage.sol`, `contracts/ICapsuleFactory.sol`, `contracts/Errors.sol`

Description: Functions to manage the `blacklist` are implemented in the codebase, but are not used in the audited contracts. Perhaps, these functions are used in contracts outside the scope of this audit. On the other hand, in this report we indicate that the transfer of Capsule's ownership in function `CapsuleFactory.updateCapsuleCollectionOwner(address _previousOwner, address _newOwner)` could check if the `_newOwner` is contained in the `blacklist`.

Recommendation: The `whitelist` and `blacklist` features do not seem to be used at their full potential. We recommend the development team to review all code in order to check if these features are applied in all required places. Maybe we do not see an extensive use of these features because we are auditing only a subset of the whole system.

Update: Response from the client: the initial plan was to use blacklist in this release which we defer for now. Sure, we can remove code for now or keep it as is and use it in next release. At this moment we want to keep these, if we think otherwise, we will remove those and update you the same in future iterations.

QSP-13 Privileged roles and ownership

Severity: *Informational*

Status: Acknowledged

File(s) affected: `contracts/*`

Description: Smart contracts will often have `owner` variables to designate the person with special privileges to make modifications to the smart contract.

Recommendation: This centralization of power needs to be made clear to the users, especially depending on the level of privilege the contract allows to the owner.

Update: Response from the client: there are some business logic which needs action by privileged roles such as maintaining whitelist/blacklist. Also, the Capsule community may decide to go with DAO and eventually have a governor and this will require privileged roles.

QSP-14 Possibility of minting more than one NFT having the same `URI`

Severity: *Informational*

Status: Acknowledged

File(s) affected: `contracts/Capsule.sol`

Description: The function `Capsule.mint(...)` does not validate the input parameters `_account` and `_uri`. Thus, it is possible to mint more than one NFT having the same URI, and it is also possible to mint NFTs for `address(0x0)`. Finally, it is also possible to mint NFTs having empty URIs. The function is reproduced below:

```
    function mint(address _account, string memory _uri) external onlyMinter {
        require(!isCollectionLocked(), 'collection is locked');
        _mint(_account, counter);
        _setTokenURI(counter, _uri);
        counter++;
    }
```

Recommendation:

1. Validate the input parameters;
2. The URI can be checked off-chain for its uniqueness.

Update: Response from the client: overall, there is not really anything we can do on-chain. Also, we are fine with multiple NFT's using same URI

QSP-15 The same address can be at the `whitelist` and the `blacklist` simultaneously

Severity: *Informational*

Status: Fixed

File(s) affected: `contracts/CapsuleFactory.sol`

Description: As one can note from the code snippet presented below, the very same address can be added to both `whitelist` and the `blacklist` at the same time. This use case and the code itself is not clear concerning of what happens if an address is added to both lists. The code of both functions is reproduced below.

```
    function addToWhitelist(address _user) external onlyOwner {
        require(whitelist.add(_user), Errors.ADDRESS_ALREADY_EXIST);
    }

    function addToBlacklist(address _user) external onlyOwner {
        require(blacklist.add(_user), Errors.ADDRESS_ALREADY_EXIST);
    }
```

It is unclear what happens to the system if the same address is added to both lists. Perhaps the best solution is to ensure that the same user cannot integrate both lists. Moreover, it is unclear what happens if a Capsule owner is added to the `blacklist`, and how such an action may damage the users having funds locked in the Capsule. The code of the functions `addToWhitelist(...)` and `addToBlacklist(...)` are reproduced below.

```
    function addToWhitelist(address _user) external onlyOwner {
        require(whitelist.add(_user), Errors.ADDRESS_ALREADY_EXIST);
    }

    function addToBlacklist(address _user) external onlyOwner {
        require(blacklist.add(_user), Errors.ADDRESS_ALREADY_EXIST);
    }
```

Recommendation: This use case must be clearly stated by the CapsuleNFT team. There are several possible actions, and they depend on the business logic and the intended behavior of the system.

Update: Response from the client: Fixed in commit (67caf73be8026d986f516786a8a7cb4e64475e8b).

QSP-16 Users in minting whitelist can send Ether

Severity: *Informational*

Status: Acknowledged

File(s) affected: `contracts/CapsuleMinter.sol`

Description: All payable functions using `checkStatus` modifier allow whitelisted users in `mintWhitelist` to send ether:

- `mintSimpleCapsule(...)`
- `mintSingleERC20Capsule(...)`
- `mintSingleERC721Capsule(...)`
- `mintMultiERC20Capsule(...)`
- `mintMultiERC721Capsule(...)`

Recommendation: Please, confirm this is the expected behavior. If not, whenever a whitelisted user in `mintWhitelist` sends ether the operation should revert.

Update: Response from the client: whitelisted users are allowed to send 0 as payable. They can still send fee if they want.

Automated Analyses

Slither

Slither has analyzed 34 contracts with 77 detectors and it has reported 228 issues. Most of the reported issues refer to contracts outside the scope of this audit. Such issues were discarded for being off the scope. When considering the scope of this audit, Slither has reported issues related to reentrancy in CapsuleMinter.sol and Capsule.sol, which our team consider as false-positive indications. Slither has also pointed out that the function CapsuleFactory.createCapsuleCollection(...) is ignoring the return value by `capsulesOf[_owner].add(_capsuleAddress)`. The tool has also warned about the need for emitting events in state changes (which are reported by us), and the need to perform zero address validation (already reported by us). Finally, there are issues related to non using the mixedCase solidity convention for names. All the relevant issues are listed in our report.

Adherence to Specification

The code adheres to the specification. The only observation we have is that the name of some functions may have been changed in the programming step and not updated in the `NFTMinter-Docs.md` document. These are likely minor naming changes that were made during the coding stage, but not reflected in the original specification.

Code Documentation

The code follows good programming practices and can be improved by increasing the amount of inline comments and adopting the NatSpec style.

Adherence to Best Practices

The code of the audited files presents a good programming style, consistent with the good programming practices of software engineering. The code is easy to read, which demonstrates the development team's care for code reusability and maintainability. Throughout this report, given the nature of our work as auditors, we have pointed out some issues to be reviewed and discussed by the development team. The three most relevant issues are: a) The possibility of changing the owner of a given Capsule by other Capsule owners just by calling the function `CapsuleFactory.updateCapsuleCollectionOwner(...)`; b) It is unclear how the Capsules will deal with deflationary tokens; c) The lack of access control in initializable functions. Some minor issues are present in the code. There are variables not following the mixedCase naming convention, such as proposed in the Solidity Style Guide. A non-exhaustive list of variables not following the naming conventions are:

1. In file `CapsuleFactoryStorage.sol`
    - `capsulesOf` => `_capsulesOf`
    - `whitelist` => `_whitelist`
    - `blacklist` => `_blacklist`
    - `CapsuleMinterStorage.sol`
    - `multiERC20Capsule` => `_multiERC20Capsule`
    - `multiERC721Capsule` => `_multiERC721Capsule`
    - `mintWhitelist` => `_mintWhitelist`

Finally, as already presented in the list of issues, some contracts are not inheriting their respective interfaces, which can be a problem since the functions of the interface are not implemented.

Test Results

Test Suite Results

```
npm test

> capsule-test-suite@0.1.0 test
> hardhat test

Generating typings for: 36 artifacts in dir: typechain-types for target: ethers-v5
Successfully generated 61 typings!
Compiled 35 Solidity files successfully


  Capsule NFT tests
     ✓ Should lock collection at 4 (150ms)
     ✓ Should revert if tries to lock at count less than counter (50ms)
     ✓ Should revert if tries to lock at 0 NFT count
     ✓ Should revert if tries to lock twice
    TokenURIOwner
       ✓ Should allow current owner to update tokenURIOwner
       ✓ Should revert if user is not tokenURIOwner

  Capsule Factory tests
     ✓ Should verify factory initialization
     ✓ Should update tax collector
     ✓ Should revert on invalid update
     ✓ Should update capsule creation tax
     ✓ Should revert on invalid update
     ✓ Should create capsule (40ms)
     ✓ Should fail capsule creation if incorrect tax sent
     ✓ Should verify whitelisted can create capsule without tax (38ms)
     ✓ Should verify Capsule configuration (38ms)
     ✓ Should update capsule owner at ownership transfer (80ms)
     ✓ Should not allow direct calls to updateCapsuleCollectionOwner (106ms)
     ✓ Should flush tax amount (40ms)
     ✓ Should allow only authorized to call flushTaxAmount
     ✓ Should verify that CapsuleMinter can be updated only once (73ms)

  Capsule Minter tests
     ✓ Should verify Capsule Minter is created properly
    Simple Capsule
      Mint simple Capsule
         ✓ Should fail when try to mint non capsule NFT
         ✓ Should be able to mint simple capsule (42ms)
         ✓ Should verify whitelisted can create simple capsule without tax
         ✓ Should show correct nft counter after 2 capsules are minted (53ms)
         ✓ Should set correct token uri (51ms)
      Mint simple capsule for private collection
         ✓ Should verify that collection is private
         ✓ Should allow owner to mint capsule
         ✓ Should fail when non owner try to mint capsule
      Burn simple Capsule
         ✓ Should revert when burning non capsule NFT
         ✓ Should revert when burning some others NFT
         ✓ Should burn simple capsule
    ERC20 Capsule
      Mint ERC20 Capsule
         ✓ Should mint ERC20 Capsule NFT (57ms)
      Burn ERC20 Capsule
         ✓ Should burn ERC20 Capsule NFT (43ms)
    Multi ERC20 Capsule
      Mint multi ERC20 Capsule
         ✓ Should mint multi ERC20 Capsule NFT (75ms)
      Burn multi ERC20 Capsule
         ✓ Should burn multi ERC20 Capsule NFT (56ms)
    ERC721 Capsule
      Mint ERC721 Capsule
```

```
        ✓ Should mint ERC721 Capsule NFT (45ms)
      Burn ERC721 Capsule
        ✓ Should burn ERC721 Capsule NFT
    Multi ERC721 Capsule
      Mint multi ERC721 Capsule
        ✓ Should mint multi ERC721 Capsule NFT (57ms)
      Burn multi ERC721 Capsule
        ✓ Should burn multi ERC721 Capsule NFT (48ms)

  Capsule Timelock tests
    Mint and burn simple timelock Capsule
      ✓ Should mint simple timelock nft
      ✓ Should revert if tries to burn before lock period
      ✓ Should burn NFT after unlock (64ms)
    Mint and burn ERC20 timelock Capsule
      ✓ Should mint ERC20 timelock Capsule (76ms)
      ✓ Should burn ERC20 timelock Capsule (126ms)

  Capsule ERC20 tests
    ✓ Should allow max 5M mint during 1st year
    ✓ Should allow to mint more than 5M after 1st year

47 passing (8s)
```

Code Coverage

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	86.06	56.82	77.59	86.82	
Capsule.sol	82.35	85	81.25	83.33	… ,99,103,104
CapsuleFactory.sol	75.93	52.27	64.71	75.93	… 106,111,121
CapsuleFactoryStorage.sol	100	100	100	100	
CapsuleMinter.sol	91.67	51.47	84	92.31	… 129,405,431
CapsuleMinterStorage.sol	100	100	100	100	
Errors.sol	100	100	100	100	
contracts/access/	53.85	33.33	60	57.14	
Governable.sol	**53.85**	**33.33**	**60**	**57.14**	**… 64,65,66,67**
All files	84.16	55.8	76.19	85.04	

```
npm run coverage

> capsule-test-suite@0.1.0 coverage
> hardhat coverage


Version
=======
> solidity-coverage: v0.7.20

Instrumenting for coverage...
============================

> access/  Governable.sol
>   Capsule.sol
>   CapsuleFactory.sol
>   CapsuleFactoryStorage.sol
>   CapsuleMinter.sol
>   CapsuleMinterStorage.sol
>   Errors.sol

Coverage skipped for:
=====================

>   CapsuleTimelock.sol
>   CapToken.sol
> interfaces/  ICapsule.sol
> interfaces/  ICapsuleFactory.sol
> interfaces/  ICapsuleMinter.sol
> interfaces/  IGovernable.sol
> openzeppelin/contracts/access/  Ownable.sol
> openzeppelin/contracts/proxy/utils/  Initializable.sol
> openzeppelin/contracts/security/  ReentrancyGuard.sol
> openzeppelin/contracts/token/ERC20/  ERC20.sol
> openzeppelin/contracts/token/ERC20/extensions/  IERC20Metadata.sol
> openzeppelin/contracts/token/ERC20/  IERC20.sol
> openzeppelin/contracts/token/ERC20/utils/  SafeERC20.sol
> openzeppelin/contracts/token/ERC721/  ERC721.sol
> openzeppelin/contracts/token/ERC721/extensions/  ERC721Enumerable.sol
> openzeppelin/contracts/token/ERC721/extensions/  ERC721URIStorage.sol
> openzeppelin/contracts/token/ERC721/extensions/  IERC721Enumerable.sol
> openzeppelin/contracts/token/ERC721/extensions/  IERC721Metadata.sol
> openzeppelin/contracts/token/ERC721/  IERC721.sol
> openzeppelin/contracts/token/ERC721/  IERC721Receiver.sol
> openzeppelin/contracts/utils/  Address.sol
> openzeppelin/contracts/utils/  Context.sol
> openzeppelin/contracts/utils/introspection/  ERC165.sol
> openzeppelin/contracts/utils/introspection/  IERC165.sol
> openzeppelin/contracts/utils/  Strings.sol
> openzeppelin/contracts/utils/structs/  EnumerableSet.sol
> test/  ERC20Mock.sol
> test/  ERC721Mock.sol

Compilation:
============
```

```
Warning: Contract code size exceeds 24576 bytes (a limit introduced in Spurious Dragon). This contract may not be deployable on mainnet. Consider enabling the optimizer (with a low "runs" value!), turning off revert
strings, or using libraries.
  --> contracts/  CapsuleFactory.sol:14:1:
   |
14 | contract CapsuleFactory is Initializable, Governable, CapsuleFactoryStorage {
   | ^ (Relevant source part starts here and spans across multiple lines).


Warning: Contract code size exceeds 24576 bytes (a limit introduced in Spurious Dragon). This contract may not be deployable on mainnet. Consider enabling the optimizer (with a low "runs" value!), turning off revert
strings, or using libraries.
  --> contracts/  CapsuleMinter.sol:19:1:
   |
19 | contract CapsuleMinter is Initializable, Governable, ReentrancyGuard, IERC721Receiver, CapsuleMinterStorage {
   | ^ (Relevant source part starts here and spans across multiple lines).


Generating typings for: 36 artifacts in dir: typechain-types for target: ethers-v5
Successfully generated 61 typings!
Compiled 35 Solidity files successfully

Network Info
============
> HardhatEVM: v2.9.0
> network:    hardhat

No need to generate any newer typings.


  Capsule NFT tests
     ✓ Should lock collection at 4 (168ms)
     ✓ Should revert if tries to lock at count less than counter (55ms)
     ✓ Should revert if tries to lock at 0 NFT count
     ✓ Should revert if tries to lock twice
    TokenURIOwner
       ✓ Should allow current owner to update tokenURIOwner
       ✓ Should revert if user is not tokenURIOwner

  Capsule Factory tests
     ✓ Should verify factory initialization
     ✓ Should update tax collector
     ✓ Should revert on invalid update (40ms)
     ✓ Should update capsule creation tax
     ✓ Should revert on invalid update
     ✓ Should create capsule (47ms)
     ✓ Should fail capsule creation if incorrect tax sent
     ✓ Should verify whitelisted can create capsule without tax (47ms)
     ✓ Should verify Capsule configuration (49ms)
     ✓ Should update capsule owner at ownership transfer (90ms)
     ✓ Should not allow direct calls to updateCapsuleCollectionOwner (113ms)
     ✓ Should flush tax amount (47ms)
     ✓ Should allow only authorized to call flushTaxAmount
     ✓ Should verify that CapsuleMinter can be updated only once (112ms)

  Capsule Minter tests
     ✓ Should verify Capsule Minter is created properly
    Simple Capsule
      Mint simple Capsule
         ✓ Should fail when try to mint non capsule NFT
         ✓ Should be able to mint simple capsule (53ms)
         ✓ Should verify whitelisted can create simple capsule without tax (41ms)
         ✓ Should show correct nft counter after 2 capsules are minted (65ms)
         ✓ Should set correct token uri (63ms)
      Mint simple capsule for private collection
         ✓ Should verify that collection is private
         ✓ Should allow owner to mint capsule
         ✓ Should fail when non owner try to mint capsule
      Burn simple Capsule
         ✓ Should revert when burning non capsule NFT
         ✓ Should revert when burning some others NFT
         ✓ Should burn simple capsule (40ms)
    ERC20 Capsule
      Mint ERC20 Capsule
         ✓ Should mint ERC20 Capsule NFT (72ms)
      Burn ERC20 Capsule
         ✓ Should burn ERC20 Capsule NFT (52ms)
    Multi ERC20 Capsule
      Mint multi ERC20 Capsule
         ✓ Should mint multi ERC20 Capsule NFT (92ms)
      Burn multi ERC20 Capsule
         ✓ Should burn multi ERC20 Capsule NFT (68ms)
    ERC721 Capsule
      Mint ERC721 Capsule
         ✓ Should mint ERC721 Capsule NFT (58ms)
      Burn ERC721 Capsule
         ✓ Should burn ERC721 Capsule NFT (47ms)
    Multi ERC721 Capsule
      Mint multi ERC721 Capsule
         ✓ Should mint multi ERC721 Capsule NFT (81ms)
      Burn multi ERC721 Capsule
         ✓ Should burn multi ERC721 Capsule NFT (65ms)

  Capsule Timelock tests
    Mint and burn simple timelock Capsule
       ✓ Should mint simple timelock nft (40ms)
       ✓ Should revert if tries to burn before lock period (40ms)
       ✓ Should burn NFT after unlock (81ms)
    Mint and burn ERC20 timelock Capsule
       ✓ Should mint ERC20 timelock Capsule (95ms)
       ✓ Should burn ERC20 timelock Capsule (490ms)

  Capsule ERC20 tests
     ✓ Should allow max 5M mint during 1st year
     ✓ Should allow to mint more than 5M after 1st year

  47 passing (10s)
```

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Contracts

```
a37095f287f63e0bbff0da27057e124c3afe66dbe187c23d8ab955791eac3f7d ./contracts/CapsuleFactory.sol

1757ece44e0903bb607df5190cf5a0c1a83b89bf6682eb5e7298e605f32eb20c ./contracts/Capsule.sol

5fceccd0fabf8edeb9229c038599160e74af8987370bced19256a4eee87cadbb ./contracts/CapsuleMinter.sol

737feab4a7cd4a5189c96ef154046a71cf34c6eb9a5d58eb23c6b3afb426b03c ./contracts/CapsuleMinterStorage.sol

28648713cc854ee7745daf42d836146402f5a680c25bbc7720cf911fc8ad6ea6 ./contracts/CapsuleTimelock.sol

49e8a0bf2cbeb455beb6f9ae6c25f65b3750b550d7f94673b6b1274c208f9be4 ./contracts/CapToken.sol

096ac2409ecebf16f932fa561c26e5a4ee33aec1936159131ead102daec5cebc ./contracts/CapsuleFactoryStorage.sol

bd6864d0e7f8b7385e091b439487e73f94bb3d5566f18761bba5eadc0434a7eb ./contracts/Errors.sol

760c89ce607c0804800794f06846c21141aa7e1ca8b9571b87383c151c1a06c4 ./contracts/test/ERC721Mock.sol

b82d780551503306c1ac7227e573f2e7c0a9ec564cb5697bebd2efbcd5e748a1 ./contracts/test/ERC20Mock.sol

87100213715ea79b68ed21b6c7b3fd869b2e2c1962ec37bc0b242d3c9881ac6a ./contracts/access/Governable.sol

88ecede3e5fa0b806b398e8a48ae86e557e64573606741cc2e9bee95aeb07eec ./contracts/openzeppelin/contracts/access/Ownable.sol

1b3ed391a1f3ad62dd8df45868450dba05938221f0c1e74e339e7e0015fa59cc ./contracts/openzeppelin/contracts/proxy/utils/Initializable.sol

d010dea25c1806a301684d8eeaa534102b949b3f041bf208ba7ef369f5d0cba2 ./contracts/openzeppelin/contracts/token/ERC721/IERC721.sol

a37535816653dbaea48dc8da487f35190ba81aea8b76a8d38312fb6a042bfa4c ./contracts/openzeppelin/contracts/token/ERC721/ERC721.sol

d6df589726d997eb52ba219df5c26f42d2ab78077bbb2def0c34ba7d50ba20a4 ./contracts/openzeppelin/contracts/token/ERC721/IERC721Receiver.sol

719b26a1eb236df099e7df9e78e7975dfdae4e812fe11926f5929e15bc68a081
./contracts/openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol

f9021240600e8867a24b4867e45bf81663ae140a1ec824adafaa58ae66dc972c
./contracts/openzeppelin/contracts/token/ERC721/extensions/IERC721Enumerable.sol

67df9d0ef32606c9efc6abbcfff96f567dfa4128702bbdfd707f7952eb4d03a8
./contracts/openzeppelin/contracts/token/ERC721/extensions/IERC721Metadata.sol

3b496c321ca6a44ab5dc8a70434ddfc184741e1ab3bbec8a683d36adeda6cd07
./contracts/openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol

7d95685c3cf5373b96559c40799c15a13a3605025916669197e5e932b783c29a ./contracts/openzeppelin/contracts/token/ERC20/ERC20.sol

5f4e89bc7ee8aeb26b724218151ebe2b5787f2c73b084d3e2b54ef5716223b18 ./contracts/openzeppelin/contracts/token/ERC20/IERC20.sol

943407eb0f4401c37da6d3efa32a4e5d6846a283942663fbf8e4bfa929eafc6d ./contracts/openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol

e5bee49b39ee4ed0914490d23e3e85ac732cad2c523312978ff673fcb1d18c61 ./contracts/openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol

86dc7333b9eebb700776751d67c190000da38b6d1156c49a1609713975acc6e9 ./contracts/openzeppelin/contracts/security/ReentrancyGuard.sol

543c46d0f81fd4e5d9d6a92beef3d2be18badb483b0b4718c819fe3dbbc37587 ./contracts/openzeppelin/contracts/utils/Context.sol

f0e8fc7f9475f793c6988374d6f237a8639f888158234307e51bc5a21eab80e0 ./contracts/openzeppelin/contracts/utils/Strings.sol

b4663f1c1800c0b635ecf42d33018afa9785f47c02247ab0f27f9a67d57fcea4 ./contracts/openzeppelin/contracts/utils/Address.sol

17b9dd0046758767e35f41abe264bdb1893377cb666fb0ed176d3cd15acc7c38 ./contracts/openzeppelin/contracts/utils/structs/EnumerableSet.sol

0c0ff26fb8503da6cdea91851edfd0796058823121fb4b12f0ed093ed39eb326 ./contracts/openzeppelin/contracts/utils/introspection/ERC165.sol

072805b211a653c333b232a3199b9e65fa7b82fc7a40ee5a3bc8a2dadd1cba01 ./contracts/openzeppelin/contracts/utils/introspection/IERC165.sol

b2673978b7744a20c43735904ce7479bd5928140105a065a92a16687b9635b27 ./contracts/interfaces/ICapsuleMinter.sol

f70a3c3a4b1eb83db81f98e3294140d5cf4cfbc7575574296ecc225d4783148c ./contracts/interfaces/ICapsule.sol

c439d49a44fd2a90bec5539db862ffc7d61b681f993b27b9084f4c72bca15fcd ./contracts/interfaces/ICapsuleFactory.sol

8dfb1fe8dd4e15b744112d7eddf64cd869c70787e37d86eb8b9ae83815c99e4b ./contracts/interfaces/IGovernable.sol
```

Tests

```
760c89ce607c0804800794f06846c21141aa7e1ca8b9571b87383c151c1a06c4 ./test/ERC721Mock.sol

b82d780551503306c1ac7227e573f2e7c0a9ec564cb5697bebd2efbcd5e748a1 ./test/ERC20Mock.sol
```

Changelog

- 2022-04-07 - Initial report
- 2022-04-18 - Final report

About Quantstamp

Quantstamp is a Y Combinator-backed company that helps to secure blockchain platforms at scale using computer-aided reasoning tools, with a mission to help boost the adoption of this exponentially growing technology.

With over 1000 Google scholar citations and numerous published papers, Quantstamp's team has decades of combined experience in formal verification, static analysis, and software verification. Quantstamp has also developed a protocol to help smart contract developers and projects worldwide to perform cost-effective smart contract security scans.

To date, Quantstamp has protected $5B in digital asset risk from hackers and assisted dozens of blockchain projects globally through its white glove security assessment services. As an evangelist of the blockchain ecosystem, Quantstamp assists core infrastructure projects and leading community initiatives such as the Ethereum Community Fund to expedite the adoption of blockchain technology.

Quantstamp's collaborations with leading academic institutions such as the National University of Singapore and MIT (Massachusetts Institute of Technology) reflect our commitment to research, development, and enabling world-class blockchain security.