Smart Contract Audit Report for Certhis

Testers

1. Or Duan
2. Avigdor Sason Cohen

Table of Contents

Management Summary

Certhis contacted Sayfer to perform a security audit on their smart contracts.

This report documents the research carried out by Sayfer targeting the selected resources defined under the research scope. Particularly, this report displays the security posture review for the smart contracts included in the scope.

Over the research period of 3 research weeks, we discovered 8 vulnerabilities in the contract. 2 of them are marked as critical as deploying the contract as is, which can cause an unusable contract and direct loss of funds.

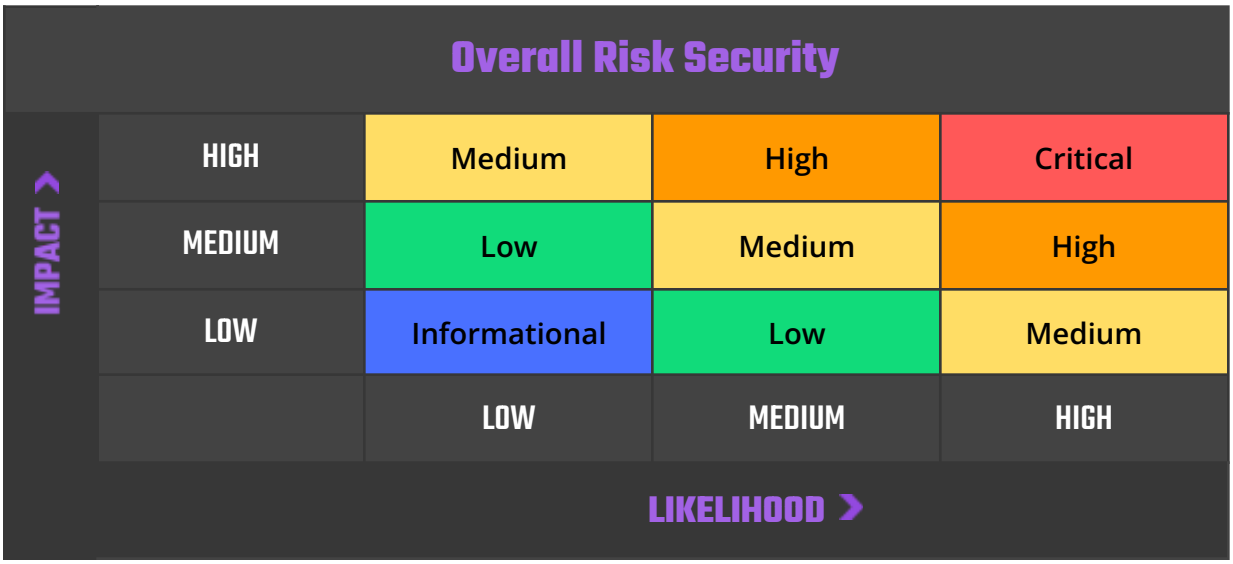
Risk Methodology

At Sayfer, we are committed to delivering the highest quality smart contract audits to our clients. That's why we have implemented a comprehensive risk assessment model to evaluate the severity of our findings and provide our clients with the best possible recommendations for mitigation.

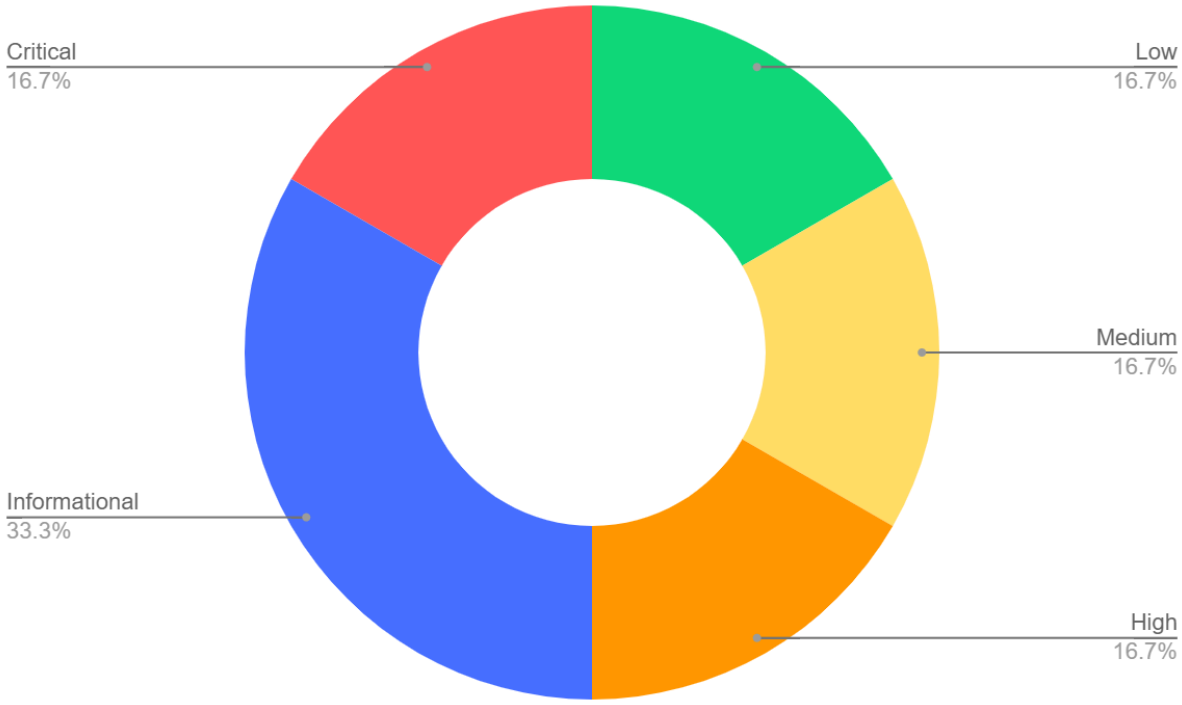
Our risk assessment model is based on two key factors: **IMPACT** and **LIKELIHOOD**. Impact refers to the potential harm that could result from an issue, such as financial loss, reputational damage, or a non-operational system. Likelihood refers to the probability that an issue will occur, taking into account factors such as the complexity of the contract and the number of potential attackers.

By combining these two factors, we can create a comprehensive understanding of the risk posed by a particular issue and provide our clients with a clear and actionable assessment of the severity of the issue. This approach allows us to prioritize our recommendations and ensure that our clients receive the best possible advice on how to protect their smart contracts.

Risk is defined as follows:



Vulnerabilities by Risk



Risk	Low	Medium	High	Critical	Informational
# of issues	4	2	0	2	4

Approach

Introduction

Certhis contacted Sayfer to perform a security audit on their smart contracts.

This report documents the research carried out by Sayfer targeting the selected resources defined under the research scope. Particularly, this report displays the security posture review for the aforementioned contracts.

Scope Overview

Together with the client team we defined the following contract as the scope of the project.

Contract	Sha-256
certhis_collection.sol	f6c7a3d562bf379debd6a312deaaa0bd90388154af3ca8580362ffc9281202c7
certhis_main.sol	47938cdaad83a46b4741e91bc7a5973dee64667069a25ea08ca23f37e76f08b8
certhis_struct.sol	dbb6e1dde08a2ce826b94da2698f86ad5e1bd868972474dafc4e49c59678d93e

Our tests were performed between 03/7 to 23/7 2023

Scope Validation

We began by ensuring that the scope defined to us by the client was technically logical. Deciding what scope is right for a given system is part of the initial discussion.

Threat Model

We defined that the largest current threat to the system is the ability of malicious users to steal funds from the contract.

Security Evaluation

The following test cases were the guideline while auditing the system. This checklist is a modified version of the SCSVS v1.2, with improved grammar, clarity, conciseness, and additional criteria. Where there is a gap in the numbering, an original criterion was removed. Criteria that are marked with an asterisk were added by us.

Architecture, Design and Threat Modeling	Test Name
G1.2	Every introduced design change is preceded by threat modeling.
G1.3	The documentation clearly and precisely defines all trust boundaries in the contract (trusted relations with other contracts and significant data flows).
G1.4	The SCSVS, security requirements or policy is available to all developers and testers.
G1.5	The events for the (state changing/crucial for business) operations are defined.
G1.6	The project includes a mechanism that can temporarily stop sensitive functionalities in case of an attack. This mechanism should not block users' access to their assets (e.g. tokens).
G1.7	The amount of unused cryptocurrencies kept on the contract is controlled and at the minimum acceptable level so as not to become a potential target of an attack.
G1.8	If the fallback function can be called by anyone, it is included in the threat model.
G1.9	Business logic is consistent. Important changes in the logic should be applied in all contracts.
G1.10	Automatic code analysis tools are employed to detect vulnerabilities.
G1.11	The latest major release of Solidity is used.
G1.12	When using an external implementation of a contract, the most recent version is used.
G1.13	When functions are overridden to extend functionality, the super keyword is used to maintain previous functionality.
G1.14	The order of inheritance is carefully specified.
G1.15	There is a component that monitors contract activity using events.
G1.16	The threat model includes whale transactions.
G1.17	The leakage of one private key does not compromise the security of the entire project.

Policies and Procedures	Test Name

G2.2	The system's security is under constant monitoring (e.g. the expected level of funds).
G2.3	There is a policy to track new security vulnerabilities and to update libraries to the latest secure version.
G2.4	The security department can be publicly contacted and that the procedure for handling reported bugs (e.g., thorough bug bounty) is well-defined.
G2.5	The process of adding new components to the system is well defined.
G2.6	The process of major system changes involves threat modeling by an external company.
G2.7	The process of adding and updating components to the system includes a security audit by an external company.
G2.8	In the event of a hack, there's a clear and well known mitigation procedure in place.
G2.9	The procedure in the event of a hack clearly defines which persons are to execute the required actions.
G2.10	The procedure includes alarming other projects about the hack through trusted channels.
G2.11	A private key leak mitigation procedure is defined.

Upgradability	**Test Name**
G2.2	Before upgrading, an emulation is made in a fork of the main network and everything works as expected on the local copy.
G2.3	The upgrade process is executed by a multisig contract where more than one person must approve the operation.
G2.4	Timelocks are used for important operations so that the users have time to observe upcoming changes (please note that removing potential vulnerabilities in this case may be more difficult).
G2.5	*initialize()* can only be called once.
G2.6	*initialize()* can only be called by an authorized role through appropriate modifiers (e.g. *initializer*, *onlyOwner*).
G2.7	The update process is done in a single transaction so that no one can front-run it.
G2.8	Upgradeable contracts have reserved gap on slots to prevent overwriting.
G2.9	The number of reserved (as a gap) slots has been reduced appropriately if new variables have been added.
G2.10	There are no changes in the order in which the contract state variables are declared, nor their types.
G2.11	New values returned by the functions are the same as in previous versions of the contract (e.g. *owner()*, *balanceOf(address)*).
G2.12	The implementation is initialized.
G2.13	The implementation can't be destroyed.

Business Logic	Test Name
G4.2	The contract logic and protocol parameters implementation corresponds to the documentation.
G4.3	The business logic proceeds in a sequential step order and it is not possible to skip steps or to do it in a different order than designed.
G4.4	The contract has correctly enforced business limits.
G4.5	The business logic does not rely on the values retrieved from untrusted contracts (especially when there are multiple calls to the same contract in a single flow).
G4.6	The business logic does not rely on the contract's balance (e.g., *balance == 0*).
G4.7	Sensitive operations do not depend on block data (e.g., *block hash*, *timestamp*).
G4.8	The contract uses mechanisms that mitigate transaction-ordering (front-running) attacks (e.g. pre-commit schemes).
G4.9	The contract does not send funds automatically, but lets users withdraw funds in separate transactions instead.

Access Control	Test Name
G5.2	The principle of the least privilege is upheld. Other contracts should only be able to access functions and data for which they possess specific authorization.
G5.3	New contracts with access to the audited contract adhere to the principle of minimum rights by default. Contracts should have a minimal or no permissions until access to the new features is explicitly granted.
G5.4	The creator of the contract complies with the principle of the least privilege and their rights strictly follow those outlined in the documentation.
G5.5	The contract enforces the access control rules specified in a trusted contract, especially if the dApp client-side access control is present and could be bypassed.
G5.6	Calls to external contracts are only allowed if necessary.
G5.7	Modifier code is clear and simple. The logic should not contain external calls to untrusted contracts.
G5.8	All user and data attributes used by access controls are kept in trusted contracts and cannot be manipulated by other contracts unless specifically authorized.
G5.9	the access controls fail securely, including when a revert occurs.
G5.10	If the input (function parameters) is validated, the positive validation approach (whitelisting) is used where possible.

Communication	Test Name
G6.2	Libraries that are not part of the application (but the smart contract relies on to operate) are identified.

G6.3	Delegate call is not used with untrusted contracts.
G6.4	Third party contracts do not shadow special functions (e.g. revert).
G6.5	The contract does not check whether the address is a contract using *extcodesize* opcode.
G6.6	Re-entrancy attacks are mitigated by blocking recursive calls from other contracts and following the Check-Effects-Interactions pattern. Do not use the *send* function unless it is a must.
G6.7	The result of low-level function calls (e.g. *send*, *delegatecall*, *call*) from other contracts is checked.
G6.8	Contract relies on the data provided by the right sender and does not rely on tx.origin value.

Arithmetic	**Test Name**
G7.2	The values and math operations are resistant to integer overflows. Use SafeMath library for arithmetic operations before solidity 0.8.*.
G7.3	the unchecked code snippets from Solidity ≥ 0.8.* do not introduce integer under/overflows.
G7.4	Extreme values (e.g. maximum and minimum values of the variable type) are considered and do not change the logic flow of the contract.
G7.5	Non-strict inequality is used for balance equality.
G7.6	Correct orders of magnitude are used in the calculations.
G7.7	In calculations, multiplication is performed before division for accuracy.
G7.8	The contract does not assume fixed-point precision and uses a multiplier or store both the numerator and denominator.

Denial of Service	**Test Name**
G8.2	The contract does not iterate over unbound loops.
G8.3	Self-destruct functionality is used only if necessary. If it is included in the contract, it should be clearly described in the documentation.
G8.4	The business logic isn't blocked if an actor (e.g. contract, account, oracle) is absent.
G8.5	The business logic does not disincentivize users to use contracts (e.g. the cost of transaction is higher than the profit).
G8.6	Expressions of functions assert or require have a passing variant.
G8.7	If the fallback function is not callable by anyone, it is not blocking contract functionalities.
G8.8	There are no costly operations in a loop.
G8.9	There are no calls to untrusted contracts in a loop.
G8.10	If there is a possibility of suspending the operation of the contract, it is also

	possible to resume it.
G8.11	If whitelists and blacklists are used, they do not interfere with normal operation of the system.
G8.12	There is no DoS caused by overflows and underflows.

Blockchain Data	Test Name
G9.2	Any saved data in contracts is not considered secure or private (even private variables).
G9.3	No confidential data is stored in the blockchain (passwords, personal data, token etc.).
G9.4	Contracts do not use string literals as keys for mappings. Global constants are used instead to prevent Homoglyph attack.
G9.5	Contract does not trivially generate pseudorandom numbers based on the information from blockchain (e.g. seeding with the block number).

Gas Usage and Limitations	Test Name
G10.2	Gas usage is anticipated, defined and has clear limitations that cannot be exceeded. Both code structure and malicious input should not cause gas exhaustion.
G10.3	Function execution and functionality does not depend on hard-coded gas fees (they are bound to vary).

Clarity and Readability	Test Name
G11.2	The logic is clear and modularized in multiple simple contracts and functions.
G11.3	Each contract has a short 1-2 sentence comment that explains its purpose and functionality.
G11.4	Off-the-shelf implementations are used, this is made clear in comment. If these implementations have been modified, the modifications are noted throughout the contract.
G11.5	The inheritance order is taken into account in contracts that use multiple inheritance and shadow functions.
G11.6	Where possible, contracts use existing tested code (e.g. token contracts or mechanisms like *ownable*) instead of implementing their own.
G11.7	Consistent naming patterns are followed throughout the project.
G11.8	Variables have distinctive names.
G11.9	All storage variables are initialized.
G11.10	Functions with specified return type return a value of that type.

G11.11	All functions and variables are used.
G11.12	*require* is used instead of *revert* in *if* statements.
G11.13	The *assert* function is used to test for internal errors and the *require* function is used to ensure a valid condition in input from users and external contracts.
G11.14	Assembly code is only used if necessary.

Test Coverage	Test Name
G12.2	Abuse narratives detailed in the threat model are covered by unit tests.
G12.3	Sensitive functions in verified contracts are covered with tests in the development phase.
G12.4	Implementation of verified contracts has been checked for security vulnerabilities using both static and dynamic analysis.
G12.5	Contract specification has been formally verified.
G12.6	The specification and results of the formal verification is included in the documentation.

Decentralized Finance	Test Name
G14.1	The lender's contract does not assume its balance (used to confirm loan repayment) to be changed only with its own functions.
G14.2	Functions that change lenders' balance and/or lend cryptocurrency are non-re-entrant if the smart contract allows borrowing the main platform's cryptocurrency (e.g. Ethereum). It blocks the attacks that update the borrower's balance during the flash loan execution.
G14.3	Flash loan functions can only call predefined functions on the receiving contract. If it is possible, define a trusted subset of contracts to be called. Usually, the sending (borrowing) contract is the one to be called back.
G14.4	If it includes potentially dangerous operations (e.g. sending back more ETH/tokens than borrowed), the receiver's function that handles borrowed ETH or tokens can be called only by the pool and within a process initiated by the receiving contract's owner or another trusted source (e.g. multisig).
G14.5	Calculations of liquidity pool share are performed with the highest possible precision (e.g. if the contribution is calculated for ETH it should be done with 18 digit precision - for Wei, not Ether). The dividend must be multiplied by the 10 to the power of the number of decimal digits (e.g. dividend * 10^18 / divisor).
G14.6	Rewards cannot be calculated and distributed within the same function call that deposits tokens (it should also be defined as non-re-entrant). This protects from momentary fluctuations in shares.
G14.7	Governance contracts are protected from flash loan attacks. One possible

	mitigation technique is to require the process of depositing governance tokens and proposing a change to be executed in different transactions included in different blocks.
G14.8	When using on-chain oracles, contracts are able to pause operations based on the oracles' result (in case of a compromised oracle).
G14.9	External contracts (even trusted ones) that are allowed to change the attributes of a project contract (e.g. token price) have the following limitations implemented: thresholds for the change (e.g. no more/less than 5%) and a limit of updates (e.g. one update per day).
G14.10	Contract attributes that can be updated by the external contracts (even trusted ones) are monitored (e.g. using events) and an incident response procedure is implemented (e.g. during an ongoing attack).
G14.11	Complex math operations that consist of both multiplication and division operations first perform multiplications and then division.
G14.12	When calculating exchange prices (e.g. ETH to token or vice versa), the numerator and denominator are multiplied by the reserves (see the *getInputPrice* function in the *UniswapExchange* contract).

Audit Findings

Require Verification Bypass

ID	SAY-01
Status	Fixed
Risk	Critical
Business Impact	This error could allow attackers to mint NFTs by choosing to pay with Ether, even if they don't actually send any Ether.
Location	- certhis_collection.sol; 341-352; safeMint(address, uint256, address)

Description	The following snippet requires a successful ERC20 token transfer (in case of ERC20 minting), or if the minting is not of type ERC20, that the *msg.value* is greater than or equal to *price_for_mint*.
	
	• safeMint(address, uint256, address):
	`bool success = true;`
	`// if erc 20 mint`
	`if (type_mint == 1) {`
	`success = IERC20(collection_object.currency_for_mint)`
	`.transferFrom(msg.sender, address(this), price_for_mint);`
	`}`
	
	`require(`
	`success ||`
	`(msg.value >= collection_object.price_for_mint &&`
	`type_mint != 1)`
	`);`
	
	This means that if the *type_mint* is not 1, then the require in the snippet will always pass, since *success* has been set to *true*.

Mitigation	Unless there is additional business logic that should be considered, this can be fixed with a simple if-else statement:
	`if (type_mint == 1) {`
	`// ERC20 minting`
	`require(success, "ERC20 transfer failed");`

```
} else {
   // Ether minting
   require(msg.value >= collection_object.price_for_mint, "Not
enough Ether sent for minting.");
}
```

Potential Bypass of Require Statement

ID	SAY-02
Status	Fixed
Risk	Critical
Business Impact	This could allow users to buy tokens even if the amount they paid is not greater than the *sellable_amount*.
Location	- certhis_collection.sol; 156-168; buy_token(address, uint256, address)

Description	During our assessment, we came across the following require:
	● buy_token(address, uint256, address):
	`bool success = true;`
	`if (_nfts[_nft_id].sellable_type != 2) {`
	`success = IERC20(_nfts[_nft_id].sellable_currency).transferFrom(`
	`msg.sender,`
	`address(this),`
	`_nfts[_nft_id].sellable_amount`
	`);`
	`}`
	
	`require(`
	`(success == true || // Check can be bypassed at this point`
	`(msg.value >= _nfts[_nft_id].sellable_amount &&`
	`_nfts[_nft_id].sellable_type == 2))`
	`);`
	
	Since success was set to true earlier, and the *sellable_type* is 2, the require statement will pass even if the *msg.value* is smaller than *NFT.sellable_amount*.

Mitigation	Depending on your application's requirements, it might be necessary to use multiple checks, to make sure all conditions must be satisfied, like this:
	`require(success, "ERC20 transfer failed");`
	`require(msg.value >= _nfts[_nft_id].sellable_amount &&`
	`type_mint == 2, "Insufficient funds for mint");`

Inefficient Storage Access

ID	SAY-03
Status	Fixed
Risk	Medium
Business Impact	Gas waste for users.
Location	- certhis_collection.sol; safeMint(address, uint256, address)
Description	In Solidity, storage is expensive and every storage operation costs gas. This can make the safeMint function expensive to call, leading to inefficiencies and unnecessary costs for the users. The variable collection_object ● safeMint(address, uint256, address): `if (collection_object.collection_full == 2) { require(!_exists(_nft.nft_id)); } if (collection_object.check_contract != address(0)) { require( Icerthis_check(collection_object.check_contract).check( _to ) == true ); }` The problem here is that the object is being repeatedly loaded into memory every time the code calls the *collection_object.*
Mitigation	More tests and use cases like that should be evaluated internally to see if there are more places where gas could be saved

Lack of Input Validation

ID	Say-04
Status	Acknowledged
Risk	Medium
Business Impact	Malicious actors may exploit this lack of checks to mint NFTs with inappropriate or unexpected values. Moreover, if these unchecked parameters are used in critical computations or logic, they can lead to erroneous outcomes and potential manipulation of your contract's intended behavior. In the case of the certhis_main constructor, invalid addresses (like a null address) may even mean that the sent funds are irretrievably lost.
Location	- certhis_collection.sol; safeMint(address, uint256, address)

Description	<u>safeMint:</u>
	The crucial arguments provided by the minter in the *safeMint* do not seem to pass any input validation.
	
	● The *safeMint* signature:
	`function safeMint(`
	`address _to,`
	`certhis_struct.NFT memory _nft,`
	`certhis_struct.mint_proof memory _mint_proof`
	`) public payable returns (uint256)`
	
	<u>certhis_main constructor:</u>
	The constructor of the certhis_main contract takes two address parameters (*_default_certhis_payout* and *_default_certhis_payout_mint*). These are presumably addresses of accounts that will receive certain payouts. However, the constructor does not validate these important addresses.

Mitigation	It's always worth validating your input before using it. For instance, that addresses provided (like *_to*) not null, etc. Some examples of checks you can add for safe mint:

```
require(_to != address(0), "Receiver address cannot be zero
address");
require(bytes(_nft.token_uri).length > 0, "Token URI cannot be
empty");
require(_nft.royalties > 0 && _nft.royalties <= MAX_ROYALTY,
"Royalties out of range");
// Add similar checks for `_mint_proof` as needed
```

Notes: in this example, *MAX_ROYALTY* is simply a hypothetical constant representing the maximum allowable value for royalties. It may not be relevant to your protocol. It's also important to add that these examples are simply to provide illustration, the checks that you should implement depend entirely on your business logic.

Missing Event Emissions

ID	SAY-05
Status	Acknowledged
Risk	Low
Business Impact	Event emissions can serve as an important way to track changes and actions in the contract. Events provide a way to trigger functionality and transfer information from the contract to security tools, dApps or analytics tools, enabling users or developers to react to specific contract changes.
Location	- certhis_main.sol - certhis_collection.sol
Description	The following is a list of seemingly important functions that lack any event emissions: ● certhis_main.sol; update_default(address) ● certhis_collection; update_collection_object(address) ● certhis_collection; freeze(bool, bool) ● certhis_collection; buy_token(address, uint256, address) ● certhis_collection.sol; safeMint(address, uint256, address)
Mitigation	We recommend that you add emit events in these functions, and when other important state changes are made. By doing so, you increase the transparency of your contract and make it easier to track and verify contract activity. Additionally, events make your contract more compatible with third-party services and front-end interfaces.

Structural Problems in *safeMint*

ID	SAY-06
Status	Acknowledged
Risk	Low
Business Impact	It's critical to make sure that functions that contain central business logic (such as minting) are absolutely solid in terms of design and programming. Doing so can prevent a lot of issues from popping up in the future.
Location	- certhis_collection.sol; safeMint(address, uint256, address)
Description	This highly important function appears to suffer from multiple design weaknesses relating to code structure, design patterns, and potentially expensive operations. Specific problems include: ● **High Complexity:** The safeMint function is doing many things, it is advised to separate these into smaller, more manageable functions, adhering to the Single Responsibility Principle. ● **Repeated and Costly Storage Operations:** There are multiple instances where data is being stored in state variables, which is a costly operation. For example, the _nfts[new_nft_id] mapping is being updated multiple times within the function. ● **Require Statements Scattered:** The function has require statements scattered throughout, rather than being grouped at the beginning. The issue here is that a lot of computation may happen before hitting a require statement that causes the transaction to revert, resulting in wasted gas.
Mitigation	1. **Refactor for Simplicity:** Decompose the function into smaller, more manageable functions. This would make the code more readable and easier to maintain. 2. **Optimize State Variables Usage:** Try to minimize the usage of state variables and reduce the frequency of storage operations. You can do this by using local variables to hold interim computation results. 3. **Use Events:** Minting NFTs is an important state change, but currently, no event is emitted. 4. **Adopt the Checks-Effects-Interactions Pattern:** This is a well-established pattern in Solidity to prevent re-entrancy attacks and

	to reduce wasted gas in failed transactions. You can read more about it here.

Redundant Condition Checks

ID	SAY-07
Status	Acknowledged
Risk	Low
Business Impact	Redundant condition checks may not pose a direct security risk, but they increase the computational complexity of the function, leading to inefficiencies and higher gas costs for users.
Location	- certhis_collection.sol; safeMint(address, uint256, address)
Description	There appear to be some redundant condition checks in the *safeMint* function, specifically conditional checks on *collection_object.max_mint_sell* and *collection_object.max_mint*. • safeMint(address, uint256, address), lines 277-289: `if ((collection_object.max_mint_sell != 0 && collection_object.max_mint_sell < index_nft) || (collection_object.mint_start != 0 && collection_object.mint_start > block.timestamp) || (collection_object.mint_end != 0 && collection_object.mint_end < block.timestamp) || collection_object.max_supply <= index_nft || (sell_id_p_1 > collection_object.max_mint && collection_object.max_mint != 0)) { return_error = true; }` The checks on collection_object.max_mint_sell != 0 and collection_object.max_mint != 0 are not necessary if the variables are always initialized to a value other than zero.
Mitigation	To optimize the contract, you should simplify and short-circuit the condition checks where possible. If this is not the case, then these checks can be moved outside the if statement and only be performed once.

Lack of Descriptive Revert Messages

ID	SAY-08
Status	Acknowledged
Risk	Low
Business Impact	Revert messages can help both developers and users understand why their transactions failed.
Location	- certhis_collection.sol

Description	We noticed that most require statements in the code do not have proper revert messages. Here are a few examples from certhis_collection.sol:
	• safeMint(address, uint256, address), line 256:
	`require(_nft.royalties <= 2500 && freeze_mint == false);`
	• safeMint(address, uint256, address), line 291:
	`require(return_error == false);`
	• update_collection_object(address); line 78:
	`require(msg.sender == certhis_contract);`
	• _beforeTokenTransfer(address, address, uint256, uint256); 442:
	`require(from == address(0) || sbt == false);`

Mitigation	Let's take the first example. We can do it like this: `require(_nft.royalties <= 2500 && freeze_mint == false, "Royalties above 2500.");` The user will then know exactly why his minting failed.

Incorrect Function Description

ID	SAY-09
Status	Acknowledged
Risk	Informational
Business Impact	Future developers might be confused about what the function is doing
Location	certhis_main.sol; manage_collection(uint16, uint256, address, string, string)
Description	The description for the *_edit* variable in the function *manage_collection* appears to be wrong: • certhis_main.sol; line 52: `bool _edit  : true for edit false for create and deploy` It describes the variable as a *bool*, but it is, in fact, a *unint16* with three possible values (0, 1, 2), as can be seen in the function itself.
Mitigation	Amend the comment documentation to accurately represent the _edit parameter as a uint16 value and clarify its different states.

Inconsistent Naming Conventions

ID	
Status	Acknowledged
Risk	Informational
Business Impact	Good naming conventions improve code readability and maintainability, which can help prevent bugs and misunderstandings.
Location	—
Description	Variables: We noticed that some variables have names that do not clearly convey information about their intended purpose. Generic variable names like *_id*, *_edit*, *_collections*, *_lables* are not helpful to the reader. We are also not sure what is the intended meaning of underscores before variable names. Some Solidity devs use it to indicate function arguments, but arguments of certain functions like *calculate_payouts* in certhis_collection do not follow that convention.
Mitigation	We recommend that you review your code and ensure that a consistent and clear naming convention is used.

Missing Indexed Parameters and Underutilized Events

ID	SAY-10
Status	Acknowledged
Risk	Informational
Business Impact	The issues we listed here can result in reduced usability and effectiveness of the events, making it difficult to track and analyze specific events or understand their context.
Location	- certhis_main.sol

Description	We found two small issues regarding events:
	● The event *return_label_id* and *return_collection_id*, both in certhis_collection, do not have index parameters, limiting the efficiency of searching and filtering event logs.
	● These two events emit only the ID of the label or collection, without providing additional information about the states or changes that occurred during the function's execution.
	● certhis_main.sol; lines 20-21:
	`event return_label_id(uint256 _label_id);`
	`event return_collection_id(uint256 _collection_id);`

Mitigation	To address these vulnerabilities, the following actions are recommended:
	1. Add indexed parameters to the events to improve the efficiency of searching and filtering event logs.
	`event return_label_id(uint256 indexed _label_id);`
	`event return_collection_id(uint256 indexed _collection_id);`
	2. Emit additional relevant information in the events to provide a better understanding of the states or changes that occurred during the function's execution. This can include attributes such as the name of the collection, the address of the collection contract, or any other pertinent details that would be useful for external entities reading these events.
	`event return_label_id(uint256 indexed _label_id, string name, string creator);`
	`event return_collection_id(uint256 indexed _collection_id, string name, string creator);`

Order of Layout Violation

ID	SAY-11
Status	Fixed
Risk	Informational
Business Impact	Following the standard order of layout can increase the readability and maintainability of the contract.
Location	- certhis_collection.sol
Description	certhis_collection.sol strays a bit from the standard order of layout: *MAX_SUPPLY()*, a public function, comes before the constructor.
Mitigation	The standard order of layout is as follows: variables, events, modifiers, constructor/initializer, fallback function, external functions, public functions, internal functions, and finally, private functions. Following this order, *MAX_SUPPLY()* should come after the constructor.