Smart Contract Audit Report for Funtico

Testers

1. Or Duan
2. Avigdor Sason Cohen

Table of Contents

Management Summary

Funtico 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 Funtico's smart contracts.

Over the research period of 40 research hours, we discovered 16 vulnerabilities in the contract. One of them is marked as critical, as deploying the contract as-is may lead to direct loss of funds and functionality.

After a review by the Sayfer team, we certify that all the security issues mentioned in this report have been addressed by the Funtico team.

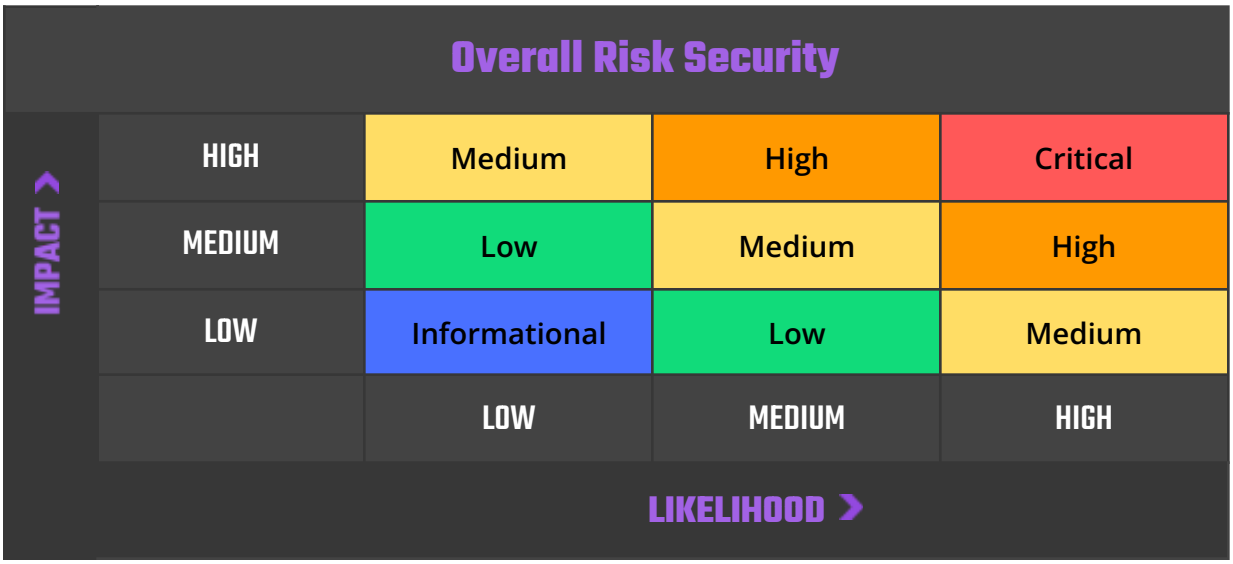
Risk Methodology

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.

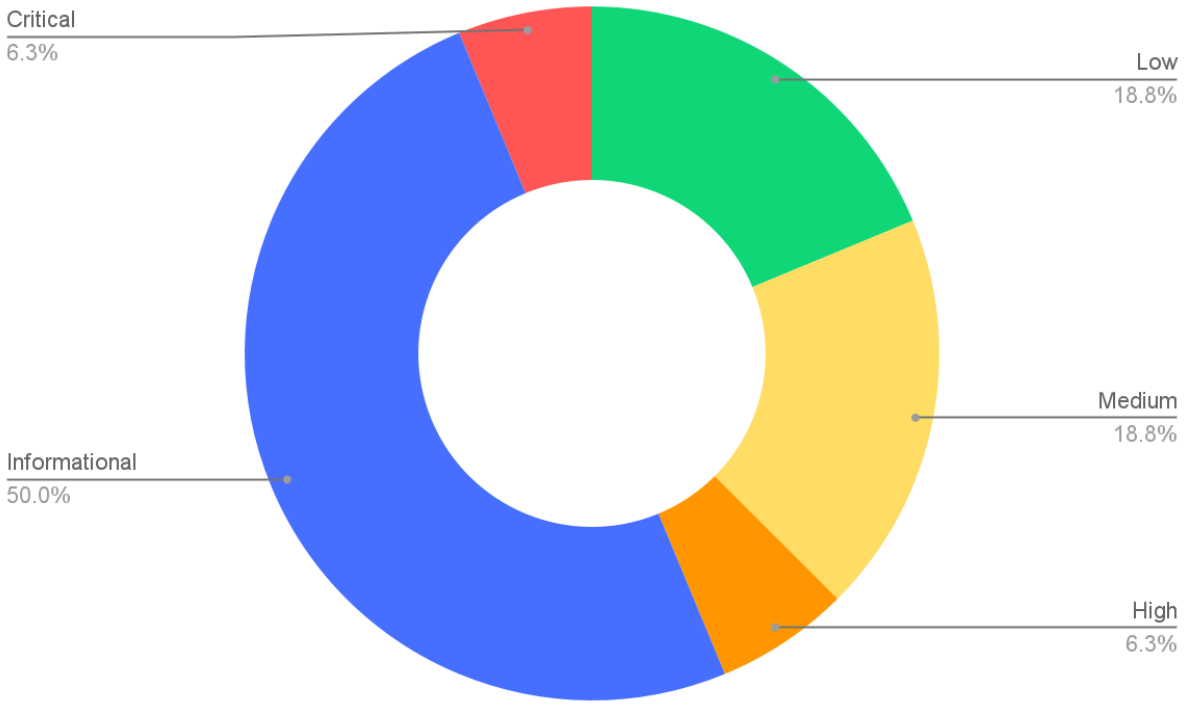
Our risk assessment model is based on two key factors: **IMPACT** and **LIKELIHOOD**. Impact refers to the potential harm that could result from an issue, such as financial loss, reputational damage, or a non-operational system. Likelihood refers to the probability that an issue will occur, taking into 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	3	3	1	1	8

Approach

Introduction

Funtico 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
`FanticoNFT.sol`	`719d147bdf72bccb27b0ee181b303f2ab1088ffda2425f4552a13ff58764d5bd`
`PaymentGatewayContract.sol`	`2419120efd59d326929768839ab45a5a0b045b4a39d7b5768fd5d833956357bb`
`Timelock.sol`	`ff6a811e4dbcd3e3efd4069edc74061516b6c839981a0906cf1c996f3c5bb0d4`

Our tests were performed in March 2024.

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

[C] Critical Issues in *withdraw(uint256)*

ID	SAY-01
Status	Fixed
Risk	Critical
Business Impact	The crucial withdraw function will not work correctly, entirely breaking the contract.
Location	- PaymentGatewayContract; withdraw(uint256)

Description	`withdraw(uint256)` starts by converting the 6-decimal USDC `_amount` to 18 decimals.
	`_amount = _amount * (10 ** 18);`
	This will certainly render the function unusable, thus raising this finding to Critical. The reasoning behind this operation is also erroneous, as we will later see.
	
	Next, the function miscalculates its gas price.
	`uint gasCost = gasleft() * tx.gasprice;`
	`gasleft()` simply returns the amount of gas remaining for the function before reversion. It's therefore higher than the amount of gas used. To calculate the amount of gas used in a chunk of code, you do something like
	`uint gasleftStart = gasleft();`
	`~~~ code ~~~`
	`~~~ code ~~~`
	`~~~ code ~~~`
	`uint gasCost = (gasleftStart - gasleft()) * tx.gasprice;`
	This can only be done after the execution. Accurately estimating gas costs ahead of time, which seems to be the developers' intention, is much more difficult.
	
	Finally, `withdraw(uint256)` mixes up USDC (`_amount`) and ether (`gasCost`, which is incorrect in any case) and then requires the user to have more USDC than the sum.
	`uint amountToWithdraw = _amount + gasCost;`

```
require(amountToWithdraw > 0, "Amount must be greater than zero");

// Ensure contract has sufficient balance
require(usdcToken.balanceOf(address(this)) ⩾ amountToWithdraw,
"Insufficient USDC balance");
```

This will not work in practice, as you don't pay gas in USDC.

Note: We have been told by the client that the assumption of 18 decimals is due to the test-token that they have been employing instead of USDC, which does have 18 decimals. However, in its current state, the function will not work correctly.

Mitigation	Rethink the function and restructure it accordingly.

[H] Native Coins Sent to *Pay(uint256)* Will Be Lost

ID	SAY-02
Status	Fixed
Risk	High
Business Impact	Any ether sent to `Pay(unit256)` will be irrecoverably locked away in the contract.
Location	- `PaymentGatewayContract:28; Pay(uint256)`
Description	The payable keyword is only needed when dealing with native tokens (such as ether). `Pay(uint256)` has it, but it doesn't handle `msg.value`. `Pay(uint256)` has the `payable` keyword, So the payment gateway will gladly receive ether, but it has no provision to withdraw it, meaning it will be permanently locked away.
Mitigation	Remove the payable keyword from the function declaration.

[M] Pause Is Implemented but Cannot Be Used

ID	SAY-03
Status	Fixed (The pausing feature has been removed)
Risk	Medium
Business Impact	Calling `pause()` will not pause the execution of any function.
Location	- `FanticoNFT.sol; pause(), unpause()`
Description	With `ERC721Pausable`, which the contract inherits, in order for a function to revert when called when the contract is paused, it has to be decorated with the `whenNotPaused` modifier, but no function in `FanticoNFT.sol` is.
Mitigation	Decide which functions should revert when the contract is paused and add the modifier to their declarations.

[M] USDC Only Supports 6 Decimals

ID	SAY-04
Status	Fixed
Risk	Medium
Business Impact	This issue has the potential to break the contract, just like in SAY-01, but we downgraded it to medium risk because the variable is not even used, as things stand.
Location	- `PaymentGatewayContract.sol:9`
Description	`USDC_DECIMALS` is set to 18, although USDC only supports 6 decimals.
Mitigation	Set `USDC_DECIMALS` to 6.

[M] Centralization Risk

ID	SAY-05
Status	Acknowledged
Risk	Medium
Business Impact	Excessive centralization can reduce trust in the project and lead to a generally more fragile system.
Location	- `PaymentGatewayContract.sol; withdraw(uint256), Pay(uint256)` - `FanticoNFT.sol; safeMint(address)`
Description	Owner Privileges: 1. In `PaymentGateway`, the owner can decide to withdraw an arbitrarily large amount of funds from the contract. NFT Minting Process: Here is a general outline of the minting process: 1. The user sends the server an amount of fiat currency along with an address. 2. The server converts the fiat to USDC and the server calls the pay function of `PaymentGateway` with the USDC. 3. The server calls `safeMint(address)` in `FanticoNFT.sol` to mint the next Funtico NFT with the address the user provided in step 1. As you can see, much of the process happens off-chain and requires the user to *trust* that the server will provide an NFT for their payment.
Mitigation	Consider analyzing the level of centralization/decentralization of the project and ensure they meet the project requirements. Ensure to test and analyze the server for edge cases like below and others: 1. If the user sends the server more fiat than necessary to mint an NFT, is he refunded the reminder? 2. 1 USDC is not always equal to $1. For instance, the exchange rate was 1USDC = $0.9912 at one point in 2023. 3. What if the USDC transfer fails? For example, the USDC contract can be paused. 4. What happens when a user supplies a contract address that reverts when trying to mint NFT to it?

[L] Ownership Issues

ID	SAY-06
Status	Fixed
Risk	Low
Business Impact	The process in `Ownable2Step` first has the current owner to initiate a transfer, and then requires the new owner to accept it. This can prevent accidental loss of ownership.
Location	- `PaymentGatewayContract.sol` - `FanticoNFT.sol`
Description	1. `FanticoNFT.sol` inherits `Ownable`, which is not ideal. Using `Ownable2Step` is considered a better practice. 2. `PaymentGatewayContract.sol` defines an `owner` state variable and an `onlyOwner()` modifier but does not inherit Ownable nor does it implement an ownership transferal process. 3. Having inherited `Ownable`, `FanticoNFT.sol` allows the owner to renounce their ownership. In your case, this would mean that no new tokens could be minted. This is not too much of an issue, but to stay on the safe side, `renounceOwnership()` could be disabled.

Mitigation	For 1 and 2, inherit OpenZeppelin's `Ownable2Step` in both contracts. For 3, disable `renounceOwnership()` by overriding it:
	`function renounceOwnership() public override onlyOwner {`
	`revert("Ownership can not renounced");`
	`}`

[L] Function Can Be Declared External

ID	SAY-07
Status	Fixed
Risk	Informational
Business Impact	When declaring a function external, the compiler allows arguments to be read directly from calldata, rather than first copying them to memory. This saves on gas. When the number of arguments and their size is small, the benefits are minimal, but it's still a worthwhile habit to have.
Location	- `PaymentGatewayContract.sol; Pay(uint256)`
Description	This public function is never used within the contract, but is not declared external.
Mitigation	Add the external keyword to this function declaration.

[L] *PaymentGateway* Is Not Upgradeable

ID	SAY-08
Status	Acknowledged
Risk	Informational
Business Impact	Upgradability allows the developer to easily repair issues in the contract.
Location	- `PaymentGatewayContract.sol; contract PaymentGateway`
Description	PaymentGateway is a prime target for implementing upgradeability since it has a lot of functionality that could easily go wrong.
Mitigation	Consider making PaymentGateway upgradeable.

[I] Consider Adding a Token Rescue Functionality

ID	SAY-09
Status	Acknowledged
Risk	Informational
Business Impact	Letting the owner rescue unsupported tokens can be helpful if users accidentally send them to the contract.
Location	- PaymentGatewayContract.sol
Description	If unsupported tokens are accidentally sent to the contract, there is no way to retrieve them since the contract only allows the withdrawal of USDC.

Mitigation	Consider adding a simple rescue function:
	`function rescueTokens(address token, uint amount) external nonReentrant onlyOwner {`
	`  require(token != usdcToken, "Only Unsupported tokens");`
	`  token.transfer(owner, amount);`
	`}`

[I] Unnecessary Allowance Check

ID	SAY-10
Status	Fixed
Risk	Informational
Business Impact	We suspect that this is simply incomplete dummy code, but we thought it would be useful to include this as an informational finding.
Location	- `PaymentGatewayContract.sol:30-32; pay(uint256)`
Description	`usdcToken.transferFrom( ... )` already checks allowance and would revert if the allowance is less than the `_amount`.
Mitigation	The check can be safely removed.

[I] Invalid *baseURI*

ID	SAY-11
Status	Fixed
Risk	Informational
Business Impact	We suspect that this is simply incomplete dummy code, but we thought it would be useful to include this as an informational finding.
Location	- FanticoNFT.sol:19; _baseURI()

Description	The URL provided by `_baseURI()` is invalid:
	`function _baseURI() internal pure override returns (string memory) {`
	`  return "https://..../metadata01/";`
	`}`

Mitigation	Replace the dummy URL with a real one.

[I] Lack of Natspec Comments

ID	SAY-12
Status	Fixed
Risk	Informational
Business Impact	Natspec comments help generate rich documentation for the contract.
Location	- `FanticoNFT.sol` - `PaymentGatewayContract.sol`
Description	The specified contracts have no Natspec comments. This stands in great contrast to the two token comments, which are well commented.
Mitigation	Consider adding Natspec comments to these two contracts.

[I] Avoid Floating Pragmas

ID	SAY-13
Status	Fixed
Risk	Informational
Business Impact	It's considered best practice to agree on a single solidity version and roll with it throughout the project, so there are no mismatches between contracts.
Location	- `Timelock.sol:2` - `FanticoNFT.sol:2` - `PaymentGatewayContract.sol:2`
Description	The contracts all use floating pragmas. It's also notable that they don't agree on which pragma version to use.
Mitigation	Agree on a single solidity version and enforce it throughout your project.

[I] Solidity 0.8.20+ May Not Work on All L2 Chains

ID	SAY-14
Status	Acknowledged
Risk	Informational
Business Impact	Deployment on some L2 will fail.
Location	- `FanticoNFT.sol:2`
Description	The compiler for Solidity 0.8.20 switches the default target EVM version to Shanghai, which includes the new PUSH0 opcode. This opcode may not yet be implemented on all L2s, so deployment on these chains will fail.
Mitigation	To work around this issue, use an earlier EVM version like 0.8.19 if the contract is going to be deployed to L2s or networks that don't support the PUSH0 opcode yet.

[I] Unused Event

ID	SAY-15
Status	Fixed
Risk	Informational
Business Impact	This finding is purely informational and has no impact.
Location	- `PaymentGatewayContract.sol:20`
Description	The event `Allowance(address, address, uint)`, declared in line 20, is left unused throughout the contract.
Mitigation	We recommend reviewing the code to determine whether the event is even necessary.

[I] Deviations from Naming Scheme

ID	SAY-16
Status	Fixed
Risk	Informational
Business Impact	This finding is purely informational and has no impact.
Location	- `FanticoNFT.sol` - `PaymentGateway.sol; Pay(uint256)`
Description	1. Contract names in the project use a PascalCase convention, with the exception of `FanticoNFT.sol`, which is lowercase. 2. Functions names use a camelCase, `Pay(uint256)`, which is in Title Case.
Mitigation	If possible, add a linter to your workflow to rectify these tiny deviations when they appear.