Smart Contract Audit Report for Modex

<u>Testers</u>

1. Or Duan
2. Avigdor Sason Cohen

Table of Contents

Management Summary

Modex contacted Sayfer to perform a security audit on their smart contracts in December 2024.

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 Modex's smart contracts.

Over the research period of 4 weeks, we discovered 9 vulnerabilities in the contract.

Several fixes should be implemented following the report, to ensure the system's security posture is competent.

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:

Overall Risk Security

IMPACT		LIKELIHOOD		
HIGH	Medium	High	Critical
MEDIUM	Low	Medium	High
LOW	Informational	Low	Medium
	LOW	**MEDIUM**	**HIGH**

Vulnerabilities by Risk

Informational
22.2%

Low
44.4%
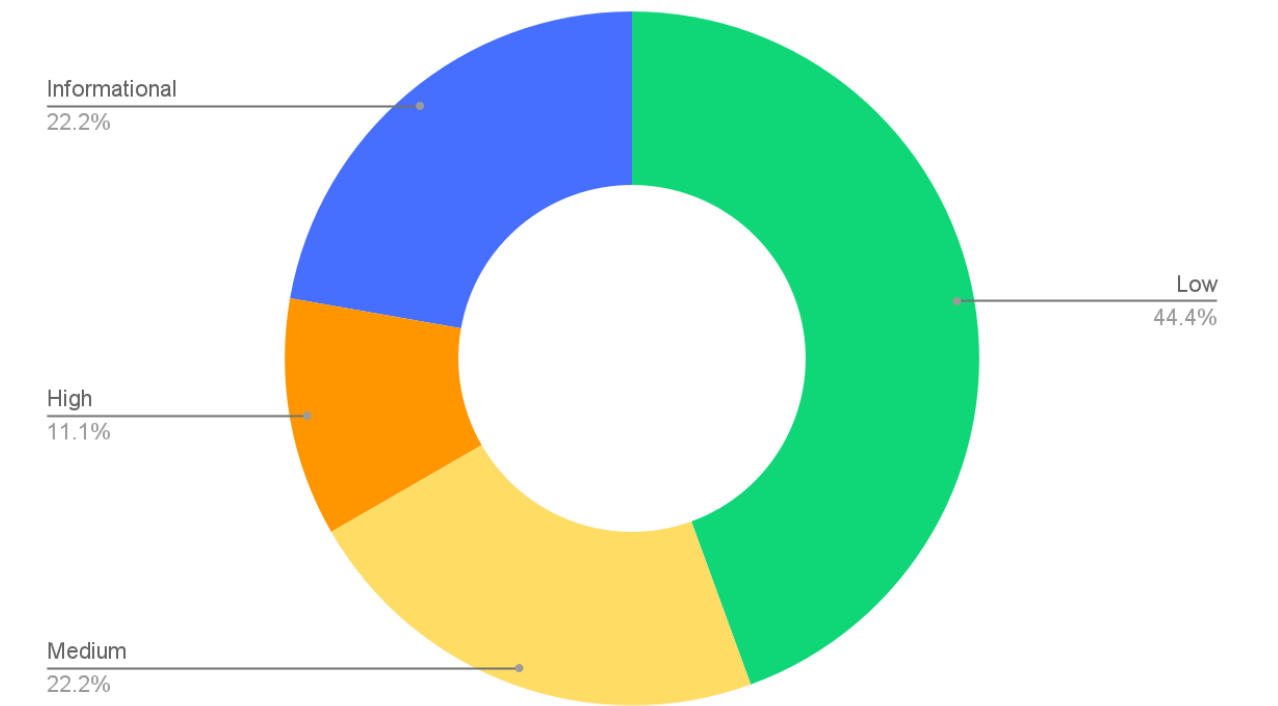
High
11.1%

Medium
22.2%

Risk	Low	Medium	High	Critical	Informational
# of issues	4	2	1	0	2

Approach

Introduction

Modex 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 contracts as the scope of the project.

Contract	SHA-256
ERC20.sol	1548afcecc27ee7b2d643d661c7d6620855c328329f2fcb2a568dbf508e1b049
ERC721CWithBasicRoyalties.sol	9a3c2334369e1f811ed157f59ed7a055e7cc7d990d001d538808833d2c4fd80c
Factory.sol	8c8a373b36142ab1bad4a2e1966d5bea407a031a69d04443191db083f21feeb5
KYCNFT2.sol	b1c0dab279bed86c871e6456f11827751c8a32d0d32f88ed595525aac08e6421

Our tests were performed from 12/12/2024 to 02/01/2025.

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.

Protocol Overview

Protocol Introduction

Modex is a Web3 technology company that provides blockchain-based solutions to help businesses enhance user engagement and create new monetization opportunities. It enables enterprises to integrate blockchain technology seamlessly, offering innovative ways to interact with audiences.

One of its flagship products is the **Modex Ledger DB**, a patented Blockchain Database (BCDB) that combines traditional database functionality with blockchain features. This technology ensures real-time data integrity, secure sharing, and simplified regulatory compliance, catering to diverse industries.

The **Modex Token (MODEX)** is the ecosystem's utility token, fostering user engagement and community interaction while serving as a foundation for future blockchain-based applications.

Modex is also a partner of FIFA, powering the **FIFA Collect platform**, a digital collectibles initiative for football fans globally. This collaboration highlights Modex's capability to integrate blockchain into large-scale projects, blending sports legacy with Web3 innovation.

With a mission to revolutionize enterprise blockchain adoption, Modex provides cutting-edge solutions tailored to world-class brands. Their focus is on empowering businesses with tools to amplify revenue streams and enhance user experiences, setting new benchmarks in the Web3 space.

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).

Security Assessment Findings

Unrestricted KYC Token Transfers

ID	SAY-01
Status	Open
Risk	High
Business Impact	The current implementation allows for bypassing the KYC identity mechanism by simply transferring KYC tokens.
Location	- KYCNft2.sol:13-20 - Factory.sol; transferTickets(address, address, uint256[], uint256[])
Description	The KYC token is implemented as a standard ERC1155 token without transfer restrictions in KYCNft2.sol. But the planned KYC verification system, as evidenced by commented code in Factory.sol, intends to verify user KYC status by checking token balance. • Factory.sol:241-245 ```// IERC1155 kyc = IERC1155(kycTokenAddr);\n// require(\n// kyc.balanceOf(transferTo, 0) == 1,\n// "Recipient doesn't have KYC"\n//);``` Once the planned KYC verification is implemented, users could bypass it entirely by transferring KYC tokens between addresses.
Mitigation	Override ERC1155's transfer functions to prevent KYC token transfers. Implement the token as soulbound by having all transfer-related functions revert for token ID 0 (KYC). Store KYC verification data on-chain with the token serving as a non-transferrable proof of verification.

Non-Sequential Match IDs Break Ticket Collection Logic

ID	SAY-02
Status	Open
Risk	Medium
Business Impact	Non-sequential match IDs create a mismatch between the contract's counting and iteration logic, causing the system to miss user tickets and provide incomplete data, potentially affecting ticket management and user access. The issue can only be introduced by an Admin, but the likelihood of its occurrence is increased by the project's design.
Location	- Factory.sol; createFifaEventNft(address, address, string, string, uint256, uint256, uint96, uint256, uint256) - Factory.sol; _collectAllTickets(address)
Description	createFifaEventNft(…) allows the caller to choose their own arbitrary matchIDs. ● Factory.sol; createFifaEventNft(address, address, string, string, uint256, uint256, uint96, uint256, uint256)

```
function createFifaEventNft(
    address _minter,
    address _owner,
    string memory _name,
    string memory _symbol,
    uint256 matchID,
    uint256 startTimestamp,
    uint96 royaltyPercent,
    uint256 totalSupply_,
    uint256 amountPerAddress_
) external onlyManager onlyNoPaused {
    if (matchContractAddresses[matchID] ≠ address(0)) revert
MatchIDAlreadyExists();
    // ... creation logic
                            [ … ]
        matchContractCount++;
        matchContractAddresses[matchID] = NFTAddr;
                            [ … ]
}
```

However, `_collectAllTickets(address)` iterates based on `matchContractCount`.

- `Factory.sol; _collectAllTickets(address)`

```solidity
function _collectAllTickets(address user) internal view returns
(Ticket[] memory) {
    uint256 totalMatches = matchContractCount;
    // ... initialization
    for (uint256 i = 0; i < totalMatches; i++) {
        address matchContractAddr = matchContractAddresses[i];
        // ... ticket collection logic
    }
    // ... return logic
}
```

This creates a dangerous inconsistency. For example, if match IDs are created as [1, 2, 1000, 10000], the system will only check addresses 0-3 in `matchContractAddresses[]`, missing all tickets from matches with higher IDs.

Mitigation	Modify `createFifaEventNft( ... )` to enforce sequential match IDs by using `matchContractCount` as the matchID parameter.

Incorrect Maximum Ticket Validation

ID	SAY-03
Status	Open
Risk	Medium
Business Impact	Users who already own tickets from previous purchases are incorrectly prevented from receiving new tickets even when below the global per-address limit, leading to failed mints and potential loss of access to desired tickets.
Location	- `Factory.sol; mintTickets(address, Ticket[])`

Description	`mintTickets(address, Ticket[])` implements a validation check for the maximum number of tickets per address by checking `nftContract.balanceOf(mintTo)`.
	● Factory.sol:339-342
	
	```solidity\nif (nftContract.balanceOf(mintTo) + 1 > amountOfTicketsPerAddr) {\n amountToReturn = amountToReturn + tickets[i].price;\n continue;\n}\n```
	
	This check is redundant and incorrect because the NFT contract already implements proper validation through its `canMint(address, uint256)` modifier.
	● ERC721CWithBasicRoyalties.sol:107-110
	
	```solidity\nmodifier canMint(address to, uint256 amount) {\n require(_addressTokenCount[to] + amount ≤ amountPerAddress,\n"EMTPA");\n _;\n}\n```
	
	This check uses `_addressTokenCount` which correctly tracks all mints to an address, making the Factory's balance check both unnecessary and potentially harmful by preventing legitimate mints that would otherwise be allowed by the NFT contract's tracking system.

Mitigation	Remove the balance check from `Factory.sol` and rely solely on the NFT contract's `canMint(address, uint256)` modifier, which already provides accurate validation of the per-address limit using proper accounting of historical mints.

Unbounded Loop

ID	SAY-04
Status	Open
Risk	Low
Business Impact	Functions that retrieve user tickets lack pagination and bounds, potentially causing transaction failures for users with many tickets and inefficient gas usage across the protocol.

Location	- `ERC721CWithBasicRoyalties.sol; getUserTickets(address)`
	- `Factory.sol`
	    - `getAllUserTickets(address)`
	    - `_collectAllTickets(address)`

Description	The specified functions implement two different ticket retrieval mechanisms that process entire arrays without pagination or limits. The nested loops and unbounded array operations could become problematic as users accumulate tickets over time or if the protocol scales to handle more matches.
Mitigation	Implement pagination in both functions by adding `startIndex` and limit parameters. Additionally, consider creating a separate function to retrieve tickets per specific match to allow more granular data fetching. This would help manage gas costs and prevent potential out-of-gas scenarios.

Minter Permissions Cannot Be Revoked

ID	SAY-05
Status	Open
Risk	Low
Business Impact	The inability to revoke minting privileges from compromised or incorrectly assigned minter accounts creates a permanent vector for attack that cannot be remediated even if a minter is compromised.
Location	- `ERC721CWithBasicRoyalties.sol`
Description	`ERC721CWithBasicRoyalties.sol` provides functionality to add minters through `addMinter(address)` but lacks the ability to remove them. While minter addition is an admin-controlled operation, the absence of a removal mechanism creates a security risk if a minter account is compromised or incorrectly assigned.
Mitigation	Implement a `removeMinter(address)` function or replace the current implementation with a function like `setMinter(address minter, bool status)` function that can both grant and revoke minting privileges. This provides the necessary operational flexibility for secure role management.

Missing Validation in `transferTickets( ... )`

ID	SAY-06
Status	Open
Risk	Low
Business Impact	Inconsistent array lengths between the `tokenIds` and `matchIds` parameters may cause transaction failures.
Location	- `Factory.sol:235; transferTickets(address, address, uint256[], uint256[])`
Description	`transferTickets( ... )` accepts two arrays, `tokenIds` and `matchIds`, and processes them in parallel using a loop, but lacks a check to ensure that these arrays have matching lengths. If `matchIds` is shorter than `tokenIds`, the function will attempt to access an out-of-bounds index, causing a revert and unnecessarily increasing gas costs.
Mitigation	Add a check at the beginning of the function to ensure both arrays have equal length.

Centralization Risk

ID	SAY-07
Status	Open
Risk	Low
Business Impact	The centralization of the minting mechanism puts users at risk if the owner is compromised or acts maliciously; token holders must place complete trust in the owner's discretion over monetary policy.
Location	- ERC20.sol; mint(address, uint256)
Description	ERC20.sol implements a token minting mechanism that is exclusively controlled by the contract owner through the onlyOwner() modifier. While this is a common pattern, it represents a centralization risk since the owner has unrestricted ability to mint new tokens at any time. ● ERC20.sol:26-30
Mitigation	Consider implementing supply caps and rate limiting mechanisms to provide predictable token emission. Additionally, a time-lock or multi-signature scheme could be introduced to increase the transparency and security of minting operations.

```solidity
function mint(address to, uint256 amount) public onlyOwner {
    _mint(to, amount);
    uint256 totalAmount = balanceOf(to);
    emit PaymentTokenMinted(to, amount, totalAmount);
}
```

Commented Code

ID	SAY-08
Status	Open
Risk	Informational
Business Impact	Leftover commented code and unfinished development notes create maintenance overhead and potential confusion for future developers, increasing the risk of implementation errors during future updates.
Location	- `Factory.sol:7, 130-140, 163, 431` - `ERC721CWithBasicRoyalties.sol:321-326, 183`
Description	The codebase contains multiple instances of commented-out code blocks and incomplete features marked as "future implementation." While comments are valuable for documentation, abandoned code fragments and incomplete features can mislead developers about the contract's intended functionality and implementation status.
Mitigation	Remove all commented-out code blocks and move any planned feature documentation to a separate technical specification document. If certain features are planned for future implementation, document them in the project roadmap rather than within the source code.

Missing Event Emissions for Key State Changes

ID	SAY-09
Status	Open
Risk	Informational
Business Impact	Lack of event emissions reduces transparency and makes it difficult for off-chain applications and users to track important contract state changes.
| Location | ```
-  Multiple occurrences:
    -  ERC721CWithBasicRoyalties.sol; setAmountPerAddress()
    -  Factory.sol; initialize(), setPause()
    -  Various constructor functions
``` |
| Description | The specified functions modify important contract parameters but do not emit corresponding events. This makes it harder for users and monitoring tools to track the contract's state changes effectively. |
| Mitigation | Add appropriate event emissions in important state-changing functions. Define events that capture the old and new values when applicable, and emit them at the end of the function. |

We are available at security@sayfer.io

If you want to encrypt your message please use our public PGP key:

https://sayfer.io/pgp.asc

Key ID: 9DC858229FC7DD38854AE2D88D81803C0EBFCD88

Website: https://sayfer.io

Public email: info@sayfer.io

Phone: +972-559139416