sigma prime

Satori Sports Token Royalties

Smart Contract Security Review

Version: 2.0

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of a set of Satori Sports smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contracts, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Satori Sports smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Satori Sports smart contracts.

Overview

Satori Sports is a blockchain-based solution to allow monetisation of athletes' images in the form of Non-Fungible Tokens (NFTs). By using NFTs, fans are given opportunities to own digital collectibles of their favourite athletes. At the same time, Satori maintains an on-chain record of NFT sales' royalty distribution to specified parties, each with their own portions.

Satori utilises ERC-721 and ERC-1155 to generate NFTs. Royalties are stored separately in a `Royalties` contract, while configuration on how the royalty is distributed is stored in `RoyaltiesConfig` contract.

Satori handles off-chain payments separately from the on-chain components. The royalties owed to related entities are stored on-chain and are cleared once payments are made. Most of the smart contract operations are handled by Satori as the contract owner.

Security Assessment Summary

This review was conducted on the files hosted on the Satori Sports repository and were assessed at commit a5022db.

Specifically, the files in scope are as follows:

- `Token1155.sol`
- `TokenCollectibles721.sol`
- `TokenFactory721.sol`
- `TokenFactory1155.sol`

- `TokenRoyalties.sol`
- `TokenRoyaltiesHandler.sol`
- `Royalties.sol`
- `RoyaltiesConfig.sol`

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

Retesting activities targeted commit 6581910.

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [?, ?].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 8 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 1 issue.
- Low: 1 issue.
- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Satori Sports smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- *Open:* the issue has not been addressed by the project team.

- *Resolved:* the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.

- *Closed:* the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
SAT-01	Strict Equality Causes Token Purchase Failure	**Critical**	**Resolved**
SAT-02	Abritrary Users Can Manipulate Profit Calculations	High	**Resolved**
SAT-03	Large Entities Array May Revert During `purchaseToken()`	Low	**Resolved**
SAT-04	Token Purchase Behaviour Subject to Single Point of Failure	Informational	**Resolved**
SAT-05	Owner's Excessive Access Control on `Royalties` Contract	Informational	**Resolved**
SAT-06	Inefficient Operation of Tokens with Pending Payment	Informational	**Closed**
SAT-07	Miscellaneous General Comments	Informational	**Resolved**
SAT-08	Additional Miscellaneous General Comments	Informational	**Resolved**

SAT-01	Strict Equality Causes Token Purchase Failure		
Asset	`TokenRoyaltiesHandler.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `purchased()` function in `TokenRoyaltiesHandler.sol` splits the payment made from an NFT sale and divides it amongst the `entities` according to the percentages allocated in `royaltiesBPS`. Under certain circumstances, it is possible for users to submit an order with Satori, and have the payment correctly processed, but the delivery of the NFT fail due to strict equality checks.

As can be seen on line [22] in the code block below, the external royalties contract is updated to reflect the share of NFT `purchaseCost`. The following require statement on line [26] then checks that this calculation has been performed correctly and that the calculated `totalPaid` sums to the `purchaseCost`.

```
18   for(uint256 i = 0; i < cfg.getRoyaltiesConfiguration().entities.length; ++i) {
         uint256 owed = cfg.getRoyaltiesConfiguration().royaltiesBPS[i] * purchaseCost / 10000; // bps 2dp
20       totalPaid += owed;
         royalties.recordRoyalties(cfg.getRoyaltiesConfiguration().entities[i], owed);
22       emit RoyaltiesPaid(address(this), cfg.getRoyaltiesConfiguration().entities[i], tokenId,
           ↪   cfg.getRoyaltiesConfiguration().royaltiesBPS[i], owed);
     }
24
     require(totalPaid == purchaseCost, "Royalties payment not calculated correctly");
```

This `require` statement acts as an invariant check. However, the strict equality (using operator `==`) will fail under certain circumstances due to the calculation performed on line [20]. If the `cfg.getRoyaltiesConfiguration().royaltiesBPS[i] * purchaseCost` is not a direct multiple of `10000` the resultant `owed` will be rounded down. Any rounding errors will cause the `require(totalPaid == purchaseCost)` to fail, effectively preventing any calls to `Token._transferPendingToOwner()` from finalising. This stops the user from actually becoming the owner of the purchased tokens.

Recommendations

The testing team recommends avoiding strict equalities in this instance. If the `require` statement is aimed at ensuring the amount of funds paid for the `purchaseToken()` does not exceed the amount paid to individual `entities`, then it may be worth considering the following assertion instead:

```
require(purchaseCost >= totalPaid, "Royalties payment not calculated correctly");
```

Additionally, most tests are written as follows: `expect(event.args[1]).equal(ethers.utils.parseEther("1"));` which only validate royalties payment for very simple purchases. The testing team recommends writing more complex scenarios to ensure that edge cases such as the one discussed here are well accounted for.

Resolution

The issue has been fixed on commit 923c35b. The responsibility of dividing values by 1bps is handed over to the UI. The testing team recommends documenting this behaviour.

SAT-02	Abritrary Users Can Manipulate Profit Calculations
Asset	`TokenFactory1155.sol, TokenFactory721.sol, Royalties.sol`
Status	**Resolved:** See Resolution
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The Royalties contract tracks royalties earned by stakeholders during the sale of an athlete's collectibles. This calculation is based on a set `RoyaltiesConfig` which requires a set of addresses corresponding to `entities` that take a share by percentage (`royaltiesBPS`) of the NFT sale in `Token1155.purchaseToken()` and `TokenCollectibles721.purchaseToken()`.

The protocol makes several critical assumptions with regards to access control, some of which can be bypassed to impact profit calculations, as follows:

1. Satori System Administrator is the only user who can mint new athlete tokens and complete token purchases. Therefore, the System Administrator is the only user who can directly impact the royalties calculation of an entity.

2. Royalties are calculated upon successful purchase (credit card payments) and notification by Satori (indicated by the `onlyOwner` modifier in function `purchaseToken()` of `Token1155` and `TokenCollectibles721` contracts).

3. Satori will hold the custody of these tokens and manage the users' credit card transactions and purchases of these tokens.

These assumptions have several unaccounted edge cases:

`TokenFactory1155.createToken(IRoyalties _royalties)` accepts any `Royalties` contract as an input parameter. As the `TokenFactory1155` contract has the permissions needed to execute `_royalties.grantRoyaltiesTo(address(proxy))`, anyone who creates a token has the role type `ROYALTIES_ROLE`. Anyone who holds this role has the ability to impact the profit calculation of all deployments of the `Royalties` contract. As there is no access control on the `createToken()` function, any arbitrary user is capable of impacting any royalties calculated by the Satori smart contracts.

Token owners are able to settle payment transactions off-chain and declare the purchase has been completed via the `purchaseToken()` function. Since anyone can create a token, anyone can make claims that payments have been settled. A malicious user could make these claims even when assets have not been successfully purchased and paid for.

Satori may act as the custodian for some tokens and manage users transactions, but this custodianship is not all-encompassing as some tokens can be created without any vetting from Satori.

Proof of Concept

1. `TokenA` is created and then minted, with a normal `pricePerToken` that reflects 1 USDC per token by a vetted athlete or their custodian (ie Satori may do this on behalf of an athlete). The royalties of `TokenA` is tracked using `RoyaltiesA`

2. An attacker (arbitrary user) creates an unvetted and unapproved `TokenB` through `TokenFactory.createToken()`. This `TokenB` points back to the royalties contract that tracks royalties for `TokenA`, namely `RoyaltiesA`. The attacker that is the owner of `TokenB` then mints tokens that have an arbitrarily high value 1M USDC.

3. The attacker purchases their own token, namely `TokenB`, by calling `TokenB.purchaseToken()` with `_pendingPayment=false`. Through this transaction, the attacker manipulates the `RoyaltiesA` and increases `royaltiesOwed.rollupTotal` of all related parties in the royalties configuration.

Note: As long as they are part of the entities in the original royalties contract they will earn a share of this fake sale. This is not a requirement however, malicious users who are not entities can use the same attack vector just to manipulate profit calculations.

Recommendations

If arbitrary users are not supposed to impact the royalties calculation of any `Royalties` contract, they should not be able to create and mint tokens. The testing team recommends providing stricter access control for the `TokenFactory.createToken()`, for example, by using the `OwnableUpgradeable` library on the factory contract and adding the modifier `onlyOwner` on `createToken()`. Likewise, tokens owned by separate users should not impact royalty calculations unexpectedly for another user.

Resolution

The issue has been fixed on commit 2227525. The implemented fixes ensure that only the contract owner can call `createToken()`.

SAT-03	Large Entities Array May Revert During `purchaseToken()`
Asset	`RoyaltiesConfig.sol, TokenRoyaltiesHandler.sol`
Status	**Resolved:** See Resolution
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `RoyaltiesConfig.sol` contract manages the configuration of payment allocation to each `entity` during an NFT sale. The `TokenRoyalties.sol` library then validates that the amount of royalties matches the expected configured amount from `RoyaltiesConfig.sol`.

Due to a discrepancy in gas usage between these two contracts, it is possible to initialise a `Token1155` or `TokenCollectibles721` contract and facilitate the purchase of tokens where payment finalisation is still pending. When payment finalisation occurs, the increased gas cost usage in `TokenRoyaltiesHandler.purchased()` will cause the transaction to revert. This effectively prevents any finalisation of purchased tokens, when the entity array exceeds a certain value (this was determined to be roughly 20-40 entities depending on gas usage limits).

We illustrate the discrepancy below:

The following snippet is extracted from `RoyaltiesConfig.sol`, here we can see a simple calculation which checks that the percentage of allocated shares given to each entity is exactly 100%.

```
for (uint256 i = 0; i < _royalties.royaltiesBPS.length; ++i) {
    bps += _royalties.royaltiesBPS[i];
}
require(bps == 10000, "Royalties config must add up to 10000 (100.00\%)");
```

Below we see the code snippet from `TokenRoyaltiesHandler.sol`. Here we notice several additional calculations, along with external calls which all result in considerably larger gas usage.

```
for(uint256 i = 0; i < cfg.getRoyaltiesConfiguration().entities.length; ++i) {

    uint256 owed = cfg.getRoyaltiesConfiguration().royaltiesBPS[i] * purchaseCost / 10000; // bps 2dp
    totalPaid += owed;
    royalties.recordRoyalties(cfg.getRoyaltiesConfiguration().entities[i], owed);
    emit RoyaltiesPaid(address(this), cfg.getRoyaltiesConfiguration().entities[i], tokenId,
      ↪ cfg.getRoyaltiesConfiguration().royaltiesBPS[i], owed);
}
```

Recommendations

The testing team recommends setting an upper bound for the number of entities. This should prevent inconsistent states where the system appears to be functional but cannot process token purchases successfully.

Resolution

The issue has been fixed on commit f70abb4. The maximum length of entities is now 10.

SAT-04	Token Purchase Behaviour Subject to Single Point of Failure	
Asset	`Token1155.sol, TokenCollectibles721.sol`	
Status	**Resolved:** See Resolution	
Rating	Informational	

Description

Currently there are no access control restrictions on token creation, therefore any user can be the `owner` of a `Token1155` or `TokenCollectibles721` contract. The expectation from the testing team based on information provided from the development team, is that regular usage of the Satori smart contracts would typically have Satori as the custodian of the smart contracts. However, in both situations, errors can occur during token purchasing that may render the `Token1155` and `TokenCollectibles721` irretrievable.

Token purchasing is handled in two parts and managed by the `onlyOwner` role (the same user who created the token initially). The first part of the process is managed by `purchaseToken()` where the `onlyOwner` designates whether the payment has been finalised. If payment has been finalised, then no the tokens are transferred straight from the owner to the purchaser. However, if the payment is marked as `_pendingPayment = true`, then the amount is allocated to the `Token1155` or `TokenCollectibles721` contract.

Once the payment has been completed, the owner can then use the `transferPendingToOwner()` function to specify whether to transfer the withheld tokens back to the contract owner, or whether to transfer them to the purchaser. The assumption is that failed payment transfers will be refunded back to the contract owner, and successfully finalised payments will be forwarded to the relevant user.

This process has several drawbacks:

1. The owner can recall token purchases that have been successfully paid off chain, whilst claiming they have not been paid.

2. The owner is able to ignore valid token purchases.

Recommendations

At this stage anyone is able to use the `TokenFactory.createToken()` function, therefore some users might mistakenly create their own tokens. This may increase the likelihood that token sales are mismanaged or errors are introduced. Make sure this behaviour is understood and risks are minimised (e.g. through adding access control mechanisms to token creation).

Resolution

The issue has been fixed on commit 2227525. The new implementation ensures that only the contract owner can call `createToken()` which minimises rogue tokens.

SAT-05	Owner's Excessive Access Control on `Royalties` Contract
Asset	`Royalties.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `Royalties` contract's main function is to record royalties owed to entities regarding the sales of collectibles. The role of recording royalties data is performed by entities assigned with `ROYALTIES_ROLE`. In normal circumstances, the entities with this role are `Token1155` and `TokenCollectibles721` contracts. These contracts are granted `ROYALTIES_ROLE` by their respective factory contracts.

However, the owner of the `Royalties` contract also has `ROYALTIES_ROLE` by default. This is a potential centralisation concern, since the owner can directly update the royalties owed to entities by calling the `recordRoyalties()` function.

Recommendations

To improve the system's accountability and transparency, the testing team recommends that the owner of the contract should not have access to function `recordRoyalties()`. This means removing `ROYALTIES_ROLE` from the permissions granted to the owner. The role is assigned to the owner on line [**67**].

The case described above is just one example of what the owner can do on `Royalties` contract. The owner can also revoke or grant the `ROYALTIES_ROLE` from or to any entities. Not only that, the owner can also grant and revoke any roles to any entities of their preference.

The testing team understands that without the owner's role-granting power (using the `grantRole()` function derived from `AccessControlUpgradeable` library), there would be a circular reference between the `Royalties` contract and the factory contracts. This is because `Royalties` contract must grant `OPERATOR_ROLE` to factory contracts (in order to grant `ROYALTIES_ROLE` to the created contracts), while the factory contracts take `Royalties` as an input when creating tokens.

The development team may also want to reserve the role to address potential future security issues.

The testing team recommends exercising caution over the private key management of the owner account to prevent unauthorised access.

Resolution

The issue has been fixed on commit 73d646e. The owner no longer has `ROYALTIES_ROLE`.

SAT-06	Inefficient Operation of Tokens with Pending Payment	
Asset	`TokenRoyalties.sol, TokenCollectibles721.sol, Token1155.sol`	
Status	**Closed:** See Resolution	
Rating	Informational	

Description

The system allows for a purchaser to acquire a token with a pending payment. In this case, the system potentially conducts two transactions to finalise the purchase:

1. When the purchase order is received (and while the payment is still *pending*), the owner transfers the token ownership to the contract.

2. When the payment is cleared, the owner transfers the token to the purchaser. Otherwise, the token is reclaimed to the owner.

This method may be inefficient because it requires two transactions as described above. The owner (Satori) risks wasting transaction fees if payment is never cleared, because the token needs to be reclaimed to re-sell. Satori may also suffer potential loss if the purchase is cancelled because other buyers might be interested in acquiring the collectibles while the tokens are temporarily held by the contract.

Recommendations

Make sure this behaviour is understood. The testing team recommends not transferring tokens before payment is finalised.

Resolution

The issue has been acknowledged and marked as no fix by the development team with the following comment:

"The escrow token design was used to support certain payment gateways that had a 3-day+ hold on any credit-card purchases. As this is no longer an issue (stripe manages the transactions more effectively) we will directly credit the user with their tokens upon successful transaction with Stripe. This feature will not be used in our initial release and we acknowledge that this behaviour is understood."

SAT-07	Miscellaneous General Comments	
Asset	`contracts/*`	
Status	**Resolved:** See Resolution	
Rating	Informational	

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. `Token1155.sol`

   1a) **`_tokenIdTracker` increments unnecessarily during initialisation**

   The code on line [**48**] increments `_tokenIdTracker` unnecessarily. Instead of incrementing, the contract should directly assign a default value of `1`.

   1b) **Does not implement `transfer-accept-ownership` pattern.**

   The current transfer of ownership pattern calls the function `transferOwnership(address newOwner)` (inherited from OpenZeppelin's `OwnableUpgradeable` contract), which instantly changes the owner to the `newOwner`. This allows the current owner of the contracts to set an arbitrary address (excluding the 0 address). If the address is entered incorrectly or set to an unowned address, the owner role of the contract is lost forever.

   This issue can be mitigated by implementing a two-step process `transferOwnership`, whereby a new owner address is selected, then the selected address must call a `claimOwnership()` before the owner is changed.

   1c) **Unused variable `_target`**

   The `_transferPendingToOwner()` function overrides the parent `TokenRoyalties._transferPendingToOwner()` virtual function. However the `_target` variable appears to be unused and unnecessary.

   1d) **Important functions do not fire events**

   For convenience, it is recommended to add events to all important, state-changing functions such as `mintToken()` and `_transferPendingToOwner()`.

   1e) **Possible Redundant `RoyaltiesConfig.isValid()`**

   There is a possible redundant check on the `isValid()` function which is called in `Token1155.sol` on line [**147-148**]:

   ```
   _royaltiesConfig.isValid() &&
   !_royaltiesConfig.isEmpty(),
   ```

   This function checks `royaltiesCfg.entities.length == royaltiesCfg.royaltiesBPS.length`. However, the same check is already done in function `initialize()` in `RoyaltiesConfig.sol`. Assuming that `RoyaltiesConfig` contract is initialized, then this redundant check can be safely removed. Also, since the snippet above is inside the function `mintToken()` with the `onlyOwner` modifier, mistakes in `RoyaltiesConfig` are unlikely unless the owner behaves maliciously.

2. `TokenRoyaltiesHandler.sol`

   Event emitted during for-loop

   The code on line [**23**] emits an event inside a `for-loop`. As the emission of events costs substantial gas, the testing team recommends moving the event outside the for loop and providing a more meaningful event for the overall execution of the function itself (not the individual for loop events). With a simple use case of three entities, the gas savings in `Token1155.purchaseToken()` is estimated to be `44403` units.

3. `Royalties.sol`

   3a) **Redundant `RoyaltiesObj.entity`**

   The structure `RoyaltiesObj` is used in `mapping(address => RoyaltiesObj) public royaltiesOwed;` (line [**30**]). The mapping `royaltiesOwed` already contains the entity address as a key. The same information is stored in the `RoyaltiesObj` structure (line [**109**]), namely `RoyaltiesObj.entity`. Since the `RoyaltiesObj.entity` structure is not used anywhere else, it can be safely removed.

   3b) **Keyword `totalUSDCPaid` might be misleading**

   The notes received by the testing team as quoted below indicates that USDC is not used in the system:
   We are no longer using USDC in the platform (no treasury). All royalty payments are made off-chain based on the royalty calculations recorded on-chain
   However, USDC is still mentioned in the `Royalties` contract on at least five occurences. For example, line [**35**] specifies the use of the `totalUSDCPaid` variable in event `RoyaltiesRecorded`, variable `totalUSDC` in the `recordRoyalties()` function, and an in-line comment in line [**94**]. This might confuse royalty receivers if the royalty is not in USDC or equivalent.

   3c) **Royalties ownership management**

   The `Royalties` contract relies on the owner to update royalties owed to specific entities after payments are made to the entities through the `claimRoyalties()` function. However, the owner can renounce ownership by calling the `renounceOwnership()` function, either by mistake or intentionally. If this occurs, the contract loses the capability of updating royalties information on the contract permanently.
   The testing team recommends disabling the `renounceOwnership()` function to avoid such scenario.

4. `TokenRoyalties.sol`

   Improving variable type on `pendingPayments`

   Variable `pendingPayments` is a two-dimensional mapping of `address` to `uint256` to `uint256` as specified in line [**16**]:

   ```solidity
   mapping(address => mapping(uint256 => uint256)) public pendingPayments;
   ```

   This variable stores pending payments of a purchaser to a token. Based on the current implementation, each record in `pendingPayments` will only have a value of either zero or one. In this case, the mapping `pendingPayments` can be modified to store `bool` which becomes the following:

   ```solidity
   mapping(address => mapping(uint256 => bool)) public pendingPayments;
   ```

   Our test shows 97 gas saving when using `bool` instead of `uint256`. Not only for gas efficiency, this modification can also simplify the contract logic, for example on line [**32**] and line [**41**] of `TokenRoyalties.sol` or on line [**71-72**] of `TokenCollectibles.sol`.

5. `TokenCollectibles721.sol`, `Token1155.sol`

   Potentially confusing `balanceOf()`

   Naturally, the `balanceOf()` function returns the number of tokens owned by an account. However, in `TokenCollectibles721` and `Token1155` contracts, the function `balanceOf()` also takes into account purchased tokens with a *pending* payment. This is a potentially confusing behaviour, because there is no guarantee that the payment is cleared which allows the purchasers to have full control over the tokens. In the current implementation, tokens with pending payment are held by the token contract and not the purchasers' account.

6. `RoyaltiesConfig.sol`

   Unmodifiable `RoyaltiesBPS`

   Contract `RoyaltiesConfig` stores information about how NFT sales royalties are distributed among related entities, namely the addresses and percentages. The contract does not support changes or modifications to the configuration, such that `TokenCollectibles721` or `Token1155` contracts created through their respective factory contracts have permanent royalty distribution configuration. On the other hand, athletes' circumstances may change, which affect the royalty distribution of their NFTs.

   Make sure this behaviour is intended.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team's responses to the raised issues above are as follows.

1. `Token1155.sol`

    1a) Fixed on commit 67d865f. The variable `tokenIdTracker` is now directly assigned with value `1`.

    1b) The issue has been acknowledged and marked as no fix with the following notes:

    Token1155 Does not implement transfer-accept-ownership pattern. All tokens in the satori platform are expected to be owned and managed by Satori on behalf of the athletes. Transfers of ownership will only occur within Satori managed wallets (either within a key management system or a multi-sig). Satori's governance model is aligned to a web 2.5 company; and we will secure against any malicious activites through company internal controls (e.g multisig)

    1c) The issue has been acknowledged and marked as no fix with the following notes:

    This function is overridden from parent contract (`TokenRoyalties`). The `_target` is required in `TokenCollectibles721` and both are used by `TokenRoyalties` to signify transfer

    1d) Fixed on commit 6581910. The mentioned functions now emit events.

    1e) Fixed on commit dad0d3f. Both checks were removed. See SAT-08 for additional recommendation regarding this issue.

2. `TokenRoyaltiesHandler.sol`

    Fixed on commit 7098ef0. The event `RoyaltiesPaid()` is now emitted outside the loop. See SAT-08 for additional recommendation regarding this issue.

3. `Royalties.sol`

    3a) Fixed on commit 496351e. The redundant variable was removed and all necessary changes were made.

    3b) Fixed on commit 2d99845. The term USDC was replaced with USD.

    3c) Fixed on commit 2d99845. The function `renounceOwnership()` was overridden with an empty function.

4. `TokenRoyalties.sol`

    The issue has been acknowledged and marked as no fix with the following notes:

    Pending payments capture the number of tokens that a user is owed. In the case of 1155 a user may have purchased more than one token; while in the case of 721 this value is always either 0 or 1. We should not replace this with a boolean value as it does not accurately capture the use-case for pending payments. (i.e. tokens owed to user pending payment from user).

5. `TokenCollectibles721.sol, Token1155.sol`

    The issue has been acknowledged and marked as no fix with the following notes:

    In cases where the payment gateway has a delay in collecting and settling payments (either due to fraud or charge-backs) the system should still allow the user who has purchased the token to use and unlock utility for the token while Satori waits for the payment gateway to settle the transactions. In cases where the transaction is rejected these tokens are reclaimed. A user should not wait for the payment to be settled before they are able to access the token utility. We use the token balance as a way to track token ownership and hence we have this expected behaviour.

6. `RoyaltiesConfig.sol`

The issue has been acknowledged and marked as no fix with the following notes:

Once royalties have been configured it may not be modified as this makes the calculations of royalties and reconciliation complex. If an athlete decides to modify their existing royalties they will mint new tokens with the updated royalties calculations. We are considering the case when royalties configurations need to be upgraded and will address this potential use-case in a future release beyond this first version of the Satori platform.

SAT-08	Additional Miscellaneous General Comments
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. `Token1155.sol`

   Removed `RoyaltiesConfig.isEmpty()`

   The previous suggestion in SAT-07 (point 1e) states that the redundant check was exclusive for `RoyaltiesConfig.isValid()`. However, the related fixes on commit dad0d3f also remove `RoyaltiesConfig.isEmpty()`.

   This fix would be less helpful for the owner to check whether the input `_royaltiesConfig` exists or not. Therefore, the testing team recommends to reinstate `RoyaltiesConfig.isEmpty()` on contract `RoyaltiesConfig.sol` and use it on `Token1155.mintToken()`.

   The testing team also suggests implementing the same check on `TokenFactory721.createToken()`.

2. `TokenRoyaltiesHandler.sol`

   Event emitted during for-loop

   The testing team suggests reconsidering whether it is necessary to emit a large amount of data (`entities` and `royaltiesBPS`) on the event `RoyaltiesPaid` emitted on function `purchased()` which consumes gas. Our experiment indicates that without emitting `entities` and `royaltiesBPS`, a related transaction spends 7.8% less gas.

   The program logic already guarantees that all entities will receive a royalty portion based on a predefined configuration. Therefore, if one wishes to know each entity's royalty from a purchase, the computation can be done off-chain by taking the configuration and the purchase cost.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

1. `Token1155.sol`

   Fixed on commit 9494cb48. The function `RoyaltiesConfig.isEmpty()` was reinstated.

2. `TokenRoyaltiesHandler.sol`

   The issue has been acknowledged and marked as no fix with the following notes:

   We acknowledge this comment however we are using the events and smart contract data for our calculations in our external systems and need these events as part of our verification process..

Appendix A   Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `brownie` framework was used to perform these tests and the output is given below.

```
test_init                             PASSED  [3%]
test_grantRoyaltiesTo                 PASSED  [6%]
test_recordRoyalties_claimRoyalties   PASSED  [10%]
test_renounceOwnership                PASSED  [13%]
test_access_control                   PASSED  [16%]
test_royalties_gas_exhaustion         PASSED  [20%]
test_invalid_royalties                PASSED  [23%]
test_mintToken                        PASSED  [26%]
test_purchaseToken                    PASSED  [30%]
test_token_transfers                  PASSED  [33%]
test_batch_transfer                   PASSED  [36%]
test_token_implementation_upgrade     PASSED  [40%]
test_token_purchase_failure_fixed     PASSED  [43%]
test_token_transfer_attack            PASSED  [46%]
test_initialize                       PASSED  [50%]
test_upgradeTo                        PASSED  [53%]
test_upgradeToAndCall                 PASSED  [56%]
test_renounceOwnership                PASSED  [60%]
test_purchaseToken_royalties          PASSED  [63%]
test_purchase_token_pbt               SKIPPED [66%]
test_721_mintMultipleTokens           PASSED  [70%]
test_721_mintMultipleTokens_multi     PASSED  [73%]
test_721_purchaseToken                PASSED  [76%]
test_721_token_implementation_upgrade PASSED  [80%]
test_init                             PASSED  [83%]
test_721_purchaseToken_pending_multi  PASSED  [86%]
test_create_token                     PASSED  [90%]
test_token_implementation_initialize  PASSED  [93%]
test_createToken_check                PASSED  [96%]
test_create_token                     PASSED  [100%]
```

Appendix B   Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

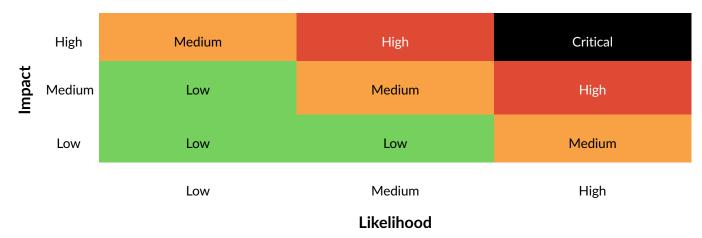| Impact			
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium
	Low	Medium	High

Likelihood

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.