Entertainmint

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen notdeghost@osec.io

Akash Gurugunti Sud0u53r.ak@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Entertainmint engaged OtterSec to perform an assessment of the `emint` program. This assessment was conducted between December 12th and December 20th, 2022.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team to streamline patches and confirm remediation. We delivered final confirmation of the patches January 4th, 2023.

Key Findings

Over the course of this audit engagement, we produced 6 findings total.

In particular, we found a critical issue that could lead to the stealing of protocol funds by the project owner (OS-ENT-ADV-00), as well as an issue with locking of protocol fee funds in the contract(OS-ENT-ADV-01).

We also made recommendations around gas optimizations on some functions (OS-ENT-SUG-00) and having re-configurable protocol fee percentages (OS-ENT-SUG-03).

Overall, we commend the Entertainmint team for being responsive and knowledgeable throughout the audit.

02 | Scope

The source code was delivered to us in a git repository at github.com/entertainmintlive/emint. This audit was performed against commit `26a6fae`.

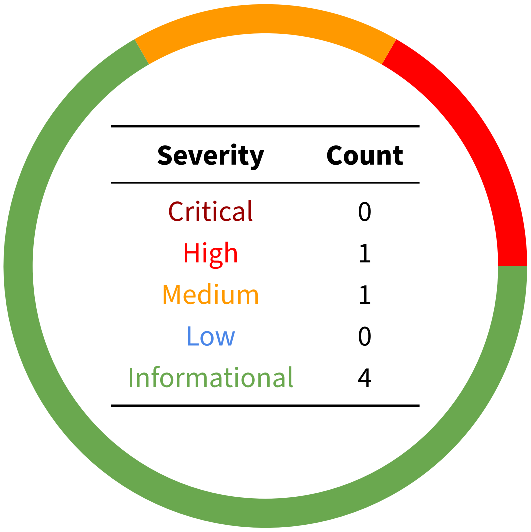
A brief description of the programs is as follows.

Name	Description
emint	A fundraising protocol for streamers and content creators.

03 | Findings

Overall, we report 6 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	1
Medium	1
Low	0
Informational	4

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-ENT-ADV-00	High	Resolved	Project owner can steal protocol funds by manipulating raise's parameters.
OS-ENT-ADV-01	Medium	Resolved	Protocol fee collected from a raise is locked in the contract.

OS-ENT-ADV-00 [high] [resolved] | Stealing Tokens From Protocol

Description

The protocol has a list of allowed addresses called creators. A creator can create projects and start raises for those projects. The `Raise` struct is used to store the values related to a raise. A raise also has tiers for storing different types of tokens, which have different supply caps and prices. These `Raise` and `Tier` values can be updated by the project owner when the raise is in `Scheduled` state.

The raise will be in `scheduled` state in two cases, before presale and in between presale end and public sale start. However, since the raise tokens are minted in presale, the project owner should not be able to modify the prices and the currency of the raise. This is not enforced properly, leading to a scenario where a project owner can steal tokens and ETH from the contract that is raised by other projects.

Proof of Concept

1. The controller adds a creator's address to the creators list.
2. Then, the creator creates a project and creates a raise onto that project with a pre-approved token as currency, along with their own address to the allowlist of address (for example, USDT) that can mint tokens in presale and a very low amount as the goal for the raise.
3. Then in the presale, the creator can mint the tokens and make the raise reach its goal.
4. When the raise is in its scheduled phase again (between presale end and public sale start), the creator can update the currency of the raise to a higher value currency (for example, ETH) and close the raise where the raise's state goes to `Funded`.
5. Now, the project owner can use the `withdraw` function to withdraw the `raise.balance` amount from the contract address, but instead of getting USDT, they will get ETH transferred to their account.

Remediation

This can be fixed by not allowing the project owner to change the raise and tier parameters after the start of minting raise tokens. This can be done by changing the condition in the `update` function, which checks if the raise's state is `Scheduled` to check if the `block.timestamp` is less than the `raise.presaleStart`.

Patch

Fixed by checking the block timestamp in 35c8d1f

OS-ENT-ADV-01 [med] [resolved] | Locking Of Protocol Fee In Contract

Description

In `Raises.sol`, the `close` function is called by the project owner to close a raise and change it's state to `RaiseState.Funded` if it has reached its goal amount. If a raise's state is changed from `RaiseState.Active` to `RaiseState.Funded`, the protocol fee amount from that raise should be stored in the global fee balance. Otherwise, the protocol fee collected from the raise will be stuck in the contract balance and cannot be retrieved.

src/Raises.sol SOLIDITY

```solidity
/// @inheritdoc IRaises
function close(uint32 projectId, uint32 raiseId) external override
    ↪  onlyCreators whenNotPaused {
    // Checks
    Raise storage raise = _getRaise(projectId, raiseId);
    if (raise.state != RaiseState.Active) revert RaiseInactive();
    if (raise.raised < raise.goal) revert RaiseGoalNotMet();

    // Effects
    emit CloseRaise(projectId, raiseId, raise.state = RaiseState.Funded);
}
```

Proof of Concept

1. A project owner creates a raise and closes the raise after it has reached its goal.
2. Now, if the protocol tries to collect the protocol fee for that raise, no amount is transferred, since the `raise.fees` is not added to `fees[raise.currency]`.

Remediation

A simple fix for this is to add the `raise.fees` value to `fees[raise.currency]` before changing the state to `RaiseState.Funded`.

Patch

Fixed by adding raise fees to global fees in 689b6f2

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent antipatterns and could lead to security issues in the future.

ID	Description
OS-ENT-SUG-00	Code refactoring that optimizes gas usage of the functions.
OS-ENT-SUG-01	Protocol Fee should be reconfigurable instead of hardcoded.
OS-ENT-SUG-02	Checking if project and raise exists before returning tier.
OS-ENT-SUG-03	Enforcing additional constraints on TierParams.

OS-ENT-SUG-00 | Gas Optimizations

Description

In RaiseCodec.sol and TokenCodec.sol, the decode functions decode the data from bits by first left shifting the respective mask to respective offset and apply bitwise AND operator on the bits and then again right shifting the result with respective offset to get the value of the field.

Remediation

This can be optimized by just repeatedly right shifting the bits and apply respective mask to get the field values. The below code snippet shows example implementation of decode function in RaiseCoded.sol:

src/libraries/codecs/RaiseCodec.sol SOLIDITY

```solidity
import {RaiseData, TierType} from "../../structs/RaiseData.sol";
import {ONE_BYTE, ONE_BYTE_MASK, FOUR_BYTES, FOUR_BYTE_MASK} from "../../constants/Codecs.sol";

uint240 constant PROJECT_ID_SIZE = uint240(FOUR_BYTES);
uint240 constant RAISE_ID_SIZE = uint240(FOUR_BYTES);
uint240 constant TIER_ID_SIZE = uint240(FOUR_BYTES);

function decode(bytes30 tokenData) external pure returns (RaiseData memory) {
    uint240 bits = uint240(tokenData);

    uint32 projectId = uint32(bits & FOUR_BYTE_MASK);
    uint32 raiseId = uint32((bits >>= PROJECT_ID_SIZE) & FOUR_BYTE_MASK);
    uint32 tierId = uint32((bits >>= RAISE_ID_SIZE) & FOUR_BYTE_MASK);
    TierType tierType = uint8((bits >>= TIER_ID_SIZE) & ONE_BYTE_MASK);

    return RaiseData({tierType: tierType, tierId: tierId, raiseId: raiseId, projectId: projectId});
}
```

Description

In RaiseToken.sol, the projectId unnecessarily decodes the entire tokenId just to get the projectId.

Remediation

This can be optimized by getting the required `projectId` field only from the `tokenId` by using bitwise operators. The code snippet below shows the implementation that can be done:

src/libraries/RaiseToken.sol SOLIDITY

```solidity
import {TokenCodec, DATA_OFFSET} from "./codecs/TokenCodec.sol";
import {RaiseCodec, PROJECT_ID_MASK} from "./codecs/RaiseCodec.sol";
function projectId(uint256 tokenId) internal pure returns (uint32 projectId) {
    uint32 projectId = uint32((tokenId >> DATA_OFFSET) & PROJECT_ID_MASK);
}
```

Description

In `Raises.sol`, in `redeem` and `_mint` functions, the `tierId` is ensured to be in bounds by throwing error if the given `tierId` is greater than length of the `tiers` array minus 1.

Remediation

This condition can be optimized by changing it to check if the given `tierId` is greater than or equal to length of the `tiers` array.

src/Raises.sol DIFF

```diff
    // Get the tier if it exists
-   if (tierId > tiers[projectId][raiseId].length - 1) revert NotFound();
+   if (tierId >= tiers[projectId][raiseId].length) revert NotFound();
    Tier storage tier = tiers[projectId][raiseId][tierId];
```

OS-ENT-SUG-01 | Reconfigurable Protocol Fee Percentage

Description

In `Fees.sol`, the `calculate` function is used to calculate the fee that should be taken by the protocol based on `tierType` and `mintPrice`. These percentage values are hardcoded into the code.

src/libraries/Fees.sol SOLIDITY

```
/// @title Fees - Fee calculator
/// @notice Calculates protocol fee based on token mint price.
library Fees {
    function calculate(TierType tierType, uint256 mintPrice)
        internal
        pure
        returns (uint256 protocolFee, uint256 creatorTake)
    {
        uint256 feeBps = (tierType == TierType.Fan) ? 500 : 2500;
        protocolFee = (feeBps * mintPrice) / BPS_DENOMINATOR;
        creatorTake = mintPrice - protocolFee;
    }
}
```

Remediation

It is recommended to take these values from function parameters, which can only be controlled by the controller.

OS-ENT-SUG-02 | Checking Existence Before Returning

Description

In `Raises.sol`, the `getTiers` function is used to get the tiers list from a given `projectId` and `raiseId`. This function doesn't throw error if the project or raise does not exist.

Remediation

It is recommended to check the existence of project with given `projectId` and raise with given `raiseId`. Below is a code snippet showing the changes that could be made to the function:

src/Raises.sol DIFF

```diff
/// @inheritdoc IRaises
function getTiers(uint32 projectId, uint32 raiseId) external view override returns (Tier[] memory tier) {
+   // Check that project exists
+   if (totalRaises[projectId] == 0) revert NotFound();
+
+   // Get the raise if it exists
+   raise = raises[projectId][raiseId];
+   if (raise.projectId == 0) revert NotFound();
+
    return tiers[projectId][raiseId];
}
```

OS-ENT-SUG-03 | Enforcing Constraints On TierParams

Description

In `TierValidator.sol`, the `validate` function is used to check if the provided tier parameters satisfies some constraints. This function currently checks if the `tier.supply` value is greater than 0.

Remediation

The `validate` function should also enforce constraints on `tier.limitPerAddress` to check if it is greater than 0.

src/libraries/validators/TierValidator.sol DIFF

```
/// @title TierValidator - Tier parameter validator
library TierValidator {
    function validate(TierParams memory tier) internal pure {
        if (tier.supply == 0) {
            revert ValidationError("zero supply");
        }
+       if (tier.limitPerAddress == 0) {
+           revert ValidationError("zero limitPerAddr");
+       }
    }
}
```

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical

Vulnerabilities that immediately lead to loss of user funds with minimal preconditions

Examples:

- Misconfigured authority or access control validation
- Improperly designed economic incentives leading to loss of funds

High

Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions
- Exploitation involving high capital requirement with respect to payout

Medium

Vulnerabilities that could lead to denial of service scenarios or degraded usability.

Examples:

- Malicious input that causes computational limit exhaustion
- Forced exceptions in normal user flow

Low

Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions

Informational

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants
- Improved input validation