Perfect Abstractions

Xeenon Credit Ledger Audit

Table of Contents

1 Xeenon Credit Ledger Audit

Perfect Abstractions conducted a smart contract audit of GravitonINC/xeenon from Sep 30 2022 until Oct 5 2022. The audit is performed on commit GravitonINC/xeenon/tree/c349c5e9f7e27b0c5ddde17ceb3a5103bc0115f6.

The client has updated the repository after receiving the report in commit 9d05872.

Auditors:

- Kurt Willis

Audit report reviewed by Nick Mudge.

1.1 Overview

Xeenon is a protocol designed to support the transfer of credits between users for a content-creator platform. Most of the contract's functionality is handled by an automated backend given special access-control roles.

1.1.1 Objectives

- Uncover exploitable mechanics
- Validate protocol invariants
- Inspect protocol design choices
- Present optimizations

1.1.2 Scope

The following files are in the scope of the audit:

- AccessControl.sol
- Admin.sol
- CreditLedger.sol
- Xeenon.sol

I. High-Risk

2 Incorrect Credit Accounting for Withdrawals

High Risk

Fixed

Fixed as per recommendation.

In Xeenon.sol, protocol fees are not applied to user credits and incorrectly added as protocol revenue.

```
    function withdraw(uint256 _creditAmount) external {
        uint256 withdrawFee = feeComponents["withdraw"].value;

        // ...

        if (withdrawFee > 0) {
            _addRevenue(withdrawFee);
            _removeCredits(msg.sender, _creditAmount - withdrawFee);
            acceptedToken.safeTransfer(
                msg.sender,
                convertFromCredits(_creditAmount - withdrawFee)
            );
        }

        // ...
    }
```

The `withdrawFee` is deducted from the user's credit amount before being converted to the underlying asset and is added as protocol revenue. The user receives the underlying asset calculated by: `convertFromCredits(_creditAmount - withdrawFee)`. The amount deducted from the user's balance, however, is also `_creditAmount - withdrawFee`, whereas it should be the full `_creditAmount`. This means that the user is awarded the same amount in credits as is deducted from their balance without applying any fees. The non-applied fees are now incorrectly added as protocol revenue.

The consequence of this error is that the protocol's internal accounting will be incorrect. The problem is illustrated with an example scenario.

2.0.1 Example:

- Alice deposits 10$ which gets converted to 1000 credits (assuming no deposit fees)
- The protocol's withdrawal fees are set to a fixed amount of 5 credits
- Alice withdraws the 1000 credits and has 1000 - 5 = 995 credits deducted from her credit balance
- Alice has received 9.95$ and is left with 5 credits under her balance in the protocol
- The protocol has added 5 credits as protocol revenue
- The total amount of credits now is 995 + 5 + 5 = 1005 while it should only be 1000 credits

- Now either, if Alice withdraws more, the protocol fees will not be able to be withdrawn
- or, if the protocol fees are withdrawn first, Alice will not be able to withdraw the left-over credits

The impact/importance of this issue is considered high, as the mis-calculation is effective immediately and cannot be circumvented by the protocol as soon as withdrawal fees are enabled.

2.1 Recommendation

Remove the full amount `_creditAmount` from the user's balance, without subtracting the fees:

```
    function withdraw(uint256 _creditAmount) external {
        uint256 withdrawFee = feeComponents["withdraw"].value;

        // ...

        if (withdrawFee > 0) {
            _addRevenue(withdrawFee);
            _removeCredits(msg.sender, _creditAmount);
            acceptedToken.safeTransfer(
                msg.sender,
                convertFromCredits(_creditAmount - withdrawFee)
            );
        }

        // ...
    }
```

II. Medium-Risk

3 Credits Use Low Decimals

Medium Risk

> Client: "Business decision to keep 1 stable coin = 100 credits."

Due to the nature of the credits conversion, any credit transfer/deposit/withdrawal or fee calculation is performed with limited precision.

```
    function convertToCredits(uint256 _stableCoins)
        public
        pure
        returns (uint256)
    {
        return _stableCoins / creditsConversion;
    }
```

CreditLedger.sol defines `creditsConversion = 1e16`. An ERC20 token with 18 decimals will get divided by this conversion rate. Due to this, credits are effectively handled as tokens with 2 decimals (0.01) lending them a limited precision compared to native ERC20s.

This might be intentional as to result in a favorable fee rounding on withdrawals/deposits, yet this design choice could heavily limit the functioning of the protocol. As the system is designed to transfer credits between content producers and consumers, ensuring a fair payment to any user in the system could mean that sub-cent values are required.

A quick check on related streaming platforms shows payments of around 0.001-0.004$ per stream. Further, in this case, the restriction of a minimum of 1 credit per percentage fee could be a bad design choice, as this would mean a minimum multiple of the creator fee to be taxed.

```
    function _calcPercentageRateFee(uint256 _amount, uint256 _feePercentage)
        private
        pure
        returns (uint256, uint256)
    {
        // ...
        } else if (fee == 0) {
            // Xeenon will take a minimum fee of 1 credit for the transaction
            fee = 1;
        }
    }
```

Increasing the decimal precision should come at no additional computational costs while still allowing for favorable rounding mechanics. In fact, keeping it at the same conventional 18 decimals would remove the need for conversion, reducing computation and complexity.

3.1 Recommendation

Do not perform a conversion of credits by dividing the token amount by `1e16`. In this case, the backend of the protocol would need to be adjusted to count `1e16` credits (0.01$) as 1 credit compared to the current implementation.

III. Low-Risk

4 The Use of Percentage Rate is Misleading

Low Risk

> Client: "Keeping the percentage naming."

In the contract `Xeenon.sol` the word "percentage rate" is used. This is misleading because points per thousandths (per mille) are used.

```
    function _calcPercentageRateFee(uint256 _amount, uint256 _feePercentage) {
        // ...
        uint256 fee = (_amount * _feePercentage) / 1000;
    }
```

Furthermore, when adding a fee component, there is no indication of scale (i.e. 18 or 2 decimals, percentage or permillage).

```
    function addFeeComponent(
        string memory _key,
        uint128 _code,
        uint128 _value
    ) {
        // ...
    }
```

4.1 Recommendation

Rename the fee parameter of `_calcPercentageRateFee()` from `_feePercentage` to `_feePerMillage`.

Add natspec comments that further explain the parameters of functions. These comments can be utilized on block explorers, such as etherscan.

5 Percentage Fees Are Not Accounted For in Deposits/Withdrawals

Low Risk

Fixed

The client has implemented an update that prevents the fee component code of 'deposit' and 'withdraw' from being changed.

Deposits and withdrawals do not take percentage fees into account and operate under the assumption of always receiving fixed rate fees. The current contract does not have the ability to adapt for a change in the fee type.

```
    function deposit(uint256 _amount) external {
        uint256 depositFee = feeComponents["deposit"].value;
        // ...
        uint256 creditAmount = convertToCredits(_amount);
        require(creditAmount > depositFee, "Can't deposit less than fee.");
    }
```

If the deposit/withdraw fee component type is updated at some point in the future, this could have adverse effects as there are no safe-guards to prevent such a change.

5.1 Recommendation

Add code in the `addFeeComponent` function that prevents it from changing the fee type for "deposit" or "withdraw".

Or update `deposit()` and `withdraw()` functions to handle percentage fees (in a similar way that `_transfer` does).

IV. Informational

6 Risks of Centralization

Informational

> Client: "Actions to secure owner wallet will be taken."

The contract "owner" has maximum privileges. In particular, the owner is able to:

- set the treasury receiver `treasuryReceiver` (fee receiver) (as the owner)
- add/change fee components (as the owner)
- set the protocol admin `admin` (as the owner)
- grant the `TRANSFER` role (as the admin)
- transfer funds on behalf of the user (via `TRANSFER` role)
- withdraw funds to the `treasuryReceiver` (as the admin)

Furthermore, it is possible to set a fee component to a 100% fee.

```
    function _addPercentageRate(string memory _key, uint128 _value) private {
        require(_value <= 1000, "_value can't be more than 1000.");
        // ...
    }
```

The owner has the full power to access any funds of the protocol, including any user funds (by setting a 100% fee and invoking a transfer). A compromised owner wallet could lead to all of the user funds to be withheld by the smart contract or transferred out to a malicious actor.

The security of the protocol depends on the security of the owner address.

If the owner address does not have sufficient security it could be compromised in a number of ways. Here are some examples:

- Targeted phishing attacks
- Accidental leaks (e.g. by committing a private key)
- Hacks or a compromised computer with the private key
- A maliciously acting insider

6.1 Recommendation

If not already done, we recommend taking steps to secure the owner address and/or mitigate its power.

Here are some ways to secure the owner address or mitigate its power:

- Set the owner address to a multisignature contract that requires multiple approvals from different people.
- Use time-locks and/or on-chain governance for sensitive actions (withdrawals, setting a new admin/owner).

- Use reasonable limits on Fees for user fund protection.
- Replace unlimited credit usage by admin roles with limited user credit approvals through on-chain signature verification (see EIP-2612 permits).

7 Conform Events

Informational

Fixed

The events now track the credit receiver as `address to`.

7.1 `Transfer` event

The declared `Transfer` event tracks the `fee`, and credit balances `fromCreditsAfter` and `toCreditsAfter`, but does not indicate who is the credit receiver and what value is transferred.

```
event Transfer(
    address indexed from,
    uint256 fee,
    uint256 fromCreditsAfter,
    uint256 toCreditsAfter
);
```

7.1.1 Suggestion:

Add `address to`, the `credits` that are being transferred and the included `fee`.

```
event TransferCredits(
    address indexed from,
    address indexed to,
    uint256 credits,
    uint256 fee
);
```

Though the exact event should depend on the protocol's needs and backend design.

7.2 `Deposit` event

The `Deposit` event emits the value of `_amount` converted without fees. In its current form, the event does not give any information on how much credit was added and how high the fee was.

```
    _addCredits(msg.sender, creditAmount - depositFee);

    // ...

    emit Deposit(
        msg.sender,
        convertToCredits(_amount),
        creditBalance(msg.sender)
    );
```

7.2.1 Suggestion:

```
    event Deposit(
        address indexed from,
        uint256 credits,
        uint256 fee
    );
```

In this case, `depositFee` is deducted from `creditAmount` in the field `credits`.

```
    emit Deposit(
        msg.sender,
        creditAmount - depositFee,
        depositFee
    );
```

8 Declare Certain Variables Public

Informational

Fixed

The functions `getAccumulatedRevenue()` and `getTreasuryReceiver()` were added. Declaring these functions as `view` makes the data they return more difficult to access by addresses that are not the owner of the contract but the data can still be accesses by others because by design of Ethereum the data stored in contracts is publicly accessible.

8.0.1 CreditLedger.sol:

`totalRevenue` and `treasuryReceiver` in CreditLedger.sol are internal variables. Their values cannot be access easily outside the contract.

8.1 Recommendation

Declare `totalRevenue` and `treasuryReceiver` variables public or add getter functions for them.

```
    uint256 public totalRevenue;
    address public treasuryReceiver;
```

9 Emit `Frozen` event

Informational

Fixed

The client has added an event `WalletFreeze(address wallet)` to track individual wallets being frozen, but has decided to not emit events for a global and individual unfreeze.

`freezeWithdraw()` emits an event `GlobalFreeze()`. To be able to track any global unfreeze, `unFreezeWithdraw()` could also emit an event `GlobalUnFreeze()`.

```
    function freezeWithdraw() external onlyAdmin {
        withdrawFrozen = true;
        emit GlobalFreeze();
    }

    function unFreezeWithdraw() external onlyOwner {
        withdrawFrozen = false;
    }
```

9.1 Recommendation

```
    function unFreezeWithdraw() external onlyOwner {
        withdrawFrozen = false;
        emit GlobalUnFreeze(); // add this event
    }
```

Further, a `Freeze(address wallet)` and `UnFreeze(address wallet)` event could be emitted to track frozen and unfrozen user accounts.

```
    function _freezeWalletWithdraw(address _wallet) private {
        frozenWithdrawWallets[_wallet] = true;
    }

    function unFreezeWalletWithdraw(address _wallet) external onlyRole(FREEZE) {
        require(frozenWithdrawWallets[_wallet], "Wallet was not frozen.");
        delete frozenWithdrawWallets[_wallet];
    }
```

9.2 Recommendation

```
    function _freezeWalletWithdraw(address _wallet) private {
        frozenWithdrawWallets[_wallet] = true;
        emit Freeze(_wallet); // add this event
    }

    function unFreezeWalletWithdraw(address _wallet) external onlyRole(FREEZE) {
        require(frozenWithdrawWallets[_wallet], "Wallet was not frozen.");
        delete frozenWithdrawWallets[_wallet];
        emit UnFreeze(_wallet); // add this event
    }
```

10 Unable to Transfer Without Specifying A Fee Component

Informational

> Client: "Transfer events are made by the backend, if we make a mistake we don't want the transfer to go through without fees, this needs to be detected and actions should be taken. If we want a fee to be 0 then we can add a feeComponent with key: 'fee' that has a value of 0."

In order to successfully transfer credits, a fee component must be specified. Even when no fee is desired for a transfer, the second require statement means that a fee component must be included, even if it is 0, and cannot be bypassed.

```
    function _transfer(
        address _from,
        address _to,
        uint256 _creditAmount,
        string calldata _key
    ) private {
        require(creditBalance(_from) >= _creditAmount, "Not enough credits.");
        require(feeComponents[_key].code != 0, "Key does not exist.");
        // ...
    }
```

10.1 Recommendation

Allowing for more flexibility, the function could also be allowed to operate without a fee component. This could be achieved by removing the second require check, or by allowing a specific key (0x0000...) to signal that no fees are applied.

V. Optimization

11 AccessControl tracks duplicate state

Optimization

The client has added a function to get all members as an array via the function `function getRoleAccounts(bytes32 _role) external view onlyAdmin returns (address[] memory)`. Declaring this function as `view` makes the data they return more difficult to access by addresses that are not the owner of the contract but the data can still be accessed by others because by design of Ethereum the data stored in contracts is publicly accessible.

AccessControl currently keeps track of `members` twice (in the mappings `roles` and `members`).

```
    struct RoleData {
        mapping(address => bool) members;
    }
    mapping(bytes32 => RoleData) private roles;
    mapping(bytes32 => EnumerableSet.AddressSet) private members;
```

11.1 Recommendation

Simply tracking

```
    mapping(bytes32 => EnumerableSet.AddressSet) private members;
```

is sufficient.

Checking whether an `address` is contained in the `EnumerableSet.AddressSet` for a specific role can be checked via the library function `.contains`.

```
    function hasRole(bytes32 _role, address _account) public view virtual override returns (bool)
{
        return members[_role].contains(_account);
    }
```

Furthermore, a getter function could be written to access all enumerated values `members[_role].values()` for convenience.

12 Declare Non-Changing Variable `immutable`

Optimization

Fixed

Implemented according to recommendation.

```
    // Stable coin used
    IERC20 public acceptedToken;
```

`acceptedToken` is defined as a state variable and thus read from storage every time it is accessed. As there is no possibility of changing this variable, `acceptedToken` can be defined as `immutable`.

The variable then becomes part of the contract code and saves the cost of an `SLOAD` operation (2100 gas) when it is read.

12.1 Recommendation

Add the `immutable` key-word to the variable `acceptedToken`.

```
    // Stable coin used
    IERC20 public immutable acceptedToken;
```

13 Simplify Credit Rounding

Optimization

Fixed

Implemented according to recommendation.

In `_calcPercentageRateFee` credits are rounded up for a `.5` cent remainder and above.

```
    uint256 fee = (_amount * _feePercentage) / 1000;
    uint256 remainder = (_amount * _feePercentage) % 1000;

    // subcent rounding so its fair to Xeenon
    if (remainder > 499) {
        // If the remainder was .5 of a cent or greater
        fee = fee + 1;
    } else if (fee == 0) {
        // Xeenon will take a minimum fee of 1 credit for the transaction
        fee = 1;
    }
```

13.1 Recommendation

The code can be simplified to the following to allow for rounded-up values.

```
    uint256 fee = (_amount * _feePercentage + 500) / 1000;
    if (fee == 0) {
        fee = 1;
    }
```

14 `feeComponents` Key Could Be Declared `Bytes32`

Optimization

Fixed

Implemented according to recommendation.

Fee components are currently accessed via a key of type `string`.

```
mapping(string => FeeComponent) public feeComponents;
```

As solidity is tuned to operate on 32 bytes data, changing the key type from `string` to `bytes32` would save gas costs.

The gas savings are even more pronounced on layer-2 rollups that commit calldata to the layer-1 chain, such as Arbitrum. Here, over 50% of the transaction cost is made up of calldata cost. Changing from `string` to `bytes32` would additionally reduce the calldata encoding for a fee component key by a factor of 3: from 96 bytes (32 bytes location + 32 bytes length + 32 bytes data) to only 32 bytes.

14.1 Recommendation

The key type could be changed from `string` to `bytes32`.

```
mapping(bytes32 => FeeComponent) public feeComponents;
```

Solidity is able to automatically convert short string literals to `bytes32`, thus the changes to the current code would be minimal.

Further, `bulkTransfer(...)` requires fee components to be specified as an array.

To reduce calldata gas costs, a function overload could be defined, allowing only one fee component to be specified for a batched transfer transaction.

```
function bulkTransfer(
    address[] calldata _from,
    address[] calldata _to,
    uint256[] calldata _creditAmount,
    string calldata _key
) external onlyRole(TRANSFER) {
    for (uint256 i = 0; i < _from.length; i++) {
        _transfer(_from[i], _to[i], _creditAmount[i], _key);
    }
}
```

15 Disclaimer