



QuillAudits

Audit Report February, 2022

For



XEGGO
DROP

BETA

Contents

| | |
|---|----|
| Overview | 01 |
| Scope of Audit | 01 |
| Checked Vulnerabilities | 02 |
| Techniques and Methods | 03 |
| Issue Categories | 04 |
| Issues Found | 05 |
| High Severity Issues | 05 |
| Medium Severity Issues | 05 |
| Low Severity Issues | 05 |
| Informative Issues | 05 |
| 1. Inefficient use of balanceOf function that leads to... | 05 |
| Call Graphs | 07 |
| Functional Testing | 08 |
| Automated Testing | 09 |
| Closing Summary | 11 |

Overview

Xeggo Stream

A one-stop solution for handling streamed subscriptions for any services like salaries, rewards, or anything.

Scope of the Audit

The scope of this audit was to analyze the Xeggo Stream smart contract's codebase for quality, security, and correctness.

Date: **February 2, 2022 - February 5, 2022**

Commit: 6b9f2bab457ed593db83ad4fb803aafce18b644c

Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC20 transfer() does not return boolean
- ERC20 approve() race
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Slither, MythX, Truffle, Remix, Ganache, Solidity Metrics

Issue Categories

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

| Risk-level | Description |
|----------------------|---|
| High | A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment. |
| Medium | The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed. |
| Low | Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future. |
| Informational | These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact. |

Number of issues per severity

| Type | High | Medium | Low | Informational |
|---------------------|------|--------|-----|---------------|
| Open | 0 | 0 | 0 | 0 |
| Acknowledged | 0 | 0 | 0 | 1 |
| Closed | 0 | 0 | 0 | 0 |

Issues Found – Code Review / Manual Testing

High severity issues

No issues were found.

Medium severity issues

No issues were found.

Low severity issues

No issues were found.

Informational issues

1. Inefficient use of balanceOf function that leads to high gas usage.

[#L247][#L281][#L282] calling balanceOf function which internally reads stream struct again while the parent function already reading stream struct before calling the balanceOf function that leads to multiple loads which is unnecessary.

Recommendation

We recommend optimizing the function call. Instead of calling the public version of the balanceOf function creates an internal function of balanceOf function which takes Stream struct as the input param instead of reading it again.

```
function balanceOf(uint256 streamId, address who) public view streamExists(streamId)
returns (uint256 balance) {
    Types.Stream memory stream = streams[streamId];
    return _balanceOf(streamId, who, stream);
}
```


While `_balanceOf()` definition will be something like this.

```
function _balanceOf(uint256 streamId, address who, Types.Stream memory stream)
internal view streamExists(streamId) returns (uint256 balance) {
    BalanceOfLocalVars memory vars;

    uint256 delta = deltaOf(streamId);
    (vars.mathErr, vars.recipientBalance) = mulUInt(delta, stream.ratePerSecond);
    require(vars.mathErr == MathError.NO_ERROR, "recipient balance calculation error");

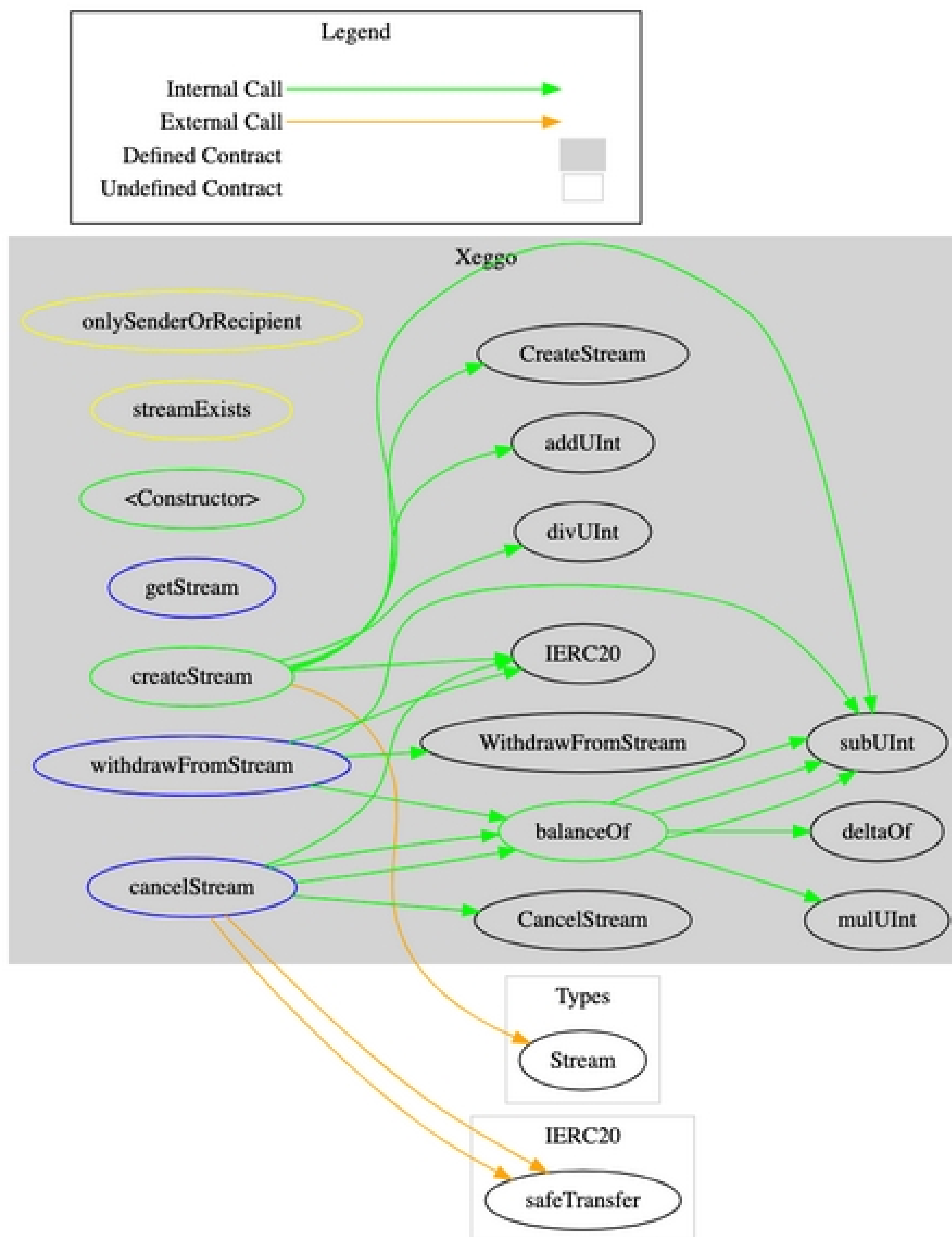
    /*
     * If the stream `balance` does not equal `deposit`, it means there have been
    withdrawals.
     * We have to subtract the total amount withdrawn from the amount of money that
    has been
     * streamed until now.
     */
    if (stream.deposit > stream.remainingBalance) {
        (vars.mathErr, vars.withdrawalAmount) = subUInt(stream.deposit,
stream.remainingBalance);
        assert(vars.mathErr == MathError.NO_ERROR);
        (vars.mathErr, vars.recipientBalance) = subUInt(vars.recipientBalance,
vars.withdrawalAmount);
        /* `withdrawalAmount` cannot and should not be bigger than `recipientBalance`.
    */
        assert(vars.mathErr == MathError.NO_ERROR);
    }

    if (who == stream.recipient) return vars.recipientBalance;
    if (who == stream.sender) {
        (vars.mathErr, vars.senderBalance) = subUInt(stream.remainingBalance,
vars.recipientBalance);
        /* `recipientBalance` cannot and should not be bigger than `remainingBalance`. */
        assert(vars.mathErr == MathError.NO_ERROR);
        return vars.senderBalance;
    }
    return 0;
}
```

And use `_balanceOf()` at [#L247][#L281][#L282] instead of `balanceOf`.

Status: Acknowledged

Call Graphs



Functional Testing Results

Complete functional testing report has been attached below:

[Xeggo test case](#)

- ☒ should be able to create a stream.
- ☒ should be able to withdraw from the stream by the sender.
- ☒ should be able to withdraw from the stream by the recipient.
- ☒ should be able to cancel the stream by the sender.
- ☒ should be able to cancel the stream by the recipient.
- ☒ should be able to correctly get the balance using balanceOf.
- ☒ should be able to retrieve the stream details using the getStream
- ☒ should be able to retrieve the delta using the deltaOf.

Automated Testing

Slither Analysis Results

SLITHER

ANALYSIS

High (0)

Medium (16)

CarefulMath.divUInt(uint256,uint256) uses a dangerous strict equality:

CarefulMath.mulUInt(uint256,uint256) uses a dangerous strict equality:

Xeggo.balanceOf(uint256,address) uses a dangerous strict equality:

Xeggo.balanceOf(uint256,address) uses a dangerous strict equality:

Xeggo.balanceOf(uint256,address) uses a dangerous strict equality:

Xeggo.balanceOf(uint256,address) uses a dangerous strict equality:

Xeggo.balanceOf(uint256,address) uses a dangerous strict equality:

Xeggo.balanceOf(uint256,address) uses a dangerous strict equality:

Xeggo.balanceOf(uint256,address).vars is a local variable never initialized

Xeggo.createStream(address,uint256,address,uint256,uint256) uses a dangerous strict equality:

Xeggo.createStream(address,uint256,address,uint256,uint256) uses a dangerous strict equality:

Xeggo.createStream(address,uint256,address,uint256,uint256) uses a dangerous strict equality:

Xeggo.createStream(address,uint256,address,uint256,uint256) uses a dangerous strict equality:

Xeggo.createStream(address,uint256,address,uint256,uint256).vars is a local variable never initialized

Xeggo.onlySenderOrRecipient(uint256) uses a dangerous strict equality:

Xeggo.withdrawFromStream(uint256,uint256) uses a dangerous strict equality:

Low (9)

Modifier Migrations.restricted() does not always execute _; or revert

Reentrancy in Xeggo.cancelStream(uint256):

Reentrancy in Xeggo.createStream(address,uint256,address,uint256,uint256):

Reentrancy in Xeggo.withdrawFromStream(uint256,uint256):

Xeggo.balanceOf(uint256,address) uses timestamp for comparisons

Xeggo.cancelStream(uint256) uses timestamp for comparisons

Xeggo.createStream(address,uint256,address,uint256,uint256) uses timestamp for comparisons

Xeggo.deltaOf(uint256) uses timestamp for comparisons

Xeggo.withdrawFromStream(uint256,uint256) uses timestamp for comparisons

Informational (13)

Address.isContract(address) uses assembly

CarefulMath.addThenSubUInt(uint256,uint256,uint256) is never used and should be removed

Different versions of Solidity is used:

Low level call in Address.sendValue(address,uint256):

Low level call in SafeERC20.callOptionalReturn(IERC20,bytes):

Parameter Migrations.setCompleted(uint256)._completed is not in mixedCase

Parameter Migrations.upgrade(address)._newAddress is not in mixedCase

Pragma version^0.5.0 allows old versions

Pragma version^0.5.0 allows old versions

Pragma version^0.5.0 allows old versions

Pragma version^0.5.0 allows old versions

Pragma version^0.5.5 is known to contain severe issues (<https://solidity.readthedocs.io/en/latest/bugs.html>)

Xeggo.constructor() uses literals with too many digits:

audits.quillhash.com

09

MythX Analysis Results

Issues

These were the issues detected in the scan. You can use the toggles to filter by severity and display or hide ignored issues.

Severity

Low (1)

Medium (0)

High (0)

Ignored Issues

Hidden (0)

| ID | Severity | Name | File | Location |
|-------------------------|----------|--|-----------------------------|---------------|
| SWC-116 | Low | A control flow decision is made based on The block.timestamp environment variable. | Address.sol | L: 71 C: 4276 |

No issues as issue above is in a dependency and not relevant.

Closing Summary

One issue of Informational severity was found, which has been Acknowledged by the Auditee.



Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the Xeggo. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the Xeggo team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

Audit Report February, 2022

For



QuillAudits

📍 Canada, India, Singapore, United Kingdom

🌐 audits.quillhash.com

✉️ audits@quillhash.com