Aurigami Finance Audit

February 28, 2022

Table of Contents

Summary

This report has been prepared for **Aurigami Finance** smart contracts, to discover issues and vulnerabilities in the source code of their Smart Contract as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

Overview

Project Summary

Project Name	**Aurigami Finance**
Codebase	**https://github.com/Aurigami-Finance/aurigami-contracts**
Commit	**134eeea0970033bb487791075cdc9a8b75d55b26**
Language	**Solidity**
Platform	**Aurora**

Audit Summary

Delivery Date	**Feb 28, 2022**
Audit Methodology	**Static Analysis, Manual Review**
Total Isssues	**8**

AF-1: Misleading comments

Informational

1. contracts/AuErc20.sol#L138-L171

```
/**
 * @dev Similar to EIP20 transfer, except it handles a False result from `transferFrom` and reverts in that case.
 *      This will revert due to insufficient balance or insufficient allowance.
 *      This function returns the actual amount received,
 *      which may be less than `amount` if there is a fee attached to the transfer.
 *
 *      Note: This wrapper safely handles non-standard ERC-20 tokens that do not return a value.
 *            See here: https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521ca
 */
function doTransferIn(address from, uint amount) internal override returns (uint) {
    EIP20NonStandardInterface token = EIP20NonStandardInterface(underlying);
    token.transferFrom(from, address(this), amount);

    bool success;
    assembly {
        switch returndatasize()
            case 0 {                      // This is a non-standard ERC-20
                success := not(0)         // set success to true
            }
            case 32 {                     // This is a compliant ERC-20
                returndatacopy(0, 0, 32)
                success := mload(0)       // Set `success = returndata` of external call
            }
            default {                     // This is an excessively non-compliant ERC-20, revert.
                revert(0, 0)
            }
    }
    require(success, "TOKEN_TRANSFER_IN_FAILED");

    // Skip the check for fee on transfer tokens

    // Calculate the amount that was *actually* transferred
    return amount;    // underflow already checked above, just subtract
}
```

The support of tokens with "fee on transfer" was removed at commit db2cc8f4acb05fa96f9b19c767f72e59c3db4c10, but the comment at L170 remain unchanged.

2. contracts/AuToken.sol#L302-L307

```
/**
     * @notice Calculates the exchange rate from the underlying to the AuToken
     * @dev This function does not accrue interest before calculating the exchange rate
     * @return (error code, calculated exchange rate scaled by 1e18)
     */
    function exchangeRateStoredInternal() internal view returns (uint) {
```

`error code` is no longer returned.
More instances (about 10 results) can be found by searching `error code` in the codebase.

Status

✓ Fixed in commit: 9d83a46fd64034518600101cbff13f7c8d39dec3.

AF-2: Redundant check of msg.sender != address(0)

Informational

msg.sender wont be address(0).

contracts/AuToken.sol#L941-L957

```
function _acceptAdmin() external override{
     // Check caller is pendingAdmin and pendingAdmin ≠ address(0)
     require(!(msg.sender != pendingAdmin || msg.sender == address(0)), "unauthorized");
     // ...
}
```

Can be changed to:

```
require(msg.sender == pendingAdmin, "unauthorized");
```

Similarly, Unitroller._acceptAdmin():

contracts/Unitroller.sol#L100-L104

```
function _acceptAdmin() public {
        // Check caller is pendingAdmin and pendingAdmin ≠ address(0)
      if (msg.sender != pendingAdmin || msg.sender == address(0)) {
          revert Unauthorized();
      }
```

Can be changed to:

```
function _acceptAdmin() public {
  if (msg.sender != pendingAdmin) {
    revert Unauthorized();
  }
  ...
```

Status

✓ Fixed in commit: 9d83a46fd64034518600101cbff13f7c8d39dec3.

AF-3: AuToken.sol#nonReentrant Switching between 1, 2 instead of true, false is more gas efficient

Informational

contracts/AuToken.sol#L1205-L1210

```
modifier nonReentrant() {
    require(_notEntered, "re-entered");
    _notEntered = false;
    _;
    _notEntered = true; // get a gas-refund post-Istanbul
}
```

The current implementation of nonReentrant is switching between true and false.

SSTORE from false (0) to true (1) (or any non-zero value), the cost is 20000; SSTORE from 1 to 2 (or any other non-zero value), the cost is 5000.

By storing the original value once again, a refund is triggered (https://eips.ethereum.org/EIPS/eip-2200).

Since refunds are capped to a percentage of the total transaction's gas, it is best to keep them low, to increase the likelihood of the full refund coming into effect.

Therefore, switching between 1, 2 instead of false (0), true (1) will be more gas efficient.

See: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/86bd4d73896afcb35a205456e361436701823c7a/contracts/security/ReentrancyGuard.sol#L29-L33

Recommendation

Consider using OpenZeppelin's security/ReentrancyGuard.sol.

Status

✓ Fixed in commit: 8e71e2794d7eebed5c741c77a1e4d82135bab4ba.

AF-4: AuriLens#getAccountLimits() will always revert due to incorrect ABI of comptroller.getAccountLiquidity()

Informational

AuriLens.sol is out of scope, but the issue might need your attention.

contracts/AuriLens.sol#L271-L287

```
function getAccountLimits(ComptrollerLensInterface comptroller, address account)
  public
  view
  returns (AccountLimits memory)
{
  (uint256 errorCode, uint256 liquidity, uint256 shortfall) = comptroller.getAccountLiquidity(
    account
  );
  require(errorCode == 0);

  return
    AccountLimits({
      markets: comptroller.getAssetsIn(account),
      liquidity: liquidity,
      shortfall: shortfall
    });
}
```

L276 expects comptroller.getAccountLiquidity() to return (uint256, uint256, uint256).

The implementation of comptroller.getAccountLiquidity() will actually return (uint256, uint256).

/contracts/Comptroller.sol#L411-L413

```
function getAccountLiquidity(address account) public view returns (uint, uint) {
    return getHypotheticalAccountLiquidityInternal(account, AuToken(address(0)), 0, 0);
}
```

That's because the first return of comptroller.getAccountLiquidity(), the error code, was removed in commit 15eebca46862fd481dcfde933d8cb3f186df069e.

The interface of ComptrollerLensInterface.getAccountLiquidity() is defined incorrectly.

contracts/AuriLens.sol#L10-L43

```
interface ComptrollerLensInterface {
  // ...
  function getAccountLiquidity(address)
    external
    view
    returns (
      uint256,
      uint256,
      uint256
    );
  // ...
}
```

Status

✓ Fixed in commit: afbefddc57c826b609470110048dc21140c9fe70.

AF-5: Redundant code

Informational

contracts/Comptroller.sol#L944-L946

```
function claimReward(uint8 rewardType, address payable holder) public {
    return claimReward(rewardType, holder, allMarkets);
}
```

claimReward() has no returns, therefore the return is redundant.

Status

✓ **Fixed** in commit: 8e71e2794d7eebed5c741c77a1e4d82135bab4ba.

AF-6: Unconventional implementation of SafeCast

Low

contracts/AuriMathLib.sol#L63-L71

The max value of the target type is usually allowed in SafeCast.

```
function safe216(uint256 n) internal pure returns (uint216) {
  require(n < type(uint216).max, "safe216");
  return uint216(n);
}

function safe32(uint256 n) internal pure returns (uint32) {
  require(n < type(uint32).max, "safe32");
  return uint32(n);
}
```

Consider changing to:

```
function safe216(uint256 n) internal pure returns (uint216) {
  require(n <= type(uint216).max, "safe216");
  return uint216(n);
}

function safe32(uint256 n) internal pure returns (uint32) {
  require(n <= type(uint32).max, "safe32");
  return uint32(n);
}
```

See: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.5.0/contracts/utils/math/SafeCast.sol

Status

✓ **Fixed** in commit: 8e71e2794d7eebed5c741c77a1e4d82135bab4ba.

AF-7: transfer() is not recommended for sending native token

Informational

Since the introduction of transfer(), it has typically been recommended by the security community because it helps guard against reentrancy attacks. This guidance made sense under the assumption that gas costs wouldn't change. It's now recommended that transfer() and send() be avoided, as gas costs can and will change and reentrancy guard is more commonly used.

Any smart contract that uses transfer() is taking a hard dependency on gas costs by forwarding a fixed amount of gas: 2300.

It's recommended to stop using transfer() and switch to using call() instead.

contracts/AuETH.sol#L137-L140

```
function doTransferOut(address payable to, uint256 amount) internal override {
  /* Send the Ether, with minimal gas and revert on failure */
  to.transfer(amount);
}
```

Consider changing to:

```
function doTransferOut(address payable recipient, uint256 amount) internal override {
  /* Send the native token, with revert on failure */
  (bool success, ) = recipient.call{value: amount}("");
  require(success, "native token transfer failed");
}
```

Or, consider using OpenZeppelin's Address#sendValue().

See: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/3e74681e779e74391d4116dedfcd77aadfe1579c/contracts/utils/Address.sol#L44-L65

Status

ⓘ Acknowledged

AF-8: AuriOracle.sol network congestion may disrupt price feeds

High

Per the document, the AuriOracle.sol contract is:

> our custom oracle implementation...
>
> Aurigami will operate our own price feeds:
>
> There are 3 bots, holding updator1, updator2 and updator3 keys
>
> Each bot will get prices from Chainlink oracles on Ethereum and call updateMainFeedData() to update their feeds.

contracts/AuriOracle.sol#L131-L150

```
function updateMainFeedData(address auToken, int256 newUnderlyingPrice) external onlyUpdator {
    require(newUnderlyingPrice > 0, "bad price");
    // safe uint256 cast since answer > 0
    mainFeedRaw[msg.sender][auToken] = RawData(
      Math.safe216(uint256(newUnderlyingPrice)),
      Math.safe32(block.timestamp)
    );
    ...
```

The implementation only takes two parameters: auToken and newUnderlyingPrice. And the updatedAt of the record will always be set to the current block.timestamp.

This makes it possible for the price feeds to be disrupted when the network is congested and transactions with stale prices get accepted as fresh prices.

Since the price feeds are essential to the protocol, that can result in users' positions being liquidated wrongfully and case fund loss to users.

PoC

Given:

- updator1 and updator2 are connected to an RPC endpoint currently experiencing degraded performance;
- backupFeedAddr is working properly;
- Bitcoin price is $100,000;
- The collateralFactor of Bitcoin is 60%.

1. Alice borrowed `50,000 USDC` with `1 BTC` as collateral;
2. Bitcoin price dropped to `$90,000`, to avoid liquidation, Alice repaid `10,000 USD`;
3. The price of Bitcoin dropped to `$80,000`; `updator1` and `updator2` tries to `updateMainFeedData()` with the latest price: `$80,000`, however, since the network is congested, the transactions were not get packed timely;
4. Bitcoin price rebound to `$100,000`; Alice borrowed another `10,000 USDC`;
5. The txs send by `updator1` and `updator2` at step 3 finally got packed, the protocol now believes the price of Bitcoin has suddenly dropped to `$80,000`, as a result, Alice's position got liquidated.

Recommendation

Change to:

```
function updateMainFeedData(address auToken, int256 newUnderlyingPrice, uint256 newUpdatedAt) external onlyUpdator {
  require(newUnderlyingPrice > 0, "bad price");
  if (block.timestamp > newUpdatedAt) {
      // reject stale price
      require(block.timestamp - newUpdatedAt < validPeriod, "bad updatedAt");
  } else {
      // reject future timestamp (< 3s is allowed)
      require(newUpdatedAt - block.timestamp < 3, "bad updatedAt");
      newUpdatedAt = block.timestamp;
  }

  // safe uint256 cast since answer > 0
  mainFeedRaw[msg.sender][auToken] = RawData(
    Math.safe216(uint256(newUnderlyingPrice)),
    Math.safe32(newUpdatedAt)
  );

  (uint256 answer, uint256 updatedAt) = aggregateAllRawData(auToken);

  if (updatedAt != 0) {
    // updatedAt == 0 only if 2 out of 3 feeds are outdated
    // answer != 0 since updatedAt != 0
    mainFeed[auToken].answer = Math.safe216(answer);
    mainFeed[auToken].updatedAt = Math.safe32(updatedAt);
    emit MainFeedSync(auToken, uint256(answer), msg.sender);
  } else {
    emit MainFeedFail(auToken, msg.sender);
  }
}
```

Status

✓ Fixed in commit: 100be77584e833af1239cbf83dc173c772dc02c6.

Appendix

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by WatchPug; however, WatchPug does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication.

Disclaimer

This report is based on the scope of materials and documentation provided for a limited review at the time provided. Results may not be complete nor inclusive of all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Smart Contract technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. A report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, we disclaim all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.