# Security Audit Report

## fxUSD

## by AladdinDAO

SECBIT

**February 23, 2024**

# 1. Introduction

The AladdinDAO is a decentralized network to shift crypto investments from venture capitalists to the wisdom of crowds through collective value discovery. As a part of the AladdinDAO ecosystem, the f(x) protocol creates two new ETH derivative assets, one with stablecoin-like low volatility called fractional ETH (fETH) and the second a leveraged long ETH perpetual token called leveraged ETH (xETH). Building upon the legacy f(x) protocol, the AladdinDAO team introduced a more competitive stablecoin, fxUSD, which boasts advantages such as Built-in yield, Scalability, Stability, and Resilience. SECBIT Labs conducted an audit from January 24 to February 23, 2024, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the fxUSD protocol has no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising (see part 4 for details).

| Type | Description | Level | Status |
|---|---|---|---|
| Design & Implementation | 4.3.1 Lack of validation on some crucial input parameters in FxUSD.sol. | Medium | Fixed |
| Design & Implementation | 4.3.2 Discussion on the redemption logic of the `redeem()` function. | Info | Discussed |
| Design & Implementation | 4.3.3 Lack of validation on some crucial input parameters in FxUSDFacet.sol. | Info | Discussed |
| Design & Implementation | 4.3.4 The return value of the `get_gauge_weight()` function may lag behind the actual value. | Medium | Fixed |
| Design & Implementation | 4.3.5 When modifying the weight of a gauge through the `change_gauge_weight()` function, the impact of type weight is not considered. | Low | Fixed |
| Design & Implementation | 4.3.6 Discussion on the logic of functions `fxMintFTokenV2()`, `fxRedeemFTokenV2()`, and `fxSwapV2()`. | Info | Discussed |
| Design & Implementation | 4.3.7 The admin should avoid the expected total weight sum of all gauges within the whitelist being 1e18. | Info | Discussed |

# 2. Contract Information

This part describes the basic contract information and code structure.

## 2.1 Basic Information

The basic information about the fxUSD protocol is shown below:

- Smart contract code
  - initial review commit
    - 6c00393
  - final review commit
    - 6fd526c

## 2.2 Contract List

The following content shows the contracts included in the fxUSD protocol, which the SECBIT team audits:

| Name | Lines | Description |
| --- | --- | --- |
| FxStableMath.sol | 97 | The core code for minting and redeeming fTokens and xTokens. |
| FxFrxETHTwapOracle.sol | 60 | Auxiliary contract to provide the price of frxETH |
| FxUSDShareableRebalancePool.sol & ShareableRebalancePoolV2.sol & ShareableRebalancePool.sol | 603 | Support the sharing of veFXN tokens held by the owner to enhance the earnings of stakers. |
| FxUSDRebalancer.sol | 53 | Acting as the liquidator, centrally managing all rebalance pools. |
| ReservePoolV2.sol | 69 | The auxiliary contract for additional allowances issued for the protocol. |
| FractionalTokenV2.sol | 36 | Responsible for the standard ERC20 contract for minting and burning fTokens. |
| FxInitialFund.sol | 84 | The contract for raising startup capital for the fxUSD protocol. |
| FxUSD.sol | 318 | The core contract of the fxUSD protocol, through which users acquire fxUSD tokens. |
| LeveragedTokenV2.sol | 43 | Responsible for the standard ERC20 contract for minting and burning xTokens. |
| MarketV2.sol | 422 | The core contract for minting and redeeming fTokens and xTokens. |
| TreasuryV2.sol | 393 | A contract to store the baseToken, where the core functions can only be called by the Market contract. |
| WrappedTokenTreasuryV2.sol | 53 | Harvest pending rewards to stability pool. |
| FxMarketV1Facet.sol | 104 | The peripheral contract for the old protocol, facilitating users to interact with the protocol using a wider variety of tokens. |
| FxUSDFacet.sol | 230 | The peripheral contract for the current protocol facilitates users interacting with the protocol using a wider variety of tokens. |
| TokenConvertManagementFacet.sol & LibGatewayRouter.sol | 147 | The routing contract used for token exchanges. |
| GaugeControllerOwner.sol | 113 | The contract automatically adjusts the weights of gauges within the whitelist. |

# 3. Contract Analysis

This part describes code assessment details, including "role classification" and "functional analysis".

## 3.1 Role Classification

Two key roles in the fxUSD protocol are the Governance Account and the Common Account.

- Governance Account
    - Description

      Contract Administrator
    - Authority
        - Update protocol parameter
        - Transfer ownership
        - Add / Remove rebalance pool

        - Add / Remove market
        - Update gauges weight
    - Method of Authorization

      The contract administrator is the contract's creator or authorized by transferring the governance account.
- Common Account
    - Description

      Mint fxUSD token and xToken by utilizing authorized base token
    - Authority
        - Mint / Burn fxUSD token
        - Mint / Burn xToken
        - Deposit fToken to rebalance pool
    - Method of Authorization

      No authorization required

### 3.2 Functional Analysis

The f(x) protocol implements a decentralized quasi-stablecoin with high collateral utilization efficiency and leveraged contracts with low liquidation risks and no funding costs. In contrast to the earlier version, the current iteration of the protocol substitutes fToken with the fxUSD token. As a liquidity incentive platform for LSDs, fxUSD presents a cost-effective and efficient method to incentivize token holders and foster on-chain utility and liquidity. The SECBIT team conducted a detailed audit of some of the contracts in the protocol. We can divide the critical functions of the contract into three parts:

#### FxUSD

Users acquiring fxUSD tokens must stake an equivalent amount of fTokens in the contract. Holding fxUSD tokens grants users greater liquidity compared to fTokens. The main functions in this contract are as follows:

- `wrap()`

  Anyone could call this function to wrap fToken to fxUSD token.
- `wrapFrom()`

  Users could withdraw fTokens from the rebalance pool and exchange them for fxUSD token.

- `mint()`

  Users could mint the fxUSD token with the base token.

- `earn()`

  This function exchanges fxUSD token for fTokens and then deposits them into the corresponding rebalance pool.

- `mintAndEarn()`

  Users can mint fTokens and deposit them directly into the rebalance pool.

- `redeem()`

  Users redeem base tokens through this function.

**FxUSDFacet**

As an auxiliary contract of the fxUSD protocol, this contract supports users in participating using a wider range of mainstream tokens. The main functions in this contract are as follows:

- `fxInitialFundDeposit()`

  Deposit to initial fund with given token and convert parameters.

- `fxMintFTokenV2()`

  Mint some fToken with given token and convert parameters.

- `fxMintXTokenV2()`

  Mint some xToken with given token and convert parameters.

- `fxRedeemFTokenV2()`

  Redeem fToken and convert to some other token.

- `fxRedeemXTokenV2()`

  Redeem xToken and convert to some other token.

- `fxSwapV2()`

  Swap between fToken and xToken.

- `fxSwapFxUSD()`

  Swap between fxUSD token and xToken.

- `fxRebalancePoolWithdraw()`

  Withdraw fToken from rebalance pool as fxUSD token.

- `fxRebalancePoolWithdrawAs()`

  Withdraw fToken from the rebalance pool as the target token.

- `fxMintFxUSD()`

Mint some fxUSD tokens with the given token and convert parameters.

- `fxMintFxUSDAndEarn()`

  Mint some fxUSD tokens and earn in the rebalance pool with the given token and convert parameters.

- `fxRedeemFxUSD()`

  Redeem fxUSD tokens and convert them to some other token.

- `fxAutoRedeemFxUSD()`

  Redeem fxUSD tokens and convert them to some other token.

- `fxBaseTokenSwap()`

  Swap base through fxUSD tokens.

**GaugeControllerOwner**

Due to the inability of the GaugeController contract to set the weights of each gauge to a fixed ratio, the development team has designed this contract. Administrators can dynamically adjust the weights of each gauge within the whitelist using this contract, ensuring they are maintained at a fixed ratio. The main function of this contract is as follows:

- `normalizeGaugeWeight()`

  Normalize the whitelisted gauge to the correct weight.

# 4. Audit Detail

This part describes the process, and the detailed audit results also demonstrate the problems and potential risks.

## 4.1 Audit Process

The audit strictly followed the audit specification of SECBIT Lab. We analyzed the project from code bugs, logical implementation, and potential risks. The process consists of four steps:

- Fully analysis of contract code line by line.
- Evaluation of vulnerabilities and potential risks revealed in the contract code.
- Communication on assessment and confirmation.
- Audit report writing.

### 4.2 Audit Result

After scanning with adelaide, sf-checker, and badmsg.sender (internal version) developed by SECBIT Labs and open source tools, including Mythril, Slither, SmartCheck, and Securify, the auditing team performed a manual assessment. The team inspected the contract line by line, and the result could be categorized into the following types:

| Number | Classification | Result |
|---|---|---|
| 1 | Normal functioning of features defined by the contract | ✓ |
| 2 | No obvious bug (e.g., overflow, underflow) | ✓ |
| 3 | Pass Solidity compiler check with no potential error | ✓ |
| 4 | Pass common tools check with no obvious vulnerability | ✓ |
| 5 | No obvious gas-consuming operation | ✓ |
| 6 | Meet with ERC20 standard | ✓ |
| 7 | No risk in low-level call (call, delegatecall, callcode) and in-line assembly | ✓ |
| 8 | No deprecated or outdated usage | ✓ |
| 9 | Explicit implementation, visibility, variable type, and Solidity version number | ✓ |
| 10 | No redundant code | ✓ |
| 11 | No potential risk manipulated by timestamp and network environment | ✓ |
| 12 | Explicit business logic | ✓ |
| 13 | Implementation consistent with annotation and other info | ✓ |
| 14 | No hidden code about any logic that is not mentioned in the design | ✓ |
| 15 | No ambiguous logic | ✓ |
| 16 | No risk threatening the developing team | ✓ |
| 17 | No risk threatening exchanges, wallets, and DApps | ✓ |

| | | |
|---|---|---|
| 18 | No risk threatening token holders | ✓ |
| 19 | No privilege on managing others' balances | ✓ |
| 20 | No non-essential minting method | ✓ |
| 21 | Correct managing hierarchy | ✓ |

## 4.3 Issues

### 4.3.1 Lack of validation on some crucial input parameters in FxUSD.sol

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Medium | Design logic | Fixed |

**Location**

FxUSD.sol#L160

FxUSD.sol#L177

**Description**

According to normal code design logic, users can convert fxUSD tokens to specified fTokens via the `earn()` function and deposit them into the corresponding rebalance pool contract to earn profits. Consider the following scenario: Most users interact with the contract through a frontend webpage; they may not carefully verify whether the `_pool` contract address provided by the website is indeed the address of a genuine rebalance pool contract. In such a scenario, if an attacker maliciously substitutes the `_pool` contract address provided by the frontend, the user's fTokens will be sent to a forged `_pool` contract address, leading to the loss of funds.

Due to the contract's failure to verify whether the `_pool` parameter is a legit rebalance pool address, attackers can forge this address. Below is a sample attack contract for reference:

```
contract FakeRebalancePool {

  //@audit genuine baseToken of rebalancePool
  address public originalBaseToken;
```

```
  address public fToken;

  //@audit set the genuine basetoken and fToken
  function setBaseToken(address _originalBaseToken, address _fToken)
public {
    originalBaseToken = _originalBaseToken;
    fToken = _fToken;
  }

  function baseToken() public returns (address) {
    return originalBaseToken;
  }

  function deposit(uint256 _amount, address _receiver) public
returns(bool) {
    IERC20Upgradeable(fToken).safeTransferFrom(_msgSender(),
address(this), _amount);
    return true;
  }

  // @audit withdraw fToken from this contract
  function sendfTokenToAttacker(address _attacker) public {
    uint256 _amount = IERC20Upgradeable(fToken).balanceOf(address(this));
    IERC20Upgradeable(fToken).safeTransfer(_attacker,_amount);
  }
}
```

In this scenario, users' fTokens could be stolen by attackers. Similarly, the parameter `_pool` in the `mintAndEarn()` function also faces the same issue. Furthermore, from a protocol security perspective, it is highly dangerous not to perform verification on the `_pool` parameter. When the rebalance pool cannot receive users' fTokens under the above scenario, which would flow out of the protocol, violating the original design intent and compromising the protocol's integrity.

```
function earn(
  address _pool,  // @audit attackers could forge this parameter
  uint256 _amount,
  address _receiver
) external override {
  if (isUnderCollateral()) revert ErrorUnderCollateral();

  address _baseToken = IFxShareableRebalancePool(_pool).baseToken();
  _checkBaseToken(_baseToken);
```

```
    _burnShares(_baseToken, _msgSender(), _amount);
    emit Unwrap(_baseToken, _msgSender(), _receiver, _amount);

    _deposit(markets[_baseToken].fToken, _pool, _receiver, _amount);
  }

  function mintAndEarn(
    address _pool,  //@audit attackers could forge this parameter
    uint256 _amountIn,
    address _receiver,
    uint256 _minOut
  ) external override returns (uint256 _amountOut) {
    if (isUnderCollateral()) revert ErrorUnderCollateral();

    address _baseToken = IFxShareableRebalancePool(_pool).baseToken();
    _checkBaseToken(_baseToken);
    _checkMarketMintable(_baseToken);

    address _fToken = markets[_baseToken].fToken;
    _amountOut = _mintFToken(_baseToken, _fToken, _amountIn, _minOut);
    _deposit(_fToken, _pool, _receiver, _amountOut);
  }

  function _deposit(
    address _fToken,
    address _pool,
    address _receiver,
    uint256 _amount
  ) internal {
    IERC20Upgradeable(_fToken).safeApprove(_pool, 0);
    IERC20Upgradeable(_fToken).safeApprove(_pool, _amount);
    IFxShareableRebalancePool(_pool).deposit(_amount, _receiver);
  }
```

**Suggestion**

At the contract level, it's crucial to validate the correctness of the parameter `_pool` to prevent the input of fake rebalance pool contract addresses.

**Status**

The development team addressed this issue by adding validation for the `_pool` parameter in commit cb83a2c, restricting users to utilize only pools listed in the whitelist `supportedPools`.

### 4.3.2 Discussion on the redemption logic of the `redeem()` function.

| Risk Type | Risk Level | Impact | Status |
| --- | --- | --- | --- |
| Design & Implementation | Info | Design logic | Discussed |

#### Location

FxUSD.sol#L206

#### Description

Users can exchange fxUSD tokens for specified baseToken via the `redeem()` function. The parameter `_amountIn` represents the quantity of fxUSD tokens (fTokens) the user wishes to burn. When this parameter's value exceeds the number of fTokens held in the contract, calling `_market.redeemFToken()` will fail. Since fxUSD tokens are composed of multiple fTokens, such as fTokenA, fTokenB, and fTokenC, the quantity of fxUSD tokens will always be greater than the quantity of a particular fToken (e.g., fTokenA) held in the contract. It could lead to insufficient fTokenA in the contract, failing the `redeem()` function call. In such a scenario, should users be allowed to exchange the remaining portion of fTokens held in the contract for baseToken?

```
function redeem(
  address _baseToken,
  uint256 _amountIn,
  address _receiver,
  uint256 _minOut
) external override onlySupportedMarket(_baseToken) returns (uint256
_amountOut, uint256 _bonusOut) {
  if (isUnderCollateral()) revert ErrorUnderCollateral();

  address _market = markets[_baseToken].market;
  address _fToken = markets[_baseToken].fToken;

  uint256 _balance = IERC20Upgradeable(_fToken).balanceOf(address(this));

  // @audit if the value of `_amountIn` is greater than the fToken
balance under the contract, the operation will revert.
  (_amountOut, _bonusOut) = IFxMarketV2(_market).redeemFToken(_amountIn,
_receiver, _minOut);
  // the real amount of fToken redeemed
  _amountIn = _balance -
IERC20Upgradeable(_fToken).balanceOf(address(this));
```

```
    _burnShares(_baseToken, _msgSender(), _amountIn);
    emit Unwrap(_baseToken, _msgSender(), _receiver, _amountIn);
  }
```

**Status**

The development team explained, "Allowing partial redemption may not align with user expectations. Therefore, we do not permit partial redemption of assets."

### 4.3.3 Lack of validation on some crucial input parameters in FxUSDFacet.sol

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Info | Design logic | Discussed |

**Location**

FxUSDFacet.sol#L46

FxUSDFacet.sol#L64

FxUSDFacet.sol#L82

FxUSDFacet.sol#L102

FxUSDFacet.sol#L131

FxUSDFacet.sol#L203

**Description**

Similar to the first issue, attackers can also forge the following addresses to steal user funds or compromise the integrity of the protocol. It is recommended to validate the legitimacy of parameters such as `_vault`, `_market`, `_pool`, and others at the contract level.

```
  function fxInitialFundDeposit(LibGatewayRouter.ConvertInParams memory
  _params, address _vault) //@audit attackers could forge parameter of
  _vault
    external
    payable
    returns (uint256 _baseOut)
  {
    address _baseToken = FxInitialFund(_vault).baseToken();
    _baseOut = LibGatewayRouter.transferInAndConvert(_params, _baseToken);
```

```
  LibGatewayRouter.approve(_baseToken, _vault, _baseOut);
  FxInitialFund(_vault).deposit(_baseOut, msg.sender);
}

function fxMintFTokenV2(
  LibGatewayRouter.ConvertInParams memory _params,
  address _market,  //@audit attackers could forge parameter of _market
  uint256 _minFTokenMinted
) external payable returns (uint256 _fTokenMinted) {
  address _baseToken = IFxMarketV2(_market).baseToken();
  uint256 _amount = LibGatewayRouter.transferInAndConvert(_params,
_baseToken);
  LibGatewayRouter.approve(_baseToken, _market, _amount);
  _fTokenMinted = IFxMarketV2(_market).mintFToken(_amount, msg.sender,
_minFTokenMinted);
  LibGatewayRouter.refundERC20(_baseToken, msg.sender);
}

function fxMintXTokenV2(
    LibGatewayRouter.ConvertInParams memory _params,
    address _market,  //@audit attackers could forge parameter of _market
    uint256 _minXTokenMinted
  ) external payable returns (uint256 _xTokenMinted, uint256 _bonusOut) {
    address _baseToken = IFxMarketV2(_market).baseToken();
    uint256 _amount = LibGatewayRouter.transferInAndConvert(_params,
_baseToken);
    LibGatewayRouter.approve(_baseToken, _market, _amount);
    (_xTokenMinted, _bonusOut) = IFxMarketV2(_market).mintXToken(_amount,
msg.sender, _minXTokenMinted);
    LibGatewayRouter.refundERC20(_baseToken, msg.sender);
  }

function fxRedeemFTokenV2(
    LibGatewayRouter.ConvertOutParams memory _params,
    address _market,  //@audit attackers could forge parameter of _market
    uint256 _fTokenIn,
    uint256 _minBaseToken
  )
    external
    returns (
      uint256 _baseOut,
      uint256 _dstOut,
      uint256 _bonusOut
    )
  {
```

```
    address _fToken = IFxMarketV2(_market).fToken();
    address _baseToken = IFxMarketV2(_market).baseToken();
    _fTokenIn = LibGatewayRouter.transferTokenIn(_fToken, address(this),
_fTokenIn);

    (_baseOut, _bonusOut) = IFxMarketV2(_market).redeemFToken(_fTokenIn,
address(this), _minBaseToken);
    _dstOut = LibGatewayRouter.convertAndTransferOut(_params, _baseToken,
_baseOut + _bonusOut, msg.sender);
    LibGatewayRouter.refundERC20(_fToken, msg.sender);
  }

function fxRedeemXTokenV2(
    LibGatewayRouter.ConvertOutParams memory _params,
    address _market,  //@audit attackers could forge parameter of _market
    uint256 _xTokenIn,
    uint256 _minBaseToken
  ) external returns (uint256 _baseOut, uint256 _dstOut) {
    address _xToken = IFxMarketV2(_market).xToken();
    address _baseToken = IFxMarketV2(_market).baseToken();
    _xTokenIn = LibGatewayRouter.transferTokenIn(_xToken, address(this),
_xTokenIn);

    _baseOut = IFxMarketV2(_market).redeemXToken(_xTokenIn,
address(this), _minBaseToken);
    _dstOut = LibGatewayRouter.convertAndTransferOut(_params, _baseToken,
_baseOut, msg.sender);
    LibGatewayRouter.refundERC20(_xToken, msg.sender);
  }

function fxMintFxUSDAndEarn(
    LibGatewayRouter.ConvertInParams memory _params,
    address _pool,  //@audit attackers could forge parameter of _pool
    uint256 _minFxUSDMinted
  ) external payable returns (uint256 _fxUSDMinted) {
    address _baseToken = IFxShareableRebalancePool(_pool).baseToken();
    uint256 _amount = LibGatewayRouter.transferInAndConvert(_params,
_baseToken);
    LibGatewayRouter.approve(_baseToken, fxUSD, _amount);
    _fxUSDMinted = IFxUSD(fxUSD).mintAndEarn(_pool, _amount, msg.sender,
_minFxUSDMinted);
    LibGatewayRouter.refundERC20(_baseToken, msg.sender);
  }
```

**Status**

From a contract-level perspective, those user-input parameters will not threaten the protocol now but will only affect their own funds. Therefore, the development team has decided not to adjust the corresponding code logic in the current version of the protocol.

### 4.3.4 The return value of the `get_gauge_weight()` function may lag behind the actual value.

| Risk Type | Risk Level | Impact | Status |
| --- | --- | --- | --- |
| Design & Implementation | Medium | Design logic | Fixed |

**Location**

GaugeControllerOwner.sol#L100

**Description**

The function `normalizeGaugeWeight()` aims to reset the weights of each gauge according to a fixed proportion. Before computing the combined weights of each gauge, this function calls the `checkpoint()` function under the `GaugeController` contract to update the weight statuses of each gauge. However, it's important to note that the `checkpoint()` function does not update the value of the parameter `time_weight[addr]`, which may result in the `get_gauge_weight()` function returning weights that are not the latest. It could lead to deviations in the calculation results of the combined weights.

In reality, the parameter `time_weight[addr]` needs to be updated through the `checkpoint_gauge()` function. Therefore, before calling the `get_gauge_weight()` function, the `checkpoint_gauge()` function should be called first.

```
function normalizeGaugeWeight() external onlyRole(WEIGHT_NORMALIZER_ROLE)
{
    uint256 _length = gauges.length();
    if (_length == 0) return;

    IGaugeController(controller).checkpoint();
    uint256 _totalWeight =
IGaugeController(controller).get_total_weight();
    address[] memory _gauges = new address[](_length);
    uint256[] memory _gaugeWeights = new uint256[](_length);
    uint256[] memory _typeWeights = new uint256[](_length);
    for (uint256 i = 0; i < _length; i++) {
```

```
        _gauges[i] = gauges.at(i);
        // @audit this value might be lagging.
        _gaugeWeights[i] =
IGaugeController(controller).get_gauge_weight(_gauges[i]);
        int128 _type =
IGaugeController(controller).gauge_types(_gauges[i]);
        _typeWeights[i] =
IGaugeController(controller).get_type_weight(_type);
        _totalWeight -= _gaugeWeights[i] * _typeWeights[i];
    }
    ......
}
```

```
// @audit GaugeController.vy :
// https://github.com/AladdinDAO/aladdin-v3-
contracts/blob/6c0039340c563e43d2125321d5f076c2b96a83bd/contracts/voting-
escrow/gauges/GaugeController.vy

@external
def checkpoint():
    """
    @notice Checkpoint to fill data common for all gauges
    """
    self._get_total()

@internal
def _get_total() -> uint256:
    """
    @notice Fill historic total weights week-over-week for missed
checkins
            and return the total for the future week
    @return Total weight
    """
    t: uint256 = self.time_total
    _n_gauge_types: int128 = self.n_gauge_types
    if t > block.timestamp:
        # If we have already checkpointed - still need to change the
value
        t -= WEEK
    pt: uint256 = self.points_total[t]

    for gauge_type in range(100):
        if gauge_type == _n_gauge_types:
            break
```

```
            self._get_sum(gauge_type)
            self._get_type_weight(gauge_type)

        for i in range(500):
            if t > block.timestamp:
                break
            t += WEEK
            pt = 0
            # Scales as n_types * n_unchecked_weeks (hopefully 1 at most)
            for gauge_type in range(100):
                if gauge_type == _n_gauge_types:
                    break
                type_sum: uint256 = self.points_sum[gauge_type][t].bias
                type_weight: uint256 = self.points_type_weight[gauge_type][t]
                pt += type_sum * type_weight
            self.points_total[t] = pt

            if t > block.timestamp:
                self.time_total = t
        return pt

    @external
    @view
    def get_gauge_weight(addr: address) -> uint256:
        """
        @notice Get current gauge weight
        @param addr Gauge address
        @return Gauge weight
        """
        return self.points_weight[addr][self.time_weight[addr]].bias
```

**Status**

The development team addressed this issue in commit cb83a2c.

### 4.3.5 When modifying the weight of a gauge through the `change_gauge_weight()` function, the impact of type weight is not considered.

| Risk Type | Risk Level | Impact | Status |
| --- | --- | --- | --- |
| Design & Implementation | Low | Design logic | Fixed |

**Location**

GaugeControllerOwner.sol#L111

**Description**

When gauges (with weights greater than 0) set under the `GaugeController` contract are all accounted for in this contract, it directly adjusts the weights of the corresponding gauges to ensure their combined weight equals 1 (*PRECISION*). It's important to note that the `GaugeController` contract considers both gauge weight and type weight when calculating the combined weight, while this contract only modifies the gauge weight of these gauges through the `change_gauge_weight()` function without considering the type weight of each gauge.

```
function normalizeGaugeWeight() external onlyRole(WEIGHT_NORMALIZER_ROLE)
{
    uint256 _length = gauges.length();
    if (_length == 0) return;

    IGaugeController(controller).checkpoint();
    uint256 _totalWeight =
IGaugeController(controller).get_total_weight();
    address[] memory _gauges = new address[](_length);
    uint256[] memory _gaugeWeights = new uint256[](_length);
    uint256[] memory _typeWeights = new uint256[](_length);
    for (uint256 i = 0; i < _length; i++) {
      _gauges[i] = gauges.at(i);
      _gaugeWeights[i] =
IGaugeController(controller).get_gauge_weight(_gauges[i]);
      int128 _type =
IGaugeController(controller).gauge_types(_gauges[i]);
      _typeWeights[i] =
IGaugeController(controller).get_type_weight(_type);
      _totalWeight -= _gaugeWeights[i] * _typeWeights[i];
    }

    if (_totalWeight == 0) {
      uint256 sum;
      for (uint256 i = 0; i < _length; i++) {
        uint256 w = weights[_gauges[i]];
        sum += w;
        //@audit type weight was not taken into consideration
        IGaugeController(controller).change_gauge_weight(_gauges[i], w);
      }
      if (sum != PRECISION) revert ErrorNoSolution();
```

```
        return;
      }

    ......
  }
```

**Status**

The development team addressed this issue in commit cb83a2c.

### 4.3.6 Discussion on the logic of functions `fxMintFTokenV2()`, `fxRedeemFTokenV2()`, and `fxSwapV2()`.

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Info | Design logic | Discussed |

**Location**

FxUSDFacet.sol#L62

FxUSDFacet.sol#L100

FxUSDFacet.sol#L151

**Description**

The current protocol does not allow users to mint fTokens directly; they can only mint fxUSD tokens. However, the three functions listed below may violate the protocol design requirements. Considering two scenarios: (1) Before using the fxUSD contract, any user can directly call the `MarketV2.mintFToken()` function to mint fTokens. In this case, users can obtain fTokens instead of fxUSD through the `MarketV2.mintFToken()` function or the `fxMintFTokenV2()` function and `fxSwapV2()` function listed below, which will cause fTokens to flow out of the protocol, contrary to the protocol's requirements. (2) If the fxUSD contract and the MarketV2 contract are deployed synchronously, ordinary users cannot directly call the `MarketV2.mintFToken()` function to mint fTokens. In this case, since the `mintFToken()` function can only be called by the fxUSD contract, the `fxMintFTokenV2()` function and `fxSwapV2()` function listed below will not be usable. If so, retaining these functions would be meaningless. The `fxRedeemFTokenV2()` function requires users to transfer fTokens. Normally, users cannot directly hold fTokens but should hold fxUSD instead. Therefore, this function cannot be used either. In summary, it is worth considering whether to remove the `fxMintFTokenV2()` function, `fxRedeemFTokenV2()` function, and `fxSwapV2()` function.

---

```
function fxMintFTokenV2(
    LibGatewayRouter.ConvertInParams memory _params,
    address _market,
    uint256 _minFTokenMinted
  ) external payable returns (uint256 _fTokenMinted) {
    address _baseToken = IFxMarketV2(_market).baseToken();
    uint256 _amount = LibGatewayRouter.transferInAndConvert(_params,
_baseToken);
    LibGatewayRouter.approve(_baseToken, _market, _amount);
    // @audit use can receive fToken istead of fxUSD
    _fTokenMinted = IFxMarketV2(_market).mintFToken(_amount, msg.sender,
_minFTokenMinted);
    LibGatewayRouter.refundERC20(_baseToken, msg.sender);
  }

//@audit according to the protocol design, users do not directly hold
fTokens.
function fxRedeemFTokenV2(
    LibGatewayRouter.ConvertOutParams memory _params,
    address _market,
    uint256 _fTokenIn,
    uint256 _minBaseToken
  )
    external
    returns (
      uint256 _baseOut,
      uint256 _dstOut,
      uint256 _bonusOut
    )
  {
    address _fToken = IFxMarketV2(_market).fToken();
    address _baseToken = IFxMarketV2(_market).baseToken();

    _fTokenIn = LibGatewayRouter.transferTokenIn(_fToken, address(this),
_fTokenIn);
    (_baseOut, _bonusOut) = IFxMarketV2(_market).redeemFToken(_fTokenIn,
address(this), _minBaseToken);
    _dstOut = LibGatewayRouter.convertAndTransferOut(_params, _baseToken,
_baseOut + _bonusOut, msg.sender);
    LibGatewayRouter.refundERC20(_fToken, msg.sender);
  }

function fxSwapV2(
    address _market,
```

```
    uint256 _amountIn,
    bool _fTokenForXToken,
    uint256 _minOut
  ) external returns (uint256 _amountOut, uint256 _bonusOut) {
    address _fToken = IFxMarketV2(_market).fToken();
    address _xToken = IFxMarketV2(_market).xToken();
    address _baseToken = IFxMarketV2(_market).baseToken();

    if (_fTokenForXToken) {
      _amountIn = LibGatewayRouter.transferTokenIn(_fToken,
address(this), _amountIn);
      //@audit users do not hold fTokens directly
      (uint256 _baseOut, uint256 _redeemBonus) =
IFxMarketV2(_market).redeemFToken(_amountIn, address(this), 0);
      _bonusOut = _redeemBonus;
      LibGatewayRouter.approve(_baseToken, _market, _baseOut);
      (_amountOut, _redeemBonus) =
IFxMarketV2(_market).mintXToken(_baseOut, msg.sender, 0);
      _bonusOut += _redeemBonus;
      LibGatewayRouter.refundERC20(_fToken, msg.sender);
    } else {
      _amountIn = LibGatewayRouter.transferTokenIn(_xToken,
address(this), _amountIn);
      uint256 _baseOut = IFxMarketV2(_market).redeemXToken(_amountIn,
address(this), 0);
      LibGatewayRouter.approve(_baseToken, _market, _baseOut);
      // @audit user can receive fToken directly
      _amountOut = IFxMarketV2(_market).mintFToken(_baseOut, msg.sender,
_minOut);
      LibGatewayRouter.refundERC20(_xToken, msg.sender);
    }

    LibGatewayRouter.refundERC20(_baseToken, msg.sender);
  }
```

**Status**

The development team explains that these functions are retained, considering the possibility of markets not supporting minting fxUSD tokens.

### 4.3.7 The admin should avoid the expected total weight sum of all gauges within the whitelist being 1e18.

| Risk Type | Risk Level | Impact | Status |
| --- | --- | --- | --- |
| Design & Implementation | Info | Design logic | Discussed |

#### Location

GaugeControllerOwner.sol#L142

#### Descriptions

Assuming that not all gauges are currently whitelisted, meaning that the parameter _totalWeight > 0, in this case, it is advisable to avoid setting the sum of expected weights (relative weights, denoted as w) of all gauges in the whitelist to 1e18, as doing so may lead to the system of equations having no solution.

If the system of equations has no solution, the code `GaussElimination.solve(a, b)` will return false, and the function `normalizeGaugeWeight()` will fail to execute without explicitly providing a failure reason. This can make it difficult for administrators to make appropriate weight adjustments. In the scenario where `_totalWeight > 0`, it is advisable to add a check for the sum of expected weights in the `normalizeGaugeWeight()` function, disallowing the sum to be precisely 1e18. Additionally, when administrators set the expected weights for each gauge in the whitelist, they should also be reminded to ensure that the weight settings do not lead to an unsolvable system of equations.

```
function normalizeGaugeWeight() external onlyRole(WEIGHT_NORMALIZER_ROLE)
{
    uint256 _length = gauges.length();
    if (_length == 0) return;

    ......

    // we need to solve the following equation with gauss elimination.
    // w[i] / 1e18 = tw[i] * x[i] / (total + sum tw[j] * x[j])
    // where
    //   w[i] is the expected relative weight
    //   tw[i] is current type weight
    //   x[i] is the expected gauge weight to set
    int256[][] memory a = new int256[][](_length);
    int256[] memory b = new int256[](_length);
    for (uint256 r = 0; r < _length; r++) {
```

```
        a[r] = new int256[](_length);
        uint256 w = weights[_gauges[r]];
        b[r] = int256((_totalWeight * w) / PRECISION);
        for (uint256 c = 0; c < _length; c++) {
          if (r == c) a[r][c] = int256(_typeWeights[r] - (_typeWeights[r] *
w) / PRECISION);
          else a[r][c] = -int256((_typeWeights[c] * w) / PRECISION);
        }
      }
      // solve the equation and save solution in b
      if (!GaussElimination.solve(a, b)) revert ErrorNoSolution();
      for (uint256 i = 0; i < _length; i++) {
        b[i] /= int256(PRECISION);
        if (b[i] <= 0) revert ErrorInvalidSolution();
        IGaugeController(controller).change_gauge_weight(_gauges[i],
uint256(b[i]));
      }
    }
```

**Status**

The development team has confirmed that they will avoid setting the total weight of all gauges in the whitelist to 1e18 when configuring each gauge.

# 5. Conclusion

After auditing and analyzing the fxUSD protocol, SECBIT Labs found some issues to optimize and proposed corresponding suggestions, which have been shown above.

# Disclaimer

SECBIT smart contract audit service assesses the contract's correctness, security, and performability in code quality, logic design, and potential risks. The report is provided "as is", without any warranties about the code practicability, business model, management system's applicability, and anything related to the contract adaptation. This audit report is not to be taken as an endorsement of the platform, team, company, or investment.

# APPENDIX

## Vulnerability/Risk Level Classification

| Level | Description |
| --- | --- |
| High | Severely damage the contract's integrity and allow attackers to steal Ethers and tokens, or lock assets inside the contract. |
| Medium | Damage contract's security under given conditions and cause impairment of benefit for stakeholders. |
| Low | Cause no actual impairment to contract. |
| Info | Relevant to practice or rationality of the smart contract, could possibly bring risks. |

**SECBIT Lab is devoted to constructing a common-consensus, reliable, and ordered blockchain economic entity.**


https://secbit.io

audit@secbit.io

@secbit_io