



Audit Boutique

Fiat24 Smart Contracts Audit Report

Produced for:

SR Saphirstein AG

Produced by:

Dominik Spicher, Audit Boutique

March 3rd 2024



Contents

1. Executive Summary	4
2. Scope	4
3. Methodology	5
4. System description	6
4.1. Contracts	6
4.2. Roles	7
4.3 Upgradeability	7
5. Best practices checklist	8
6. Findings	9
6.1 High severity	9
6.1.1 Faulty indication that an account number is not for sale	9
6.2 Medium severity	9
6.2.1 Unused minDigitForSale variable in Fiat24Account	9
6.2.2 Uncalled for conversion factor in mintByClientWithETH	10
6.3 Low severity	10
6.3.1 Fiat24Account is missing some initialization calls on parent contracts	10
6.3.2 Missing check for minDigitForSale	10
6.3.3 mintByWallet can be performed under inactive F24	10
6.3.4 mintByWallet fails to perform minting setup tasks	11
6.3.5 upgradeWithF24 seems to allow accounts to own multiple token IDs in succession and “squat” token IDs	11
6.3.6 historicOwnership only allows tracking one previously owned token	11
6.3.7 Inconsistent time point at which historicOwnership is set	12
6.3.8 Spurious setupRole method	12
6.3.9 Reentrancy vulnerability in Fiat24Account transfer functions	12
6.3.10 Ineffectual Uniswap trade deadline	13
6.3.11 Disabled Uniswap trade safety parameter	13
6.3.12 Race condition when Fiat24CryptoDeposit holds an ETH balance	14
6.3.13 Inconsistent handling of USDC amounts in Fiat24CryptoDeposit	14
6.4 Informational	14
6.4.1 Virtually identical cash payout methods	14
6.4.2 Two conditionals in upgradeWithF24 can be simplified into a single require	15
6.4.3 Leftover state variables after token burn	15
6.4.4 Excess purchase of F24 token	15
6.4.5 Inconsistent internal account names across contracts	15
6.4.6 Computing a multiplication on the result of a division	15
6.4.7 Superfluous reimplementations of safe subtraction logic provided by SafeMath contract	16
6.4.8 Wasteful representation of uint256 digit count	16
6.4.9 Unused EnumerableUintsToUintsMapUpgradeable import	16



6.4.10 Inconsistent method of determining msg.sender	16
6.4.11 Unused access control functionality on the Fiat24PriceList contract	17
6.4.12 Multi-clause catch expression can be simplified	17
6.4.13 State updates don't emit suitable events	17
6.4.14 Preconditions are checked through brittle code repetition instead of modifiers	17
6.4.15 Custom Solidity errors are used incomprehensively	18
6.4.16 Unnecessary STATICCALL overhead through the usage of "this"	18
6.4.17 State variables that could be declared immutable	18
6.4.18 Implementations don't implement their interface	19
6.4.19 Unused state variables and constant on Fiat24CryptoDeposit contract	19
6.4.20 Event with duplicated argument	19
6.4.21 Arguments supplied in the wrong order	19
6.4.22 Duplicated logic to determine most liquid exchange pool	20
6.4.23 Repeated calls to getQuote with identical arguments	20
6.4.24 Confusing and inconsistent usage of corrections by basis point multiplier	20
6.4.25 Significant duplicated code in three deposit methods	20
7. Limitations	22



1. Executive Summary

The audit has uncovered one issue of high severity, two issues of medium severity, and thirteen issues of low severity. We also list twenty-five minor suggestions of how the contracts could be improved.

2. Scope

The review pertained to the following smart contract source files, specified here with their sha256 prefixes:

F24Sales.sol	a91f2e0c29b6755e0396
F24.sol	ef273341fddd5f7ef39f
Fiat24Account.sol	1c576e07976515c640b4
Fiat24Airdrop.sol	e7ff1d3ed8981d11b0dd
Fiat24CardAuthorization.sol	281a61c604bf7f78e611
Fiat24CHF.sol	8bfcc7fe526cbe110ef7
Fiat24ClientApplication.sol	281658ccf617374144f0
Fiat24CryptoDeposit.sol	2cba626cec64abbf1361
Fiat24EUR.sol	bb5363c07f65b06b3b6f
Fiat24GBP.sol	7978b00115d700710cc6
Fiat24Lock.sol	cab842cd9bec47141649
Fiat24PriceList.sol	facf7dbdcbc3d35b2cce
Fiat24Token.sol	d5532387155309027e74
Fiat24USD.sol	a58b7884ebd02eab1c20
interfaces/ArbSys.sol	8f445ddc6e1763a555eb
interfaces/IF24Sales.sol	e6db599f25fbf22e0ae4
interfaces/IF24.sol	a23bf51219c71669c139
interfaces/IF24TimeLock.sol	519f8ddb6f6c30b3db50
interfaces/IFiat24Account.sol	04522003d311ee15a7a1
interfaces/IFiat24Lock.sol	11aa432df3b31daf59a9
interfaces/IFiat24Token.sol	106c3482627d404f46b9
interfaces/IUSDC.sol	ab7167ea4f6bc51dc4c4
interfaces/SanctionsList.sol	10993c85504c555d3d23
libraries/DigitsOfUint.sol	81226609170ec316c681
libraries/EnumerableUintToUmapUpgradeable.sol	f679b7e3c85678fc844b

Imported contracts from well-known libraries such as Open-Zeppelin were not part of the review.

The review was constrained to the Solidity source file. Low-level assembly code generated thereof was not inspected.



3. Methodology

The review consisted of the following steps:

- Check for compliance with smart contract development best practices
- Check for compliance with specification, where available and applicable
- Manual inspection and analysis of the smart contract and test code
- Usage of static analysis tools



4. System description

4.1. Contracts

Fiat24 is a comprehensive system of smart contracts designed to integrate payments in different fiat currencies and credit/debit cards with their on-chain counterparts and an on-chain accounting system.

Starting with the accounting system, the Fiat24Account smart contract implements an accounting system based on ERC721 where NFT token IDs correspond to client account numbers. This ID is part of the user's IBAN number of Fiat24 as well. For onboarding, clients register their application data and geolocation on the Fiat24ClientApplication contract. Afterwards, clients usually have a single, long-lived NFT that represents their account. In order to "buy" an account number (where the price is determined by a PriceList contract, giving a higher price to lower account numbers), F24 utility tokens (a simple ERC20) need to be purchased and burned. F24 tokens can be bought for the native blockchain currency through a dedicated F24Sales contract. An operator-chosen set of addresses is also able to claim specific airdrop amounts of F24 or other tokens (such as ARB, etc.) by submitting a suitable Merkle proof to the Fiat24Airdrop contract.

On the fiat payments side, the core functionality is provided by four Fiat24Token contracts that map to the CHF, EUR, GBP and USD fiat currencies respectively. These tokens are used to tokenize the client's Fiat24 deposits. They can be used to make P2P payments, card payments (currently EUR only), deposits and withdrawals. Client balances and transfers in these tokens originate from operators, registering corresponding fiat movements in the token smart contracts. Payments made with debit cards are registered and result in token movements through the Fiat24CardAuthorization contract.

Moreover, the Fiat24Token contract contains the functions to deposit or withdraw money to or from respectively the client's Fiat24 account using the SIC/euroSIC Network offered by Fiat24. In case a deposit cannot be conducted due to exceeding limits or wrong status (anything but "Live"), the money is locked in the Fiat24Lock contract, from where the user can claim the money as soon as he is at the correct status or the 30-day limit has been reset.

Clients may also increase their token CHF, EUR, GBP, or USD balances by making their arbitrary token available to a Fiat24CryptoDeposit contract, which will exchange it on a Uniswap-like platform either for USDC or the native blockchain currency, which in turn will be transformed into the user's chosen token through a fixed configured exchange rate.

Throughout, clients are categorized into either "tourists" (before having gone through compliance checks) and active accounts (after having performed compliance checks). In various places, tourists have limited functionality or lower limits associated with them.



4.2. Roles

Almost all contracts make pervasive use of roles and permissioned access. Besides an `OPERATOR_ROLE` which is present on every permissioned contract, there is little integration and / or structure to enlist, and we refrain from a comprehensive enumeration.

4.3 Upgradeability

Most contracts, except the F24 and Fiat24Airdrop contracts are upgradeable using OpenZeppelin proxy contracts.¹

¹ <https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable>



5. Best practices checklist

Non-adherence to the characteristics listed in the below list does not represent a security issue itself. However, following best practices is a good indicator of care and attention to detail, it allows the review process to be more focused and efficient, thus making it more likely for issues to be uncovered.

- The code was provided as a Git repository
- There exists specification, covering the most important aspects of smart contract functionality
- The development process is understandable through well-delineated, atomic commits
- Code duplication is minimal
- ✓ The smart contract source files are provided in an unflattened manner
- ✓ The code compiles with a recent Solidity version
- The code is consistently formatted
- ✓ Code comments are in line with the associated code
- ✓ There are tests
- The code is well documented
- There is no commented code
- There is no unused code
- The code follows standard Solidity naming conventions



6. Findings

We rank our findings according to their perceived severity (high, medium, low), where an issue's severity is understood to be the product of its likelihood of being triggered and the impact of its consequences.

Section 6.4 lists suggestions for improvements which have no discernible security impact.

6.1 High severity

6.1.1 Faulty indication that an account number is not for sale

The `Fiat24PriceList` contract returns a price for a given account number, with lower account numbers demanding a higher price. Above a maximum of 8999, and if the given account number does not start with a dedicated merchant digit, the contract intends to signal that a number is not for sale by returning a special price. However, the used value of 100 is out of agreement with the call site in the `upgradeWithF24` method of the `Fiat24Account` contract, which expects a zero return value in this case. Thus, all account numbers will be mistakenly up for sale.

Care should be taken when fixing this issue, as a zero return value is potentially dangerous for other call sites in the `Fiat24Account` contract, for example in `mintByClient` where a zero return value would lead to a zero burn cost for the caller.

Code locations:

- `Fiat24PriceList.sol:L39`
- `Fiat24Account.sol:L157-158`
- `Fiat24Account.sol:L94-95`

6.2 Medium severity

6.2.1 Unused `minDigitForSale` variable in `Fiat24Account`

The `Fiat24Account` contract defines `minDigitForSale` and `maxDigitForSale` state variables to constrain the space of available token IDs. However, `minDigitForSale` fails to be utilized for those checks, and the hard-coded value 5 is used instead.

Note that this issue is merely of informational severity in case the hard-coded usage of a constant 5 represents the intended behavior, and the `minDigitForSale` variable was simply forgotten to be removed.

Code locations:

- `Fiat24Account.sol:L133`



6.2.2 Uncalled for conversion factor in mintByClientWithETH

The `Fiat24Account` contract in the `mintByClientWithETH` method computes the `priceETH` variable based on the comparison of two other values, `quotePerETH` and `accountPrice`. Depending on their relative size, a factor of 10^{18} , which is the conversion factor between wei and Ether, is either applied or not (although this fact is slightly obscured through a double division).

However, the origin locations of `quotePerETH` and `accountPrice` do not suggest at all that these values originate under two regimes that correspond to a factor-of- 10^{18} separation. Rather, they both originate from a comparably much more contiguous range of possible values. It is thus unlikely that such a huge correction represents intended behavior.

Code locations:

- `Fiat24Account.sol:L103-107`

6.3 Low severity

6.3.1 Fiat24Account is missing some initialization calls on parent contracts

The `Fiat24Account` contract fails to call the appropriate initialization calls on some of its parent upgradeable contracts, for example `__Pausable_init_unchained` which initializes the `_paused` state variable.

We were unable to verify the impact of this.

Code locations:

- `Fiat24Account.sol:L71-81`

6.3.2 Missing check for minDigitForSale

The `_mintByClient` method of the `Fiat24Account` contract only checks the number of token ID digits against the maximal value, not the minimal value.²

There is some uncertainty in regard to the intentionality of this behavior, which is why we classify this issue as low severity.

Code locations:

- `Fiat24Account.sol:L120`

6.3.3 mintByWallet can be performed under inactive F24

The `mintByWallet` method of the `Fiat24Account` contract fails to assert `f24IsActive` (as opposed to `_mintByClient`), even though it calls token transfer functions on the F24 contract.

Code locations:

² See also issue 6.2.1.



- `Fiat24Account.sol:L128-142`

6.3.4 mintByWallet fails to perform minting setup tasks

The `mintByWallet` method of the `Fiat24Account` contract fails to call `initilizeTouristLimit` and to set the `nickName`, both of which are done by `mint()` and `_mintByClient()`, possibly indicating an inconsistent minting scenarios.

There is some uncertainty in regard to the intentionality of this behavior.

Code locations:

- `Fiat24Account.sol:L128-142`

6.3.5 upgradeWithF24 seems to allow accounts to own multiple token IDs in succession and “squat” token IDs

The `upgradeWithF24` method of the `Fiat24Account` contract allows privileged parties to “upgrade” their token ID to a new one, forfeiting their old one to the unnamed 9106 internal account. This seems to violate the expectation that users only ever own one account number, thus possibly invalidating some expectations of code elsewhere (see issue 6.3.6). For the purposes of this finding, the method seems to allow users to “squat” interesting token IDs which subsequently become unavailable to others. The only precondition for this behavior is that their `tokenID` status is `Live` at the time of upgrading.

There is some uncertainty in regard to the intentionality of this behavior.

Code locations:

- `Fiat24Account.sol:L144-176`

6.3.6 historicOwnership only allows tracking one previously owned token

The `historicOwnership` mapping of the `Fiat24Account` contract tracks a single token ID that a particular account previously owned. However, as it seems possible for addresses to buy multiple token IDs in succession,³ this fails to entirely capture the ownership history of an individual account. Indeed, some call sites which use the `historicOwnership` mapping indicate that this would violate their expectations about how this mapping works.

There is some uncertainty in regard to the intentionality of this behavior.

Code locations:

- `Fiat24Account.sol:L47`

³ See also issue 6.3.5.



6.3.7 Inconsistent time point at which historicOwnership is set

The historicOwnership mapping of the Fiat24Account contract seems to be inconsistently updated with new values, as it is set to *the new token ID* within upgradeWithF24, but not at all in all other minting entrypoints. Possibly the mapping should have been set to the old token ID in upgradeWithF24.

Code locations:

- Fiat24Account.sol:L83-90
- Fiat24Account.sol:L117-126
- Fiat24Account.sol:L165

6.3.8 Spurious setupRole method

The Fiat24ClientApplication contract contains a setupRole method which appears spurious on multiple accounts:

- It is the only permissioned contract that exposes this method.
- The called delegate _setupRole of the AccessControlUpgradeable contract is marked as deprecated.⁴
- The documentation of the delegate _setupRole of the AccessControlUpgradeable explicitly warns against exposing this method outside of constructors.⁵

Code locations:

- Fiat24ClientApplication.sol:L169-L174

6.3.9 Reentrancy vulnerability in Fiat24Account transfer functions

The public transferFrom and safeTransferFrom methods of the Fiat24Account smart contract call super.safeTransferFrom and afterwards conditionally write to the historicOwnership state variable. This is vulnerable to cross-function reentrancies due to the fact that the ERC721 safeTransferFrom function will explicitly call a standardized callback on smart contract token receivers to avoid locked tokens.⁶ Furthermore, the historicOwnership variable is used in numerous other methods which may subsequently read outdated historicOwnership values:

- Fiat24Account._beforeTokenTransfer
- Fiat24Account._mintAllowed
- Fiat24Account.activateWithReferral
- Fiat24Account.changeClientStatus
- Fiat24Account.historicOwnership
- Fiat24Account.removeHistoricOwnership

⁴

<https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/v4.4.1/contracts/access/AccessControlUpgradeable.sol#L192>

⁵

<https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/v4.4.1/contracts/access/AccessControlUpgradeable.sol#L183-L190>

⁶

<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/7a29f7df5190625c41c9948820ee9421b5d44d59/contracts/token/ERC721/IERC721.sol#L50-L51>



- `Fiat24Account.safeTransferFrom`
- `Fiat24Account.transferFrom`
- `Fiat24Account.upgradeWithF24`

The impact of these methods working off outdated `historicOwnership` values remains unclear, which is why we classify this issue as low severity.

Code locations:

- `Fiat24Account.sol:L185-188`
- `Fiat24Account.sol:L192-195`

6.3.10 Ineffectual Uniswap trade deadline

The `Fiat24CryptoDeposit` contract at multiple locations creates arguments for a Uniswap trade. Those arguments include a `deadline` parameter that can be set to the maximum timestamp at which a trade should be executable, to protect against excessive exchange rate changes through time.

The contract always sets this value to `block.timestamp + 15`, which is equivalent to having no deadline at all, due to the trades by definition being executed at the `block.timestamp` time. The usage of the number 15 suggests that there was some effect that was intended to be achieved, but which remains unrealized here.

Code locations:

- `Fiat24CryptoDeposit.sol: L171`
- `Fiat24CryptoDeposit.sol: L230`
- `Fiat24CryptoDeposit.sol: L287`
- `Fiat24CryptoDeposit.sol: L306`

6.3.11 Disabled Uniswap trade safety parameter

The `Fiat24CryptoDeposit` contract at multiple locations creates arguments for a Uniswap trade. Those arguments include a `sqrtPriceLimitX96` parameter that can be set to constrain the range of possible exchange rates that the caller is permitting. Unfortunately, this safety parameter is deactivated by setting it to zero throughout, which is discouraged.⁷

Code locations:

- `Fiat24CryptoDeposit.sol: L174`
- `Fiat24CryptoDeposit.sol: L233`
- `Fiat24CryptoDeposit.sol: L290`
- `Fiat24CryptoDeposit.sol: L309`

⁷ <https://docs.uniswap.org/contracts/v3/guides/swaps/single-swaps#swap-input-parameters>



6.3.12 Race condition when Fiat24CryptoDeposit holds an ETH balance

The `Fiat24CryptoDeposit` contract has a `depositETH` method that will transfer excess ETH back to the caller. This is done through the `wholesale address(this).balance` amount, instead of using an exact value. This is a reasonable choice if the contract starts out with a zero ETH balance when `depositETH` is called. However, this precondition may not hold. Indeed, the possibility that the contract holds an ETH balance seems to have been envisaged due to the explicit inclusion of a `receive()` payable `external {}` function.

Whenever this scenario materializes, calling `depositETH` presents a race condition where users may race to be the first ones to call the method and collect the prize.

Code locations:

- `Fiat24CryptoDeposit.sol`: L178

6.3.13 Inconsistent handling of USDC amounts in Fiat24CryptoDeposit

The three methods `depositETH`, `depositTokenViaUsdc` and `depositTokenViaEth` in the `Fiat24CryptoDeposit` contract contain identical code blocks⁸ where USDC and potentially USD24 are transferred to a deposit address and a fee recipient. In the case where USD24 ends up being used, the `outputAmount` is not corrected for the fee, and their sum seems to exceed the intended total value, compared with the else-branch where this correction is being done.

There is some uncertainty in regard to the intentionality of this behavior.

Code locations:

- `Fiat24CryptoDeposit.sol`: L189-190
- `Fiat24CryptoDeposit.sol`: L248-249
- `Fiat24CryptoDeposit.sol`: L321-322

6.4 Informational

6.4.1 Virtually identical cash payout methods

The `Fiat24Token` contract contains the `cashPayoutOK` and `cashPayoutNOK` methods which only differ in the value of a single account number (9104 and 9103 respectively). Those methods could be reimplemented in terms of a commonly used private method for better clarity.

Code locations:

- `Fiat24Token.sol`: L112-119
- `Fiat24Token.sol`: L121-128

⁸ See also issue 6.4.25.



6.4.2 Two conditionals in upgradeWithF24 can be simplified into a single require

In the upgradeWithF24 method of the Fiat24Account contract, the two conditionals could be combined into a single

```
require(_oldTokenId.hasFirstDigit(MERCHANTDIGIT) == _tokenId.hasFirstDigit(MERCHANTDIGIT))
```

together with a suitable message.

Code locations:

- Fiat24Account.sol:L150-155

6.4.3 Leftover state variables after token burn

In the burn method of the Fiat24Account contract, only the value of the limit mapping is removed, whereas the status, walletProvider and nickNames mapping values remain, possibly leading to inconsistent and confusing state.

Code locations:

- Fiat24Account.sol:L178-L182

6.4.4 Excess purchase of F24 token

The mintByClientWithETH method of the Fiat24Account purchases F24 tokens on the behalf of the user, burns part of it, and sends the rest back to the user. Alternatively, the correct amount of F24 could also be bought, and the rest of the ETH refunded to the user, which would absolve him of the need to fine-tune the message value.

Code locations:

- Fiat24Account.sol:L109-114

6.4.5 Inconsistent internal account names across contracts

The internal account number constants are sometimes called inconsistently across different contracts:

- TREASURY vs TREASURY_DESK
- TREASURY_DESK vs CASHDESK (note the present or absent underscore)

Code locations:

- Fiat24CryptoDeposit.sol:L47
- F24Sales.sol:L23
- Fiat24Airdrop.sol:L34
- Fiat24CardAuthorization.sol:L26
- Fiat24Lock.sol:L24

6.4.6 Computing a multiplication on the result of a division

Numerous code locations in the Fiat24CryptoDeposit contract compute a multiplication on a result of a division, leading to potential precision loss due to Solidity's truncation behavior.



Code locations:

- Fiat24CryptoDeposit.sol:L196
- Fiat24CryptoDeposit.sol:L197
- Fiat24CryptoDeposit.sol:L255
- Fiat24CryptoDeposit.sol:L256
- Fiat24CryptoDeposit.sol:L328
- Fiat24CryptoDeposit.sol:L329
- Fiat24CryptoDeposit.sol:L348
- Fiat24CryptoDeposit.sol:L366
- Fiat24CryptoDeposit.sol:L415

6.4.7 Superfluous reimplementations of safe subtraction logic provided by SafeMath contract

The `eligibleClaimAmount` method of the F24 contract contains logic to return zero when a call to `trySub` resulted in an otherwise underflowing subtraction. However, this is exactly what `trySub` will return already in this case, rendering the inspection of the success return value unnecessary.

Code locations:

- F24.sol:L59-69

6.4.8 Wasteful representation of uint256 digit count

The `DigitsOfUint` and `Fiat24Token` contracts both contain logic to count the number of digits in a 256-bit unsigned integer, for which they use a `uint256 digits`. This is wasteful, as the base-10-logarithm of 2^{256} is roughly 77, easily fitting into a `uint8`.

Code locations:

- DigitsOfUint.sol:L9
- Fiat24Token.sol:L357

6.4.9 Unused EnumerableUintToUintMapUpgradeable import

The `EnumerableUintToUintMapUpgradeable` import in the `Fiat24PriceList` contract is unused and can be removed.

Code locations:

- Fiat24PriceList.sol:L6

6.4.10 Inconsistent method of determining msg.sender

The contracts inconsistently use `msg.sender` and the `_msgSender()` method provided by the `ContextUpgradeable` contract to determine the transaction origin. Those two methods are sometimes used right next to each other, for example in the initialization of the `Fiat24CardAuthorization` contract.



The documentation on the ContextUpgradeable contract provides more information about the motivation to not use `msg.sender` directly.⁹

Code locations:

- `Fiat24CardAuthorization.sol:L41-42`

6.4.11 Unused access control functionality on the Fiat24PriceList contract

The `Fiat24PriceList` contract inherits from `AccessControlUpgradeable`, but does not set up any roles, nor checks permissions inside its method.¹⁰ The parent class is thus unused and can be removed.

Code locations:

- `Fiat24PriceList.sol:L10`

6.4.12 Multi-clause catch expression can be simplified

In the `tokenExists` method of the `Fiat24ClientApplication` contract, two catch clauses are handling different error scenarios with the same response behavior. This can be simplified by using a single empty catch clause `catch { ... }`.¹¹

Code locations:

- `Fiat24ClientApplication.sol:L148-152`

6.4.13 State updates don't emit suitable events

For new values of important contract state variables, it is customary to emit an event to signal the state change. Indeed, this is done in almost all places. The two missing exceptions are both in the `Fiat24Account` when updating the `maxDigitForSale` and `limitTourist` variables.

Code locations:

- `Fiat24Account.sol:L251`
- `Fiat24Account.sol:L289`

6.4.14 Preconditions are checked through brittle code repetition instead of modifiers

Throughout the smart contracts, methods check important preconditions whose dissatisfaction usually results in a transaction failure. Often, these checks are identical across many different functions. Illustrative examples include the functions in the `Fiat24Lock` contract, or the three deposit methods in the `Fiat24CryptoDeposit` contract.

⁹

<https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/0e298be63eff0a9d90ba091817f147395aee651e/contracts/access/OwnableUpgradeable.sol#L9-L20>

¹⁰ Nor do any other contracts inherit from `Fiat24PriceList`, that could themselves use the access control functionality.

¹¹ <https://docs.soliditylang.org/en/latest/control-structures.html#try-catch>



Custom Solidity function modifiers¹² are the perfect tool to express these repeating blocks of code. Instead, the smart contracts replicate the code across different methods, making the setup much more brittle and subject to error. An illustrative example is the check for the operator role,¹³ where the accidental omission of the single-character negation operator would completely change the semantics.

Code locations:

- Fiat24CryptoDeposit.sol:L155-159
- Fiat24CryptoDeposit.sol:L209-213
- Fiat24CryptoDeposit.sol:L268-272
- Fiat24Lock.sol:L56
- Fiat24Lock.sol:L66
- Fiat24Lock.sol:L79

6.4.15 Custom Solidity errors are used incomprehensively

Many of the smart contracts use the user-friendly and gas-efficient custom error feature. However, this could be applied more consistently throughout the smart contract suite. For example, the Fiat24Token and Fiat24Account smart contracts both contain numerous role checks that are handled through a require statement instead of a custom error, which would be more cost-effective.

Code locations:

- Fiat24Token.sol:L84, L179, L203, L208, L224, L249, L254, L259, L277, L285, L292, L316, L338, L345, L350, L356, L397, L402
- Fiat24Account.sol:L62, L67, L89, L104, L113, L122, L208, L213, L218, L223, L228, L233, L238, L243, L401, L406

6.4.16 Unnecessary STATICCALL overhead through the usage of “this”

The Fiat24Account contract contains numerous contract variable reads that use this to refer to the contract, adding an unnecessary overhead to the transaction gas consumption.¹⁴

Code locations:

- Fiat24Account.sol: L129, L130, L145, L210, L217, L227, L231, L241, L242, L243, L333, L361, L362, L407, L413, L417, L419, L424

6.4.17 State variables that could be declared immutable

The F24 and Fiat24Airdrop contracts contain variables that are never updated following deployment. These could be declared immutable to save gas.¹⁵

Code locations:

- F24.sol: L17

¹² <https://solidity-by-example.org/function-modifier/>

¹³ `if(!hasRole(OPERATOR_ROLE, msg.sender)){ ... }`

¹⁴ <https://github.com/crytic/slither/wiki/Detector-Documentation#public-variable-read-in-external-context>

¹⁵ Note that this finding is only applicable to non-upgradeable contracts, as the values have to be set in the constructor, which upgradeable contracts may not have.



- F24.sol: L18
- F24.sol: L19
- F24.sol: L22
- Fiat24Airdrop.sol: L36

6.4.18 Implementations don't implement their interface

The F24Sales and Fiat24Lock don't inherit from their interface contracts (IF24Sales and IFiat24Lock respectively), which would increase clarity.

Code locations:

- F24Sales.sol: L18
- Fiat24Lock.sol: L18

6.4.19 Unused state variables and constant on Fiat24CryptoDeposit contract

The Fiat24CryptoDeposit contract contains the f24AirdropPaused, f24AirdropStart, f24PerUSDC variables, which remain unused and can be removed. The same holds for the misspelled FEE_HUNDRET_PERCENT constant.

Code locations:

- Fiat24CryptoDeposit.sol: L44
- Fiat24CryptoDeposit.sol: L88
- Fiat24CryptoDeposit.sol: L89
- Fiat24CryptoDeposit.sol: L90

6.4.20 Event with duplicated argument

The Fiat24CryptoDeposit__InputTokenOutputTokenSame event of the Fiat24CryptoDeposit contract contains two arguments, which are always instantiated with identical values (as the event name suggests). The second argument is thus superfluous and can be removed, saving gas.

Code locations:

- Fiat24CryptoDeposit.sol: L24

6.4.21 Arguments supplied in the wrong order

The Fiat24CryptoDeposit contract in the moneyExchangeExactOut method calls the getSpread function with mistakenly swapped _outputToken and _inputToken arguments. Due to the fact that the two are handled equivalently by getSprad, this has no functional consequences, but remains confusing for the reader.

Code locations:

- Fiat24CryptoDeposit.sol: L366



6.4.22 Duplicated logic to determine most liquid exchange pool

The `Fiat24CryptoDeposit` contract in the `getPoolFeeOfMostLiquidPool` method performs identical logic for four different Uniswap pools in order to determine the most liquid pool.

Instead of duplicating the code, this would more concisely and robustly be expressed as a loop which gets iterated four times.

Code locations:

- `Fiat24CryptoDeposit.sol`: L463-497

6.4.23 Repeated calls to `getQuote` with identical arguments

The `Fiat24CryptoDeposit` contract in both the `depositETH` and `depositTokenViaEth` methods contains duplicated calls to the `getQuote` function with identical arguments, which is also not envisioned to potentially return meaningful distinct values for identical calls. Instead, this call could be saved in a variable and reused. Indeed, this is exactly what is being done by the `depositTokenViaEth` method, and suggested by commented out code statements in both occurrences.

Code locations:

- `Fiat24CryptoDeposit.sol`: L173
- `Fiat24CryptoDeposit.sol`: L289

6.4.24 Confusing and inconsistent usage of corrections by basis point multiplier

The `Fiat24CryptoDeposit` on numerous occasions needs to correct equations by a factor of 10'000. Confusingly, there are two constants associated with this value, `USDC_DIVISOR` and `XXX24_DIVISOR` (as well as a third unused constant, see issue 6.4.19). Furthermore, however, often the raw constant 10'000 is used. Possibly, it would be clearer if all of these occurrences would be replaced with usage of a single `BASIS_POINT_MULTIPLIER` constant.

Code locations:

- `Fiat24CryptoDeposit.sol`: L42
- `Fiat24CryptoDeposit.sol`: L43
- And various usage sites in `Fiat24CryptoDeposit.sol`

6.4.25 Significant duplicated code in three deposit methods

The `Fiat24CryptoDeposit` contains three deposit methods (`depositETH`, `depositTokenViaUsdc` and `depositTokenViaEth`) that have a substantial amount of code copy-pasted between them (in the case of `depositTokenViaEth`, this includes a trivial variable rename).

It would be much more robust and understandable for the reader if this code was extracted into a commonly used private method instead.

Code locations:

- `Fiat24CryptoDeposit.sol`: L181-198



- Fiat24CryptoDeposit.sol: L240-257
- Fiat24CryptoDeposit.sol: L313-330



7. Limitations

Even though the code has been reviewed carefully on a best-effort basis, undiscovered issues can not be excluded. This report does not consist in a guarantee that no undeclared issues remain, nor should it be interpreted as such.