JPYC contest
Findings & Analysis Report

2022-04-11

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the JPYC smart contract system written in Solidity. The audit contest took place between February 24—February 26 2022.

Wardens

28 Wardens contributed reports to the JPYC contest:

1. hickuphh3
2. leastwood
3. defsec

4. cmichel
5. TerrierLover
6. kyliek
7. gzeon
8. Dravee
9. lllllll
10. peritoflores
11. kenta
12. robee
13. csanuragjain
14. 0xwags
15. Rhynorater
16. securerodd
17. 0x1f8b
18. Omik
19. minhquanym
20. pedroais
21. Ruhum
22. jayjonah8
23. cccz
24. kirk-baird
25. ye0lde
26. rfa
27. sorrynotsorry
28. Tomio

This contest was judged by Jack the Pug.

Final report assembled by liveactionllama.

Summary

The C4 analysis included reports from 24 wardens detailing recommendations that fall under the risk rating of LOW severity or non-critical. There were also reports from 17 wardens describing gas optimizations. All of the submissions presented here are linked back to their original reports.

Notably, 0 vulnerabilities were found during this audit contest that received a risk rating in the category of HIGH or MEDIUM severity.

Scope

The code under review can be found within the C4 JPYC contest repository, and is composed of 23 smart contracts written in the Solidity programming language and includes 1552 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

Low Risk and Non-Critical Issues

For this contest, 24 wardens submitted reports detailing low risk and non-critical issues. The report highlighted below by warden **hickuphh3** received the top score from the judge.

The following wardens also submitted reports: leastwood, defsec, cmichel, TerrierLover, kyliek, gzeon, peritoflores, IIIIIII, Dravee, robee, csanuragjain, 0xwags, Rhynorater, minhquanym, securerodd, Ruhum, kenta, 0x1f8b, jayjonah8, Omik, cccz, kirk-baird, *and* pedroais.

Codebase Impressions & Summary

Overall, code quality for the JPYC contracts is very high. Supporting documentation provided adequate information on design choices made, such as why the UUPS proxy was chosen over the transparent proxy pattern.

The test suite could be easily run, and are rather comprehensive. One thing that stood out and that I'm impressed with was a test checklist. すばらしい! Test coverage was close to 100% with the following functions / branches missed (not significant IMO):

- else case in `EIP712Domain._domainSeparatorV4()`
- upgrading of FiatTokenV2

I could be mistaken, but I would like to mention that while the coverage tool highlights that zero inputs for FiatTokenV2's initializer weren't tested, they actually are in the FiatTokenV2_proxy test file.

The findings I made revolved around the upgradeability aspect of the contracts. I also made recommendations on adding / removing functionality when the contract is paused.

[L-01] Add constructor initializer in implementation contracts

Description

As per OpenZeppelin's (OZ) recommendation, "The guidelines are now to make it impossible for *anyone* to run `initialize` on an implementation contract, by

adding an empty constructor with the `initializer` modifier. So the implementation contract gets initialized automatically upon deployment."

Note that this behaviour is also incorporated the **OZ Wizard** since the UUPS vulnerability discovery: "Additionally, we modified the code generated by the **Wizard 19** to include a constructor that automatically initializes the implementation when deployed."

Furthermore, **this thwarts any attempts to frontrun the initialization tx of the implementation contract**.

Incorporating this change would require inheriting the `Initializable` contract instead of having an explicit `initialized` variable.

Recommended Mitigation Steps

`FiatTokenV1`, `FiatTokenV2` and subsequent implementation contracts should inherit OZ's `Initializable` contract and have the following constructor method:

```
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initiali

// TODO: remove bool internal initialized;
// TODO: remove initialized = true;
contract FiatTokenV1 is
        Initializable
        ...
{
        ...
        /// @custom:oz-upgrades-unsafe-allow constructor
        constructor() initializer {
                // so that users won't accidentally send JPYC to
                blocklisted[address(this)] = true;
        }
}
```

[L-02] Use OZ upgrades (hardhat) plugin to handle proxy deployments and upgrades

Description

The project manually deploys and manages their own proxy contract and upgrades. I strongly recommend that the team use the upgrades plugin from OpenZeppelin instead, because it provides an important feature of validating that the incoming implementations are upgrade safe.

I note that the plugin is part of `package.json` and was imported into **hardhat config** file, but am puzzled why it wasn't used (at least in tests).

Recommended Mitigation Steps

Strongly consider using the **OZ upgrades plugin** to manage deployments. More information about its usage can be found in **their documentation** and **UUPS Proxy** guide.

An example is provided below:

```
const { ethers, upgrades } = require("hardhat");

contract('TestDeploymentAndUpgrade', async function (accounts) {
  const minter = accounts[6];
  const masterMinter = accounts[3];
  const pauser = accounts[4];
  const blocklister = accounts[5];

  it('should do deployment and upgrades', async () => {
    // Deploying
    const FiatTokenV1 = await ethers.getContractFactory("FiatTok
    const instance = await upgrades.deployProxy(FiatTokenV1, [
      'JPY Coin',
      'JPYC',
      'JPY',
      18,
      masterMinter,
      pauser,
      blocklister,
      minter
      ],
      { kind: 'uups' });
      await instance.deployed();

      // Upgrading
```

```
        const FiatTokenV2 = await ethers.getContractFactory("FiatT
        const upgraded = await upgrades.upgradeProxy(instance.addr
      });
  });
```

[L-03] Contracts are not using their OZ upgradeable counterparts

Tools Used

Diffchecker

Description

The non-upgradeable standard version of OpenZeppelin's library, such as `Ownable`, `Pausable`, `Address`, `Context`, `SafeERC20`, `ERC1967Upgrade` etc, are inherited / used by both the proxy and the implementation contracts.

As a result, when attempting to use the upgrades plugin mentioned, the following errors are raised:

```
Error: Contract `FiatTokenV1` is not upgrade safe

contracts/v1/FiatTokenV1.sol:58: Variable `totalSupply_` is assi
  Move the assignment to the initializer
  https://zpl.in/upgrades/error-004

contracts/v1/Pausable.sol:49: Variable `paused` is assigned an i
  Move the assignment to the initializer
  https://zpl.in/upgrades/error-004

contracts/v1/Ownable.sol:28: Contract `Ownable` has a constructo
  Define an initializer instead
  https://zpl.in/upgrades/error-001

contracts/util/Address.sol:186: Use of delegatecall is not allow
  https://zpl.in/upgrades/error-002
```

Having reviewed these errors, none had any adversarial impact:

- `totalSupply_` and `paused` are explictly assigned the default values `0` and `false`
- the implementation contracts utilises the internal `_transferOwnership()` in the initializer, thus transferring ownership to `newOwner` regardless of who the current owner is
- `Address`'s `delegatecall` is only used by the `ERC1967Upgrade` contract. Comparing both the `Address` and `ERC1967Upgrade` contracts against their upgradeable counterparts show similar behaviour (differences are some refactoring done to shift the delegatecall into the `ERC1967Upgrade` contract).

Nevertheless, it would be safer to use the upgradeable versions of the library contracts to avoid unexpected behaviour.

Recommended Mitigation Steps

Where applicable, use the contracts from `@openzeppelin/contracts-upgradeable` instead of `@openzeppelin/contracts`.

[L-04] FiatTokenV1 / V2: Remove `whenNotPaused` modifier from `cancelAuthorization()` and `decreaseAllowance()` functions

Line References

https://github.com/code-423n4/2022-02-jpyc/blob/main/contracts/v1/FiatTokenV1.sol#L409-L414

https://github.com/code-423n4/2022-02-jpyc/blob/main/contracts/v1/FiatTokenV1.sol#L535-L541

https://github.com/code-423n4/2022-02-jpyc/blob/main/contracts/v2/FiatTokenV2.sol#L420-L425

https://github.com/code-423n4/2022-02-jpyc/blob/main/contracts/v2/FiatTokenV2.sol#L558-L564

Description

Just like how `removeMinter()` doesn't have the `whenNotPaused` modifier, it would be very beneficial useful (eg. when a hack / rug pull occurs) to allow users to revoke allowances and cancel authorizations whilst having transfers paused.

Recommended Mitigation Steps

Remove the `whenNotPaused` modifiers for the `cancelAuthorization()` and `decreaseAllowance()` functions.

[L-05] FiatTokenV2: `whitelist()` should be unusable if contract is paused

Line References

https://github.com/code-423n4/2022-02-jpyc/blob/main/contracts/v2/FiatTokenV2.sol#L645

Description

Should the contract be paused, it would be safer to prevent additional addresses from being whitelisted.

Recommended Mitigation Steps

Add the `whenNotPaused` modifier for the `whitelist()` function.

[L-06] Incorrect versioning of `FiatTokenV2`

Line References

https://github.com/code-423n4/2022-02-jpyc/blob/main/contracts/v2/FiatTokenV2.sol#L111

https://github.com/code-423n4/2022-02-jpyc/blob/main/contracts/v2/FiatTokenV2.sol#L114

Description

Since V2 is an upgrade of V1, its versioning should be updated to reflect the upgrade as well. It is important for the correct version to be reflected since it also part of the EIP712 data to be signed, which is used by EIP2612 and EIP3009 for fungible asset transfer authorizations. Authorizations given for outdated versions should rightfully be made invalid.

Recommended Mitigation Steps

```
DOMAIN_SEPARATOR = EIP712.makeDomainSeparator(name, "2");
VERSION = "2";
```

[N-01] Bump OZ packages to `^4.5.0`.

Line Reference

https://github.com/code-423n4/2022-02-jpyc#about-solidittys-version

https://github.com/OpenZeppelin/openzeppelin-contracts/commit/e192fac2769386b7d4b61a3541073ab47bb7723a

Description

The section referenced referred to a change in the `UUPSUpgradeable` and `ERC1967Upgrade` contracts that are only included in the latest package version of the OZ npm packages . However, the version specified in `package-lock.json` is `4.4.1`, which does **not** include this change (and hence I assume was manually imported).

I can verify that the installed version is `4.4.1` by executing the following commands:

```
yarn install
yarn list @openzeppelin/contracts
```

Recommended Mitigation Steps

Update the versions of `@openzeppelin/contracts` and `@openzeppelin/contracts-upgradeable` to be the latest in `package.json`. I also recommend double checking the versions of other dependencies as a precaution, as they may include important bug fixes.

0xywzx (JPYC) commented:

> Thank you for your report. It is very helpful. ありがとうございます！

L01

> Your solution looks good. We will change the code to use initializable for constructor.

L02

> Since we don't use the hardhat plugin in the actual deployment, we didn't use it in the test code. By the way, if you know how to deploy in a secure way, I would appreciate it if you could us me know.

L03

> We have compliant USDC, so the code is looks like. However, 'functionDelegateCall()' was deleted from OZ upgradable contract, so we will check it.

L04

> We didn't include 'whenNotPaused()' for whitelist because it doesn't have a significant impact on the contract itself.

L5 & L6

> We agree with your advice.

thurendous (JPYC) resolved and commented:

> Incorrect versioning of FiatTokenV2

> Fixed and thanks!

> The change can be viewed here.

Gas Optimizations

For this contest, 17 wardens submitted reports detailing gas optimizations. The report highlighted below by warden **Dravee** received the top score from the judge.

The following wardens also submitted reports: kenta, IIIIIII, ye0lde, defsec, gzeon, robee, Omik, securerodd, TerrierLover, pedroais, rfa, sorrynotsorry, 0x1f8b, peritoflores, Rhynorater, *and* Tomio.

Foreword

- **Storage-reading optimizations**

> The code can be optimized by minimising the number of SLOADs. SLOADs are expensive (100 gas) compared to MLOADs/MSTOREs (3 gas). In the paragraphs below, please see the `@audit-issue` tags in the pieces of code's comments for more information about SLOADs that could be saved by caching the mentioned **storage** variables in **memory** variables.

- **Unchecking arithmetics operations that can't underflow/overflow**

> Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation, or the operation doesn't depend on user input), some gas can be saved by using an `unchecked` block: https://docs.soliditylang.org/en/v0.8.10/control-structures.html#checked-or-unchecked-arithmetic

- `@audit` **tags**

> The code is annotated at multiple places with `//@audit` comments to pinpoint the issues. Please, pay attention to them for more details.

[G-01] File: ERC1967Proxy.sol

function _implementation()

```
File: ERC1967Proxy.sol
30:     function _implementation() internal view virtual overric
31:         return ERC1967Upgrade._getImplementation(); //@audit
32:     }
```

Using both named returns and a return statement isn't necessary

Removing unused named returns variables can reduce gas usage (MSTOREs/MLOADs) and improve code clarity. To save gas and improve code quality: consider using only one of those.

[G-02] File: UUPSUpgradeable.sol

function upgradeToAndCall()

```
File: UUPSUpgradeable.sol
095:     function upgradeToAndCall(address newImplementation, by
096:         external
097:         payable
098:         virtual
099:         onlyProxy
100:     {
101:         _authorizeUpgrade(newImplementation);
102:         _upgradeToAndCallUUPS(newImplementation, data, true
103:     }
```

Use `calldata` instead of `memory` for external functions where the function argument is read-only

Here, `bytes memory data` should be `bytes calldata data`

[G-03] File: Blocklistable.sol

function updateBlocklister()

```
File: Blocklistable.sol
90:     function updateBlocklister(address _newBlocklister) exte
```

```
91:         require(
92:             _newBlocklister != address(0),
93:             "Blocklistable: new blocklister is the zero addr
94:         );
95:         blocklister = _newBlocklister;
96:         emit BlocklisterChanged(blocklister); //@audit emitt
97:     }
```

Emitting a storage value (`blocklister` vs `_newBlocklister`)

I suggest going from

```
96:         emit BlocklisterChanged(blocklister);
```

to

```
96:         emit BlocklisterChanged(_newBlocklister);
```

[G-04] File: EIP3009.sol

function _requireValidAuthorization()

```
File: EIP3009.sol
219:     function _requireValidAuthorization(
220:         address authorizer,
221:         bytes32 nonce,
222:         uint256 validAfter,
223:         uint256 validBefore
224:     ) private view {
225:         require(
226:             block.timestamp > validAfter, //@audit make it
227:             "FEIP3009: authorization is not yet valid"
228:         );
229:         require(
230:             block.timestamp < validBefore,//@audit make it
231:             "EIP3009: authorization is expired"
232:         );
233:         _requireUnusedAuthorization(authorizer, nonce);
```

```
234:     }
```

Non-strict inequalities are cheaper than strict ones (1)

It's possible to save 3 gas on `block.timestamp > validAfter` by making it inclusive: `block.timestamp >= validAfter`. I believe it wouldn't change much functionally here.

Non-strict inequalities are cheaper than strict ones (2)

It's possible to save 3 gas on `block.timestamp < validBefore` by making it inclusive: `block.timestamp <= validBefore`. I believe it wouldn't change much functionally here.

[G-05] File: FiatTokenV1.sol

function initialize()

Use `tokenName` instead of `name`

This can save 2 SLOADs (around 200 gas). Replace:

```
File: FiatTokenV1.sol
96:         name = tokenName;
...
105:         DOMAIN_SEPARATOR = EIP712.makeDomainSeparator(name,
106:         CHAIN_ID = block.chainid;
107:         NAME = name;//@audit name SLOAD
```

with

```
105:         DOMAIN_SEPARATOR = EIP712.makeDomainSeparator(toker
...
107:         NAME = tokenName;//@audit tokenName MLOAD
```

Additionally, `EIP712.makeDomainSeparator()` signature is as such:

```
File: EIP712.sol
40:     function makeDomainSeparator(string memory name, string
```

As the first argument is a `memory` and not a `storage`, passing the storage `name` instead of the memory `tokenName` would imply another copy in memory

function mint()

```
File: FiatTokenV1.sol
127:     function mint(address _to, uint256 _amount)
...
139:         require(
140:             _amount <= mintingAllowedAmount,
...
146:         minterAllowed[msg.sender] = mintingAllowedAmount -
```

Unchecked block L146

This line can't underflow due to L140. Therefore, it should be wrapped in an `unchecked` block.

function transferFrom()

```
File: FiatTokenV1.sol
258:     function transferFrom(
...
271:         require(
272:             value <= allowed[from][msg.sender], //@audit al
273:             "ERC20: transfer amount exceeds allowance"
274:         );
275:         _transfer(from, to, value);
276:         allowed[from][msg.sender] = allowed[from][msg.sende
```

Unchecked block L276

This line can't underflow due to L272. Therefore, it should be wrapped in an `unchecked` block.

Cache `allowed[from][msg.sender]`

Caching this in memory can save around 1 SLOAD. This is similar to an already implemented optimization in `L361: function burn()` for `uint256 balance = balances[msg.sender]`

function _transfer()

```
File: FiatTokenV1.sol
304:     function _transfer(
...
311:         require(
312:             value <= balances[from],//@audit SLOAD 1
313:             "ERC20: transfer amount exceeds balance"
314:         );
315:
316:         balances[from] = balances[from] - value; //@audit s
317:         balances[to] = balances[to] + value;
```

Unchecked block L316

This line can't underflow due to L312. Therefore, it should be wrapped in an `unchecked` block.

Cache `balances[from]`

Caching this in memory can save around 1 SLOAD. This is similar to an already implemented optimization in `L361: function burn()` for `uint256 balance = balances[msg.sender]`

function burn()

```
File: FiatTokenV1.sol
361:     function burn(uint256 _amount)
...
369:         require(balance >= _amount, "FiatToken: burn amount
370:
371:         totalSupply_ = totalSupply_ - _amount;
```

```
372:         balances[msg.sender] = balance - _amount; //@audit
```

Unchecked block L372

This line can't underflow due to L369. Therefore, it should be wrapped in an `unchecked` block.

function updateMasterMinter()

```
File: FiatTokenV1.sol
377:     function updateMasterMinter(address _newMasterMinter) e
378:         require(
379:             _newMasterMinter != address(0),
380:             "FiatToken: new masterMinter is the zero addres
381:         );
382:         masterMinter = _newMasterMinter;
383:         emit MasterMinterChanged(masterMinter); //@audit em
384:     }
```

Emitting a storage value (`masterMinter` vs `_newMasterMinter`)

I suggest going from

```
383:         emit MasterMinterChanged(masterMinter);
```

to

```
383:         emit MasterMinterChanged(_newMasterMinter);
```

function _decreaseAllowance()

```
File: FiatTokenV1.sol
440:     function _decreaseAllowance(
...
445:         require(
446:             decrement <= allowed[owner][spender],//@audit S
```

```
447:                 "ERC20: decreased allowance below zero"
448:             );
449:             _approve(owner, spender, allowed[owner][spender] -
450:         }
```

Unchecked block L449

This line can't underflow due to L446. Therefore, it should be wrapped in an `unchecked` block.

Cache `allowed[owner][spender]`

Caching this in memory can save around 1 SLOAD. This is similar to an already implemented optimization in `L361: function burn()` for `uint256 balance = balances[msg.sender]`

[G-06] File: FiatTokenV2.sol

function initialize()

Use `tokenName` instead of `name`

This can save 2 SLOADs (around 200 gas). The explanation is the same as Use `tokenName` instead of `name`

function mint()

```
File: FiatTokenV2.sol
133:     function mint(address _to, uint256 _amount)
...
146:         require(
147:             _amount <= mintingAllowedAmount,
...
153:         minterAllowed[msg.sender] = mintingAllowedAmount -
```

Unchecked block L153

This line can't underflow due to L147. Therefore, it should be wrapped in an `unchecked` block.

function transferFrom()

```
File: FiatTokenV2.sol
266:     function transferFrom(
...
280:         require(
281:             value <= allowed[from][msg.sender],//@audit all
...
285:         allowed[from][msg.sender] = allowed[from][msg.sende
```

Unchecked block L285

This line can't underflow due to L281. Therefore, it should be wrapped in an `unchecked` block.

Cache `allowed[from][msg.sender]`

Caching this in memory can save around 1 SLOAD.

function _transfer()

```
File: FiatTokenV2.sol
314:     function _transfer(
...
321:         require(
322:             value <= balances[from],//@audit SLOAD 1
323:             "ERC20: transfer amount exceeds balance"
...
326:         balances[from] = balances[from] - value; //@audit s
```

Unchecked block L326

This line can't underflow due to L322. Therefore, it should be wrapped in an `unchecked` block.

Cache `balances[from]`

Caching this in memory can save around 1 SLOAD.

function burn()

```
File: FiatTokenV2.sol
371:     function burn(uint256 _amount)
...
379:         require(balance >= _amount, "FiatToken: burn amount
...
382:         balances[msg.sender] = balance - _amount; //@audit
```

Unchecked block L382

This line can't underflow due to L379. Therefore, it should be wrapped in an `unchecked` block.

function updateMasterMinter()

```
File: FiatTokenV2.sol
387:     function updateMasterMinter(address _newMasterMinter) e
388:         require(
389:             _newMasterMinter != address(0),
390:             "FiatToken: new masterMinter is the zero addres
391:         );
392:         masterMinter = _newMasterMinter;
393:         emit MasterMinterChanged(masterMinter);
394:     }
```

Emitting a storage value (`masterMinter` vs `_newMasterMinter`)

I suggest going from

```
393:         emit MasterMinterChanged(masterMinter);
```

to

```
393:         emit MasterMinterChanged(_newMasterMinter);
```

function _decreaseAllowance()

```
File: FiatTokenV2.sol
451:     function _decreaseAllowance(
452:         address owner,
453:         address spender,
454:         uint256 decrement
455:     ) internal override {
456:         require(
457:             decrement <= allowed[owner][spender],//@audit S
458:             "ERC20: decreased allowance below zero"
459:         );
460:         _approve(owner, spender, allowed[owner][spender] -
461:     }
```

Unchecked block L460

This line can't underflow due to L457. Therefore, it should be wrapped in an `unchecked` block.

Cache `allowed[owner][spender]`

Caching this in memory can save around 1 SLOAD.

function updateWhitelister()

```
File: FiatTokenV2.sol
659:     function updateWhitelister(address _newWhitelister) ext
660:         require(
661:             _newWhitelister != address(0),
662:             "Whitelistable: new whitelister is the zero adc
663:         );
664:         whitelister = _newWhitelister;
665:         emit WhitelisterChanged(whitelister); //@audit emit
666:     }
```

Emitting a storage value (`whitelister` vs `_newWhitelister`)

I suggest going from

```
665:         emit WhitelisterChanged(whitelister);
```

to

```
665:         emit WhitelisterChanged(_newWhitelister);
```

[G-07] General Recommendations

Variables

No need to explicitly initialize variables with default values

If a variable is not set/initialized, it is assumed to have the default value (`0` for `uint` , `false` for `bool` , `address(0)` for address...). Explicitly initializing it with its default value is an anti-pattern and wastes gas.

As an example: `for (uint256 i = 0; i < numIterations; ++i) {` should be replaced with `for (uint256 i; i < numIterations; ++i) {`

Instances include:

```
v1/FiatTokenV1.sol:58:    uint256 internal totalSupply_ = 0;
v1/Pausable.sol:49:    bool public paused = false;
v2/FiatTokenV2.sol:58:    uint256 internal totalSupply_ = 0;
```

I suggest removing explicit initializations for default values.

Pre-increments cost less gas compared to post-increments

Comparisons

`> 0` is less efficient than `!= 0` for unsigned integers (with proof)

`!= 0` costs less gas compared to `> 0` for unsigned integers in `require` statements with the optimizer enabled (6 gas)

Proof: While it may seem that `> 0` is cheaper than `!=`, this is only true without the optimizer enabled and outside a require statement. If you enable the optimizer at 10k AND you're in a `require` statement, this will save gas. You can see this tweet for more proofs: https://twitter.com/gzeon/status/1485428085885640706

I suggest changing `> 0` with `!= 0` here:

```
v1/FiatTokenV1.sol:136:        require(_amount > 0, "FiatToken:
v1/FiatTokenV1.sol:368:        require(_amount > 0, "FiatToken:
v2/FiatTokenV2.sol:143:        require(_amount > 0, "FiatToken:
v2/FiatTokenV2.sol:378:        require(_amount > 0, "FiatToken:
```

Also, please enable the Optimizer.

Visibility

Errors

Reduce the size of error messages (Long revert Strings)

Shortening revert strings to fit in 32 bytes will decrease deployment time gas and will decrease runtime gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional mstore, along with additional overhead for computing memory offset, etc.

For strings > 32 bytes, see issue page for details.

I suggest shortening the revert strings to fit in 32 bytes, or that using custom errors as described next.

Use Custom Errors instead of Revert Strings to save Gas

Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met)

Source: https://blog.soliditylang.org/2021/04/21/custom-errors/:

> Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g., `revert("Insufficient funds.");`), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them.

Custom errors are defined using the `error` statement, which can be used inside and outside of contracts (including interfaces and libraries).

For instances included, see issue page for details.

I suggest replacing revert strings with custom errors.

thurendous (JPYC) commented:

> function _implementation()
> File: UUPSUpgradeable.sol

> We used OpenZeppelin's library and it is like this. Should we change this?

> File: Blocklistable.sol
> Emitting a storage value (blocklister vs _newBlocklister)

> We will fix this.

> File: EIP3009.sol
> Non-strict inequalities are cheaper than strict ones (1)

> We are not sure about this. Is this true?

> File: FiatTokenV1.sol
> function initialize()

> This can be true. But the initialize function is rarely called. We keep it this way.

> File: FiatTokenV1.sol
> File: FiatTokenV2.sol
> Unchecked block L146
> Unchecked block L276
> Unchecked block L316
> Unchecked block L372
> Unchecked block L449
> Cache allowed[from][msg.sender]
> Cache balances[from]

> We maybe will fix this after checking in detail.

> function updateMasterMinter()
> Emitting a storage value (masterMinter vs _newMasterMinter)

> We will fix this.

> 0 is less efficient than != 0 for unsigned integers (with proof)

> Thank you for the proof.

> Reduce the size of error messages (Long revert Strings)

> We maybe will fix this after checking in detail. Do you suggest do this strongly?

thurendous (JPYC) resolved and commented:

> No need to explicitly initialize variables with default values
> Cache allowedfrom
> Cache allowed[from][msg.sender]
> Cache balances[from]
> Cache allowed[owner][spender]

> Above issues were fixed and thanks!

> Final change can be viewed here.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top