XDEFI contest
Findings & Analysis Report

2022-02-10

Table of contents

- Gas Optimizations (28)
- Disclosures

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 code 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 code contest outlined in this document, C4 conducted an analysis of XDEFI contest smart contract system written in Solidity. The code contest took place between January 4—January 6 2022.

Wardens

41 Wardens contributed reports to the XDEFI contest:

1. WatchPug (jtp and ming)
2. onewayfunction
3. sirhashalot
4. cmichel
5. kenzo
6. Fitraldys
7. Tomio
8. Czar102
9. cccz
10. tqts
11. egjlmn1
12. robee

13. pedroais
14. Dravee
15. defsec
16. TomFrenchBlockchain
17. PierrickGT
18. OriDabush
19. leastwood
20. MaCree
21. 0xsanson
22. rfa
23. Jujic
24. hack3r-0m
25. agusduha
26. ye0lde
27. GiveMeTestEther
28. wuwe1
29. certora
30. jayjonah8
31. danb
32. gpersoon
33. harleythedog
34. StErMi
35. ACai
36. bitbopper
37. mtz
38. p4st13r4
39. saian
40. BouSalman

This contest was judged by Ivo Georgiev.

Final report assembled by captainmango and dzhawsh.

Summary

The C4 analysis yielded an aggregated total of 13 unique vulnerabilities and 50 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity, 1 received a risk rating in the category of MEDIUM severity, and 10 received a risk rating in the category of LOW severity.

C4 analysis also identified 9 non-critical recommendations and 28 gas optimizations.

Scope

The code under review can be found within the C4 XDEFI contest repository, and is composed of 2 smart contracts written in the Solidity programming language and includes 539 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.

High Risk Findings (2)

[H-01] Malicious early user/attacker can malfunction the contract and even freeze users' funds in edge cases

Submitted by WatchPug

https://github.com/XDeFi-tech/xdefi-distribution/blob/3856a42df295183b40c6eee89307308f196612fe/contracts/XDEFIDistribution.sol#L151-L151

```
    _pointsPerUnit += ((newXDEFI * _pointsMultiplier) / totalUnitsCa
```

In the current implementation, `_pointsPerUnit` can be changed in `updateDistribution()` which can be called by anyone.

A malicious early user can `lock()` with only `1 wei` of XDEFI and makes `_pointsPerUnit` to be very large, causing future users not to be able to `lock()` and/or `unlock()` anymore due to overflow in arithmetic related to `_pointsMultiplier`.

As a result, the contract can be malfunctioning and even freeze users' funds in edge cases.

Proof of Concept

Given:

- bonusMultiplierOf[30 days] = 100
- Alice `lock() 1 wei` of XDEFI for 30 days as the first user of the contract. Got `1` units, and `totalUnits` now is `1`;
- Alice sends `170141183460469 wei` of `XDEFI` to the contract and calls `updateDistribution()`:

```
_pointsPerUnit += ((170141183460469 * 2**128) / 1);
```

3. Bob tries to `lock() 1,100,000 * 1e18` of `XDEFI` for 30 days, the tx will fail, as `_pointsPerUnit * units` overlows;
4. Bob `lock() 1,000,000 * 1e18` of `XDEFI` for 30 days;
5. The rewarder sends `250,000 * 1e18` of `XDEFI` to the contract and calls `updateDistribution()`:

```
_pointsPerUnit += ((250_000 * 1e18 * 2**128) / (1_000_000 * 1e18
```

6. 30 days later, Bob tries to call `unlock()`, the tx will fail, as `_pointsPerUnit * units` overflows.

Recommended Mitigation Steps

Uniswap v2 solved a similar problem by sending the first 1000 lp tokens to the zero address.

The same solution should work here, i.e., on constructor set an initial amount (like 1e8) for `totalUnits`

https://github.com/XDeFi-tech/xdefi-distribution/blob/3856a42df295183b40c6eee89307308f196612fe/contracts/XDEFIDistribution.sol#L39-L44

```
constructor (address XDEFI_, string memory baseURI_, uint256 zer
    require((XDEFI = XDEFI_) != address(0), "INVALID_TOKEN");
    owner = msg.sender;
    baseURI = baseURI_;
    _zeroDurationPointBase = zeroDurationPointBase_;

    totalUnits = 100_000_000;
}
```

deluca-mike (XDEFI) confirmed:

> This is a great catch! I just tested it:

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

const totalSupply = '240000000000000000000000000';

const toWei = (value, add = 0, sub = 0) => (BigInt(value) * 1_0(

describe("XDEFIDistribution", () => {
    it("Can overflow _pointsPerUnit", async () => {
        const [god, alice, bob] = await ethers.getSigners();

        const XDEFI = await (await (await ethers.getContractFact
        const XDEFIDistribution = await (await (await ethers.get

        // Give each account 2,000,000 XDEFI
        await (await XDEFI.transfer(alice.address, toWei(2_000_(
        await (await XDEFI.transfer(bob.address, toWei(2_000_00(

        // bonusMultiplierOf[30 days] = 100
        await (await XDEFIDistribution.setLockPeriods([2592000],

        // 1. Alice lock() 1 wei of XDEFI for 30 days as the fir
        await (await XDEFI.connect(alice).approve(XDEFIDistribut
        await (await XDEFIDistribution.connect(alice).lock(1, 25
        expect(await XDEFIDistribution.balanceOf(alice.address))
        const nft1 = (await XDEFIDistribution.tokenOfOwnerByInde
        expect((await XDEFIDistribution.positionOf(nft1)).units)

        // 2. Alice sends 170141183460469 wei of XDEFI to the cc
        await (await XDEFI.connect(alice).transfer(XDEFIDistribu
        await (await XDEFIDistribution.connect(alice).updateDist

        // 3. Bob tries to lock() 1,100,000 * 1e18 of XDEFI for
        await (await XDEFI.connect(bob).approve(XDEFIDistributic
        await expect(XDEFIDistribution.connect(bob).lock(toWei(1

        // 4. Bob lock() 1,000,000 * 1e18 of XDEFI for 30 days
        await (await XDEFI.connect(bob).approve(XDEFIDistributic
        await (await XDEFIDistribution.connect(bob).lock(toWei(1
        expect(await XDEFIDistribution.balanceOf(bob.address)).t
        const nft2 = (await XDEFIDistribution.tokenOfOwnerByInde
        expect((await XDEFIDistribution.positionOf(nft2)).units)
```

```
        // 5. The rewarder sends 250,000 * 1e18 of XDEFI to the
        await (await XDEFI.transfer(XDEFIDistribution.address, t
        await (await XDEFIDistribution.updateDistribution()).wai

        // 6. 30 days later, Bob tries to call unlock(), the tx
        await hre.ethers.provider.send('evm_increaseTime', [2592
        await expect(XDEFIDistribution.connect(bob).unlock(nft2,
    });
});
```

> While I do like the suggestion to to `totalUnits = 100_000_000;` in the constructor, it will result "uneven" numbers due to the `totalUnits` offset. I wonder if I can resolve this but just reducing `_pointsMultiplier` to `uint256(2**96)` as per https://github.com/ethereum/EIPs/issues/1726#issuecomment-472352728.

deluca-mike (XDEFI) commented:

> OK, I think I can solve this with `_pointsMultiplier = uint256(2**72)`:

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

const totalSupply = '2400000000000000000000000';

const toWei = (value, add = 0, sub = 0) => (BigInt(value) * 1_0(

describe("XDEFIDistribution", () => {
    it("Can overflow _pointsPerUnit", async () => {
        const [god, alice, bob] = await ethers.getSigners();

        const XDEFI = await (await (await ethers.getContractFact
        const XDEFIDistribution = await (await (await ethers.get

        // Give each account 100M XDEFI
        await (await XDEFI.transfer(alice.address, toWei(100_00(
        await (await XDEFI.transfer(bob.address, toWei(100_000_(

        // bonusMultiplierOf[30 days] = 255
        await (await XDEFIDistribution.setLockPeriods([2592000],

        // 1. Alice lock() 1 wei of XDEFI for 30 days as the fir
        await (await XDEFI.connect(alice).approve(XDEFIDistribut
```

```
            await (await XDEFIDistribution.connect(alice).lock(1, 25
            expect(await XDEFIDistribution.balanceOf(alice.address))
            const nft1 = (await XDEFIDistribution.tokenOfOwnerByInde
            expect((await XDEFIDistribution.positionOf(nft1)).units)

            // 2. Alice sends 100M XDEFI minus 1 wei to the contract
            await (await XDEFI.connect(alice).transfer(XDEFIDistribu
            await (await XDEFIDistribution.connect(alice).updateDist

            // 3. Bob can lock() 100M XDEFI for 30 days
            await (await XDEFI.connect(bob).approve(XDEFIDistributic
            await (await XDEFIDistribution.connect(bob).lock(toWei(1
            expect(await XDEFIDistribution.balanceOf(bob.address)).t
            const nft2 = (await XDEFIDistribution.tokenOfOwnerByInde
            expect((await XDEFIDistribution.positionOf(nft2)).units)

            // 4. The rewarder sends 40M XDEFI to the contract and c
            await (await XDEFI.transfer(XDEFIDistribution.address, t
            await (await XDEFIDistribution.updateDistribution()).wai

            // 5. 30 days later, Bob can call unlock()
            await hre.ethers.provider.send('evm_increaseTime', [2592
            await (await XDEFIDistribution.connect(bob).unlock(nft2,
        });
    });
```

deluca-mike (XDEFI) commented:

> In the **release candidate contract**, in order to preserve the math (formulas), at the cost of some accuracy, we went with a `_pointsMultiplier` **of 72 bits**.

> Also, a **minimum units locked** is enforced, to prevent "dust" from creating a very very high `_pointsPerUnit`.

> Tests were written in order to stress test the contract against the above extreme cases.

> Further, a "no-going-back" **emergency mode setter** was implemented that allows (but does not force) users to **withdraw only their deposits** without any of the funds distribution math from being expected, in the event that some an edge case does arise.

Ivshti (Judge) commented:

> fantastic finding, agreed with the proposed severity!

> The sponsor fixes seem adequate: a lower `_poinsMultiplier`, a minimum lock and an emergency mode that disables reward math, somewhat similar to emergency withdraw functions in contracts like masterchef.

[H-02] The reentrancy vulnerability in _safeMint can allow an attacker to steal all rewards

Submitted by cccz, also found by cmichel, Fitraldys, kenzo, onewayfunction, and tqts

There is a reentrancy vulnerability in the _safeMint function

```
    function _safeMint(
        address to,
        uint256 tokenId,
        bytes memory _data
    ) internal virtual {
        _mint(to, tokenId);
        require(
            _checkOnERC721Received(address(0), to, tokenId, _data),
            "ERC721: transfer to non ERC721Receiver implementer"
        );
    }
    ...
    function _checkOnERC721Received(
        address from,
        address to,
        uint256 tokenId,
        bytes memory _data
    ) private returns (bool) {
        if (to.isContract()) {
            try IERC721Receiver(to).onERC721Received(_msgSender(), f
                return retval == IERC721Receiver.onERC721Received.se
```

The lock function changes the totalDepositedXDEFI variable after calling the _safeMint function

```
    function lock(uint256 amount_, uint256 duration_, address destir
        // Lock the XDEFI in the contract.
```

```
        SafeERC20.safeTransferFrom(IERC20(XDEFI), msg.sender, addres

        // Handle the lock position creation and get the tokenId of
        return _lock(amount_, duration_, destination_);
    }
    ...
        function _lock(uint256 amount_, uint256 duration_, address c
        // Prevent locking 0 amount in order generate many score-les
        require(amount_ != uint256(0) && amount_ <= MAX_TOTAL_XDEFI_

        // Get bonus multiplier and check that it is not zero (which
        uint8 bonusMultiplier = bonusMultiplierOf[duration_];
        require(bonusMultiplier != uint8(0), "INVALID_DURATION");

        // Mint a locked staked position NFT to the destination.
        _safeMint(destination_, tokenId_ = _generateNewTokenId(_getI

        // Track deposits.
        totalDepositedXDEFI += amount_;
```

Since the updateDistribution function does not use the noReenter modifier, the attacker can re-enter the updateDistribution function in the _safeMint function. Since the value of totalDepositedXDEFI is not updated at this time, the _pointsPerUnit variable will become abnormally large.

```
        function updateDistribution() external {
            uint256 totalUnitsCached = totalUnits;

            require(totalUnitsCached> uint256(0), "NO_UNIT_SUPPLY");

            uint256 newXDEFI = _toUint256Safe(_updateXDEFIBalance());

            if (newXDEFI == uint256(0)) return;

            _pointsPerUnit += ((newXDEFI * _pointsMultiplier) / total

            emit DistributionUpdated(msg.sender, newXDEFI);
        }
        ...
        function _updateXDEFIBalance() internal returns (int256 newFu
            uint256 previousDistributableXDEFI = distributableXDEFI;
            uint256 currentDistributableXDEFI = distributableXDEFI =
```

```
        return _toInt256Safe(currentDistributableXDEFI)-_toInt256
    }
```

If the attacker calls the lock function to get the NFT before exploiting the reentrance vulnerability, then the unlock function can be called to steal a lot of rewards, and the assets deposited by the user using the reentrance vulnerability can also be redeemed by calling the unlock function. Since the unlock function calls the _updateXDEFIBalance function, the attacker cannot steal the assets deposited by the user

```
function unlock(uint256 tokenId_, address destination_) external
    // Handle the unlock and get the amount of XDEFI eligible to
    amountUnlocked_ = _unlock(msg.sender, tokenId_);

    // Send the the unlocked XDEFI to the destination.
    SafeERC20.safeTransfer(IERC20(XDEFI), destination_, amountUn

    // NOTE: This needs to be done after updating `totalDeposite
    _updateXDEFIBalance();
}
...
function _unlock(address account_, uint256 tokenId_) internal re
    // Check that the account is the position NFT owner.
    require(ownerOf(tokenId_) == account_, "NOT_OWNER");

    // Fetch position.
    Position storage position = positionOf[tokenId_];
    uint96 units = position.units;
    uint88 depositedXDEFI = position.depositedXDEFI;
    uint32 expiry = position.expiry;

    // Check that enough time has elapsed in order to unlock.
    require(expiry != uint32(0), "NO_LOCKED_POSITION");
    require(block.timestamp >= uint256(expiry), "CANNOT_UNLOCK")

    // Get the withdrawable amount of XDEFI for the position.
    amountUnlocked_ = _withdrawableGiven(units, depositedXDEFI,

    // Track deposits.
    totalDepositedXDEFI -= uint256(depositedXDEFI);

    // Burn FDT Position.
    totalUnits -= units;
```

```
        delete positionOf[tokenId_];

        emit LockPositionWithdrawn(tokenId_, account_, amountUnlocke
    }
    ...
    function _withdrawableGiven(uint96 units_, uint88 depositedXDEFI
        return
            (
                _toUint256Safe(
                    _toInt256Safe(_pointsPerUnit * uint256(units_))
                    pointsCorrection_
                ) / _pointsMultiplier
            ) + uint256(depositedXDEFI_);
    }
```

Proof of Concept

https://github.com/XDeFi-tech/xdefi-distribution/blob/v1.0.0-beta.0/contracts/XDEFIDistribution.sol#L253-L281

Recommended Mitigation Steps

```
-     function updateDistribution() external  {
+     function updateDistribution() external  noReenter {
```

deluca-mike (XDEFI) resolved:

> Valid and a big issue. However, due to other recommendations, I will not solve it this way. Instead, `updateDistribution()` will be called at the start of every lock/unlock function (so it can't have a `noReenter` modifier), and the `_safeMint` calls will be moved to the end of their respective operations to prevent the effect of the re-entrancy (i.e. position will created with a `_pointsPerUnit` before a re-entering from `_safeMint` can affect it). Tests will be added to show this is not longer possible.

deluca-mike (XDEFI) commented:

> In our release candidate contract, as mentioned above, `updateDistribution()` is called before each locking and unlocking function, via a

> `updatePointsPerUnitAtStart` **modifier**, and thus, `updateDistribution()` is now a public fucntion, and since it is used by other functions, cannot be behind a `noReenter`.

> See:
>
> - `lock`
> - `lockWithPermit`
> - `relock`
> - `unlock`
> - `relockBatch`
> - `unlockBatch`

> Also, **a test was written** to ensure that this is no longer exploitable, and that the contract behaves properly if a re-entrancy call `updateDistribution()`.

Ivshti (Judge) commented:

> Agreed with the severity.

> Resolution of reordering the calls seems to be adequate

Medium Risk Findings (1)

[M-01] `_safeMint` Will Fail Due To An Edge Case In Calculating `tokenId` Using The `_generateNewTokenId` Function

Submitted by leastwood, also found by cmichel, cmichel, egjlmn1, kenzo, MaCree, onewayfunction, sirhashalot, and WatchPug

Impact

NFTs are used to represent unique positions referenced by the generated `tokenId`. The `tokenId` value contains the position's score in the upper 128 bits and the index wrt. the token supply in the lower 128 bits.

When positions are unlocked after expiring, the relevant position stored in the `positionOf` mapping is deleted, however, the NFT is not. The `merge()` function is used to combine points in unlocked NFTs, burning the underlying NFTs upon merging. As a result, `_generateNewTokenId()` may end up using the same `totalSupply()` value, causing `_safeMint()` to fail if the same `amount_` and `duration_` values are used.

This edge case only occurs if there is an overlap in the `points_` and `totalSupply() + 1` values used to generate `tokenId`. As a result, this may impact a user's overall experience while interacting with the `XDEFI` protocol, as some transactions may fail unexpectedly.

Proof of Concept

```
    function _lock(uint256 amount_, uint256 duration_, address desti
        // Prevent locking 0 amount in order generate many score-les
        require(amount_ != uint256(0) && amount_ <= MAX_TOTAL_XDEFI_

        // Get bonus multiplier and check that it is not zero (which
        uint8 bonusMultiplier = bonusMultiplierOf[duration_];
        require(bonusMultiplier != uint8(0), "INVALID_DURATION");

        // Mint a locked staked position NFT to the destination.
        _safeMint(destination_, tokenId_ = _generateNewTokenId(_getI

        // Track deposits.
        totalDepositedXDEFI += amount_;

        // Create Position.
        uint96 units = uint96((amount_ * uint256(bonusMultiplier)) /
        totalUnits += units;
        positionOf[tokenId_] =
            Position({
                units: units,
                depositedXDEFI: uint88(amount_),
                expiry: uint32(block.timestamp + duration_),
                created: uint32(block.timestamp),
                bonusMultiplier: bonusMultiplier,
                pointsCorrection: -_toInt256Safe(_pointsPerUnit * ur
            });
```

```
        emit LockPositionCreated(tokenId_, destination_, amount_, du
    }


    function _generateNewTokenId(uint256 points_) internal view retu
        // Points is capped at 128 bits (max supply of XDEFI for 10
        return (points_ << uint256(128)) + uint128(totalSupply() + 1
    }


    function merge(uint256[] memory tokenIds_, address destination_)
        uint256 count = tokenIds_.length;
        require(count > uint256(1), "MIN_2_TO_MERGE");

        uint256 points;

        // For each NFT, check that it belongs to the caller, burn i
        for (uint256 i; i < count; ++i) {
            uint256 tokenId = tokenIds_[i];
            require(ownerOf(tokenId) == msg.sender, "NOT_OWNER");
            require(positionOf[tokenId].expiry == uint32(0), "POSITI

            _burn(tokenId);

            points += _getPointsFromTokenId(tokenId);
        }

        // Mine a new NFT to the destinations, based on the accumula
        _safeMint(destination_, tokenId_ = _generateNewTokenId(point
    }
```

Recommended Mitigation Steps

Consider replacing `totalSupply()` in `_generateNewTokenId()` with an internal counter. This should ensure that `_generateNewTokenId()` always returns a unique `tokenId` that is monotomically increasing .

deluca-mike (XDEFI) confirmed:

> In the release candidate contract, `_generateNewTokenId` now used an internal `_tokensMinted` variable instead of `totalSupply()`, as seen here. Ivshti

> **(Judge) commented:** Agreed with sponsor

> As for mitigation, the new way to generate token IDs seems cleaner, but more gas consuming

Low Risk Findings (10)

- [L-01] Distribution Updates Can Be Gamed *Submitted by leastwood, also found by cmichel, danb, egjlmn1, gpersoon, hack3r-0m, harleythedog, kenzo, StErMi, and WatchPug*
- [L-02] setLockPeriods function lack of input validation *Submitted by cccz, also found by agusduha, certora, hack3r-0m, jayjonah8, Jujic, Tomio, WatchPug, and ye0lde*
- [L-03] Owner can steal XDEFI without any capital risk *Submitted by onewayfunction*
- [L-04] Possible profitability manipulations *Submitted by Czar102*
- [L-06] Assert instead require to validate user inputs *Submitted by robee, also found by egjlmn1 and WatchPug*
- [L-07] `_zeroDurationPointBase` can potentially be exploited to get more scores *Submitted by WatchPug, also found by pedroais*
- [L-08] Unsafe type casting *Submitted by WatchPug*
- [L-09] Use of return value from assignment hampers readability *Submitted by TomFrenchBlockchain, also found by egjlmn1, robee, and WatchPug*
- [L-10] No option to unlock funds before set duration *Submitted by sirhashalot*
- [L-11] in function setLockPeriods, multiplier can be set to lower than 100 *Submitted by Tomio*

Non-Critical Findings (9)

- [N-01] Require with not comprehensive message *Submitted by robee*
- [N-02] Event for merge *Submitted by 0xsanson*
- [N-03] Missing event for admin function setBaseURI *Submitted by BouSalman, also found by WatchPug*
- [N-04] Wrong revert message *Submitted by Czar102*

- [N-05] Improper event declaration *Submitted by Czar102*
- [N-06] Implicit casts should be explicit as per the global code style *Submitted by Dravee*
- [N-07] Various Non-Conformance to Solidity naming conventions *Submitted by Dravee*
- [N-08] Avoid inline code for better readibility *Submitted by StErMi*
- [N-09] Constants are not explicitly declared *Submitted by WatchPug*

Gas Optimizations (28)

- [G-01] XDEFIDistribution: lock should be reused in lockWithPermit *Submitted by PierrickGT*
- [G-02] Gas: `XDEFIDistribution.sol` 's `withdrawAmount` substraction can be unchecked *Submitted by Dravee, also found by 0xsanson, Jujic, WatchPug, and ye0lde*
- [G-03] "Safe" ERC20 functions for XDEFI? *Submitted by 0xsanson*
- [G-04] MAX*TOTAL*XDEFI_SUPPLY should be constant *Submitted by agusduha, also found by 0xsanson, Czar102, Dravee, GiveMeTestEther, p4st13r4, saian, sirhashalot, and WatchPug*
- [G-05] Usage of zero storage for reentrancy guard increases chance that gas refund is capped *Submitted by TomFrenchBlockchain, also found by 0xsanson, bitbopper, Czar102, leastwood, mtz, and WatchPug*
- [G-06] Public functions to external *Submitted by robee, also found by ACai, agusduha, defsec, and Dravee*
- [G-07] Use `calldata` instead of `memory` for external functions where the function argument is read-only. *Submitted by Dravee, also found by Czar102, defsec, and TomFrenchBlockchain*
- [G-08] `> 0 can be replaced with != 0 for gas optimization` *Submitted by defsec, also found by Dravee and Jujic*
- [G-09] Prefix increments are cheaper than postfix increments *Submitted by robee, also found by Dravee, Tomio, and WatchPug*
- [G-10] Use Custom Errors to save Gas *Submitted by Dravee, also found by GiveMeTestEther*

- [G-11] Gas: avoid unnecessary SSTORE on `proposeOwnership` *Submitted by Dravee*
- [G-12] Gas Optimization: Tight variable packing in `XDEFIDistribution.sol` *Submitted by Dravee*
- [G-13] gas optimization *Submitted by Fitraldys*
- [G-14] Gas optimization in XDEFIDistribution.sol - variable that is not used *Submitted by OriDabush, also found by WatchPug*
- [G-15] Gas optimization in XDEFIDistribution.sol - inlining some functions *Submitted by OriDabush*
- [G-16] Gas optimization in XDEFIDistribution.sol - shifting instead of multiplying or dividing by power of 2 *Submitted by OriDabush, also found by WatchPug*
- [G-17] Unneccessary check on total supply of XDEFI token *Submitted by TomFrenchBlockchain, also found by onewayfunction*
- [G-18] `pointCorrection` can be stored in a uint256 rather than int256 to save gas from casting. *Submitted by TomFrenchBlockchain, also found by WatchPug*
- [G-19] Sub-optimal calls should be allowed instead of reverted as resending the transaction will cost more gas *Submitted by WatchPug*
- [G-20] `XDEFIDistribution.sol#relock()` Implementation can be simpler and save some gas *Submitted by WatchPug*
- [G-21] Field bonusMultiplier of struct Position can be removed *Submitted by wuwe1, also found by WatchPug*
- [G-22] `XDEFIDistribution.sol#_updateXDEFIBalance()` Avoiding unnecessary storage writes can save gas *Submitted by WatchPug*
- [G-23] Adding unchecked directive can save gas *Submitted by defsec*
- [G-24] Less than 256 uints are not gas efficient *Submitted by defsec*
- [G-25] && operator can use more gas *Submitted by rfa*
- [G-26] Unnecessary array boundaries check when loading an array element twice *Submitted by robee*
- [G-27] Unnecessary require statement *Submitted by sirhashalot*

- [G-28] XDEFIDistribution: *unlock function should only be called with tokenId parameter* *Submitted by PierrickGT*

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