Introducing Code4rena Blue: Dedicated defense. Competitive bounties. Independent judging.
Learn more →

FIAT DAO veFDT contest Findings & Analysis Report

2022-10-03

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 FIAT DAO veFDT smart contract system written in Solidity. The audit contest took place between August 12—August 15 2022.

Following the C4 audit contest, warden IllIllI reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit contest report.

Wardens

135 Wardens contributed reports to the FIAT DAO veFDT contest:

1. Respx
2. CertoraInc (egjlmn1, OriDabush, ItayG, shakedwinder, and RoiEvenHaim)
3. scaraven
4. ak1
5. jonatascm
6. oyc_109

7. reassor
8. KIntern_NA (TrungOre and duc)
9. cryptphi
10. JohnSmith
11. PwnedNoMore (izhuer, ItsNio, and papr1ka2)
12. 0x1f8b
13. csanuragjain
14. DecorativePineapple
15. rokinot
16. CodingNameKiki
17. 0xSky
18. cccz
19. ayeslick
20. Noah3o6
21. Waze
22. pedr02b2
23. peritoflores
24. IIIIIII
25. cRat1st0s
26. ladboy233
27. rvierdiiev
28. robee
29. d3e4
30. carlitox477
31. joestakey
32. Dravee
33. Aymen0909
34. Deivitto
35. auditor0517

36. bin2chen
37. wagmi
38. 0xf15ers (remora and twojoy)
39. tabish
40. yixxas
41. defsec
42. ajtra
43. MiloTruck
44. bobirichman
45. pfapostol
46. Bnke0x0
47. 0xNazgul
48. ret2basic
49. Rolezn
50. GalloDaSballo
51. ellahi
52. gogo
53. JC
54. ReyAdmirado
55. c3phas
56. TomJ
57. PaludoX0
58. 0xLovesleep
59. 0xDjango
60. paribus
61. RedOneN
62. ElKu
63. Junnon
64. sikorico

65. __141345__
66. mics
67. durianSausage
68. brgltd
69. Funen
70. rbserver
71. simon135
72. LeoS
73. erictee
74. medikko
75. Sm4rty
76. apostle0x01
77. Chom
78. 0xNineDec
79. delfin454000
80. a12jmx
81. 0xbepresent
82. Ruhum
83. djxploit
84. natzuu
85. asutorufos
86. sach1r0
87. bulej93
88. Rohan16
89. Yiko
90. fatherOfBlocks
91. 0x52
92. Bahurum
93. RoiEvenHaim

94. neumo
95. 0xmatt
96. seyni
97. p_crypt0
98. saneryee
99. Vexjon
100. exd0tpy
101. Lambda
102. 0xsolstars (Varun_Verma and masterchief)
103. byndooa
104. sseefried
105. wastewa
106. m_Rassska
107. newfork01
108. Tomio
109. 0xSmartContract
110. Amithuddar
111. jag
112. 0x040
113. Metatron
114. saian
115. sashik_eth
116. 0xHarry
117. 2997ms
118. ignacio
119. SooYa
120. gerdusx
121. SpaceCake
122. 0xackermann

123. chrisdior4
124. CRYP7O
125. Fitraldys
126. NoamYakov

This contest was judged by Justin Goro.

Mitigations reviewed by IllIllI.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 10 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 8 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 93 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 93 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 FIAT DAO veFDT contest repository, and is composed of 5 smart contracts and interfaces written in the Solidity programming language and includes 746 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/non-critical.

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](https://code4rena.com).

High Risk Findings (2)

[H-01] Unsafe usage of ERC20 transfer and transferFrom

Submitted by CertoraInc, also found by 0x1f8b, 0xSky, CodingNameKiki, DecorativePineapple, jonatascm, Noah3o6, oyc_109, pedr02b2, peritoflores, and Waze

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L425-L428
https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L485-L488
https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L546
https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L657
https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L676

Some ERC20 tokens functions don't return a boolean, for example USDT, BNB, OMG. So the `VotingEscrow` contract simply won't work with tokens like that as the `token`.

Proof of Concept

The USDT's `transfer` and `transferFrom` functions doesn't return a bool, so the call to these functions will revert although the user has enough balance and the `VotingEscrow` contract won't work, assuming that token is USDT.

Tools Used

Manual auditing - VS Code, some hardhat tests and me :)

Recommended Mitigation Steps

Use the OpenZepplin's `safeTransfer` and `safeTransferFrom` functions.

lacoop6tu (FIAT DAO) disputed and commented:

> In our case the token is a BalancerV2 Pool Token which returns the boolean

Justin Goro (judge) commented:

> This should be acknowledged, not disputed, since there is nothing in documentation suggesting the token is inherently safe to use.

elnilz (FIAT DAO) commented:

> @Justin Goro it's a no-issue in our specific case bc we will use VotingEscrow in combination with `token` which returns bool upon transfer/transferFrom. So at best this is a QA issue bc we should document that. some wardens actually asked us about what token we will be using pointing out the issue.

> Now even if you'd want to award wardens who reported the issue it should then be a Med Risk bc if VotingEscrow is deployed with an unsafe `token` ppl would simply not be able to deposit into the contract but no funds would be at risk.

elnilz (FIAT DAO) commented:

> Fyi, even though we don't think this is an issue, we will make use of safeTransfer and safeTransferFrom so its a helpful submission nonetheless.

Justin Goro (judge) commented:

> It’s tokens like BNB that led me to maintain the high risk rating. For BNB, transferFrom returns a bool but transfer doesn’t. In other words, users can stake but not unstake on any protocol that doesn’t use safeTransfer.

> I agree that wardens should contact sponsors but it’s not a channel we can really monitor. So although the net result is a documentation fix rather than a bug fix, it’s a documentation fix informed by the identification of a potentially show stopping bug rather than something like “Comment typo: it should be Bitcoin, not bit coin”.

IllIllI (warden) reviewed mitigation:

> The sponsor disputed the issue because the token it’s planned to be used with does correctly return a boolean. However, the sponsor decided to make a change to address the finding as [Issue 18](). The fix properly replaces the `require()` statements that check for successful transfers, with calls to OpenZeppelin’s `safeTransfer()`. The PR also replaces the internal definition of the `IERC20` interface with OpenZeppelin’s version. The prior version of the code’s `IERC20` included the function `decimals()`, which is not one of the required functions for the interface, so it’s possible for the code to encounter a token without this function, but it would be immediately apparent what happened because the constructor is the function that calls `decimals()`. The change to using OpenZeppelin required making this distinction more visible due to the fact that they’re defined separately as `IERC20` and `IERC20Metadata`. The new code is not checking that the token actually supports the function (e.g. using a `safeDecimals()`-like function), but it is not any worse off that it had been prior to the change.

[H-02] Delegators can Avoid Lock Commitments if they can Reliably get Themselves Blocked when Needed

Submitted by Respx

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L526-L625

Users can enjoy the voting power of long lock times whilst not committing their tokens. This could cause the entire system to break down as the incentives don't work any more.

Exploit Method

This exploit only works if a user is able to use the system and reliably get themselves blocked. Blocking policies are not in scope, so I am assuming there would be a list of bannable offences, and thus this condition could be fulfilled.
Consider a user with two accounts, called Rider and Horse.
Rider has 100,000 tokens.
Horse has 1 token.
Rider is a smart contract (required for an account to be bannable).
Rider locks for 1 week.
Horse locks for 52 weeks.
Rider delegates to Horse.
Horse can continue to extend its lock period and enjoy the maximised voting power.
Whenever the user wants their tokens back, they simply need to get the Rider account blocked.
When Rider is blocked, `Blocklist.block(RiderAddress)` is called, which in turn calls `ve.forceUndelegate(RiderAddress)`.
Rider is now an undelegated account with an expired lock. It can call `ve.withdraw()` to get its tokens back.
The user can repeat this process with a fresh account taking the role of Rider.

Recommended Mitigation Steps

`forceUndelegate()` could be made to set `locked_.end = fromLocked.end`. This would mean that blocked users are still locked into the system for the period they delegated for. However, this does have the downside of tokens being locked in the system without the full rights of the system which other users enjoy.
Alternatively, this might be addressable through not blocking users that seem to be doing this, but of course that might have other undersirable consequences.

Proof of Concept

Please see warden's full report for proof of concept.

lacoop6tu (FIAT DAO) confirmed, but disagreed with severity and commented:

> `2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

Justin Goro (judge) commented:

> Well spotted by warden! The inflation of voting points may lead to an exploit, depending on possible proposals. Severity maintained.

IllIllI (warden) reviewed mitigation:

> The sponsor disagreed with the severity and the judge updated the issue to be of Medium risk, and I agree with that severity. The finding was addressed via the fix for Issue 6 where the sponsor implemented the suggestion of the warden, to use the delegatee's lock endpoint in the re-delegation to self, rather than using the delegator's existing endpoint, since that endpoint may be far in the past. The delegate() and undelegate() functions have checks to ensure that the target for the votes always has at least as long a duration as the source of the votes. The fix enforces the same requirement for `forceUndelegate()` by assigning a longer duration.

> There are only two places in the code that change `LockedBalance.end` to a smaller value, which could possibly violate the contract invariants: in `quitLock()` where the struct is never written back to storage, and in `withdraw()` where it is indeed written back to storage. However, if the delegatee was able to withdraw, that means the delegator already would have been able to withdraw (since the delegatee's timestamp must always be greater than or equal to the delegator's when delegating or increasing), and therefore the mitigation is correct. The only extra wrinkle that the change makes, is that it now allows a malicious delegatee to front-run a delegator's block with an `increaseUnlock(MAXTIME)`, but it's not clear what advantage that would give the delegatee, and furthermore, the delegator already put his/her trust in the delegatee, so it's something that could have occurred anyway, even without a call to `forceUndelegate()`.

Medium Risk Findings (8)

[M-01] The current implementation of the VotingEscrow contract doesn't support fee on transfer tokens

Submitted by CertoraInc, also found by cccz, csanuragjain, jonatascm, and scaraven

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L418

Some ERC20 tokens implemented so a fee is taken when transferring them, for example `STA` and `PAXG`. The current implementation of the `VotingEscrow` contract will mess up the accounting of the locked amounts if `token` will be a token like that, what will lead to a state where users won't be able to receive their funds.

This will happen because the value that is added to the locked amount is not the actual value received by the contract, but the value supplied by the user (the value which the fee is taken from).

Proof of Concept

The `STA` token burns 1% of the value provided to the `transfer` function, which means the recipient gets only 99% of the transferred asset. Let's assume that `token` is the address of the `STA` token.

1. Bob wants to lock 100 STA tokens and calls `createLock(100 * 10**18, unlockTime)`.
2. The addition to the locked amount variable is done with `100 * 10**18`, while the actual amount that was received by the contract is `99 * 10**18`.
3. When the lock expires Bob will try to withdraw his tokens, and the transfer function will be called with the accounted locked amount (which is `100 * 10**18`). This might succeed due to other users locking too, so the transferred tokens will be taken from "their tokens", but in the end there will be users left without an option to withdraw their funds, because the balance of the contract will be less than the locked amount that the contract is trying to transfer.

Tools Used

Manual auditing - VS Code and me :)

Recommended Mitigation Steps

Calculate the amount to add to the locked amount by the difference between the balances before and after the transfer instead of using the supplied value.

lacoop6tu (FIAT DAO) disputed and commented:

> In our case, the token will be BalancerV2 Pool Token , which has no fee on transfer, in case someone else would like to fork this contract and use it, that fix will be required.

Justin Goro (judge) commented:

> Given that the warden couldn't know the use of Balancer only tokens, the severity will still be upheld.

elnilz (FIAT DAO) commented:

> @Justin Goro so should we explicitly exclude all weird implementations of e.g. ERC20 in the future in the contest docs? I mean there are other wild examples of ERC20 implementations that someone could point out would cause problems with this contract. I am not trying to discount the work of any warden here but I think the correct response here as well as for #231 would be to improve docs stating which ERC20 implementations are safe to use in combination with VotingEscrow.

Justin Goro (judge) commented:

> @elnilz This topic is very important to get right and the more it's debated, the more it's clear that there is no one size fits all answer. When I sponsored a contest, I only figured out after the fact the sorts of things that would have reduced unhelpful warden submissions, through no fault of C4.

> My suggestion is that we curate an open source optional questionnaire for sponsors. The more detail the sponsor gives a priori, the more we can mark unhelpful issues as invalid. As an example, the tools I use to deploy my dapps do not allow me to accidentally omit addresses in constructor arguments. So wardens who warn me that I don't check for the zero address in my constructors are not helping me. The questionnaire would cover many common case submissions such as that.

> In your case, I had to dance a bit of a fine line: on the one hand, the wardens are not wrong in the event that you're unaware of token design nuance and so they shouldn't be penalized. On the other hand, this is a DAO and the wardens should at least suspect that the choice of token is a central decision. In the end, since both parties have good points to make and there's no clear decider, I chose to side with the wardens since it doesn't incur any additional cost to you as the sponsor and since it would seem unfair to penalize the wardens for an honest report with no flaws at the correct severity level.

IllIlll (warden) reviewed mitigation:

> The sponsor disputed the issue because the Balancer V2 Pool tokens it's planned to be used with do not implement a fee-on-transfer mechanic. The tokens do not appear to be upgradeable so there is no risk of fees being added to existing tokens via upgrade. Without more information about how which pool tokens are chosen/allowed/used, and what prevents future pool tokens that implement such a mechanic from being used with the same contract, I have to agree with the warden and judge that this is a Medium risk issue. The suggested mitigation is to measure the balance of the token that the contract holds before the `transferFrom()` call is made, and afterwards, and use the difference as the value, rather than the amount the user states. You could also add a `require()` enforcing the invariant that the change in balance must equal the stated amount, which would *prevent* fee-on-transfer tokens from being used.

> In the final PR, the sponsor has acknowledged the issue and added a code comment saying that fee-on-transfer tokens are not supported.

[M-02] Attacker contract can avoid being blocked by BlockList.sol

Submitted by JohnSmith, also found by ayeslick, reassor, rokinot, and scaraven

To block an address it must pass the `isContract(address)` check:
https://github.com/code-423n4/2022-08-fiatdao/blob/main/contracts/features/Blocklist.sol#L25

```
contracts/features/Blocklist.sol
25:          require(_isContract(addr), "Only contracts");
```

Which just checks code length at the address provided.

```
contracts/features/Blocklist.sol
    function _isContract(address addr) internal view returns
        uint256 size;
        assembly {
            size := extcodesize(addr)
        }
        return size > 0;
    }
```

Attacker can interact with the system and selfdestruct his contract, and with help of CREATE2 recreate it at same address when he needs to interact with the system again.

Proof of concept

Below is a simple example of salted contract creation, which you can test against `_isContract(address)` function.

```
pragma solidity 0.8.15;

contract BlockList {
    function _isContract(address addr) external view returns (bo
        uint256 size;
        assembly {
            size := extcodesize(addr)
        }
        return size > 0;
    }
}

contract AttackerContract {
  function destroy() external {
    selfdestruct(payable(0));
  }
}

contract AttackerFactory {
    function deploy() external returns (address) {
        return address(new AttackerContract{salt: bytes32("123")
    }
    }
```

Recommended Mitigation Steps

One of the goals of Ethereum is for humans and smart contracts to both be treated equally. This leads into a future where smart contracts interact seamlessly with humans and other contracts. It might change in the future , but for now an arbitrary address is ambiguous.
We should consider blacklisting addresses without checking if they are contracts.

lacoop6tu (FIAT DAO) commented:

> Duplicate of #168

Justin Goro (judge) commented:

> This is a valid attack vector that undermines the blocking mechanism and is not a duplicate of #168.

lacoop6tu (FIAT DAO) commented:

> IMO this is more acknowledged in this case, the only interaction possible is locking LP tokens first so if someone then selfdestructs, another attacker could create a contract with that address and take ownership (and quitLock for example) similar to what happened with optimism and wintermute.

Justin Goro (judge) commented:

> The reason it's maintained as a medium risk is because there is a bit of circumventing of protocol restrictions. But as you indicated, it's not serious enough that marking it acknowledged is irresponsible.

elnilz (FIAT DAO) acknowledged

IllIllI (warden) reviewed mitigation:

> The sponsor acknowledges that the `BlockList` can be bypassed, but states that "the only interaction possible is locking LP tokens first". However, looking at the code, the `checkBlocklist` modifier is applied to not just `createLock()` , but

> `increaseAmount()`, `increaseUnlockTime()`, and `delegate()`. An attacker can bypass the block list for every one of these functions by making their SmartWallet a specially-constructed `create2()` contract that does external calls to an other contract in its constructor, for instructions on what to execute, before self-destructing. Whenever the attacker wants to interact with the token, they update their external instruction-providing contract with the action to take, re-create the attack contract. It's not clear why the block list is only for contracts, and if it can be bypassed by using this method, or by transferring the tokens to an EOA.
> In discussions of the issue, the sponsor clarified that the `BlockList`'s purpose is to prevent lock tokenization, and acknowledged that using an updated `BlockList` that blocks specific EOAs may be required if an attacker uses the features described above to work around being blocked.

[M-03] Inconsistent logic of increase unlock time to the expired locks

Submitted by KIntern_NA, also found by ak1, cryptphi, and scaraven

Can not prevent expired locks being extended.

Proof of Concept

https://github.com/code-423n4/2022-08-fiatdao/blob/main/contracts/VotingEscrow.sol#L493-L523

Call function function `increaseUnlockTime()` with an expired lock (locked[msg.sender].end < block.timestamp)

- Case 1: if sender's lock was not delegated to another address, function will be revert because of the requirement

https://github.com/code-423n4/2022-08-fiatdao/blob/main/contracts/VotingEscrow.sol#L511

- Case 2: if sender's lock was delegated to another address, function will not check anything and the lock can be extended.

But in case 1, sender's lock was not delegated to another, the sender can delegate to new address with end time of lock equal to new end time. After that he can call `increaseUnlockTime()` and move to case 2. Then sender can undelegate and the lock will be extended, and sender will take back vote power.

Here is the script :

```
    describe("voting escrow", async () => {
        it("increase unlock time issue", async () => {
          await createSnapshot(provider);
          //alice creates lock
          let lockTime = WEEK + (await getTimestamp());
          await ve.connect(alice).createLock(lockAmount, lockTime);
          // bob creates lock
          lockTime = 50 * WEEK + (await getTimestamp());
          await ve.connect(bob).createLock(10 ** 8, lockTime);
          //pass 1 week, alice's lock is expired
          await ethers.provider.send("evm_mine", [await getTimestamp
          expect(await ve.balanceOf(alice.address)).to.eq(0);
          //alice can not increase unlock timme
          await expect(ve.connect(alice).increaseUnlockTime(lockTime
          //alice delegate to bob then can increase unlock time
          await ve.connect(alice).delegate(bob.address);
          await expect(ve.connect(alice).increaseUnlockTime(lockTime
          //alice delegate back herself
          await ve.connect(alice).delegate(alice.address);
          expect(await ve.balanceOf(alice.address)).to.gt(0);
        });
```

Recommended Mitigation Steps

In every cases, expired locks should able to be extended -> should remove line VotingEscrow.sol#L511.

lacoop6tu (FIAT DAO) confirmed

Justin Goro (judge) commented:

> Very good report, especially because of that script.

IllIIII (warden) reviewed mitigation:

> The sponsor addressed the finding with the fix for Issue 4. The fix chosen was to not allow the increasing of lock time or non-self re-delegation if the delegatee's lock has expired. The fix didn't require the undelegate flavor to duplicate the blocklist check since `msg.sender` is already checked by the `checkBlocklist` modifier. The refactored code properly re-used some variables rather than duplicating the allocations done in the delegation/re-delegation case in order to save some gas. The refactoring introduced a new issue, [M.N-01], described below in the Mitigation Review section.

[M-04] Error in Updating `_checkpoint` in the `increaseUnlockTime` Function

Submitted by Aymen0909, also found by 0xf15ers, 0xSky, auditor0517, bin2chen, CertoraInc, csanuragjain, JohnSmith, scaraven, tabish, wagmi, and yixxas

The potentiel impact of this error are :

- Give wrong voting power to a user at a given block.
- Give wrong total voting power at a given block.
- Give wrong total voting power.

Proof of Concept

The error occured in this line : https://github.com/code-423n4/2022-08-fiatdao/blob/main/contracts/VotingEscrow.sol#L513

In the **increaseUnlockTime** function the oldLocked.end passed to the function **_checkpoint** is wrong as it is the same as the new newLock end time (called unlock_time) instead of being equal to **oldUnlockTime** .

In the given CheckpointMath.md file it is stated that checkpoint details for **increaseUnlockTime** function should be :

Lock	amount	end
old	owner.delegated	owner.end
new	owner.delegated	T

BUT with this error you get a different checkpoint details :

Lock	amount	end
old	owner.delegated	T
new	owner.delegated	T

The error is illustrated in the code below :

```
LockedBalance memory locked_ = locked[msg.sender];
uint256 unlock_time = _floorToWeek(_unlockTime); // Lock
/* @audit comment
        the unlock_time represent the newLock end time
*/
// Validate inputs
require(locked_.amount > 0, "No lock");
require(unlock_time > locked_.end, "Only increase lock e
require(unlock_time <= block.timestamp + MAXTIME, "Excee
// Update lock
uint256 oldUnlockTime = locked_.end;
locked_.end = unlock_time;
/* @audit comment
        The locked_ end time is update from  oldUnlockT
*/
locked[msg.sender] = locked_;
if (locked_.delegatee == msg.sender) {
    // Undelegated lock
    require(oldUnlockTime > block.timestamp, "Lock expir
    LockedBalance memory oldLocked = _copyLock(locked_);
    oldLocked.end = unlock_time;
    /* @audit comment
        The oldLocked.end is set to unlock_time instead
    */
    _checkpoint(msg.sender, oldLocked, locked_);
}
```

The impact of this is when calculating the **userOldPoint.bias** in the **_checkpoint** function you get an incorrect value equal to **userNewPoint.bias** (because oldLocked.end == _newLocked.end which is wrong).

```
240         userOldPoint.bias =
241                     userOldPoint.slope *
242                     int128(int256(_oldLocked.end - block.time
```

The wrong **userOldPoint.bias** value is later used to calculate and update the bias value for the new point in **PointHistory.**

```
359       lastPoint.bias =
360                lastPoint.bias +
361                userNewPoint.bias -
362                userOldPoint.bias;

372       pointHistory[epoch] = lastPoint;
```

And added to that the wrong **oldLocked.end** is used to get oldSlopeDelta value which is used to update the **slopeChanges.**

```
271       oldSlopeDelta = slopeChanges[_oldLocked.end];

380       oldSlopeDelta = oldSlopeDelta + userOldPoint.slope;
381       if (_newLocked.end == _oldLocked.end) {
382                oldSlopeDelta = oldSlopeDelta - userNewPoin
383        }
384       slopeChanges[_oldLocked.end] = oldSlopeDelta;
```

As the **PointHistory** and the **slopeChanges** values are used inside the functions **balanceOfAt()** , **_supplyAt(), totalSupply(), totalSupplyAt()** to calculate the voting power, **this error could give wrong voting power at a given block of a user or can give wrong total voting power.**

Recommended Mitigation Steps

The line 513 in the VotingEscrow.sol contract :

```
513      oldLocked.end = unlock_time;
```

Need to be replaced with the following :

```
        oldLocked.end = oldUnlockTime;
```

lacoop6tu (FIAT DAO) confirmed, but disagreed with severity and commented:

> As majority of wardens reported, this is Medium finding
> `2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

Justin Goro (judge) decreased severity to Medium and commented:

> The severity will be downgraded but otherwise a good report.

IIIIIII (warden) reviewed mitigation:

> The sponsor addressed the finding with the fix for Issue 5. The fix properly changes the code to match the invariants specification, and matches the logical expectation that the 'old' field uses the 'old' timestamp.

[M-05] Unsafe casting from int128 can cause wrong accounting of locked amounts

Submitted by CertoraInc, also found by 0x1f8b, carlitox477, cRat1st0s, DecorativePineapple, joestakey, ladboy233, reassor, and rvierdiiev

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L418

The unsafe casting to int128 variable can cause its value to be different from the correct value. For example in the createLock function, the addition to the locked amount variable is done by `locked_.amount += int128(int256(_value))`. In that case, if `_value` is greater than `type(int128).max` which is `2**127 - 1`, then the accounting will be wrong and the amount that will be added to `locked_.amount` will be less than the amount of token that will be transferred from the user. Then the user won't be able to withdraw the tokens that he transferred, and they'll be stuck in the contract forever.

Proof of Concept

1. Alice tries to lock `2**128` tokens. She calls `createLock(2**128, unlockTime)` with the time she wants to lock for.
2. The addition of the given value is done by `locked_.amount += int128(int256(_value))`, which actually does nothing to the `locked_.amount` variable and it remains 0. That's because when casting `int128(int256(2**128))` truncates to 0, and that leaves the locked amount unchanged but the tokens are transferred.

Tools Used

Manual auditing - VS Code and me :)

Recommended Mitigation Steps

Make sure that the values fit in the variables you are trying to assign them to when casting variables to smaller types.

lacoop6tu (FIAT DAO) acknowledged and commented:

> This is true but doesn't apply in our case, we use a BalV2 Pool Token which would never reach those values in terms of existing supply.

Justin Goro (judge) decreased severity and commented:

> The use of Balancer tokens doesn't preclude numbers above 128bit. In the BalancerV2 source code, all amounts are in uint256. However, the widespread practice of standard Ethereum tokens makes the likelihood of even encountering a token balance above 128 bits is negligible and Balancer does scale down big tokens if the other tokens in the pool are lower when minting.

> Marking this as high risk is simply not realistic. This and its duplicates will be downgraded to medium risk (2) as it's a type of technicality that will have no bite in reality.

IIIIIII (warden) reviewed mitigation:

> The sponsor acknowledges that overflow is technically possible, but considers this as unlikely to happen in practice.

> In the final PR, the sponsor added a code comment saying that the contract does not support tokens where `maxSupply>2^128-10^[decimals]`.

[M-06] `increaseUnlockTime` missing `_checkpoint` for delegated values

Submitted by PwnedNoMore, also found by ak1, CertoraInc, and scaraven

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L509-L515

In the VotingEscrow contract, users can increase their voting power by:

- Adding more funds to their delegated value
- Increasing the time of their lock
- Being delegated by another user

Specifically, when users are delegated by other users through the `delegate` function, the delegated user gains control over the delegate funds from the delegating user.

The delegated user can further increase this power by increasing the time that the delegated funds are locked by calling `increaseUnlockTime`, resulting in ALL the delegated funds controlled by the delegated user, including those that do not originate from the delegated user, being used to increase the voting power of the user.

The issue lies in the following scenario: If user A delegates to user B, and then user B delegates to user C, user B loses the ability to extend his or her voting power by `increaseUnlockTime` due to a missing `_checkpoint` operation. If user B calls the `increaseUnlockTime` function, the `_checkpoint` operation will not proceed, as user B is delegating to user C. However, B still owns delegated funds, in the form of the funds delegated from user A. Therefore, user B should still gain voting power from `increaseUnlockTime`, even though user B is delegating.

Proof of Concept

Assume three users, Alice, Bob, and Carol, who each possess `locks` with 10 units of `delegate` value. Also assume that the unlock time is 1 week.

- Alice delegates her 10 units to Bob.
- Bob then delegates his 10 units to Carol.
- At this point, Alice has 0 `delegate`, value, Bob has 10 `delegate` value, and Carol has 20 `delegate` value.
- Carol calls `increaseUnlockTime` to 2 weeks, resulting in `_checkpoint` raising her voting power accordingly.
- Bob calls `increaseUnlockTime` to 2 weeks, resulting in no change in his voting power, even though he has 10 units of `delegate` value.

Recommended Mitigation Steps

Move the `_checkpoint` outside of the `if` statement on line 514.

lacoop6tu (FIAT DAO) confirmed, but disagreed with severity and commented:

> As most of wardens reported in duplicated, this is Medium finding
> `2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

Justin Goro (judge) decreased severity to Medium

IllIIII (warden) reviewed mitigation:

> The sponsor addressed the finding with the fix for Issue 2. In cases where a user is both a delegator and a delegatee, the original code did not create a checkpoint for calls to `increaseUnlockTime()`. Self-delegation and being delegated to both increase the `LockedBalance.delegated` field, so the change to the condition of

> the if-statement now includes both cases. The code will not get to the if-statement if the user has already withdrawn, due to a `require()`, so a user that has delegates but has withdrawn, cannot increase their now-zero unlock time.
> `increaseAmount()` has a similar if-statement and comment, but the else-block is already covered by a checkpoint, so there is no analogous issue there.

[M-07] Blocking Through Change of Blocklist Could Trap Tokens

Submitted by Respx

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/features/Blocklist.sol#L27 https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L531 https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L637

In the normal flow, an account that is blocked is protected from having its funds locked by a call to `forceUndelegate()`, as occurs on line 27 of `Blocklist.sol`.
However, this protection could potentially be circumvented if the value of `blocklist` is changed to an address which returns `True` for `isBlocked()` (as tested in the modifier `checkBlocklist()`) and if this account was not previously blocked (ie. `forceUndelegate()` was never called on it).
In this situation, if the account has delegated, its tokens will be rendered irretrievable.

Proof of Concept

The blocked account would not be able to call `wthdraw()` successfully because of the check on line 531.
The blocked account would not be able to call `quitLock()` successfully because of the check on line 637.
The blocked account would not be able to call `delegate()` to undelegate and thereby allow it to make these calls because `delegate()` uses the `checkBlocklist` modifier.
`Blocklist` has no unblock functionality, so the only way to release the tokens would be through a redeployment of `Blocklist`.

Recommended Mitigation Steps

This situation is most likely to occur as an error during a blocklist migration. In that case, it could be mitigated by adding an unblock functionality to the blocklist contract.

lacoop6tu (FIAT DAO) acknowledged

elnilz (FIAT DAO) disagreed with severity and commented:

> Not High Risk as no funds at risk. In the scenario outlined above, a clear path to restoring user's access to the blocked tokens exists.

Justin Goro (judge) decreased severity to Meidum and commented:

> Severity downgraded.

IllIllI (warden) reviewed mitigation:

> The sponsor acknowledges the possible rug vector. It is common for admin rug vectors to not be addressed.

[M-08] Attackers can abuse the `quitLock` function to get a very large amount of votes

Submitted by CertoraInc

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L632-L659

An attacker can use a flashloan and the quitLock function to achieve a large amount of votes for one transaction. It can, depends on the implementation of the modules that will use this contract, be used to pass malicious proposals or exploit any feature that depends on the voting balance.

Proof of Concept

We assume here that there is a contract that provides a flashloan (for simplicity without fees, but can also work with fees, just requires the attacker to provide a larger amount of tokens) for the token that is used by the VotingEscrow contract.

1. The attacker deploys a smart contract that implements the following logic and approves the contract for the token that is used in the `VotingEscrow` contract (of course assigning all the values of the variables).
2. The attacker calls the `attack` function with the amount he wants to take as a flashloan (he will need to cover a penalty based on that amount).
3. The `attack` function calculates the penalty and transfers it from the attacker.
4. The `flashloan` function is called, which provides the tokens to the contract and then calls the `flashloanCallback` with the lent amount.
5. The `flashloanCallback` function creates a lock with the amount received from the flashloan for a week (the unlock time can be changed to achieve larger votes balance, but it must be considered when calculating the penalty).
6. The attacker can do whatever he wants with the amount of votes the contract currently has.
7. The quitLock function is called to get back the funds (excluding the penalty), and the loan is payed back.

```
contract VotingEscrowAttack {
    IERC20 constant token = IERC20(0x...); // the token used in
    IVotingEscrow constant votingEscrow = IVotingEscrow(0x...);

    uint constant WEEK = 1 weeks;
    uint constant MAXTIME = 365 days;
    uint constant MAXPENALTY = 10**18;

    uint constant PENALTY_RATE = (WEEK * MAXPENALTY) / MAXTIME;

    IFlashLoan constant flashloanContract = IFlashLoan(0x...); /

    function attack(uint amount) external {
        uint penalty = (amount * PENALTY_RATE) / (10 ** 18);
        token.transferFrom(msg.sender, penalty); // assuming no
        IFlashLoan.flashloan(token, amount);
    }

    function flashloanCallback(uint amount) {
        votingEscrow.createLock(amount, block.timestamp + WEEK);

        // do whatever you want with your large amount of votes

        votingEscrow.quitLock();
        token.transfer(msg.sender, amount); // pay back the flas
    }
}
```

- The function names, arguments and return values might not be accurate and might change depends on the used flashloan platform, but this contract is just to give a general idea of an attack vector.

Tools Used

Manual auditing - VS Code and me :)

Recommended Mitigation Steps

Think about implementing a mechanism that prevents users from creating a lock and quitting it in the same transaction, that way attackers won't be able to use flashloan in order to achieve large voting power. However, regular loans will still be a problem with that fix, and if this isn't a wanted behavior, additional fix is needed to be thought of.

lacoop6tu (FIAT DAO) acknowledged

elnilz (FIAT DAO) disagreed with severity and commented:

> We actually could mitigate the issue by restricting increasing voting power and quitting in the same block. This would make the implementation safer.

> But even then, the severity is rather 2 as funds are not directly at risk.

Justin Goro (judge) commented:

> The scope of voting power is unclear. It may be that a proposal becomes possible that enables large funds transfers. For this reason, severity is maintained.

elnilz (FIAT DAO) disputed and commented:

> Did review the economics of this attack and they are the same than without quitLock:

> First, note that both balanceOf (voting power) and fee paid in quitLock are proportional to `(locked.end-block.timestamp)/MAXTIME` or remaining lock duration. In other words, in order to gain voting power, user also accepts higher quitLock fee.

> For max voting power user locks with MAXTIME. When quitting, this also results in the loss of all her locked tokens. Because the voting power decreases linearly with remaining lock duration, the amount of tokens that need be locked in order to achieve same voting power increases with lower lock durations. This results in the same quitLock fee as if the user would chose max lock duration or if user would not be able to quit at all.

> See this interactive graph and play with the t (remaining lock duration) and N (tokens locked) sliders: https://www.desmos.com/calculator/yjby9zempb

> Thus, quitLock doesn't change the economics of governance attacks and so we dispute this issue.

[Justin Goro (judge) decreased severity to Medium and commented]():

> Upon reconsideration, it appears the dampening effect of the penalty ensures that this FlashLoan attack is only really viable at very low impact levels. The larger the attack, the more the cost of the attack will exceed any benefit. The only case where this wouldn't be the case is when a small vote can tip the balance of an important decision which is unlikely in a gauge style vote. But even if a threshold emerges, the attacker may as well just get the votes through normal channels.

> Nonetheless, the vector is only dampened, not eradicated and so I'll be downgrading this to Medium severity.

[elnilz (FIAT DAO) commented]():

> Thanks for reconsidering. I'd also be curious about feedback from CertoraInc as it's been reported by the user.

> The implications of the above analysis are, however, that the cost of an attack is the same as with Curve's VotingEscrow which doesn't implement a quitLock function. So unless the attack vector on Curve itself is also a medium severity this issue should be invalidated.

IllIllI (warden) reviewed mitigation:

> The sponsor disputed the issue on the basis of an economic analysis of the penalty taken vs the votes gained, the outcome of which was that the penalty always covers the votes gained. I spoke with the sponsor and the sponsor explained that for the original Curve Finance code, the number of votes one gets is not equal one-for-one to the number of tokens locked: locking for the maximum duration will get you close to one-for-one, but every second under that number, the locking user gets fewer and fewer votes. Therefore in veFTD, the penalty is always chosen such that when the penalty is subtracted from the votes gained, the number of votes a user is left with after quitting is equal to the number of votes they would have been given had they used the quit time as their lock end time instead. In other words, the votes one would get for locking for the week the warden mentions, would be equal to the penalty they gather ahead of time, so the flash loan does not help the attacker.

> To verify all of this, I wrote two tests that make use of hardhat's ability to mine specific blocks with specific times. The [first] test confirmed that indeed, when one subtracts the penalty from the number of votes gained, the remaining number of votes is less than the number of votes a separate user gains for locking for the shorter duration, so it's always better to specify the correct lock time rather and unlocking early. The [second] test verifies that the same is true even if one quits the second after the lock is created. Finally, I wrote a [test] that specifically does the flash loan scenario the warden outlined, and was able to show that when the attacking contract checks its balance between locking and quitting for the previous block, the votes are zero, and for the current block, the penalty is larger than the votes gained (and one cannot query vote balances for future blocks). Once the contract's attack call completes, checks for the votes for same blocks show zero votes for both, so I believe the warden's finding is invalid.

Low Risk and Non-Critical Issues

For this contest, 93 reports were submitted by wardens detailing low risk and non-critical issues. The [report highlighted below] by **oyc_109** received the top score from the judge.

The following wardens also submitted reports: d3e4, robee, Deivitto, CodingNameKiki, lllllll, defsec, bobirichman, Dravee, 0xNazgul, pfapostol, GalloDaSballo, Rolezn, ellahi, 0x52, gogo, Respx, Bnke0x0, Aymen0909, JC, PaludoX0, Bahurum, ret2basic, TomJ, ElKu, Junnon, RedOneN, ReyAdmirado, RoiEvenHaim, sikorico, cRat1st0s, auditor0517, reassor, JohnSmith, c3phas, ladboy233, mics, 0xDjango, ajtra, bin2chen, brgltd, Funen, Noah3o6, ak1, CertoraInc, erictee, MiloTruck, neumo, simon135, 0x1f8b, __141345__, apostle0x01, 0xmatt, seyni, 0xNineDec, rbserver, rvierdiiev, Waze, a12jmx, delfin454000, LeoS, KIntern_NA, Ruhum, Sm4rty, p_crypt0, wagmi, Chom, saneryee, Vexjon, bulej93, cryptphi, djxploit, exd0tpy, natzuu, 0xLovesleep, Lambda, paribus, asutorufos, 0xbepresent, 0xsolstars, csanuragjain, DecorativePineapple, durianSausage, jonatascm, medikko, Rohan16, rokinot, sach1r0, byndooa, sseefried, fatherOfBlocks, Yiko, *and* wastewa.

[L-01] Upgrade Open Zeppelin contract dependency

An outdated OZ version is used (which has known vulnerabilities, see https://github.com/OpenZeppelin/openzeppelin-contracts/security/advisories).

The solution uses:

```
    "@openzeppelin/contracts": "^4.4.2",
```

[L-02] No Transfer Ownership Pattern

Recommend considering implementing a two step process where the owner or admin nominates an account and the nominated account needs to call an acceptOwnership() function for the transfer of ownership to fully succeed. This ensures the nominated EOA account is a valid and active account.

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L139-L143

[L-03] Unspecific Compiler Version Pragma

Avoid floating pragmas for non-library contracts.

While floating pragmas make sense for libraries to allow them to be included with multiple different versions of applications, it may be a security risk for application implementations.

A known vulnerable compiler version may accidentally be selected or security tools might fall-back to an older compiler version ending up checking a different EVM compilation that is ultimately deployed on the blockchain.

It is recommended to pin to a concrete compiler version.

```
Blocklist.sol::2 => pragma solidity ^0.8.3;
IBlocklist.sol::2 => pragma solidity ^0.8.3;
IERC20.sol::2 => pragma solidity ^0.8.3;
IVotingEscrow.sol::2 => pragma solidity ^0.8.3;
VotingEscrow.sol::2 => pragma solidity ^0.8.3;
```

[N-01] Use a more recent version of solidity

Use a solidity version of at least 0.8.4 to get bytes.concat() instead of abi.encodePacked(,) Use a solidity version of at least 0.8.12 to get string.concat() instead of abi.encodePacked(,) Use a solidity version of at least 0.8.13 to get the ability to use using for with a list of free functions

```
Blocklist.sol::2 => pragma solidity ^0.8.3;
IBlocklist.sol::2 => pragma solidity ^0.8.3;
IERC20.sol::2 => pragma solidity ^0.8.3;
IVotingEscrow.sol::2 => pragma solidity ^0.8.3;
VotingEscrow.sol::2 => pragma solidity ^0.8.3;
```

[N-02] Large multiples of ten should use scientific notation

Use (e.g. 1e6) rather than decimal literals (e.g. 1000000), for better code readability

```
VotingEscrow.sol::57 => Point[1000000000000000000] public pointH
VotingEscrow.sol::58 => mapping(address => Point[1000000000]) pul
```

[N-03] Use scientific notation (e.g. 1e18) rather than exponentiation (e.g. 10**18)

Scientific notation should be used for better code readability

```
VotingEscrow.sol::48 => uint256 public constant MULTIPLIER = 10*
VotingEscrow.sol::51 => uint256 public maxPenalty = 10**18; // p
VotingEscrow.sol::653 => uint256 penaltyAmount = (value * penalt
```

[N-04] Event is missing indexed fields

Each event should use three indexed fields if there are three or more fields

```
VotingEscrow.sol::38 => event TransferOwnership(address owner);
VotingEscrow.sol::39 => event UpdateBlocklist(address blocklist)
VotingEscrow.sol::40 => event UpdatePenaltyRecipient(address rec
VotingEscrow.sol::41 => event CollectPenalty(uint256 amount, add
```

[N-05] Missing NatSpec

Code should include NatSpec

```
IERC20.sol::1 => // SPDX-License-Identifier: Apache-2.0
```

[N-06] Constants should be defined rather than using magic numbers

It is bad practice to use numbers directly in code without explanation

```
VotingEscrow.sol::309 => for (uint256 i = 0; i < 255; i++) {
```

[N-07] Public functions not called by the contract should be declared external instead

Contracts are allowed to override their parents' functions and change the visibility from external to public.

```
Blocklist.sol::33 => function isBlocked(address addr) public view
VotingEscrow.sol::754 => function balanceOf(address _owner) publ
VotingEscrow.sol::864 => function totalSupply() public view over
```

Gas Optimizations

For this contest, 93 reports were submitted by wardens detailing gas optimizations. The report highlighted below by IllIllI received the top score from the judge.

The following wardens also submitted reports: Dravee, JohnSmith, 0x1f8b, ajtra, MiloTruck, Bnke0x0, Deivitto, ret2basic, pfapostol, oyc_109, Aymen0909, m_Rassska, defsec, c3phas, ReyAdmirado, CodingNameKiki, gogo, JC, TomJ, 0xLovesleep, 0xDjango, paribus, Rolezn, CertoraInc, newfork01, robee, Tomio, __141345__, 0xSmartContract, 0xNazgul, cRat1st0s, durianSausage, reassor, RedOneN, Amithuddar, jag, rbserver, 0x040, brgltd, ElKu, LeoS, simon135, Waze, medikko, Metatron, saian, Sm4rty, Chom, GalloDaSballo, Noah3o6, sashik_eth, 0xHarry, rokinot, 0xbepresent, 2997ms, bobirichman, ignacio, SooYa, mics, ak1, asutorufos, djxploit, Junnon, natzuu, PaludoX0, Ruhum, sach1r0, apostle0x01, bulej93, d3e4, delfin454000, gerdusx, ladboy233, rvierdiiev, SpaceCake, 0xackermann, 0xNineDec, carlitox477, chrisdior4, CRYP70, sikorico, a12jmx, csanuragjain, Respx, Rohan16, Yiko, ellahi, erictee, fatherOfBlocks, Fitraldys, Funen, *and* NoamYakov.

Summary

	Issue	Instances
[G-01]	Multiple `address`/ID mappings can be combined into a single `mapping` of an `address`/ID to a `struct`, where appropriate	1
[G-02]	State variables only set in the constructor should be declared `immutable`	14
[G-03]	Structs can be packed into fewer storage slots	1

	Issue	Instances
[G-04]	Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas	2
[G-05]	Using `storage` instead of `memory` for structs/arrays saves gas	9
[G-06]	Avoid contract existence checks by using solidity version 0.8.10 or later	3
[G-07]	State variables should be cached in stack variables rather than re-reading them from storage	3
[G-08]	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables	1
[G-09]	`internal` functions only called once can be inlined to save gas	3
[G-10]	Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if`-statement	2
[G-11]	`++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for`- and `while`-loops	4
[G-12]	Optimize names to save gas	4
[G-13]	Using `bool`s for storage incurs overhead	1
[G-14]	Use a more recent version of solidity	5
[G-15]	Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement	2
[G-16]	`++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)	4
[G-17]	Usage of `uints`/`ints` smaller than 32 bytes (256 bits) incurs overhead	3
[G-18]	Using `private` rather than `public` for constants, saves gas	3
[G-19]	Division by two should use bit shifting	2
[G-20]	Stack variable used as a cheaper cache for a state variable is only used once	1

	Issue	Instances
[G-21]	`require()` or `revert()` statements that check input arguments should be at the top of the function	2
[G-22]	Superfluous event fields	2
[G-23]	Use custom errors rather than `revert()` / `require()` strings to save gas	42

Total: 114 instances over 23 issues

[G-01] Multiple `address` /ID mappings can be combined into a single `mapping` of an `address` /ID to a `struct` , where appropriate

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (**20000 gas**) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot. Finally, if both fields are accessed in the same function, can save **~42 gas per access** due to [not having to recalculate the key's keccak256 hash](https://gist.github.com/IllIllI000/ec23a57daa30a8f8ca8b9681c8ccefb0) (Gkeccak256 - 30 gas) and that calculation's associated stack operations.

There is 1 instance of this issue:

```
File: contracts/VotingEscrow.sol

58          mapping(address => Point[1000000000]) public userPoint
59:         mapping(address => uint256) public userPointEpoch;
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L58-L59

[G-02] State variables only set in the constructor should be declared `immutable`

Avoids a Gsset (**20000 gas**) in the constructor, and replaces the first access in each transaction (Gcoldsload - **2100 gas**) and each access thereafter (Gwarmacces - **100 gas**) with a `PUSH32` (**3 gas**).

There are 14 instances of this issue:

```
File: contracts/features/Blocklist.sol

/// @audit manager (constructor)
15:            manager = _manager;

/// @audit manager (access)
24:            require(msg.sender == manager, "Only manager");

/// @audit ve (constructor)
16:            ve = _ve;

/// @audit ve (access)
27:            IVotingEscrow(ve).forceUndelegate(addr);
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/features/Blocklist.sol#L15

```
File: contracts/VotingEscrow.sol

/// @audit token (constructor)
107:           token = IERC20(_token);

/// @audit token (access)
426:               token.transferFrom(msg.sender, address(this),

/// @audit token (access)
486:               token.transferFrom(msg.sender, address(this),

/// @audit token (access)
546:           require(token.transfer(msg.sender, value), "Transf

/// @audit token (access)
657:           require(token.transfer(msg.sender, remainingAmount

/// @audit token (access)
```

```
676:            require(token.transfer(penaltyRecipient, amount),

/// @audit name (constructor)
118:            name = _name;

/// @audit symbol (constructor)
119:            symbol = _symbol;

/// @audit decimals (constructor)
115:            decimals = IERC20(_token).decimals();

/// @audit decimals (access)
116:            require(decimals <= 18, "Exceeds max decimals");
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L107

```
diff --git a/contracts/VotingEscrow.sol b/contracts/VotingEscrow
index f15781a..d2c7666 100644
--- a/contracts/VotingEscrow.sol
+++ b/contracts/VotingEscrow.sol
@@ -42,7 +42,7 @@ contract VotingEscrow is IVotingEscrow, Reentr
     event Unlock();

     // Shared global state
-    IERC20 public token;
+    IERC20 public immutable token;
     uint256 public constant WEEK = 7 days;
     uint256 public constant MAXTIME = 365 days;
     uint256 public constant MULTIPLIER = 10**18;
@@ -61,9 +61,9 @@ contract VotingEscrow is IVotingEscrow, Reentr
     mapping(address => LockedBalance) public locked;

     // Voting token
-    string public name;
-    string public symbol;
-    uint256 public decimals = 18;
+    string public constant name = "veFDT";
+    string public constant symbol = "veFDT";
+    uint256 public immutable decimals;

     // Structs
     struct Point {
```

```
@@ -112,11 +112,10 @@ contract VotingEscrow is IVotingEscrow, Re
             blk: block.number
         });

-        decimals = IERC20(_token).decimals();
-        require(decimals <= 18, "Exceeds max decimals");
+        uint256 _dec = IERC20(_token).decimals();
+        require(_dec <= 18, "Exceeds max decimals");
+        decimals = _dec;

-        name = _name;
-        symbol = _symbol;
         owner = _owner;
         penaltyRecipient = _penaltyRecipient;
     }
diff --git a/contracts/features/Blocklist.sol b/contracts/featur
index 27db9b0..ca3a226 100644
--- a/contracts/features/Blocklist.sol
+++ b/contracts/features/Blocklist.sol
@@ -8,8 +8,8 @@ import { IVotingEscrow } from "../interfaces/IVo
 /// @dev This is a basic implementation using a mapping for add
 contract Blocklist {
     mapping(address => bool) private _blocklist;
-    address public manager;
-    address public ve;
+    address public immutable manager;
+    address public immutable ve;

     constructor(address _manager, address _ve) {
         manager = _manager;
```

```
diff --git a/tmp/gas_before b/tmp/gas_after
index 3deb415..cf71599 100644
--- a/tmp/gas_before
+++ b/tmp/gas_after
@@ -167,7 +167,7 @@ No need to generate any newer typings.
 ····················|·······················|··············|·····
 |  Contract         ·  Method               ·  Min         ·  Max
 ····················|·······················|··············|·····
-|  Blocklist        ·  block                ·       45000  ·
+|  Blocklist        ·  block                ·       40797  ·
 ····················|·······················|··············|·····
 |  MockERC20        ·  approve              ·       46176  ·
 ····················|·······················|··············|·····
@@ -177,46 +177,46 @@ No need to generate any newer typings.
```

```
  ·····················|·······················|··············|·····
  |  MockERC20         ·  transfer             ·       51588  ·
  ·····················|·······················|··············|·····
-|  MockSmartWallet   ·  createLock           ·      334002  ·
+|  MockSmartWallet   ·  createLock           ·      331896  ·
  ·····················|·······················|··············|·····
  |  MockSmartWallet   ·  delegate             ·           -  ·
  ·····················|·······················|··············|·····
  |  MockSmartWallet   ·  increaseUnlockTime   ·           -  ·
  ·····················|·······················|··············|·····
-|  MockSmartWallet   ·  quitLock             ·      131158  ·
+|  MockSmartWallet   ·  quitLock             ·      129058  ·
  ·····················|·······················|··············|·····
-|  MockSmartWallet   ·  withdraw             ·           -  ·
+|  MockSmartWallet   ·  withdraw             ·           -  ·
  ·····················|·······················|··············|·····
  |  VotingEscrow      ·  checkpoint           ·       82307  ·     3
  ·····················|·······················|··············|·····
-|  VotingEscrow      ·  collectPenalty       ·           -  ·
+|  VotingEscrow      ·  collectPenalty       ·           -  ·
  ·····················|·······················|··············|·····
-|  VotingEscrow      ·  createLock           ·      293060  ·     3
+|  VotingEscrow      ·  createLock           ·      290954  ·     3
  ·····················|·······················|··············|·····
  |  VotingEscrow      ·  delegate             ·      246709  ·     4
  ·····················|·······················|··············|·····
-|  VotingEscrow      ·  increaseAmount       ·      234777  ·     1
+|  VotingEscrow      ·  increaseAmount       ·      232671  ·     1
  ·····················|·······················|··············|·····
  |  VotingEscrow      ·  increaseUnlockTime   ·       46794  ·
  ·····················|·······················|··············|·····
-|  VotingEscrow      ·  quitLock             ·      127639  ·     1
+|  VotingEscrow      ·  quitLock             ·      125539  ·     1
  ·····················|·······················|··············|·····
  |  VotingEscrow      ·  unlock               ·           -  ·
  ·····················|·······················|··············|·····
  |  VotingEscrow      ·  updateBlocklist      ·           -  ·
  ·····················|·······················|··············|·····
-|  VotingEscrow      ·  withdraw             ·      106462  ·     3
+|  VotingEscrow      ·  withdraw             ·      104356  ·     3
  ·····················|·······················|··············|·····
  |  Deployments                               ·
  ·············································|··············|·····
-|  Blocklist                                 ·      278212  ·
+|  Blocklist                                 ·      248613  ·
  ·············································|··············|·····
```

```
 |  MockERC20                                 ·             -   ·
 ·······································|·············|·····
 |  MockSmartWallet                           ·             -   ·
 ·······································|·············|·····
-|  VotingEscrow                              ·       4374338   ·     4
+|  VotingEscrow                              ·       4280168   ·     4
 ·--------------------------------------|-------------|-----


-  117 passing (48s)
+  117 passing (46s)
```

[G-03] Structs can be packed into fewer storage slots

Each slot saved can avoid an extra Gsset (**20000 gas**) for the first setting of the struct. Subsequent reads as well as writes have smaller gas savings

There is 1 instance of this issue:

```
File: contracts/VotingEscrow.sol

/// @audit Variable ordering with 3 slots instead of the current
///           uint256(32):end, address(20):delegatee, int128(16)
75        struct LockedBalance {
76            int128 amount;
77            uint256 end;
78            int128 delegated;
79            address delegatee;
80:       }
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L75-L80

```
diff --git a/contracts/VotingEscrow.sol b/contracts/VotingEscrow
index f15781a..0318bc3 100644
--- a/contracts/VotingEscrow.sol
+++ b/contracts/VotingEscrow.sol
@@ -73,10 +73,10 @@ contract VotingEscrow is IVotingEscrow, Reen
         uint256 blk;
     }
     struct LockedBalance {
```

```diff
-         int128 amount;
          uint256 end;
-         int128 delegated;
          address delegatee;
+         int128 amount;
+         int128 delegated;
      }

      // Miscellaneous
@@ -420,7 +420,7 @@ contract VotingEscrow is IVotingEscrow, Reen
          locked_.delegated += int128(int256(_value));
          locked_.delegatee = msg.sender;
          locked[msg.sender] = locked_;
-         _checkpoint(msg.sender, LockedBalance(0, 0, 0, address(
+         _checkpoint(msg.sender, LockedBalance({amount:0, end:0,
          // Deposit locked tokens
          require(
              token.transferFrom(msg.sender, address(this), _valu
diff --git a/test/votingEscrowDelegationMathTest.ts b/test/votin
index 5e43096..088fb9c 100644
--- a/test/votingEscrowDelegationMathTest.ts
+++ b/test/votingEscrowDelegationMathTest.ts
@@ -138,10 +138,10 @@ describe("VotingEscrow Delegation Math tes
    };

    interface LockedBalance {
-     amount: BN;
      end: BN;
-     delegated: BN;
      delegatee: string;
+     amount: BN;
+     delegated: BN;
    }

    interface Point {
@@ -173,10 +173,10 @@ describe("VotingEscrow Delegation Math tes
        epoch,
        userEpoch,
        userLocked: {
-         amount: locked[0],
-         end: locked[1],
-         delegated: locked[2],
-         delegatee: locked[3],
+         end: locked[0],
+         delegatee: locked[1],
+         amount: locked[2],
+         delegated: locked[3],
        },
        userLastPoint: {
          bias: userLastPoint[0],
diff --git a/test/votingEscrowGasTest.ts b/test/votingEscrowGasT
index d6d03fd..af94f35 100644
--- a/test/votingEscrowGasTest.ts
+++ b/test/votingEscrowGasTest.ts
@@ -174,10 +174,10 @@ describe("Gas usage tests", () => {
        epoch,
        userEpoch,
        userLocked: {
-         amount: locked[0],
-         end: locked[1],
-         delegated: locked[2],
-         delegatee: locked[3],
+         end: locked[0],
+         delegatee: locked[1],
+         amount: locked[2],
+         delegated: locked[3],
        },
        userLastPoint: {
          bias: userLastPoint[0],
diff --git a/test/votingEscrowMathTest.ts b/test/votingEscrowMat
index 9d0b9b6..e084b30 100644
--- a/test/votingEscrowMathTest.ts
+++ b/test/votingEscrowMathTest.ts
@@ -207,8 +207,10 @@ describe("VotingEscrow Math test", () => {
    });

    interface LockedBalance {
-     amount: BN;
      end: BN;
+     delegatee: string;
+     amount: BN;
+     delegated: BN;
    }

    interface Point {
@@ -252,8 +254,10 @@ describe("VotingEscrow Math test", () => {
        //totalStaticWeight: await votingLockup.totalStaticWeight
        // userStaticWeight: await votingLockup.staticBalanceOf(s
        userLocked: {
-         amount: locked[0],
-         end: locked[1],
+         end: locked[0],
+          delegatee: locked[1],
+          amount: locked[2],
+          delegated: locked[3],
         },
         userLastPoint: {
           bias: userLastPoint[0],
```

```
diff --git a/tmp/gas_before b/tmp/gas_after
index 3deb415..09dd64b 100644
--- a/tmp/gas_before
+++ b/tmp/gas_after
@@ -167,7 +167,7 @@ No need to generate any newer typings.
  ·····················|·······················|··············|·····
  |  Contract           ·  Method                ·  Min         ·  Max
  ·····················|·······················|··············|·····
-|  Blocklist          ·  block                 ·       45000  ·
+|  Blocklist          ·  block                 ·       42887  ·
  ·····················|·······················|··············|·····
  |  MockERC20          ·  approve               ·       46176  ·
  ·····················|·······················|··············|·····
@@ -177,35 +177,35 @@ No need to generate any newer typings.
  ·····················|·······················|··············|·····
  |  MockERC20          ·  transfer              ·       51588  ·
  ·····················|·······················|··············|·····
-|  MockSmartWallet    ·  createLock            ·      334002  ·
+|  MockSmartWallet    ·  createLock            ·      311601  ·
  ·····················|·······················|··············|·····
-|  MockSmartWallet    ·  delegate              ·           -  ·
+|  MockSmartWallet    ·  delegate              ·           -  ·
  ·····················|·······················|··············|·····
-|  MockSmartWallet    ·  increaseUnlockTime    ·           -  ·
+|  MockSmartWallet    ·  increaseUnlockTime    ·           -  ·
  ·····················|·······················|··············|·····
-|  MockSmartWallet    ·  quitLock              ·      131158  ·
+|  MockSmartWallet    ·  quitLock              ·      140849  ·
  ·····················|·······················|··············|·····
-|  MockSmartWallet    ·  withdraw              ·           -  ·
+|  MockSmartWallet    ·  withdraw              ·           -  ·
  ·····················|·······················|··············|·····
-|  VotingEscrow       ·  checkpoint            ·       82307  ·     3
+|  VotingEscrow       ·  checkpoint            ·       82319  ·     3
  ·····················|·······················|··············|·····
  |  VotingEscrow       ·  collectPenalty        ·           -  ·
  ·····················|·······················|··············|·····
-|  VotingEscrow       ·  createLock            ·      293060  ·     3
```

```
+|  VotingEscrow      ·  createLock             ·      270653  ·    3
  ······················|·························|···············|······
-|  VotingEscrow      ·  delegate               ·      246709  ·    4
+|  VotingEscrow      ·  delegate               ·      224688  ·    4
  ······················|·························|···············|······
-|  VotingEscrow      ·  increaseAmount         ·      234777  ·    1
+|  VotingEscrow      ·  increaseAmount         ·      229422  ·    1
  ······················|·························|···············|······
-|  VotingEscrow      ·  increaseUnlockTime     ·       46794  ·
+|  VotingEscrow      ·  increaseUnlockTime     ·       44338  ·
  ······················|·························|···············|······
-|  VotingEscrow      ·  quitLock               ·      127639  ·    1
+|  VotingEscrow      ·  quitLock               ·      137330  ·    1
  ······················|·························|···············|······
 |  VotingEscrow      ·  unlock                 ·           -  ·
  ······················|·························|···············|······
 |  VotingEscrow      ·  updateBlocklist        ·           -  ·
  ······················|·························|···············|······
-|  VotingEscrow      ·  withdraw               ·      106462  ·    3
+|  VotingEscrow      ·  withdraw               ·      116189  ·    3
  ······················|·························|···············|······
 |  Deployments                                 ·
  ················································|···············|······
@@ -215,8 +215,8 @@ No need to generate any newer typings.
  ················································|···············|······
 |  MockSmartWallet                             ·           -  ·
  ················································|···············|······
-|  VotingEscrow                                ·     4374338  ·    4
+|  VotingEscrow                                ·     4245313  ·    4
  ·-----------------------------------------------|---------------|-----


-  117 passing (48s)
+  117 passing (44s)
```

[G-04] Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. **Each iteration of this for-loop costs at least 60 gas** (i.e. `60 * <mem_array>.length`). Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution. Note that even if an interface defines a function as having `memory` arguments, it's still valid for implementation contracs to use `calldata` arguments instead.

If the array is passed to an `internal` function which passes the array to another internal function where the array is modified and therefore `memory` is used in the `external` call, it's still more gass-efficient to use `calldata` when the `external` function uses modifiers, since the modifiers may prevent the internal functions from being called. Structs have the same overhead as an array of length one

Note that I've also flagged instances where the function is `public` but can be marked as `external` since it's not called by the contract, and cases where a constructor is involved

There are 2 instances of this issue:

```
File: contracts/VotingEscrow.sol

/// @audit _name
/// @audit _symbol
100        constructor(
101            address _owner,
102            address _penaltyRecipient,
103            address _token,
104            string memory _name,
105:           string memory _symbol
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L100-L105

[G-05] Using `storage` instead of `memory` for structs/arrays saves gas

When fetching data from a storage location, assigning the data to a `memory` variable causes all fields of the struct/array to be read from storage, which incurs a Gcoldsload (**2100 gas**) for *each* field of the struct/array. If the fields are read from the new memory variable, they incur an additional `MLOAD` rather than a cheap stack read. Instead of declearing the variable with the `memory` keyword, declaring the variable with the `storage` keyword and caching any fields that need to be re-read in stack variables, will be much cheaper, only incuring the Gcoldsload for the fields actually read. The only time it makes sense to read the whole struct/array into a `memory` variable, is if the full struct/array is being returned by the function, is being passed to a function that requires `memory`, or if the array/struct is being read from another `memory` array/struct

There are 9 instances of this issue:

```
File: contracts/VotingEscrow.sol

410:            LockedBalance memory locked_ = locked[msg.sender];

446:            LockedBalance memory locked_ = locked[msg.sender];

499:            LockedBalance memory locked_ = locked[msg.sender];

527:            LockedBalance memory locked_ = locked[msg.sender];

561:            LockedBalance memory locked_ = locked[msg.sender];

633:            LockedBalance memory locked_ = locked[msg.sender];

788:            Point memory point0 = pointHistory[epoch];

866:            Point memory lastPoint = pointHistory[epoch_];

882:            Point memory point = pointHistory[targetEpoch];
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L410

[G-06] Avoid contract existence checks by using solidity version 0.8.10 or later

Prior to 0.8.10 the compiler inserted extra code, including `EXTCODESIZE` (**100 gas**), to check for contract existence for external calls. In more recent solidity versions, the compiler will not insert these checks if the external call has a return value

There are 3 instances of this issue:

```
File: contracts/VotingEscrow.sol

/// @audit decimals()
115:            decimals = IERC20(_token).decimals();

/// @audit isBlocked()
126:                !IBlocklist(blocklist).isBlocked(msg.sender),

/// @audit isBlocked()
563:            require(!IBlocklist(blocklist).isBlocked(_addr), "
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L115

[G-07] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replace each Gwarmaccess (**100 gas**) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

There are 3 instances of this issue:

```
File: contracts/VotingEscrow.sol

/// @audit penaltyRecipient on line 676
677:            emit CollectPenalty(amount, penaltyRecipient);

/// @audit pointHistory on line 788
796:                Point memory point1 = pointHistory[epoch + 1];

/// @audit pointHistory on line 882
891:                Point memory pointNext = pointHistory[targetEp
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L677

[G-08] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

Using the addition operator instead of plus-equals saves 113 gas

There is 1 instance of this issue:

```
File: contracts/VotingEscrow.sol

654:            penaltyAccumulated += penaltyAmount;
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L654

[G-09] `internal` functions only called once can be inlined to save gas

Not inlining costs **20 to 40 gas** because of two extra `JUMP` instructions and additional stack operations needed for function calls.

There are 3 instances of this issue:

```
File: contracts/features/Blocklist.sol

37:        function _isContract(address addr) internal view retur
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/features/Blocklist.sol#L37

```
File: contracts/VotingEscrow.sol
```

```
662        function _calculatePenaltyRate(uint256 end)
663            internal
664            view
665:           returns (uint256)

732        function _findUserBlockEpoch(address _addr, uint256 _b
733            internal
734            view
735:           returns (uint256)
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L662-L665

[G-10] Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` -statement

`require(a <= b); x = b - a` => `require(a <= b); unchecked { x = b - a }`

There are 2 instances of this issue:

```
File: contracts/VotingEscrow.sol

/// @audit if-condition on line 299
301:                       (MULTIPLIER * (block.number - lastPoint.bl

/// @audit if-condition on line 299
302:                       (block.timestamp - lastPoint.ts);
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L301

[G-11] `++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for

them to overflow, as is the case when used in `for` - and `while` -loops

The `unchecked` keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves **30-40 gas** [per loop](https://gist.github.com/hrkrshnn/ee8fabd532058307229d65dcd5836ddc#the-increment-in-for-loop-post-condition-can-be-made-unchecked)

There are 4 instances of this issue:

```
File: contracts/VotingEscrow.sol

309:            for (uint256 i = 0; i < 255; i++) {

717:            for (uint256 i = 0; i < 128; i++) {

739:            for (uint256 i = 0; i < 128; i++) {

834:            for (uint256 i = 0; i < 255; i++) {
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L309

[G-12] Optimize names to save gas

`public` / `external` function names and `public` member variable names can be optimized to save gas. See [this]() link for an example of how it works. Below are the interfaces/abstract contracts that can be optimized so that the most frequently-called functions use the least amount of gas possible during method lookup. Method IDs that have two leading zero bytes can save **128 gas** each during deployment, and renaming functions to have lower method IDs will save **22 gas** per call, [per sorted position shifted]()

There are 4 instances of this issue:

```
File: contracts/features/Blocklist.sol

/// @audit block(), isBlocked()
9:    contract Blocklist {
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/features/Blocklist.sol#L9

```
File: contracts/interfaces/IBlocklist.sol

/// @audit isBlocked()
6:    interface IBlocklist {
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/interfaces/IBlocklist.sol#L6

```
File: contracts/interfaces/IVotingEscrow.sol

/// @audit createLock(), increaseAmount(), increaseUnlockTime(),
4:    interface IVotingEscrow {
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/interfaces/IVotingEscrow.sol#L4

```
File: contracts/VotingEscrow.sol

/// @audit updateBlocklist(), updatePenaltyRecipient(), unlock()
23:   contract VotingEscrow is IVotingEscrow, ReentrancyGuard {
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L23

[G-13] Using `bool` s for storage incurs overhead

```
    // Booleans are more expensive than uint256 or any type that
    // word because each write operation emits an extra SLOAD to
    // slot's contents, replace the bits taken up by the boolean
```

```
        // back. This is the compiler's defense against contract upg
        // pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27 Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas) for the extra SLOAD, and to avoid Gsset (**20000 gas**) when changing from `false` to `true`, after having been `true` in the past

There is 1 instance of this issue:

```
File: contracts/features/Blocklist.sol

10:        mapping(address => bool) private _blocklist;
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/features/Blocklist.sol#L10

[G-14] Use a more recent version of solidity

Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

There are 5 instances of this issue:

```
File: contracts/features/Blocklist.sol

2:    pragma solidity ^0.8.3;
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/features/Blocklist.sol#L2

```
File: contracts/interfaces/IBlocklist.sol

2:    pragma solidity ^0.8.3;
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/interfaces/IBlocklist.sol#L2

```
File: contracts/interfaces/IERC20.sol

2:    pragma solidity ^0.8.3;
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/interfaces/IERC20.sol#L2

```
File: contracts/interfaces/IVotingEscrow.sol

2:    pragma solidity ^0.8.3;
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/interfaces/IVotingEscrow.sol#L2

```
File: contracts/VotingEscrow.sol

2:    pragma solidity ^0.8.3;
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L2

[G-15] Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement

This change saves [6 gas](#) per instance. The optimization works until solidity version [0.8.13](#) where there is a regression in gas costs.

There are 2 instances of this issue:

```
File: contracts/VotingEscrow.sol

412:            require(_value > 0, "Only non zero amount");

448:            require(_value > 0, "Only non zero amount");
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L412

[G-16] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)

Saves **5 gas per loop**

There are 4 instances of this issue:

```
File: contracts/VotingEscrow.sol

309:            for (uint256 i = 0; i < 255; i++) {

717:            for (uint256 i = 0; i < 128; i++) {

739:            for (uint256 i = 0; i < 128; i++) {

834:            for (uint256 i = 0; i < 255; i++) {
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L309

[G-17] Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead

> When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html Each operation involving a `uint8` costs an extra 22-28 gas (depending on whether the other operand is also a variable of type `uint8`) as compared to ones involving `uint256` , due to the compiler having to clear the higher bits of the memory word before operating on the `uint8` , as well as the associated stack operations of doing so. Use a larger size then downcast where needed

There are 3 instances of this issue:

```
File: contracts/VotingEscrow.sol

/// @audit int128 oldSlopeDelta
380:                    oldSlopeDelta = oldSlopeDelta + userOldPoi

/// @audit int128 oldSlopeDelta
382:                        oldSlopeDelta = oldSlopeDelta - userNe

/// @audit int128 newSlopeDelta
388:                        newSlopeDelta = newSlopeDelta - userNe
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L380

[G-18] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

There are 3 instances of this issue:

```
File: contracts/VotingEscrow.sol

46:        uint256 public constant WEEK = 7 days;

47:        uint256 public constant MAXTIME = 365 days;

48:        uint256 public constant MULTIPLIER = 10**18;
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L46

[G-19] Division by two should use bit shifting

`<x> / 2` is the same as `<x> >> 1`. While the compiler uses the `SHR` opcode to accomplish both, the version that uses division incurs an overhead of [20 gas](https://gist.github.com/IllIllI000/ec0e4e6c4f52a6bca158f137a3afd4ff) due to `JUMP`s to and from a compiler utility function that introduces checks which can be avoided by using `unchecked {}` around the division by two

There are 2 instances of this issue:

```
File: contracts/VotingEscrow.sol

719:                uint256 mid = (min + max + 1) / 2;

743:                uint256 mid = (min + max + 1) / 2;
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L719

[G-20] Stack variable used as a cheaper cache for a state variable is only used once

If the variable is only accessed once, it's cheaper to use the state variable directly that one time, and save the **3 gas** the extra stack assignment would spend

There is 1 instance of this issue:

```
File: contracts/VotingEscrow.sol

865:          uint256 epoch_ = globalEpoch;
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L865

[G-21] `require()` or `revert()` statements that check input arguments should be at the top of the function

Checks that involve constants should come before checks that involve state variables, function calls, and calculations. By doing these checks first, the function is able to revert before wasting a Gcoldsload (**2100 gas***) in a function that may ultimately revert in the unhappy case.

There are 2 instances of this issue:

```
File: contracts/VotingEscrow.sol

/// @audit expensive op on line 410
412:          require(_value > 0, "Only non zero amount");

/// @audit expensive op on line 446
448:          require(_value > 0, "Only non zero amount");
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L412

[G-22] Superfluous event fields

`block.timestamp` and `block.number` are added to event information by default so adding them manually wastes gas

There are 2 instances of this issue:

```
File: contracts/VotingEscrow.sol

30:            uint256 ts

36:            uint256 ts
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L30

[G-23] Use custom errors rather than `revert()` / `require()` strings to save gas

Custom errors are available from solidity version 0.8.4. Custom errors save [~50 gas]() each time they're hit by [avoiding having to allocate and store the revert string](). Not defining the strings also save deployment gas

There are 42 instances of this issue:

```
File: contracts/features/Blocklist.sol

24:            require(msg.sender == manager, "Only manager");

25:            require(_isContract(addr), "Only contracts");
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/features/Blocklist.sol#L24

```
File: contracts/VotingEscrow.sol
```

```
116:            require(decimals <= 18, "Exceeds max decimals");

125             require(
126                 !IBlocklist(blocklist).isBlocked(msg.sender),
127                 "Blocked contract"
128:            );

140:            require(msg.sender == owner, "Only owner");

147:            require(msg.sender == owner, "Only owner");

154:            require(msg.sender == owner, "Only owner");

162:            require(msg.sender == owner, "Only owner");

171:            require(msg.sender == blocklist, "Only Blocklist")

412:            require(_value > 0, "Only non zero amount");

413:            require(locked_.amount == 0, "Lock exists");

414:            require(unlock_time >= locked_.end, "Only increase

415:            require(unlock_time > block.timestamp, "Only futur

416:            require(unlock_time <= block.timestamp + MAXTIME,

425             require(
426                 token.transferFrom(msg.sender, address(this),
427                 "Transfer failed"
428:            );

448:            require(_value > 0, "Only non zero amount");

449:            require(locked_.amount > 0, "No lock");

450:            require(locked_.end > block.timestamp, "Lock expir

469:                require(locked_.amount > 0, "Delegatee has no

470:                require(locked_.end > block.timestamp, "Delega

485             require(
486                 token.transferFrom(msg.sender, address(this),
487                 "Transfer failed"
```

```
488:            );

502:            require(locked_.amount > 0, "No lock");

503:            require(unlock_time > locked_.end, "Only increase

504:            require(unlock_time <= block.timestamp + MAXTIME,

511:                require(oldUnlockTime > block.timestamp, "Lock

529:            require(locked_.amount > 0, "No lock");

530:            require(locked_.end <= block.timestamp, "Lock not

531:            require(locked_.delegatee == msg.sender, "Lock del

546:            require(token.transfer(msg.sender, value), "Transf

563:            require(!IBlocklist(blocklist).isBlocked(_addr), "

564:            require(locked_.amount > 0, "No lock");

565:            require(locked_.delegatee != _addr, "Already deleg

587:            require(toLocked.amount > 0, "Delegatee has no loc

588:            require(toLocked.end > block.timestamp, "Delegatee

589:            require(toLocked.end >= fromLocked.end, "Only dele

635:            require(locked_.amount > 0, "No lock");

636:            require(locked_.end > block.timestamp, "Lock expir

637:            require(locked_.delegatee == msg.sender, "Lock del

657:            require(token.transfer(msg.sender, remainingAmount

676:            require(token.transfer(penaltyRecipient, amount),

776:            require(_blockNumber <= block.number, "Only past b

877:            require(_blockNumber <= block.number, "Only past b
```

https://github.com/code-423n4/2022-08-fiatdao/blob/fece3bdb79ccacb501099c24b60312cd0b2e4bb2/contracts/VotingEscrow.sol#L116

lacoop6tu (FIAT DAO) commented:

> Good one

Mitigation Review

Mitigation review by IllIllI

Contest repository commit: https://github.com/code-423n4/2022-08-fiatdao/tree/fece3bdb79ccacb501099c24b60312cd0b2e4bb2
Project repository commit: https://github.com/fiatdao/veFDT/tree/8e88edb452b73bbe3461a8f4a5ed502db140322a
Final commit: https://github.com/fiatdao/veFDT/tree/3f822125e05e2927eab0a3a3e797508b363083ab

Mitigation PRs reviewed:

- Issue 2: _checkpoint function won't be called for a user which is both a delegator and a delegatee in increaseUnlockTime
- Issue 3: Unnecessary if statement in _checkpoint
- Issue 4: Lock owner can delegate after lock expiration
- Issue 5: increaseUnlockTime uses wrong unlock time for old lock.
- Issue 6: Delegators can Avoid Lock Commitments if they can Reliably get Themselves Blocked when Needed
- Issues 7 & 8: QA and gas optimization
- Issue 18: Use safe transfer methods

Intro

veFDT is a solidity implementation of Curve's voting-escrow, enhanced to give users the ability to delegate their locked tokens, to quit their locks early for a penalty, and to have their smart wallets optimistically approved.

Disclaimer

This mitigation review does not guarantee the absence of any further vulnerabilities, being, however, the result of exercising the reviewer's best efforts. At the time of the review, the final judging report was not yet available, so in addition to covering the resolved issues, the review also included an investigation of the disputed and acknowledged issues, as well as the issues downgraded to QA.

Contest issues overview

The following is a high-level overview of the issues identified during FiatDAO's August 2022 Code4rena audit contest, and any potential changes/feedback provided by the project sponsors in both the contest issue comments as well as the PRs listed above.

High and Medium Risk Issues

- [H-01] - **Unsafe usage of ERC20 transfer and transferFrom** (sponsor disputed)

  The sponsor disputed the issue because the token it's planned to be used with does correctly return a boolean. However, the sponsor decided to make a change to address the finding as Issue 18. The fix properly replaces the `require()` statements that check for successful transfers, with calls to OpenZeppelin's `safeTransfer()`. The PR also replaces the internal definition of the `IERC20` interface with OpenZeppelin's version. The prior version of the code's `IERC20` included the function `decimals()`, which is not one of the required functions for the interface, so it's possible for the code to encounter a token without this function, but it would be immediately apparent what happened because the constructor is the function that calls `decimals()`. The change to using OpenZeppelin required making this distinction more visible due to the fact that they're defined separately as `IERC20` and `IERC20Metadata`. The new code is not checking that the token actually supports the function (e.g. using a `safeDecimals()`-like function), but it is not any worse off that it had been prior to the change.

- [H-02] - **Delegators can Avoid Lock Commitments if they can Reliably get Themselves Blocked when Needed** (sponsor confirmed, severity disagreement)

  The sponsor disagreed with the severity and the judge updated the issue to be of Medium risk, and I agree with that severity. The finding was addressed via the fix for Issue 6 where the sponsor implemented the suggestion of the warden, to use the delegatee's lock endpoint in the re-delegation to self, rather than using the delegator's existing endpoint, since that endpoint may be far in the past. The delegate() and undelegate() functions have checks to ensure that the target for the votes always has at least as long a duration as the source of the votes. The fix enforces the same requirement for `forceUndelegate()` by assigning a longer duration.

  There are only two places in the code that change `LockedBalance.end` to a smaller value, which could possibly violate the contract invariants: in `quitLock()` where the struct is never written back to storage, and in `withdraw()` where it is indeed written back to storage. However, if the delegatee was able to withdraw, that means the delegator already would have been able to withdraw (since the delegatee's timestamp must always be greater than or equal to the delegator's when delegating or increasing), and therefore the mitigation is correct. The only extra wrinkle that the change makes, is that it now allows a malicious delegatee to front-run a delegator's block with an `increaseUnlock(MAXTIME)`, but it's not clear what advantage that would give the delegatee, and furthermore, the delegator already put his/her trust in the delegatee, so it's something that could have occurred anyway, even without a call to `forceUndelegate()`.

- [M-01] - **The current implementation of the VotingEscrow contract doesn't support fee on transfer tokens** (sponsor disputed)

  The sponsor disputed the issue because the Balancer V2 Pool tokens it's planned to be used with do not implement a fee-on-transfer mechanic. The tokens do not appear to be upgradeable so there is no risk of fees being added to existing tokens via upgrade. Without more information about how which pool tokens are chosen/allowed/used, and what prevents future pool tokens that implement such a mechanic from being used with the same contract, I have to agree with the warden and judge that this is a Medium risk issue. The suggested mitigation is to measure the balance of the token that the contract holds before the `transferFrom()` call is made, and afterwards, and use the difference as the value, rather than the amount the user states. You could also add a `require()` enforcing the invariant that the change in balance must equal the stated amount, which would *prevent* fee-on-transfer tokens from being used.

In the final PR, the sponsor has acknowledged the issue and added a code comment saying that fee-on-transfer tokens are not supported.

- [M-02] - **Attacker contract can avoid being blocked by BlockList.sol** (sponsor acknowledged)

  The sponsor acknowledges that the `BlockList` can be bypassed, but states that “the only interaction possible is locking LP tokens first”. However, looking at the code, the `checkBlocklist` modifier is applied to not just `createLock()`, but `increaseAmount()`, `increaseUnlockTime()`, and `delegate()`. An attacker can bypass the block list for every one of these functions by making their SmartWallet a specially-constructed `create2()` contract that does external calls to an other contract in its constructor, for instructions on what to execute, before self-destructing. Whenever the attacker wants to interact with the token, they update their external instruction-providing contract with the action to take, re-create the attack contract. It’s not clear why the block list is only for contracts, and if it can be bypassed by using this method, or by transferring the tokens to an EOA.

  In discussions of the issue, the sponsor clarified that the `BlockList`’s purpose is to prevent lock tokenization, and acknowledged that using an updated `BlockList` that blocks specific EOAs may be required if an attacker uses the features described above to work around being blocked.

- [M-03] - **Inconsistent logic of increase unlock time to the expired locks** (sponsor confirmed)

  The sponsor addressed the finding with the fix for Issue 4. The fix chosen was to not allow the increasing of lock time or non-self re-delegation if the delegatee’s lock has expired. The fix didn’t require the undelegate flavor to duplicate the blocklist check since `msg.sender` is already checked by the `checkBlocklist` modifier. The refactored code properly re-used some variables rather than duplicating the allocations done in the delegation/re-delegation case in order to save some gas. The refactoring introduced a new issue, [M.N-01], described below.

- [M-04] - **ERROR IN UPDATING `_checkpoint` IN THE `increaseUnlockTime` FUNCTION** (sponsor confirmed)

  The sponsor addressed the finding with the fix for Issue 5. The fix properly changes the code to match the invariants specification, and matches the logical expectation that the 'old' field uses the 'old' timestamp.

- [M-05] - **Unsafe casting from int128 can cause wrong accounting of locked amounts** (sponsor acknowledged)

  The sponsor acknowledges that overflow is technically possible, but considers this as unlikely to happen in practice.

  In the final PR, the sponsor added a code comment saying that the contract does not support tokens where `maxSupply>2^128-10^[decimals]`.

- [M-06] - `increaseUnlockTime` **missing** `_checkpoint` **for delegated values** (sponsor confirmed)

  The sponsor addressed the finding with the fix for Issue 2. In cases where a user is both a delegator and a delegatee, the original code did not create a checkpoint for calls to `increaseUnlockTime()`. Self-delegation and being delegated to both increase the `LockedBalance.delegated` field, so the change to the condition of the if-statement now includes both cases. The code will not get to the if-statement if the user has already withdrawn, due to a `require()`, so a user that has delegates but has withdrawn, cannot increase their now-zero unlock time. `increaseAmount()` has a similar if-statement and comment, but the else-block is already covered by a checkpoint, so there is no analogous issue there.

- [M-07] - **Blocking Through Change of Blocklist Could Trap Tokens** (sponsor acknowledged)

  The sponsor acknowledges the possible rug vector. It is common for admin rug vectors to not be addressed.

- [M-08] - **Attackers can abuse the quitLock function to get a very large amount of votes** (sponsor disputed)

  The sponsor disputed the issue on the basis of an economic analysis of the penalty taken vs the votes gained, the outcome of which was that the penalty always covers the votes gained. I spoke with the sponsor and the sponsor explained that for the original Curve Finance code, the number of votes one gets is not equal one-for-one to the number of tokens locked: locking for the maximum duration will get you close to one-for-one, but every second under that number, the locking user gets fewer and fewer votes. Therefore in veFTD, the penalty is always chosen such that when the penalty is subtracted from the votes gained, the number of votes a user is left with after quitting is equal to the number of votes they would have been given had they used the quit time as their lock end time instead. In other words, the votes one would get for locking for the week the warden mentions, would be equal to the penalty they gather ahead of time, so the flash loan does not help the attacker.

  To verify all of this, I wrote two tests that make use of hardhat's ability to mine specific blocks with specific times. The first test confirmed that indeed, when one subtracts the penalty from the number of votes gained, the remaining number of votes is less than the number of votes a separate user gains for locking for the shorter duration, so it's always better to specify the correct lock time rather and unlocking early. The **second** test verifies that the same is true even if one quits the second after the lock is created. Finally, I wrote a **test** that specifically does the flash loan scenario the warden outlined, and was able to show that when the attacking contract checks its balance between locking and quitting for the previous block, the votes are zero, and for the current block, the penalty is larger than the votes gained (and one cannot query vote balances for future blocks). Once the contract's attack call completes, checks for the votes for same blocks show zero votes for both, so I believe the warden's finding is invalid.

- [Issue 227] - **Wrong penalty allocation computation** (invalid)

  The sponsor walks though an example where a user is able to quit without a penalty, given a specific number of decimals, due to loss of precision. While having a smaller number of decimals can increase the value of each unit to the point where one wei is a worth-while amount, another way for each wei to increase in value is for the majority of the outstanding tokens to be burned. Yet another way to take advantage of the incorrect penalty is to split a large number of tokens into separate smaller chunks. As long as the gas cost to transfer the tokens, lock them, then unlock them is smaller than the economic value gained from the votes, it will be worth while to do the attack.

I was able to write a test that shows that by distributing tokens among nyms, an attacker is able to pay zero penalty on tiny amounts of wei because the loss of precision favors the quitter rather than the penalty recipient. In addition to fixing the loss of precision mentioned in Q-67 below, this advantage given to users that split their tokens among multiple addresses can be further mitigated by always ceil-ing fractional amounts:

```
- uint256 penaltyAmount = (value * penaltyRate) / 10**18; // quitlock_p
+ uint256 penaltyAmount = (value * penaltyRate + (10**18 - 1)) / 10**18
```

A simpler alternative is to just add a penalty of one wei to all quits.

The sponsor disputes the part of the issue relating to a user being able to avoid paying any penalties on the basis of the fact that it only makes economic sense to do the attack if the value of one wei of the token is larger than the gas cost to create and quit a lock, otherwise the attacker would be better off losing the locked tokens to a penalty instead. The sponsor acknowledges that the rounding is done in the favor of the user having the penalty applied, rather than in the favor of the penalty recipient.

Low Risk, Non-Critical, and Gas

- [Issue 230] - **Divide before multiply in slope calculations** (sponsor acknowledged)

  The sponsor acknowledges the possible rounding issue, but points out that if there is one, it's in the Curve code base too. Curve, however, uses four years rather than veFTD's one year, so the math may be slightly different.

- [Issue 60] - **Delegation would be successful when delegatee lock expiration is same as delegator's lock expiration.** (sponsor disputed)

  The sponsor states that the documentation is wrong, not the code. Agreed, but there is no documentation change in the provided PRs.

The sponsor provided a followup PR that properly updates the documentation.

- [Issue 248] - **First `userPointHistory` is never recorded** (duplicate of Issue 294)

  Duplicate of [Issue 294] below

- [Issue 162] - **Miscalculation in `_calculatePenaltyRate` function** (sponsor confirmed)

  The issue is that the 365-day `MAXTIME` is not divisible cleanly by `WEEK` (527 = *364), so due to the flooring that takes place, a user can never get his/her maximum voting power. I confirmed with a test that calling ̀createLock((WEEK*53)-1)` results in a lock duration of 364 days. This issue has not been addressed by any of the PRs provided by the sponsor.

  In discussions with the sponsor, the sponsor acknowledges the issue and will let the values remain as-is, since the issue only means that users are more disincentivized to quit early, since they will pay a slightly higher fee than if the two values were cleanly divisible.

- [Issue 114] - **penaltyAmount is deflated and remainingAmount is inflated when calling quitLock function of VotingEscrow contract according to current implementation** (sponsor acknowledged)

  The sponsor acknowledges loss of precision leading to dust amounts not being accounted for properly, due to multiplication after division

- [Issue 117] - **VotingEscrow implementation does not match specification** (sponsor disputed)

  Same as [Issue 60] above

- [Issue 152] - **The unlockTime ≥ owner.end in createLock function is inconsistent with design.** (sponsor disputed)

  Same as [Issue 60] above

- [Issue 294] - **Wrong logic in `_checkpoint()` function might lead to wrong value of `balanceOfAt()`, `totalSupplyAt()`** (sponsor confirmed)

The sponsor addressed the finding with the fix for Issue 3. The fix removes the setting of the first epoch, which is safe to do since everywhere that the contract references the `userPointHistory`, index zero is always special-cased to zero. Furthermore, there is no case where a `Point` is written with non-zero values, which would cause the public function `userPointHistory(addr,0)` to return wrong values. I would also note that the code prior to the fix was using `uEpoch + 1` whereas it should have been using `uEpoch` instead.

- **Miscellaneous**

  The sponsor addressed a subset of the QA and gas findings with the fix for Issues 7 & 8. Some of the gas fixes were related to `for`-loops, splitting `+=`, state variable caching, `!=` in `require()`s, converting storage to immutables, removing casts where not necessary, and adding comments explaining the safety of mathematical operations, all of which were done properly. The PR also switched the rest of the code from using internal definitions of the ERC-20 interface, to using the OpenZeppelin versions (the changes started as a part of Issue 18), and renamed the `block()` function.

Findings

Findings are labeled with the following notation `M.S-N`, representing `Mitigation.Severity-Number`.

[M.L-01] Code does not follow check-effects-interaction best practice

One of the PRs under review for this mitigation, not tied to any issue identified during the contest, was PR 22 where the sponsor introduced a new state variable that tracks the net number of tokens held by the contract, excluding any tokens externally transferred to the contract. While the code does properly track the net number (not including any discrepancies due to fee-on-transfer tokens already excluded by the sponsor), it does so by updating the state after the external calls that handle the transfer of tokens into and out of the contract. The best practice of check-effects-interaction requires that one perform all state updates (effects) *before* any external calls (interactions), so that the state is not vulnerable to re-entrancy attacks.

While no funds are at risk, since the new variable is never used by the contract in any calculations, if Balancer LP tokens ever introduce transfer hooks, it would be possible for an attacker to get the contract to give the wrong answer about its balance, which may affect calculations that other contracts make.

Proof of Concept

Each place that modifies the new `supply` variable, does so after a call that may be re-enterable in the future

`createLock()`:

```
451        token.safeTransferFrom(msg.sender, address(this), _va
452        // Total supply of token deposited
453        supply = supply + _value;
```

https://github.com/fiatdao/veFDT/blob/51f0001fd0576049341cfc8b9146cde9b9378797/contracts/VotingEscrow.sol#L451-L453

`increaseAmount()`:

```
517        token.safeTransferFrom(msg.sender, address(this), _va
518        // Total supply of token deposited
519        supply = supply + _value;
```

https://github.com/fiatdao/veFDT/blob/51f0001fd0576049341cfc8b9146cde9b9378797/contracts/VotingEscrow.sol#L517-L519

`withdraw()`:

```
583        token.safeTransfer(msg.sender, value);
584        // Total supply of token deposited
585        supply = supply - value;
```

https://github.com/fiatdao/veFDT/blob/51f0001fd0576049341cfc8b9146cde9b9378797/contracts/VotingEscrow.sol#L583-L585

`quitLock()` (assumes penalty is deducted immediately):

```
715         token.safeTransfer(msg.sender, remainingAmount);
716         // Total supply of token deposited
717         supply = supply - value;
```

https://github.com/fiatdao/veFDT/blob/51f0001fd0576049341cfc8b9146cde9b9378797/contracts/VotingEscrow.sol#L715-L717

Recommended Mitigation Steps

Move the update of the `supply` state variable so that it occurs before each external call

Remediation

The sponsor properly addressed the issue with this commit.

[M.N-01] Mitigation of [M-03/Issue 4] uses non-standard return behaviors

The fix in PR 14 for M-03 incorrectly returns the result of a non-return-valued function in another non-return-valued function. While the code compiles and works, it’s confusing to see a function returning the result of a function that has no return values.

Proof of Concept

```
554      // See IVotingEscrow for documentation
555      function delegate(address _addr)
556          external
557          override
558          nonReentrant
559          checkBlocklist
560      {
561          // Different restrictions apply to undelegation
562          if (_addr == msg.sender) {
563              return _undelegate();
564          }
```

https://github.com/fiatdao/veFDT/blob/b9afd265fac9b3b3a3dc1440d47f421a41ff9639/contracts/VotingEscrow.sol#L554-L564

Recommended Mitigation Steps

Move the return to after the function call:

```
            // Different restrictions apply to undelegation
            if (_addr == msg.sender) {
-                return _undelegate();
+                _undelegate();
+                return;
            }
```

Remediation

The sponsor properly addressed the issue with PR 20.

Final changes

All of the contest-related issue commits were properly combined into PR 21, along with some minor, correct, comment changes, and appear as https://github.com/fiatdao/veFDT/commit/4e80f80786bd8143c2e5b59ac2f66f99bb589094. The PRs for the non-contest issues and their mitigations were combined into PR 22, and the final repository state is https://github.com/fiatdao/veFDT/tree/3f822125e05e2927eab0a3a3e797508b363083ab.

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