SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Bond
Prepared by:	Sherlock
Lead Security Expert:	
Dates Audited:	July 3 - July 8, 2023
Prepared on:	August 8, 2023

Introduction

Acquire assets, own liquidity, and diversify treasuries on the permissionless Bond Marketplace. Bonds builds stronger protocols with more resilient treasuries

Scope

Repository: Bond-Protocol/options

Branch: master

Commit: 817c62b24a8447870bdf4a618a3bc240f4cc86e3

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
10	2

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

stuxy	pep7siup	Vagner
lil.eth	0xhunter526	dopeflamingo
techOptimizor	Yuki	circlelooper
Juntao	0xcrunch	pks_

TrungOre	kutugu	ctf_sec
Musaka	bin2chen	tsvetanovv
berndartmueller	Jiamin	Auditwolf
Delvir0	Sm4rty	Kow
BenRai	caventa	tnquanghuy0512
tvdung94	qandisa	

Issue H-1: All fund from Teller contract can be drained because a malicious receiver can call reclaim repeatedly

Source: https://github.com/sherlock-audit/2023-06-bond-judging/issues/79

Found by

0xcrunch, Jiamin, Juntao, Kow, Musaka, Sm4rty, Yuki, berndartmueller, bin2chen, circlelooper, ctf_sec, kutugu, stuxy, tnquanghuy0512, tvdung94

Summary

All fund from Teller contract can be drained because a malicious receiver can call reclaim repeatedly

Vulnerability Detail

When mint an option token, the user is required to transfer the payout token for a call option or quote token for a put option

if after the expiration, the receiver can call reclaim to claim the payout token if the option type is call or claim the quote token if the option type is put

however, the root cause is when reclaim the token, the corresponding option is not burnt (code)

```
// Revert if caller is not receiver
 if (msg.sender != receiver) revert Teller_NotAuthorized();

 // Transfer remaining collateral to receiver
 uint256 amount = optionToken.totalSupply();
 if (call) {
     payoutToken.safeTransfer(receiver, amount);
 } else {
     // Calculate amount of quote tokens equivalent to amount at strike price
     uint256 quoteAmount = amount.mulDiv(strikePrice, 10 **
↪   payoutToken.decimals());
     quoteToken.safeTransfer(receiver, quoteAmount);
 }
```

the Teller contract is likely to hold fund from multiple option token

a malicious actor can create call Teller#deploy and set a receiver address that can control by himself

and then wait for the option expiry and repeated call reclaim to steal the fund from the Teller contract

Impact

All fund from Teller contract can be drained because a malicious receiver can call reclaim repeatedly

Code Snippet

https://github.com/sherlock-audit/2023-06-bond/blob/fce1809f83728561dc75078d41ead6d60e15d065/options/src/fixed-strike/FixedStrikeOptionTeller.sol#L395

Tool used

Manual Review

Recommendation

Burn the corresponding option burn when reclaim the fund

Discussion

Oighty

Agree with this issue. Option tokens can't be burned with reclaim because they could be held by a large number of accounts. A simpler solution would be to store whether a specific option token has been reclaimed (in a mapping) and not allow that token to be reclaimed again.

ctf-sec

> A simpler solution would be to store whether a specific option token has been reclaimed (in a mapping) and not allow that token to be reclaimed again.

Agree with the fix

Oighty

Fix implemented at https://github.com/Bond-Protocol/options/pull/2

ctf-sec

Fix looks good, the fix use a map and revert if the receiver already claimed!

Issue H-2: All funds can be stolen from FixedStrikeOptionTeller using a token with malicious decimals

Source: https://github.com/sherlock-audit/2023-06-bond-judging/issues/90

Found by

Juntao, berndartmueller, bin2chen, ctf_sec

Summary

`FixedStrikeOptionTeller` is a single contract which deploys multiple option tokens. Hence this single contract holds significant payout/quote tokens as collateral. Also the `deploy`, `create` & `exercise` functions of this contract can be called by anyone.

This mechanism can be exploited to drain `FixedStrikeOptionTeller` of all tokens.

Vulnerability Detail

This is how the create functions looks like:

```
function create(
    FixedStrikeOptionToken optionToken_,
    uint256 amount_
) external override nonReentrant {
    ...
    if (call) {
        ...
    } else {
        uint256 quoteAmount = amount_.mulDiv(strikePrice, 10 **
↪   payoutToken.decimals());
        ...
        quoteToken.safeTransferFrom(msg.sender, address(this), quoteAmount);
        ...
    }

    optionToken.mint(msg.sender, amount_);
}
```

exercise function:

```
function exercise(
    FixedStrikeOptionToken optionToken_,
    uint256 amount_
) external override nonReentrant {
    ...
    uint256 quoteAmount = amount_.mulDiv(strikePrice, 10 **
↪   payoutToken.decimals());

    if (msg.sender != receiver) {
        ...
    }

    optionToken.burn(msg.sender, amount_);

    if (call) {
        ...
    } else {
        quoteToken.safeTransfer(msg.sender, quoteAmount);
    }
}
```

Consider this attack scenario:

Let's suppose the `FixedStrikeOptionTeller` holds some DAI tokens.

- An attacker can create a malicious payout token of which he can control the `decimals`.
- The attacker calls `deploy` to create an option token with malicious payout token and DAI as quote token and `put` option type
- Make `payoutToken.decimals` return a large number and call `FixedStrikeOptionTeller.create` with input X. Here quoteAmount will be calculated as 0.

```
// Calculate amount of quote tokens required to mint
uint256 quoteAmount = amount_.mulDiv(strikePrice, 10 ** payoutToken.decimals());

// Transfer quote tokens from user
// Check that amount received is not less than amount expected
// Handles edge cases like fee-on-transfer tokens (which are not supported)
uint256 startBalance = quoteToken.balanceOf(address(this));
quoteToken.safeTransferFrom(msg.sender, address(this), quoteAmount);
```

So 0 DAI will be pulled from the attacker's account but he will receive X option token.

- Make `payoutToken.decimals` return a small value and call `FixedStrikeOptionTeller.exercise` with X input. Here `quoteAmount` will be calculated as a very high number (which represents number of DAI tokens). So he will receive huge amount of DAI against his X option tokens when exercise the option or when reclaim the token

```
// Transfer remaining collateral to receiver
```

```
uint256 amount = optionToken.totalSupply();
if (call) {
    payoutToken.safeTransfer(receiver, amount);
} else {
    // Calculate amount of quote tokens equivalent to amount at strike price
    uint256 quoteAmount = amount.mulDiv(strikePrice, 10 **
 ↪  payoutToken.decimals());
    quoteToken.safeTransfer(receiver, quoteAmount);
}
```

Hence, the attacker was able to drain all DAI tokens from the `FixedStrikeOptionTeller` contract. The same mechanism can be repeated to drain all other ERC20 tokens from the `FixedStrikeOptionTeller` contract by changing the return value of the decimal external call

Impact

Anyone can drain `FixedStrikeOptionTeller` contract of all ERC20 tokens. The cost of attack is negligible (only gas cost).

High impact, high likelyhood.

Code Snippet

https://github.com/sherlock-audit/2023-06-bond/blob/main/options/src/fixed-strike/FixedStrikeOptionTeller.sol#L283

https://github.com/sherlock-audit/2023-06-bond/blob/main/options/src/fixed-strike/FixedStrikeOptionTeller.sol#L339

Tool used

Manual Review

Recommendation

Consider storing the `payoutToken.decimals` value locally instead of fetching it real-time on all `exercise` or `reclaim` calls.

or support payout token and quote token whitelist, if the payout token and quote token are permissionless created, there will always be high risk

Discussion

ctf-sec

https://github.com/sherlock-audit/2023-06-bond-judging/issues/8 is the duplicate of this issue

Oighty

Agree with this issue. The simplest solution seems to be stored the decimal values used when the option token is deployed.

Oighty

Fix implemented at https://github.com/Bond-Protocol/options/pull/3

ctf-sec

This is a big one and a important one, will look into the fix

ctf-sec

The fix looks good, the decimals call is properly cached, I highly recommend adding a test to make sure it is working as intended

Oighty

I added some tests to confirm the cached decimal behavior on create and exercise to the PR.

ctf-sec

Thanks for adding the test cases! All good!

Issue M-1: Funds can be stolen from the `FixedStrikeOptionTeller` contract by creating put option tokens without providing collateral

Source: https://github.com/sherlock-audit/2023-06-bond-judging/issues/61

Found by

berndartmueller, pks_, techOptimizor

Summary

Due to a rounding error when calculating the `quoteAmount` in the `create` function of the `FixedStrikeOptionTeller` contract, it is possible to create (issue) option tokens without providing the necessary collateral. A malicious receiver can exploit this to steal funds from the `FixedStrikeOptionTeller` contract.

Vulnerability Detail

Anyone can create (issue) put option tokens with the `create` function in the `FixedStrikeOptionTeller` contract. However, by specifying a very small `amount_`, the `quoteAmount` calculation in line 283 can potentially round down to zero. This is the case if the result of the multiplication in line 283, $amount * strikePrice$ is smaller than $10^{decimals}$, where `decimals` is the number of decimals of the payout token.

For example, assume the following scenario:

() Parameter	Description
()	
Quote token	$USDC. 6 decimals
Payout token	$GMX. 18 decimals
$payoutToken_{decimals}$	18 decimals
$amount$	`1e10`. Amount (`amount_`) supplied to the `create` function, in payout token deci
$strikePrice$	`50e6` ~ 50 USD. The strike price of the option token, in quote tokens.
()	

Please note that the option token has the same amount of decimals as the payout t oken.

Calculating `quoteAmount` leads to the following result:

$$\begin{aligned} quoteAmount &= \frac{amount * strikePrice}{10^{payoutToken_{decimals}}} \\ &= \frac{1e10 * 50e6}{10^{18}} \\ &= \frac{5e17}{10^{18}} \\ &= \frac{5 * 10^{17}}{10^{18}} \\ &= 0 \end{aligned}$$

As observed, the result is rounded down to zero due to the numerator being smaller than the denominator.

This results in **0** quote tokens to be transferred from the caller to the contract, and in return, the caller receives `1e10` ($amount$) option tokens. This process can be repeated to mint an arbitrary amount of option tokens for free.

Impact

Put options can be minted for free without providing the required `quoteToken` collateral.

This is intensified by the fact that a malicious receiver, which anyone can be, can exploit this issue by deploying a new option token (optionally with a very short expiry), repeatedly minting free put options to accumulate option tokens, and then, once the option expires, call `reclaim` to receive quote token collateral.

This collateral, however, was supplied by other users who issued (created) option tokens with the same quote token. Thus, the malicious receiver can drain funds from other users and cause undercollateralization of those affected option tokens.

Code Snippet

src/fixed-strike/FixedStrikeOptionTeller.sol#L283

```
236: function create(
237:     FixedStrikeOptionToken optionToken_,
238:     uint256 amount_
239: ) external override nonReentrant {
...      // [...]
268:
269:     // Transfer in collateral
270:     // If call option, transfer in payout tokens equivalent to the amount
 ↪  of option tokens being issued
271:     // If put option, transfer in quote tokens equivalent to the amount of
 ↪  option tokens being issued * strike price
272:     if (call) {
```

```
...          // [...]
281:     } else {
282:         // Calculate amount of quote tokens required to mint
283: @>      uint256 quoteAmount = amount_.mulDiv(strikePrice, 10 **
↪  payoutToken.decimals()); // @audit-issue Rounds down
284:
285:         // Transfer quote tokens from user
286:         // Check that amount received is not less than amount expected
287:         // Handles edge cases like fee-on-transfer tokens (which are not
↪  supported)
288:         uint256 startBalance = quoteToken.balanceOf(address(this));
289:         quoteToken.safeTransferFrom(msg.sender, address(this), quoteAmount);
290:         uint256 endBalance = quoteToken.balanceOf(address(this));
291:         if (endBalance < startBalance + quoteAmount)
292:             revert Teller_UnsupportedToken(address(quoteToken));
293:     }
294:
295:     // Mint new option tokens to sender
296:     optionToken.mint(msg.sender, amount_);
297: }
```

Tool used

Manual Review

Recommendation

Consider adding a check after line 283 to ensure `quoteAmount` is not zero.

Discussion

Oighty

Agree with this issue. Checking for a zero quote amount is a potential solution, but it would be better to fix the precision issue. I'm going to explore other ways to do this (including something like the price scale system used in the other bond protocol contracts).

ctf-sec

I agree, if only checking the amount > 0, the precision loss can still be large and round down to 1 wei

scaling amount with decimals seems the way to go!

ctf-sec

In report #62

the auditor is correct the user can game the code to not pay the quote amount and receive the payout token just like the receiver

but there is no loss of fund, because when minting option, the payout token collateral has to be supplied.

In this report

it is true the user can game the code to not supply the collateral and get the option token and then call reclaim the drain the fund.

the impact is clearly more severe

but the reclaim issue has been extensively covered in report:

https://github.com/sherlock-audit/2023-06-bond-judging/issues/79

the report takes the advantage of the rounding

in fact in my report:

https://github.com/sherlock-audit/2023-06-bond-judging/issues/90

I highlight the decimals is large and collateral is round down to 0 then call reclaim to drain the fund as well

After internal discussion, Agree with the de-duplication and leave as a separate medium

0x3b33

I mean the issue is valid in a way that there needs to be a check for quote of 0, but how is anyone gonna profit from it? The attacker's profit is only 1e10, compared to GMX's decimals 18, he is making nothing! If my calculations are not wrong, the attacker will make 0.00000055$ per call... When I press the button to turn my PC on I spend more money, let alone to run a TX on any network.

ctf-sec

> I mean the issue is valid in a way that there needs to be a check for quote of 0, but how is anyone gonna profit from it? The attacker's profit is only 1e10, compared to GMX's decimals 18, he is making nothing! If my calculations are not wrong, the attacker will make 0.00000055$ per call... When I press the button to turn my PC on I spend more money, let alone to run a TX on any network.

To profits, the attacker would need to repeated call reclaim (which is another issue)

The decimal rounding plays a role as well (which is another issue)

Even the fact that can mint token without providing collateral make it a valid issue.

Oighty

Fix implemented at https://github.com/Bond-Protocol/options/pull/11

ctf-sec

Fix looks good

Issue M-2: `optionTokens` can be expired even though the epoch is not over

Source: https://github.com/sherlock-audit/2023-06-bond-judging/issues/63

Found by

BenRai, qandisa

Summary

When deploying an `optionToken` the parameter `expiry` is rounded down to the "nearest day at 0000 UTC" but since the end of an epoch is calculated by the `epochDuration` and the exact time the epoch has stared and the `optionToken` was created this can lead to an epoch still being active but the corresponding `optionToken` to be already expired.

Vulnerability Detail

When starting a new epoch, the variable `epochStart` is set to the current time (`block.timestamp`) and the end of the epoch is calculated by adding the `epochDuration` to the `epochStart` variable.

The `optionToken` of the new epoch is deployed with the parameter `expire` calculated based on the current time stamp, the `timeUntilEligible` and the `eligibleDuration`. (`uint48(block.timestamp) + timeUntilEligible + eligibleDuration`). The final expiration date of the optionToken is rounded down to the "nearest day at 0000 UTC" before the token is deployed.

Since the `epochDuration` can be as close as 1 second to the sum of `timeUntilEligible + eligibleDuration` this can lead to an epoch still being active but its `optionToken` to be already expired.

Example:

epochDuration = 7 days timeUntilEligible = 0 eligibleDuration = 7 days + 12 hours

New epoch is launched on the 01.01.2024 at 11:45 am.

=> epochStart = block.timestamp = 01.01.2024 at 11:45 am epochEnd = epochStart + epochDuration = 08.01.2024 at 11:45 am

`initial expire` = block.timestamp + timeUntilEligible + eligibleDuration = 08.01.2024 at 11:45 pm

`final expire` after rounding down = uint48(`initial expire`/ 1day) * 1 day = 08.01.2024 at 00:00 am

The epoch is still active between `final expire` and `epochEnd` even though the option has already expired.

Impact

Users that wait until the epoch has ended to claim their rewards expecting the options to be exercisable for 12 hours after the epoch end cannot claim their options since they are expired already and lose out on all the value the options would have had which can be significant depending on the current price of the `payoutToken`

Code Snippet

https://github.com/sherlock-audit/2023-06-bond/blob/fce1809f83728561dc75078d41ead6d60e15d065/options/src/fixed-strike/liquidity-mining/OTLM.sol#L514-L534

https://github.com/sherlock-audit/2023-06-bond/blob/fce1809f83728561dc75078d41ead6d60e15d065/options/src/fixed-strike/FixedStrikeOptionTeller.sol#L122

https://github.com/sherlock-audit/2023-06-bond/blob/fce1809f83728561dc75078d41ead6d60e15d065/options/src/fixed-strike/liquidity-mining/OTLM.sol#L605-L611

https://github.com/sherlock-audit/2023-06-bond/blob/fce1809f83728561dc75078d41ead6d60e15d065/options/src/fixed-strike/liquidity-mining/OTLM.sol#L629-L643

Tool used

Manual Review

Recommendation

The expiration of the `optionTokens` should be rounded up instead of down. This would increase the time an option can be redeemed long enough to prevent the scenario described above.

Discussion

Oighty

Agree with this issue, but would prefer to fix by requiring the expiry to be atleast 1 day longer than the epoch duration. Having expiries all round-down is simpler and consistent with some of our other contracts.

ctf-sec

Based on impact, agree with the finding

> Users that wait until the epoch has ended to claim their rewards expecting the options to be exercisable for 12 hours after the epoch end cannot claim their options since they are expired already and lose out on all the value the options would have had which can be significant depending on the current price of the payoutToken

Oighty

Fix implemented at https://github.com/Bond-Protocol/options/pull/10

ctf-sec

Looks likes ok, may also want to round up as well O(_)O

> The expiration of the optionTokens should be rounded up instead of down. This would increase the time an option can be redeemed long enough to prevent the scenario described above.

Oighty

See comment above. Our other contracts use expiry values that are rounded down. We want to maintain consistency.

ctf-sec

Ok Then the fix looks good!

Issue M-3: Blocklisted address can be used to lock the option token minter's fund

Source: https://github.com/sherlock-audit/2023-06-bond-judging/issues/81

Found by

Vagner, berndartmueller, bin2chen, caventa, ctf_sec

Summary

Blocklisted address can be used to lock the option token minter's fund

Vulnerability Detail

When deploy a token via the teller contract, the contract validate that receiver address is not address(0)

However, a malicious option token creator can save a seemingly favorable strike price and pick a blocklisted address and set the blocklisted address as receiver

https://github.com/d-xo/weird-erc20#tokens-with-blocklists

> Some tokens (e.g. USDC, USDT) have a contract level admin controlled address blocklist. If an address is blocked, then transfers to and from that address are forbidden.
>
> Malicious or compromised token owners can trap funds in a contract by adding the contract address to the blocklist. This could potentially be the result of regulatory action against the contract itself, against a single user of the contract (e.g. a Uniswap LP), or could also be a part of an extortion attempt against users of the blocked contract.

then user would see the favorable strike price and mint the option token using payout token for call option or use quote token for put option

However, they can never exercise their option because the transaction would revert when transferring asset to the recevier for call option and transferring asset to the receiver for put option when exercise the option.

the usre's fund that used to mint the option are locked

Impact

Blocklisted receiver address can be used to lock the option token minter's fund

Code Snippet

https://github.com/sherlock-audit/2023-06-bond/blob/fce1809f83728561dc75078d41ead6d60e15d065/options/src/fixed-strike/FixedStrikeOptionTeller.sol#L139

https://github.com/sherlock-audit/2023-06-bond/blob/fce1809f83728561dc75078d41ead6d60e15d065/options/src/fixed-strike/FixedStrikeOptionTeller.sol#L361

(https://github.com/sherlock-audit/2023-06-bond/blob/fce1809f83728561dc75078d41ead6d60e15d065/options/src/fixed-strike/FixedStrikeOptionTeller.sol#L378

Tool used

Manual Review

Recommendation

Valid that receiver is not blacklisted when create and deploy the option token or add an expiry check, if after the expiry the receiver does not reclaim the fund, allows the option minter to burn their token in exchange for their fund

Discussion

Oighty

Agree with this issue. Checking blocklists seems challenging to handle since they could be implemented differently. The fix mentioned in #29 , #52 , and #70 of using a pull mechanism to claim proceeds could work, but isn't a great UX from the receiver's perspective. That may be the best option though.

juntzhan

Escalate

Disagree with severity, this is not a valid high:

1. Users should be wary of tokens they interact with;
2. Users are able to see if receiver is blocklisted before interacting with;
3. It's not up to option token creator to get receiver blocklisted, even if receiver did bad things, for example, only 174 address are blacklisted by USDC till now.

sherlock-admin

> Escalate
>
> Disagree with severity, this is not a valid high:
>
> 1. Users should be wary of tokens they interact with;

> 2. Users are able to see if receiver is blocklisted before interacting with;
> 3. It's not up to option token creator to get receiver blocklisted, even if receiver did bad things, for example, only 174 address are blacklisted by USDC till now.

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.

sherlock-admin

> Escalate
>
> This should be a valid high issue
>
> Agree that it is possible for users are able to see if receiver is blocklisted / blacklisted before interacting with.
>
> But an address can be not blacklisted in the first interaction (Deploy or create), and then blacklisted in the second onwards interaction (Exercise or reclaim)
>
> which leads to transfer failure and leads to fund stuck in the contract
>
> Yes, it is not up to FixedStrikeOptionTeller to get receiver blocklisted / blacklisted. But what we need to do is how to handle if the receiver get blocklisted / blacklisted and to prevent fund stuck in the contract

```
You've deleted an escalation for this issue.
```

ctf-sec

> Users are able to see if receiver is blocklisted before interacting with;

This would be difficult for user give the current audit scope. Option creating process has nothing to do with receiver address.

plus blocklisted receiver can lock user's fund infinitely long time.

jingyi2811

Agree that it is possible for users are able to see if receiver is blocklisted / blacklisted before interacting with.

But an address can be not blacklisted in the first interaction (Deploy or create), and then blacklisted in the second onwards interaction (Exercise or reclaim)

which leads to transfer failure and leads to fund stuck in the contract

Yes, it is not up to FixedStrikeOptionTeller to get receiver blocklisted / blacklisted. But what we need to do is how to handle if the receiver get blocklisted / blacklisted and to prevent fund stuck in the contract

Oot2k

I think this is on the verge of medium and high. The worst scenario is, as mentioned above, that the receiver gets backlisted between option start and expiry. I think this is is fairly unlikely, and have not tested how easy it is to get a address blacklisted, but if this is still the case the damage is really high compared to the attack cost. I still think this is a valid high. (small attack cost, large damage)

juntzhan

There is no large damage, user's funds won't be locked because the the transaction will revert, the only problem is that user won't be able to exercise option token, but there is only 50% chance that user will try to exercise.

So I think it's fair to think this is not a valid high:

1. Low possibility that receiver gets backlisted between option start and expiry
2. Small damage to user if option token won't be exercised.

Oighty

We don't plan to implement a fix for this. We don't believe the increased security is worth the UX degradation. For OTLM, the user is not paying for the option token, therefore, the loss is minimal (other than opportunity cost). A bad receiver can be swapped out by the OTLM owner, or users will withdraw when they figure out that the receiver is blacklisted. If tokens are sold, a check can be made on the front-end if the receiver is blacklisted or not for specific tokens. Overall, this is an edge case as stated above. cc @0xTex

ctf-sec

> We don't plan to implement a fix for this. We don't believe the increased security is worth the UX degradation. For OTLM, the user is not paying for the option token, therefore, the loss is minimal (other than opportunity cost). A bad receiver can be swapped out by the OTLM owner, or users will withdraw when they figure out that the receiver is blacklisted. If tokens are sold, a check can be made on the front-end if the receiver is blacklisted or not for specific tokens. Overall, this is an edge case as stated above. cc @0xTex

That make sense!

ctf-sec

Recommend checking #70 before resolving the escalation :)

hrishibhat

Result: Medium Has duplicates Considering this a valid medium, based on the above discussion and given the impact. This does not break core functionality, not causing irreversible damage for all users. Agree with the Sponsor comment below. Additional Sponsor comment:

> If options are sold or reward to uses for some risk-taking activity and then the receiver prevents execution, then there would be some loss of value for the user in the aggregate. I personally don't think this is a high issue because it is possible to verify if this is the case before interacting with the token.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- juntzhan: accepted

Issue M-4: Loss of option token from Teller and reward from OTLM if L2 sequencer goes down

Source: https://github.com/sherlock-audit/2023-06-bond-judging/issues/82

Found by

ctf_sec, qandisa

Summary

Loss of option token from Teller and reward from OTLM if L2 sequencer goes down

Vulnerability Detail

In the current implementation, if the option token expires, the user is not able to exerise the option at strike price

```
// Validate that option token is not expired
    if (uint48(block.timestamp) >= expiry) revert Teller_OptionExpired(expiry);
```

if the option token expires, the user lose rewards from OTLM as well when claim the reward

```
function _claimRewards() internal returns (uint256) {
    // Claims all outstanding rewards for the user across epochs
    // If there are unclaimed rewards from epochs where the option token has
↪   expired, the rewards are lost

    // Get the last epoch claimed by the user
    uint48 userLastEpoch = lastEpochClaimed[msg.sender];

```

and

```
// If the option token has expired, then the rewards are zero
    if (uint256(optionToken.expiry()) < block.timestamp) return 0;
```

And in the onchain context, the protocol intends to deploy the contract in arbitrum and optimsim

```
Q: On what chains are the smart contracts going to be deployed?
Mainnet, Arbitrum, Optimism
```

However, If Arbitrum and optimism layer 2 network, the sequencer is in charge of process the transaction

For example, the recent optimism bedrock upgrade cause the sequencer not able to process transaction for a hew hours

https://cryptopotato.com/optimism-bedrock-upgrade-release-date-revealed/

> Bedrock Upgrade According to the official announcement, the upgrade will require 2-4 hours of downtime for OP Mainnet, during which there will be downtime at the chain and infrastructure level while the old sequencer is spun down and the Bedrock sequencer starts up.
>
> Transactions, deposits, and withdrawals will also remain unavailable for the duration, and the chain will not be progressing. While the read access across most OP Mainnet nodes will stay online, users may encounter a slight decrease in performance during the migration process.

In Arbitrum

https://thedefiant.io/arbitrum-outage-2

> Arbitrum Goes Down Citing Sequencer Problems Layer 2 Arbitrum suffers 10 hour outage.

and

https://beincrypto.com/arbitrum-sequencer-bug-causes-temporary-transaction-pause/

> Ethereum layer-2 (L2) scaling solution Arbitrum stopped processing transactions on June 7 because its sequencer faced a bug in the batch poster. The incident only lasted for an hour.

If the option expires during the sequencer down time, the user basically have worthless option token because they cannot exercise the option at strike price

the user would lose his reward as option token from OTLM.sol, which defeats the purpose of use OTLM to incentive user to provide liquidity

Impact

Loss of option token from Teller and reward from OTLM if L2 sequencer goes down

Code Snippet

https://github.com/sherlock-audit/2023-06-bond/blob/fce1809f83728561dc75078d41ead6d60e15d065/options/src/fixed-strike/FixedStrikeOptionTeller.sol#L336

https://github.com/sherlock-audit/2023-06-bond/blob/fce1809f83728561dc75078d41ead6d60e15d065/options/src/fixed-strike/liquidity-mining/OTLM.sol#L496

Tool used

Manual Review

Recommendation

chainlink has a sequencer up feed

https://docs.chain.link/data-feeds/l2-sequencer-feeds

consider integrate the up time feed and give user extra time to exercise token and claim option token reward if the sequencer goes down

Discussion

Oighty

Acknowledge this issue. Generally, we expect option token durations to be over a week+ duration so users will have a lot of time to exercise. Observed sequencer outages have been measured in hours. Therefore, we view the overall risk to the user as low. However, we will keep this in mind and explore how much complexity it would add to account for this on L2s.

ctf-sec

Ok! Thanks!

#24 is not a duplicate of this issue, the sequencer check for price should be done in Bond Oracle and it is out of scope in this audit.

ctf-sec

Past similar finding:
https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/422

ctf-sec

The report has "won't fix" tag, assume the sponsor acknowledge the report.

Oighty

Yep, acknowledge the issue, but we don't plan on implementing a fix for this.

Issue M-5: Use A's staked token balance can be used to mint option token as reward for User B if the payout token equals to the stake token

Source: https://github.com/sherlock-audit/2023-06-bond-judging/issues/83

Found by

ctf_sec

Summary

User's staked token balance can be used to mint option token as reward if the payout token equals to the stake token, can cause user to loss fund

Vulnerability Detail

In OTLM, user can stake stakeToken in exchange for the option token minted from the payment token

when staking, we transfer the stakedToken in the OTLM token

```
// Increase the user's stake balance and the total balance
stakeBalance[msg.sender] = userBalance + amount_;
totalBalance += amount_;

// Transfer the staked tokens from the user to this contract
stakedToken.safeTransferFrom(msg.sender, address(this), amount_);
```

before the stake or unstake or when we are calling claimReward

we are calling _claimRewards -> _claimEpochRewards -> we use payout token to mint and create option token as reward

```
payoutToken.approve(address(optionTeller), rewards);
optionTeller.create(optionToken, rewards);

// Transfer rewards to sender
ERC20(address(optionToken)).safeTransfer(msg.sender, rewards);
```

the problem is, if the stake token and the payout token are the same token, the protocol does not distingush the balance of the stake token and the balance of payout token

suppose both stake token and payout token are USDC

suppose user A stake 100 USDC

suppose user B stake 100 USDC

time passed, user B accure 10 token unit reward

now user B can claimRewards,

the protocol user 10 USDC to mint option token for B

the OTLM has 190 USDC

if user A and user B both call emergencyUnstakeAll, whoeve call this function later will suffer a revert and he is not able to even give up the reward and claim their staked balance back

because a part of the his staked token balance is treated as the payout token to mint option token reward for other user

Impact

If there are insufficient payout token in the OTLM, the expected behavior is that the transaction revert when claim the reward and when the code use payout token to mint option token

and in the worst case, user can call emergencyUnstakeAll to get their original staked balane back and give up their reward

however, if the staked token is the same as the payout token,

a part of the user staked token can be mistakenly and constantly mint as option token reward for his own or for other user and eventually when user call emergencyUnstakeAll, there will be insufficient token balance and transaction revert

so user will not able to get their staked token back

Code Snippet

https://github.com/sherlock-audit/2023-06-bond/blob/fce1809f83728561dc75078d41ead6d60e15d065/options/src/fixed-strike/liquidity-mining/OTLM.sol#L334

https://github.com/sherlock-audit/2023-06-bond/blob/fce1809f83728561dc75078d41ead6d60e15d065/options/src/fixed-strike/liquidity-mining/OTLM.sol#L508

Tool used

Manual Review

Recommendation

Seperate the accounting of the staked user and the payout token or check that staked token is not payout token when creating the OTLM.sol

Discussion

Oighty

Agree with this issue. We'll explore both the recommendations.

ctf-sec

In the beginning

https://github.com/sherlock-audit/2023-06-bond-judging/issues/85

the issue #85 is considered as a separate medium

root cause is the code does not really distinguish the payout token and the staking token balance

leads to two issue

1. user's own staked balance is minted as reward (this even lead to lose of fund but only in the case of staking token equals to payment token, which I see a very likely supported use case, the core invariant that user should always call emergencyUnstakeAll is broken)
2. owner that should not remove the user fund can remove the user's fund as payment token

but for issue #85

it requires admin owner mistake although the code is not designed to owner remove staking token

but sherlock rules still apply:

https://docs.sherlock.xyz/audits/judging/judging#duplication-rules

```
 Issues identifying a core vulnerability can be considered duplicates.
 Scenario 1:
There is a root cause/error/vulnerability A in the code. This vulnerability A ->
↪  leads to two attacks paths:
  -> high severity path
-> medium severity attack path/just identifying the vulnerability.
```

so after internal discussion, issue #85 close and not rewarded just to be fair.

berndartmueller

Escalate

Disagree with the validity of the issue. This is not a valid medium finding

Due to relying on the contract owner not providing sufficient payout tokens used as rewards, this is considered an admin error and thus not eligible for medium severity.

It is assumed that the trusted contract owner of the `OTLM` contract supplies a sufficient amount of payout tokens, even supplying more than anticipated (based on the reward rate). "Unused" payout tokens topped up by the owner can always be withdrawn via the `withdrawPayoutTokens` function.

In the case that insufficient payout tokens have been supplied by the contract owner, thus utilizing the deposited payout tokens from the users, the owner can always add additional payout token capital to the contract, ensuring that calls to `emergencyUnstakeAll` succeed.

sherlock-admin

> Escalate
>
> Disagree with the validity of the issue. This is not a valid medium finding
>
> Due to relying on the contract owner not providing sufficient payout tokens used as rewards, this is considered an admin error and thus not eligible for medium severity.
>
> It is assumed that the trusted contract owner of the `OTLM` contract supplies a sufficient amount of payout tokens, even supplying more than anticipated (based on the reward rate). "Unused" payout tokens topped up by the owner can always be withdrawn via the `withdrawPayoutTokens` function.
>
> In the case that insufficient payout tokens have been supplied by the contract owner, thus utilizing the deposited payout tokens from the users, the owner can always add additional payout token capital to the contract, ensuring that calls to `emergencyUnstakeAll` succeed.

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.

Oot2k

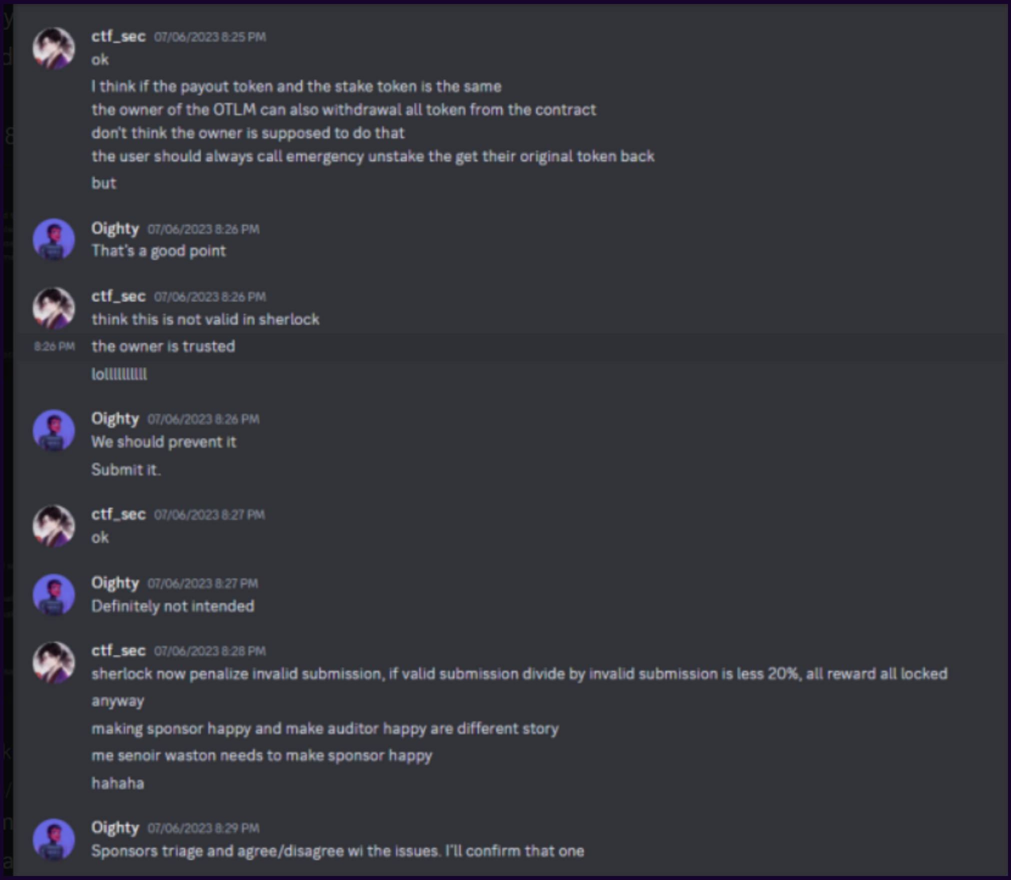
I think the escalation is valid. More of admin input.

JeffCX

I already invalidate my finding #85, which is more fair to the reward pot

Invalid this finding is not fair to me :) here is the argument

Sponsor DM is only for reference



ctf_sec 07/06/2023 8:25 PM
ok
I think if the payout token and the stake token is the same
the owner of the OTLM can also withdrawal all token from the contract
don't think the owner is supposed to do that
the user should always call emergency unstake the get their original token back
but

Oighty 07/06/2023 8:26 PM
That's a good point

ctf_sec 07/06/2023 8:26 PM
think this is not valid in sherlock
8:26 PM the owner is trusted
lolllllllll

Oighty 07/06/2023 8:26 PM
We should prevent it
Submit it.

ctf_sec 07/06/2023 8:27 PM
ok

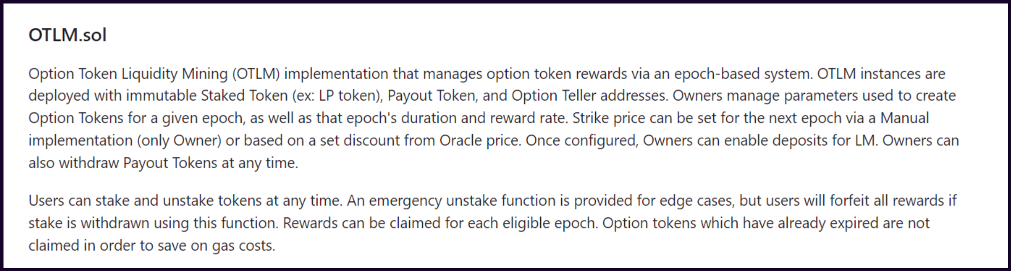
Oighty 07/06/2023 8:27 PM
Definitely not intended

ctf_sec 07/06/2023 8:28 PM
sherlock now penalize invalid submission, if valid submission divide by invalid submission is less 20%, all reward all locked
anyway
making sponsor happy and make auditor happy are different story
me senoir waston needs to make sponsor happy
hahaha

Oighty 07/06/2023 8:29 PM
Sponsors triage and agree/disagree wi the issues. I'll confirm that one

We need to refer to the doc and read me

https://github.com/sherlock-audit/2023-06-bond/tree/fce1809f83728561dc75078d41ead6d60e15d065/options#otlmsol



OTLM.sol

Option Token Liquidity Mining (OTLM) implementation that manages option token rewards via an epoch-based system. OTLM instances are deployed with immutable Staked Token (ex: LP token), Payout Token, and Option Teller addresses. Owners manage parameters used to create Option Tokens for a given epoch, as well as that epoch's duration and reward rate. Strike price can be set for the next epoch via a Manual implementation (only Owner) or based on a set discount from Oracle price. Once configured, Owners can enable deposits for LM. Owners can also withdraw Payout Tokens at any time.

Users can stake and unstake tokens at any time. An emergency unstake function is provided for edge cases, but users will forfeit all rewards if stake is withdrawn using this function. Rewards can be claimed for each eligible epoch. Option tokens which have already expired are not claimed in order to save on gas costs.

In the doc, I don't see the doc saying the the owner has to provide the payout token so not providing payout token is not an admin error.

plus

> It is assumed that the trusted contract owner of the OTLM contract supplies a sufficient amount of payout tokens, even supplying more than anticipated (based on the reward rate). "Unused" payout tokens topped up by the owner can always be withdrawn via the withdrawPayoutTokens function.

In normal yield farming protocol and staking protocol, the owner can add payment token or he can choose not to,

Even Sushiswap Master Chef has a emergency unstaking to make sure the insufficient reward does not block user's staking fund

> In the case that insufficient payout tokens have been supplied by the contract owner, thus utilizing the deposited payout tokens from the users, the owner can always add additional payout token capital to the contract, ensuring that calls to emergencyUnstakeAll succeed.

Even we use the assumption the admin has to provide the payout token, it is not possible, he can't just provide payout token forever,

token has a limited total supply

The only case the admin can provide payout token forever is he can control the token minting.

The claim reward and stake toward is designed to let user make money, not lose money

the emergency unstake is designed to protect user's which should always not reverting.

Still feel like leave #85 as invalid and leave this one as medium is a very reasonable severity categorization.

Oot2k

I agree that emergency unstake should never revert. I also agree with that the admin cant provide tokens forever. Even if the root cause here is that there is not enough supply provided, I think the impact is severe and quit likely to happen.

Sherlock should keep this in mind when reviewing the escaltion.

berndartmueller

> Even we use the assumption the admin has to provide the payout token, it is not possible, he can't just provide payout token forever,
>
> token has a limited total supply
>
> The only case the admin can provide payout token forever is he can control the token minting.

That depends on the used token. Many tokens allow minting additional funds by the owner, and many protocols have large token reserves.

Additionally, if rewards should be stopped, the owner can use the `setRewardRate` function to reset the reward rate to 0, **before** running out of payout tokens (payout tokens which were topped up by the owner to be used for minting option rewards).

JeffCX

> That depends on the used token. Many tokens allow minting additional funds by the owner, and many protocols have large token reserves.

In the case when the admin owner can do that, this is not a issue

Many protocol don't allow token minting and has a fixed supply, it is very common to hardcode the total supply.

Setting the reward rate to zero don't seem to help

Because in the current codebase, setting the reward rate to zero will only stop the reward from accuring in the current and future epoch

```
function setRewardRate(
    uint256 rewardRate_
) external onlyOwner requireInitialized updateRewards {
    // Check if a new epoch needs to be started
    // We do this to avoid bad state if the reward rate is changed when a new
 ↪  epoch should have started
    if (block.timestamp >= epochStart + epochDuration) _startNewEpoch();

    // Set the reward rate
    rewardRate = rewardRate_;
}
```

but the reward earned from past epoch is still for user to claim

If not payout token balance, user's staked balance is used as reward o()o

Oighty

I tend to side with the submitter on this one. The main reason being that staking contracts have often been used in the past with the staked token == payout token and others may want to use the contracts in that way. They also often work that rewards are halted if the owner does not add more funds to them (vs. having direct minting) as a safety measure. With that configuration, one user's stake could be used to pay rewards to another, which would be very bad.

Since the protocol is permissionless, anyone can create an OTLM pool. Therefore, the input isn't "admin" from the protocol perspective. I did message submitter that I thought this was a valuable find. The high likelihood that someone would try to use this configuration (based on past staking contract use) and users would lose funds make this a medium issue.

Oighty

Fix implemented at https://github.com/Bond-Protocol/options/pull/9

ctf-sec

To support the use case when payout token is staking token, the contract would need to distinguish the payout token and staking token balance, which add more complexity

the current fix make sure the staking token is not the same as the payout token when deploying the OTLM contract, fix looks good!

berndartmueller

> I tend to side with the submitter on this one. The main reason being that staking contracts have often been used in the past with the staked token == payout token and others may want to use the contracts in that way. They also often work that rewards are halted if the owner does not add more funds to them (vs. having direct minting) as a safety measure. With that configuration, one user's stake could be used to pay rewards to another, which would be very bad.
>
> Since the protocol is permissionless, anyone can create an OTLM pool. Therefore, the input isn't "admin" from the protocol perspective. I did message submitter that I thought this was a valuable find. The high likelihood that someone would try to use this configuration (based on past staking contract use) and users would lose funds make this a medium issue.

If the deployer of the permissionless OTML contract is not considered an admin, all issues which have their severity lowered, due to considering them as “admin issues”, should be re-evaluated IMHO (e.g., https://github.com/sherlock-audit/2023-06-bond-judging/issues/55#issuecomment-1632426869). Or alternatively, the OTML owner is considered an admin.

Happy to hear everyone’s thoughts!

SilentYuki

@berndartmueller does have a point here, by the sponsor comment:

```
Since the protocol is permissionless, anyone can create an OTLM pool.
Therefore, the input isn't "admin" from the protocol perspective.
```

Some of the other issues are judged as lows based on counting the owner of the OTLM as a trusted authority.

juntzhan

It was confirmed by sponsor that OTLM owner is trusted. https://discord.com/channels/812037309376495636/1125440840333013073/1126125997675249674

The dispute is if `not providing payout token is not an admin error` rather than if OTLM owner is admin.

0ighty

> Since the protocol is permissionless, anyone can create an OTLM pool. Therefore, the input isn't "admin" from the protocol perspective.

Sorry if this was confusing. I do think the OTLM owner is considered trusted. As stated before though, it's likely that someone would try to setup with this configuration and the impact is severe (user's lose deposited principal if it is used to fund rewards). Therefore, I view this as a bigger issue than more standard "misconfigurations", which do not put user principal at risk.

To phrase differently, I don't think it would be obvious to OTLM owners deploying the contract that this could happen. Therefore, the system should not allow this to happen.

hrishibhat

Result: Medium Unique Considering this a valid issue based on the points raised in the final comments from the Sponsor here. The information provided in the readme was not sufficient to understand the core functioning. Sherlock will make sure to avoid such situations in the future.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- berndartmueller: accepted

Issue M-6: IERC20(token).approve revert if the underlying ERC20 token approve does not return boolean

Source: https://github.com/sherlock-audit/2023-06-bond-judging/issues/86

Found by

Auditwolf, ctf_sec, pks_, tsvetanovv

Summary

IERC20(token).approve revert if the underlying ERC20 token approve does not return boolean

Vulnerability Detail

When transferring the token, the protocol use safeTransfer and safeTransferFrom

but when approving the payout token, the safeApprove is not used

for non-standard token such as USDT,

calling approve will revert because the solmate ERC20 enforce the underlying token return a boolean

https://github.com/transmissions11/solmate/blob/bfc9c25865a274a7827fea5abf6e4fb64fc64e6c/src/tokens/ERC20.sol#L68

```
function approve(address spender, uint256 amount) public virtual returns (bool) {
    allowance[msg.sender][spender] = amount;

    emit Approval(msg.sender, spender, amount);

    return true;
}
```

while the token such as USDT does not return boolean

https://etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7#code#L126

Impact

USDT or other ERC20 token that does not return boolean for approve is not supported as the payout token

Code Snippet

https://github.com/sherlock-audit/2023-06-bond/blob/fce1809f83728561dc75078d41ead6d60e15d065/options/src/fixed-strike/liquidity-mining/OTLM.sol#L190

https://github.com/sherlock-audit/2023-06-bond/blob/fce1809f83728561dc75078d41ead6d60e15d065/options/src/fixed-strike/liquidity-mining/OTLM.sol#L504

Tool used

Manual Review

Recommendation

Use safeApprove instead of approve

Discussion

Oighty

Agree with proposed solution.

Oighty

Fix implemented in https://github.com/Bond-Protocol/options/pull/8

ctf-sec

Fix looks good!

Issue M-7: Division before multiplication result in loss of token reward if the reward update time elapse is small

Source: https://github.com/sherlock-audit/2023-06-bond-judging/issues/87

Found by

TrungOre, berndartmueller, ctf_sec

Summary

Division before multiplication result in loss of token reward

Vulnerability Detail

When calcuting the reward, we are calling

```
function currentRewardsPerToken() public view returns (uint256) {
       // Rewards do not accrue if the total balance is zero
       if (totalBalance == 0) return rewardsPerTokenStored;

       // @audit
       // loss of precision
       // The number of rewards to apply is based on the reward rate and the
↪   amount of time that has passed since the last reward update
       uint256 rewardsToApply = ((block.timestamp - lastRewardUpdate) *
↪   rewardRate) /
           REWARD_PERIOD;

       // The rewards per token is the current rewards per token plus the
↪   rewards to apply divided by the total staked balance
       return rewardsPerTokenStored + (rewardsToApply * 10 **
↪   stakedTokenDecimals) / totalBalance;
   }
```

the precision loss can be high because the accumulated reward depends on the time elapse:

(block.timestamp - lastRewardUpdate)

and the REWARD_PERIOD is hardcoded to one days:

```
/// @notice Amount of time (in seconds) that the reward rate is distributed over
uint48 public constant REWARD_PERIOD = uint48(1 days);
```

if the time elapse is short and the currentRewardsPerToken is updated frequently, the precision loss can be heavy and even rounded to zero

the lower the token precision, the heavier the precision loss

> https://github.com/d-xo/weird-erc20#low-decimals
>
> Some tokens have low decimals (e.g. USDC has 6). Even more extreme, some tokens like Gemini USD only have 2 decimals.

consider as extreme case, if the reward token is Gemini USD, the reward rate is set to 1000 * 10 = 10 ** 4 = 10000

if the update reward keep getting called within 8 seconds

8 * 10000 / 86400 is already rounded down to zero and no reward is accuring for user

Impact

Division before multiplication result in loss of token reward if the reward update time elapse is small

Code Snippet

https://github.com/sherlock-audit/2023-06-bond/blob/fce1809f83728561dc75078d41ead6d60e15d065/options/src/fixed-strike/liquidity-mining/OTLM.sol#L544

Tool used

Manual Review

Recommendation

Avoid division before multiplcation and only perform division at last

Discussion

Oighty

Agree with potential precision loss at low decimal numbers and suggested fix.

Oighty

Fix implemented at https://github.com/Bond-Protocol/options/pull/7

ctf-sec

Fix looks good

Issue M-8: FixedStrikeOptionTeller: create can be invoked when block.timestamp == expiry but exercise reverts

Source: https://github.com/sherlock-audit/2023-06-bond-judging/issues/91

Found by

0xhunter526, TrungOre, berndartmueller, ctf_sec, dopeflamingo, lil.eth

Summary

In `FixedStrikeOptionTeller` contract, new option tokens can be minted when `block.timestamp == expiry` but these option tokens cannot be exercised even in the same transaction.

Vulnerability Detail

The `create` function has this statement:

```
if (uint256(expiry) < block.timestamp) revert Teller_OptionExpired(expiry);
```

The `exercise` function has this statement:

```
if (uint48(block.timestamp) >= expiry) revert Teller_OptionExpired(expiry);
```

Notice the >= operator which means when `block.timestamp == expiry` the `exercise` function reverts.

The `FixedStrikeOptionTeller.create` function is invoked whenever a user claims his staking rewards using `OTLM.claimRewards` or `OTLM.claimNextEpochRewards`. (here)

So if a user claims his rewards when `block.timestamp == expiry` he receives the freshly minted option tokens but he cannot exercise these option tokens even in the same transaction (or same block).

Moreover, since the receiver do not possess these freshly minted option tokens, he cannot `reclaim` them either (assuming `reclaim` function contains the currently missing `optionToken.burn` statement).

Impact

Option token will be minted to user but he cannot exercise them. Receiver cannot reclaim them as he doesn't hold that token amount.

This leads to loss of funds as the minted option tokens become useless. Also the scenario of users claiming at expiry is not rare.

Code Snippet

https://github.com/sherlock-audit/2023-06-bond/blob/main/options/src/fixed-strike/FixedStrikeOptionTeller.sol#L267 https://github.com/sherlock-audit/2023-06-bond/blob/main/options/src/fixed-strike/FixedStrikeOptionTeller.sol#L336

Tool used

Manual Review

Recommendation

Consider maintaining a consistent timestamp behaviour. Either prevent creation of option tokens at expiry or allow them to be exercised at expiry.

Discussion

Oighty

Agree with the proposed solution. Will change the timestamp checks to be consistent.

Oighty

Fix implemented at https://github.com/Bond-Protocol/options/pull/6

ctf-sec

Fix looks great!

Issue M-9: stake() missing set lastEpochClaimed when userBalance equal 0

Source: https://github.com/sherlock-audit/2023-06-bond-judging/issues/108

Found by

Delvir0, TrungOre, Yuki, berndartmueller, bin2chen, ctf_sec

Summary

because `stake()` don't set `lastEpochClaimed[user] = last epoch` if `userBalance` equal 0 So all new stake user must loop from 0 to `last epoch` for `_claimRewards()` As the epoch gets bigger and bigger it will waste a lot of GAS, which may eventually lead to `GAS_OUT`

Vulnerability Detail

in `stake()`, when the first-time stake() only `rewardsPerTokenClaimed[msg.sender]` but don't set `lastEpochClaimed[msg.sender]`

```
    function stake(
        uint256 amount_,
        bytes calldata proof_
    ) external nonReentrant requireInitialized updateRewards tryNewEpoch {
...
        uint256 userBalance = stakeBalance[msg.sender];
        if (userBalance > 0) {
            // Claim outstanding rewards, this will update the rewards per token
 ↪  claimed
            _claimRewards();
        } else {
            // Initialize the rewards per token claimed for the user to the
 ↪  stored rewards per token
@>          rewardsPerTokenClaimed[msg.sender] = rewardsPerTokenStored;
        }

        // Increase the user's stake balance and the total balance
        stakeBalance[msg.sender] = userBalance + amount_;
        totalBalance += amount_;

        // Transfer the staked tokens from the user to this contract
        stakedToken.safeTransferFrom(msg.sender, address(this), amount_);
    }
```

so every new staker , needs claims from 0

```
    function _claimRewards() internal returns (uint256) {
        // Claims all outstanding rewards for the user across epochs
        // If there are unclaimed rewards from epochs where the option token has
 ↪  expired, the rewards are lost

        // Get the last epoch claimed by the user
@>      uint48 userLastEpoch = lastEpochClaimed[msg.sender];

        // If the last epoch claimed is equal to the current epoch, then only
 ↪  try to claim for the current epoch
        if (userLastEpoch == epoch) return _claimEpochRewards(epoch);

        // If not, then the user has not claimed all rewards
        // Start at the last claimed epoch because they may not have completely
 ↪  claimed that epoch
        uint256 totalRewardsClaimed;
@>     for (uint48 i = userLastEpoch; i <= epoch; i++) {
            // For each epoch that the user has not claimed rewards for, claim
 ↪  the rewards
            totalRewardsClaimed += _claimEpochRewards(i);
        }

        return totalRewardsClaimed;
    }
```

With each new addition of epoch, the new stake must consumes a lot of useless loops, from loop 0 to `last epoch` When `epoch` reaches a large size, it will result in GAS_OUT and the method cannot be executed

Impact

When the `epoch` gradually increases, the new take will waste a lot of GAS When it is very large, it will cause GAS_OUT

Code Snippet

https://github.com/sherlock-audit/2023-06-bond/blob/main/options/src/fixed-strike/liquidity-mining/OTLM.sol#L324-L327

Tool used

Manual Review

Recommendation

```
    function stake(
        uint256 amount_,
        bytes calldata proof_
    ) external nonReentrant requireInitialized updateRewards tryNewEpoch {
...
        if (userBalance > 0) {
            // Claim outstanding rewards, this will update the rewards per token
↪   claimed
            _claimRewards();
        } else {
            // Initialize the rewards per token claimed for the user to the
↪   stored rewards per token
            rewardsPerTokenClaimed[msg.sender] = rewardsPerTokenStored;
+           lastEpochClaimed[msg.sender] = epoch;
        }
```

Discussion

0ighty

Agree with the proposed solution.

ctf-sec

Great finding, agree with medium severity

0ighty

Fix implemented at https://github.com/Bond-Protocol/options/pull/5

ctf-sec

Will look into this, seems all the duplicate suggest the fix:

```
lastEpochClaimed[msg.sender] = epoch;
```

but the implemented fix is

```
lastEpochClaimed[msg.sender] = epoch -1
```

maybe testing can help as well, just want to make sure there is no off-by-one issue o()o

0ighty

> Will look into this, seems all the duplicate suggest the fix:

```
lastEpochClaimed[msg.sender] = epoch;
```

but the implemented fix is

```
lastEpochClaimed[msg.sender] = epoch -1
```

maybe testing can help as well, just want to make sure there is no off-by-one issue o()o

The reason to set lastEpochClaimed to `epoch - 1` is that you want the user state to appear like they have claimed everything before the epoch they started staking on. They haven't claimed any tokens for the current epoch yet, so that would be inaccurate.

Oighty

@ctf-sec did you have a chance to review this more?

ctf-sec

Yes, fix looks good

Issue M-10: claimRewards() If a rewards is too small, it may block other epochs

Source: https://github.com/sherlock-audit/2023-06-bond-judging/issues/110

Found by

TrungOre, bin2chen, pep7siup

Summary

When `claimRewards()`, if some `rewards` is too small after being round down to 0 If `payoutToken` does not support transferring 0, it will block the subsequent epochs

Vulnerability Detail

The current formula for calculating rewards per cycle is as follows.

```
    function _claimEpochRewards(uint48 epoch_) internal returns (uint256) {
...
@>        uint256 rewards = ((rewardsPerTokenEnd - userRewardsClaimed) *
↪   stakeBalance[msg.sender]) /
                10 ** stakedTokenDecimals;
        // Mint the option token on the teller
        // This transfers the reward amount of payout tokens to the option
↪   teller in exchange for the amount of option tokens
        payoutToken.approve(address(optionTeller), rewards);
        optionTeller.create(optionToken, rewards);
```

Calculate `rewards` formula : `uint256 rewards = ((rewardsPerTokenEnd - userRewardsClaimed) * stakeBalance[msg.sender]) /10 ** stakedTokenDecimals;`

When `rewardsPerTokenEnd` is very close to `userRewardsClaimed`, `rewards` is likely to be round downs to 0 Some tokens do not support transfer(amount=0) This will revert and lead to can't claims

Impact

Stuck `claimRewards()` when the rewards of an epoch is 0

Code Snippet

https://github.com/sherlock-audit/2023-06-bond/blob/main/options/src/fixed-strike/liquidity-mining/OTLM.sol#L499

Tool used

Manual Review

Recommendation

```
    function _claimEpochRewards(uint48 epoch_) internal returns (uint256) {
.....

        uint256 rewards = ((rewardsPerTokenEnd - userRewardsClaimed) *
 ↪  stakeBalance[msg.sender]) /
            10 ** stakedTokenDecimals;
+       if (rewards == 0 ) return 0;
        // Mint the option token on the teller
        // This transfers the reward amount of payout tokens to the option
 ↪  teller in exchange for the amount of option tokens
        payoutToken.approve(address(optionTeller), rewards);
        optionTeller.create(optionToken, rewards);
```

Discussion

Oighty

Agree with the proposed solution.

ctf-sec

Agree, while the revert in 0 transfer is an edge case, this does block reward distribution and new epoch from starting

Oighty

Fix implemented at https://github.com/Bond-Protocol/options/pull/4

ctf-sec

Hi Oighty, just one more than to highlight

https://github.com/Bond-Protocol/options/blob/c4c365cf27983a175f844fe74d160c0610783585/src/fixed-strike/liquidity-mining/OTLM.sol#L192

```
/// @notice Modifier that tries to start a new epoch before a function is
 ↪  executed and rewards the caller for doing so
 modifier tryNewEpoch() {
     // If the epoch has ended, try to start a new one
     if (uint48(block.timestamp) >= epochStart + epochDuration) {
         _startNewEpoch();
         // Issue reward to caller for starting the new epoch
         payoutToken.approve(address(optionTeller), epochTransitionReward);
```

```
            FixedStrikeOptionToken optionToken = epochOptionTokens[epoch];
            optionTeller.create(optionToken, epochTransitionReward);
            ERC20(address(optionToken)).safeTransfer(msg.sender,
↪   epochTransitionReward);
        }
        _;
    }
```

if the owner set the epochTransitionReward to zero and the underlying payout token revert in zero amount transfer,

new epoch starting is blocked, the owner is not able to adjust epochTransitionReward in the current codebase

emm we can modify the code to

```
if (uint48(block.timestamp) >= epochStart + epochDuration) {
            _startNewEpoch();
            // Issue reward to caller for starting the new epoch
            if(epochTransitionReward > 0) {
                payoutToken.approve(address(optionTeller),
↪   epochTransitionReward);
                FixedStrikeOptionToken optionToken = epochOptionTokens[epoch];
                optionTeller.create(optionToken, epochTransitionReward);
                ERC20(address(optionToken)).safeTransfer(msg.sender,
↪   epochTransitionReward);
            }
        }
```

that would resolve the issue

also in the constructor of OTLM, can validate the epochTransitionReward to make sure it is not 0

or add a seperate admin function so the owner can update the epochTransitionReward parameter.

ctf-sec

Other than the comments, fix looks good

Oighty

@ctf-sec I added the zero check for the epoch transition reward and also added a setter function so the owner can change it.

ctf-sec

Thanks Oighty, then all good!