SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public Best Efforts
Prepared for:	Zap Protocol
Prepared by:	Sherlock
Lead Security Expert:	bughuntoor
Dates Audited:	March 14 - March 20, 2024
Prepared on:	May 15, 2024

Introduction

ZAP is a token launch and distribution protocol, initially built on Blast. It offers curated and permissionless token sales as well as chapter based air drops for projects. It operates on Mission Control where users complete tasks for allocation in sales, and points in air drops. ZAP also offers an overallocation layer where users can allocate their refunded capital as liquidity to protocols on Blast to double and triple farm airdrop points as well as earn additional APY's and Native Yield.

Scope

Repository: Lithium-Ventures/zap-contracts-labs

Branch: main

Commit: 1e0e114f8296c0fff62b8a2a96681eb357b0c0aa

Repository: Lithium-Ventures/zap-launches-contracts

Branch: develop

Commit: 671cf33bc2e6588542fe85a9cf7004e4a2b77d81

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
3	3

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

s1ce
bughuntoor
ydlee
Krace
ZdravkoHr.
Silvermist
GatewayGuardians
AMOW
audithare
cats
enfrasico
thank_you
0x4non
HonorLt
klaus
NickV
merlin
Varun_05
404666
0xhashiman
mike-watson
dipp
UbiquitousComputing
no
nilay27
cawfree
denzi_
turvec
BengalCatBalu
novaman33
0xR360
aman
psb01

Issue H-1: Tax refund is calculated based on the wrong amount

Source: https://github.com/sherlock-audit/2024-03-zap-protocol-judging/issues/57

The protocol has acknowledged this issue.

Found by

Krace, bughuntoor, s1ce, ydlee

Summary

Tax refund is calculated based on the wrong amount

Vulnerability Detail

After the private period has finished, users can claim a tax refund, based on their max tax free allocation.

```
(s.share, left) = _claim(s);
require(left > 0, "TokenSale: Nothing to claim");
uint256 refundTaxAmount;
if (s.taxAmount > 0) {
    uint256 tax = userTaxRate(s.amount, msg.sender);
    uint256 taxFreeAllc = _maxTaxfreeAllocation(msg.sender) * PCT_BASE;
    if (taxFreeAllc >= s.share) {
        refundTaxAmount = s.taxAmount;
    } else {
        refundTaxAmount = (left * tax) / POINT_BASE;
    }
    usdc.safeTransferFrom(marketingWallet, msg.sender, refundTaxAmount);
}
```

The problem is that in case `s.share > taxFreeAllc`, the tax refund is calculated wrongfully. Not only it should refund the tax on the unused USDC amount, but it should also refund the tax for the tax-free allocation the user has.

Imagine the following.

1. User deposits 1000 USDC.
2. Private period finishes, token oversells. Only half of the user's money actually go towards the sell (s.share = 500 USDC, s.left = 500 USDC)
3. The user has 400 USDC tax-free allocation

4. The user must be refunded the tax for the 500 unused USDC, as well as their 400 USDC tax-free allocation. In stead, they're only refunded for the 500 unused USDC. (note, if the user had 500 tax-free allocation, they would've been refunded all tax)

Impact

Users are not refunded enough tax

Code Snippet

https://github.com/sherlock-audit/2024-03-zap-protocol/blob/main/zap-contracts-labs/contracts/TokenSale.sol#L385

Tool used

Manual Review

Recommendation

change the code to the following:

```
refundTaxAmount = ((left + taxFreeAllc) * tax) / POINT_BASE;
```

Discussion

ZdravkoHr

Escalate

Users are not supposed to be refunded for the tax free allocation, same reasoning as in 58.

sherlock-admin2

> Escalate
>
> Users are not supposed to be refunded for the tax free allocation, same reasoning as in 58.

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.

koreanspicygarlic1

escalation by watson is plain wrong. Users have taxfree allocation for which their tax should be refunded. The total refund should be for the taxfree allocation + tax on unused funds (s.left)

ZdravkoHr

@koreanspicygarlic1, users are not paying tax for the tax free allocation when depositing. What should they be refunded for then?

merc1995

@ZdravkoHr , users pay tax for the tax-free allocation when depositing because the `taxFreeAllcOfUser` is zero, as I mentioned in #7 .

vsharma4394

@merc1995 initially there is no tax free allocation so why should tax free amount be taken into consideration when claim function is called. The tax amount refunded to the user should be calculated for the left amount only i.e not taking into consideration the tax free amount. Otherwise every user would purposely make totalPrivateSold > totalSupplyValue so that they can avoid the tax.

merc1995

@vsharma4394 because the code says it should consider the tax free.

```
function claim() external {
    checkingEpoch();
    require(
        uint8(epoch) > 1 && !admin.blockClaim(address(this)),
        "TokenSale: Not time or not allowed"
    );

    Staked storage s = stakes[msg.sender];
    require(s.amount != 0, "TokenSale: No Deposit");
    require(!s.claimed, "TokenSale: Already Claimed");

    uint256 left;
    (s.share, left) = _claim(s);
    require(left > 0, "TokenSale: Nothing to claim");
    uint256 refundTaxAmount;
    if (s.taxAmount > 0) {
        uint256 tax = userTaxRate(s.amount, msg.sender);
        uint256 taxFreeAllc = _maxTaxfreeAllocation(msg.sender) * PCT_BASE;
        ↪  //==> tax free
        if (taxFreeAllc >= s.share) {
            refundTaxAmount = s.taxAmount;
        } else {
            refundTaxAmount = (left * tax) / POINT_BASE;
```

```
        }
        usdc.safeTransferFrom(marketingWallet, msg.sender, refundTaxAmount);
    }
    s.claimed = true;
    usdc.safeTransfer(msg.sender, left);
    emit Claim(msg.sender, left);
}
```

ZdravkoHr

@merc1995, that's because if the taxFree was set to 500 for example amd I deposit 600, the claim function should consider this and let me claim tax for only 100 (assumung I can claim 100%, which of course is not true, but makes the example more simple)

merc1995

The code use `left` rather than `s.shrare - taxFree` to calculate the `refundAmount`. According to your comment, this issue is valid because of the wrong implementation.

merc1995

> The tax amount refunded to the user should be calculated for the left amount only i.e not taking into consideration the tax free amount @vsharma4394 Could you please provide the doc which said that the tax free amount should not be taken into consideration?

vsharma4394

@merc1995 It is clear from the deposit function (which calls the _processPrivate function) that currently they don't allow tax free allocation.

merc1995

So there should be no tax-free releated code in the `claim`, and this issue should be valid?

vsharma4394

I agree that there should not be tax free related code and refundTaxAmount should only be equal to (left * tax) / POINT_BASE.

vsharma4394

I don't agree with the following lines as stated in the vulnerability detail that The problem is that in case s.share > taxFreeAllc, the tax refund is calculated wrongfully. Not only it should refund the tax on the unused USDC amount, but it should also refund the tax for the tax-free allocation the user has. They should

never refund tax for the tax Free allocation because while calling deposit taxFreeAllocation was set to zero.

vsharma4394

According to me there is a logic error in the code where they first don't take into account taxFreeAllocation but they do so in claim function due to which there has been lot of miss interpretation of the code.

Hash01011122

I missed the part where user is not supposed to be refunded for tax free allocation, it would be better if someone with deeper knowledge of protocol can shed some light on taxFreeAllocation concept. Requesting insight from someone with deeper knowledge of the protocol is a prudent step. @ZdravkoHr @deadrosesxyz ??

ZdravkoHr

@Hash01011122, if you look at TokenSale.processPrivate and assume the tax free amount is set to an extremely large number, the if statement where the actual tax transfer happens will never be entered.

So the tax free allocation is tax-free because users don't pay for it from the very beginning. That's why a refund is not needed.

spacegliderrrr

@Hash01011122 When users deposit, they always have to pay tax. The users then have a tax-free allocation. Meaning, that for this amount, they don't have to pay tax and they'll be refunded for it.

Tax refund must always be for the tax on their tax-free allocation.+ the tax on their unused funds (s.left)

niketansainiantier

we are taking the tax on the whole invested amount including all allocations, including the whitelist. So will refund the only tax on left amount(extra Amount).

deadrosesxyz

@niketansainiantier if this was the case, then what is the purpose of `_maxTaxfreeAllocation` and the following lines of code? Why would they refund the whole tax amount if it was always intended to refund only based on the left/extra amount. What you've said simply contradicts the code.

```
uint256 taxFreeAllc = _maxTaxfreeAllocation(msg.sender) * PCT_BASE;
if (taxFreeAllc >= s.share) {
    refundTaxAmount = s.taxAmount;
```

Hash01011122

Highlighting the specific code line where `taxFreeAllocOfUser` is hardcoded to zero implies that users are indeed intended to be processed for a tax refund according to the protocol's logic. Let me know if I am getting anything wrong here @ZdravkoHr

vsharma4394

@Hash01011122 the following is said by @niketansainiantier `we are taking the tax on the whole invested amount including all allocations, including the whitelist. So will refund the only tax on left amount(extra Amount).` Due to this users are only refunded on the left amount i.e not considering the tax free allocation. While depositing also taxFreeAllocation was not considered as taxFreeAllocOfUser is equal to zero so it should also not be considered while calling claim function.

I think you have misunderstood the code ,`taxfreeAllocOfUser` is hardcoded to zero implies that users are processed tax on whole amount neglecting the tax free amount.So as now tax free amount is not taken into consideration , it should not be taken into account while calling claim function.

vsharma4394

I think sponsers have also added the won't fix tag because it is intended behaviour to refund tax only on left amount. As initially also taxFreeAlloc was taken as zero. Thus making this finding as invalid.

ZdravkoHr

@vsharma4394 is right imo. There is a comment next to taxAlloc = 0 that says that all pools have tax

vsharma4394

> @vsharma4394 is right imo. There is a comment next to taxAlloc = 0 that says that all pools have tax

Yes,there has been a lot of misunderstanding because people have not understood wwhat taxAlloc = 0 means.

vsharma4394

> @niketansainiantier if this was the case, then what is the purpose of `_maxTaxfreeAllocation` and the following lines of code? Why would they refund the whole tax amount if it was always intended to refund only based on the left/extra amount. What you've said simply contradicts the code.
>
> ```
> uint256 taxFreeAllc = _maxTaxfreeAllocation(msg.sender) * PCT_BASE;
> if (taxFreeAllc >= s.share) {
>     refundTaxAmount = s.taxAmount;
> ```

Again this is because they have taken taxFreeAlloc = 0. Hence this issue is invalid.

niketansainiantier

Yes, that's why I changed the logic here.

`uint256 left; (s.share, left) = _claim(s); require(left > 0, "TokenSale: Nothing to claim"); uint256 refundTaxAmount; if (s.taxAmount > 0) { uint256 tax = userTaxRate(s.amount, msg.sender); refundTaxAmount = (left * tax) / POINT_BASE; usdc.safeTransferFrom(marketingWallet, msg.sender, refundTaxAmount); }` You guys can check this on 2nd PR

vsharma4394

> Yes, that's why I changed the logic here.
>
> `uint256 left; (s.share, left) = _claim(s); require(left > 0, "TokenSale: Nothing to claim"); uint256 refundTaxAmount; if (s.taxAmount > 0) { uint256 tax = userTaxRate(s.amount, msg.sender); refundTaxAmount = (left * tax) / POINT_BASE; usdc.safeTransferFrom(marketingWallet, msg.sender, refundTaxAmount); }` You guys can check this on 2nd PR

@Hash01011122 now things should be very clear. This issue should be invalid and #159 should be a valid unique finding.Also #58 should also be invalid.

ZdravkoHr

131 is also a dup of 159

Hash01011122

Seems logical enough, @vsharma4394 thanks for clearing the misunderstanding regarding `taxfreeAllocOfUser` I am inclined towards invalidating this issue. @deadrosesxyz Your input would be appreciated.

deadrosesxyz

@Hash01011122 The other watson simply assumes that `_maxTaxFreeAllocation` is always going to return 0. They assume the logic here, including the call to the staking contract, the tokensale tiers will all return 0 for whatever reason. There's contract logic based on the calculated `taxFreeAllc` and logic to calculate its value. `taxFreeAllc` always having a value of 0 is not enforced anywhere, hence cannot be expected this would be the case (in fact it's just the opposite considering the function designed to calculate it). I'd politely ask other watsons to refrain from making any more comments. I believe everyone has made their point clear and it's simply time to wait for judge's decision.

```
uint256 taxFreeAllc = _maxTaxfreeAllocation(msg.sender) * PCT_BASE;
if (taxFreeAllc >= s.share) {
```

```
    refundTaxAmount = s.taxAmount;
```

```
function _maxTaxfreeAllocation(address _sender) internal returns (uint256) {
    uint256 userTierAllc = stakingContract.getAllocationOf(_sender);
    uint256 giftedTierAllc = tokensaleTiers[_sender];

    if (userTierAllc > giftedTierAllc) {
        return userTierAllc;
    } else {
        return giftedTierAllc;
    }
}
```

Hash01011122

Even I thought the same when I first looked at `taxFreeAllc` in codebase, but after @niketansainiantier cleared that this is not the case. Just want to know whether this was mentioned in by the sponsors at the time of contest. If nothing was mentioned this should remain a valid issue.

Evert0x

Planning to reject escalation and keep issue as is.

vsharma4394

> Yes, that's why I changed the logic here.
>
> `uint256 left; (s.share, left) = _claim(s); require(left > 0, "TokenSale: Nothing to claim"); uint256 refundTaxAmount; if (s.taxAmount > 0) { uint256 tax = userTaxRate(s.amount, msg.sender); refundTaxAmount = (left * tax) / POINT_BASE; usdc.safeTransferFrom(marketingWallet, msg.sender, refundTaxAmount); }` You guys can check this on 2nd PR

@Evert0x see here the changed code by sponsers now tax amount which is returned is only calculated on the left amount excluding tax free allocation because which is contradiction to this finding, so doesn't that make this finding invalid.

vsharma4394

@Evert0x In the following loc https://github.com/sherlock-audit/2024-03-zap-protocol/blob/c2ad35aa844899fa24f6ed0cbfcf6c7e611b061a/zap-contracts-labs/contracts/TokenSale.sol#L227 It is clearly commented that tax is charged for every pool, i.e the tax free calculation is not done so it doesn't makes sense that they would refund the tax on tax free allocation and similarly the protocol doesn't wishes to take into account tax free allocation while calling claim function that is why they changed the code in latest pr.Now its best you ask the sponsers and take your

decision. Sponsers have made that change by looking at #159 so asking sponsers would definitely resolve that issue too.

Evert0x

Result: High Has Duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- ZdravkoHr: rejected

Issue H-2: If token does not oversell, users cannot claim tax refund on their tax free allocation.

Source: https://github.com/sherlock-audit/2024-03-zap-protocol-judging/issues/58

The protocol has acknowledged this issue.

Found by

bughuntoor, s1ce

Summary

Users may not be able to claim tax refund

Vulnerability Detail

Within TokenSale, upon depositing users, users have to pay tax. Then, users can receive a tax-free allocation - meaning they'll be refunded the tax they've paid on part of their deposit.

The problem is that due to a unnecessary require check, users cannot claim their tax refund, unless the token has oversold.

```
function claim() external {
        checkingEpoch();
        require(
            uint8(epoch) > 1 && !admin.blockClaim(address(this)),
            "TokenSale: Not time or not allowed"
        );

        Staked storage s = stakes[msg.sender];
        require(s.amount != 0, "TokenSale: No Deposit");
        require(!s.claimed, "TokenSale: Already Claimed");

        uint256 left;
        (s.share, left) = _claim(s);
        require(left > 0, "TokenSale: Nothing to claim");  // @audit -
↪   problematic line
        uint256 refundTaxAmount;
        if (s.taxAmount > 0) {
            uint256 tax = userTaxRate(s.amount, msg.sender);
            uint256 taxFreeAllc = _maxTaxfreeAllocation(msg.sender) * PCT_BASE;
            if (taxFreeAllc >= s.share) {
                refundTaxAmount = s.taxAmount;
```

```
            } else {
                refundTaxAmount = (left * tax) / POINT_BASE; // tax refund is on
↪  the wrong amount
            }
            usdc.safeTransferFrom(marketingWallet, msg.sender, refundTaxAmount);
        }
        s.claimed = true;
        usdc.safeTransfer(msg.sender, left);
        emit Claim(msg.sender, left);
    }
```

```
function _claim(Staked memory _s) internal view returns (uint120, uint256) {
    uint256 left;
    if (state.totalPrivateSold > (state.totalSupplyInValue)) {
        uint256 rate = (state.totalSupplyInValue * PCT_BASE) /
            state.totalPrivateSold;
        _s.share = uint120((uint256(_s.amount) * rate) / PCT_BASE);
        left = uint256(_s.amount) - uint256(_s.share);
    } else {
        _s.share = uint120(_s.amount);
    }

    return (_s.share, left);
}
```

`left` only has value if the token has oversold. Meaning that even if the user has an infinite tax free allocation, if the token has not oversold, they won't be able to claim a tax refund.

Impact

loss of funds

Code Snippet

https://github.com/sherlock-audit/2024-03-zap-protocol/blob/main/zap-contracts-labs/contracts/TokenSale.sol#L377

Tool used

Manual Review

Recommendation

Remove the require check

Discussion

ZdravkoHr

Escalate

This should not be a valid issue. The idea of the `claim` function is to let investors claim the surplus amount that is left after the ICO has ended, i.e when tokens `oversell`.

```
If the the demand is higher than supply, the number of tokens investors will
↪  receive is adjusted, and then the native token used to invest are partially
↪  refunded.
```

The following claim stated in the report is also wrong: `Then, users can receive a tax-free allocation - meaning they'll be refunded the tax they've paid on part of their deposit.`

Tax free allocation does not mean users will pay all the taxes and will be refunded later for the tax free amount. They are just not charged for the given amount from the very beginning of the deposit process. So they should not receive any refund.

This is evident from the way the tax is calculated inTokenSale._processPrivate()

```
if (sum > taxFreeAllcOfUser) {
    uint256 userTxRate = userTaxRate(sum, _sender);
    if (s.amount < taxFreeAllcOfUser) {
        userTaxAmount =
            ((sum - taxFreeAllcOfUser) * userTxRate) /
            POINT_BASE;
    } else {
        userTaxAmount = (amount * userTxRate) / POINT_BASE;
    }
}
```

The reason why there is a tax refund logic in the `claim` function is because users that claim back `amount` of tokens will not have these tokens as deposited in the end of the ICO, therefore they should be refunded the tax they have paid for them.

sherlock-admin2

> Escalate
>
> This should not be a valid issue. The idea of the `claim` function is to let investors claim the surplus amount that is left after the ICO has ended,

i.e when tokens `oversell`.

```
If the the demand is higher than supply, the number of tokens investors
↪  will receive is adjusted, and then the native token used to invest are
↪  partially refunded.
```

The following claim stated in the report is also wrong: `Then, users can receive a tax-free allocation - meaning they'll be refunded the tax they've paid on part of their deposit.`

Tax free allocation does not mean users will pay all the taxes and will be refunded later for the tax free amount. They are just not charged for the given amount from the very beginning of the deposit process. So they should not receive any refund.

This is evident from the way the tax is calculated inTokenSale._processPrivate()

```
if (sum > taxFreeAllcOfUser) {
    uint256 userTxRate = userTaxRate(sum, _sender);
    if (s.amount < taxFreeAllcOfUser) {
        userTaxAmount =
            ((sum - taxFreeAllcOfUser) * userTxRate) /
            POINT_BASE;
    } else {
        userTaxAmount = (amount * userTxRate) / POINT_BASE;
    }
}
```

The reason why there is a tax refund logic in the `claim` function is because users that claim back `amount` of tokens will not have these tokens as deposited in the end of the ICO, therefore they should be refunded the tax they have paid for them.

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.

koreanspicygarlic1

This escalation is also plain wrong, watson has not properly understood the design of the system.

vsharma4394

If the above reasoning is wrong then my issue #159 which has been invalidated

becomes a valid issue,so can someone escalate that too. @Hash01011122 please look into it carefully please.

vsharma4394

Hardcoding the value of taxfreeAllocation to zero implies that the protocol doesn't allow for taxFreeAllocation to occur as of now. So this directly implies that while claiming also taxFreeAmount should not be taken into consideration while refunding the amount to the user.

Hash01011122

Well I don't thoroughly understand the basis of this escalation as it is clearly mentioned in the codebase how tax system is calculated, where the tax-free allocation isn't zero.

vsharma4394

@Hash01011122 the following loc in _processPrivate function which is called when user deposits usdc has caused different understanding of he code

```
uint256 taxFreeAllcOfUser = 0; // hardcode zero - all pools have ax

        uint256 userTaxAmount;

        if (sum > taxFreeAllcOfUser) {
            uint256 userTxRate = userTaxRate(sum, _sender);
            if (s.amount < taxFreeAllcOfUser) {
                userTaxAmount =
                    ((sum - taxFreeAllcOfUser) * userTxRate) /
                    POINT_BASE;
            } else {
                userTaxAmount = (amount * userTxRate) / POINT_BASE;
            }
        }
```

Due to hardcoded value of taxFreeAllocation as zero ,user tax amount is calculated as follows userTaxAmount = (amount * userTxRate) / POINT_BASE i.e while depositing users have to pay tax irrespective of having taxFreeAllocation. If logic is never executed. So while claiming also taxFreeAllocation should also not be taken into account so as be consistent with the code. (Making taxFreeAllocation to zero is intended design.)

I think confusion has arised from considering taxFreeAllocation but it is considered as zero from starting due to hardcoded value of taxFreeAllocation as zero.

Asking from sponsers is the best way to deal with all the issue related to claim function and tax related issue.

ZdravkoHr

The issue is not that tax free allocation is 0. Even if it wasn't and the firsr if was entered, the user would not have been taxed for it. That's why I believe a refund should not be made

niketansainiantier

we are not giving a refund for the tax if a sale does not reach the hard cap.

vsharma4394

@niketansainiantier so this issue should be invalid right?

Hash01011122

Well I agree with what @vsharma4394 has mentioned above and came to the same conclusion when I revisited codebase. @ZdravkoHr and even sponsors have confirmed this one. This is valid finding.

vsharma4394

@Hash01011122 i dont think that this issue is valid because @niketansainiantier clearly mentioned the following `we are taking the tax on the whole invested amount including all allocations, including the whitelist. So will refund the only tax on left amount(extra Amount).` Now if token oversells tax is refunded based on the extra amount which should not consider taxFreeAllocation as said by the sponsers. Thus in claim function

```
if (s.taxAmount > 0) {
            uint256 tax = userTaxRate(s.amount, msg.sender);
            uint256 taxFreeAllc = _maxTaxfreeAllocation(msg.sender) * PCT_BASE;
            if (taxFreeAllc >= s.share) {
                refundTaxAmount = s.taxAmount;
            } else {
                refundTaxAmount = (left * tax) / POINT_BASE; // tax refund is on
↪  the wrong amount
            }
            usdc.safeTransferFrom(marketingWallet, msg.sender, refundTaxAmount);
        }
```

RefundTaxAmount should always be refundTaxAmount = (left * tax) / POINT_BASE because taxFreeAlloc is hardcoded as zero initially which clearly indicates protocol doesn't allow for taxFreeAllocation as of now.

Evert0x

I believe the escalation should be rejected the issue should stay as is.

Users should be able to get a tax refund on their tax-free allocation.

> **Tax free allocation does not mean users will pay all the taxes and will be refunded later for the tax free amount.** They are just not charged for the given amount from the very beginning of the deposit process. So they should not receive any refund.

This is not true, would like to see a link to a public message or to a code comment as a counter argument

vsharma4394

> I believe the escalation should be rejected the issue should stay as is.
>
> Users should be able to get a tax refund on their tax-free allocation.
>
> > **Tax free allocation does not mean users will pay all the taxes and will be refunded later for the tax free amount.** They are just not charged for the given amount from the very beginning of the deposit process. So they should not receive any refund.
>
> This is not true, would like to see a link to a public message or to a code comment as a counter argument

@Evert0x i agree the above reasoning is incorrect, please look at my reasoning and then decide

Evert0x

@vsharma4394 My understanding that the codebase is taxing all deposited and provide a tax-return on unallocated part + tax-free allocation. Which makes this issue a valid issue.

So if the token doesn't oversell, we should still take into account the tax free allocation.

I don't understand how your previous comment provides an argument against that.

vsharma4394

> @vsharma4394 My understanding that the codebase is taxing all deposited and provide a tax-return on unallocated part + tax-free allocation. Which makes this issue a valid issue.
>
> So if the token doesn't oversell, we should still take into account the tax free allocation.
>
> I don't understand how your previous comment provides an argument against that.

This would most probably because of the protocol design @niketansainiantier can answer it the best.

Evert0x

Result: High Has Duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- ZdravkoHr: rejected

Issue H-3: Reentrancy in Vesting.sol:claim() will allow users to drain the contract due to executing .call() on user's address before setting s.index = uint128(i)

Source:
https://github.com/sherlock-audit/2024-03-zap-protocol-judging/issues/157

Found by

0x4non, 0xR360, 0xhashiman, 404666, AMOW, BengalCatBalu, HonorLt, Silvermist, UbiquitousComputing, Varun_05, ZdravkoHr., aman, bughuntoor, cats, cawfree, denzi_, dipp, enfrasico, klaus, mike-watson, nilay27, no, novaman33, psb01, s1ce, thank_you, turvec

Summary

Reentrancy in Vesting.sol:claim() will allow users to drain the contract due to executing .call() on user's address before setting s.index = uint128(I)

Vulnerability Detail

Here is the Vesting.sol:claim() function:

```
function claim() external {
        address sender = msg.sender;

        UserDetails storage s = userdetails[sender];
        require(s.userDeposit != 0, "No Deposit");
@>      require(s.index != vestingPoints.length, "already claimed");
        uint256 pctAmount;
        uint256 i = s.index;
        for (i; i <= vestingPoints.length - 1; i++) {
            if (block.timestamp >= vestingPoints[i][0]) {
                pctAmount += (s.userDeposit * vestingPoints[i][1]) / 10000;
            } else {
                break;
            }
        }
        if (pctAmount != 0) {
            if (address(token) == address(1)) {
@>              (bool sent, ) = payable(sender).call{value: pctAmount}("");
                require(sent, "Failed to send BNB to receiver");
            } else {
                token.safeTransfer(sender, pctAmount);
```

```
            }
@>          s.index = uint128(i);
            s.amountClaimed += pctAmount;
        }
    }
```

From the above, You'll notice the claim() function checks if the caller already claimed by checking if the s.index has already been set to vestingPoints.length. You'll also notice the claim() function executes .call() and transfer the amount to the caller before setting the s.index = uint128(i), thereby allowing reentrancy.

Let's consider this sample scenario:

- An attacker contract(alice) has some native pctAmount to claim and calls `claim()`.
- "already claimed" check will pass since it's the first time she's calling `claim()` so her s.index hasn't been set
- However before updating Alice s.index, the Vesting contract performs external .call() to Alice with the amount sent as well
- Alice reenters `claim()` again on receive of the amount
- bypass index "already claimed" check since this hasn't been updated yet
- contract performs external .call() to Alice with the amount sent as well again,
- Same thing happens again
- Alice ends up draining the Vesting contract

Impact

Reentrancy in Vesting.sol:claim() will allow users to drain the contract

Code Snippet

https://github.com/sherlock-audit/2024-03-zap-protocol/blob/main/zap-contracts-labs/contracts/Vesting.sol#L84 https://github.com/sherlock-audit/2024-03-zap-protocol/blob/main/zap-contracts-labs/contracts/Vesting.sol#L89

Tool used

Manual Review

Recommendation

Here is the recommended fix:

```
if (pctAmount != 0) {
+           s.index = uint128(i);
            if (address(token) == address(1)) {
                (bool sent, ) = payable(sender).call{value: pctAmount}("");
                require(sent, "Failed to send BNB to receiver");
            } else {
                token.safeTransfer(sender, pctAmount);
            }
-           s.index = uint128(i);
            s.amountClaimed += pctAmount;
        }
```

I'll also recommend using reentrancyGuard.

Discussion

midori-fuse

Escalate

Per Sherlock's duplication rule:

> In the above example if the root issue A is one of the following generic vulnerabilities:
>
> - Reentrancy
> - Access control
> - Front-running
>
> Then the submissions with valid attack paths and higher vulnerability are considered valid. If the submission is vague or does not identify the attack path with higher severity clearly it will be considered low.
>
> - B is a valid issue
> - C is low

The following submissions fail to and/or incorrectly identify the root cause that enables the attack path: #6 #34 #66 #68 #79 #90 #98 #132 #149 .

- The issues in this category should be Low.

The following submissions are somewhat vague, but did manage to identify the erroneous storage variable that leads to re-entrancy (`s.index`): #10 #53 #104 #138 #186 (and a few more).

- While they did not (or vaguely) described the "attack path", the attack path here is just "directly calling `claim()` in your `receive()`", so I suppose one can be ok with just spelling out the function and the wrong storage variable.
- Since submission quality is subjective, I am flagging these issues so the judges can help with reviewing dupes. Personally I think these submissions are still acceptable, but leaving to the judges to decide the where the bar is.

sherlock-admin2

> Escalate
>
> Per Sherlock's duplication rule:
>
> > In the above example if the root issue A is one of the following generic vulnerabilities:
> >
> > - Reentrancy
> > - Access control
> > - Front-running
> >
> > Then the submissions with valid attack paths and higher vulnerability are considered valid. If the submission is vague or does not identify the attack path with higher severity clearly it will be considered low.
> >
> > - B is a valid issue
> > - C is low
>
> The following submissions fail to and/or incorrectly identify the root cause that enables the attack path: #6 #34 #66 #68 #79 #90 #98 #132 #149 .
>
> - The issues in this category should be Low.
>
> The following submissions are somewhat vague, but did manage to identify the erroneous storage variable that leads to re-entrancy (`s.index`): #10 #53 #104 #138 #186 (and a few more).
>
> - While they did not (or vaguely) described the "attack path", the attack path here is just "directly calling `claim()` in your `receive()`", so I suppose one can be ok with just spelling out the function and the wrong storage variable.
> - Since submission quality is subjective, I am flagging these issues so the judges can help with reviewing dupes. Personally I think these submissions are still acceptable, but leaving to the judges to decide the where the bar is.

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.

Nilay27

> Escalate
>
> Per Sherlock's duplication rule:
>
> > In the above example if the root issue A is one of the following generic vulnerabilities:
> >
> > - Reentrancy
> > - Access control
> > - Front-running
> >
> > Then the submissions with valid attack paths and higher vulnerability are considered valid. If the submission is vague or does not identify the attack path with higher severity clearly it will be considered low.
> >
> > - B is a valid issue
> > - C is low
>
> The following submissions fail to and/or incorrectly identify the root cause that enables the attack path: #6 #34 #66 #68 #79 #90 #98 #132 #134 #149 .
>
> - The issues in this category should be Low.
>
> The following submissions are somewhat vague, but did manage to identify the erroneous storage variable that leads to re-entrancy (`s.index`): #10 #53 #104 #138 #186 (and a few more).
>
> - While they did not (or vaguely) described the "attack path", the attack path here is just "directly calling `claim()` in your `receive()`", so I suppose one can be ok with just spelling out the function and the wrong storage variable.
> - Since submission quality is subjective, I am flagging these issues so the judges can help with reviewing dupes. Personally I think these submissions are still acceptable, but leaving to the judges to decide the where the bar is.

#134 identifies the issue of how the re-entrance occurs and suggests the same remediation. It clearly explains the following: "The vulnerability arises from the

contract's failure to update a user's claim state (s.index and s.amountClaimed) before transferring funds to the user, which allows a malicious contract to receive the funds and re-enter the claim function before the original call completes, potentially claiming more funds repeatedly."

The recommendation suggests updating the state before or using a reentrancy guard.

I am unsure why that has been included in the `low` category per your escalation?

midori-fuse

@Nilay27 I suppose you are right. Sorry about that, there are just too many dupes here, I might have confused it with another issue that got lost somewhere.

But be assured that unless the head of judging downright disagrees with me, all dupes will be reviewed and judged accordingly. Once again I'm sorry for my mistake.

novaman33

My issue - #10 does show the root cause clearly and does suggest a thorough recommendation for the mitigation. I do not agree it is vague.

keesmark

It is the same as this one, but why is it considered invalid? #119

novaman33

Probably because #119 says that reentrancy will occur when transferring erc20 tokens while call is used to transfer eth.

ZdravkoHr

Also, BNB is out of scope

Hash01011122

Acknowledging that every mentioned issue accurately identifies both the root cause of the vulnerability and the correct attack paths, yet noting the straightforward nature of the issue as a reason for minimal effort in Watson's issue, suggests a potential oversight in the importance of comprehensive reporting.

shubham-antier

Issue resolved: Moved the updations above the transfers. Also, added a reentrancy guard to better the security.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Lithium-Ventures/zap-contracts-labs/pull/2

Evert0x

@Hash01011122 what's your proposal on the exact family for this issue? Which reports should be excluded/included?

Hash01011122

@Evert0x Had a indepth review of this family of issues: Issues which can be excluded are: #6, #10, #34, #66, #79, #90, #132, #138, #149. The pinpoint the root cause but fail to explain any attack vector.

armormadeofwoe

> @Evert0x Had a indepth review of this family of issues: Issues which can be excluded are: #6, #10, #34, #66, #79, #90, #132, #138, #149. The pinpoint the root cause but fail to explain any attack vector.

Hi @Hash01011122, with all due respect, I believe #138 should remain valid since it showcases: root cause - sending funds before updating variables (breach of CEI pattern) attack path - the ability to trigger an arbitrary fallback function due to sending native ETH that could re-enter the same function and continue claiming funds due to the unchanged variables.

I do agree that my report is a little short as this is arguably the most known and recognizable issue in this space, decided to spare the judges some extra reading.

Hash01011122

lmao #138 should be excluded as I mentioned above,

0x3agle

@Hash01011122 #6 accurately identifies the root cause and the attack path. Root cause:

> If the token == address(1) (i.e. the native token) it performs an external call which sends the token to msg.sender and then updates the storage variable.

Attack Path:

> This allows an attacker to reenter the claim function until the contract is drained completely.

Hash01011122

@0x3agle with all due respect your report doesn't mention any appropriate Attack Path.

0x3agle

@Hash01011122

Issue: storage variable updated after external call Attack path: reentering the claim function Impact: Contract drained Mitigation: Follow CEI, add non-reentrant

Isn't this enough for this issue to be considered a valid one?

This issue is so obvious I didn't feel the need for a PoC to convey my point.

Having said that, I respect your decision and will accept it.

Hash01011122

Hey, if we look from that lens even issues like #10, #34, #66, #132 and #138 should be valid too. I understand what you are pointing even I don't want to invalidate any of the issues as I understand watson's would not spend more effort on writing low hanging fruit issues, However, I'm just adhering to Sherlock's rulebook. Do you want to add anything here @0x3agle?

Evert0x

> @Evert0x Had a indepth review of this family of issues: Issues which can be excluded are: #6, #10, #34, #66, #79, #90, #132, #138, #149. The pinpoint the root cause but fail to explain any attack vector.

Planning to accept escalation and move remove the reports mentioned by the Lead Judge as duplicates

0x3agle

@Hash01011122 @Evert0x You missed #53 and #104

P.S. I'm not a fan of pulling down other reports but if a selected portion of reports are being disqualified because they didn't mention a "detailed" attack path for an obvious issue, then every report that did not include a detailed description/PoC should be considered for disqualification.

novaman33

@Evert0x could you please identify how #10 fails to explain the attack vector. I believe the attack path is stated clearly and that the solution is also very detailed.

Hash01011122

Agreed, @0x3agle we can add those issues in our list. Updated issues to get excluded will be: #6, #10, #34, #53, #66, #79, #90, #104, #132, #138, #149

Hash01011122

@novaman33 I don't see any valid attack path mentioned in #10 report.

Evert0x

I believe #10 identified the attack pack and shows a good understanding of the issue.

After taken a detailed look at all reports, I believe only the following ones should be excluded as all other reports pinpoint the exact logic in the code that allows the reentrancy to happen.

https://github.com/sherlock-audit/2024-03-zap-protocol-judging/issues/6, https://github.com/sherlock-audit/2024-03-zap-protocol-judging/issues/34, https://github.com/sherlock-audit/2024-03-zap-protocol-judging/issues/66, https://github.com/sherlock-audit/2024-03-zap-protocol-judging/issues/79, https://github.com/sherlock-audit/2024-03-zap-protocol-judging/issues/90, https://github.com/sherlock-audit/2024-03-zap-protocol-judging/issues/132, https://github.com/sherlock-audit/2024-03-zap-protocol-judging/issues/149

Evert0x

Result: High Has Duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- midori-fuse: accepted

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-1: Vesting contract cannot work with ETH, although it's supposed to.

Source: https://github.com/sherlock-audit/2024-03-zap-protocol-judging/issues/54

Found by

0x4non, AMOW, GatewayGuardians, HonorLt, NickV, ZdravkoHr., bughuntoor, cats, enfrasico, klaus, merlin, s1ce, thank_you, ydlee

Summary

Vesting contract cannot work with native token, although it's supposed to.

Vulnerability Detail

Within the claim function, we can see that if `token` is set to address(1), the contract should operate with ETH

```
function claim() external {
    address sender = msg.sender;

    UserDetails storage s = userdetails[sender];
    require(s.userDeposit != 0, "No Deposit");
    require(s.index != vestingPoints.length, "already claimed");
    uint256 pctAmount;
    uint256 i = s.index;
    for (i; i <= vestingPoints.length - 1; i++) {
        if (block.timestamp >= vestingPoints[i][0]) {
            pctAmount += (s.userDeposit * vestingPoints[i][1]) / 10000;
        } else {
            break;
        }
    }
    if (pctAmount != 0) {
        if (address(token) == address(1)) {
            (bool sent, ) = payable(sender).call{value: pctAmount}("");  //
↪   @audit - here
            require(sent, "Failed to send BNB to receiver");
        } else {
            token.safeTransfer(sender, pctAmount);
        }
        s.index = uint128(i);
        s.amountClaimed += pctAmount;
```

```
    }
}
```

However, it is actually impossible for the contract to operate with ETH, since `updateUserDeposit` always attempts to do a token transfer.

```
function updateUserDeposit(
    address[] memory _users,
    uint256[] memory _amount
) public onlyRole(DEFAULT_ADMIN_ROLE) {
    require(_users.length <= 250, "array length should be less than 250");
    require(_users.length == _amount.length, "array length should match");
    uint256 amount;
    for (uint256 i = 0; i < _users.length; i++) {
        userdetails[_users[i]].userDeposit = _amount[i];
        amount += _amount[i];
    }
    token.safeTransferFrom(distributionWallet, address(this), amount); // @audit
↪   - this will revert
}
```

Since when the contract is supposed to work with ETH, token is set to address(1), calling `safeTransferFrom` on that address will always revert, thus making it impossible to call this function.

Impact

Vesting contract is unusable with ETH

Code Snippet

https://github.com/sherlock-audit/2024-03-zap-protocol/blob/main/zap-contracts-labs/contracts/Vesting.sol#L64

Tool used

Manual Review

Recommendation

make the following check

```
if (address(token) != address(1)) token.safeTransferFrom(distributionWallet,
↪   address(this), amount);
```

Discussion

shubham-antier

Removed ETH functionality from the contract.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Lithium-Ventures/zap-contracts-labs/pull/5

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-2: Blocklisted investors can still claim USDC in `TokenSale.sol`

Source: https://github.com/sherlock-audit/2024-03-zap-protocol-judging/issues/82

Found by

AMOW, Silvermist, ZdravkoHr., audithare, s1ce, ydlee

Summary

A wrong argument is passed when checking if a user is blacklisted for claiming in TokenSale.claim(). Because the check is insufficient, blocked users can claim their USDC.

Vulnerability Detail

Admin.setClaimBlock() blocks users from claiming. The function accepts the address of the user to be blocked and adds it to the `blockClaim` mapping.

```
/**
 @dev Whitelist users
 @param _address Address of User
 */
function setClaimBlock(address _address) external onlyRole(OPERATOR) {
    blockClaim[_address] = true;
}
```

The check in `Admin.claim()` wrongly passes `address(this)` as argument when calling `Admin.blockClaim`.

```
require(
    uint8(epoch) > 1 && !admin.blockClaim(address(this)),
    "TokenSale: Not time or not allowed"
);
```

In this context, `address(this)` will be the address of the token sale contract and the require statement can be bypassed even by a blocked user.

Impact

The whole functionality for blocking claims doesn't work properly.

Code Snippet

```
function claim() external {
    checkingEpoch();
    require(
        uint8(epoch) > 1 && !admin.blockClaim(address(this)),
        "TokenSale: Not time or not allowed"
    );

    Staked storage s = stakes[msg.sender];
    require(s.amount != 0, "TokenSale: No Deposit");
    require(!s.claimed, "TokenSale: Already Claimed");

    uint256 left;
    (s.share, left) = _claim(s);
    require(left > 0, "TokenSale: Nothing to claim");
    uint256 refundTaxAmount;
    if (s.taxAmount > 0) {
        uint256 tax = userTaxRate(s.amount, msg.sender);
        uint256 taxFreeAllc = _maxTaxfreeAllocation(msg.sender) * PCT_BASE;
        if (taxFreeAllc >= s.share) {
            refundTaxAmount = s.taxAmount;
        } else {
            refundTaxAmount = (left * tax) / POINT_BASE;
        }
        usdc.safeTransferFrom(marketingWallet, msg.sender, refundTaxAmount);
    }
    s.claimed = true;
    usdc.safeTransfer(msg.sender, left);
    emit Claim(msg.sender, left);
}
```

Tool used

Manual Review

Recommendation

Pass the address of the user.

```
        require(
-           uint8(epoch) > 1 && !admin.blockClaim(address(this)),
+           uint8(epoch) > 1 && !admin.blockClaim(msg.sender)),
            "TokenSale: Not time or not allowed"
        );
```

Discussion

0502lian

BlockClaim function is used for instance(block the whole tokeSale), not for one user.

Coareal

Escalate

Issue is invalid. Implementation is correct and intended. The mentioned check is used to block a specific tokenSale instance, and not that of a user.

sherlock-admin2

> Escalate
>
> Issue is invalid. Implementation is correct and intended. The mentioned check is used to block a specific tokenSale instance, and not that of a user.

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.

s1ce

Issue should be valid. Comments in the code seem to suggest that this is on a per user basis.

ZdravkoHr

Agree with @s1ce. According to the Sherlock hierarchy of truth `protocol documentation (including code comments) > protocol answers on the contest public Discord channel.`

omar-ahsan

Issue should be invalid, the only issue here is that the comments are wrong.

Hash01011122

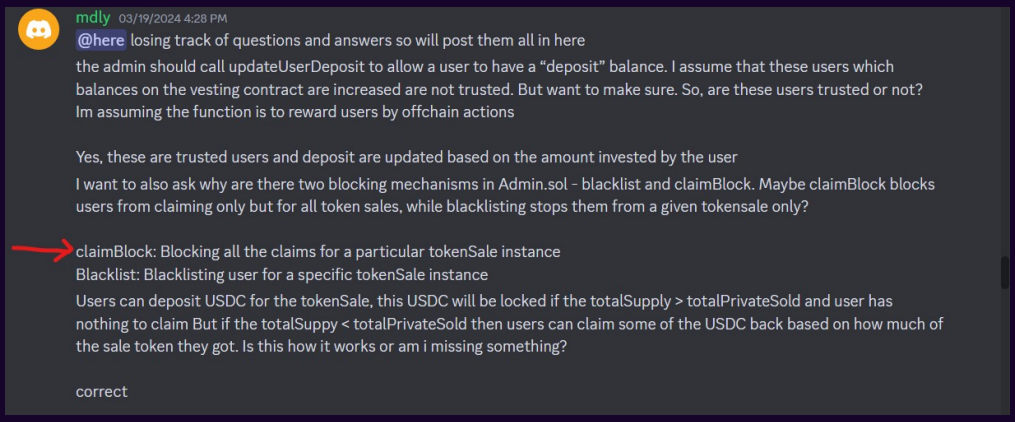
Can you give any reason on why this issue should be invalidated?? @Coareal @omar-ahsan

omar-ahsan

@Hash01011122

```
/**
 @dev Whitelist users
 @param _address Address of User
 */
function setClaimBlock(address _address) external onlyRole(OPERATOR) {
    blockClaim[_address] = true;
}
```

The comments above the function indicate whitelisting of users but this function is not intended to whitelist any address. `setClaimBlock()` as the name suggests is used to block an address by setting it to true in `blockClaim`. Similarly



The comments by sponsor team indicate that this function is used to block all incoming claims for a particular Token Sale which means all users can not claim from the token sale during the blocked duration. Currently the function does as intended according to this description.

Further more the code already contains a function to blacklist users i.e addToBlackList(). This function performs the blocking of single users by blocking the deposit() function for blacklisted users which is the entry point to the token sale.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Lithium-Ventures/zap-contracts-labs/pull/3

Hash01011122

Glad @omar-ahsan you did point out this mistake. My question to you is were watsons aware of this at the time of contest if not this is a valid finding.

omar-ahsan

> Glad @omar-ahsan you did point out this mistake. My question to you is were watsons aware of this at the time of contest if not this is a valid finding.

Since the comments are misleading and not correct, the next source of information was the dev in the public chat. The screenshot of the message in my previous reply is from the public chat hence everyone knew this information.

Evert0x

Planning to accept escalation and invalidate issue

ZdravkoHr

@Evert0x, so the Sherlock documentation should be updated to discord > comments?

Hash01011122

@Evert0x I think this is a valid issue, as Watson's were not aware of it before or at the time of the contest. @omar-ahsan I understand that you had conversation with sponsors and they responded you in discord but not every watson was aware of it. I would like @Evert0x to reconsider his decision.

Evert0x

@Hash01011122 do you know if there was any other language for this function in the code or docs?

Hash01011122

As far as I know there were no mentions about this in docs or in codebase

detectiveking123

@Evert0x @Czar102

There are a lot, and I mean a lot, of issues with the documentation in this codebase. This has left a lot of ambiguity in terms of what the sponsors actually want vs the design decisions they've consciously made.

I would recommend only rewarding issues that actually cause a loss of funds or very, very clearly break protocol functionality (i.e. it's just definitely not a design decision). This issue, as well as #87 and #56, fall into the category of "maybe the sponsors intended this, maybe they didn't", and I don't think any of them should be valid issues.

Nilay27

> @Evert0x @Czar102
>
> There are a lot, and I mean a lot, of issues with the documentation in this codebase. This has left a lot of ambiguity in terms of what the sponsors actually want vs the design decisions they've consciously made.
>
> I would recommend only rewarding issues that actually cause a loss of funds or very, very clearly break protocol functionality (i.e. it's just

> definitely not a design decision). This issue, as well as #87 and #56, fall into the category of "maybe the sponsors intended this, maybe they didn't", and I don't think any of them should be valid issues.

@detectiveking123, I understand your concerns about potentially overreporting issues that may be interpreted as design decisions rather than genuine flaws.

However, as a Watson, we rely heavily on the documentation provided to guide our auditing process. When the documentation is unclear and the sponsors are not available for clarification, we must address potential vulnerabilities based on our best understanding of the intended functionality.

Considering the nature of a competitive audit, dismissing ambiguities that arise from unclear documentation could inadvertently overlook genuine issues and waste a lot of Watsons' time due to a lack of due diligence before the audit.

@Evert0x @Czar102 @Hash01011122, I would like to request that we only finalize these issues once after the sponsors' confirmation. While this might be a bit of a hassle, but this would be the fairest approach.

Hash01011122

I stand by what I mentioned earlier that this should remain a valid issue. @Evert0x @Czar102

Evert0x

With the hierarchy of truth at the time of the contest I believe the right judgment is to reject the escalation and keep the issue valid.

detectiveking123

@Evert0x Judgement doesn't make sense. By that logic, there are so many other issues in this contest that should be valid, just because the documentation is completely wrong.

Hash01011122

@detectiveking123 I've already justified the validity of this issue above. If you can provide a counterargument using any rule from Sherlock's documentation, please do so. If not please refrain to comment on this issue.

Evert0x

Result: Medium Has Duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- Coareal: rejected

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-3: Max allocations can be bypassed with multiple addresses because of guaranteed allocations

Source: https://github.com/sherlock-audit/2024-03-zap-protocol-judging/issues/152

Found by

GatewayGuardians, Silvermist, ZdravkoHr., s1ce

Summary

TokenSale._processPrivate() ensures that a user cannot deposit more than their allocation amount. However, each address can deposit up to at least `maxAllocations`. This can be leveraged by a malicious user by using different addresses to claim all tokens without even staking.

Vulnerability Detail

The idea of the protocol is to give everyone the right to have at least `maxAlocations` allocations. By completing missions, users level up and unlock new tiers. This process will be increasing their allocations. The problem is that when a user has no allocations, they have still a granted amount of `maxAllocations`.

TokenSale.calculateMaxAllocation returns $max(maxTierAlloc(), maxAllocation)$

For a user with no allocations, `_maxTierAlloc()` will return 0. The final result will be that this user have `maxAllocation` allocations (because maxAllocation > 0).

```
if (userTier == 0 && giftedTierAllc == 0) {
    return 0;
}
```

Multiple Ethereum accounts can be used by the same party to take control over the IDO and all its allocations, on top of that without even staking.

NOTE: setting `maxAllocation = 0` is not a solution in this case because the protocol wants to still give some allocations to their users.

Impact

Buying all allocations without staking. This also violates a key property that only ION holders can deposit.

Code Snippet

```
function calculateMaxAllocation(address _sender) public returns (uint256) {
    uint256 userMaxAllc = _maxTierAllc(_sender);

    if (userMaxAllc > maxAllocation) {
        return userMaxAllc;
    } else {
        return maxAllocation;
    }
}
```

Tool used

Manual Review

Recommendation

A possible solution may be to modify `calculateMaxAllocation` in the following way:

```
    function calculateMaxAllocation(address _sender) public returns (uint256) {
        uint256 userMaxAllc = _maxTierAllc(_sender);
+       if (userMaxAllc == 0) return 0;

         if (userMaxAllc > maxAllocation) {
            return userMaxAllc;
        } else {
            return maxAllocation;
        }
    }
```

Discussion

ZdravkoHr

Escalate

@Hash01011122, why was this issue excluded? It shows how users can bypass a core restriction and on top of that - doing it without staking.

sherlock-admin2

> Escalate
>
> @Hash01011122, why was this issue excluded? It shows how users can bypass a core restriction and on top of that - doing it without staking.

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.

Hash01011122

The function is operating as designed. If you believe the issue warrants further investigation, please submit a Proof of Concept (PoC).

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Lithium-Ventures/zap-contracts-labs/pull/1

ZdravkoHr

@Hash01011122 basically everyone, regardless of the staked amount, can deposit up to maxAllocations

Hash01011122

If you don't provide a valid PoC for this issue within 48 hours, I will consider this as invalid

ZdravkoHr

@Hash01011122, here is my PoC. You have to do two things before running it:

1. Add this `MockStaking.sol` file in the same folder as the PoC.

```
contract MockStaking {
    function getUserState(address) public returns(uint256, uint256, uint256,
↪   uint256) {
        return (0, 0, 0, 0);
    }
}
```

2. Add this function to `TokenSale.sol`.

```
function setUsdc(address _new) public {
    usdc = IERC20D(_new);
}
```

The main file `TokenSale.t.sol` demonstrates how a user that hasn't staked in the staking contract, can use different accounts to deposit as many assets as they want to.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.11;

import "forge-std/Test.sol";
import "../contracts/Admin.sol";
import "../contracts/USDC.sol";
import "../contracts/interfaces/ITokenSale.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "forge-std/console.sol";
import {TokenSale} from "../contracts/TokenSale.sol";
import {MockStaking} from "./MockStaking.sol";


contract TokenSaleTest is Test {
    using stdStorage for StdStorage;
     address public saleContract;
     Admin public admin;
     USDCWithSixDecimal public mockUSDC;
     MockStaking public stakingContract;
     TokenSale public saleInstance;



    function setUp() public {
        saleContract = address(new TokenSale());
        mockUSDC = new USDCWithSixDecimal("MockUSDC", "USDC");
        _config();
    }

    function _config() internal {
       stakingContract = new MockStaking();
       admin = new Admin();
       admin.initialize(address(this));
       admin.setStakingContract(address(stakingContract));
       admin.addOperator(address(this));
       admin.setMasterContract(saleContract);
       admin.setWallet(address(this));


       ITokenSale.Params memory params = ITokenSale.Params({
        totalSupply: 1000e18,
        privateStart: uint32(block.timestamp),
        privateTokenPrice: 20e18,
        privateEnd: uint32(block.timestamp + 1 weeks)
       });
```

```
        admin.createPoolNew(params, 15000e18, 1000, false, 0);

        assertTrue(admin.tokenSalesM(0x5B0091f49210e7B2A57B03dfE1AB9D08289d9294));
        saleInstance =
↪   TokenSale(payable(0x5B0091f49210e7B2A57B03dfE1AB9D08289d9294));

        // A setUsdc function is added to the TokenSale contract for test purposes
        saleInstance.setUsdc(address(mockUSDC));
    }

    function testDeposit() public {
        uint160 addressCount = 20;
        // A single user, WITHOUT EVER DEPOSITING TO THE STAKING CONTRACT, can
↪   deposit as much as they want using different addresses.
            for (uint160 i = 1; i < addressCount; i++) {
            address currentAddress = address(i);
            mockUSDC.mint(currentAddress, type(uint256).max / addressCount);

            vm.startPrank(currentAddress);
            mockUSDC.approve(address(saleInstance), type(uint256).max);
            // Just 100 because of decimals mess
            saleInstance.deposit(100);
            vm.stopPrank();

            (uint128 totalPrivateSold, ) = saleInstance.getState();
            console.log(totalPrivateSold);
        }

    }

}
```

s1ce

@Evert0x @Hash01011122

I am not sure if this issue is valid which is why I didn't escalate my own, but mine (#196) is a duplicate of this

My issue #172 that has been a marked a dup of this one is actually a dup of #158 instead

Evert0x

@Hash01011122 what's your opinion about the poc?

Hash01011122

@ZdravkoHr your PoC isn't working, @Evert0x this issue can be invalidated. Also their is a check in TokenSale.sol:_maxTierAllc whether if `userTier` or `giftedTierAllc` of user is zero then allocation will be zero. @ZdravkoHr do you want to add anything?? Correct me if I am wrong.

ZdravkoHr

The 0 will be used in calculateMaxAllocation and maxAllocations will be returned. Which part of the PoC is not working?

Hash01011122

The `deposit` function in your PoC isn't working, if you still feel it's valid can you please provide the exact lines of code which justifies the logic you are trying to portray with proper mitigation for it.

ZdravkoHr

Have you added the MockStaking contract and the setUsdc function?

detectiveking123

@Evert0x I believe this one and it's duplicates are actually the same issue family as #158 (they point out the same root cause issue)

You have invalidated issue family #158, but in the documentation in `TokenSale.sol` there are the following two lines:

```
1. Private round. Only ion token holders can participate in this round.

 A single investor can purchase up to their maximum allowed investment
defined by the tier.
```

Clearly, based on this documentation (which I strongly believe is incorrect and should not be used as a source of truth, but my personal opinions are besides the point), both issues should be valid.

Hash01011122

@ZdravkoHr I've added the setUsdc function and MockStaking contract, and also deleted the entire file and environment. I reinstalled everything from scratch to rerun your PoC, but the results remain unchanged.

Hash01011122

@detectiveking123 Still verifying this issue's validity, whereas for issue #158, I will take confirmation from sponsors.

detectiveking123

@Hash01011122 They are the exact same root cause though, so they should be valid and duplicates of each other. What the sponsors say shouldn't matter; I've shown you excerpts from the documentation that prove validity.

Evert0x

Agree with @detectiveking123

For that reason I will reject the escalation and invalidate the issue

detectiveking123

@Evert0x I'm saying the issue should be valid. They're all valid issues, and duplicates of each other, because of the above excerpts from the documentation.

Evert0x

The language you quoted supports the invalidation of this issue.

> A single investor can purchase up to **their** maximum allowed investment defined by the tier.

Indicating it's a maximum allowed investment personalized to the investor.

Besides that, they have clarified it in a discord message https://github.com/sherlock-audit/2024-03-zap-protocol-judging/issues/158#issuecomment-2047444809

I will reject the escalation and invalidate the issue

ZdravkoHr

@Evert0x, this issue shows how tokens can be obtained without any prior staking. I think the root cause here is the missing if statement from my recommendation

detectiveking123

@Evert0x You misunderstand. The tier based max allocation is defined by `_maxTierAllc`. The current code would allow them to purchase potentially more than that if maxAllocation is higher.

Furthermore, the language here is pretty clear:

```
1. Private round. Only ion token holders can participate in this round.
```

Clearly, this code allows non-ion token holders/stakers to participate.

detectiveking123

Also, to clarify my point, I am suggesting that it is not just this issue that is valid based on the documentation, but also the entire #158 family (they have the exact same root cause). This issue is a strict subset of #158 , although still valid.

My recommendation is to duplicate the two issue families and validate all of them.

Evert0x

Thank you for correcting me @detectiveking123. I indeed misunderstood this issue.

Still looking into the outcome for this issue. But I don't believe this is a duplicate of #158.

detectiveking123

@Evert0x This is actually a strict subset of #158 .

Both issues reference the exact code snippet and lead to the exact same impact, but have different opinions about what the correct fix is. In #158 , the fix that is suggested is to replace the max with a min, while in this issue adding an extra if statement is suggested.

In fact, #158 even states that: `However, swapped return values allow a user to have 0 allocations and get themaxAllocation or to exceed the maximum allocations`

Clearly all Watsons have identified the core issue but have different suggestions / opinions on how to fix it

Evert0x

I believe the following issues are a family #152, #158, #161, #172.

They all describe the following issue

> get maximum allocation while having 0

Planning to make #152 the main report issue as the fix recommendation is correct

Thanks @detectiveking123

detectiveking123

@Evert0x No problem. Your proposed judgement makes sense to me.

Hash01011122

Thanks for bring this to light @detectiveking123, I agree that this should be considered a valid issue with dups #019, #158, #161, #172.

Evert0x

Result: Medium Has Duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- ZdravkoHr: accepted

sherlock-admin2

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.