SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Dinari
Prepared by: Sherlock
Lead Security Expert:
Dates Audited: July 3 - July 6, 2023
Prepared on: August 11, 2023

Introduction

Dinari securities backed tokens (dShares) provide direct exposure to the world's most trusted assets such as Google and Apple shares.

Scope

Repository: dinaricrypto/sbt-contracts

Branch: main

Commit: 6d36760def25449c3f35f6ed38128a7eaf352903

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	1

Issues not fixed or acknowledged

Medium	High
1	0

Security experts who found valid issues

volodya
ArmedGoose
dirk_y
Kodyvim
ast3ros
p-tsanev
ctf_sec
hals
toshii
chainNue
0x007
bin2chen

Delvir0
osmanozdemir1

shtesesamoubiq
bitsurfer

james_wu

Issue H-1: Bypass the blacklist restriction because the blacklist check is not done when minting or burning

Source: https://github.com/sherlock-audit/2023-06-dinari-judging/issues/64

Found by

ctf_sec, dirk_y, p-tsanev, toshii

Summary

Bypass the blacklist restriction because the blacklist check is not done when minting or burning

Vulnerability Detail

In the whitepaper:

> the protocol emphasis that they implement a blacklist feature for enforcing OFAC, AML and other account security requirements A blacklisted will not able to send or receive tokens

the protocol want to use the whitelist feature to be compliant to not let the blacklisted address send or receive dSahres

For this reason, before token transfer, the protocol check if address from or address to is blacklisted and the blacklisted address can still create buy order or sell order

```
function _beforeTokenTransfer(address from, address to, uint256) internal
↪   virtual override {
    // Restrictions ignored for minting and burning
    // If transferRestrictor is not set, no restrictions are applied

    // @audit
    // why don't you not apply mint and burn in blacklist?
    if (from == address(0) || to == address(0) || address(transferRestrictor)
↪   == address(0)) {
        return;
    }

    // Check transfer restrictions
    transferRestrictor.requireNotRestricted(from, to);
}
```

this is calling

```
function requireNotRestricted(address from, address to) external view virtual {
    // Check if either account is restricted
    if (blacklist[from] || blacklist[to]) {
        revert AccountRestricted();
    }
    // Otherwise, do nothing
}
```

but as we can see, when the dShare token is burned or minted, the blacklist does not apply to address(to)

this allows the blacklisted receiver to bypass the blacklist restriction and still send and receive dShares and cash out their dShares

because the minting dShares is not blacklisted

a blacklisted user create a buy order with payment token and set the order receiver to a non-blacklisted address

then later when the buy order is filled, the new dShares is transferred and minted to an not-blacklisted address

because the burning dShares is not blacklisted

before the user is blacklisted, a user can frontrun the blacklist transaction to create a sell order and transfer the dShares into the OrderProcessor

then later when the sell order is filled, the dShares in burnt from the SellOrderProcess escrow are burnt and the user can receive the payment token

Impact

Bypass the blacklist restriction because the blacklist check is not done when minting or burning

Code Snippet

https://github.com/sherlock-audit/2023-06-dinari/blob/4851cb7ebc86a7bc26b8d0d399a7dd7f9520f393/sbt-contracts/src/issuer/SellOrderProcessor.sol#L88

https://github.com/sherlock-audit/2023-06-dinari/blob/4851cb7ebc86a7bc26b8d0d399a7dd7f9520f393/sbt-contracts/src/issuer/SellOrderProcessor.sol#L115

Tool used

Manual Review

Recommendation

implement proper check when burning and minting of the dShares to not let user game the blacklist system, checking if the receiver of the dShares is blacklisted when minting, before filling sell order and burn the dShares, check if the requestor of the sell order is blacklisted

do not let blacklisted address create buy order and sell order

Discussion

jaketimothy

Fixes for this should be considered in combination with #55 as it creates more opportunities for locking up orders.

jaketimothy

Fixed in

- https://github.com/dinaricrypto/sbt-contracts/pull/126
- https://github.com/dinaricrypto/sbt-contracts/pull/131

ctf-sec

PR #131 fix goods good

PR #126 only fix only check if the recipient or requestor is blocklisted by the asset token transferRestrictor

```
if (

↪  BridgedERC20(orderRequest.assetToken).isBlacklisted(orderRequest.recipient)
            || BridgedERC20(orderRequest.assetToken).isBlacklisted(msg.sender)
     ) revert Blacklist();
```

the protocol may want to consider checking if the orderRequest.recipient or msg.sender is blocklisted by the payment token as well

jaketimothy

Additional blacklist checks in

- https://github.com/dinaricrypto/sbt-contracts/pull/164

ctf-sec

Fix looks good!

Issue M-1: In case of stock split and reverse split, the Dshare token holder will gain or loss his Dshare token value

Source: https://github.com/sherlock-audit/2023-06-dinari-judging/issues/29

Found by

ast3ros

Summary

Stock split and reverse split may cause the token accounting to be inaccurate.

Vulnerability Detail

Stock split and reverse split are very common in the stock market. There are many examples here: https://companiesmarketcap.com/amazon/stock-splits/

For instance, in a 2-for-1 stock split, a shareholder receives an additional share for each share held. However, the DShare token holder still holds only one DShare token after the split. If a DShare token holder owns 100 DShare tokens before the split, he will still own 100 DShare tokens after the split. However, he should own 200 DShare tokens after the split.

Currently, users can buy 1 DShare token at the current market price of the underlying share. https://sbt.dinari.com/tokens

This means that after the stock split, a new Dshare token holder can buy a Dshare at half the price of the previous Dshare token holder. This is unfair to the previous Dshare token holder. In other words, the original Dshare token holder will lose 50% of his Dshare token value after the stock split.

The same logic applies to stock reverse split.

Impact

The Dshare token holder will gain or loss his Dshare token value after the stock split or reverse split.

Code Snippet

https://github.com/sherlock-audit/2023-06-dinari/blob/50eb49260ab54e02748c2f6382fd95284d271f06/sbt-contracts/src/BridgedERC20.sol#L13

Tool used

Manual Review

Recommendation

The Operator should have a mechanism to mint or burn DShare tokens of holders when the underlying share is split or reverse split.

Discussion

jaketimothy

While this would be catastrophic if unaddressed, our current - admittedly undocumented - approach is to halt and cancel all orders for assets at the end of the day the split is announced. Then deploy a new token with a burn->mint migration contract at the appropriate split ratio, add the new token to the issuers, and re-enable orders with the new token.

This accounts for various scenarios where the original tokens may be locked in other contracts without Dinari having to push airdrop transactions for every split - or maintaining burn privileges on all owners/holders.

ctf-sec

The amazon last stock split is 20 years ago. I think this report is out of scope

> The Operator should have a mechanism to mint or burn DShare tokens of holders when the underlying share is split or reverse split.

The operator is a privileged central party that can do that.

Oot2k

I still think this issue is valid. Stock splits happen fairly often, for example Tesla in 2020 and 2022. I think this is a valid medium, because the offchain system was not mentioned anywhere and there still can be issues when deploying a token manually. There wont be direct loss of funds, but protocols that integrate with dinari will have problems (Dex, lending etc.)

ctf-sec

While the stock and offline logic is out of scope,

> In 2-for-1 stock split, a shareholder receives an additional share for each share held

In the current implementation, this would require the admin mint stock for user

In a stock reverse split,

In the current implementation, admin can't burn stock for user

```
function burn(uint256 value) external virtual onlyRole(BURNER_ROLE) {
      _burn(msg.sender, value);
  }
```

if the implementation is

```
function burn(address from, uint256 value) external virtual
↪   onlyRole(BURNER_ROLE) {
      _burn(from, value);
  }
```

I would agree this is a low severity and out of scope finding,

but since the admin can't actually burn for user,

this can be valid medium, severity is definitely not high :)

thangtranth

Escalate

I believe this issue is a high severity because:

- Impact: As the sponsor mentioned, the impact is catastrophic if unaddressed. It breaks the invariant 1 share : 1 token of the protocol. Some token holders will unfairly lose/gain X times the value. In addition, protocols that integrate with Dinari will have problems. It is not considered an acceptable risk
- Frequency: Stock splits and reverse splits are very common events. Because Dinari covers many publicly traded securities, not just one stock, the frequency of the events should be counted using the whole stock market, not frequency of one stock. There are events happening every month. https://www.marketbeat.com/stock-splits/history/
- The mechanism that the sponsor mentioned was not available anywhere during the contest: from the contest README to the white paper. It is new information that came after the contest was finished.
- As the lead Watson pointed out, even the workaround of owner minting and burning directly to users does not work, because the admin cannot burn other users' tokens. And I don't think using the mint and burn functions in the token contract and minting manually to each user is a good solution.

sherlock-admin

> Escalate
>
> I believe this issue is a high severity because:

- Impact: As the sponsor mentioned, the impact is catastrophic if unaddressed. It breaks the invariant 1 share : 1 token of the protocol. Some token holders will unfairly lose/gain X times the value. In addition, protocols that integrate with Dinari will have problems. It is not considered an acceptable risk
- Frequency: Stock splits and reverse splits are very common events. Because Dinari covers many publicly traded securities, not just one stock, the frequency of the events should be counted using the whole stock market, not frequency of one stock. There are events happening every month.
  https://www.marketbeat.com/stock-splits/history/
- The mechanism that the sponsor mentioned was not available anywhere during the contest: from the contest README to the white paper. It is new information that came after the contest was finished.
- As the lead Watson pointed out, even the workaround of owner minting and burning directly to users does not work, because the admin cannot burn other users' tokens. And I don't think using the mint and burn functions in the token contract and minting manually to each user is a good solution.

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.

ctf-sec

> Escalate
>
> I believe this issue is a high severity because:
>
> - Impact: As the sponsor mentioned, the impact is catastrophic if unaddressed. It breaks the invariant 1 share : 1 token of the protocol. Some token holders will unfairly lose/gain X times the value. In addition, protocols that integrate with Dinari will have problems. It is not considered an acceptable risk
> - Frequency: Stock splits and reverse splits are very common events. Because Dinari covers many publicly traded securities, not just one stock, the frequency of the events should be counted using the whole stock market, not frequency of one stock. There are events happening every month.
>   https://www.marketbeat.com/stock-splits/history/
> - The mechanism that the sponsor mentioned was not available anywhere during the contest: from the contest README to the white

paper. It is new information that came after the contest was finished.

- As the lead Watson pointed out, even the workaround of owner minting and burning directly to users does not work, because the admin cannot burn other users' tokens. And I don't think using the mint and burn functions in the token contract and minting manually to each user is a good solution.

I find it difficult to think this is a high severity issue, stock just don't split every day. How does the sponsor handle the stock split is not in scope of auditing, severity at most medium if not out of scope, we could say, if the stock company rug and delisted, all dShare lose its value... this is expected, which is not in scope of the auditing as well

https://docs.sherlock.xyz/audits/judging/judging

> Medium: There is a viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost. The attack path is possible with assumptions that either mimic on-chain conditions or reflect conditions that have a reasonable chance of becoming true in the future.

and

> As the lead Watson pointed out, even the workaround of owner minting and burning directly to users does not work, because the admin cannot burn other users' tokens. And I don't think using the mint and burn functions in the token contract and minting manually to each user is a good solution

there is no mention about this in the original report

and

I really feel like not dispute as low and out of scope and argue a medium is already what I can do the best for this report because this report does show creativity.

https://www.marketplace.org/2022/02/11/whats-a-stock-split-anyway/

> While stock splits were once a common occurrence, they have become rare in recent years. In the S&P 500, a broader index than the Dow, only two companies — Tesla and Apple — split their stock in the last few years, compared to almost 50 splits among S&P companies in 2006 and 2007.
>
> One reason for this decline is the rise of technologies that allow investors to trade fractions of a single share.
>
> "Because of fractional trading, the price of your share doesn't really matter for people. They can buy half a share if they want," Rawley said. "I think companies are just less interested in doing stock splits because it's a hassle, it does cost something and it has no real economic value."

ctf-sec

btw the while the resolving the escalation is one story, the fix is another.

@jaketimothy

Maybe can use this

https://docs.openzeppelin.com/contracts/3.x/api/token/erc20#ERC20Snapshot

Implement as rebasing token to modify the ERC20 token balance can help as well!

Agree this is not a easy fix.

jaketimothy

Thank you for sharing @ctf-sec. I'm racking my brain trying to avoid rebasing token.

Oot2k

Stocksplits are known well in advance, so trading can be haltet and shares converted. I still think all offchain fixes are not recommended which means this is a valid medium issue.

jaketimothy

Fixed in

- https://github.com/dinaricrypto/sbt-contracts/pull/135

hrishibhat

Result: Medium Unique Given that this is an external condition that is well-known beforehand. This issue can be fairly considered a valid medium because the code cannot handle the stock-split situation and the off-chain/on-chain solutions were not previously present.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- thangtranth: rejected

Issue M-2: Escrow record not cleared on cancellation and order fill

Source: https://github.com/sherlock-audit/2023-06-dinari-judging/issues/56

Found by

Delvir0, bin2chen, bitsurfer, chainNue, ctf_sec, dirk_y, hals, volodya

Summary

In `DirectBuyIssuer.sol`, a market buy requires the operator to take the payment token as escrow prior to filling the order. Checks are in place so that the math works out in terms of how much escrow has been taken vs the order's remaining fill amount. However, if the user cancels the order or fill the order, the escrow record is not cleared.

The escrow record will exists as a positive amount which can lead to accounting issues.

Vulnerability Detail

Take the following example:

- Operator broadcasts a `takeEscrow()` transaction around the same time that the user calls `requestCancel()` for the order
- Operator also broadcasts a `cancelOrder()` transaction
- If the `cancelOrder()` transaction is mined before the `takeEscrow()` transaction, then the contract will transfer out token when it should not be able to.

`takeEscrow()` simply checks that the `getOrderEscrow[orderId]` is less than or equal to the requested amount:

```
bytes32 orderId = getOrderIdFromOrderRequest(orderRequest, salt);
uint256 escrow = getOrderEscrow[orderId];
if (amount > escrow) revert AmountTooLarge();


// Update escrow tracking
getOrderEscrow[orderId] = escrow - amount;
// Notify escrow taken
emit EscrowTaken(orderId, orderRequest.recipient, amount);
```

```
// Take escrowed payment
IERC20(orderRequest.paymentToken).safeTransfer(msg.sender, amount);
```

Cancelling the order does not clear the `getOrderEscrow` record:

```
    function _cancelOrderAccounting(OrderRequest calldata order, bytes32
↪   orderId, OrderState memory orderState)
        internal
        virtual
        override
    {
        // Prohibit cancel if escrowed payment has been taken and not returned
↪   or filled
        uint256 escrow = getOrderEscrow[orderId];
        if (orderState.remainingOrder != escrow) revert UnreturnedEscrow();


        // Standard buy order accounting
        super._cancelOrderAccounting(order, orderId, orderState);
    }
}
```

This can lead to an good-faith and trusted operator accidentally taking funds from the contract that should not be able to leave.

coming up with the fact that the transaction does not have deadline or expiration date:

consider the case below:

1. a good-faith operator send a transaction, takeEscrow
2. the transaction is pending in the mempool for a long long long time
3. then user fire a cancel order request
4. the operator help user cancel the order
5. the operator send a transcation cancel order
6. cancel order transaction land first
7. the takeEscrow transaction lands

because escrow state is not clear up, the fund (other user's fund) is taken

It's also worth noting that the operator would not be able to call `returnEscrow()` because the order state has already been cleared by the cancellation. `getRemainingOrder()` would return **0**.

```
function returnEscrow(OrderRequest calldata orderRequest, bytes32 salt, uint256
↪   amount)
    external
    onlyRole(OPERATOR_ROLE)
{
    // No nonsense
    if (amount == 0) revert ZeroValue();
    // Can only return unused amount
    bytes32 orderId = getOrderIdFromOrderRequest(orderRequest, salt);
    uint256 remainingOrder = getRemainingOrder(orderId);
    uint256 escrow = getOrderEscrow[orderId];
    // Unused amount = remaining order - remaining escrow
    if (escrow + amount > remainingOrder) revert AmountTooLarge();
```

Impact

- Insolvency due to pulling escrow that should not be allowed to be taken

Code Snippet

https://github.com/sherlock-audit/2023-06-dinari/blob/4851cb7ebc86a7bc26b8d0d399a7dd7f9520f393/sbt-contracts/src/issuer/DirectBuyIssuer.sol#L130-L142

Tool used

Manual Review

Recommendation

Clear the escrow record upon canceling the order.

Discussion

jaketimothy

Fixed in

- https://github.com/dinaricrypto/sbt-contracts/pull/122

ctf-sec

The fix reset the escrow balance after cancelling

the protocol may want to consider handling the escrow accounting properly when the order is filled as well (such as reset the escrow balance to 0)

jaketimothy

> The fix reset the escrow balance after cancelling
>
> the protocol may want to consider handling the escrow accounting properly when the order is filled as well (such as reset the escrow balance to 0)

Because the escrow must be taken before filling, the escrow balance will always be zero after total fulfillment.

ctf-sec

ok, then fix looks good

Issue M-3: Cancellation refunds should return tokens to order creator, not recipient

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

Found by

0x007, ArmedGoose, Kodyvim, ctf_sec, dirk_y, james_wu, osmanozdemir1, shtesesamoubiq

Summary

When an order is cancelled, the refund is sent to `order.recipient` instead of the order creator because it is the order creator (requestor) pay the payment token for buy order or pay the dShares for sell order

As is the standard in many L1/L2 bridges, cancelled deposits should be returned to the order creator instead of the recipient. In Dinari's current implementation, a refund acts as a transfer with a middle-man.

Vulnerability Detail

Simply, the `_cancelOrderAccounting()` function returns the refund to the `order.recipient`:

```
function _cancelOrderAccounting(OrderRequest calldata orderRequest, bytes32
↪   orderId, OrderState memory orderState)
    internal
    virtual
    override
{
    ...

    uint256 refund = orderState.remainingOrder +
↪   feeState.remainingPercentageFees;

    ...

    if (refund + feeState.feesEarned == orderRequest.quantityIn) {
        _closeOrder(orderId, orderRequest.paymentToken, 0);
        // Refund full payment
        refund = orderRequest.quantityIn;
    } else {
        // Otherwise close order and transfer fees
        _closeOrder(orderId, orderRequest.paymentToken, feeState.feesEarned);
    }
```

```
    // Return escrow
    IERC20(orderRequest.paymentToken).safeTransfer(orderRequest.recipient,
↪   refund);
}
```

Refunds should be returned to the order creator in cases where the input recipient was an incorrect address or simply the user changed their mind prior to the order being filled.

Impact

- Potential for irreversible loss of funds
- Inability to truly cancel order

Code Snippet

https://github.com/sherlock-audit/2023-06-dinari/blob/4851cb7ebc86a7bc26b8d0d399a7dd7f9520f393/sbt-contracts/src/issuer/BuyOrderIssuer.sol#L193-L215

Tool used

Manual Review

Recommendation

Return the funds to the order creator, not the recipient.

Discussion

bizzyvinci

Escalate

This issue ought to be high because it doesn't require external conditions or specific states for users to lose a significant amount. It is very likely that `msg.sender != recipient` cause msg.sender and recipient could be different entities trying to have a deal either EOA-EOA or contract-EOA or contract-contract. Canceling order is meant to be a **perfect unwind** as mentioned by the sponsor.

This issue affects every single cancelled order, both buy, direct buy and sell orders.

sherlock-admin

> Escalate

> This issue ought to be high because it doesn't require external conditions or specific states for users to lose a significant amount. It is very likely that `msg.sender != recipient` cause msg.sender and recipient could be different entities trying to have a deal either EOA-EOA or contract-EOA or contract-contract. Canceling order is meant to be a **perfect unwind** as mentioned by the sponsor.
>
> This issue affects every single cancelled order, both buy, direct buy and sell orders.

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.

jaketimothy

Fixed in

- https://github.com/dinaricrypto/sbt-contracts/pull/117

ctf-sec

Fix looks good