SHELLBOXES

CAP V4

Smart Contract Security Audit

Prepared by ShellBoxes

Feb 7th, 2023 - Feb 13th, 2023

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	Cap
Version	1.0
Classification	Public

Scope

Repository	Commit Hash
https://github.com/capofficial/contracts/tree/audit	6b7945b2a6f4c8db1d101700af1db275ed94fd56

Re-Audit

Repository	Commit Hash
https://github.com/beskay/cap-contracts	e41c4e5755171a370826ee3d1ac2d2a0f5041311

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

Cap engaged ShellBoxes to conduct a security assessment on the CAP V4 beginning on Feb 7th, 2023 and ending Feb 13th, 2023. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Cap

CAP is decentralized trading protocol designed to be powerful and easy to use.It allows you to trade crypto and forex perpetuals directly from your Web3 wallet, pool funds to make real yield ,and stake CAP, the protocol's native token.

Issuer	Cap
Website	https://cap.io
Type	Solidity Smart Contract
Documentation	https://docs.cap.io/intro/whats-cap
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

		Likelihood		
		High	Medium	Low
Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the CAP V4 implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include 1 critical-severity, 1 high-severity, 4 medium-severity, 7 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Missing expiry update in the TP and SL orders	CRITICAL	Fixed
SHB.2. Stakers may lose their rewards due to rounding errors	HIGH	Fixed
SHB.3. Fees can be bypassed	MEDIUM	Fixed
SHB.4. Keeper's native tokens can get locked	MEDIUM	Fixed
SHB.5. Excessive Privileges Granted to the Governance Account	MEDIUM	Acknowledged
SHB.6. Blocked Contract Features Due to Missing Link Function Call	MEDIUM	Acknowledged
SHB.7. Unchecked return value in granting roles	LOW	Fixed
SHB.8. Missing value verification	LOW	Fixed
SHB.9. Lack of Contract Verification for Granting the CONTRACT Role	LOW	Acknowledged
SHB.10. Lack of Two-Factor Verification for Updating gov Address	LOW	Acknowledged

SHB.11. Transaction Order Dependency & Potential Loss of Precision in Fee Calculation	LOW	Partially Fixed
SHB.12. Potential Reentrancy Attack	LOW	Fixed
SHB.13. Floating Pragma	LOW	Fixed

3 Finding Details

SHB.1 Missing expiry update in the TP and SL orders

- Severity : CRITICAL
- Status : Fixed

- Likelihood : 3
- Impact : 3

Description:

The submitOrder function is used to submit an order that will be executed later by the Keeper, the expiry attribute is the timestamp at which the order expires. The function accepts as arguments the tpPrice and the slPrice. If any of these arguments is different from zero, the contract creates a separate stop/limit reduce-only order depending on the arguments. However, to create these orders, the function makes use of the same parameters that were used to create the initial order without updating the expiry attribute. This represents a huge risk to the user's position.

Exploit Scenario:

- The user submits a market order A at 3:10pm with an expiry set to 3:14pm, in addition to a slPrice that will create an order to protect his position if the price drops to a limit.
- The market order gets executed at 3:13pm and the price decreases significantly and passes the limit set in the SL order at 3:20pm.
- The SL order is expired by the time, therefore the Keepers will not be able to execute the SL order.

This results in a huge risk on the user's position where a price decrease can directly result in the position's liquidation, bypassing the SL order protection. The same thing applies on TP orders.

Files Affected:

SHB.1.1: Orders.sol

```
// submit take profit order
if (tpPrice > 0) {
    params.price = tpPrice;
    params.orderType = 1;
    params.isReduceOnly = true;

    // Order is reduce-only so valueConsumed is always zero
    (tpOrderId, ) = _submitOrder(params);
}

// submit stop loss order
if (slPrice > 0) {
    params.price = slPrice;
    params.orderType = 2;
    params.isReduceOnly = true;

    // Order is reduce-only so valueConsumed is always zero
    (slOrderId, ) = _submitOrder(params);
}

// Update orders to cancel each other
if (tpOrderId > 0 && slOrderId > 0) {
    orderStore.updateCancelOrderId(tpOrderId, slOrderId);
    orderStore.updateCancelOrderId(slOrderId, tpOrderId);
}
```

Recommendation:

Consider resetting the expiry timestamp of the TP and SL orders when these options are enabled.

Updates

The CAP team resolved the issue by setting the expiry to zero before creating the TP and SL orders.

SHB.1.2: Orders.sol

```
// reset order expiry for TP/SL orders
if (params.expiry > 0) params.expiry = 0;
```

SHB.2 Stakers may lose their rewards due to rounding errors

- Severity: HIGH
- Status: Fixed
- Likelihood: 2
- Impact: 3

Description:

By staking CAP, the users receive a portion of protocol fees directly in ETH and USDC. The fees are distributed among stakers based on their share of the staking pool and can be withdrawn at any time. However, there is a possibility of a rounding error in the incrementRewardPerToken. If the pendingReward[asset] * UNIT is lower than the totalSupply the amount variable will round to zero. This results in the stakers losing a part of their rewards since pendingReward[asset] is set to zero afterwards and the rewardPerTokenSum[asset] is not incremented. There is also a precision loss even in the case where the amount is different from zero, this will occur whenever the pendingReward[asset] * UNIT is not divisible by the totalSupply. The UNIT multiplier reduces the risk but it does not solve the issue. The probability of this issue depends on the collected fees by the protocol, and it increases by the increase of the CAP tokens staked.

Files Affected:

SHB.2.1: StakingStore.sol

```
/// @notice Increments `asset` reward per token
/// @dev Only callable by other protocol contracts
```

```solidity
function incrementRewardPerToken(address asset) external onlyContract {
    if (totalSupply == 0) return;
    uint256 amount = (pendingReward[asset] * UNIT) / totalSupply;
    rewardPerTokenSum[asset] += amount;
    pendingReward[asset] = 0;
}
```

Recommendation:

Consider using the following code to update the reward per token:

SHB.2.2: StakingStore.sol

```solidity
/// @notice Increments `asset` reward per token
/// @dev Only callable by other protocol contracts
function incrementRewardPerToken(address asset) external onlyContract {
    if (totalSupply == 0) return;
    uint256 nonWithdrawableFunds = (pendingReward[asset] * UNIT) %
        ↪ totalSupply;
    uint256 amount = (pendingReward[asset] * UNIT - nonWithdrawableFunds
        ↪ ) / totalSupply;
    rewardPerTokenSum[asset] += amount;
    pendingReward[asset] = nonWithdrawableFunds/UNIT;
}
```

Updates

The CAP team resolved the issue by using the following code:

SHB.2.3: StakingStore.sol

```solidity
function incrementRewardPerToken(address asset) external onlyContract {
    if (totalSupply == 0) return;
    uint256 amount = (pendingReward[asset] * UNIT) / totalSupply;
    rewardPerTokenSum[asset] += amount;
    // due to rounding errors a fraction of fees stays in the contract
    // pendingReward is set to the amount which is left over, and will be distributed later
    pendingReward[asset] -= (amount * totalSupply) / UNIT;
}
```

SHB.3 Fees can be bypassed

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 1
- Impact : 3

Description:

The withdraw is utilized to withdraw funds from the pool, calling this function costs a withdrawal fee that is capped at 5% of the amount to be withdrawn. However, the fees can be bypassed by the caller by withdrawing his amount over multiple rounds of small amounts. The amount to be withdrawn in each step should be less than 100 units to be able to cause a rounding error in line 205 to be able to bypass the withdrawal fee. It is worth mentioning that this is only possible if the gas fees are low, which is possible in a layer 2 chain (Arbitrum).

Files Affected:

SHB.3.1: Pool.sol

```
189 function withdraw(address asset, uint256 amount) public {
190     require(amount > 0, '!amount');
191     require(assetStore.isSupported(asset), '!asset');
192
193     address user = msg.sender;
194
195     // check pool balance and clp supply
196     uint256 balance = poolStore.getBalance(asset);
197     uint256 clpSupply = poolStore.getClpSupply(asset);
198     require(balance > 0 && clpSupply > 0, '!empty');
```

```

    // check user balance
    uint256 userBalance = poolStore.getUserBalance(asset, user);
    if (amount > userBalance) amount = userBalance;

    // calculate pool withdrawal fee
    uint256 feeAmount = (amount * poolStore.getWithdrawalFee(asset)) / BPS_DIVIDER;
    uint256 amountMinusFee = amount - feeAmount;
```

Recommendation:

Consider implementing a restriction on the amount to be withdrawn and setting a minimum value that will prevent any fee bypass caused by rounding errors.

Updates

The CAP team resolved the issue by verifying the amount argument to be greater than BPS_DIVIDER.

SHB.3.2: Pool.sol

```
function withdraw(address asset, uint256 amount) public {
    require(amount > BPS_DIVIDER, '!amount');
    require(assetStore.isSupported(asset), '!asset');
```

SHB.4 Keeper's native tokens can get locked

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 1
- Impact : 3

Description:

The executeOrders is used by the Keeper to execute submitted orders, the Keeper needs to pay a fee that will be transferred to pyth. However, the require statement makes sure that the msg.value is greater than the fee. Therefore, if the keeper deposits a msg.value that is higher than fee, the msg.value - fee will not be used, and the keeper will not be able to get it back.

Files Affected:

SHB.4.1: Processor.sol

```solidity
function executeOrders(
    uint256[] calldata orderIds,
    bytes[] calldata priceUpdateData
) external payable nonReentrant ifNotPaused {
    // updates price for all submitted price feeds
    uint256 fee = pyth.getUpdateFee(priceUpdateData);
    require(msg.value >= fee, '!fee');
    pyth.updatePriceFeeds{value: fee}(priceUpdateData);
```

Recommendation:

Consider verifying the msg.value to be equal to the fee, or to transfer back the msg.value - fee.

Updates

The CAP team resolved the issue by refunding the diff to the sender.

SHB.4.2: Processor.sol

```solidity
// Refund msg.value excess, if any
if (msg.value > fee) {
    uint256 diff = msg.value - fee;
    payable(msg.sender).sendValue(diff);
}
```

SHB.5 Excessive Privileges Granted to the Governance Account

- Severity : MEDIUM
- Status : Acknowledged
- Likelihood : 1
- Impact : 3

Description:

The Governable contract is designed to have the governance account have excessive privileges within the project business logic. The onlyGov modifier is used to restrict access to certain functions and enforce that they can only be called by the governance account. However, this creates a centralized risk as the governance account has control over many critical functions such as setting assets in the AssetStore contract, setting funding intervals in the FundingStore, updating markets in the MarketStore contract, setting the maxMarketOrderTTL, maxTriggerOrderTTL and chainlinkCooldown using the OrderStore functions and all functions within the storage contract DataStore. This concentration of power in a single account increases the risk of a single point of failure and goes against the principles of decentralization.

Files Affected:

SHB.5.1: Governable.sol

```
    constructor() {
        _setGov(msg.sender);
    }

    /// @dev Reverts if called by any account other than gov
    modifier onlyGov() {
        require(msg.sender == gov, '!gov');
        _;
    }
```

Recommendation:

To mitigate the centralization risk, it is recommended to implement a more robust governance structure, such as a multi-sig or an on-chain voting mechanism. This would ensure that there is no single point of failure and that the system is more resilient to attacks. Additionally, it may be a good idea to limit the scope of the onlyGov modifier to only those functions that truly require governance-level access.

Updates

The CAP team acknowledged the risk stating that they are planning to set the gov to a multi-sig.

SHB.6 Blocked Contract Features Due to Missing Link Function Call

- Severity : MEDIUM
- Status : Acknowledged
- Likelihood : 1
- Impact : 3

Description:

The API Contracts, including Finding, Orders, Pool, Positions, Processor, and Staking, requires communication with the store contracts. If the link function that initializes protocol contracts is not called immediately after contract deployment, the features of these API Contracts will be blocked.

Files Affected:

All link functions in the Finding.sol, Orders.sol, Pool.sol, Positions.sol, Processor.sol, and Staking.sol contracts.

Recommendation:

Consider calling the link function in the contract's constructor or implementing a fail-safe mechanism that automatically gets the required store contract address from the DataStore, if it has not already been set through the link function.

Updates

The CAP team acknowledged the risk stating that they will be using a deployment script to solve the issue.

SHB.7 Unchecked return value in granting roles

- Severity : LOW
- Status : Fixed
- Likelihood : 2
- Impact : 1

Description:

The RoleStore contract makes use of the EnumerableSet contract from OpeenZepplin. The add and remove functions return a boolean value that represents the status of the call. This boolean value is not being checked in the contract. This makes the grantRole and revokeRole calls succeed in all cases, even if the add and remove fail.

Files Affected:

SHB.7.1: RoleStore.sol

```
/// @notice Grants `role` to `account`
/// @dev Only callable by governance
function grantRole(address account, bytes32 role) external onlyGov {
    roles.add(role);
    roleMembers[role].add(account);
}
```

SHB.7.2: RoleStore.sol

```
/// @notice Revokes `role` from `account`
/// @dev Only callable by governance
function revokeRole(address account, bytes32 role) external onlyGov {
    roleMembers[role].remove(account);
}
```

Recommendation:

Consider wrapping the add and remove calls inside a require to make sure the transaction status accurately represents the state changes.

Updates

The CAP team resolved the issue by wrapping the add and remove calls inside a require.

SHB.7.3: RoleStore.sol

```
function grantRole(address account, bytes32 role) external onlyGov {
    // add role if not already present
    if (!roles.contains(role)) roles.add(role);

    require(roleMembers[role].add(account));
}
```

SHB.7.4: RoleStore.sol

```
function revokeRole(address account, bytes32 role) external onlyGov {
    require(roleMembers[role].remove(account));

    // Remove role if it has no longer any members
    if (roleMembers[role].length() == 0) {
        roles.remove(role);
    }
}
```

SHB.8 Missing value verification

- Severity : LOW
- Status : Fixed
- Likelihood : 2
- Impact : 1

Description:

Certain functions lack a value safety check. The values of the arguments should be verified to allow only the ones that comply with the contract's logic.

- In the OrderStore contract, the setMaxMarketOrderTTL and setMaxTriggerOrderTTL functions are called by the gov to set the maxMarketOrderTTL and maxTriggerOrderTTL variables. By default, maxMarketOrderTTL is set to 5 minutes and maxTriggerOrderTTL is set to 180 days. However, there is no verification to ensure that the amount parameter passed to these functions is not equal to zero and the maxMarketOrderTTL should be verified to be lower than the maxTriggerOrderTTL argument. This can result in the submitOrder process being blocked, as the _submitOrder function requires that ttl must be less than or equal to the value of maxMarketOrderTTL() or maxTriggerOrderTTL(). The same requirement is present in the _executeOrder function in the Processor contract.

- In the OrderStore contract, the setChainlinkCooldown function takes the amount parameter, which is the duration in seconds. However, there is no verification performed to ensure that the amount parameter is not equal to zero, which can result in the _executeOrder function in the Processor contract being blocked as if the order was submitted less than chainlinkCooldown seconds ago, this function will return false with an error message.

- In the FundingStore contract, there is no verification to ensure that the amount parameter passed to the setFundingInterval function is not equal to zero. This can result in the updateFundingTracker function in the Funding contract failing as the calculation lastUpdated + fundingStore.fundingInterval() may exceed the value of _now if fundingInterval is set to zero.

- The setFeeShare function from PoolStore and StakingStore contracts is missing a limitation over the value of the fee.
- The same issue in the setRemoveMarginBuffer and setKeeperFeeShare functions from PositionStore contract
- The same issue in the setPoolProfitLimit function in the RiskStore contract.

Files Affected:

SHB.8.1: OrderStore.sol

```
    function setMaxMarketOrderTTL(uint256 amount) external onlyGov {
        maxMarketOrderTTL = amount;
    }

    /// @notice Set duration until trigger orders expire
    /// @dev Only callable by governance
    /// @param amount Duration in seconds
    function setMaxTriggerOrderTTL(uint256 amount) external onlyGov {
        maxTriggerOrderTTL = amount;
    }
```

SHB.8.2: OrderStore.sol

```
    function setChainlinkCooldown(uint256 amount) external onlyGov {
        chainlinkCooldown = amount;
    }
```

SHB.8.3: FundingStore.sol

```
  function setFundingInterval(uint256 amount) external onlyGov {
        fundingInterval = amount;
    }
```

SHB.8.4: StakingStore.sol and PoolStore.sol

```
function setFeeShare(uint256 bps) external onlyGov {
    feeShare = bps;
}
```

SHB.8.5: RiskStore.sol

```
function setPoolProfitLimit(address asset, uint256 bps) external onlyGov
    ↪  {
    require(bps <= MAX_POOL_PROFIT_LIMIT, '!profit-limit');
    poolProfitLimit[asset] = bps;
}
```

SHB.8.6: PositionStore.sol

```
    function setRemoveMarginBuffer(uint256 bps) external onlyGov {
        removeMarginBuffer = bps;
    }

    /// @notice Sets keeper fee share
    /// @dev Only callable by governance
    /// @param bps new `keeperFeeShare` in bps
    function setKeeperFeeShare(uint256 bps) external onlyGov {
        require(bps <= MAX_KEEPER_FEE_SHARE, '!keeper-fee-share');
        keeperFeeShare = bps;
    }
```

Recommendation:

- It is recommended that the OrderStore contract be updated to include a check that verifies that the amount parameter passed to the setMaxMarketOrderTTL and setMaxTriggerOrderTTL functions is not equal to zero.Additionally, the maxMarketOrderTTL should be verified to be lower than the maxTriggerOrderTTL argument.
- It is recommended to add a verification to ensure that the amount parameter passed to the setChainlinkCooldown function is not equal to zero, in order to avoid potential issues with the _executeOrder function.
- It is recommended to add a verification check in the setFundingInterval function to ensure that the amount parameter is not equal to zero.

Updates

The CAP team resolved the issue by verifying the values as recommended.

SHB.9 Lack of Contract Verification for Granting the CONTRACT Role

- Severity : LOW
- Status : Acknowledged
- Likelihood : 1
- Impact : 2

Description:

In the Roles contract, the onlyContract modifier is used to ensure that the calling account has the CONTRACT role.In the RoleStore contract, the grantRole function is only accessible by the gov, but there is no condition to ensure that the account being granted the CONTRACT role is a smart contract. This lack of verification can lead to security risks, as a non-contract account could potentially be granted the CONTRACT role and have access to sensitive functionality within the system.

Files Affected:

SHB.9.1: RoleStore.sol

```solidity
    function grantRole(address account, bytes32 role) external onlyGov {
        roles.add(role);
        roleMembers[role].add(account);
    }
```

Recommendation:

Consider adding a condition to the grantRole function in the RoleStore contract to verify that the account being granted the CONTRACT role is a smart contract.

Updates

The CAP team acknowledged the risk stating that verifying the account to be a smart contract will not reduce the risk.

SHB.10 Lack of Two-Factor Verification for Updating gov Address

- Severity : LOW
- Status : Acknowledged
- Likelihood : 1
- Impact : 2

Description:

The setGov function is used by the governance in order to change the governance address, there is a risk of the governance being set to address(0) or a wrong address by accident, which can lead to a denial of service in all the functions protected by the onlyGov modifier.

Files Affected:

SHB.10.1: Governable.sol

```
/// @notice Sets a new governance address
/// @dev Only callable by governance
function setGov(address _gov) external onlyGov {
    _setGov(_gov);
}
```

Recommendation:

Consider changing the gov address over two steps, where the first is setting up a pendingGov and the second call is done by the pendingGov where they can take the ownership and be the new gov.

Updates

The CAP team acknowledged the risk as they decided to keep it a one step modification.

SHB.11 Transaction Order Dependency & Potential Loss of Precision in Fee Calculation

- Severity : LOW
- Status : Partially Fixed

- Likelihood : 1
- Impact : 2

Description:

The creditFee function in the Position contract calculates fees based on the keeperFeeShare and feeShare set in the store contracts. However, these fees are modifiable by the governance and there is an order dependency between the calculation of the fee and the modification of the fee share, which may lead to an unexpected result. In addition to that, the fee calculation can result in a precision loss due to the percentages that are taken from the trading fees, which depend on the order size.

Files Affected:

SHB.11.1: Positions.sol

```
    function creditFee(
        uint256 orderId,
        address user,
        address asset,
        string memory market,
        uint256 fee,
        bool isLiquidation,
        address keeper
    ) public onlyContract {
        if (fee == 0) return;

        uint256 keeperFee;

        if (keeper != address(0)) {
            keeperFee = (fee * positionStore.keeperFeeShare()) / BPS_DIVIDER;
        }

        // Calculate fees
        uint256 netFee = fee - keeperFee;

        uint256 feeToStaking = (netFee * stakingStore.feeShare()) / BPS_DIVIDER;
        uint256 feeToPool = (netFee * poolStore.feeShare()) / BPS_DIVIDER;
        uint256 feeToTreasury = netFee - feeToStaking - feeToPool;
```

Recommendation:

Consider improving the precision in the fees calculation, also adding the modifiable variables as arguments and verifying that they match the values stored in the contracts.

Updates

The CAP team resolved the precision loss issue by multiplying the fee by 10^{18} to increase the precision.

SHB.11.2: Positions.sol

```
function creditFee(
    uint256 orderId,
    address user,
    address asset,
    string memory market,
    uint256 fee,
    bool isLiquidation,
    address keeper
) public onlyContract {
    if (fee == 0) return;

    // multiply fee by UNIT (10^18) to increase position
    fee = fee * UNIT;

    uint256 keeperFee;
    if (keeper != address(0)) {
        keeperFee = (fee * positionStore.keeperFeeShare()) / BPS_DIVIDER;
    }

    // Calculate fees
    uint256 netFee = fee - keeperFee;
    uint256 feeToStaking = (netFee * stakingStore.feeShare()) /
        ↪ BPS_DIVIDER;
    uint256 feeToPool = (netFee * poolStore.feeShare()) / BPS_DIVIDER;
    uint256 feeToTreasury = netFee - feeToStaking - feeToPool;

    // Increment balances, transfer fees out
    // Divide fee by UNIT to get original fee value back
    poolStore.incrementBalance(asset, feeToPool / UNIT);
    stakingStore.incrementPendingReward(asset, feeToStaking / UNIT);
    fundStore.transferOut(asset, DS.getAddress('treasury'),
        ↪ feeToTreasury / UNIT);
    fundStore.transferOut(asset, keeper, keeperFee / UNIT);

    emit FeePaid(
        orderId,
        user,
        asset,
        market,
        fee / UNIT, // paid by user
        feeToPool / UNIT,
        feeToStaking / UNIT,
        feeToTreasury / UNIT,
        keeperFee / UNIT,
        isLiquidation
    );
}
```

SHB.12 Potential Reentrancy Attack

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The withdraw function allows users to withdraw their balance in a specific asset from the pool store. The function calls the transferOut function to transfer the withdrawn funds to the user’s address. However, the implementation of the transferOut function is vulnerable to re-entrancy attacks.

The problem with sendValue is that it is a low-level function that sends Ether directly to the recipient without any protection from re-entrancy attacks. This can lead to potential security risks and unauthorized funds transfer.

This issue is not limited to the withdraw function, but rather it is a widespread problem that affects all functions that calls the transferOut function.

Files Affected:

SHB.12.1: Pool.sol

```
        // transfer funds out
        fundStore.transferOut(asset, user, amountMinusFee);
```

SHB.12.2: FundStore.sol

```
    function transferOut(address asset, address to, uint256 amount) external onlyContractOrGov {
        if (amount == 0  to == address(0)) return;
        if (asset == address(0)) {
            payable(to).sendValue(amount);
        } else {
            IERC20(asset).safeTransfer(to, amount);
        }
    }
```

Recommendation:

To mitigate this risk, it is recommended to use the nonReentrant modifier from the ReentrancyGuard by Openzeppelin in the transferOut function.

Updates

The CAP team resolved the issue by implementing the use of the nonReentrant modifier.

SHB.12.3: FundStore.sol

```
function transferOut(address asset, address to, uint256 amount) external onlyContractOrGov {
    if (amount == 0  to == address(0)) return;
    if (asset == address(0)) {
        payable(to).sendValue(amount);
    } else {
        IERC20(asset).safeTransfer(to, amount);
    }
}
```

SHB.13 Floating Pragma

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 1

Description:

The contract makes use of the floating-point pragma 0.8.13. Contracts should be deployed using the same compiler version. Locking the pragma helps ensure that contracts will not unintentionally be deployed using another pragma, which in some cases may be an obsolete version, that may introduce issues to the contract system.

Files Affected:

All Contracts

Recommendation:

Consider locking the pragma version. It is advised that floating pragma should not be used in production. Both truffle-config.js and hardhat.config.js support locking the pragma version.

Updates

The CAP team resolved the issue by locking the pragma version to 0.8.17.

4 Best Practices

BP.1 Removing Roles Without Members

Description:

The revokeRole function in the RoleStore contract allows governance to remove a specific role from a given account by removing the account from the roleMembers mapping for that role. To maintain a clean and efficient role management system, it is a best practice to also remove any roles that no longer have any members. This can be done by adding the following code to the revokeRole function:

BP.1.1: RoleStore.sol

```
if (roleMembers[role].isEmpty()) {
    roles.remove(role);
}
```

This will ensure that the getRoleCount function returns the correct number of roles in the system and prevents the accumulation of unused roles.

Files Affected:

BP.1.2: RoleStore.sol

```
function revokeRole(address account, bytes32 role) external onlyGov {
    roleMembers[role].remove(account);
}
```

Status - Fixed

BP.2 Use EnumerableSet.AddressSet for Asset List

Description:

In the AssetStore contract, the assetList array is used to keep track of all assets stored in the system. To optimize the asset management, it is recommended to use the EnumerableSet.AddressSet data structure instead of the standard array. This will ensure that each asset is stored only once, avoiding duplication and improving performance. The set function in the code can be modified to directly use the add function provided by the EnumerableSet.AddressSet.

Files Affected:

BP.2.1: AssetStore.sol

```
    address[] public assetList;
    mapping(address => Asset) private assets;

    constructor(RoleStore rs) Roles(rs) {}

    /// @notice Set or update an asset
    /// @dev Only callable by governance
    /// @param asset Asset address, e.g. address(0) for ETH
    /// @param assetInfo Struct containing minSize and chainlinkFeed
    function set(address asset, Asset memory assetInfo) external onlyGov {
        assets[asset] = assetInfo;
        for (uint256 i = 0; i < assetList.length; i++) {
            if (assetList[i] == asset) return;
        }
        assetList.push(asset);
    }
```

Status - Acknowledged

5 Tests

Results:

```
Running 13 tests for test/foundry/Orders.t.sol:OrderTest
[PASS] testCancelOrder() (gas: 433232)
[PASS] testCancelOrderUSDC() (gas: 467372)
[PASS] testRefundMsgValueExcess() (gas: 535964)
[PASS] testReverTPBelowSLPrice() (gas: 497723)
[PASS] testRevertAboveMaxLeverage() (gas: 103200)
[PASS] testRevertBelowMinLeverage() (gas: 103512)
[PASS] testRevertBelowMinSize() (gas: 67814)
[PASS] testRevertExpiry() (gas: 340924)
[PASS] testRevertOrdersPaused() (gas: 73099)
[PASS] testRevertUnsupportedAsset() (gas: 84117)
[PASS] testRevertValue() (gas: 127636)
[PASS] testSubmitOrder() (gas: 504033)
[PASS] testSubmitOrderAssetUSDC() (gas: 549487)
Test result: ok. 13 passed; 0 failed; finished in 72.66ms

Running 2 tests for test/foundry/Funding.t.sol:FundingTest
[PASS] testFundingTrackerLong() (gas: 1771017)
[PASS] testFundingTrackerShort() (gas: 1707408)
Test result: ok. 2 passed; 0 failed; finished in 75.84ms

Running 4 tests for test/foundry/RiskStore.t.sol:RiskStoreTest
[PASS] testMaxOI() (gas: 1089752)
[PASS] testMaxPoolDrawdown() (gas: 1505028)
[PASS] testProfitTracker() (gas: 1552667)
[PASS] testProfitTrackerNegative() (gas: 1576304)
Test result: ok. 4 passed; 0 failed; finished in 86.15ms

Running 12 tests for test/foundry/Positions.t.sol:PositionsTest
[PASS] testAddMargin() (gas: 1092328)
```

```
[PASS] testAddMarginUSDC() (gas: 1131972)
[PASS] testClosePositionWithoutProfit() (gas: 1110899)
[PASS] testCreditFee() (gas: 1073156)
[PASS] testCreditFeeAssetUSDC() (gas: 1117742)
[PASS] testDecreasePosition() (gas: 2494982)
[PASS] testDecreasePositionReduceOnly() (gas: 1519096)
[PASS] testIncreasePosition() (gas: 1069740)
[PASS] testRemoveMargin() (gas: 1099159)
[PASS] testRevertAddMargin() (gas: 1090203)
[PASS] testRevertClosePositionWithoutProfit() (gas: 1095736)
[PASS] testRevertRemoveMargin() (gas: 1124907)
Test result: ok. 12 passed; 0 failed; finished in 114.54ms

Running 11 tests for test/foundry/Processor.t.sol:ProcessorTest
[PASS] testCancelReduceOnlyOrder() (gas: 574428)
[PASS] testChainlinkDeviation() (gas: 737791)
[PASS] testExecuteLimitOrder() (gas: 1077137)
[PASS] testExecuteMarketOrder() (gas: 1831522)
[PASS] testExecuteStopOrder() (gas: 1076644)
[PASS] testLiquidatePosition() (gas: 1235012)
[PASS] testProtectedOrder() (gas: 645932)
[PASS] testSelfExecuteOrder() (gas: 938264)
[PASS] testSelfLiquidatePosition() (gas: 1190715)
[PASS] testSkipOrderStale() (gas: 707592)
[PASS] testSkipOrderTooEarly() (gas: 703074)
Test result: ok. 11 passed; 0 failed; finished in 374.15ms

Running 4 tests for test/foundry/Pool.t.sol:PoolTest
[PASS] testCreditTraderLoss() (gas: 3268902)
[PASS] testDebitTraderProfit() (gas: 3320660)
[PASS] testFuzzDepositAndWithdraw(uint256) (runs: 256,  : 142626, ~:
   ↪ 142838)
[PASS] testFuzzDepositAndWithdrawUSDC(uint256) (runs: 256,  : 164379, ~:
   ↪  165874)
```

```
Test result: ok. 4 passed; 0 failed; finished in 388.16ms

Running 1 test for test/foundry/Staking.t.sol:StakingTest
[PASS] testFuzzStakeAndUnstake(uint256) (runs: 256,  : 163821, ~:
   ↪ 163822)
Test result: ok. 1 passed; 0 failed; finished in 409.41ms
```

6 Conclusion

In this audit, we examined the design and implementation of CAP V4 contracts and discovered several issues of varying severity. Cap team addressed 8 issues raised in the initial report and implemented the necessary fixes, while classifying the rest as a risk with low-probability of occurrence. Shellboxes' auditors advised Cap Team to maintain a high level of vigilance and to keep those findings in mind in order to avoid any future complications.

7 Scope Files

7.1 Audit

Files	MD5 Hash
contracts/stores/RoleStore.sol	0a324954878e91f51ac506934be09e9c
contracts/stores/RiskStore.sol	705faa2db8b2dee9974f699fc2df2978
contracts/stores/AssetStore.sol	fb536f3e26ab34bf89f78e44b3c88281
contracts/stores/FundStore.sol	e3708b8cb573fbb2558467bb1957ace3
contracts/stores/FundingStore.sol	bd38b3fc0833bdc5dc3042a20329db2d
contracts/stores/DataStore.sol	359e13898de8c423d8a7a3d9f142d77c
contracts/stores/MarketStore.sol	d9a7071d921ef78bedebaeec370a378a
contracts/stores/StakingStore.sol	9d3a268e7027d028befafdd592975aae
contracts/stores/PoolStore.sol	c527d632aa857d8c2fad908a3fc24b3f
contracts/stores/OrderStore.sol	43dccffda93f2c64656307070bc9a499
contracts/stores/PositionStore.sol	5ae78c4e6523478628f124509f5502e3
contracts/api/Staking.sol	96a59cbd6c733c2baa53eb97634f458f
contracts/api/Orders.sol	3065e5fb1e1f90c789a55e9348acd5d3
contracts/api/Positions.sol	3f062788efae2e24e6834f11fde4e21b
contracts/api/Pool.sol	c00ae3d9134d7a4026fe01f23bdf3124
contracts/api/Processor.sol	adc05a585da53f666c3171328273bcfc
contracts/api/Funding.sol	4b17a68d0de303a92775cb9ef57626e7

contracts/utils/Roles.sol	0345a7568f52c44240742d7382c38bb1
contracts/utils/Governable.sol	19eff15601e393bb7024937b7a4c11ff
contracts/utils/Chainlink.sol	2e8029a1b108fb7a77ff73e8f974699e

7.2 Re-Audit

Files	MD5 Hash
contracts/utils/Chainlink.sol	2e8029a1b108fb7a77ff73e8f974699e
contracts/utils/Governable.sol	f7344c65b08f8ad392a6a97425318ccc
contracts/utils/Roles.sol	5bad220d7fbd65cbc710c6ac9746de33
contracts/stores/AssetStore.sol	8777fdc137859c87196db12bc4c485fc
contracts/stores/DataStore.sol	f30c824754a397ae885fbc7809f8c332
contracts/stores/FundingStore.sol	b2171793a1c33760d57f8fbd8eebd31a
contracts/stores/FundStore.sol	81eba8e45f23de90081851f76fc19c13
contracts/stores/MarketStore.sol	a01b7206ba3cb201aefd3f79de8ad510
contracts/stores/OrderStore.sol	1dd93706ed89a21cc35260b9e7e7888e
contracts/stores/PoolStore.sol	cbf31be5b4e4b743b1f036b0c4934054
contracts/stores/PositionStore.sol	2e29f828c9574b0c411521b2191c33a8
contracts/stores/RiskStore.sol	f08526becb4a19841baa5e96a80f6b32
contracts/stores/RoleStore.sol	61dedc8c242e4f3507205ad6886fb75e
contracts/stores/StakingStore.sol	8c25e559daa55fad64cafe302e37ff12

contracts/api/Funding.sol	4ae63f93472cb05a3f7e55bb886532b3
contracts/api/Orders.sol	5304f923cfb1450322fe1cd001afd255
contracts/api/Pool.sol	0287404b1a219e70de4dac71256f26c3
contracts/api/Positions.sol	415219f4ae748f47d0a36b57ee61b14c
contracts/api/Processor.sol	541ceeadb0511fe5c02370489fcefb88
contracts/api/Staking.sol	cfcdf1d74e85ff0f1e43e1a68da9b2f2

8 Disclaimer

Shellboxes reports should not be construed as ”endorsements” or ”disapprovals” of particular teams or projects. These reports do not reflect the economics or value of any ”product” or ”asset” produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology’s proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don’t offer any kind of investing advice and shouldn’t be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.

SHELLBOXES

For a Contract Audit, contact us at contact@shellboxes.com