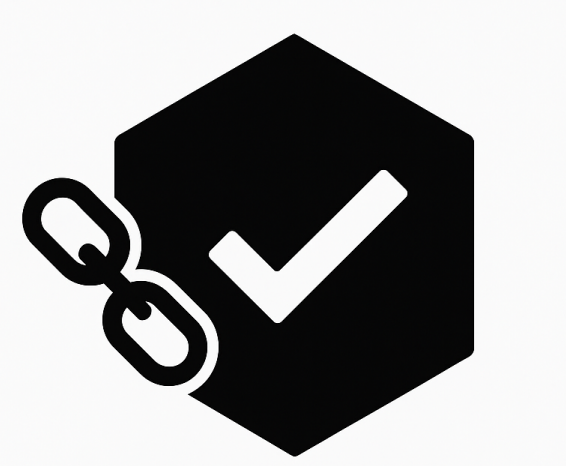

Company Simulator First Flight

Version 1.0

github.com/lucasfhope

October 31, 2025

Company Simulator First Flight

Lucas Hope

October 31, 2025

Table of Contents

Protocol Summary

Company Simulator is a decentralized smart contract system built in Vyper that simulates the operations of a virtual company. It includes modules for production, inventory management, customer demand, shareholding, reputation, and financial health.

First Flight Details

The findings described in this report corresponds to the following commit hash:

```
8bfead8b8a48bf7d20f1ee78a0566ab2b0b76e2d
```

Scope

```
src/
 --- Cyfrin_Hub.vy          # Core contract managing company operations
 --- CustomerEngine.vy      # Simulates customer demand and triggers
    sales
```

Roles

```
Owner: Deploys and controls core company functions such as production,
   share cap increases, and debt repayment.
Investor: Public user who funds the company and receives proportional
   shares.
Customer: Simulates demand by purchasing items from the company.
```

Issues found

The audit found 4 important vulnerabilities/issues that were submitted in the contest. The audit also found some other potential issues and problems in the codebase that were not submitted.

Submitted Findings

[S-1] Investors can steal funds supplied by the owner

Description

The protocol intends for investors to invest ETH for company shares which can be redeemed for their supplied ETH and any company profits. The contract owner can also supply funding to the contract, but the owner does not receive any shares.

Because the owner does not receive shares, and the shares can be redeemed based on the percentage of shares and the net worth of the company, any funding the owner supplies can be redeemed by an investor.

```
@view
@internal
def get_share_price() -> uint256:
    """
    @notice Calculates the current share price based on net worth.
    @dev Net worth = company_balance - holding_debt (capped at 0).
```

```
7       @dev Share price = net_worth / issued_shares.
8       @dev If no shares issued, returns INITIAL_SHARE_PRICE.
9       @return Price per share in wei.
10      """
11      if self.issued_shares == 0:
12          return INITIAL_SHARE_PRICE
13  @>  net_worth: uint256 = max(self.company_balance - self.holding_debt,
        0)
14  @>  return net_worth // self.issued_shares
```

Risk

Likelihood: High

This will occur whenever the owner funds the company through `fund_owner`, which requires calling `fund_cyfrin` with 0 as the parameter. An investor would also have funded the contract by calling `fund_cyfrin` with 1 as the parameter, which would allow them to receive shares. An investor could redeem their shares and collect some of the owner's funds, since the share price is based off of the net worth of the contract and the amount of shares. The owner does not receive any shares, but their supplied funding is directly tied to the share price.

Impact: High

Funds supplied by the owner directly impact the share price and will be used to payout shareholders. While the MAX_PAYOUT_PER_SHARE is capped, repeated investing and withdrawing will allow malicious investors to steal ETH from the contract.

Proof of Concept

Add this test to `tests/unit/test_Industry.py`.

```
1  def test_Attacker_Can_Steal_Owner_Provided_Funds(industry_contract,
       OWNER, PATRICK):
2      amount_to_invest = boa.env.get_balance(PATRICK)
3      boa.env.set_balance(OWNER, SET_OWNER_BALANCE)
4      with boa.env.prank(OWNER):
5          industry_contract.fund_cyfrin(0, value=SET_OWNER_BALANCE)
6
7      initial_balance = industry_contract.get_balance()
8      print(f"Initial Industry Contract Balance: {initial_balance}")
9
10     user_balance_before = boa.env.get_balance(PATRICK)
11     print(f"Initial Attacker Balance:          {user_balance_before}")
12     print(f"Attacker invest amount:            {amount_to_invest}")
13
```

```
    with boa.env.prank(PATRICK):
        industry_contract.fund_cyfrin(1, value=amount_to_invest)
        print(f"Attacker shares: {industry_contract.get_my_shares(
            caller=PATRICK)}")
        industry_contract.withdraw_shares()
    user_balance_after = boa.env.get_balance(PATRICK)
    print(f"End Attacker Balance:              {user_balance_after}")

    assert user_balance_after > user_balance_before, "Attacker should
        have increased their balance"
```

This will show that because the owner has funded the contract, an investor can invest and withdraw in one transaction, stealing 80% of the owners funds provided. Note that it is only 80% percent because of the 10% early withdrawal fee over the entire company net worth.

Recommended Mitigation

Consider giving the owner shares for the funds they provide. Another option would be to calculate share price based on the precentage of share funding vs owner funding, giving the owner the correct percentage of the company based on their supplied funds.

[S-2] Reputation falling too low results in a denial of service

Description

`CustomerEngine::trigger_demand` will call `sell_to_customer` on the company contract, requesting between 1 and 6 of the product. It is on a 60 second cooldown for each caller, and forces the company contract to have a reputation above or equal to 60. The company function, `sell_to_customer`, can only be called by the `CustomerEngine`, and `trigger_demand` is the only function that can call `sell_to_customer`. Furthermore, reputation can only increase with a successful sale in `sell_to_customer`. Therefore, if a company's reputation falls below 60, `trigger_demand` will always revert and the company's reputation will never be able to recover.

```
@payable
@external
def trigger_demand():

    ...

    rep: uint256 = staticcall CompanyGame(self.company).reputation()
@>  assert rep >= MIN_REPUTATION, "Reputation too low for demand!!!"

    ...
```

```
11
12
13 @external
14 @payable
15 def sell_to_customer(requested: uint256):
16
17     ...
18
19 @>  assert msg.sender == self.CUSTOMER_ENGINE, "Not the customer engine
       !!!"
20
21     ...
```

Risk

Likelihood: High

This will occur whenever the company falls below a 60 reputation score. This is fairly common because the only way for the company to produce any inventory is for the owner to call `produce`. Since the `REPUTATION_PENALTY` is 5, if only 9 demand requests come in while the company has no inventory, they will fall to a reputation of 55.

Impact: High

If this occurs, any attempts to trigger demand will be denied. The company will be unable to function and sell any more product.

Proof of Concept

Add this test in `tests/unit/test_Engine.py`.

```
1  def test_DoSWithLowReputation(industry_contract,
       customer_engine_contract, OWNER, PATRICK):
2      with boa.env.prank(PATRICK):
3          for i in range(0,9):
4              customer_engine_contract.trigger_demand(value=to_wei(0.1, "
                   ether"))
5              boa.env.time_travel(seconds=61)
6
7      with boa.env.prank(OWNER):
8          industry_contract.fund_cyfrin(0, value=to_wei(10, "ether"))
9          industry_contract.produce(50)
10
11     with boa.reverts("Reputation too low for demand!!!"):
12         with boa.env.prank(PATRICK):
13             customer_engine_contract.trigger_demand(value=to_wei(0.1, "
                   ether"))
```

This test will show that 9 demand requests while no inventory has been produced will prevent any more demand requests.

Recommended Mitigation

Consider implementing other ways for the reputation to increase. One good option would be when the company produces more inventory. Another would be when the company receives funding and is brought out of debt.

Also consider allowing the company to produce more inventory within `sell_to_customer`, removing the sole reliance on the owner to continuously produce inventory. This would allow for requests to be satisfied when the company has enough funds to produce more inventory.

[S-3] The owner cannot redeem any of their supplied funds or profit

Description

`Cyfrin_Hub` is set up to allow for investors to supply funds for shares of the company. It also allows the company owner to supply funds, but they do not get shares for their investment. This means that investors can redeem portions of the company's holdings with their shares, but the owner cannot access or redeem any of the holdings.

This is specifically for when the owner funds using `fund_cyfrin` with 0 as the parameter which would call `fund_owner`.

```
@payable
@internal
def fund_owner():
    """
    @notice Allows the owner to inject ETH into the company without
       receiving shares.
    @dev Increases company_balance directly. No shares are issued.
        Only the owner can call this function.
    @dev This simulates owner capital injections or personal investment
       .
    """
    assert msg.sender == OWNER, "Not the owner!!!"
    self.company_balance += msg.value
```

Note that there is nothing stopping the owner from going through `fund_investor`, which would give the owner shares for their investment.

There is no clear way for funds, including profits, to be recovered from this contract by the owner. If the company grows large, most of the funds will be locked in the contract.

Risk

Likelihood: High

The owner would have to invest into the compnay without receiving shares. Most of the problems would occur if the company grew large, leaving most funds in the contract inaccessible due to the `MAX_PAYOUT_PER_SHARE`.

Impact: High

The owner cannot recover any of their investment into the company. They also cannot access any of the company's profits.

Recommended Mitigation

Consider tracking the contributions of the owner. They should be able to gain shares, but their shares should not have a cap.

Also consider allowing the owner to be able to increase the amount of public shares to more than `TOTAL_SHARES`, and allow the owner to reduce the number of shares when not all are being held. This would allow the owner to control the amount of the company they own.

[S-4] Smart contract wallets without a default payable function will not be able to withdraw shares

Description

When investors call `withdraw_shares` to redeem shares for ETH, `Cyfrin_Hub` attempts a `raw_call` to send the ETH to the caller. However, if the shareholder is a smart contract without a default payable function, the call will fail and the withdraw will revert. Therefore, shares held by one of these contracts will be unredeemable.

```
@external
def withdraw_shares():

    ...

@>  raw_call(
       msg.sender,
```

```
        b"",
        value=payout,
        revert_on_failure=True,
    )

    ...
```

Risk

Likelihood: Low

The shareholder must be a smart contract wallet without the ability to receive payment.

Impact: Medium

The affected shareholder will never be able to redeem their shares, and the shares will be permanently out of circulation.

Proof of Concept

Create this DeadShares contract in `tests/unit/attackContracts`.

```
company: immutable(address)

@deploy
def __init__(_company: address):
    company = _company

@payable
@external
def buy_shares():
    data: Bytes[36] = concat(
        method_id("fund_cyfrin(uint256)"), convert(1, bytes32)
    )
    raw_call(company, data, value=msg.value, revert_on_failure=True)

@external
def sell_shares():
    data: Bytes[4] = method_id("withdraw_shares()")
    raw_call(
        company,
        data,
        revert_on_failure=True,
    )
```

Then add this test in `tests/unit/test_Industry.py`.

```python
def test_BadContractCantSellShares(industry_contract, OWNER, PATRICK):
    boa.env.set_balance(OWNER, SET_OWNER_BALANCE)
    with boa.env.prank(OWNER):
        industry_contract.fund_cyfrin(0, value=SET_OWNER_BALANCE)

    with boa.env.prank(PATRICK):
        dead_shares_contract = boa.load("tests/unit/attackContracts/
            DeadShares.vy", industry_contract.address)
        dead_shares_contract.buy_shares(value=boa.env.get_balance(
            PATRICK))

    with boa.env.prank(dead_shares_contract.address):
        print(f"Attacker shares: {industry_contract.get_my_shares()}")

    with boa.reverts():
        with boa.env.prank(PATRICK):
            dead_shares_contract.sell_shares()
```

This test will show that attempts for the contract to withdraw ETH for the shares will always revert.

Recommended Mitigation

You should not be pushing ETH inside `withdraw_shares`. Instead, consider adopting a method for users to pull ETH from the contract rather than push it to them. This could be done but setting pull amounts in a mapping during `withdraw_shares` and creating another function for a user to pull out the funds. Even if a smart contract wallet doesn't redeem their ETH, the shares will then be available for other investors.

You could also consider adding a function to convert the ETH into WETH and send it to the contract if they are unable to redeem the ETH directly. This could be callable by any EOA on behalf of the smart contract wallet.

Other Potential Issues

[Note-1] Internal accounting does not reflect contract ETH balance

In `Cyfrin_Hub:sell_to_customers`, `apply_holding_cost` is called, which will charge the company for the inventory based on the time since the last holding cost was applied. This either subtracts from the company balance or adds to the holding debt. While these balances are updated, the ETH balance in the contract is unaffected. This allows for the ETH balance to be far above the calculated company net worth (company balance - holding debt).

Furthermore, `apply_holding_cost` calculates the the holding time based on the last time holding costs were applied, disregarding the amount of time the particular inventory has been in holding.

Also, only the owner can pay off holding debt, meaning that the company balance cannot be used to negate the holding debt.

This means that `Cyfrin_Hub::get_balance` will return a balance that does not reflect the ETH balance of the contract.

[Note-2] `CustomerEngine::trigger_demand` has issues with pseudo randomness

The pseudo-randomness can be manipulated by validators in `trigger_demand` since it relies on `block.timestamp`.

```
seed: uint256 = convert(
    keccak256(
        concat(
            convert(block.timestamp, bytes32), convert(msg.sender,
               bytes32)
        )
    ),
    uint256,
)
```

This issue is that this value determines the amount of inventory, which also determines the amount of ETH they are required to send with the transaction. This blocks users that can only send a certain amount of ETH. If enough ETH is not sent along the transaction to cover the max demand, then malicious validators can force the transaction to revert, costing the sender gas fees without completeing the transaction.

It is best practice to allow the caller to determine the amount of demand they would like to trigger.

[Note-3] `Cyfrin_Hub::sell_to_customer` only logs reputation when it decreases

The function only logs the reutation chnage when it increases, making it harder for off chain entities to track the reputation. There is also a pointless else block that sets the reputation to 100 when it is already at 100.

```
@external
@payable
def sell_to_customer(requested: uint256):

    ...

```

```
    if self.inventory >= requested:
        self.inventory -= requested
        revenue: uint256 = requested * SALE_PRICE
        self.company_balance += revenue
        if self.reputation < 100:
            # Increase reputation for successful sale
            self.reputation = min(self.reputation + REPUTATION_REWARD,
                100)
+           log ReputationChanged(new_reputation=self.reputation)
-       else:
-           # Maintain reputation if already at max
-           self.reputation = 100
        log Sold(amount=requested, revenue=revenue)
    else:
        self.reputation = min(max(self.reputation - REPUTATION_PENALTY,
            0), 100)

        log ReputationChanged(new_reputation=self.reputation)
```

[Note-4] `Cyfrin_Hub::withdraw_shares` should apply the early withdraw penalty after the max payout calculation.

If investors withdraw shares early but are still above the max payout, they will receeive the max payout. They should probably still receive a penalty on the max payout for early withdrawal.

[Note-5] `Cyfrin_Hub::set_customer_engine` should check `CUSTOMER_ENGINE` is not address(0)

The boolean variable `customer_engine_set` is just a waste of gas, since `CUSTOMER_ENGINE` can be checked against address(0) to determine if it has been set.

[Note-6] `Cyfrin_Hub::fund_owner` should check msg.value is above 0

Even though it is the owner, who should know to send balance, it is good practice for the function to require a msg.value above 0.