// HALBORN

Uranium3o8 - Launchpad

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **October 6th, 2023 - October 17th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	10/17/2023
0.2	Document Updates	10/17/2023
0.3	Draft Review	10/17/2023
1.0	Remediation Plan	10/26/2023
1.1	Remediation Plan Review	10/27/2023

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Ferran Celades	Halborn	Ferran.Celades@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Uranium3o8 engaged Halborn to conduct a security assessment on their smart contract beginning on October 6th, 2023 and ending on October 17th, 2023. The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 ASSESSMENT SUMMARY

Halborn was provided one week for the engagement and assigned one full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified several vulnerabilities of varying severity in the smart contract code, which were mostly addressed by the Uranium3o8 team.

During the assessment, the following components and functionalities were scrutinized:

Redundant Logic in Smart Contract: It was observed that certain variables and logic in the contract, such as `startTime` and started, were redundant. Simplifying the logic can reduce the potential attack surface and make the contract more efficient.

Oracle Address Update: The ability to update the oracle address after the protocol has started was identified. This can lead to potential

inconsistencies and severe issues if the oracle address is set to an invalid or malicious address.

Tier Details Validation: The function setting tier details lacked proper validation for parameters like discount and tier values. Proper checks ensure that only valid values are set, reducing potential errors or vulnerabilities.

Stale Oracle Prices: The contract did not account for the possibility of stale prices from oracles. Using stale prices can lead to incorrect calculations, affecting the contract's key functionalities.

Centralization Risk with ETH Withdrawal: A function that allows the owner to withdraw all Ether from the contract was identified. This presents a centralization risk and could potentially empty the contract.

Ownership Transfer: The contract used Ownable instead of a more secure Ownable2Step. This could lead to the contract being left without an owner if ownership is wrongly transferred to an invalid address.

EIP20 Standard Compliance: Some tokens, like USDT, do not correctly implement the EIP20 standard. The contract did not account for this, which could lead to certain tokens being unusable in the protocol.

Reentrancy and Cause-Effect Pattern: Certain functions in the contract did not follow the cause-effect pattern, leading to potential reentrancy vulnerabilities.

Receive Function Oversight: The contracts' receive function did not increment the refundReserves, which could lead to discrepancies in the contract's state.

1.3 SCOPE

The assessment was scoped into the following smart contracts:

- src/ClaimingCenter.sol

- src/PurchasingCenter.sol
- src/Tiers.sol

Commit ID: ac6ab0f3c4d709531f1b20b441c50a70b2edeea6
Remediation commit ID: 634d477c7fd5f1c1f3e1b9e86464f571d623bf8e

Repository URL: https://github.com/u308/uranium-launchpad

1.4 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Testnet deployment (Foundry).

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	0	0	2	9

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) UNHANDLED STALE ORACLE PRICES	Critical (10)	SOLVED - 10/24/2023
(HAL-02) INACCURATE COMPUTATION OF COMMODITY AMOUNT EXCEEDS SELLING LIMITS	Critical (10)	SOLVED - 10/24/2023
(HAL-03) UNRESTRICTED ORACLE ADDRESS UPDATE	Low (4.4)	SOLVED - 10/24/2023
(HAL-04) CENTRALIZATION RISK IN ETH WITHDRAWAL	Low (2.8)	RISK ACCEPTED
(HAL-05) POTENTIAL OWNERLESS CONTRACT USING OWNABLE	Informational (1.6)	SOLVED - 10/24/2023
(HAL-06) RECEIVE FUNCTION NOT UPDATING REFUNDRESERVES	Informational (1.5)	SOLVED - 10/24/2023
(HAL-07) INCONSISTENT CLAIM LOGIC	Informational (1.3)	SOLVED - 10/24/2023
(HAL-08) REDUNDANT STATE VARIABLE	Informational (1.2)	ACKNOWLEDGED
(HAL-09) UNVALIDATED PARAMETER INPUTS	Informational (1.2)	SOLVED - 10/24/2023
(HAL-10) INCOMPATIBILITY WITH NON-STANDARD ERC20 TOKENS	Informational (1.0)	ACKNOWLEDGED
(HAL-11) POTENTIAL RE-ENTRANCY IN CLAIMREFUNDS FUNCTION	Informational (1.0)	SOLVED - 10/17/2023
(HAL-12) REDUNDANT STATE VARIABLES IN START STATUS	Informational (0.6)	ACKNOWLEDGED
(HAL-13) FUNCTION ALWAYS RETURNS TRUE	Informational (0.6)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) UNHANDLED STALE ORACLE PRICES - CRITICAL(10)

Description:

In the PurchasingCenter.sol contract, the ethPrice and getLatestData functions retrieve price data from an oracle, but do not implement checks to ensure that the returned price data is fresh and not stale. Oracle price feeds can become stale due to various reasons, such as network congestion, failure of data reporters, or oracle downtime. Utilizing a stale price can lead to incorrect calculations and mispriced transactions, affecting key functionalities of the contract and potentially leading to financial misappropriations or other unintended consequences.

BVSS:

AO:A/AC:L/AX:M/C:N/I:H/A:H/D:H/Y:H/R:N/S:C (10)

Recommendation:

To mitigate the risks associated with utilizing stale oracle prices, it is recommended to implement a check within the ethPrice and getLatestData functions to validate the freshness of the retrieved price data. This can involve checking the timestamp of the latest update and ensuring it is within an acceptable range before using the price data. If the price data is determined to be stale, the function should revert or handle the situation in a manner that prevents mispriced transactions and maintains the integrity of the contract's operations.

Here's a modified code snippet with a simple check for price staleness:

Listing 1

```
function ethPrice() public view returns (int256) {
    int256 price = eth_pricer.getLatestData();
    require(price != 0, "Stale price data");
    return price;
```

```
5 }
6
7 function getLatestData() public view returns (int) {
8     // prettier-ignore
9     (
10         ,
11         int answer,
12         ,
13         uint256 updatedAt,
14
15     ) = dataFeed.latestRoundData();
16     require(block.timestamp - updatedAt < acceptableDelay, "Stale
↳ price data");
17     return answer;
18 }
```

In the above snippet, acceptableDelay is a predefined duration (in seconds) that determines the maximum acceptable age of the price data. If the age of the price data exceeds acceptableDelay, the function reverts, preventing the usage of stale price data.

By implementing these checks, the PurchasingCenter.sol contract will safeguard against the risks associated with stale price data, ensuring that transactions and calculations are conducted with accurate and fresh data, thereby maintaining the integrity and reliability of the contract's operations.

Remediation Plan:

SOLVED: The Uranium3o8 team solved this issue. Price is being checked for different from zero and measures will be implemented on the external contract.

4.2 (HAL-02) INACCURATE COMPUTATION OF COMMODITY AMOUNT EXCEEDS SELLING LIMITS - CRITICAL(10)

Description:

In the PurchasingCenter.sol contract, the _amountToGet function, and consequently, its public counterpart amountToGet, can return an inaccurate amount of commodities when tier_oversubscribed is false. The issue arises from the fact that _amountToGet calculates the amount of commodities based on the current price, not the price at the time of the buy. This discrepancy can lead to the calculated amount being greater than the actual value at the time of purchase and, critically, the amount can be inflated if the ETH/USD price increases. This poses a significant risk as the calculated amount can exceed the amountOnSale for that tier, leading to inconsistencies and potential financial discrepancies in the contract's operation.

POC:

Listing 2

```
function test_amount_get_inflation_overpass() external {

        uint256 tier = 1;

        console.log("ETH price set to 2000");
        eth_consumer.mock_setPrice(2000);
        console.log("Commodity price set to 1");
        center.setCommodityPrice(1 * 1e8);

        center.startPurchase();

        (
            uint256 amountOnSale,
            ,
            ,
        ) = center.getTierDetails(tier);
```

```
17
18
19
20          console.log("Amount on sale:");
21          console.log(amountOnSale / 1e18);
22
23          console.log("GUY 1 buys commodity for the value of 4000
↳ ether on tier 1");
24          startHoax(GUY_1, 40 ether);
25          center.buyTokens{value: 40 ether}(tier);
26          vm.stopPrank();
27
28          console.log("Amount of commodities calculated for GUY 1:")
↳ ;
29          console.log(center.amountToGet(GUY_1, tier) / 1e18);
30          console.log("Total sold for the tier 1:");
31          console.log(center.tier_to_amountSold(Tiers.Tier.tier1) /
↳ 1e18);
32          console.log("Remaining amount to sale:");
33          console.log(center.getRemainingAmountByTier(tier) / 1e18);
34
35          console.log("Ethereum prices now surges to 4000");
36          eth_consumer.mock_setPrice(4000);
37
38          console.log("Amount of commodities calculated for GUY 1:")
↳ ;
39          console.log("============================");
40          console.log(center.amountToGet(GUY_1, tier) / 1e18);
41          console.log("============================");
42          console.log("Total sold for the tier 1:");
43          console.log(center.tier_to_amountSold(Tiers.Tier.tier1) /
↳ 1e18 );
44          console.log("Remaining amount to sale");
45          console.log(center.getRemainingAmountByTier(tier) / 1e18);
46
47          console.log("GUY 1 buys commodity for the value of 4 ether
↳  on tier 1, exhausting the available amount");
48          startHoax(GUY_1, 4 ether);
49          center.buyTokens{value: 4 ether}(tier);
50          vm.stopPrank();
51
52          console.log("Amount of commodities calculated for GUY 1 (
↳ using pro rata, as tier is oversubscribed):");
53          console.log("============================");
```

```
54          console.log(center.amountToGet(GUY_1, tier) / 1e18);
55          console.log("============================");
56          console.log("Total sold for the tier 1:");
57          console.log(center.tier_to_amountSold(Tiers.Tier.tier1) /
↳ 1e18 );
58          console.log("Remaining amount to sale");
59          console.log(center.getRemainingAmountByTier(tier) / 1e18);
60      }
```

Output:

Listing 3

```
1   ETH price set to 2000
2   Commodity price set to 1
3   Amount on sale:
4   87500
5   GUY 1 buys commodity for the value of 4000 ether on tier 1
6   Amount of commodities calculated for GUY 1:
7   80000
8   Total sold for the tier 1:
9   80000
10  Remaining amount to sale:
11  7500
12  Ethereum prices now surges to 4000
13  Amount of commodities calculated for GUY 1:
14  ============================
15  160000
16  ============================
17  Total sold for the tier 1:
18  80000
19  Remaining amount to sale
20  7500
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:H/Y:H/R:N/S:C (10)

Recommendation:

A more accurate and consistent approach to calculating the amount of commodities in the _amountToGet function could indeed leverage the already tracked price weighted contribution (pwc) and the pro-rata amount calculations.

In the _amountToGet function, instead of having two separate branches of logic (one for when tier_oversubscribed is true and another for when it is false), you could unify the logic using the _pro_rata_amount function. This function should calculate the output amount based on the tier_to_amountSold instead of amountOnSale, ensuring that the calculations are consistent with the actual state of the contract and the contributions made by the address guy.

Here's a simplified version of the _amountToGet function:

Listing 4

```
function _amountToGet(address guy, Tier tier) internal view
↳ returns (SD59x18 out) {
    uint256 amountSold = tier_to_amountSold[tier];
    SD59x18 _amountSold = sd(int256(amountSold) * sd59x18_decimals
↳ );
    uint256 pwc = address_to_tier_to_pwc[guy][tier];
    uint256 tier_pwc = tier_to_tierPwc[tier];
    SD59x18 _pwc = sd(int256(pwc) * sd59x18_decimals);
    SD59x18 _tier_pwc = sd(int256(tier_pwc) * sd59x18_decimals);
    out = _amountSold.mul(_pwc).div(_tier_pwc);
    return out;
}
```

In this modified function, _amountToGet calculates the output amount based on the tier_to_amountSold instead of amountOnSale, which should provide a more accurate and consistent calculation of the commodities to be received by guy. This approach ensures that the calculated amounts are in line with the actual contributions and the current state of the contract, preventing potential discrepancies due to oversubscription and ensuring a fair distribution of commodities based on the contributions and the price at the time of contribution. This also simplifies the logic

in `_amountToGet`, making the contract easier to understand and potentially reducing gas costs.

Remediation Plan:

SOLVED: The Uranium3o8 team solved this issue. There is no conspicuous consideration of whether a tier is oversubscribed or not in the form of “if” statements. The consideration is baked into the logic by the fact that whatever amount you’re getting is the amountSold * relative contribution.

4.3 (HAL-03) UNRESTRICTED ORACLE ADDRESS UPDATE - LOW (4.4)

Description:

In the PurchasingCenter.sol contract, the setEthUsdConsumerAddress function allows the owner to update the eth_usd_consumer_address at any time, even after the contract has been started. This could potentially introduce severe issues, as setting the oracle address to 0 or an invalid address could disrupt the contract's ability to retrieve accurate price data from the Chainlink oracle, thereby affecting any functionality that relies on this data. This could lead to inconsistencies, unintended behavior, or even exploitation if malicious actors gain control of the owner account.

BVSS:

AO:S/AC:L/AX:L/C:M/I:C/A:M/D:C/Y:C/R:N/S:C (4.4)

Recommendation:

To safeguard against unauthorized or unintended updates to the oracle address, it is recommended to restrict the ability to update the eth_usd_consumer_address once the contract has been started. This can be achieved by utilizing the whenNotStarted modifier in the setEthUsdConsumerAddress function, ensuring that the oracle address cannot be modified once the contract is active.

However, it is crucial to note that utilizing the whenNotStarted modifier will prevent updating the oracle address in the future, once the contract has started. This could be problematic if, for instance, the oracle address becomes obsolete or compromised.

To address this, consider implementing an additional function that allows the contract to verify the validity of the new oracle address before updating it, even after the contract has started. This function could

involve multi-signature approval, a timelock, or other mechanisms to ensure that updates to the oracle address are intentional and secure, thereby providing a balance between flexibility and security.

By implementing these recommendations, the PurchasingCenter.sol contract will prevent unauthorized or unintended updates to the oracle address, ensuring consistent and reliable interactions with the Chainlink oracle and safeguarding against potential exploitation and unintended behavior.

The modified code snippet might resemble:

Listing 5

```
function setEthUsdConsumerAddress(
    address _eth_usd_consumer_address
) external onlyOwner whenNotStarted {
    require(_eth_usd_consumer_address != address(0), "Address
↳ cannot be zero");
    // TODO: Additional checks to validate the new address could
↳ be added here
    eth_usd_consumer_address = _eth_usd_consumer_address;
    eth_pricer = IChainlinkPriceConsumer(_eth_usd_consumer_address
↳ );
    emit NewEthUsdConsumerAddressSet(_eth_usd_consumer_address);
}
```

Remediation Plan:

SOLVED: The Uranium3o8 team solved this issue. The code is now checking for address being different from zero. Moreover, the Uranium3o8 team stated that extra precaution will be taken when updating the address if ever needed.

4.4 (HAL-04) CENTRALIZATION RISK IN ETH WITHDRAWAL - LOW (2.8)

Description:

The withdrawEth function in the PurchasingCenter.sol and the withdrawEntireEthBalance under ClaimingCenter.sol contract allows the contract owner to withdraw all the Ether stored in the contract. This introduces a centralization risk, as it gives the contract owner the power to empty the contract's Ether balance at any time. Such a design can erode trust in the contract, as users might be concerned about the potential for malicious or unintended withdrawals that could jeopardize their funds or the contract's intended operations.

BVSS:

AO:S/AC:L/AX:L/C:N/I:H/A:H/D:H/Y:N/R:N/S:C (2.8)

Recommendation:

To mitigate this risk, consider implementing one or more of the following measures:

1. **Limit Withdrawal Amounts**: Instead of allowing the withdrawal of the entire balance, set a maximum limit or a daily withdrawal cap for the owner. This can prevent large, unexpected withdrawals.
2. **Timelock or Multi-signature Mechanism**: Implement a timelock mechanism where withdrawal requests are delayed by a certain period, allowing users to be alerted of a pending withdrawal. Alternatively, use a multi-signature mechanism where multiple parties must approve a withdrawal, adding another layer of security.
3. **Transparency**: If there's a legitimate reason for the owner to withdraw funds (e.g., for operational costs), communicate this clearly to the users and provide regular transparency reports detailing the withdrawals and their purposes.

4. **Decentralization**: Consider transitioning to a more decentralized governance model where key decisions, including fund withdrawals, are subject to community or stakeholder voting.
5. **Emergency Shutdown**: Implement an emergency shutdown mechanism that can halt certain contract activities in the event of detected malicious activities or vulnerabilities.

Remember, the goal is to ensure that users' funds are safe and that they can trust the contract's operations. The more transparent and decentralized the contract's governance and fund management are, the more trust it's likely to garner from its users.

Remediation Plan:

RISK ACCEPTED: The Uranium3o8 team accepted the risk of this finding. They did not perform any code change, and the `onlyOwner` modifier is the only way to restrict the access. The client stated the following: "We will use a multisig control over the owner role to guard against abuses of the function."

4.5 (HAL-05) POTENTIAL OWNERLESS CONTRACT USING OWNABLE - INFORMATIONAL (1.6)

Description:

The PurchasingCenter and ClaimingCenter contracts utilises the Ownable contract for ownership management, which could potentially result in the contract being left without an owner. If the ownership is transferred to an invalid or 0x0 address, it could permanently lock the contract management, preventing any further administrative changes or access to certain functionalities. This scenario not only poses a risk of losing control over the contract but also might expose the contract to further vulnerabilities or mismanagement of the funds or logic it controls.

BVSS:

AO:S/AC:L/AX:M/C:N/I:H/A:H/D:N/Y:N/R:N/S:C (1.6)

Recommendation:

To safeguard against the risks associated with potential ownerless contracts and to ensure secure ownership management, consider the following recommendations:

1. **Implement Safe Transfers**: Utilize Ownable2Step or a similar mechanism to ensure that ownership transfers are conducted securely and intentionally. This could involve a two-step process where an ownership transfer needs to be initiated and then confirmed, reducing the risk of accidental transfers.
2. **Recovery Mechanism**: Implement a mechanism that allows recovering ownership in case it was transferred to an invalid address. This could be a multi-signature scheme or a decentralized autonomous organization (DAO) that can vote to change the ownership in case of

mismanagement or accidental loss.

Incorporating these recommendations will enhance the security and robustness of the contract ownership management, reducing the risk of accidental loss of control over the contract.

Remediation Plan:

SOLVED: The Uranium3o8 team solved this issue. They are now using `UnrenounceableOwnable2Step` contract instead of the `Ownable`.

4.6 (HAL-06) RECEIVE FUNCTION NOT UPDATING REFUNDRESERVES - INFORMATIONAL (1.5)

Description:

The receive function in the ClaimingCenter contract is designed to accept incoming Ether transactions. However, it does not update the refundReserves state variable, which is intended to keep track of the total amount of Ether reserved for refunds.

This oversight means that any Ether sent directly to the contract (outside of the designed functions) will not be accounted for in the refundReserves . This can lead to discrepancies in the contract's internal accounting, potentially causing unexpected behavior when users try to claim refunds or when the contract checks for sufficient balances.

BVSS:

AO:S/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (1.5)

Recommendation:

To address this issue:

1. **Update refundReserves in the receive function**: Modify the receive function to increment the refundReserves by the received Ether amount. This ensures that the contract's internal accounting remains consistent.

Listing 6

```
receive() external payable {
    refundReserves += msg.value;
}
```

2. **Limit Direct Ether Transfers**: If the contract is not intended to accept direct Ether transfers (outside of specific functions), consider adding a revert statement in the `receive` function to reject such transfers. This ensures that Ether can only be sent to the contract through the intended functions.

Remediation Plan:

SOLVED: The Uranium3o8 team solved this issue. Both `receive()` and `fallback()` functions were modified with reverts.

4.7 (HAL-07) INCONSISTENT CLAIM LOGIC - INFORMATIONAL (1.3)

Description:

The `claim` function in the `ClaimingCenter` contract is designed to allow users to claim their tokens. However, the function's logic appears to be redundant and potentially misleading.

From the developers' intention, it's clear that a user should only be able to claim once. However, the function calculates the claimable amount twice, once with `claimableAmount` and then again with `getTokensBoughtByUserAndTier`. This redundancy can lead to confusion and potential errors in future modifications.

Furthermore, the requirement `lockupTime + startTime + purchaseWindow` seems to prevent the claim from ever happening, which contradicts the purpose of the function.

BVSS:

AO:S/AC:L/AX:M/C:N/I:H/A:N/D:N/Y:N/R:N/S:C (1.3)

Recommendation:

To streamline and clarify the `claim` function, use `claimableAmount` directly. Since the developers intended for the claim to be a one-time action, you can directly use the `claimableAmount` for the transfer. This removes the need for the second call to `getTokensBoughtByUserAndTier`. And then, for the transfer:

Listing 7

```
1 success = commodity.transfer(guy, claimableAmount);
```

Remediation Plan:

SOLVED: The Uranium3o8 team solved this issue. They did refactor the `claim` function to accommodate it to the expected behavior. The `claimableAmount` is not being used instead of the `getTokensBoughtByUserAndTier`.

4.8 (HAL-08) REDUNDANT STATE VARIABLE - INFORMATIONAL (1.2)

Description:

The smart contract PurchasingCenter.sol demonstrates a crucial aspect of decentralized applications, which involves interacting with an oracle to retrieve off-chain data, specifically, the Ethereum to USD price. However, there is a discernible inefficiency related to the storage and usage of the eth_usd_consumer_address state variable and a potential misjudgment of contract readiness due to the implementation of the everythingIsSet function. The state variable is used merely for instantiating the eth_pricer interface and is not otherwise utilized in the contract, leading to unnecessary storage costs. Moreover, the everythingIsSet function, intended to verify the contract's readiness for operations, checks the eth_usd_consumer_address without ensuring the operational status of the eth_pricer interface, potentially providing a false assurance of the contract's operational readiness.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:F/S:U (1.2)

Recommendation:

A streamlined and more reliable implementation of the PurchasingCenter.sol contract can be achieved by eliminating the redundant eth_usd_consumer_address state variable and enhancing the everythingIsSet function to verify the actual readiness of the contract for operations.

Firstly, the eth_pricer interface should be instantiated directly using the constructor and setter function parameters, and when the address of the oracle is needed for validations or comparisons, it can be derived directly using address(eth_pricer).

Secondly, the everythingIsSet function should be refined to ensure that

it provides a reliable indication of the contract's readiness. This can involve checking the validity of the eth_pricer interface, perhaps by validating its address or through a test query, in addition to the existing checks for commodityPrice and purchaseWindow.

The modified code snippet might resemble:

Listing 8

```
constructor(
    uint256 _purchaseWindow,
    uint256 _commodityPrice,
    address _eth_usd_consumer_address
) {
    started = false;
    purchaseWindow = _purchaseWindow;
    commodityPrice = _commodityPrice;
    eth_pricer = IChainlinkPriceConsumer(_eth_usd_consumer_address
↳ );
}

function setEthUsdConsumerAddress(
    address _new_eth_usd_consumer_address
) external onlyOwner {
    eth_pricer = IChainlinkPriceConsumer(
↳ _new_eth_usd_consumer_address);
    emit NewEthUsdConsumerAddressSet(_new_eth_usd_consumer_address
↳ );
}

function everythingIsSet() external view returns (bool) {
    return (
        commodityPrice != 0 &&
        purchaseWindow != 0 &&
        address(eth_pricer) != address(0)
    );
}
```

By implementing these recommendations, the PurchasingCenter.sol contract will utilize storage more efficiently and provide a more reliable indication of its operational readiness, ensuring robust and dependable interactions with the Chainlink oracle and other on-chain operations.

Remediation Plan:

ACKNOWLEDGED: The Uranium3o8 team acknowledged this issue. They did not change the code and left both variables stating the following: “The is modified to clean things up stylistically. the state variable is however not eliminated, to (1) to ensure visibility, (2) sort of keeping in line with Fraxs implementation.”

4.9 (HAL-09) UNVALIDATED PARAMETER INPUTS - INFORMATIONAL (1.2)

Description:

In the PurchasingCenter.sol contract, the setTierDetails function allows the contract owner to set various parameters for different tiers, including sellAmount, discount, lockupTime, and purchaseCap. However, the function does not perform checks to validate these input parameters, which could lead to unintended consequences. Specifically, the discount parameter, which is presumably meant to be a percentage, does not have checks to ensure it is within a valid range (0-100). Similarly, the tier parameter, which is used to determine which tier is being configured, is not validated to ensure it falls within an expected range (e.g., 1-4).

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:F/S:U (1.2)

Recommendation:

To enhance the robustness and reliability of the PurchasingCenter.sol contract, it is recommended to implement input validation checks within the setTierDetails function to ensure that parameters are within expected ranges and to prevent misconfigurations.

Listing 9

```
function setTierDetails(
    uint256 tier,
    uint256 sellAmount,
    uint256 discount,
    uint256 lockupTime,
    uint256 purchaseCap
) external onlyOwner whenNotStarted {
    require(tier >= 1 && tier <= 4, "Invalid tier");
    require(discount >= 0 && discount <= 100, "Invalid discount");
```

```
10      // Additional checks for other parameters can be added as
↳ needed
11
12      Tier _tier = _t(tier);
13      tier_to_tierDetails[_tier] = TierDetails(
14          sellAmount,
15          discount,
16          lockupTime,
17          purchaseCap
18      );
19      emit NewTierDetailsSet(
20          tier,
21          sellAmount,
22          discount,
23          lockupTime,
24          purchaseCap
25      );
26  }
```

By implementing these input validation checks, the PurchasingCenter.sol contract will prevent misconfigurations and ensure that tier details are set in a manner consistent with the intended logic and functionality of the contract, thereby safeguarding against potential issues and exploitation in the future.

Remediation Plan:

SOLVED: The Uranium3o8 team solved this issue. The code is now checking for all parameters being in range and valid.

4.10 (HAL-10) INCOMPATIBILITY WITH NON-STANDARD ERC20 TOKENS - INFORMATIONAL (1.0)

Description:

The contract assumes that all tokens adhere strictly to the EIP20 standard, particularly expecting a boolean return value from the transfer and transferFrom functions. However, some tokens, notably USDT, do not correctly implement the EIP20 standard. Their transfer and transferFrom functions return void instead of a boolean. As a result, when these non-standard tokens are used with the protocol, the transaction reverts due to the unexpected return value.

This incompatibility means that tokens like USDT and others that deviate from the EIP20 specification will be unusable within the protocol, potentially limiting its utility and adoption.

BVSS:

AO:S/AC:L/AX:M/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (1.0)

Recommendation:

To ensure compatibility with both standard and non-standard ERC20 tokens:

1. **Use OpenZeppelin's SafeERC20 Library**: Integrate the SafeERC20 library from OpenZeppelin. This library provides functions like safeTransfer and safeTransferFrom that handle tokens regardless of whether they return a boolean or not.

Listing 10

```
1 using SafeERC20 for IERC20;
```

And then, in the relevant functions:

Listing 11

```
IERC20(underlyingToken).safeTransferFrom(sender, recipient, amount
↳ );
```

2. **Test with Non-Standard Tokens**: After integrating SafeERC20, conduct thorough testing using both standard ERC20 tokens and known non-standard tokens like USDT. This will ensure that the protocol is compatible with a wide range of tokens.

By implementing these recommendations, the protocol will be more robust and versatile, accommodating a wider range of tokens and ensuring smoother interactions for users and other contracts.

Remediation Plan:

ACKNOWLEDGED: The Uranium3o8 team acknowledged this issue. The `transfer` function will be used as the token is trusted and known to be ERC-20 compatible.

4.11 (HAL-11) POTENTIAL RE-ENTRANCY IN CLAIMREFUNDS FUNCTION - INFORMATIONAL (1.0)

Description:

The claimRefunds function in the ClaimingCenter contract exhibits a potential re-entrancy vulnerability due to the cause-effect pattern not being strictly followed. Specifically, the function sends out the refund to the caller before updating the internal state (address_to_tier_eth_to_refund_claimed and tier_to_refunds_claimed).

If the purchasingCenter contract, which is called in the refundClaimable function, has any vulnerabilities or if it's manipulated to allow for re-entrancy, an attacker could potentially exploit this to inflate the refund amount they receive. This is because the state that tracks how much has been refunded is updated after the funds are sent, allowing for potential recursive calls.

The same pattern is also observed in the claim function where the state is updated after the transfer, which can also be a potential point of vulnerability.

BVSS:

AO:S/AC:L/AX:M/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (1.0)

Recommendation:

To mitigate this vulnerability, update the state before transfers. Always update the contract's state before making any external calls or transfers. This follows the Checks-Effects-Interactions pattern, which is a best practice in smart contract development to prevent re-entrancy attacks.

For the claimRefunds function, the state updates should be moved before

the external call:

Listing 12

```
address_to_tier_eth_to_refund_claimed[guy][_tier] = refundAmount;
tier_to_refunds_claimed[_tier] += refundAmount;
refundReserves -= refundAmount;
(success, ) = payable(guy).call{value: refundAmount}("");
require(success, "refund didn't go through!");
```

Remediation Plan:

SOLVED: The Uranium3o8 team solved this issue. The state updates are moved to before the commodity tokens are sent out.

4.12 (HAL-12) REDUNDANT STATE VARIABLES IN START STATUS - INFORMATIONAL (0.6)

Description:

In the PurchasingCenter.sol contract, the state variables startTime and started are utilized to manage and indicate the contract's start status, introducing a notable redundancy. Specifically, started is a boolean indicating whether the contract has been started, and startTime holds the timestamp of when the contract was started. The redundancy arises because the started variable can be implicitly determined by whether startTime is non-zero, rendering started unnecessary. This redundancy could introduce unnecessary storage costs and potential complexity in contract logic.

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:F/S:U (0.6)

Recommendation:

A more efficient and streamlined implementation can be achieved by eliminating the started state variable and utilizing the startTime variable to implicitly determine whether the contract has been started. Specifically, if startTime is non-zero, it indicates that the contract has been started. This approach simplifies the contract logic and reduces storage costs.

The modified code snippet might resemble:

Listing 13

```
modifier whenNotStarted() {
    require(startTime == 0, "Started already!");
    _;
}
```

```
 5 modifier whenStarted() {
 6     require(startTime != 0, "Not started yet!");
 7     _;
 8 }
 9
10 function startPurchase() public onlyOwner whenNotStarted
↳ everythingSet {
11     // once started it's impossible unstart / pause.
12     startTime = block.timestamp;
13     emit PurchaseStarted(startTime);
14 }
```

By implementing this recommendation, the PurchasingCenter.sol contract will utilize storage more efficiently and provide a more straightforward logic regarding the contract's start status, ensuring a cleaner and potentially less error-prone codebase.

Remediation Plan:

ACKNOWLEDGED: The Uranium3o8 team acknowledged this issue.

4.13 (HAL-13) FUNCTION ALWAYS RETURNS TRUE - INFORMATIONAL (0.6)

Description:

In the PurchasingCenter.sol contract, the _bought function is designed to handle logic related to purchasing across different tiers and to track whether a particular address has already bought in a specific tier. However, the function is structured in a way that it always returns true regardless of the actual buying status or whether the operation was successful, which could mislead other functions or external calls that rely on the return value of _bought to determine the success or status of a purchase operation.

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:F/S:U (0.6)

Recommendation:

Given the context of the buyTokens function where _bought is used, it seems like the return value of _bought is utilized to inform the caller of buyTokens about the purchase status. However, considering that _bought always returns true or reverts in the current implementation, the return value might not be providing meaningful information to the callers.

In this case, it might be beneficial to remove the return value of _bought unless it provides a necessary indication of the function's operation that cannot be deduced from the state changes or emitted events. If the function's success can be inferred from the absence of a revert (which is often the case in Ethereum smart contracts), then the return value might be omitted to save gas and simplify the function interface.

Listing 14

```
function buyTokens(address guy, uint256 tier) external returns (
↳ bool) {
    ...
    _bought(guy, tier); // update the buy details.
    emit BoughtTokens(tier, guy, eth_in);
    return true; // Indicate that tokens were bought successfully
}
```

In this scenario, if _bought completes its execution without reverting, it indicates that the operation was successful. If it was not successful (e.g., if an invalid tier was provided), it would revert and the BoughtTokens event would not be emitted, thereby providing a clear indication of the function's outcome without the need for a return value. This approach simplifies the contract logic and can save gas.

Remediation Plan:

ACKNOWLEDGED: The Uranium3o8 team acknowledged this issue.

REVIEW NOTES

5.1 Tiers.sol

- The `_t` public pure function does convert a numerical value to a solidity enum type. Only 4 tiers exist.

5.2 PurchasingCenter.sol

- It is using the https://github.com/PaulRBerg/prb-math library for anything related to prices. The underlying type is `int256`.
- The constructor does set the `purchaseWindow`, `commodityPrice` and `eth_pricer/eth_usd_consumer_address`. Moreover, it does also set the following tear details:

Table 1: Tier details for the constructor

Tier	Amount on Sale	Discount	Lockup Time	Purchase Cap
Tier 1	87,500	0	2 weeks	100
Tier 2	75,000	10	4 weeks	100
Tier 3	50,000	20	12 weeks	75
Tier 4	37,500	30	24 weeks	50

- The `startPurchase` function can only be called by the owner, it does set the start time to the current `block.timestamp` and does also check if was not already started with the `whenNotStarted`. The `whenStarted` modifier will prevent from anyone calling the `buyTokens` function without having the `started` set.
- The `setPurchaseWindow`, `setCommodityPrice` and `setTierDetails` functions can be called only when the buy has not started. However, the `setEthUsdConsumerAddress` function can be called after the protocol has been started. **This is a discrepancy on the implementation, as all 3 values are being set on `everythingSet`, but the latter can be changed to zero address later on.**
- The `discountedPrice` function does correctly unwrap the type and

divides by the sd59x18_decimals factor.

- ethInCommodityOut does correctly factor all values, including price feed. **Commodity price and feed price should have the same amount of decimals. Otherwise, factors may be miscalculated.**
- **The internal _price_weighted_contribution function does calculate the weighted amount based on the current ether price. The new_contribution is a wei based value, factor of 1e18, meanwhile the eth_price is a factor of 1e8. This means that to get the true value, it should be divided by 1e18 * 1e8 = 1e26. For example, if a 1 ether contribution was made, assuming a 2000 USD ether price, the returned pwc corresponds to 2e29.**
- The getTierPwc does remove the 1e8 factor from the pwc total. This means that the price factor is being removed here and the weighted contribution returned. Same goes for the getPwcByUserAndTier function.
- The totalContribution does return the total amount contributed, in wei for a given account.
- The getRemainingAmountByTier does return the remaining amount until the amountOnSale for that tier is reached. Since the tier_to_amountSold cannot get passed the total tier amountOnSale, oversubscribed will return 0. However, the tierContribution function will return the true contributed ethers and the getAmountSoldByTier the actual amount sold, capped to a max value of amountOnSale for that tier. The getContributionByUserAndTier function is the same as the tierContribution but for a given user address.
- The buyTokens does verify the following:
  - The purchase amount does not exceed the purchase cap amount by adding previous tier contributed amount and the current one.
  - It allows calling the function after the amountOnSale has been reached, allowing for the contribution for that tier to be oversubscribed.. The eth_in amount is only increased on address_to_tier_to_contribution and tier_to_totalContribution.
- The internal _bought function, called at the end of buyTokens will push the buyer to the tier buyers list and return always true.
- The _pro_rata_amount function could have a division by zero if tier_to_tierPwc is zero. However, the function is only called inside

the _amountToGet when tier_oversubscribed is true. This means that the tier_to_tierPwc can never be zero in that case.

    - The function does use address_to_tier_to_pwc and tier_to_tierPwc, both values factored on 1e8 those the returning value does clear the price factor from the _amountOnSale.

- The _amountToGet will return a different value depending on the tier_oversubscribed. If oversubscribed, the pro-rated value is returned.

5.3 ClaimingCenter.sol

- The claimRefunds function has a re-entrancy vulnerability as it is calling the sender. However, a nonReentrant modifier is being used. It has unnecessary checks after the success require.
- The refundClaimable does use the purchasing center refundAmount view function, which does give the difference between the contributed amount and the extra amount in case of oversubscribed tiers.
- The claimable function which uses the public getTokensBoughtByUserAndTier which internally uses the amountToGet from the purchasing center, as already stated, the amount can actually be greater than the selling amount. The returned amount is the result of the getTokensBoughtByUserAndTier minus the tracked address_to_tier_to_tokens_claimed (increased on the claim function after a successful call).
- The claim function can only be called once, this means that if a user does call claim but then buys more tokens he will not be able to claim those extra ever again.

THANK YOU FOR CHOOSING

// HALBORN