CredShields

Smart Contract Audit

Aug 13th, 2024 • CONFIDENTIAL

Description

This document details the process and result of the Plutope Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Plutope between Aug 5th, 2024, and Aug 9th, 2024. A retest was performed on Aug 12th, 2024.

Author

Shashank (Co-founder, CredShields) - shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead), Shreyas Koli (Auditor), Naman Jain (Auditor), Sanket Salavi (Auditor)

Prepared for

Plutope

Table of Contents

1. Executive Summary

Plutope engaged CredShields to perform a smart contract audit from Aug 5th, 2024 to Aug 9th, 2024. During this timeframe, 12 vulnerabilities were identified. **A retest was performed on Aug 12th, 2024, and all the bugs have been addressed.**

During the audit, 2 vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Plutope" and should be prioritized for remediation, and fortunately, none were found.

The table below shows the in-scope assets and a breakdown of findings by severity per asset. Section 2.3 contains more information on how severity is calculated.

Assets in Scope	Critical	High	Medium	Low	info	Gas	Σ
Plutope Smart Contracts	2	0	5	1	2	2	**12**
	2	**0**	**5**	**1**	**2**	**2**	**12**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in the Plutope Smart Contract's scope during the testing window while abiding by the policies set forth by the Plutope team.

State of Security

To maintain a robust security posture, it is essential to continuously review and improve upon current security processes. Utilizing CredShields' continuous audit feature allows both Plutope's internal security and development teams to not only identify specific vulnerabilities but also gain a deeper understanding of the current security threat landscape.

To ensure that vulnerabilities are not introduced when new features are added, or code is refactored, we recommend conducting regular security assessments. Additionally, by analyzing the root cause of resolved vulnerabilities, the internal teams at Plutope can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, Plutope can future-proof its security posture and protect its assets.

2. Methodology

Plutope engaged CredShields to perform a Plutope Smart Contract audit. The following sections cover how the engagement was put together and executed.

2.1 Preparation phase

The CredShields team meticulously reviewed all provided documents and comments in the smart contract code to gain a thorough understanding of the contract's features and functionalities. They meticulously examined all functions and created a mind map to systematically identify potential security vulnerabilities, prioritizing those that were more critical and business-sensitive for the refactored code. To confirm their findings, the team deployed a self-hosted version of the smart contract and performed verifications and validations during the audit phase.

A testing window from Aug 5th, 2024 to Aug 9th, 2024 was agreed upon during the preparation phase.

2.1.1 Scope

During the preparation phase, the following scope for the engagement was agreed upon:

IN SCOPE ASSETS
• https://github.com/PlutopeIn/PLT/tree/b7401450b2c91d804ceba428205d7a2389e6aef0 • https://github.com/PlutopeIn/PLT/tree/961c2ffa7ccdddd156ba7db665bfecce23982250

Table: List of Files in Scope

2.1.2 Documentation

Documentation was not required as the code was self-sufficient for understanding the project.

2.1.3 Audit Goals

CredShields uses both in-house tools and manual methods for comprehensive smart contract security auditing. The majority of the audit is done by manually reviewing the contract source code, following SWC registry standards, and an extended industry standard self-developed checklist. The team emphasizes understanding core concepts, preparing test cases, and evaluating business logic for potential vulnerabilities.

2.2 Retesting phase

Plutope is actively partnering with CredShields to validate the remediations implemented towards the discovered vulnerabilities.

2.3 Vulnerability classification and severity

CredShields follows OWASP's Risk Rating Methodology to determine the risk associated with discovered vulnerabilities. This approach considers two factors - Likelihood and Impact - which are evaluated with three possible values - **Low**, **Medium**, and **High**, based on factors such as Threat agents, Vulnerability factors, and Technical and Business Impacts. The overall severity of the risk is calculated by combining the likelihood and impact estimates.

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

Overall, the categories can be defined as described below -

1. **Informational**

   We prioritize technical excellence and pay attention to detail in our coding practices. Our guidelines, standards, and best practices help ensure software stability and

reliability. Informational vulnerabilities are opportunities for improvement and do not pose a direct risk to the contract. Code maintainers should use their own judgment on whether to address them.

2. Low

Low-risk vulnerabilities are those that either have a small impact or can't be exploited repeatedly or those the client considers insignificant based on their specific business circumstances.

3. Medium

Medium-severity vulnerabilities are those caused by weak or flawed logic in the code and can lead to exfiltration or modification of private user information. These vulnerabilities can harm the client's reputation under certain conditions and should be fixed within a specified timeframe.

4. High

High-severity vulnerabilities pose a significant risk to the Smart Contract and the organization. They can result in the loss of funds for some users, may or may not require specific conditions, and are more complex to exploit. These vulnerabilities can harm the client's reputation and should be fixed immediately.

5. Critical

Critical issues are directly exploitable bugs or security vulnerabilities that do not require specific conditions. They often result in the loss of funds and Ether from Smart Contracts or users and put sensitive user information at risk of compromise

or modification. The client's reputation and financial stability will be severely impacted if these issues are not addressed immediately.

6. **Gas**

   To address the risk and volatility of smart contracts and the use of gas as a method of payment, CredShields has introduced a "Gas" severity category. This category deals with optimizing code and refactoring to conserve gas.

2.4 CredShields staff

The following individual at CredShields managed this engagement and produced this report:

- **Shashank, Co-founder CredShields**
    - shashank@CredShields.com

Please feel free to contact this individual with any questions or concerns you have about the engagement or this document.

3. Findings

This chapter contains the results of the security assessment. Findings are sorted by their severity and grouped by the asset and SWC classification. Each asset section will include a summary. The table in the executive summary contains the total number of identified security vulnerabilities per asset per risk indication.

3.1 Findings Overview

3.1.1 Vulnerability Summary

During the security assessment, 12 security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Buyer passing zero amount will block token purchases causing DoS for other buyers	Critical	Missing Input Validation
Missing buyOption validation in buyPrivateSaleToken() can leads to loss of funds	Critical	Missing Input validation
Admin passing zero amount will block token distribution causing DoS for Advisors and Founders	Medium	Missing Input Validation
User balance checks can prevent token purchases, causing DoS for token buyers	Medium	Denial of Service

Admin can reset private sale phase timings	Medium	Logic Error
Chainlink Oracle Min/Max price validation	Medium	Incorrect Validation
Missing Price Feed Validation	Medium	Incorrect Validation
Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	Missing Best Practices
Incorrect Error Message	Informational	Incorrect Statement
Missing Index Validation	Informational	Missing Input Validation
Cheaper Inequalities in if()	Gas	Gas Optimization
Cheaper Inequalities in require()	Gas	Gas Optimization

Table: Findings in Smart Contracts

3.1.2 Findings Summary

SWC ID	SWC Checklist	Test Result	Notes
SWC-100	Function Default Visibility	Not Vulnerable	Not applicable after v0.5.X (Currently using solidity v >= 0.8.6)
SWC-101	Integer Overflow and Underflow	Not Vulnerable	The issue persists in versions before v0.8.X.
SWC-102	Outdated Compiler Version	Not Vulnerable	Version 0^.8.0 and above is used
SWC-103	Floating Pragma	Not Vulnerable	The contract does not use floating pragma
SWC-104	Unchecked Call Return Value	Not Vulnerable	call() is not used
SWC-105	Unprotected Ether Withdrawal	Not Vulnerable	Appropriate function modifiers and require validations are used on sensitive functions that allow token or ether withdrawal.
SWC-106	Unprotected SELFDESTRUCT Instruction	Not Vulnerable	selfdestruct() is not used anywhere
SWC-107	Reentrancy	Not Vulnerable	No notable functions were vulnerable to it.
SWC-108	State Variable Default Visibility	Not Vulnerable	Not Vulnerable
SWC-109	Uninitialized Storage Pointer	Not Vulnerable	Not vulnerable after compiler version, v0.5.0

SWC-110	Assert Violation	Not Vulnerable	Asserts are not in use.
SWC-111	Use of Deprecated Solidity Functions	Not Vulnerable	None of the deprecated functions like block.blockhash(), msg.gas, throw, sha3(), callcode(), suicide() are in use
SWC-112	Delegatecall to Untrusted Callee	Not Vulnerable	Not Vulnerable.
SWC-113	DoS with Failed Call	Not Vulnerable	No such function was found.
SWC-114	Transaction Order Dependence	Not Vulnerable	Not Vulnerable.
SWC-115	Authorization through tx.origin	Not Vulnerable	tx.origin is not used anywhere in the code
SWC-116	Block values as a proxy for time	Not Vulnerable	Block.timestamp is not used
SWC-117	Signature Malleability	Not Vulnerable	Not used anywhere
SWC-118	Incorrect Constructor Name	Not Vulnerable	All the constructors are created using the constructor keyword rather than functions.
SWC-119	Shadowing State Variables	Not Vulnerable	Not applicable as this won't work during compile time after version 0.6.0
SWC-120	Weak Sources of Randomness from Chain Attributes	Not Vulnerable	Random generators are not used.
SWC-121	Missing Protection against Signature Replay Attacks	Not Vulnerable	No such scenario was found

SWC-122	Lack of Proper Signature Verification	Not Vulnerable	Not used anywhere
SWC-123	Requirement Violation	Not Vulnerable	Not vulnerable
SWC-124	Write to Arbitrary Storage Location	Not Vulnerable	No such scenario was found
SWC-125	Incorrect Inheritance Order	Not Vulnerable	No such scenario was found
SWC-126	Insufficient Gas Griefing	Not Vulnerable	No such scenario was found
SWC-127	Arbitrary Jump with Function Type Variable	Not Vulnerable	Jump is not used.
SWC-128	DoS With Block Gas Limit	Not Vulnerable	Not Vulnerable.
SWC-129	Typographical Error	Not Vulnerable	No such scenario was found
SWC-130	Right-To-Left-Override control character (U+202E)	Not Vulnerable	No such scenario was found
SWC-131	Presence of unused variables	Not Vulnerable	No such scenario was found
SWC-132	Unexpected Ether balance	Not Vulnerable	No such scenario was found
SWC-133	Hash Collisions With Multiple Variable Length Arguments	Not Vulnerable	abi.encodePacked() or other functions are not used.
SWC-134	Message call with hardcoded gas amount	Not Vulnerable	Not used anywhere in the code
SWC-135	Code With No Effects	Not Vulnerable	No such scenario was found
SWC-136	Unencrypted Private Data On-Chain	Not Vulnerable	No such scenario was found

4. Remediation Status

Plutope is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on Aug 12th, 2024, and all the issues have been addressed.**

Also, the table shows the remediation status of each finding.

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Buyer passing zero amount will block token purchases causing DoS for other buyers	Critical	**Fixed [12/08/2024]**
Missing buyOption validation in buyPrivateSaleToken() can leads to loss of funds	Critical	**Fixed [12/08/2024]**
Admin passing zero amount will block token distribution causing DoS for Advisors and Founders	Medium	**Fixed [12/08/2024]**
User balance checks can prevent token purchases, causing DoS for token buyers	Medium	**Fixed [12/08/2024]**
Admin can reset private sale phase timings	Medium	**Fixed [12/08/2024]**
Chainlink Oracle Min/Max price validation	Medium	**Fixed [12/08/2024]**
Missing Price Feed Validation	Medium	**Fixed [12/08/2024]**

Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	**Fixed [12/08/2024]**
Incorrect Error Message	Informational	**Fixed [12/08/2024]**
Missing Index Validation	Informational	**Fixed [12/08/2024]**
Cheaper Inequalities in if()	Gas	**Fixed [12/08/2024]**
Cheaper Inequalities in require()	Gas	**Fixed [12/08/2024]**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID #1 [Fixed]

Buyer passing zero amount will block token purchases causing DoS for other buyers

Vulnerability Type
Missing Input Validation

Severity
Critical

Description
The function `buyPrivateSaleToken()` allows users to purchase private sale tokens either with USDT or BNB. However, the function lacks validation for the `amount` parameter when using USDT or the `msg.value` when using BNB. If a buyer passes a zero value for `amount` USDT or sends zero BNB, then `tokenValue` becomes zero. Then it internally calls `check_And_Change_SuppliedToken()` function which sets `totalSuppliedToken` to `totalPhaseToken` because `tokenValue` was zero, marking the phase as fully distributed. This flaw prevents other buyers from purchasing tokens in that phase, as the sum of `amount` and `totalSuppliedToken` will be greater than the `totalPhaseToken` and this will revert with error `INSUFFICIENT_TOKENS_IN_PHASE.`

check_And_Change_SuppliedToken():

```
if (amt != 0) {
        require(
          distributeInfo.totalSuppliedToken + amt <=
            distributeInfo.totalPhaseToken,
          "INSUFFICIENT_TOKENS_IN_PHASE"
        );
```

```
        distributeInfo.totalSuppliedToken += amt;
@>    } else {
        require(
          distributeInfo.totalSuppliedToken <
            distributeInfo.totalPhaseToken,
          "NO_TOKENS_LEFT_IN_THIS_PHASE"
        );
@>        distributeInfo.totalSuppliedToken = distributeInfo.totalPhaseToken; //@audit
```

Affected Code

- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L605-L687

Impacts

If a buyer inputs zero for the `amount` or sends zero BNB, the contract will incorrectly mark the token phase as fully distributed, blocking any further purchases. This disrupts the private sale process and can prevent legitimate buyers from participating.

Remediation

To fix this issue, the function should validate that the `tokenValue` is greater than zero before proceeding with the purchase. This can be done by adding a requirement check such as:

```
function buyPrivateSaleToken(
      uint256 buyOption,
      uint256 amount
   ) public payable nonReentrant {

      ...
+     require(tokenValue > 0, "INVALID_AMOUNT");
      check_And_Change_SuppliedToken(tokenPhaseIndex, tokenValue);
      token_Sold = token_Sold + tokenValue;
```

```
    ...
}
```

Test Case:

```
it("Exploit: Freezing in buyPrivateSaleToken() function", async function () {

        // resetting the private sale time
        const start = await PlutopePresale.startPrivateSale();
        await start.wait();

        //minting alice USDT token
        const amount = "1000000000";
        await usdtToken.connect(alice).mint();
        await usdtToken.connect(alice).approve(PlutopePresale.target, amount);

        // Fetch Distribution data before first buy
        const distributionDataBefore = await PlutopePresale.getTokonomicsData();
        console.log("total phase token before first buy: ",
distributionDataBefore[0].totalPhaseToken);
        console.log("total supplied token before first buy: ",
distributionDataBefore[0].totalSuppliedToken);

        // initialiting first buy
        await PlutopePresale.connect(alice).buyPrivateSaleToken(0, 0);
        console.log("\nAlice just buyed! \n");

        // fetching alice's balance
        const buyerData = await PlutopePresale.getPrivateSaleBuyerInfo(alice.address);
        console.log("alice's plt token data: ", buyerData[0].pltToken);

        // minting bob USDT Token
        await usdtToken.connect(bob).mint();
        await usdtToken.connect(bob).approve(PlutopePresale.target, amount);


        // Fetch Distribution data after first buy
```

```
        const distributionDataAfter = await PlutopePresale.getTokonomicsData();
        console.log("token supplied after 2nd buy: ",
distributionDataAfter[0].totalPhaseToken);
        console.log("token supplied after 2nd buy: ",
distributionDataAfter[0].totalSuppliedToken);

        expect(buyerData[0].pltToken +
distributionDataBefore[0].totalSuppliedToken).to.be.not.equal(distributionDataAfter[0].to
talSuppliedToken)

expect(distributionDataAfter[0].totalSuppliedToken).to.be.equal(distributionDataAfter[0].
totalPhaseToken)
        await expect(PlutopePresale.connect(bob).buyPrivateSaleToken(0,
amount)).to.be.rejectedWith("INSUFFICIENT_TOKENS_IN_PHASE");

        console.log("\nExploit Success: totalPhaseToken is equal to totalSuppliedToken
after first alice buy with zero (0) amount!\n")
    });
```

Retest

This vulnerability has been fixed by adding a zero-amount validation at the beginning. However, it is still recommended to add a validation before calling the check_And_Change_SuppliedToken() function to make sure that the tokenValue is not equal to zero (0) [as suggested before].

Bug ID # 2 [Fixed]

Missing `buyOption` validation in `buyPrivateSaleToken()` can leads to loss of funds

Vulnerability Type
Missing Input Validation

Severity
Critical

Description
The function `buyPrivateSaleToken()` allows users to purchase private sale tokens by specifying a `buyOption` parameter that should either be `0` (for USDT) or `1` (for BNB). However, the function lacks validation to ensure that `buyOption` is restricted to these two values. If an attacker inputs a value other than `0` or `1` for `buyOption`, the function skips the necessary checks and calculates the token value based on an unchecked and untransferred `amount`.
This bypass allows the attacker to set a very large value for the `amount`, leading to the issuance of a disproportionate number of tokens. The attacker can exploit this flaw to obtain all the private sale tokens at no cost, effectively draining the token supply and leaving legitimate participants unable to purchase tokens.

Scenario

1. Bob discovers that the `buyPrivateSaleToken()` function lacks proper validation for the `buyOption` parameter.
2. Bob decides to exploit this vulnerability by sending a transaction with an invalid `buyOption` value, such as `999`, and a large `amount` value, for instance, `1000000`.
3. Because the function does not validate the `buyOption` value, it skips the payment checks and calculates the `tokenValue` based on the large `amount` Bob specified.
4. The function then processes the transaction as if Bob had paid for the tokens, issuing him a massive number of tokens at no cost.

5. As a result, Bob acquires all the available tokens in the private sale, depleting the supply and preventing any legitimate buyers from purchasing tokens.

Affected Code

- https://github.com/PlutopeIn/PLT/blob/961c2ffa7ccdddd156ba7db665bfecce23982250/contracts/PlutopePresale.sol#L629-L717

Impacts

By exploiting the missing validation for the `buyOption` parameter, an attacker can acquire all available private sale tokens for free, causing significant financial loss to both the token issuer and other participants.

Remediation

To mitigate this issue, the `buyPrivateSaleToken` function should include strict validation to ensure that the `buyOption` parameter is either `0` or `1`. This can be done by adding a requirement check such as:

```
require(buyOption == 0 || buyOption == 1, "INVALID_BUY_OPTION");
```

Test Case:

```
it("Should get pltToken without transferring any amount from bob", async function () {

        amount = "1000000000000000000";

        const tokenValue = BigInt(amount * 10000 / 670);
        console.log("calculated token value: ", tokenValue);

        await plutopePresale.connect(bob).buyPrivateSaleToken(3, amount);

        const buyerData = await plutopePresale.getPrivateSaleBuyerInfo(bob.address);

        const balance = buyerData[0].pltToken;
```

```
    console.log("bob's balance:              ", balance);
});
```

Retest

This vulnerability has been fixed by implementing validation on the buyOption parameter.

Bug ID #3 [Fixed]

Admin passing zero amount will block token distribution causing DoS for Advisors and Founders

Vulnerability Type
Missing Input Validation

Severity
Medium

Description
The functions `addAdvisorToken()` and `addFounderToken()` allow the contract owner to allocate tokens to advisors and founders. However, neither function validates that the `amount` parameter is greater than zero. When a zero value of `amount` is passed in the internally called `check_And_Change_SuppliedToken()` function which sets `totalSuppliedToken` to `totalPhaseToken` because `amount` was zero, marking the phase as fully distributed. This flaw prevents the Owner from adding tokens for the adviser and founder, as the sum of `amount` and `totalSuppliedToken` will be greater than the `totalPhaseToken` and this will revert with the error `INSUFFICIENT_TOKENS_IN_PHASE.`

Affected Code
- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L178-L208
- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L210-L237

Impacts
This vulnerability impacts advisors and founders by potentially blocking their ability to receive allocated tokens if the contract owner mistakenly passes a zero value as the amount. Once the `totalSuppliedToken` is set to `totalPhaseToken`, the phase is considered fully distributed, and no further tokens can be claimed.

Remediation

To fix this issue, the `addAdvisorToken` and `addFounderToken` functions should include a validation check to ensure the `amount` parameter is greater than zero. This can be done by adding a requirement check such as:

```
require(amount > 0, "INVALID_AMOUNT");
```

Test Case:

```
it("Should DoS for Founder and Advisor when amount is zero (0)", async function(){

        // DoS for Founder
        const FounderdistributionDataBefore = await
PlutopePresale.getTokonomicsData();
        console.log("Total Phase token of Founder before: ",
FounderdistributionDataBefore[4].totalPhaseToken);
        console.log("Total Phase token of Founder before: ",
FounderdistributionDataBefore[4].totalSuppliedToken);

        await PlutopePresale.connect(owner).addFounderToken(0, Founder.address);

        const FounderdistributionDataAfter = await PlutopePresale.getTokonomicsData();

expect(FounderdistributionDataAfter[4].totalPhaseToken).to.be.equal(Founderdistributio
nDataAfter[4].totalSuppliedToken)

        console.log("\nTotal Phase token of Founder after: ",
FounderdistributionDataAfter[4].totalPhaseToken);
        console.log("Total Phase token of Founder after: ",
FounderdistributionDataAfter[4].totalSuppliedToken);

        //DoS for Advisor
        const AdvisordistributionDataBefore = await PlutopePresale.getTokonomicsData();
```

```
        console.log("\nTotal Phase token of Advisor before: ",
AdvisordistributionDataBefore[6].totalPhaseToken);
        console.log("Total Phase token of Advisor before: ",
AdvisordistributionDataBefore[6].totalSuppliedToken);

        await PlutopePresale.connect(owner).addAdvisorToken(0, Advisor.address);

        const AdvisordistributionDataAfter = await PlutopePresale.getTokonomicsData();

expect(AdvisordistributionDataAfter[6].totalPhaseToken).to.be.equal(Advisordistribution
DataAfter[6].totalSuppliedToken)

        console.log("\nTotal Phase token of Advisor after: ", AdvisordistributionDataAfte
r[6].totalPhaseToken);
        console.log("Total Phase token of Advisor after: ",
AdvisordistributionDataAfter[6].totalSuppliedToken, "\n");

    });
```

Retest

This vulnerability has been fixed by adding Zero Amount validation.

Bug ID #4 [Fixed]

User balance checks can prevent token purchases, causing DoS for token buyers

Vulnerability Type
Denial of Service

Severity
Medium

Description
The function `buyPrivateSaleToken()` includes a require validation to ensure that the `msg.sender`'s balance of USDT tokens is greater than the `amount` they intend to use for purchasing private sale tokens. Specifically, the condition `usdt_Address.balanceOf(msg.sender) > amount` is used to validate the buyer's balance. However, this condition inadvertently causes a denial of service (DoS) when the user's balance is exactly equal to the amount being spent. In such a case, the transaction will revert, preventing the user from buying private sale tokens even though they have sufficient funds.

Affected Code

- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L635

Impact
Users who have a USDT balance that matches the exact amount they intend to spend on purchasing private sale tokens will be unable to complete the transaction and this prevents them from buying private sale tokens.

Remediation
To fix this issue, the balance check should be adjusted to allow purchases when the user's balance is exactly equal to the amount they wish to spend.

```
require(usdt_Address.balanceOf(msg.sender) >= amount, "INSUFFICIENT_TOKENS");
```

Test Case:

```
it("Exploit: Should fail due to equal balance of USDT and amount", async function () {
        //Start Private Sale
        const start = await PlutopePresale.startPrivateSale();
        await start.wait();

        //mint USDT token to alice
        const amount = "1000000000"; //change USDT mint value from USDT.sol
        await usdtToken.connect(alice).mint();
        await usdtToken.connect(alice).approve(PlutopePresale.target, amount2);

        //Buying private token
        const buy = await PlutopePresale.connect(alice).buyPrivateSaleToken(0, amount);
        expect (await buy.wait()).to.be.revertedWith("INSUFFICIENT_TOKENS");
    });
```

Retest

This vulnerability has been fixed by allowing the user to purchase when the amount is equal to their balance.

Bug ID #5 [Fixed]

Admin can reset private sale phase timings

Vulnerability Type
Logic Error

Severity
Medium

Description
The function `startPrivateSale()` allows the contract owner to initiate a private sale and set the phase times. However, the function does not check whether the private sale is already active before resetting the phase times. This means that if the private sale is already running, the contract owner can call this function again, inadvertently resetting the private sale phase times to new values based on the current time. This lack of state management can lead to disruption in the private sale process, affecting participants who are relying on the previously set schedule.

Affected Code

- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L524-L532

Impacts
This vulnerability impacts participants in the private sale by allowing the contract owner to reset the phase times, disrupting the expected schedule unintentionally. Participants may miss out on planned purchase opportunities or find the sale periods extended.

Remediation
To fix this issue, the `startPrivateSale()` function should include a check to ensure that this function cannot be called more than once. This can be done by adding checks such as:

```
+ bool public _privateSaleInitialized = false;
```

```
function startPrivateSale() public onlyOwner {


+     if(!_privateSaleInitialized) {
+         _privateSaleInitialized = true;
+     } else {
          revert("Already Initialized!");
      }

      privateSale_Status = true;
      privateSale_PhaseTime = [
         getCurrentTime() + 1 days,
         getCurrentTime() + 2 days,
         getCurrentTime() + 3 days
      ];
      emit PrivateSaleStatus(privateSale_Status);
   }
```

Retest

This vulnerability has been fixed by adding a validation which ensures that the function cannot be called again.

Bug ID #6 [Fixed]

Chainlink Oracle Min/Max price validation

Vulnerability Type
Incorrect Validation

Severity
Medium

Description
Chainlink has a library `AggregatorV3Interface` with a function called `latestRoundData()`. This function returns the price feed among other details for the latest round.
Chainlink aggregators have a built-in circuit breaker if the price of an asset goes outside of a predetermined price band. The result is that if an asset experiences a huge drop in value, the price of the oracle will continue to return the minPrice instead of the actual price of the asset. Check chainlink doc here.

Vulnerable Code

- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L371-L375

Impacts
This would allow users to store their allocations with the asset but at the wrong price.

Remediation
The contract should check the returned answer/price against the minPrice/maxPrice and revert if the answer is outside of the bounds.

```
if (price >= maxPrice or price <= minPrice) revert();
```

Retest

This Vulnerability has been fixed by adding validation on price.

Bug ID #7 [Fixed]

Missing Price Feed Validation

Vulnerability Type
Incorrect Validation

Severity
Medium

Description
Chainlink has a library `AggregatorV3Interface` with a function called `latestRoundData()`. This function returns the price feed among other details for the latest round.
The contract was found to be using `latestRoundData()` without proper input validations on the returned parameters which might result in a stale and outdated price.

Vulnerable Code

- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L372

Impacts
Having oracles with functions to fetch price feed without any validation might introduce erroneous or invalid price values that could result in an invalid price calculation further in the contract.

Remediation

It is recommended to have input validations for all the parameters obtained from the Chainlink price feed. Here's a sample implementation:

```
(uint80 roundID, int256 price, , uint256 timestamp, uint80 answeredInRound) = Aggregator(_dataOracle).latestRoundData();
```

```
require(answer > 0, "Chainlink price <= 0");
require(answeredInRound >= roundID, "Stale price");
require(timestamp != 0, "Round not complete");
```

Retest

This vulnerability has been fixed by adding validation for roundID and answeredInRound.

Bug ID #8 [Fixed]

Use safeTransfer/safeTransferFrom instead of transfer/transferFrom

Vulnerability Type
Missing best practices

Severity
Low

Description
The transfer() and transferFrom() method is used instead of safeTransfer() and safeTransferFrom(), presumably to save gas however OpenZeppelin's documentation discourages the use of transferFrom(), use safeTransferFrom() whenever possible because safeTransferFrom auto-handles boolean return values whenever there's an error.

Affected Code

- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L433
- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L565
- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L590
- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L639
- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L706
- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L739

Impacts
Using safeTransferFrom has the following benefits -

- It checks the boolean return values of ERC20 operations and reverts the transaction if they fail,
- at the same time allowing you to support some non-standard ERC20 tokens that don't have boolean return values.
- It additionally provides helpers to increase or decrease an allowance, to mitigate an attack possible with vanilla approve.

Remediation

Consider using safeTransfer() and safeTransferFrom() instead of transfer() and transferFrom().

Retest

This issue has been fixed by replacing transfer/transferFrom with safeTransfer/safeTransferFrom

Bug ID #9 [Fixed]

Incorrect Error Message

Vulnerability Type
Incorrect Statement

Severity
Informational

Description
The function `changePrivateSaleTime()` allows the contract owner to update the private sale phase time. However, it includes a requirement check with an incorrect error message. Specifically, the condition `require(_privateSalePhaseTime[index] != newTime, "SAME_PRICE");` is intended to prevent setting the new time to the same value as the current time. The error message "SAME_PRICE" is misleading because it implies an issue related to pricing rather than time.

Affected Code

- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L519

Impacts
While this issue does not directly affect the functionality of the contract, it can lead to confusion and potential errors in managing the private sale timeline.

Remediation
To fix this issue, the error message should be updated to accurately reflect the condition being checked.

```
require(_privateSalePhaseTime[index] != newTime, "SAME_TIME");
```

Retest

This issue has been fixed by updating the error message according to the condition.

Bug ID #10 [fixed]

Missing Index Validation

Vulnerability Type
Missing Input Validation

Severity
Informational

Description
The functions `changePrivateSalePrice()` and `changePrivateSaleTime()` allow the contract owner to update the private sale prices and times, respectively. However, both functions lack proper validation of the `index` parameter to ensure it falls within the valid range of indices for the `privateSale_Prices` and `privateSale_PhaseTime` arrays.

Affected Code

- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L506
- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L517

Impacts
While this issue does not directly affect the functionality of the contract, it can consume unnecessary storage slots.

Remediation
To fix this issue, proper validation should be added to ensure the `index` parameter is within the valid range.

Retest
This issue has been fixed by adding index number validation.

Bug ID #11 [Fixed]

Cheaper Inequalities in if()

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The contract was found to be making comparisons using inequalities inside the “if” statement. When inside the “if” statements, non-strict inequalities (>=, <=) are usually cheaper than the strict equalities (>, <).

Affected Code

- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L417
- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L724

Impacts
Using strict inequalities inside “if” statements costs more gas.

Remediation
It is recommended to go through the code logic, and, **if possible**, modify the strict inequalities with the non-strict ones to save gas as long as the logic of the code is not affected.

Retest
This issue has been fixed by updating the strict inequalities to non-strict checks.

Bug ID #12 [Fixed]

Cheaper Inequalities in require()

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The contract was found to be performing comparisons using inequalities inside the require statement. When inside the require statements, non-strict inequalities (>=, <=) are usually costlier than strict equalities (>, <).

Affected Code

- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L612
- https://github.com/PlutopeIn/PLT/blob/b7401450b2c91d804ceba428205d7a2389e6aef0/contracts/PlutopePresale.sol#L445-L446

Impacts
Using non-strict inequalities inside “require” statements costs more gas.

Remediation
It is recommended to go through the code logic, and, **if possible**, modify the non-strict inequalities with the strict ones to save gas as long as the logic of the code is not affected.

Retest
This issue has been fixed by updating the strict inequalities to non-strict checks.

6. Disclosure

The Report provided by CredShields is not an endorsement or condemnation of any specific project or team and does not guarantee the security of any specific project. The contents of this report are not intended to be used to make decisions about buying or selling tokens, products, services, or any other assets and should not be interpreted as such.

Emerging technologies such as Smart Contracts and Solidity carry a high level of technical risk and uncertainty. CredShields does not provide any warranty or representation about the quality of code, the business model or the proprietors of any such business model, or the legal compliance of any business. The report is not intended to be used as investment advice and should not be relied upon as such.

CredShields Audit team is not responsible for any decisions or actions taken by any third party based on the report.