PASSWORDSTORE

SECURITY AUDIT REPORT::2023-12-27

SAFIRAETHER.COM
SAFEGUARDING ETHEREUM

Prepared by 808Néstor
NESTOR@SAFIRAETHER.COM

TABLE OF CONTENTS

Table of Contents 1
Introduction 2
About The Protocol 2
Audit Scope 3
Audit Roles 3
Known Issues 3
Disclaimer 4
Methodology 5
Risk Classifications 6
Findings 10
Executive Summary 10
Critical Severity Findings 12
High Severity Findings 16
Medium Severity Findings 16
Low Severity Findings 16
Informational Level Findings 17
Conclusion 18
Appendices 19
Appendix A: PasswordStore Storage 19

INTRODUCTION

About The Protocol

Protocol Objective

A smart contract applicatoin for storing a password. Users should be able to store a password and then retrieve it later. Others should not be able to access the password.

Solc Version

0.8.18

Chains

Ethereum

Tokens

N/A

Audit Scope

Commit Hash

7d55682ddc4301a7b13ae9413095feffd9924566

Files in Scope

./src/

└─── PasswordStore.sol

Audit Roles

- Owner: The user who can set the password and read the password.
- Outsides: No one else should be able to set or read the password.

Known Issues

N/A

DISCLAIMER

All actions by The Audit Team in this project adhere to the agreed statement of work and project plan. Security assessments are time-limited and rely on client-provided information. The findings in this report may not cover all security issues in the target system or codebase.

Automated testing supplements manual security reviews but has limitations. Tools may not cover all edge cases within the allocated time. This audit doesn't replace functional tests or guarantee identifying all security issues, emphasizing the need for multiple audits and a bug bounty program.

This report isn't investment advice and is subject to the terms of the Services Agreement. Distribution or reliance by any party other than the designated client is prohibited without prior written consent.

It neither endorses nor disapproves of any project, providing no insight into economic value or legal compliance. Users access services at their risk, acknowledging uncertainties of cryptographic tokens and blockchain technology.

The assessment may not uncover all vulnerabilities, and the absence of identified issues doesn't ensure a secure system. The Audit Team focuses on source code assessments, acknowledging software development limitations and potential impacts of third-party infrastructure.

The Audit Team, dedicated to discovering vulnerabilities within a timeframe, doesn't assume responsibility for findings outlined in this document. The audit solely addresses solidity implementation, not endorsing the underlying business. Recognizing time constraints, it exclusively focuses on the security aspects of the assessed code.

Methodology

Our audit methodology focused on ensuring the security, reliability, and adherence to best practices in the context of the Ethereum blockchain.

Business Logic Assessment

Our analysis commenced with a thorough understanding of the smart contract's business logic. The goal was to identify the core functionalities and interactions with external components, laying the foundation for subsequent evaluations.

Manual Code Review

A meticulous review of the Solidity source code was conducted, adhering to industry best practices and coding standards. The purpose was to identify potential vulnerabilities, ensuring the code is robust and maintainable.

Automated Analysis

Advanced automated analysis tools, including Aderyn, and Slither, were employed to identify common vulnerabilities related to security, gas efficiency, and code style. The results contributed to a comprehensive understanding of potential risks. Additionally, testing of invariants was conducted where appropriate via fuzzing and formal verification using tools such as Halmos and Certora.

Security Patterns Application

Security patterns and anti-patterns were applied to address common vulnerabilities, including reentrancy, overflow/underflow, and timestamp dependency. Access controls and permissions were implemented judiciously to enhance overall security.

External Calls Evaluation

External calls to other contracts or external systems were scrutinized to mitigate the risk of reentrancy attacks. The assessment ensured that all calls were secure and aligned with the integrity of the contract.

Input Validation

Rigorous validation of user inputs was conducted to ensure the smart contract gracefully handles unexpected inputs, guarding against vulnerabilities such as integer overflow and underflow.

Risk Classifications

Risk is classified based on two factors: likelihood and impact.

Likelihood

Likelihood refers to the probability of a specific event or vulnerability being exploited. Here's an elaboration on the three levels of likelihood:

High Likelihood:

In situations of high likelihood, the conditions for exploitation are readily accessible or the attack vector is easily achievable. The vulnerability is considered highly exploitable, and the likelihood of occurrence is relatively high.

Example Scenario: A scenario where a hacker can directly call a function to cause a significant impact on the smart contract's behavior. This could involve straightforward and easily executable steps that do not require elaborate conditions.

Medium Likelihood:

In cases of medium likelihood, specific conditions or a more constrained set of circumstances are required for the vulnerability to be exploited. While not as easily achievable as high likelihood scenarios, the conditions for exploitation are still reasonably attainable, making the event moderately likely to occur.

Example Scenario: An example could be a vulnerability that depends on the use of a particular type of token within the platform. While not universally applicable, the conditions for exploitation are plausible and could occur under specific circumstances.

Low Likelihood:

Vulnerabilities with low likelihood are associated with rare situations that are unlikely to happen in typical scenarios. Although technically feasible, the conditions required for exploitation are infrequent or involve a combination of events that are unlikely to align.

Example Scenario: Consider a vulnerability that relies on a unique sequence of events (A, B, C) taking place at a specific time. While technically possible, the occurrence of such a sequence is rare and unlikely, making the exploitation of the vulnerability less probable.

NOTE:

Computationally Unfeasible: Events that are 'computationally unfeasible' are practically impossible due to their extreme rarity or the astronomical computational effort required for exploitation. These are not considered viable attack paths in practice.

Assessing likelihood involves a degree of subjectivity and requires a comprehensive understanding of the smart contract's architecture, the blockchain environment, and potential threat vectors. Regular reviews and updates to likelihood assessments should be conducted to adapt to changes in the threat landscape and the evolving nature of decentralized applications.

IMPACT

Impact is a crucial aspect of vulnerability assessment, representing the potential harm or consequences resulting from a vulnerability. Here are the three levels of impact:

HIGH IMPACT:

High impact vulnerabilities pose a significant threat to the protocol, where funds are directly or nearly directly at risk. The consequences involve a severe disruption of protocol functionality or availability, with potential financial losses for users.

Examples:

- Direct exposure of funds to unauthorized access.
- Severe disruption in the protocol's core functionality, leading to financial losses.

MEDIUM IMPACT:

Medium impact vulnerabilities introduce some level of risk to funds, albeit indirectly. While not as severe as high impact scenarios, these vulnerabilities can still result in disruptions to the protocol's functionality or availability, impacting the user experience and potentially leading to financial consequences.

Examples:

- Indirect risk to funds due to a vulnerability that affects the integrity of transactions or user accounts.
- Moderate disruption in protocol functionality, affecting user interactions and potentially leading to financial consequences.

Low Impact:

Low impact vulnerabilities do not directly put funds at risk. However, there might be issues related to the correctness of functions, inappropriate handling of states, or other non-financial implications. The primary concern is related to the correctness and reliability of the protocol rather than immediate financial losses.

Examples:

- Correctness issues in specific functions that do not pose a direct threat to funds but may impact the overall reliability of the protocol.
- State handling concerns that do not immediately impact financial transactions but may affect the protocol's overall performance.

Assessing impact is crucial for prioritizing the resolution of vulnerabilities and allocating resources effectively. It helps in understanding the potential consequences of each vulnerability and guides the development team in addressing the most critical issues first, ensuring the overall security and stability of the protocol. Regular impact assessments should be conducted to adapt to changes in the protocol's features and user interactions.

Severity Ratings

Severity ratings in the context of smart contract security assessments are typically derived from the combination of likelihood and impact assessments. Here are the severity ratings based on the provided likelihood and impact descriptions:

Impact	Low	Medium	High
High	MEDIUM	HIGH	CRITICAL
Medium	LOW	MEDIUM	HIGH
Low	LOW	LOW	MEDIUM

Likelihood

Critical Severity:

- High Likelihood + High Impact

Vulnerabilities with a critical severity rating represent a high likelihood of exploitation coupled with significant consequences. In these scenarios, funds are directly at risk, and there's a severe disruption of protocol functionality or availability, posing a substantial threat to users and the overall integrity of the protocol.

High Severity:

- Medium Likelihood + High Impact
- High Likelihood + Medium Impact

High severity vulnerabilities indicate a substantial risk to the protocol. While the likelihood may be high or medium, the impact is never low, with the potential for direct financial losses and severe disruption of protocol functionality. These vulnerabilities demand immediate attention and remediation efforts.

Medium Severity:

- Low Likelihood + High Impact
- Medium Likelihood + Medium Impact
- High Likelihood + Low Impact

Medium severity vulnerabilities pose a moderate risk to the protocol. While the likelihood of exploitation and the impact may vary, the potential for indirect financial risks and disruptions to the protocol's functionality remains constant. Timely remediation is recommended to maintain the overall security and stability of the system.

Low Severity:

- Low Likelihood + Low Impact
- Low Likelihood + Medium Impact
- Medium Likelihood + Low Impact

Low severity vulnerabilities represent a lower risk to the protocol. The likelihood of exploitation is low or medium, and the impact may be low or medium, primarily related to correctness issues or non-financial implications. While these vulnerabilities are not immediate threats, they should still be addressed in a timely manner to enhance the robustness of the protocol.

Informational Severity:

- Likelihood: Not Applicable
- Impact: Not Applicable

Informational severity is used for findings that provide valuable information but do not pose an immediate risk to the protocol's security or functionality. These findings may include suggestions for improvement or best practices that could enhance the overall security posture.

Severity ratings provide a structured way to prioritize and communicate the urgency of addressing identified vulnerabilities.

FINDINGS

Executive Summary

Overview

In the comprehensive audit of the smart contract, two critical findings, denoted as C-01 and C-02, reveal significant vulnerabilities with severe implications for the protocol's security. The first critical issue, identified as "Sensitive Data On Chain," brings attention to a vulnerability that exposes the password stored in the contract to potential exploitation. Despite the `PasswordStore::s_password` state variable being marked as 'private,' it's crucial to recognize that all data on the blockchain, including supposedly private information in smart contracts, is visible to anyone with the knowledge to query the blockchain's state or analyze transaction histories. The consequence of this vulnerability is a direct compromise of confidentiality. Moreover, if the compromised password is reused across other accounts, there exists the risk of off-chain account compromises. For instance, if the password is employed in encrypting a private key, an attacker could potentially decrypt the private key, leading to unauthorized access and theft of funds.

The second critical finding, labeled as "Access control vulnerability" (C-02), sheds light on a significant flaw in the contract's design. Specifically, the `PasswordStore::setPassword` function is marked as 'external,' implying that it can be executed by anyone, not solely the contract owner. This oversight introduces a critical access control vulnerability, allowing any external entity to set the password at will. Such unauthorized modification capability could lead to widespread abuse and manipulation of the contract's critical information, posing a substantial threat to the integrity and intended functionality of the protocol.

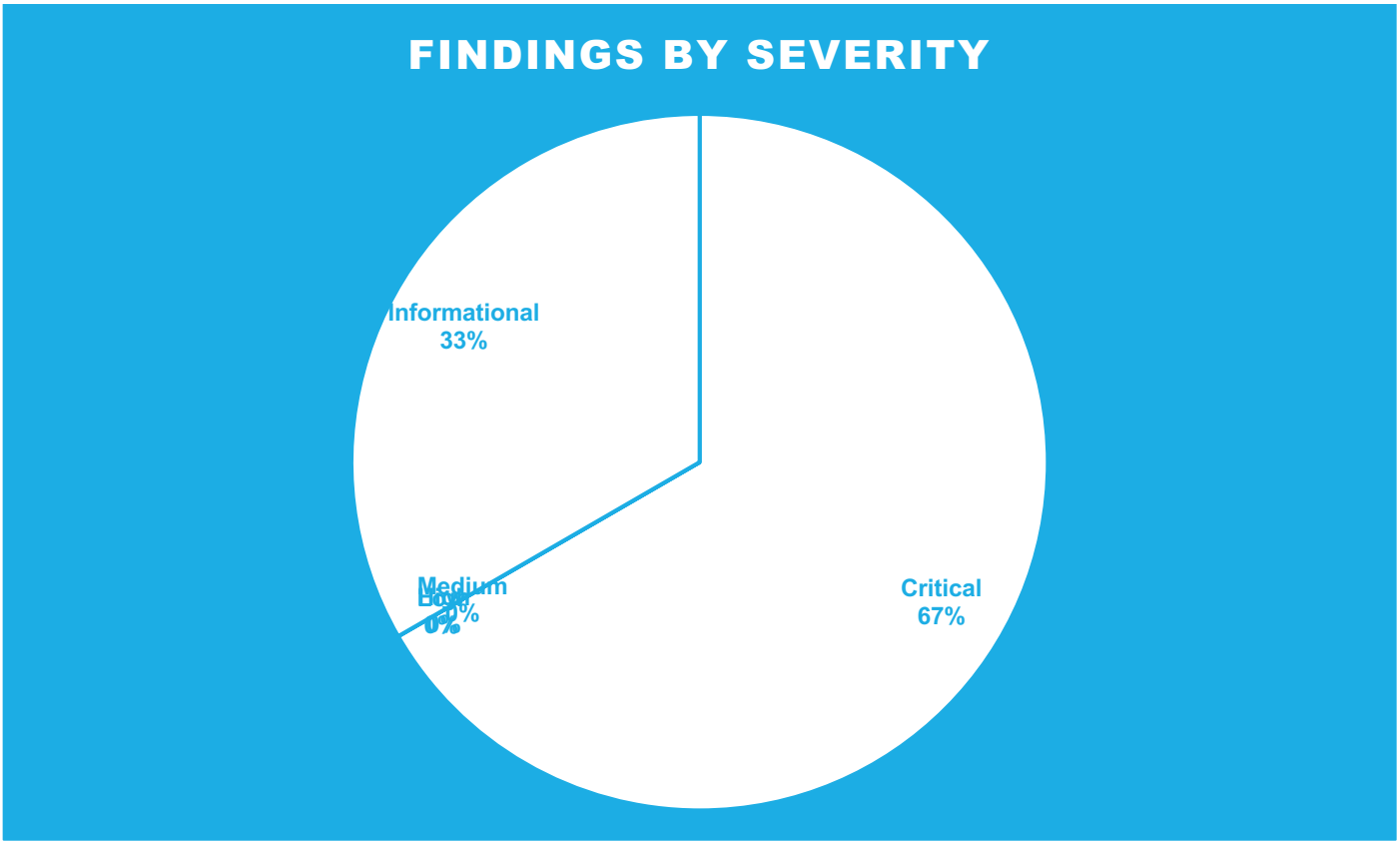
Beyond the critical issues, the audit also flagged informational findings, such as I-01, indicating areas for improvement without posing an immediate and severe threat. In this case, the `PasswordStore::getPassword` function features a parameter labeled `newPassword` that remains unused within the function body. While not constituting a critical vulnerability, this informational finding suggests potential inefficiencies or oversights in the codebase. Addressing such issues contributes to the overall clarity, efficiency, and maintainability of the code.

The cumulative impact on the protocol is profound, with critical vulnerabilities directly jeopardizing data confidentiality and access control. The "Sensitive Data On Chain" vulnerability exposes the password, putting off-chain accounts at risk, while the "Access control vulnerability" compromises the integrity of the contract itself. Addressing these critical issues is imperative to fortify the protocol against unauthorized access, mitigate the potential for data breaches, and safeguard against financial losses stemming from exploitation.

Findings By Severity Tally Table

Severity	Tally
Critical	2
High	0
Medium	0
Low	0
Informational	1

Findings By Severity Tally Chart



Critical Severity Findings

C-01: Sensitive Data On Chain vulnerability allows anyone to read the password.

File: PasswordStore

Element: s_password

Likelihood: High

Financial Impact: High

Severity: Critical

Details

`PasswordStore::s_password` state variable is marked 'private'. However, all data on the blockchain, including that marked 'private' in smart contracts, is visible to anyone who knows how to query the blockchain's state or analyze its transaction history. Private variables are not exempt from public inspection.

Impact

Anyone can read the password stored in this contract. Resulting in a loss of confidentiality. Futhermore, if the password is reused, ofchain accounts could be compromised. For example, if the password is used to encrypt a private key, an attacker could decrypt the private key and steal funds.

Proof Of Concept

```
# Step 1: Identify Target of Attack
```
```
# Confirm storage slot used for variable 's_password' using Sol2UML
sol2uml class -f png -o ./audit-notes/recon/uml_classes.png ./src

Results: See Appendix A
```
```

# Step 2: Initiate Attack
```
```

# Spin up local chain for testing
$ forge anvil
```

```
# deploy contract according to project
$ make deploy

# Use Foundry's cast tool to read the data stored at the desirted slot
# usage: cast storage <address> <storageSlot>
cast storage 0x5FbDB2315678afecb367f032d93F642f64180aa3 1

Results: `0x6d7950617373776f726400000000000000000000000000000000000000000014`

# Use Foundry's cast tool to parse the bytes32 data into a text string
cast parse-bytes32-string
0x6d7950617373776f726400000000000000000000000000000000000000000014

Results: `myPassword`
```

```
# Step 3: Confirm Results
```

```
# Line 11 of project script 'DeployPasswordStore.s.sol':
passwordStore.setPassword("myPassword");
```

Tools Used

Manual

Recommended Mitigation

Given that the purpose of this contract is to store a password, it is not clear why the password is stored on-chain. Consider storing the password off-chain, or using a hash of the password instead of the password itself.

C-02: Access control vulnerability allows anyone to set the password.

File: PasswordStore

Element: setPassword

Likelihood: High

Financial Impact: High

Severity: Critical

Details

`PasswordStore::setPassword` is marked 'external'. Functions marked "external" or "public" can be executed by anyone, not just the owner of the contract.

Impact

Anyone can set the password.

Proof Of Concept

```
function test_anyone_can_set_password(address randomAddress) public {
        // assume randomAddress is not owner
        vm.assume(randomAddress != owner);

        // owner checks password
        vm.prank(owner);
        string memory startingPassword = passwordStore.getPassword();
        console.log("starting Password: %s", startingPassword);

        // attacker sets password from random address
        vm.prank(randomAddress);
        string memory attackerPassword = "newPasswordFromRandomAddress";
        passwordStore.setPassword(attackerPassword);
        console.log("attacker changes Password...");

        // owner checks password again
        vm.prank(owner);
        string memory actualPassword = passwordStore.getPassword();
        console.log("ending Password: %s", actualPassword);

        // owner sees that password has changed
        assertEq(actualPassword, attackerPassword);
```

```
    }
```

TOOLS USED

Manual

RECOMMENDED MITIGATION

This functions needs to be called externally by the owner of the contract. Therefore, use require(msg.sender == s_owner) to restrict access to the owner. This will prevent anyone else from setting the password.

High Severity Findings

N/A

Medium Severity Findings

N/A

Low Severity Findings

N/A

INFORMATIONAL LEVEL FINDINGS

I-01: STATED PARAMETER IS NOT USED IN THE FUNCTION BODY.

File: PasswordStore

Element: getPassword

Likelihood: NA

Financial Impact: NA

Severity: Informational

DETAILS

`PasswordStore::getPassword` has a parameter `newPassword` that is not used in the function body.

IMPACT

The parameter is not used in the function body, so it is not clear why it is there.

PROOF OF CONCEPT

N/A

TOOLS USED

Manual

RECOMMENDED MITIGATION

Remove the parameter `newPassword` from the function signature.

CONCLUSION

To address the critical vulnerabilities identified in the smart contract audit, specific mitigations are recommended. For the "Sensitive Data On Chain" issue (C-01), it is advised to reassess the on-chain storage approach for the password. Considering the contract's purpose is password storage, storing the password off-chain or using a hash instead of the password itself can significantly enhance security.

As for the "Access control vulnerability" (C-02), it is crucial to restrict external access to the `PasswordStore::setPassword` function. Adding a verification check, such as `require(msg.sender == s_owner)`, ensures only the contract owner can set the password, preventing unauthorized access.

Additionally, for the informational finding (I-01), removing the unused parameter `newPassword` from the `PasswordStore::getPassword` function simplifies the code and enhances clarity.

Moving forward, adopting post-audit best practices is essential. Regularly conduct security reviews and audits to stay ahead of potential vulnerabilities. Implement a robust testing strategy, including unit tests and third-party audits, to ensure the continued resilience of the smart contract. Foster open communication within the development community, addressing concerns and collaborating on security best practices. Continuous learning and adaptation to evolving security standards are key to maintaining the integrity of smart contracts.

Remember, we're in this together to create secure and reliable solutions. Let's keep the collaboration going for a safer and more resilient smart contract ecosystem! 🌐🔒

APPENDICES

APPENDIX A: PASSWORDSTORE STORAGE

PasswordStore <<Contract>>		
slot	type: <inherited contract>.variable (bytes)	
0	unallocated (12)	address: s_owner (20)
1	string: s_password (32)	