Protocol Audit Report

Version 1.0

Maroutis

December 13, 2023

PasswordStore Audit Report

Maroutis

December, 2023

PasswordStore Audit Report

Prepared by: Maroutis Lead Security Researcher:

- Maroutis

Table of Contents

- PasswordStore Audit Report
- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
- Protocol Summary
  - Roles
- Executive Summary
  - Issues found
- Findings
- High

    - [H-1] Passwords stored on-chain are visable to anyone, not matter solidity variable visibility
    - [H-2] `PasswordStore::setPassword` is callable by anyone
- Low
    - [L-1] Initialization Timeframe Vulnerability
- Informational
    - [I-1] The `PasswordStore::getPassword` natspec indicates a parameter that doesn't exist, causing the natspec to be incorrect

Protocol Summary

PasswordStore is a protocol dedicated to storage and retrieval of a user's passwords. The protocol is designed to be used by a single user, and is not designed to be used by multiple users. Only the owner should be able to set and access this password.

Disclaimer

The Maroutis team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond the following commit hash:

```
2e8f81e263b3a9d18fab4fb5c46805ffc10a9990
```

Scope

```
src/
--- PasswordStore.sol
```

Protocol Summary

PasswordStore is a protocol dedicated to storage and retrieval of a user's passwords. The protocol is designed to be used by a single user, and is not designed to be used by multiple users. Only the owner should be able to set and access this password.

Roles

- Owner: Is the only one who should be able to set and access the password.
- For this contract, only the owner should be able to interact with the contract.

Executive Summary

Issues found

Severity	Number of issues found
High	2
Medium	0
Low	1
Info	1
Gas Optimizations	0

Severity	Number of issues found
Total	4

Findings

High

[H-1] Passwords stored on-chain are visable to anyone, not matter solidity variable visibility

Description: All data stored on-chain is visible to anyone, and can be read directly from the blockchain. The `PasswordStore::s_password` variable is intended to be a private variable, and only accessed through the `PasswordStore::getPassword` function, which is intended to be only called by the owner of the contract.

However, anyone can direclty read this using any number of off chain methodologies

Impact: The password is not private.

Proof of Concept 1: The below test case shows how anyone could read the password directly from the blockchain. We use foundry's cast tool to read directly from the storage of the contract, without being the owner.

1. Create a locally running chain

```
make anvil
```

2. Deploy the contract to the chain

```
make deploy
```

3. Run the storage tool

We use 1 because that's the storage slot of `s_password` in the contract.

```
cast storage <ADDRESS_HERE> 1 --rpc-url http://127.0.0.1:8545
```

You'll get an output that looks like this:

`0x6d7950617373776f726400000000000000000000000000000000000000000014`

You can then parse that hex to a string with:

```
cast parse-bytes32-string 0
    x6d7950617373776f726400000000000000000000000000000000000000000014
```

And get an output of:

```
myPassword
```

Proof of Concept 2: You can execute the following test using the command : forge test –mt test_non_owner_can_read_password -vvvv

Code

```
function test_non_owner_can_read_password() public {
        vm.startPrank(owner);
        string memory passwordRead = passwordStore.getPassword();
        vm.stopPrank();

        address hacker = makeAddr("Hacker");
        vm.startPrank(hacker);

        // @note The following code works with every password string
            size up to maxSize = 127 bytes or total length in slot = (
            maxSize * 2) + 1 = 255 bytes

        // Check the slot value
        bytes32 passwordSlot = vm.load(address(passwordStore), bytes32(
            uint256(1)));

        uint256 SLOT_SIZE = 32;
        // Keep only the first byte (max size of string = 127 bytes <=>
             length = 255 bytes)
        bytes32 mask = 0
            x00000000000000000000000000000000000000000000000000000000000000FF
            ;
        bytes32 maskedPassword = passwordSlot & mask;

        bytes memory passwordData;
        bytes memory trimmedPasswordData;

        // check if string password value has 32 bytes or more
        if (maskedPassword == passwordSlot) {
            // Each 32 bytes slice of the password bytes is stored on a
                 single slot
            // rounding up division
            uint256 slotsUsed = ((uint256(passwordSlot) - 1) / 2 + (
                SLOT_SIZE - 1)) / SLOT_SIZE;
            // loop over number of slots (for passwords size > 32 bytes
                )
            for (uint256 j = 0; j < slotsUsed; j++) {
                passwordSlot = vm.load(address(passwordStore), bytes32(
```

```
                    uint256(keccak256(abi.encode(uint256(1)))) + j));

                if (j < slotsUsed - 1) {
                    // concatenate each 32 bytes slice to passwordData
                    passwordData = abi.encodePacked(passwordData,
                        passwordSlot);

                    // For the last slice trim the trailings 0s padded
                        to passwordSlot
                } else {
                    uint256 length = SLOT_SIZE;
                    while (length > 0 && passwordSlot[length - 1] == 0)
                        {
                        length--;
                    }
                    // write non padded values into passwordData
                    bytes memory passwordDataTemp = new bytes(length);
                    for (uint256 i = 0; i < length; i++) {
                        passwordDataTemp[i] = passwordSlot[i];
                    }
                    passwordData = abi.encodePacked(passwordData,
                        passwordDataTemp);
                }
            }
        }
        // if string password has maximum 31 bytes
        else {
            // mask the length data in passwordSlot
            bytes32 SecondMask = 0
                xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00
                ;
            passwordSlot = passwordSlot & SecondMask;

            // Trim the trailings 0s padded to passwordSlot
            uint256 length = SLOT_SIZE;
            while (length > 0 && passwordSlot[length - 1] == 0) {
                length--;
            }
            // write non padded value into passwordData
            passwordData = new bytes(length);
            for (uint256 i = 0; i < length; i++) {
                passwordData[i] = passwordSlot[i];
            }
        }
        // check that passwords match
        assertEq(string(passwordData), passwordRead);
    }
```

Recommended Mitigation: Due to this, the overall architecture of the contract should be rethought. One could encrypt the password off-chain, and then store the encrypted password on-chain. This

would require the user to remember another password off-chain to decrypt the password. However, you'd also likely want to remove the view function as you wouldn't want the user to accidentally send a transaction with the password that decrypts your password.

[H-2] `PasswordStore::setPassword` is callable by anyone

Description: The `PasswordStore::setPassword` function is set to be an `external` function, however the natspec of the function and overall purpose of the smart contract is that `This function allows only the owner to set a new password.`

```
1      function setPassword(string memory newPassword) external {
2  @>      // @audit - There are no access controls here
3          s_password = newPassword;
4          emit SetNetPassword();
5      }
```

Impact: Anyone can set/change the password of the contract.

Proof of Concept:

Add the following to the `PasswordStore.t.sol` test suite.

```
function test_anyone_can_set_password(address randomAddress) public {
    vm.prank(randomAddress);
    string memory expectedPassword = "myNewPassword";
    passwordStore.setPassword(expectedPassword);
    vm.prank(owner);
    string memory actualPassword = passwordStore.getPassword();
    assertEq(actualPassword, expectedPassword);
}
```

Recommended Mitigation: Add an access control modifier to the `setPassword` function.

```
if (msg.sender != s_owner) {
    revert PasswordStore__NotOwner();
}
```

Low

[L-1] Initialization Timeframe Vulnerability

Description: The PasswordStore contract exhibits an initialization timeframe vulnerability. This means that there is a period between contract deployment and the explicit call to setPassword during which the password remains in its default state. It's essential to note that even after addressing this issue,

the password's public visibility on the blockchain cannot be entirely mitigated, as blockchain data is inherently public as already stated in the "Storing password in blockchain" vulnerability.

The contract does not set the password during its construction (in the constructor). As a result, when the contract is initially deployed, the password remains uninitialized, taking on the default value for a string, which is an empty string.

During this initialization timeframe, the contract's password is effectively empty and can be considered a security gap.

Impact: The impact of this vulnerability is that during the initialization timeframe, the contract's password is left empty, potentially exposing the contract to unauthorized access or unintended behavior.

Recommended Mitigation: To mitigate the initialization timeframe vulnerability, consider setting a password value during the contract's deployment (in the constructor). This initial value can be passed in the constructor parameters.

Informational

[I-1] The `PasswordStore::getPassword` natspec indicates a parameter that doesn't exist, causing the natspec to be incorrect

Description:

```
1      /*
2       * @notice This allows only the owner to retrieve the password.
3  @>   * @param newPassword The new password to set.
4       */
5      function getPassword() external view returns (string memory) {
```

The natspec for the function `PasswordStore::getPassword` indicates it should have a parameter with the signature `getPassword(string)`. However, the actual function signature is `getPassword()`.

Impact: The natspec is incorrect.

Recommended Mitigation: Remove the incorrect natspec line.

```
1  -     * @param newPassword The new password to set.
```