Oct.23

DRAFT DIGITAL SECURITY REPORT:

PASSWORDSTORE

CONTENTS

ABOUT DRAFT DIGITAL

Draft Digital is a Web3 security company dedicated to providing high standards of protection and security for our partners and their projects. Our mission is to create a secure, reliable and transparent environment for everyone on Web3 and DeFi.

Smart Contract Audits:

A professional service that will allow you to increase the security of your smart contracts and minimise the risk of being hacked. Our professionals go line by line to secure your code and create clear and concise reports that build trust with our partners.

Code analysis & Consulting:

Our consultants work closely with the developers during the project, creating a climate of trust in the code and ensuring the highest standards when placing the code in production.

Web3 Security Education:

At Draft Digital we believe that the best prevention is knowledge. We provide courses and bootcamps for developers so that their code is guaranteed safe from the first line.

News and Ecosystem Information:

In addition to our education services, research plays an essential role in web3 and is central to its security. New exploits emerge every day and we need to be aware of them immediately. Our news and information service is kept up to date to prevent any new attacks.

AUDIT INFORMATION

SCOPE

The analyzed resources are located at:
https://github.com/Cyfrin/2023-10-PasswordStore

The issues described in this report were fixed in the following commit:
https://github.com/aitorzaldua/Codehawks_FirstFligths/tree/main/src/01-PasswordStore

LEAD BY

Aitor Zaldua — Co-founder / CTO | Draft Digital

DATES

Audit Starting Date
2023 - Oct 18th

Audit Completion Date
12023 - Oct 25th

METHODOLOGY

First Review and Estimations:
After reviewing the information, we made estimates of expected costs and completion time. Each client is treated on a personalised basis according to their interests and resources.

Agreement & Start of Service:
After reaching a service agreement, we start the audit by adjusting the work to the agreed timescales and taking into account the committed partial deliverables.

Audit Time:
Our auditors are dedicated full-time to a single client during their audit time, which guarantees 100% audit attention. They maintain open channels of communication with the client's contacts at all times.

Initial report:
The so-called initial report is the one that already contains the 100% code review. It will be delivered for review by the client and, subsequently, a meeting will be held between the engineers involved to review and explain each vulnerability or possible improvement detected.

Mitigation:
The client's developers and engineers proceed to test and implement the results of the initial report, always with open channels of communication with the auditors.

Final Report:
Once all checks have been performed and the agreed fixes have been applied, our auditors will review the code again and update the report, indicating the vulnerabilities already fixed and making the final assessments.

SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

IMPACT	PROBABILITY			
	Rare	Unlikely	Likely	Very Likely
Low / Info	Low / Info	Low / Info	Medium	Medium
Medium	Info Low	Medium	Medium	High
High	/ Info	Medium	High	Critical
Critical	Medium	High	Critical	Critical

Medium

SEVERITY CHARACTERISTICS

Vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of vulnerabilities:

CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of the project. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered PasswordStore protocol, that is a part of the Codehawks First Flight program.

PasswordStore is a simple solidity protocol meant to allow the owner to store and retrieve their password securely. Never worry about forgetting your password again!

Our security assessment was a full review of the PasswordStore protocol going further than the Codehawks contests usually goes, covering the full range of possibles vulnerabilities. We have thoroughly reviewed each contract individually, as well as the system as a whole.

During our audit, we have identified 2 critical severity vulnerabilities in the main contract. Both allow a malicious actor to take control of the platform.

All of the reported issues were fixed by the development team and consequently validated by us.

We can confidently say that the overall security and code quality has increased after completion of our audit.

SUMMARY

SEVERITY	NUMBER OF FINDINGS
CRITICAL	2
HIGH	0
MEDIUM	0
LOW	1
INFORMATIONAL	0

TOTAL: 3

current Status:
All Fixed.

VULNERABILITIES

This section contains the list of discovered weaknesses.

PASSWORDSTORE-1: The password is never stored privately in a variable, even if it is marked as private.

SEVERITY: HIGH

PATH: src/01-PasswordStore/PasswordStore.sol

SUMMARY: Marking a variable as private doesn't mean that it's hidden from view. Any user with enough knowledge can see the data stored in it.

IMPACT: Since the password is no longer private, anyone can use it to change information, make transactions or steal assets.

REMEDIATION: In web3 we do not use stored passwords as a method of keeping things private. There are 2 ways to secure the systems:

1.- Asymetric encryption (wallets or signatures).

2.- Access control modifiers. We can use the Open Zeppelin libraries which are specifically designed for this: https://docs.openzeppelin.com/contracts/2.x/access-control

STATUS: FIXED

PROOF OF CONCEPT:

To prove the concept, we need to deploy a contract on the blockchain and add a value to s_password. In our case, we use Anvil (Foundry) and deploy the contract on it with the first account given to us.

Then, as the owner, using cast, we set a value in s_password:

```
x5FbDB2315678afecb367f032d93F642f64180aa3 "setPassword(string)" "banana" --rpc-url http://127.0.0.1
```

Now we are going to run 2 instructions that do not require user validation, i.e. we do not use the owner's private key to obtain the information.

First, find out which is the memory slot for `s_password``

```
➜  2023-10-PasswordStore git:(main) ✗ forge inspect PasswordStore storage
{
  "storage": [
    {
      "astId": 43436,
      "contract": "src/PasswordStore.sol:PasswordStore",
      "label": "s_owner",
      "offset": 0,
      "slot": "0",
      "type": "t_address"
    },
    {
      "astId": 43438,
      "contract": "src/PasswordStore.sol:PasswordStore",
      "label": "s_password",
      "offset": 0,
      "slot": "1",
      "type": "t_string_storage"
    }
```

As we can see, it is slot 1. Now, we check the info inside. We get the information in hexadecimal, so we have to translate it to ASCII.

```
➜  2023-10-PasswordStore git:(main) ✗ cast storage 0x5FbDB2315678afecb367f032d93F642f64180aa3 1
0x62616e616e610000000000000000000000000000000000000000000000000c
➜  2023-10-PasswordStore git:(main) ✗ cast to-ascii "0x62616e616e610000000000000000000000000000000000000000000000000c"
banana
```

PASSWORDSTORE-2: Developer forgot to add the onlyOwner protection to setPassword()

SEVERITY: HIGH

PATH: src/01-PasswordStore/PasswordStore.sol

SUMMARY: The function comment for setPassword() reads: "This function allows only the owner to set a new password", but the developer forgot to add some kind of modifier to secure the function, and it is external, so anyone can set the password at any time.

IMPACT: Because any user can set and subsequently use a new password, any application or transaction that depends on it is compromised, as is any asset.

REMEDIATION: The best way to secure an onlyOwner function is to use the Open Zeppelin library, as it has already been proven by high-level security researchers. Check it in:

https://docs.openzeppelin.com/contracts/2.x/api/ownership#Ownable

As every function in the contract needs the onlyOwner protection, another way to secure the function is creating an onlyOwner modifier

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

We have to add it to the function

```
function setPassword(string memory newPassword) external onlyOwner {
        s_password = newPassword;
        emit SetNetPassword();
    }
```

And run the same test to prove it works

```
  2023-10-PasswordStore git:(main) ✗ forge test --mt test_setPasswordAsAttacker
[⠒] Compiling...
[⠒] Compiling 3 files with 0.8.18
[⠒] Solc 0.8.18 finished in 911.27ms
Compiler run successful!

Running 1 test for test/PasswordStore.t.sol:PasswordStoreTest
[FAIL. Reason: PasswordStore__NotOwner()] test_setPasswordAsAttacker() (gas: 10771)
Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 426.38µs

Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/PasswordStore.t.sol:PasswordStoreTest
[FAIL. Reason: PasswordStore__NotOwner()] test_setPasswordAsAttacker() (gas: 10771)

Encountered a total of 1 failing tests, 0 tests succeeded
```

STATUS: FIXED

PROOF OF CONCEPT:

Check this test function created with Foundry

```
function test_setPasswordAsAttacker() external {
        vm.prank(address(1));
        passwordStore.setPassword("banana");
        vm.prank(owner);
        string memory actualPassword = passwordStore.getPassword();
        assertEq(actualPassword, "banana");
    }
```

If we execute it:

```
➜  2023-10-PasswordStore git:(main) ✗ forge test --mt test_setPasswordAsAttacker
[⠒] Compiling...
[⠒] Compiling 1 files with 0.8.18
[⠒] Solc 0.8.18 finished in 902.07ms
Compiler run successful!

Running 1 test for test/PasswordStore.t.sol:PasswordStoreTest
[PASS] test_setPasswordAsAttacker() (gas: 22234)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.31ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

The user address(1) is able to set a new password.

PASSWORDSTORE-3: Initialization Timeframe Vulnerability

SEVERITY: LOW

PATH: src/01-PasswordStore/PasswordStore.sol

SUMMARY: 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.

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.

REMEDIATION: 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.

STATUS: FIXED

PROOF OF CONCEPT:

To prove the concept, we need to create a very simple test

```
function test_initialPasswordIsEmpty() external {
    vm.prank(owner);
    string memory currentPassword = passwordStore.getPassword();
    assertEq(currentPassword, "");
}
```

The execution prove that the password is empty at the beginning

```
➜  2023-10-PasswordStore git:(main) ✗ forge test --mt test_initialPasswordIsEmpty
[⠒] Compiling...
[⠒] Compiling 2 files with 0.8.18
[⠒] Solc 0.8.18 finished in 917.06ms
Compiler run successful!

Running 1 test for test/PasswordStore.t.sol:PasswordStoreTest
[PASS] test_initialPasswordIsEmpty() (gas: 16179)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 429.58µs

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
➜  2023-10-PasswordStore git:(main) ✗
```

info@draftdigital.xyz