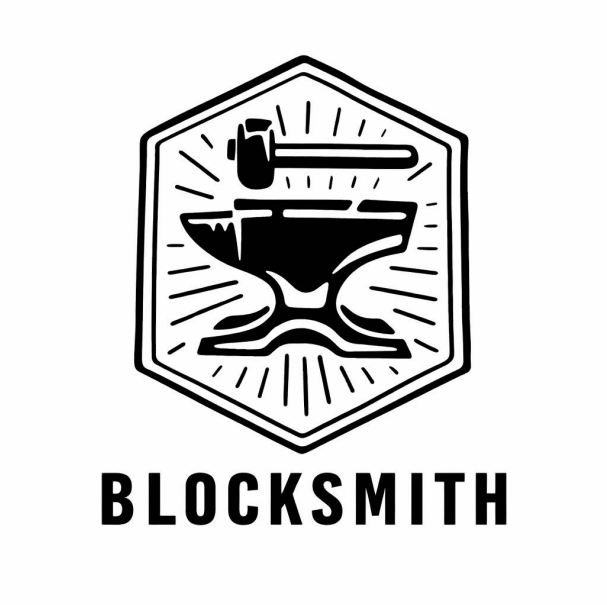

Protocol Audit Report

Version 1.0

Blocksmith

February 10, 2026

Protocol Audit Report

Blocksmith

Feb 10, 2026

Prepared by: Blocksmith Lead Security Researcher:

Table of Contents

Protocol Summary

This contract allows you to store a private password that others won't be able to see. You can update your password at any time.

Disclaimer

The Blocksmith 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 findinghs in this report it is based on the following commit hash:

```
7d55682ddc4301a7b13ae9413095feffd9924566
```

Scope

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

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.

Executive Summary

Add notes, some things that found interesting and how much time spent: X hours with Z auditords using Y tools

Issues found

Severity	Number of issues found
High	2
Medium	0
Low	0
Info	1
Total	3

Findings

High

[H-1] Storing the password on-chain makes visable to anyone dependendly if its private

Description: The contract stores the `PasswordStore::s_password` as a string in contract storage. Even though the variable is marked private, all on-chain storage is publicly readable. Any user can retrieve the password using blockchain explorers or RPC calls such as eth_getStorageAt.

Impact: The password is fully exposed to the public. Any assumption of secrecy is invalid, defeating the primary purpose of the contract. This is a critical design flaw.

Proof of Concept:

The below test case show that anyone can read the password form the blockchain.

1. Create a locally running chain using

```
make anvil
```

2. Deplaoy the contract to chain

```
make deploy
```

4. Run the storage tool

we use 1 because the slot of `s_password` in the contract

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

you take an output like this :

`0x6d7950617373776f726400000000000000000000000000000000000000000014`

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

and then you see that returns the :

```
myPassword
```

Recommended Mitigation: Due to this, the overall architect need to rethought. As a resolve you can encrypt the password and the store it to the blockchain but the user it requier to remember another password to decrypt.

[H-2] `PasswordStore::setPassword` has no access controls, meaning a non-owner could change the password

Description: The `PasswordStore::setPassword` function lacks access control and does not verify that the caller is the contract owner. Although the NatSpec comment states that only the owner can set a new password, `the implementation allows any external address to call this function` and overwrite the stored password.

```
function setPassword(string memory newPassword) external {
        //@audit - There are no access controls
        s_password = newPassword;
        emit SetNetPassword();
    }
```

Impact: Any attacker can arbitrarily change the password, permanently locking the legitimate owner out of the contract's intended functionality. This completely breaks the trust and access model of the contract.

Proof of Concept: Add the following to the `PasswordStore.t.sol` test file.

```
    function testFuzz_non_owner_can_set_password(address attacker)
        public {
        // This fuzz test demonstrates the access control vulnerability
        // A non-owner should NOT be able to set the password, but they
            can!

        // Skip if attacker is the owner (we want to test non-owners
            only)
        vm.assume(attacker != owner);

        string memory attackerPassword = "hackedPassword";

        // Attacker (non-owner) sets a new password
        vm.prank(attacker);
        passwordStore.setPassword(attackerPassword);

        // Verify the password was changed by checking as owner
        vm.prank(owner);
        string memory actualPassword = passwordStore.getPassword();

        // This assertion PASSES, proving the vulnerability exists
        // ANY random attacker can successfully change the password!
        assertEq(actualPassword, attackerPassword);
    }
```

Recommended Mitigation: Add an ownership check to the `setPassword` function using msg.sender and revert if the caller is not the owner.

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

Informational

Gas