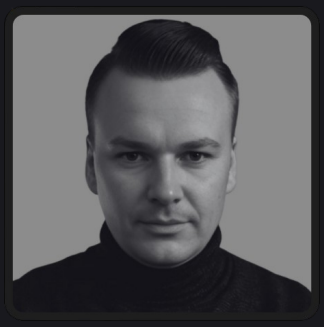

VOLODYMYR STETSENKO

Independent Security Researcher

PasswordStore

Security Review

Assessment Period: 24–26 November 2025

Advancing smart contract security through rigorous analysis

Security Review Overview

0	2	0	0	1	0
Critical	High	Medium	Low	Informational	Gas

Assessment Period:	**November 24–26, 2025**
Report Date:	**November 26, 2025**
Report Version:	**1.0**
Reviewed Commit:	`7d55682ddc4301a7b13ae9413095feffd9924566`
Assessment Methods:	Manual Review, Static Analysis Tools

Contents

About Volodymyr Stetsenko

Volodymyr Stetsenko is an independent security researcher focused on:

- **Core Focus:** Manual auditing, automated and static analysis, property-based / fuzz testing, and formal verification of EVM-based protocols.
- **Approach:** Rigorous code review, reasoning about protocol assumptions, and identifying high-impact logic, economic, and architectural weaknesses.
- **Professional Development:** Continuous learning, public audit practice, and transparent documentation of methods.

My workflow combines manual inspection with tool-driven techniques, including static analyzers (e.g., Slither), fuzzers and property-based testing (e.g., Echidna, Foundry fuzzing), symbolic checks, invariant assertions, and formal verification where appropriate to ensure comprehensive coverage of functional and economic risk vectors.

While no one can guarantee 100% security, I commit to giving the audited protocol my full attention, thorough analysis, and maximum effort to uncover risks and strengthen its design.

Volodymyr Stetsenko

Independent Security Researcher

volodymyrstetsenkoaudit@gmail.com

Volodymyr Stetsenko

Volodymyr Stetsenko

Volodymyr Stetsenko

Disclaimer

Important Notice

Volodymyr Stetsenko makes all reasonable efforts to identify vulnerabilities within the reviewed code during the specified period, but holds no liability for the findings presented in this document. A security audit does not constitute an endorsement of the underlying business, product, or protocol.

A smart contract security review **cannot verify the complete absence of vulnerabilities**. This is a time, resource, and expertise-bound effort. **No guarantee of 100% security** is provided, regardless of whether any issues were identified.

The audit was **time-boxed** and focused solely on the security aspects of the Solidity implementation within the defined scope. This report is provided **"AS IS"** without warranty of any kind.

Recommended post-audit measures: subsequent independent reviews, bug bounty programs, on-chain monitoring, and formal verification of critical invariants.

Scope Limitations

Unless explicitly stated, the following areas are **outside the scope** of this security assessment:

- Business logic beyond security implications
- Economic model and tokenomics analysis
- Frontend/backend application security
- Third-party integrations and dependencies
- Deployment scripts and infrastructure
- Future code changes and upgrades

Risk Classification

Severity Matrix

	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Matrix illustrates risk levels combining likelihood and impact.

- **High** — results in a significant loss of protocol assets or severely impacts a large user group.
- **Medium** — results in a moderate loss of protocol assets or moderately impacts users.
- **Low** — results in minor asset loss or affects a small group of users.

- **High** — attack is feasible under reasonable assumptions on-chain, with relatively low cost versus potential loss.
- **Medium** — attack is possible under specific conditions and incentives.
- **Low** — attack requires unlikely assumptions, high cost, or offers little incentive.

Protocol Summary

4.1 Introduction

PasswordStore is a smart contract protocol designed to allow users to store and retrieve passwords on the Ethereum blockchain. The protocol's primary objective is to provide a decentralized password management solution where only the authorized owner can access stored credentials.

4.2 Architecture

The protocol consists of a single contract with the following components:

- **Storage:** Owner address (`s_owner`) and password string (`s_password`) stored in contract state.
- **Access Control:** Owner-only restrictions on password operations (intended).
- **Events:** Password change notification via `SetNewPassword` event.

4.3 Functions

Function	Visibility	Description
`setPassword()`	external	Set a new password (intended: owner only).
`getPassword()`	external view	Retrieve stored password (owner only).

Executive Summary

A time-boxed security review of the **PasswordStore** protocol was conducted by **Volodymyr Stetsenko**, focusing on the security aspects of the application's smart contract implementation.

PasswordStore is a smart contract designed to allow users to store and retrieve passwords on the Ethereum blockchain. The protocol's primary objective is to provide a decentralized password management solution where only the authorized owner can access stored credentials.

5.1 Audit Overview

Attribute	Details
Protocol	PasswordStore
Auditor	Volodymyr Stetsenko
Language	Solidity 0.8.18
Blockchain	Ethereum
Methodology	Manual Code Review, Static Analysis
Review Period	November 24–26, 2025
Commit Hash	`7d55682ddc4301a7b13ae9413095feffd9924566`

5.2 Scope

Contract	Path	nSLOC
`PasswordStore.sol`	`src/PasswordStore.sol`	∼20
	Total	**∼20**

5.3 Findings Summary

Severity	Open	Mitigated	Resolved	Total
Critical	0	0	0	**0**
High	2	0	0	**2**
Medium	0	0	0	**0**
Low	0	0	0	**0**
Informational	1	0	0	**1**
Total	**3**	**0**	**0**	**3**

This table summarizes all findings identified during the assessment, categorized by severity and remediation status. At the time of report finalization, two high-severity issues and one informational finding remain open. No critical, medium, or low-severity issues were identified.

Status definitions:

- **Open**: Issue has not been addressed at the time of reporting.
- **Mitigated**: Risk has been partially reduced through compensating controls.
- **Resolved**: Issue has been fully remediated and verified.

Findings

6.1 High Severity

HIGH

[H-1] Password stored on-chain is visible to anyone, not just the owner

Severity: HIGH Status: Open

Location: `PasswordStore.sol` Variable: `s_password`

Description

The `PasswordStore::s_password` variable is stored on-chain and marked as `private`. However, all data stored on-chain is publicly visible to anyone, regardless of Solidity visibility modifiers. The `private` keyword only prevents other contracts from accessing the variable, but does not prevent reading the data directly from the blockchain storage.

Impact

Anyone can read the password directly from the blockchain, completely defeating the purpose of the protocol. The core invariant *"Others should not be able to access the password"* is broken.

Proof of Concept

The test case below demonstrates how anyone can 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.

Step 1: Create a locally running chain

```
make anvil
```

Step 2: Deploy the contract to the chain

```
make deploy
```

Step 3: Understanding Storage Slots

```
contract PasswordStore {
    address private s_owner;      // Storage slot 0
    string private s_password;    // Storage slot 1
}
```

The first variable `s_owner` is in slot 0, and the second variable `s_password` is in slot 1.

Step 4: Read the storage slot
We use `1` because that's the storage slot of `s_password` in the contract. Replace `<ADDRESS_HERE>` with your actual contract address from Step 2:

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

You'll get an output that looks like this:

```
0x6d7950617373776f726400000000000000000000000000000000000000000014
```

Step 5: Decode the hex to readable string

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

And get an output of:

```
myPassword
```

Recommended Mitigation

Due to the transparent nature of blockchain, storing passwords on-chain is fundamentally insecure. All data stored on-chain is publicly readable, regardless of Solidity visibility modifiers.
The overall architecture should be reconsidered. Possible approaches:

1. **Store password hashes instead of passwords** — This allows verification without exposing the actual password. Users can prove they know the password without revealing it.

2. **Use off-chain storage** — Store passwords in an encrypted database off-chain, and use the blockchain only for access control logic.

3. **Reconsider the use case** — Evaluate whether this functionality requires blockchain technology, as password storage conflicts with blockchain's core transparent properties.

Recommended: Implement password hashing (Option 1) if retrieval isn't required, or move to off-chain storage (Option 2) if users need to retrieve passwords.

HIGH

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

Severity: HIGH Status: Open

Location: `PasswordStore.sol` Function: `setPassword()`

Description

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

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

Impact

Anyone can set/change the stored password, severely breaking the contract's intended functionality.

Proof of Concept

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

```
function test_anyone_can_set_password(address randomAddress) public
    {
    vm.assume(randomAddress != owner);
    vm.prank(randomAddress);
    string memory expectedPassword = "myPassword";
    passwordStore.setPassword(expectedPassword);

    vm.prank(owner);
    string memory actualPassword = passwordStore.getPassword();
    assertEq(actualPassword, expectedPassword);
}
```

Run the test:

```
forge test --mt test_anyone_can_set_password
```

The test passes, proving that any non-owner address can successfully change the password.

Recommended Mitigation

Add an access control check to the `PasswordStore::setPassword` function:

```
function setPassword(string memory newPassword) external {
    if (msg.sender != s_owner) {
        revert PasswordStore__NotOwner();
    }
    s_password = newPassword;
    emit SetNewPassword();
}
```

6.2 Informational

INFORMATIONAL

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

Severity: INFO Status: Open

Location: `PasswordStore.sol` Function: `getPassword()`

Description

The `PasswordStore::getPassword` function signature is `getPassword()` which takes no parameters, however the natspec indicates it should be `getPassword(string)` and describes a `newPassword` parameter that doesn't exist in the function.

```
/*
 * @notice This allows only the owner to retrieve the password.
 * @param newPassword The new password to set.   // INCORRECT
 */
function getPassword() external view returns (string memory) {
    if (msg.sender != s_owner) {
        revert PasswordStore__NotOwner();
    }
    return s_password;
}
```

Impact

The natspec is incorrect, which can lead to confusion for developers and auditors reviewing the code.

Recommended Mitigation

Remove the incorrect natspec line:

```
/*
 * @notice This allows only the owner to retrieve the password.
 * @return The stored password string.
 */
function getPassword() external view returns (string memory) {
    if (msg.sender != s_owner) {
        revert PasswordStore__NotOwner();
    }
    return s_password;
}
```

Volodymyr Stetsenko

Lead Security Researcher

volodymyrstetsenkoaudit@gmail.com

Volodymyr Stetsenko

Volodymyr Stetsenko

Volodymyr Stetsenko

Securing the future of decentralized finance
