PatentdApp Audit Report

Sidrah Ahmed

January 14, 2024

PatentdApp Initial Audit Report

January 14, 2024

PatentdApp Audit Report

Sidrah Ahmed

January 14, 2024

PatentdApp Audit Report

Prepared by: Sidrah Ahmed

Lead Auditors:

- Sidrah Ahmed

Assisting Auditors:

- None

Table of contents

See table

Disclaimer

I make all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit by me 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
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

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

`b8b25762915dcb9011cf6eaba4da31e0b4b2ab66`

Scope

```
src/
--- PatentdApp.sol
```

Protocol Summary

PatentdApp is a decentralized application built on the Ethereum blockchain. It is designed exclusively for a single user to manage their passwords securely. The app allows users to register patents, pay associated fees, retrieve patent information, and check the approval status of a patent. Its primary objective is to provide a secure and efficient platform for individual users to handle their patent registrations and fees.

Roles

- Owner: The Owner is the person who deploys the smart contract. The owner is the only one who can interact with the contract, including viewing newly registered patents, getting fees, retrieving information about registered patents, and approving a patent.
- User: The User is the individual who connects their Ethereum wallet to the dApp. They can register a new patent by providing a name, title, and description, and pay the associated fee. They can view all registered patents, check the approval status of each patent, and retrieve details of a specific patent using its ID.

For this contract, only the owner should be able to interact with the contract.

Executive Summary

Issues found

Severity	Number of issues found
High	1
Medium	0
Low	2
Info	2
Gas Optimizations	0
Total	5

Findings

High

[H-1] Potential Reentrancy Vulnerability Detected in `payPatentFee::PatentdApp.sol`

Relevant GitHub Links

Visit GitHub Repository.

Summary

The `PatentdApp::payPatentFee` function in the `PatentdApp` contract is potentially susceptible to a reentrancy attack. Reentrancy attacks can disrupt the contract's normal operations and lead to unauthorized transactions and state changes. Specifically, the function makes an external call to transfer Ether before updating its state, which could potentially allow an attacker to exploit this by making a recursive call to `payPatentFee` before the state is updated, leading to multiple transactions being processed in a single call.

Vulnerability Details

The PatentdApp.payPatentFee function makes an external call to owner.transfer(ownerFee) before updating its state variables. This could potentially allow for reentrancy attacks, as an attacker could manipulate the external call to execute additional function calls before the state variables are updated.

Impact

A successful reentrancy attack could lead to unauthorized transactions, potential loss of funds, and disruption of the contract's normal operations. It could also potentially allow an attacker to manipulate the state of the contract, leading to unexpected behavior.

Recommended Mitigation:

Reentrancy Vulnerability: To avoid reentrancy attacks, you can follow the Checks-Effects-Interactions pattern. This involves moving the state-changing part (effects) to the beginning of the function, performing any external calls afterwards (interactions). Here's how you can modify the payPatentFee function:

```
// State-changing part (effects)
// Create a new Patent struct and store it in the 'patents' array.
patents[nextPatentId] = Patent({
  ...
});

// Perform external call (interaction)
bool success = owner.send(ownerFee);
require(success, "Failed to send Ether to owner");
```

Low

[L-01] Non-Immutable PatentdApp.owner Variable Found in PatentdApp.sol (Reference: PatentdApp.sol#26)

Relevant GitHub Links

Visit GitHub Repository.

Summary

The PatentdApp.owner variable in the PatentdApp.sol contract is not declared as immutable. This means that its value can potentially be altered after the contract is deployed, which could lead to security risks and increased gas costs. The lack of immutability could also allow for unauthorized changes to the owner's

identity after the contract is live, which could compromise the integrity and functionality of the contract.

Vulnerability Details

The PatentdApp.owner variable in the PatentdApp.sol contract is not declared as immutable. This means that its value can potentially be changed after the contract is deployed, which could lead to security risks. Immutable variables in Solidity are read-only and can only be assigned a value during contract creation. This feature helps to prevent unauthorized modifications to the variable, enhancing the security of the contract.

Impact

Using non-immutable variables can lead to increased gas costs due to the need for storage variable reads. Additionally, the lack of immutability can introduce security risks, as the variable can be changed after the contract is deployed, potentially leading to unauthorized actions.

Recommended Mitigation:

The non-immutable owner variable in the PatentdApp contract can be mitigated by declaring it as immutable. This prevents the owner from being changed after the contract is deployed. Here's the modification:

```
// Declare an immutable state variable to store the
//owner's address, marked as payable.
address payable immutable owner;

constructor() {
  // Assign the contract creator's address as the owner.
    owner = payable(msg.sender);
}
```

[L-02] Uninitialized Variable patentTitles in PatentdApp.sol (Reference: PatentdApp.sol#32).

Relevant GitHub Links

Visit GitHub Repository.

Description

The problem lies in the `PatentdApp::payPatentFee` function. The `PatentdApp::patentTitles` mapping is being updated before the new patent is added to the patents array. This means that if the function execution stops for any reason after updating `PatentdApp::patentTitles`

but before adding the new patent, the title will be marked as used in the `PatentdApp::patentTitles` mapping, but the patent won't be added to the patents array.

Impact

If the function execution stops after updating the `PatentdApp::patentTitles` mapping but before adding the new patent to the patents array, the title will be marked as used in the `PatentdApp::patentTitles` mapping, but the patent won't be added to the patents array. This inconsistency could lead to logical errors and system inconsistencies.

Proof of Concept

After checking that a patent with the given title doesn't already exist and that the correct amount of Ether has been sent, the function proceeds to create a new Patent struct and store it in the patents array. After that, it marks the patent title as used in the `PatentdApp::patentTitles` mapping.

Recommended Mitigation

To fix this issue, the line where `PatentdApp::patentTitles` is updated should be moved to after the new patent is added to the patents array. This ensures that the `PatentdApp::patentTitles` mapping is updated correctly only after a new patent is successfully added.

Here's how the modified `PatentdApp::payPatentFee` function should look:

```
// Create a new Patent struct and store it in the 'patents' array.
patents[nextPatentId] = Patent({
    patentId: nextPatentId, // Assign the next available patent ID.
    name: patentName,
    title: patentTitle,    // Assign the provided patent title.
    description: patentDescription, // Assign the provided patent description.
    sender: msg.sender, // Set the sender's address to the transaction sender.
    filingDate: filingDate, // Set the filing date.
    expirationDate: expirationDate, // Set the calculated expiration date.
    isApproved: isApproved // Set the approval status.
});

// Mark the patent title as used in the 'patentTitles' mapping.
patentTitles[patentTitle] = true;
```

Informationals

[I-01] Outdated Solidity Version Used in PatentdApp.sol (PatentdApp.sol#2)

Description:

```
pragma solidity ^0.8.0;
```

The PatentdApp.sol contract is compiled with Solidity version ^0.8.0, which allows older versions to compile the contract. This could potentially expose the contract to vulnerabilities that were fixed in later versions of Solidity, and could also introduce syntax or semantic differences that could break the contract.

Impact: Using an outdated Solidity version could expose the contract to vulnerabilities and bugs fixed in newer versions, potentially introducing syntax or semantic differences that could break the contract

Recommended Mitigation: Update the pragma version to the latest stable version of Solidity. As of today's date (January 19, 2024), the latest stable version of Solidity is likely to be 0.9.x. Therefore, the pragma version should be updated as follows:

```
pragma solidity ^0.8.0;
+     pragma solidity ^0.9.0;
```

[I-02] Misleading cryptocurrency unit in PatentdApp.sol

Description:

```
require(msg.value>=0.20 ether,"Please pay a fee of at least 1.60 bnb for the patent.");
```

This requirement could be misleading as the contract is deployed on the Ethereum blockchain, which doesn't accept BNB tokens. The use of 'bnb' in the error message could confuse users, leading them to believe that the system accepts BNB, which is incorrect.

Impact: This could lead to confusion among users who are familiar with BNB and may attempt to pay the fee using BNB, despite the fact that the contract does not accept it. Users might waste time trying to figure out why their transaction is failing.

Recommended Mitigation: To avoid this confusion, change the units to ether. Here's a suggested revision:

```
require(msg.value >= 0.20 ether,"Please pay a fee of at least 1.60 bnb for the patent.");

+require(msg.value>=0.20 ether,"Please pay a fee of at least 1.60 ether for the patent.");
-require(msg.value>=0.20 ether,"Please pay a fee of at least 1.60 bnb for the patent.");
```

Tools Used

Slither was used for analyzing the smart contract. It helped identify potential vulnerabilities and areas for improvement in the contract code.

Conclusion

In conclusion, the identified issues can be addressed as follows:

1. Reentrancy vulnerability in the payPatentFee function can be mitigated by adhering to the Checks-Effects-Interactions pattern, ensuring state-changing operations occur before external calls.
2. The owner variable should be declared as immutable to prevent unauthorized changes after contract deployment.
3. The uninitialized patentTitles variable should be updated after a new patent is added to the patents array to ensure accurate tracking of patent titles.
4. The outdated Solidity version should be updated to the latest stable release to benefit from the latest features and security improvements.
5. The misleading cryptocurrency unit in the contract should be clarified to avoid confusion.

These should help resolve the identified issues and enhance the security and clarity of the contract.