Cite(APA):
Alici, U.I., Oksuztepe, A., Kilinceker, O., Karaarslan, E. (2023). Generative AI for Smart Contract Security Testing Case Study: OpenAI ChatGPT

OpenAI ChatGPT for Smart Contract Security Testing: Discussion and Future Directions

Uzay Işın ALICI
Computer Engineering
Faculty of Engineering
Mugla Sitki Kocman University
Mugla, Türkiye,
uzayisinalici@gmail.com
0009-0005-6602-4157

Ayça ÖKSÜZTEPE
Computer Engineering
Faculty of Engineering
Mugla Sitki Kocman University
Mugla, Türkiye,
aycaoksuztepee@gmail.com
0009-0002-6858-712X

Onur Kilincceker
Computer Science
Faculty of Science
University of Antwerp and Flanders Make vzw.
Antwerp, Belgium
onur.kilincceker@uantwerpen.be
0000-0001-5996-4398

Enis KARAARSLAN
Computer Engineering
Faculty of Engineering
Mugla Sitki Kocman University
Mugla, Türkiye,
enis.karaarslan@mu.edu.tr
0000-0002-3595-8783

Abstract -

Decentralized applications (Dapps) have the potential to revolutionize many systems and are increasingly used, eg. in Web3 solutions. Smart contracts often manage valuable assets and sensitive data as the loss of any digital asset can be irreversible. There is a growing need for the security of these systems as any vulnerability can lead to irreversible financial losses. However, traditional software development and testing systems fall short of providing security for Blockchain technologies and Web3 developers. Considering the current potential of artificial intelligence, it can be used as a solution to secure Dapps. LLMs can analyze smart contract code for vulnerabilities, generate test cases, and provide recommendations for improvement. In this article, we question the use of ChatGPT for this purpose. It is shown that ChatGPT has the potential to aid developers. Advantages, limitations and improvement methods are given. Possible future work is given.

Keywords: Large Language Model, Artificial Intelligence, Smart Contract Security, Smart Contract Testing, Blockchain Security, Decentralized Applications, Vulnerability Analysis, Test Case Generation, Code Analysis, Software Testing, ChatGPT

I. INTRODUCTION

We are in an age where the importance of decentralized solutions is understood, and many implementations are emerging. Blockchain technology and smart contract-based decentralized applications promise to change the way we do business in many areas. Decentralized applications (Dapps) have the potential to revolutionize many systems, but there is an emerging need for the security of these systems. Loss of any digital asset may be irreversible because of the nature of the decentralized systems. We need to be more careful as smart contracts work autonomously and immutable records are used in the blockchain. We need to develop safe and reliable applications especially to avoid problems, especially in value transfers. However, the language and environments for Dapps are not mature enough yet. Traditional software development and testing systems are not sufficient for secure coding. There is a need for more sample applications and guidelines on how to develop and test such code [1].

There are a lot of developments in Generative AI and especially in the area of text-based chatbots. ChatGPT is one of the most widely used with its public version. This study aims to answer following research questions;

- Can we use ChatGPT for smart contract testing?
  - If yes, how to employ ChatGPT for this purpose?
  - What are advantages and disadvantages over other methods?
- How can we improve smart contract testing using ChatGPT?

We will also discuss its limitations and challenges. Section 2 covers basic information about smart contracts, their features, and testing methods. We summarize the necessary foundational knowledge on the topic. The proposed approach and essential information are described in Section 3. Section 4 contains information on how we implemented the proposed approach. Finally, our paper is concluded in Section 5, which summarizes the study.

II. FUNDAMENTALS

A. Smart Contracts and Determinism

Nick Szabo introduced the concept of smart contracts as programs that formalize and secure the relations between parties communicating over the public network [2]. Smart contracts are first implemented on the Ethereum blockchain network. Determinism is required for smart contracts to be enforceable. Determinism requires that the codes be run on all blockchain network nodes, resulting in the same result state and transaction state in the blockchain system. Decentralized platforms also do not allow non-deterministic functions such as randomness [3].

The working steps of a smart contract (contract) is given in Figure 1 [1]. The software developer loads the smart contract code to the blockchain system as a record (Step 1). This record has an address by which it can be called. The transaction records are also stored on the blockchain. These records can be publicly reachable on public blockchains such as Ethereum. We can also limit the reachability of these records among certain institutions on the private (enterprise) blockchains. The parties interact with the smart contract by reaching the code address through their accounts on the blockchain (Step 2). The code executes itself if the conditions set in the smart contract are fulfilled. Records of the transaction calls are written to the blockchain (Step 3). Audit processes are also possible by examining the transaction records afterward (Step 4). Auditors are specific control mechanisms for the implementation field and can monitor the Dapps if required.

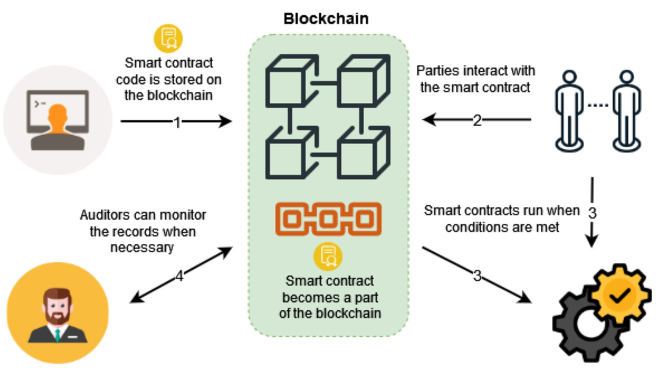


Fig 1. Working Steps of a Smart Contract [1]

Smart contracts can be upgraded when needed. New versions can be loaded on the blockchain and mechanisms such as proxy patterns can be used [4]. However special care should be taken for security. Smart contract development challenges and opportunities are discussed in [7].

B. Smart Contract Testing

Smart contracts are irreversible and immutable pieces of code, so it is impossible to fix bugs in smart contracts once they are deployed. The biggest reason smart contracts need to be tested before they are deployed is that they control sensitive data and valuable assets. Even the most minor bug can cause substantial financial losses. There are two main types of testing: automated testing and manual testing.

B.1. Automated Testing

Automated testing is an automated analysis method that finds problems such as faults and errors in the smart contract, and runs iteratively to make the contract perfect. We can classify the methods as functional testing, static and dynamic analysis.

Functional testing is testing the functionality of each smart contract operation with specific inputs. Functional testing allows us to ensure that the smart contract works as intended in various scenarios. Different types are possible such as Unit Testing, Integration Testing, and System Testing.

Static analysis is testing common security vulnerabilities with the help of source code or bytecode without deploying the smart contract.[9] *Dynamic analysis* is testing smart contracts by dynamically interacting with them at execution and giving them different inputs.

In addition to these traditional methods, there are *Mutation Testing* and *Fuzz Testing*, which are very successful and new testing methods for smart contract testing [8]. *Mutation Testing* is a unit test type-based test that evaluates the contract's ability to detect bugs in the source code by generating mutations or minor changes to reveal flaws. *Fuzz Testing* is a method of feeding the smart contract with unexpected inputs to create confusion under unusual situations.

B.2. Manual Testing

Manual testing is the testing of smart contracts manually by a human tester. These are possible with code audits. There are also bug bounty programs to encourage disclosure of weaknesses.

B.3. Analysis of the Tests

We classified the smart contract tests with their properties. The purpose of each smart contract test type, their performing times and example tools are given in Table 1. The strength and weakness of each and the required execution time is given in Table 2.

The selection of the most suitable testing approach for a smart contract is influenced by various factors, including the project requirements, developmental phase, available budget, and resources. In order to detect vulnerabilities in smart contracts with testing and get satisfactory results, more than one of the given tools should perform simultaneously [1]. Simultaneously utilizing multiple tools can lead to an overconsumption of resources and budgetary constraints. Even having enough budget, and employing various resources and tools may not yield optimal outcomes. With the inadequacy of the available tools, the testing power of artificial intelligence plays a crucial role.

Table 1. Properties of Smart Contract Tests

Testing Type	**Purpose**	**Performing Time**	**Example Tools**
Functional Testing	Verify functional requirements	After development	Truffle, Openzeppelin
Unit Testing	Test individual functions	During development	Remix Test, Brownie
Integration Testing	Verify interactions between each component	During development	Remix Test, Brownie
System Testing	Test as a whole, the final phase of functional testing	After all other tests	Testnet, Devnet
Static Analysis	Identify common security vulnerabilities from source/bytecode	Before deployment	Mythril, Slither
Dynamic Analysis	Test behavior under various conditions during execution	After deployment	Echidna, Harvey, Fuzzing [9]
Bug Bounty	Encourage people to fix problems with rewards	Throughout deployment	Immunefi, HackerOne
Mutation Testing	Verify the test's performance by altering (mutating) the source code	Before deployment	Mutmut, Solidity-coverage
Fuzz Testing	Confuse the smart contract with interesting inputs that are sent repeatedly till finding vulnerabilities [11].	Before deployment	Echidna, Manticore

Table 2. Properties of Smart Contract Tests

Testing Type	**Strength**	**Weakness**	**Time Required (Low, Moderate, High)**
Functional Testing	Validation	Can not handle all scenarios [7]	Moderate
Unit Testing	Fast and effective	Not Comprehensive	Low
Integration Testing	Cost reduction effect	Require coordination between various components	High
System Testing	End-to-end testing	Difficulty in simulating real-world scenarios and environments	High
Static Analysis	Early defect detection	Inability to detect all types of defects, particularly those related to runtime behavior or interactions between different parts of the system	Low
Dynamic Analysis	Identification of runtime issues	Time-consuming	High
Bug Bounty	Reach different minds	Trust in the third party	High
Mutation Testing	Detects problems that cannot be found with other testing methods	Can not reach a sufficient number of mutations	High
Fuzz Testing	Identify the problem with why and how	Significant amount of false negatives [10]	Low

C. Generative AI and ChatGPT

Generative AI is a paradigm mainly used to create new and unique content with trained data. Its usage areas are endless, and the chatbot is one of the most popular [5]. Computer science pioneers such as Turing coined “Can computers think?” [6]. As large language models (LLMs) can be processed more efficiently, the answer now can be terrifying for some.

We had access to ChatGPT[1] as the OpenAI company made it publicly available and without a cost (for a while). Now our question is “Can we use ChatGPT for security testing of smart contracts?”

III. THE PROPOSED APPROACH

Our approach is depicted in Fig. 2 starts with uploading smart contract under consideration and the necessary query to ChatGPT.

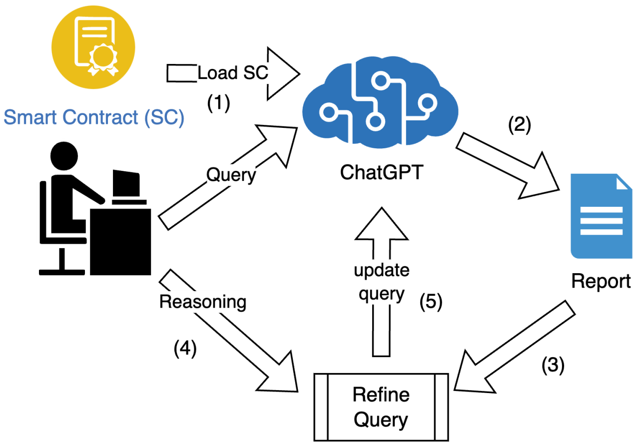


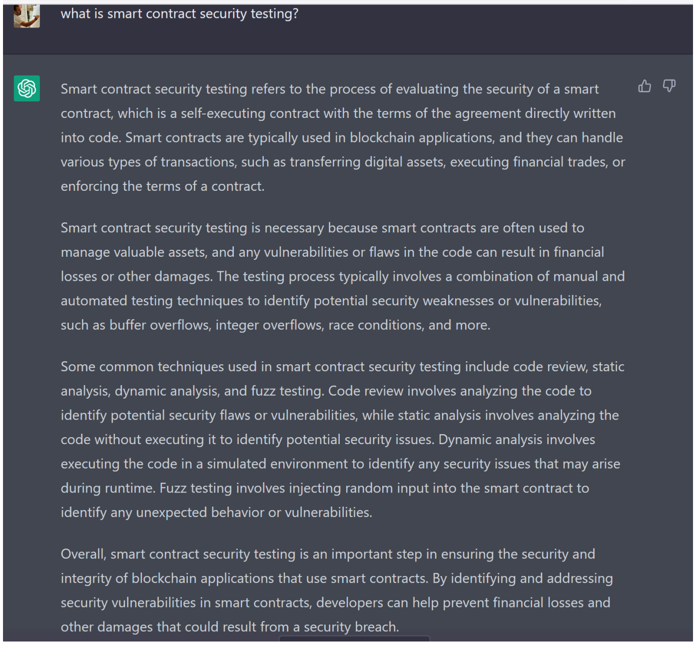
Fig 2. General Overview of the approach

The output of ChatGPT is a report that contains potential security vulnerabilities of the smart contract. Then, the user or tester refines this report by reasoning to deepen and specify the security flaws. The updated query is then input back into ChatGPT. Similar steps are continued until the user gathers sufficient information about the security vulnerabilities. At the end of the process, the user has a comprehensive test report about the smart contract under consideration. This test report obtained using ChatGPT includes a general overview of security vulnerabilities, along with detailed information for each system flaw.

[1] ChatGPT, https://openai.com/blog/chatgpt/

IV. IMPLEMENTATION AND RESULTS

We asked the ChatGPT about smart contract testing, and the output is given in Fig. 2 and Fig. 3. We then extended our studies by uploading the smart contracts and implementing extended tests.


what is smart contract security testing?

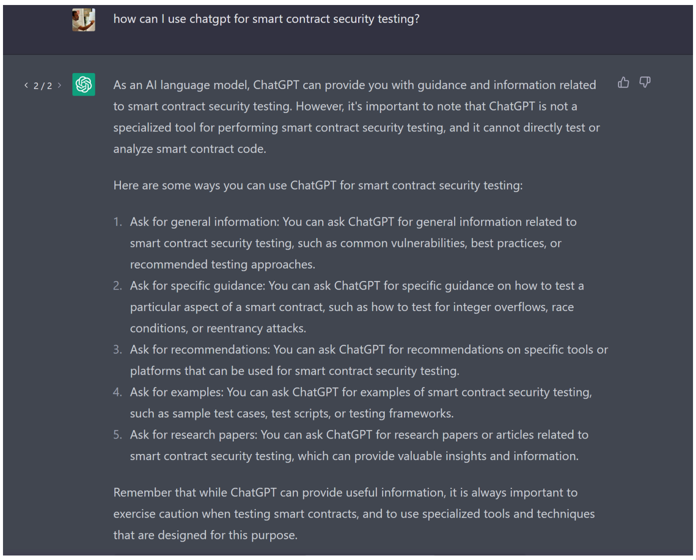
Smart contract security testing refers to the process of evaluating the security of a smart contract, which is a self-executing contract with the terms of the agreement directly written into code. Smart contracts are typically used in blockchain applications, and they can handle various types of transactions, such as transferring digital assets, executing financial trades, or enforcing the terms of a contract.

Smart contract security testing is necessary because smart contracts are often used to manage valuable assets, and any vulnerabilities or flaws in the code can result in financial losses or other damages. The testing process typically involves a combination of manual and automated testing techniques to identify potential security weaknesses or vulnerabilities, such as buffer overflows, integer overflows, race conditions, and more.

Some common techniques used in smart contract security testing include code review, static analysis, dynamic analysis, and fuzz testing. Code review involves analyzing the code to identify potential security flaws or vulnerabilities, while static analysis involves analyzing the code without executing it to identify potential security issues. Dynamic analysis involves executing the code in a simulated environment to identify any security issues that may arise during runtime. Fuzz testing involves injecting random input into the smart contract to identify any unexpected behavior or vulnerabilities.

Overall, smart contract security testing is an important step in ensuring the security and integrity of blockchain applications that use smart contracts. By identifying and addressing security vulnerabilities in smart contracts, developers can help prevent financial losses and other damages that could result from a security breach.

Fig 2. Asking ChatGPT on Smart Contract Testing


how can I use chatgpt for smart contract security testing?

< 2/2 >

As an AI language model, ChatGPT can provide you with guidance and information related to smart contract security testing. However, it's important to note that ChatGPT is not a specialized tool for performing smart contract security testing, and it cannot directly test or analyze smart contract code.

Here are some ways you can use ChatGPT for smart contract security testing:

1. Ask for general information: You can ask ChatGPT for general information related to smart contract security testing, such as common vulnerabilities, best practices, or recommended testing approaches.
2. Ask for specific guidance: You can ask ChatGPT for specific guidance on how to test a particular aspect of a smart contract, such as how to test for integer overflows, race conditions, or reentrancy attacks.
3. Ask for recommendations: You can ask ChatGPT for recommendations on specific tools or platforms that can be used for smart contract security testing.
4. Ask for examples: You can ask ChatGPT for examples of smart contract security testing, such as sample test cases, test scripts, or testing frameworks.
5. Ask for research papers: You can ask ChatGPT for research papers or articles related to smart contract security testing, which can provide valuable insights and information.

Remember that while ChatGPT can provide useful information, it is always important to exercise caution when testing smart contracts, and to use specialized tools and techniques that are designed for this purpose.

Fig 3. Asking ChatGPT on Smart Contract Testing

Implementation …

Even though the word count of the codes didn’t exceed the limits of the used ChatGPT model, we had timeouts when uploading the large files, but there are several ways we can solve this[2]. These are given as follows:

1) Divide the comprehensive document into 500-word chunks and upload it to GPT-4. Furthermore, while loading each part, inform GPT-4 that it will continue to load subsequent parts.

2) To ensure GPT-4 has all the content, test its memory by asking it for specific parts again.

3) Ask GPT-4 to provide its output in smaller chunks, and then the rest of the text can continue to load without timing out. Parts of 1,000 characters are safe.

V. CONCLUSION

It's shown that we can use ChatGPT for smart contract testing. ChatGPT has the potential to analyze smart contract code for vulnerabilities, automatically generate relevant test cases, and offer insightful recommendations for improvement. This can potentially lead to a more efficient and comprehensive security testing process than traditional methods. While ChatGPT shows promise, we should be aware of its limitations. ChatGPT can analyze code and identify potential issues, but its recommendations might require refinement and interpretation by human experts.

A method is proposed for improving smart contract testing using ChatGPT. For future studies, we can work on exploring advanced prompt engineering techniques. AI can be used to generate more effective prompts to elicit more accurate and insightful responses regarding smart contract vulnerabilities.ChatGPT can also be integrated with existing testing frameworks. Specialized LLMs can be developed for smart contract security testing.

REFERENCES

[1] Karaarslan, E., Birim, M. (2021). Blokzincirde Güvenli ve Güvenilir Uygulama Geliştirme Temelleri, Siber Güvenlik ve Savunma: Blokzinciri ve Kriptografi, p 1-48, Nobel Yayınevi

[2] N. Szabo, "Formalizing and securing relationships on public networks," First Monday, 1997.

[3] K. Chatterjee, A. K. Goharshady and A. Pourdamghani "Probabilistic smart contracts: Secure randomness on the blockchain," Presented at 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). IEEE, 2019. p. 403-412.

[4] Upgrading smart contracts, https://ethereum.org/en/developers/docs/smart-contracts/upgrading/

[5] Aydin, Ö., Karaarslan, E. (2023). Is ChatGPT Leading Generative AI? What is Beyond Expectations?

[6] Turing, A. M. (1951). Can digital computers think?. The Turing Test: Verbal Behavior as the Hallmark of Intelligence, 111-116.

[7] Zou, W., Lo, D., Kochhar, P. S., Le, X. B. D., Xia, X., Feng, Y., ... & Xu, B. (2019). Smart contract development: Challenges and opportunities. IEEE Transactions on Software Engineering, 47(10), 2084-2106.

[8] Imperius, N. P., & Alahmar, A. D. (2022). Systematic Mapping of Testing Smart Contracts for Blockchain Applications. *IEEE ACCESS*, *10*, 112845-112857.

[9] Smart Contract Testing, https://ethereum.org/en/developers/docs/smart-contracts/testing/

[10] B. Jiang, Y. Liu and W. K. Chan, "ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection," *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Montpellier, France, 2018, pp. 259-269, doi: 10.1145/3238147.3238177.

[11] Fuzz Testing, https://www.immunebytes.com/blog/getting-started-with-smart-contract-fuzzing/

Appendix

[2] A report by GPT4 on how to create large documents using GPT4, https://docs.google.com/document/d/15O-BI55FuebEoofY1k05skE1kBf1XBa_5sf4REzdiVA/edit

Cite(APA):
Alici, U.I., Oksuztepe, A., Kilinceker, O., Karaarslan, E. (2023). Generative AI for Smart Contract Security Testing Case Study: OpenAI ChatGPT

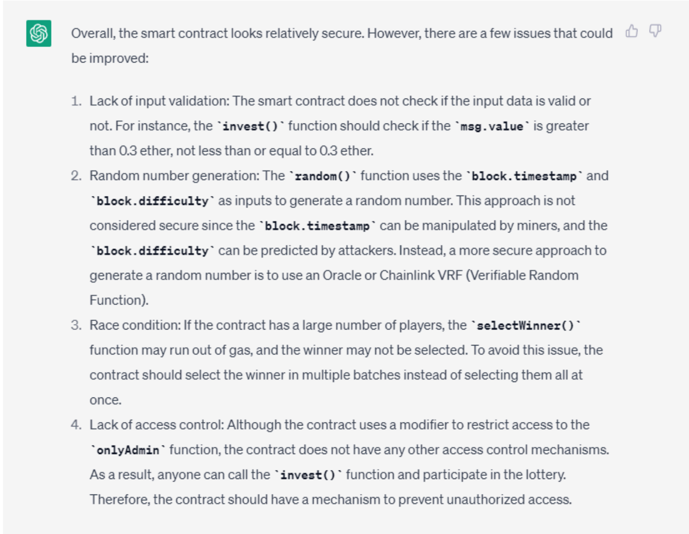
Appendix 1 -

Detected Security Vulnerabilities for Lottery Smart Contract by ChatGPT

First Attempt :

Overall, the smart contract looks relatively secure. However, there are a few issues that could be improved:

1. *Lack of input validation: The smart contract does not check if the input data is valid or not. For instance, the invest() function should check if the msg.value is greater than 0.3 ether, not less than or equal to 0.3 ether.*
2. *Random number generation: The random() function uses the block.timestamp and block.difficulty as inputs to generate a random number. This approach is not considered secure since the block.timestamp can be manipulated by miners, and the block.difficulty can be predicted by attackers. Instead, a more secure approach to generate a random number is to use an Oracle or Chainlink VRF (Verifiable Random Function).*
3. *Race condition: If the contract has a large number of players, the selectWinner() function may run out of gas, and the winner may not be selected. To avoid this issue, the contract should select the winner in multiple batches instead of selecting them all at once.*
4. *Lack of access control: Although the contract uses a modifier to restrict access to the onlyAdmin function, the contract does not have any other access control mechanisms. As a result, anyone can call the invest() function and participate in the lottery. Therefore, the contract should have a mechanism to prevent unauthorized access.*


Overall, the smart contract looks relatively secure. However, there are a few issues that could be improved:

1. Lack of input validation: The smart contract does not check if the input data is valid or not. For instance, the `invest()` function should check if the `msg.value` is greater than 0.3 ether, not less than or equal to 0.3 ether.
2. Random number generation: The `random()` function uses the `block.timestamp` and `block.difficulty` as inputs to generate a random number. This approach is not considered secure since the `block.timestamp` can be manipulated by miners, and the `block.difficulty` can be predicted by attackers. Instead, a more secure approach to generate a random number is to use an Oracle or Chainlink VRF (Verifiable Random Function).
3. Race condition: If the contract has a large number of players, the `selectWinner()` function may run out of gas, and the winner may not be selected. To avoid this issue, the contract should select the winner in multiple batches instead of selecting them all at once.
4. Lack of access control: Although the contract uses a modifier to restrict access to the `onlyAdmin` function, the contract does not have any other access control mechanisms. As a result, anyone can call the `invest()` function and participate in the lottery. Therefore, the contract should have a mechanism to prevent unauthorized access.

Last Attempt :

I have reviewed the provided Solidity smart contract and identified some potential security vulnerabilities that need to be addressed:

Randomness vulnerability:

1. The function random() uses block.timestamp and block.difficulty to generate a random number to determine the winner. However, this method is vulnerable to manipulation by miners as they can manipulate the block.timestamp to produce a predictable outcome. A better approach is to use an external randomness source, such as Chainlink VRF or other secure sources of randomness.

Reentrancy vulnerability:

2. The selectWinner() function transfers the entire contract balance to the winner, which can lead to

reentrancy attacks if the winner's contract has a fallback function that calls back into the lottery contract. To prevent this, you should transfer the funds before updating the state variables.

Denial-of-service (DoS) vulnerability:

3. The getBalance() function can be used by anyone, including attackers, to consume a lot of gas by calling it repeatedly, leading to a DoS attack. You should consider adding a limit to the number of times this function can be called within a certain time frame or remove it entirely if it's not necessary.

Lack of input validation:

4. The invest() function does not validate the input amount, which could lead to losing funds due to typos or incorrect inputs. You should consider adding input validation to ensure that the correct amount is deposited.

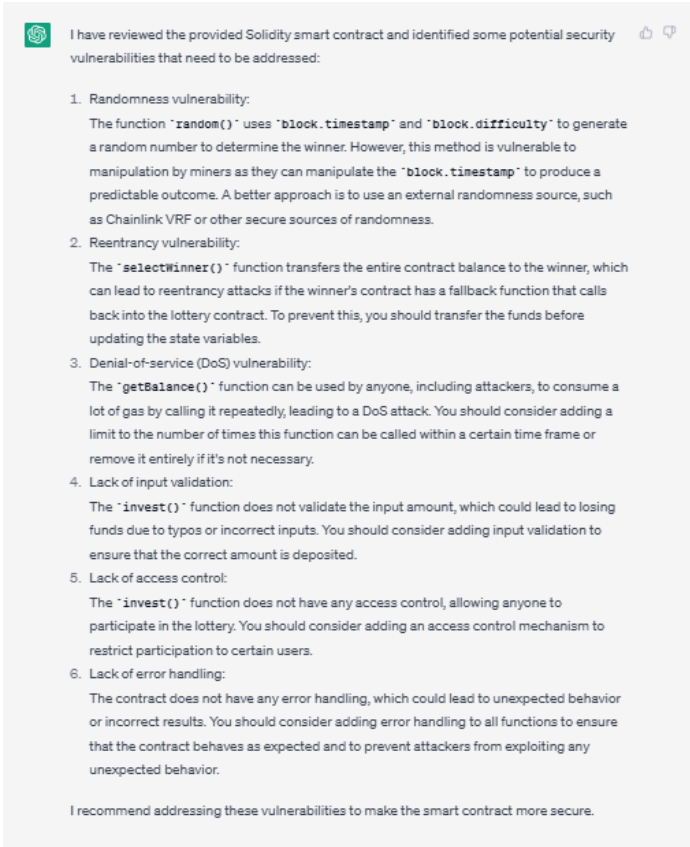
Lack of access control:

5. The invest() function does not have any access control, allowing anyone to participate in the lottery. You should consider adding an access control mechanism to restrict participation to certain users.

Lack of error handling:

6. The contract does not have any error handling, which could lead to unexpected behavior or incorrect results. You should consider adding error handling to all functions to ensure that the contract behaves as expected and to prevent attackers from exploiting any unexpected behavior.

I recommend addressing these vulnerabilities to make the smart contract more secure.



I have reviewed the provided Solidity smart contract and identified some potential security vulnerabilities that need to be addressed:

1. Randomness vulnerability:
   The function `random()` uses `block.timestamp` and `block.difficulty` to generate a random number to determine the winner. However, this method is vulnerable to manipulation by miners as they can manipulate the `block.timestamp` to produce a predictable outcome. A better approach is to use an external randomness source, such as Chainlink VRF or other secure sources of randomness.
2. Reentrancy vulnerability:
   The `selectWinner()` function transfers the entire contract balance to the winner, which can lead to reentrancy attacks if the winner's contract has a fallback function that calls back into the lottery contract. To prevent this, you should transfer the funds before updating the state variables.
3. Denial-of-service (DoS) vulnerability:
   The `getBalance()` function can be used by anyone, including attackers, to consume a lot of gas by calling it repeatedly, leading to a DoS attack. You should consider adding a limit to the number of times this function can be called within a certain time frame or remove it entirely if it's not necessary.
4. Lack of input validation:
   The `invest()` function does not validate the input amount, which could lead to losing funds due to typos or incorrect inputs. You should consider adding input validation to ensure that the correct amount is deposited.
5. Lack of access control:
   The `invest()` function does not have any access control, allowing anyone to participate in the lottery. You should consider adding an access control mechanism to restrict participation to certain users.
6. Lack of error handling:
   The contract does not have any error handling, which could lead to unexpected behavior or incorrect results. You should consider adding error handling to all functions to ensure that the contract behaves as expected and to prevent attackers from exploiting any unexpected behavior.

I recommend addressing these vulnerabilities to make the smart contract more secure.