International Research Journal of Innovations in Engineering and Technology (IRJIET)

ISSN (online): 2581-3048
Volume 7, Issue 10, pp 660-667, October-2023
https://doi.org/10.47001/IRJIET/2023.710085

Strengthening Smart Contracts: An Investigation into Vulnerability Detection and Mitigation in Smart Contracts with a Web Application Based Tool

[1]Kalana Jayasinghe, [2]Ravindu Illeperuma, [3]Induja Abeyrathne, [4]Janindu Abeywickrama, [5]Chethana Liyanapathirana

[1,2,3,4,5]Faculty of Computing, Sri Lanka Institute of Information Technology, Malabe, 10115, Sri Lanka

***Abstract* - This research paper explores enhancing the security of Ethereum smart contracts by addressing four core vulnerabilities: reentrancy, integer overflow/underflow, delegate call exploits, and true randomness. The final purpose of this study is the creation of a web application designed to automate vulnerability detection in Solidity code. This tool offers several advantages, such as automating the identification process, ensuring comprehensive analysis, and minimizing manual intervention. Furthermore, its cost-effective nature provides advanced security scanning, making it accessible to both individual developers and resource constrained organizations. The application significantly reduces the time required for vulnerability assessment. Moreover, its user-friendly interface accommodates users with various levels of technical analysis. By mitigating critical vulnerabilities and offering a practical, automated, and user-friendly approach, this research contributes to improving the security of Ethereum smart contracts in the blockchain ecosystem.**

Keywords: Smart Contract, Random Number Generator, Gas Fee, Block Chain, Vulnerability.

I. INTRODUCTION

Block chains have been an emerging topic in today's world. When the conversation is about block chains, Smart contract is a topic that cannot be not talked about. Smart Contract can be considered as the heart of the block chain. Block chain is a very secure concept, but no matter how secure the block chain is, it can be hacked due to the security flaws in a smart contract. There are many incidents reporting the compromise of block chains due to the exploit in smart contracts. Therefore, there are many vulnerabilities found on smart contracts. Out of them there are few main issues that aren't addressed properly with a good solution.

This research proposes a web application that detects smart contract vulnerabilities. The main objective of the application is to detect smart contract vulnerabilities and fix them. When any solidity code is put into the application, it will detect the vulnerabilities and output a vulnerability free code to be downloaded by any user. This research is themed to develop an application which makes smart contracts safe and applicable to use in most practical solutions. To address this theme, four main problems of smart contracts have been identified. They are,

1) Reentrancy vulnerability
2) Integer overflow and underflow vulnerability
3) Delegate call vulnerability
4) True randomness

The identified problems in smart contracts have been addressed through the implementation of an application with this research. In addition, some minor issues were identified and rectified within the application, and a web application was developed to enhance the security and resilience of smart contracts.

II. BACKGROUND STUDY AND RELATED WORK

A) Generating True Randomness in Smart Contracts

A smart contract may contain a variety of vulnerabilities. The creation of randomization on a smart contract is one of their primary issues. Several smart contract applications, including lotteries, casinos, games of chance, and other decision-making processes, depend heavily on randomness. True randomness generation can be difficult in deterministic computer systems, such as smart contracts due to many reasons. This is mainly due to the transparency which is presented in block chains. In many real-world applications, several workarounds have been developed for generating pseudo random numbers to be used in smart contracts.

A provably fair and verifiable RNG that can be used for smart contracts is the Chain-link Verifiable Randomness Function (CVRF) [1]. Here Smart contracts can provide a random seed to CVRF, which then uses the private CVRF key to calculate a value and return it as a pseudorandom integer. It should be noted that CVRF does not attain full decentralization as it relies on the reliability of the Chain-link oracle. The RANDAO smart contract serves as a library that supplies random numbers to other contracts [2]. This method

ISSN (online): 2581-3048
Volume 7, Issue 10, pp 660-667, October-2023
https://doi.org/10.47001/IRJIET/2023.710085

is also applied in Quanta [3], a blockchain-based lottery platform that runs on Ethereum. Here, the RANDAO interacts with volunteer participants to create random numbers. But this approach also has some problems. First off, RANDAO's numbers are frequently employed in several client contracts. This is a well-known vulnerability and misconduct. RANDAO also faces the same issue as an oracle. As a result, this method is not appropriate for producing random numbers. The" Random Bit Generator," a distributed random number generator, was then released by a research team (RBG) [4]. Fully decentralized RNG capabilities for smart contracts were thus built. Yet there were also some issues with this strategy. This method did not properly demonstrate how to assess the genuine randomness of the generated values. The generated numbers may not have had a high randomness rate as a result. The slow throughput of its random number generator was the next issue it had. The last issue was that everyone who took part in the RNG got the same awarding values. There was no system in place to incentivize them to generate more distinctive numbers by compensating them.

Through these approaches, it becomes evident that achieving genuine randomness generation within a decentralized and trust less system presents significant challenges. It is also apparent that the current methods in use do not provide an adequate level of security. Therefore, the goal of this research project is to create a fresh technique for producing real randomness in smart contracts to overcome these difficulties and security issues. The suggested approach will make use of the distributed ledger and consensus processes that are built into the blockchain as well as other features to enable secure and dependable creation of real randomness. The efficiency of the suggested methodology will be examined empirically and contrasted with current methodologies.

B) Improved Solution for Integer Overflow and Underflow

The issue of integer overflow and underflow vulnerabilities in smart contracts has been examined in several research. A static analysis tool that finds integer overflow and underflow vulnerabilities in smart contracts was suggested by Alharby et al. in one paper [5]. The tool uses machine learning techniques to examine the contract's bytecode and find any potential flaws. The study demonstrated that technology can accurately identify vulnerabilities in real-world smart contracts. A runtime monitoring strategy was suggested in different research by Chen et al. (2019) to reduce integer overflow and underflow vulnerabilities in smart contracts [6]. The strategy entails keeping an eye on how the contract is being carried out and looking for unusual behavior that could point to the existence of a vulnerability. The study proved how well the method worked for identifying and preventing vulnerabilities in actual smart contracts. To find integer overflow and underflow vulnerabilities in smart contracts, Zhang et al. (2021) suggested a hybrid methodology that combines static and dynamic analysis methods [7]. The strategy uses static analysis to locate possible vulnerabilities and dynamic analysis to prove their existence and address them. The study demonstrated that the suggested method may accurately identify and address vulnerabilities in real-world smart contracts. For block chain-based systems to be secure and reliable, these vulnerabilities must be minimized. Although several studies have suggested useful mitigation strategies, further investigation is still required to provide more effective and scalable solutions.

C) Improved Solution for Reentrancy Based Vulnerabilities

Main vulnerability of this part is the reentrancy vulnerability. This flaw arises when a contract function is re-invoked before the completion of its initial execution. Malicious actors can exploit this gap to execute undesired reentrant calls, potentially altering the contract's intended behavior and leading to unauthorized fund transfers. Detecting and mitigating reentrancy vulnerabilities is crucial to ensuring the immutability and reliability of smart contracts [8]

The possibility of looping calls, which occurs when external calls are performed inside within loops, is another issue.Unwanted effects like stopped contract execution or unanticipated financial transfers may result from this. Additionally, modifications in error handling techniques have been brought about by the growth of the Solidity programming language, which is utilized for Ethereum smart contracts. To ensure security in contemporary smart contracts, it is crucial to identify out-of-date error handling statements. [9].

To solve these problems, researchers have created novel strategies. To find weak spots in code, one method is to utilize regular expressions like the "regular expression" function. This automatic detection approach aids in identifying and repairing weak code fragments [10].

Research from the past has examined the problems with smart contracts and suggested fixes. Some studies concentrate on locating re-entry vulnerabilities using code analysis. Others concentrate on using control flow graph analysis to find and mitigate looped call vulnerabilities. Additionally, efforts have been undertaken to strengthen error-handling systems and reduce risks through language enhancements and developer training. [11]

International Research Journal of Innovations in Engineering and Technology (IRJIET)

ISSN (online): 2581-3048

Volume 7, Issue 10, pp 660-667, October-2023

https://doi.org/10.47001/IRJIET/2023.710085

D) Delegate Call Function Vulnerability

The DelegateCall function in Solidity is a low-level interface that allows one contract to delegate the execution of a certain message call to another contract. Delegatecall function vulnerability occurs when delegatecall is used to execute code from an untrusted contract. Because the destination contract in a delegatecall writes to the calling contract's storage, contracts can run code contained within another contract as though calling an internal function. This allows an attacker to take control of the calling contract and do malicious activities such as draining currency held in the contract.[12] Actual examples of attacks that used the delegate call vulnerability were found during the ensuing study phase. Analysis of these attacks provided important information on the defenses used against these particular attack scenarios. The Parity Multi-Sig Wallet Attack is a good example. Each Parity multi-sig wallet in this situation had access to a shared wallet library, which made it possible to perform fundamental activities like withdrawing and depositing money. The centralized design of this library attracted the attention of the attackers. They changed it so that it worked as a multi-sig wallet, which gave them the power to take over and turn off other wallets. Unfortunately, this attack is the second-largest one that the Ethereum network has ever seen. It involved a substantial sum of 15,153,037 Ethers, and as a result, about 151 wallets were frozen.

III. METHODOLOGY

The approaches that were used to produce the intended findings have been discussed by the authors in this section. The diagram that follows, provides a general overview of the completed product.

A) Application

The methodology used throughout development combined server-side and client-side implementation as the two primary components. The server-side technique took advantage of Flask, a lightweight Python framework, and was modularized to target certain smart contract vulnerabilities, such as but not limited to reentrancy, delegate call, True randomness and Integer overflow/underflow smart contract vulnerabilities. Solidity files were submitted by users, and the system conducted a thorough analysis of those files. The Solidity code was then carefully fixed as vulnerabilities were found, and users immediately received feedback on the remedies that had been made. The user-friendly React-based interface was developed for vulnerability assessments. Users could easily analyze and submit Solidity files. Cross-Origin Resource Sharing provides seamless client-server communication via HTTP POST requests. Accurate testing ensured reliability, covering both client and server functions, and user-centric testing gathered vital feedback on usability and system efficacy.

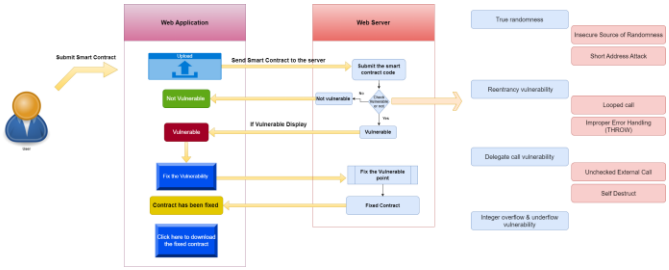

Figure 1: Overall Diagram

B) Generating True Randomness in Smart Contracts

The objective of this research component is to design and propose a secure, unmanipulable, efficient, and fast performing smart contract solution to generate a 256-bit random number. To accomplish this goal, a proposal for a random number generation game was put forward. This research component proposes the algorithm for this game and the algorithm will continue to work in the following manner.

First a person who needs to generate the random number in the Block chain should pay a deposit and call the contract for Random number generation. This contract is open to any participant in the block chain and for any person to join this contract to input his random value should initially pay a deposit. The people involved in the random number generation are called "Participants" and the person who needs the random number to be generated is the "Customer". The customer should initially determine the number of total participants participating (P) and a Number(X). Every participant should choose a number less than value of X and this will determine the number of bits each participant will be allocated in creating a 256-bit random number. Each participant will get an equal number of bits to create the final 256-bit random number. Participants are given a reward score, which determines the amount of their incentive. This is initially set at 0 points. The first person to register as a participant is designated as the" Core Player (p)" and all subsequent participants are designated as players. If Core Player can predict any value that is input by the other Participants, it will cause the Core Player's reward to increase while the Participant whose input was guessed will have a decrease in their reward. This is a tactic done to improve the randomness of number created. After the customer starts the game participants participate in the contract and first they create a random bit (x). This bit should be lower than value X (x<X). Next, they calculate the hash of it and send to the smart contract. Now the contract will ask for the value they inputted initially to see whether the right value was submitted. Here if the initial value matches the hash, then they will be given the

International Research Journal of Innovations in Engineering and Technology (IRJIET)

ISSN (online): 2581-3048

Volume 7, Issue 10, pp 660-667, October-2023

https://doi.org/10.47001/IRJIET/2023.710085

reward and the deposit back together. If values don't match, then those participants will be penalized from their deposit. The reward the participant gets will be divided from the deposit the customer pays. After collecting all the hash values from participants those will be concatenated to form a number. This number will be hashed again and converted into a binary sequence. This will be presented as a random number to the customer. When dealer and all the players pick a value at random, we compute the expected value of the award. The likelihood of a dealer and a player choosing the same value is 1/X. If the dealer makes the same decision as one of the players, the dealer earns X - 1 points, and if the dealer makes a different decision, the player receives 1 point. The expected value of the dealer's reward is computed as follows:

$$[(1/X) \times (X-1)] - [\{(X-1)/X\} \times 1] = 0$$

Similarly, the expected value of a player's payout is computed as follows:

$$[(1/X) \times (1-X)] + [\{(X-1)/X\} \times 1] = 0$$

These observations show that the predicted values of prizes for the dealer and the players are the same if the values are chosen at random. This game will work on this procedure and finally the Customer is able to generate a pure random number from the smart contract. This number can be used in the contract for random number needed purposes.

C) Improved Solution for Integer Overflow and Underflow

The research methodology applied in this study adheres to a systematic approach for addressing the challenges posed by integer overflow and underflow vulnerabilities. This approach is underscored by a dual focus: the formulation of innovative solutions for enhancing the SafeMath library and the refinement of detection techniques targeting these vulnerabilities. The initial phase of the research introduces an optimized solution that replace the existing SafeMath library. This requires a careful construction of code, implemented to tackle vulnerabilities of integer overflow and underflow in smart contracts. This code design not only protect security but also minimizes resource expenditure, addressing the cost concerns associated with gas fees.

In the context of Solidity programming, the SafeMath library serves as a tool to address integer overflow and underflow vulnerabilities. However, a notable downside of using the SafeMath library is the cost in terms of gas fees.

To overcome this, this research introduces an improved code implementation that consumes fewer gas fees compared to using the SafeMath library. In summary, this optimized library provides arithmetic operations that are designed to minimize gas costs while preventing overflow and underflow in Solidity contracts. However, while this approach can save gas compared to traditional SafeMath implementations, it relies on careful consideration of the specific conditions and behavior of the arithmetic operations.

```
library LowGasSafeMath {
    function add(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require((z = x + y) >= x);
    }
    function sub(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require((z = x - y) <= x);
    }
```

Figure 2: SafeMath Library

In the subsequent phase of the research methodology, a comprehensive detection approach is deployed to identify integer overflow and underflow vulnerabilities. Through a strategically developed set of attack strategies, the vulnerabilities are evaluated, exposing potential weaknesses. This assessment employs refined detection techniques to pinpoint arithmetic operations within Solidity code.

```
def detect_integer_overflow_underflow(file_path):
    with open(file_path, 'r') as file:
        solidity_code = file.read()

    arithmetic_patterns = [
        r'(\+|-|\*|/|%)\s*=\s*[^=]',  # Arithmetic operations
        r'(\+|-|\*|/|%)\s*[^=]',      # Arithmetic operations without assignment
    ]
```

Figure 3: Detect Integer Overflow Underflow

The implementation mainly includes two regular expression patterns which designed to identify different types of arithmetic operations within the Solidity source code. These patterns are implemented in the Python script to detect potential vulnerabilities related to integer overflow and underflow.

r'(\+|-|*|/|%) \s*=\s*[^=]'

Above pattern targets arithmetic operations where an operator (+, -, *, /, %) is followed by an equal sign. The\s*=\s* part captures the assignment symbol (=) surrounded by optional whitespace characters and [^=] ensures that the expression doesn't end with another equal sign.

r'(\+|-|*|/|%)\s*[^=]'

This pattern focuses on arithmetic operations that are not directly assigned to a variable. Similar to the first pattern, it starts by matching an operator (+, -, *, /, %). \s* accounts for optional whitespace, and [=] ensures that the expression doesn't end with an equal sign. Both patterns are used to systematically scan the Solidity code for instances of arithmetic operations, either with or without direct assignment. This comprehensive approach helps in identifying potential vulnerabilities arising from arithmetic calculations that could lead to integer overflow or underflow issues.

International Research Journal of Innovations in Engineering and Technology (IRJIET)

ISSN (online): 2581-3048

Volume 7, Issue 10, pp 660-667, October-2023

https://doi.org/10.47001/IRJIET/2023.710085

D) Improved Solution for Reentrancy Based Vulnerabilities

The research starts the process by combining Solidity contract source codes from many sources, combining resources from real-world examples. The foundation of a comprehensive data set is formed by this amalgamation, which is then carefully cleaned and standardized. These thorough procedures are designed to guarantee reliability and excellence. The methodology's main goal is to fix Reentrancy vulnerability. Here, it creates precise patterns that are intended to identify the code structures connected to these vulnerabilities. This requires a comprehensive examination of each code line using the pattern-recognition function name regular expression and it search for occurrences of the call.value() function calls. This pattern captures the content inside the parentheses of call.value(). Vulnerable occurrences are routinely found, recorded for later use, and their corresponding line numbers are methodically logged. Notably, the method goes beyond simple detection by using the regular expression function to automatically fix weak code by replacing the vulnerable call.value() calls with call.gas(2300).value(),

This method broadens its scope to include problems with looped calls. Customized patterns are used to find hidden calls hidden behind complex loops. Regular expression acquires prominence once more in search function call like transfer, send, or call these patterns. The issues are meticulously documented, and the result is the construction of an inventory that highlights problematic areas and their accompanying line numbers. It is crucial to stress that the process goes beyond simple detection and presents workable solutions. For controlling risky looped calls, it recommends the secure withdraw(msg.sender) technique as a safer option.

The methodology then moves into the area of out-of-date error handling techniques, specifically the deprecated throw statement, and makes extensive use of regular expression in combination with specialized patterns. A thorough cataloging of instances with outmoded procedures is done, along with the relevant line numbers. This rigorous information curation serves as a wake-up call for developers, urging them to update their methods.

In order to evaluate the methodology's practical impact, it is consistently applied across a wide range of Solidity contracts to confirm its efficacy. The resulting vulnerability reports go through a careful validation process to make sure they are accurate. The dependability of fixes carried out via the Regular expression technique is also thoroughly examined. As a concrete result, the methodology successfully reduces these vulnerabilities, increasing the overall resilience of contracts. As a result, a culture focusing on safe smart contract usage is fostered.

E) Delegate Call Function Vulnerability

The Vulnerability Detection Process is an advanced analytical endeavor that combines syntactic analysis, contextual analysis, and advanced pattern recognition in a continuous manner. Each element of this step works with the others to provide a thorough identification of delegate call vulnerabilities. A serious security risk exists if "msg.sender" or "msg.value" is present after the delegatecall. Delegatecall functions should not be used to avoid this issue. The process begins with the creation of a complex regular expression pattern (r'bdelegatecall'), carefully developed to go beyond simple text matching. This pattern searches the contract's source code for instances of "delegatecall" in function signatures, variable assignments, and conditional clauses. It is a manifestation of syntactic complexity and semantic intelligence. This thorough approach to pattern matching is designed to reduce false positives, improving the precision of the vulnerability detection method. After the Vulnerability detection, the web app recodes the vulnerable function line number for each smart contract. This rigorous record-keeping makes it easier to pinpoint where vulnerabilities are in the contract's source code.

The following describes how the plan to address these vulnerabilities is put into practice. The method suggests a specific strategy to strengthen the contracts' security when potential flaws involving delegate call functions are found. This solution entails replacing all instances of" delegatecall" in the contract's source code with" call." The method takes advantage of patterns that can be discovered using a specific strategy to implement this change. The modified contract code is then saved as a new file with the extension" fixed" appended. This makes it easier to discern between the vulnerable original version and the updated, more secure version. Developers may evaluate the changes and preserve the functionality of the contract thanks to this procedure, which guarantees that they have access to both the original and the enhanced versions of the code.

IV. RESULTS AND DISCUSSIONS

A) Generating True Randomness in Smart Contract

Based on the research conducted and the algorithm employed, there exists the capacity to enhance the efficiency, security, and unpredictability of the random numbers generated within a smart contract. This algorithm also has the ability to hinder all the challenges that occur when creating

International Research Journal of Innovations in Engineering and Technology (IRJIET)

ISSN (online): 2581-3048

Volume 7, Issue 10, pp 660-667, October-2023

https://doi.org/10.47001/IRJIET/2023.710085

randomness in smart contracts. The method proposed in this research beats all the available tools and methods when creating true randomness in smart contracts. More advancements can be made to this component by increasing the efficiency levels and unpredictability levels of random numbers created. The research findings revealed that the proposed random number generator completed one contract (256 bit) in an impressive 682.36 seconds, while the RBG [4] took substantially longer, requiring 352.18 seconds to complete a single contract(1 bit). These results clearly indicate that the proposed random number generator is not only faster but also significantly more efficient in terms of contract execution time.

Name	Time to Complete One Contract(Seconds)	No. Of bits Generated Per Contract	Throughput(bit per second)
Random Number Generator Proposed	682.36	256	1.3601
RBG [4]	352.18	1	0.0105

Figure 4: Throughput Evaluation

B) Improved Solution for Integer Overflow and Underflow

The outcomes of this research hold notable implications for the broader blockchain and smart contract ecosystem. The introduction of an optimized solution that replaces the existing SafeMath library not only enhances the security posture of smart contracts but also addresses a key bottleneck related to gas fees. By prioritizing resource efficiency without compromising on security, the mentioned approach aligns with the evolving needs of the blockchain industry. Furthermore, this research marks a significant step forward in addressing integer overflow and underflow vulnerabilities in smart contracts. The optimized solution for SafeMath library replacement and the refinement of detection techniques collectively contribute to a more secure and efficient smart contract environment. As the blockchain landscape continues to evolve and findings serve as a foundation for fostering greater confidence in the reliability of smart contract implementations.

Security tool	Original smart contract	With SafeMath library	With Improved SafeMath library
Mythil	Present	Absent	Absent
Oyente	Present	Absent	Absent
SmarkCheck	Not explicit	Not explicit	Not explicit
Securify	Not explicit	Not explicit	Not explicit

Figure 5: Security analysis of Improved SafeMath library using security tools

C) Improved Solution for Reentrancy Based Vulnerabilities

Through methodical detection and solution offering, the reentrancy vulnerability mitigation methodology demonstrated practical success. The method expertly located weak code structures and immediately presented better secure options. Additionally, the methodology successfully solved issues related to looped calls and outdated error-handling strategies.

The methodology's real-world impact on vulnerability reduction was highlighted via validation across a variety of Solidity contracts. Notably, 0.8586 accuracy in identifying and addressing reentrancy problems was accomplished. This result demonstrates the validity of this method and its potential to significantly increase code security. The outcomes demonstrate its viability as a reliable tool in the smart contract development environment.

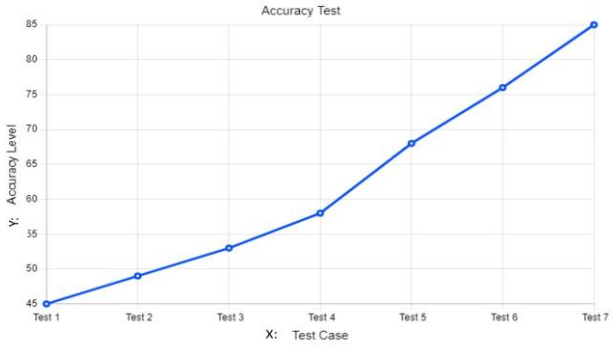

Figure 6: Evaluation of Accuracy

D) Delegate Call Function Vulnerability

The technique for detecting delegate call vulnerabilities, which combines syntactic and contextual analysis, produced encouraging results in terms of locating potential security issues related to delegate calls. The suggested approach demonstrated its effectiveness in raising security standards by replacing" delegate call" with" call" in contracts.

This research uses 194 real-world smart contracts [13] to test the delegate call function vulnerability detection and mitigation. Part, here are the accuracy levels comparing smart contract auditing tools.

Tools	Tested Contract	TP	TN	FP	FN	Accuracy
Slither	194	120	29	25	20	0.768
Smart Check	194	92	20	45	37	0.577
Mythril	194	114	23	28	29	0.706
Our Solution	194	125	92	18	19	0.809

Figure 7: Delegate function detection and mitigation Accuracy

$$\text{acurracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Figure 8: Accuracy Calculation

ISSN (online): 2581-3048
Volume 7, Issue 10, pp 660-667, October-2023
https://doi.org/10.47001/IRJIET/2023.710085

TP -True positive, TN- True negative, FP- False positive, FN- False negative

The generation of changed contract versions assisted developers in weighing changes and choosing the improved option, fostering the growth of a safety-focused smart contract culture.

V. CONCLUSION AND FUTURE WORK

This research set out to address this concern by introducing an innovative approach, a web-based automated detection tool for Solidity vulnerabilities. The development and implementation of the automated detection tool represent a significant step in blockchain security. The ability of this web application to analyze Solidity smart contract code, identifies vulnerabilities, and provide actionable insights could improve the overall security posture of blockchain applications. The results obtained from this web application show the effectiveness in detecting a wide array of vulnerabilities, including those that are extremely challenging to identify through manual review alone. As this tool becomes essential to the blockchain ecosystem, the potential future works from this study are given below.

- Advanced Vulnerability Identification: Explore the integration of more complex analysis techniques, such as machine learning and AI, to enhance the tool's capability to identify emerging vulnerabilities in Solidity smart contracts.
- Integration with Development Environments: Integrate the detection tool with popular development environments like Remix or Truffle, enabling developers to identify vulnerabilities in the code, promoting a proactive security approach.
- Support for Multiple Blockchains: Extend the tool's compatibility to different Blockchain platforms beyond Ethereum such as Biance Smart Chain, Polkadot, and others.

ACKNOWLEDGMENT

We value the advice and assistance that our supervisors and examiners have given us. Finally, we would like to convey our sincere appreciation to Sri Lanka Institute of Information Technology for helping to make our research a complete success.

REFERENCES

[1] Chainlink Developers, "Introduction to Chainlink VRF," Chainlink, 2022. [Online]. Available: https://docs.chain.link/docs/chainlink-vrf/.

[2] RANDAO: A DAO working as RNG of Ethereum," GitHub, 20-Feb-2019. [Online]. Available: https://github.com/randao/randao.

[3] Introducing Quanta blockchain lottery protocol," Medium, 16Aug-2016. [Online]. Available: https://medium.com/quanta/introducingquanta-blockchain-lottery-protocol-9b88a9c3ee5c.

[4] K. Chatterjee, A. K. Goharshady, and A. Pourdamghani, "Probabilistic Smart Contracts: Secure Randomness on the Blockchain," CoRR, vol. abs/1902.07986, 2019. [Online]. Available: https://arxiv.org/abs/1902.07986.

[5] R. A. M. A. A. A. M.,. S. Alharby, "A survey on integer overflow and underflow vulnerabilities in smart contracts," Journal of Ambient Intelligence and Humanized Computing, pp. 971-982., 2020.

[6] H. W. Y. L. X.,. H. Chen, "A runtime monitoring strategy to mitigate integer overflow and underflow vulnerabilities in smart contracts," pp. 477-481, 2019.

[7] Z. H. C. C. W. Z. M., L. X. Zhang, "A hybrid methodology for detecting integer overflow vulnerabilities in smart contracts," Journal of Computer Science and Technology, pp. 600-616.

[8] M. H. Zhai and D. Lo, "Reentrancy Attack and Mitigation in Ethereum Smart Contracts," in 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 2018, pp. 724-728.

[9] A.Juels and J. A. O. Garay, "The Tail at Scale," in 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2019, pp. 903-920.

A. J. Juels and R. S. S. Pradhan, "Grazing Attacks: A Hazards-toProperties Attack against Smart Contracts," in 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2020, pp. 853868.

[10] M. Weissbacher, C. Schubert, H. Schosser, and M. Affenzeller, "A Control Flow Graph-Based Heuristic for the Detection of Vulnerable Smart Contracts," in 2021 IEEE Congress on Evolutionary Computation (CEC), Krakow, Poland, 2021, pp. 2835-2842.

[11] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur and H. -N. Lee, "Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract," in IEEE Access, vol. 10, pp. 6605-6621, 2022, doi: 10.1109/ACCESS.2021.3140091

[12] Z. Liu, P. Qian, J. Yang, L. Liu, X. Xu, Q. He, and X. Zhang, "Rethinking Smart Contract Fuzzing: Fuzzing With Invocation Ordering and Important Branch Revisiting," arXiv preprint arXiv:2301.03943, 2023.

ISSN (online): 2581-3048
Volume 7, Issue 10, pp 660-667, October-2023
https://doi.org/10.47001/IRJIET/2023.710085

AUTHORS BIOGRAPHY

Kalana Jayasinghe,
Currently pursuing a B.Sc. in Cybersecurity from Sri Lanka Institute of Information Technology, Colombo. As an emerging cybersecurity enthusiast, their academic focus revolves around mastering the intricacies of network security, cryptography, ethical hacking, and risk management.

Ravindu Illeperuma,
Currently pursuing a B.Sc. in Cybersecurity from Sri Lanka Institute of Information Technology, Colombo. As an emerging cybersecurity enthusiast, their academic focus revolves around mastering the intricacies of network security, cryptography, ethical hacking, and risk management.

Induja Abeyrathne,
Currently pursuing a B.Sc. in Cybersecurity from Sri Lanka Institute of Information Technology, Colombo. As an emerging cybersecurity enthusiast, their academic focus revolves around mastering the intricacies of network security, cryptography, ethical hacking, and risk management.

Janindu Abeywickrama,
Currently pursuing a B.Sc. in Cybersecurity from Sri Lanka Institute of Information Technology, Colombo. As an emerging cybersecurity enthusiast, their academic focus revolves around mastering the intricacies of network security, cryptography, ethical hacking, and risk management.

Chethana Liyapathirana,
Academic by profession, having more than 2 years of experience in lecturing/training in universities and corporate training. She had already trained more than 1000 professionals in Information Security and IT management.

Citation of this Article:

Kalana Jayasinghe, Ravindu Illeperuma, Induja Abeyrathne, Janindu Abeywickrama, Chethana Liyanapathirana, "Strengthening Smart Contracts: An Investigation into Vulnerability Detection and Mitigation in Smart Contracts with a Web Application Based Tool" Published in *International Research Journal of Innovations in Engineering and Technology - IRJIET*, Volume 7, Issue 10, pp 660-667, October 2023. Article DOI https://doi.org/10.47001/IRJIET/2023.710085

****

