CEUR-WS.org/Vol-3567/paper2.pdf

Comparing smart contract vulnerability detection tools

Jarno Ottati[1], Giacomo Ibba[2] and Henrique Rocha[1]

[1]*Department of Computer Science, Loyola University Maryland, Baltimore, MD, USA.*
[2]*Department of Informatics and Mathematics, University of Cagliari, Cagliari, Italy.*

Abstract
Smart contracts are programs stored in the blockchain, and given their use cases mainly related to the management of digital assets and currency, it is crucial to ensure their security. Unfortunately, smart contracts may be vulnerable due to unsecured coding patterns that could be exploited by a malicious user. In this paper, we compared tools used to detect vulnerabilities within smart contract code. We use a curated dataset to analyze the efficacy of three tools: Oyente, Osiris, and Slither. Our results show that Slither is the best tool for detecting reentrancy (100%) and unchecked low-level calls (87%), but Oyente had better performance detecting Underflow (33%) and Osiris did better on Overflow (53%). These results indicate that developers may need to use different tools to find more vulnerable coding patterns and better secure their contracts.

Keywords
Reentrancy, Vulnerability, Coding Patterns, Smart Contract, Solidity, Code Smells

1. Introduction

Blockchain is an emerging technology that helped create digital services such as cryptocurrency [1] (e.g., Ethereum, Bitcoin) and IBM for food supply chain transactions [2]. This platform operates similarly to a ledger, registering new information on top of the previous logs without the ability to change registered information [3].

Among the main artifacts stored within the blockchain are smart contracts, programs simplifying the interaction between users and the chain without the help of third parties [1, 4]. Blockchain platforms like Ethereum can employ smart contracts written in Solidity [5] to exchange, transfer, and gather digital assets.

Smart contracts may deal with cryptocurrency, which makes them prime targets for exploits. Therefore, it is crucial to avoid building blockchain software relying on contracts with vulnerabilities. However, developing smart contracts without security leaks requires significant expertise in coding design patterns and cybersecurity. Code refactoring of vulnerable smart contracts could be pretty difficult too without proper tools. Even code smells can have more serious consequences in smart contracts. Therefore, it may be possible for developers to use insecure and vulnerable coding patterns in smart contracts due to a human error factor. One of the most critical

Benevol'23: The 22nd Belgium-Netherlands Software Evolution Workshop
jaottati@loyola.edu (J. Ottati); giacomo96ibba@outlook.it (G. Ibba); hsrocha@loyola.edu (H. Rocha)
0000-0003-3087-1969 (G. Ibba); 0000-0002-9154-0277 (H. Rocha)
© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

exposures is reentrancy, which gained infamous notoriety due to its role in the DAO attack. In 2016, an attacker exploited a reentrancy vulnerability to drain 3.6 million Ether from the DAO contract, with an approximate value of 50 million dollars [6, 7]. The DAO attack raised awareness among developers and Blockchain users about the risks of coding exposed smart contracts and the dire consequences in terms of money loss. Indeed, reentrancy is not the only vulnerability that could lead to an irreversible loss of assets. Denial of Service, for instance, consists of making a smart contract inoperable, freezing currency, and making it impossible for users to retrieve their money, and generally, to interact with the smart contract itself.

Researchers have developed several tools to find insecure patterns that lead to smart contract vulnerabilities, which are the basis for a proper code's corrective maintenance. Tools such as Oyente [1], Osiris [8], and Slither [9] can detect potential coding patterns for vulnerabilities within smart contracts. Each tool takes advantage of different methodologies to detect vulnerabilities and specific frameworks, making them different from each other.

In this paper, we compared tools used to detect vulnerabilities within smart contract code. We employed a curated dataset with multiple flawed contracts with reentrancy and other types of vulnerabilities to test the tools' efficacy in detecting the selected vulnerabilities. Our results showed that Slither performed best in detecting reentrancy, while Oyente and Osiris did better for Underflow and Overflow, respectively. These results indicate that developers may need to use different tools to better secure their contracts.

The remainder of the paper is organized as follows. Section 2 describes the vulnerabilities we selected in this study. Section 3 describes the dataset and presents the tools used in our research. Section 4 shows our results for comparing vulnerability detection tools. In Section 5, we present related work. Finally, Section 6 presents our conclusions and outlines future work possibilities.

2. Vulnerabilities

Since smart contracts are implemented by developers, they can be subject to human errors. Possible smart contract exposures derived from typical programming exposures, such as arithmetic overflows and underflows, and others arise from blockchain's inner properties, such as reentrancy, time dependency bugs, and unchecked low-level calls.

2.1. Reentrancy

The infamous DAO attack of 2016, raised more awareness among developers and blockchain users. An attacker managed to drain 3.6 million Ether (for a value of almost 50 million dollars) by exploiting a reentrancy vulnerability [6, 7].

When a smart contract calls a function from another contract, it transfers the control to the called contract. If the call happened in the middle of the function, then the original code will wait until the call is complete. A malicious user can take advantage of this scenario by re-entering the original function multiple times before the original call chain terminates. This procedure, for instance, would allow the attacker to re-enter a withdraw function, draining all the Ether from a smart contract. This gives reentrancy its name, by reentering the original code multiple times than allowed for illicit profit.

```
1 function withdraw (uint_amount) public {
2     require(balances[msg.sender] >= _amount);
3     (bool result ,) = msg.sender.call{value:_amount}("");
4     if(result) balances[msg.sender] -= _amount;
5 }
```

Listing 1: Reentrancy example in Smart Contract

Listing 1 shows an example of a Solidity function exposed to a reentrancy vulnerability. In this example, the function withdraw, allows a user to withdraw Ether from the smart contract's balance acting as a bank. Therefore, the contract will implement functions to deposit, withdraw, and transfer Ether. The function withdraw checks whether the user has enough funds in their balance (line 2). If it is true then the contract will transfer the required amount to the user's external address (line 3). Finally, if the Ether transfer is successful, then the user's balance is updated (line 4).

An attacker can call withdraw multiple times because the user's balance changes (line 4) are made after transferring the money (line 3). When transferring the money, the control is also transferred to the attacker which can recall the same withdraw function multiple times, before their balance is updated, letting them drain more money than the allowed amount.

The correct pattern to avoid reentrancy is called *checks-effects-interactions* [5]. First, we need to check the validity of the inputs, which Listing 1 example is doing correctly by verifying the balance before any withdrawal (line 2). Then we should apply the effects on the current contract and update the balance before interacting with an external entity. Finally, we could transfer the money which would make an external call (i.e., interacting with another). This simple reordering of the same code protects a contract against reentrancy.

2.2. Overflow and Underflow

Arithmetic Overflows and Underflows [8] occur when the result of a mathematical operation is too large or too small to be represented by the data type of the variable that is storing the result. Arithmetic Overflow occurs when the result of a mathematical operation is greater than the maximum value that can be stored in the variable. When this happens, the result "wraps around" to the minimum value of the variable. For example, if we add 1 to the maximum value of an unsigned 8-bit integer (255), the result will be 0. On the other hand, arithmetic Underflow occurs when the result of a mathematical operation is less than the minimum value that can be stored in the variable, then it "wraps around" to the maximum.

In Solidity [5], the developer can select how many bits (in increments of 8) their numeric variables will occupy in memory. Therefore, to avoid underflows and overflows, a developer needs to be sure their variables have enough space, or double-check their results after each operation. Moreover, some libraries like safemath wrap mathematical operations with safety checks against Underflow and Overflow.

2.3. Unchecked Low-level calls (ULLC)

Unchecked Low-level calls (ULLC) are also called "silent failed sends". Solidity offers complex low-level functions call(), callcode(), delegatecall(), and send(). Their behavior in error handling differs from other functions that transfer control. Instead of throwing an exception in case of failure, they return a boolean value set to false and leave the error handling to the developer. Not checking a returned error value could be considered a simple code smell in other software, but for smart contracts, it can have more serious consequences like loss of assets.

```
1 function withdraw(uint256 _amount) public {
2   require(balances[msg.sender] >= _amount); //check
3   balances[msg.sender] -= _amount; //effect
4   msg.sender.send(_amount); //interaction
5 }
```

Listing 2: Low-level call example in Smart Contract

Listing 2 shows an example of a ULLC. This example uses the pattern *checks-effects-interactions* but the ULLC introduces another issue. In this example, the returned value by the send() function remains unchecked (line 4). If for any reason the money transfers to msg.sender fails, the user will lose those Ether permanently as the withdraw function already updated his balance (line 3).

3. Study Design

We require a set of curated smart contracts to analyze the efficacy of the vulnerability detection tools. In this research, the experiments were conducted on contracts from the Smart Bugs [10] curated version. This dataset collects Solidity smart contracts flagged with different vulnerabilities, which is ideal for our research since it would allow us to spot the best tool to patch specific vulnerable code patterns.

Our main goal is to compare the efficacy of tools in detecting vulnerabilities in Solidity smart contracts. For this research, we selected the following tools: Oyente, Osiris, and Slither.

Oyente[1][1] is one of the first tools designed with the purpose to detect bugs within smart contracts. Oyente is a symbolic execution tool that works directly with Ethereum virtual machine (EVM) byte code. This methodology can provide a systematic and thorough exploration of a program's behavior, including edge cases that might be challenging to find using traditional testing techniques. However, the

[1]https://github.com/enzymefinance/oyente

main drawback of symbolic execution relies on its potentially high computational cost to execute programs with too many possible paths to cover. Moreover, programs including complex data structures and external interactions could be challenging to symbolic execute.

Osiris[2][8] was introduced in order to detect code patterns exposed to arithmetic bugs effectively in smart contracts. Osiris is based on Oyente, and it takes advantage of symbolic execution combined with the taint analysis method. Taint analysis tracks and analyzes the flow of data and helps answer the question of how data from untrusted or potentially malicious sources can propagate through a program and reach sensitive areas. This methodology relies on predefined taint policies that specify how data becomes tainted and what constitutes a security violation. However, it is a challenge to define these policies and determine what data should be considered tainted. This technique may not consider the broader context and may miss vulnerabilities that require a more holistic understanding of the application's behavior.

Slither[3][9] provides granular information about smart contract code and gives feedback for better code structure. Slither uses static analysis, which examines source code, program binaries, or other software artifacts without actually executing the code. Static analysis can be applied at various stages of the software development life cycle and is used to improve code quality, security, and reliability. However, static analysis lacks the context of dynamic behavior, and it may not capture issues that require runtime information or involve complex program interactions. Apart from critical exposures, Slither reports general bad programming practices, which makes it suitable for code refactoring and optimization.

Our experimental design and analysis would benefit from a combination of tools exploiting different approaches. For this very reason, we understand that the strict set of tools and techniques used in this analysis creates a threat to validity.

4. Results

Table 1 shows the achieved results in our research project. Each row displays a vulnerability we tested, and each column presents the tool used. The "Total" column shows the total amount of contracts we had in the dataset with that vulnerability flagged (i.e., the maximum possible contracts to detect). For each tool, we present how many contracts were detected with that vulnerability and the percentage according to the total contracts with that vulnerability.

Starting with our main vulnerability tested, we can see that Slither detected reentrancy in all contracts (100%) used for our test, outperforming the other tools. This may indicate that Slither is the best tool for reentrancy detection among the analyzed ones. Oyente and Osiris detected reentrancy on the same 23 contracts (which is 65%).

[2]https://github.com/christoftorres/Osiris
[3]https://github.com/crytic/slither

Table 1
Vulnerability Detection Tool Comparison.

Vulnerability	Oyente	Osiris	Slither	Total
Reentrancy	23 (65%)	23 (65%)	35 (100%)	35
Underflow	5 (33%)	3 (20%)	0 (0%)	15
Overflow	7 (46%)	8 (53%)	1 (6%)	15
ULLC	–	–	28 (87%)	32

After testing reentrancy, we expanded our set of vulnerabilities because they can be just as serious and cause damage to smart contract security. The next vulnerabilities we tested were Overflow and Underflow. In this test, Oyente performed best detecting 5 out of 15 contracts (33%) flagged with Underflow. Meanwhile, Osiris performed better in detecting Overflow and catching 8 out of 15 contracts (46%). Slither, our best-performing tool for Reentrancy, underperformed in this experiment, detecting 0% and 6% for Underflow and Overflow, respectively. It's worth noting that the best-performing tool results for Underflow and Overflow are a superset of other tools, i.e., the best tools detected all contracts detected by the others, and more.

In our last vulnerability test, Unchecked low-level calls (ULLC), were only detected by the tool Slither. Both Oyente and Osiris can't detect this type of vulnerability.

We also compared the average execution time (AET) of the three tools. Slither outclasses the other two taking just an average time of 0.54 seconds to scan the smart contract and report potential vulnerabilities. Osiris with an *AET* of 51.5 seconds is the slowest in terms of scanning and reporting vulnerabilities, while Oyente takes an average execution time of 18.2 seconds. However, the execution time is not the primary feature to evaluate these vulnerability detection tools, since it may be possible that despite being fast, a tool could produce a significant number of false positives and negatives.

Based on the results, we can see that within a specific vulnerability, the tools are not complementary. The best-performing tool detected all contracts the other tools did. This was unexpected, and we initially thought different tools would detect different contracts within the same vulnerability. This could indicate the methods for detecting vulnerabilities are similar, and consequently, the best tool will always achieve a superset of the others.

However, even though the tools do not have complementary results within the same vulnerability, the best tool differs for each vulnerability. Therefore, based on our results, the tools are complementary as a suite to detect different vulnerabilities together. Moreover, by using multiple tools we will have more confidence in the security of a smart contract against vulnerabilities.

5. Related Work

Luu et al. [1] investigate several security problems within smart contracts with the goal of gaining profit. Introducing problems like Transaction-Ordering dependence, Timestamp dependence, Mishandled exceptions, and Reentrancy. They also provide a tool called Oyente with the goal of detecting bugs and vulnerabilities, working with Ethereum virtual machine bytecode. The results provided by their tool on 19,366 contracts say that at least 8,333 contracts were susceptible to these bugs.

Torres et al. [8] introduce the tool Osiris with the goal of detecting integer bugs in EVM bytecode. Introducing potential bugs due to the behavior of integer operations within EVM and Solidity in specific scenarios. Leading to Arithmetic bugs, Truncation bugs, and Signedness bugs. They proposed their approach to detecting these bugs through Osiris. Using a dataset previously used by Zeus, another tool that detects integer bugs, to test its efficiency. Testing a total of 883 contracts where 711 are safe and 172 unsafe from arithmetic bugs, demonstrating its unique approach compared to Zeus which aims to be more complete.

Feist et al. [9] introduce their tool Slither an open-source static analysis framework. This tool provides important information about Ethereum smart contracts and displays critical properties like any static framework. Slither also uses its own intermediate representation, SlithIR, designed to make static analyses on Solidity code straightforward.

6. Conclusion

Vulnerabilities in smart contracts, like reentrancy, can jeopardize the security and reliability of blockchain software as demonstrated by the DAO attack. Therefore, the use of vulnerability detection tools to ensure smart contracts' security is paramount.

We used a curated dataset and ran experiments on three vulnerability detection tools: Slither, Oyente, and Osiris. Considering reentrancy, Slither detected 35 of 35 contracts (100%) outperforming Oyente and Osiris which detected 23 out of 35 contracts (65%).

Concerning Underflow, Oyente detected in 5 out of 15 contracts (33%), outperforming Osiris (20%) and Slither which detected none (0%). For Overflow, Osiris did best by detecting 8 out of 15 contracts (53%) compared to Oyente (46%) and Slither (6%).

For Unchecked Low-Level Calls, Slither detected 28 out of 32 contracts (87%), being the only tool capable of detecting this type of vulnerability.

The results highlight the importance of using a solid set of vulnerability detection tools to scan smart contracts. Indeed, our findings depict that the tools are not complementary with the same vulnerability, and they must used jointly to perform an accurate analysis. This finding aligns with best practices for securing smart contracts.

For future work, we plan to expand our experiments, add more tools (e.g., Securify, Manticore, Honeybadger, Mythril) to the analysis, and assess their efficacy. We also envision expanding the vulnerable code patterns lists and despite the several tools detecting different kinds of vulnerabilities, we aim to investigate whether is possible to detect common patterns between different tools. For instance, despite already knowing that Honeybadger does not detect reentrancy vulnerabilities, it detects money flows and balance disorders. Therefore, we expect that where Slither (for instance) detects a reentrancy vulnerability, Honeybadger spots a money flow and balance disorder. Additionally, we aim to confront the number of false positives and negatives produced by the different tools and the number of missed exposures. As a final contribution, we aim to detect the best configuration of tools to scan projects such as decentralized applications. Finding a configuration of tools covering a wide range of vulnerabilities would allow to widely improve code maintenance and refactoring.

References

[1] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: ACM SIGSAC Conference on Computer and Communications Security, CCS '16, ACM, NY, USA, 2016, p. 254269. doi:10.1145/2976749.2978309.

[2] J. Joo, Y. Han, An evidence of distributed trust in blockchain-based sustainable food supply chain, Sustainability 13 (2021). doi:10.3390/su131910980.

[3] J. Kolb, M. AbdelBaky, R. H. Katz, D. E. Culler, Core concepts, challenges, and future directions in blockchain: A centralized tutorial, ACM Comput. Surv. 53 (2020). URL: https://doi.org/10.1145/3366370. doi:10.1145/3366370.

[4] S. Bragagnolo, H. Rocha, M. Denker, S. Ducasse, SmartInspect: Solidity smart contract inspector, in: International Workshop on Blockchain Oriented Software Engineering (IWBOSE), 2018, pp. 9–18. doi:10.1109/IWBOSE.2018.8327566.

[5] Ethereum Foundation, Solidity documentation release 0.8.21, https://docs.soliditylang.org/_/downloads/en/v0.8.21/pdf/, 2023.

[6] S. Demeyer, H. Rocha, D. Verheijke, Refactoring solidity smart contracts to protect against reentrancy exploits, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering, Springer Nature Switzerland, Cham, 2022, pp. 324–344.

[7] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on ethereum smart contracts (SoK), in: M. Maffei, M. Ryan (Eds.), Principles of Security and Trust, Springer Berlin Heidelberg, Berlin, Heidelberg, 2017, pp. 164–186.

[8] C. F. Torres, J. Schütte, R. State, Osiris: Hunting for integer bugs in ethereum smart contracts, in: 34th Annual Computer Security Applications Conference, ACSAC '18, ACM, NY, USA, 2018, p. 664676. doi:10.1145/3274694.3274737.

[9] J. Feist, G. Greico, A. Groce, Slither: A static analysis framework for smart contracts, in: 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), IEEE Press, 2019, pp. 8–15. doi:10.1109/WETSEB.2019.00008.

[10] T. Durieux, J. F. Ferreira, R. Abreu, P. Cruz, Empirical review of automated analysis tools on 47,587 Ethereum smart contracts, in: 42nd International Conference on Software Engineering (ICSE), 2020, pp. 530–541.