Development of Bank-Logger Honeypot System Using Solidity Inheritance Patterns

NVS. Sindhura[1], L. Vakula[2], N. Girish[3], M. Rohith[4]
[1,2,3,4]*Department of Computer Science and Engineering Lendi Institute of Engineering and Technology Vizianagaram, India*

***Abstract*—When working with blockchain-based financial applications, it was noticed that smart contract security issues are becoming more common. Current analysis approaches, which are based on static code analysis, do not provide details about dynamic attacks post-deployment. In this paper, we present the design and development of a Bank-Logger Honeypot System using an inheritance approach in the Solidity smart contract language. It simulates a decentralized bank contract (with deposit and withdrawal services), and provides several layers of logging to capture the data about system interactions. The design strategy is to use three contracts: a Base Logger contract that defines event structures and logs transaction information; a Honeypot Bank contract that includes logging functionality and supports basic banking features; and a Honeypot Trap contract that inherits the banking contract and adds lures and traps to mimic malicious activity. The deployment of the system is done using Ganache so that we can test everything locally without risking real cryptocurrency. Patterns of function calls captured through Solidity events are used to understand data-flow patterns, code usage patterns and attack patterns. The experimental results show that the modular design using inheritance patterns is successful in providing separation of concerns and maintainability, and in producing a valuable dataset of interactions. This system can be used as a practical platform for learning and further research in smart contract security, and it also shows how object-oriented concepts can be applied in blockchain development.**

***Index Terms*—Solidity, smart contracts, honeypot systems, blockchain security, Ethereum, inheritance patterns.**

I. INTRODUCTION

Blockchain technology has changed how digital finance works, especially by removing the need for intermediaries in transactions. Among the many uses of blockchain, there is a growing interest in the use of smart contracts, which allow financial activities to be carried out automatically via a computer program without the involvement of a third party [1]. For example, if you wanted to create an automated bank-like service using smart contracts, developers can utilise an established platform like Ethereum to build and launch programmable smart contracts using programming languages such as Solidity on that platform, so that they can offer the same types of services as banks.

While working on smart contracts, we found that they are still vulnerable to issues like coding mistakes and unexpected behaviour. When they are deployed, you cannot change them; thus, it is extremely difficult to fix an error or prevent someone from exploiting it [2]. Attackers tend to target these vulnerabilities by changing how the contract works, which may cause the user to lose money or compromise the integrity of the system. The traditional approaches to security, such as static code analysis and vulnerability scanning, will identify problems before deployment; however, they cannot detect how an attacker interacts with the system or how that interaction behaves in a live environment.

As a response to limitations of existing methodologies, Honeypot systems used for monitoring malicious activity have been adapted for use outside traditional cybersecurity domains and within the context of Blockchain technology. Honeypots are purposely designed to appear vulnerable or inviting to would-be attackers and provide a means to monitor, log and analyse the malicious activities of an individual [3]. Within the Blockchain context, Honeypot Smart Contracts create an environment that creates exploitable situations or conditions that are intended to lure in would-be attackers for the purpose of surreptitiously logging and documenting their individual patterns or habits of interaction.

In our project, we designed a Bank Logger Honeypot System built using an Object-Oriented Programming Development Methodology of Inheritance in Solidity. This system simulates the operations of a Distributed Banking Contract and implements operational Banking Functions, such as Deposits and Withdrawals, while containing hidden monitoring capabilities of the Bank Logger. The Contract system is designed hierarchically through an Emulated Object-Oriented Programming Methodology of Inheritance and organized into modular components. The Base Logger smart contract captures transactional data, while the Parent Contracts implement the Banking Functions and the Deceptive Trap Mechanisms, respectively [4].

The Honeypot System has been deployed on a Local Blockchain Simulation Environment via Ganache, providing a low-risk & cost-effective platform for experimentation. The primary objective is to create a controlled environment for research on attacker behaviours in Smart Contract environments, and to demonstrate the ability to apply Inheritance in Solidity to create scalable, maintainable Architecture.

A. Research Motivation

Current research on blockchain security primarily focuses on static scanning of known system vulnerabilities and theoretical threat modelling methods of attack. During our research, we found that there are very few platforms where we can safely study attacker behaviour in a real environment and collect empirical data from would-be attackers in a secure manner. The proposed Honeypot System serves to provide this unique combination of traditional Honeypot Applications with the Inheritance Functionality of Smart Contracts developed under Solidity in a Controlled Local Blockchain Environment; thereby creating an Environment that is Accessible for Educators, as well as contributing to Empirical Research in the field of Blockchain [5].

II. RELATED WORK

Blockchain gained substantial interest due to the common usage of the decentralised applications. This section presents previous research on smart contract vulnerabilities and methods for their analysis and honeypot systems.

A. Smart Contract Vulnerabilities

Smart contracts are self-executing programs written in languages like Solidity. While they offer automation and transparency, smart contracts also have many logical and design vulnerabilities, such as reentrancy attacks, integer overflows, and improper access control [6]. A well-known example is the DAO hack in 2016, where a vulnerability allowed attackers to drain funds from the DAO through an automated process - this was an important reminder that companies need to implement good security practices [7]. The subsequent development of secure coding patterns and techniques for smart contracts cannot be discounted; however, most of these methods are dependent primarily on static code analysis versus observing operational use in real-time.

B. Static and Dynamic Analysis Tools

Several tools have been developed to increase the security of smart contracts. Tools like Mythril and Slither are used to analyse smart contract code before deployment and help identify possible vulnerabilities [8]. While they provide value in early detection of contract vulnerabilities, static analysis cannot capture the runtime behaviour of a contract (the execution of a smart contract in its operating environment) or the interaction patterns of users with the smart contract and, as a result, cannot protect against all possible types of vulnerabilities, including false positives and overlooking logical flaws that can be difficult to detect. Dynamic analysis tools (e.g. Remix IDE) enable users to interactively test and debug smart contracts; however, these tools are designed for development and do not provide structured logging of user interaction data to allow for future analysis [9].

C. Honeypots in Cybersecurity

Honeypots are utilized extensively in standard cybersecurity for monitoring attacker behaviour [10]. When attackers access this honeypot system, it allows an organization to gather information about their actions. The Honeynet Project is one of many organizations doing honeypot research to help collect statistics for unauthorized access attempts, malware identification, and intrusion techniques. However, honeypots designed for use in traditional centralized systems do not translate effectively to decentralized blockchain systems.

D. Blockchain-Specific Honeypots

Recent studies have begun looking at the use of honeypot smart contracts deployed on a public blockchain platform [11]. Torres et al. [12] recently conducted a study measuring actual honeypots deployed within the entire Ethereum blockchain and found thousands. The authors identified different types of honeypots designed to imitate vulnerabilities in order to attract attackers into accessing such honeypots by using their resources without actually exploiting them. Existing blockchain honeypot solutions developed till now have several limitations, including the financial risk and ethical issues linked with deploying them on active networks; the primary focus on deception rather than organized data collection; and a lack of prominence on modularity and maintainability [13].

E. Research Gap

Based on the literature reviewed, the following gaps exists: (i) there is not sufficient attention placed on analyzing how users interact with smart contracts through behavioral analysis in a controlled scenario; (ii) there is limited use of modular-based inheritance design patterns in honeypot development; (iii) there is no widely available platform that researchers can use to experiment on a honeypot without incurring financial risk; and (iv) there is no standard method for organizing logs for researchers to be able to do meaningful analysis on their data. The proposed system will address each of these gaps by using Solidity inheritance, event-driven logging, and by deploying the honeypot contract on a local blockchain [14].

III. METHODOLOGY AND SYSTEM DESIGN

A. System Architecture

Ganache was used to simulate a blockchain environment because it allowed us to test transactions quickly. There are three architecture levels: Level 1 is the interaction level, where either a person will log into the system or a simulated attacker will access/create some illegitimate action, Level 2 is the contract execution layer which consists of three Solidity smart contracts we have designed to use in this testing (BaseLogger, HoneypotBank and HoneypotTrap), and Level 3 is the data storage level where the immutable "blockchain" ledger will maintain all event logs after the event happens.

Interaction Layer Contract Execution Layer Data Storage Layer User / Attacker HoneypotTrap Contract HoneypotBank Contract BaseLogger Contract Blockchain Ledger (Immutable Event Logs) extends inherits base

Fig. 1. System architecture of the Bank-Logger Honeypot System showing the three-layer contract hierarchy.

B. Contract Inheritance Hierarchy

We created three smart contracts: BaseLogger, HoneypotBank, and HoneypotTrap as a part of a hierarchical inheritance model. The base logger is the parent contract and serves as the root contract. The base logger defines the event data structure as well as the reusable logging function. Notably, the honeypot bank uses the logging function introduced by the base logger contract when it processes the deposit and withdrawal methods, thereby adding a layer of logging automatically. The honeypot trap contract inherits the functionality of the honeypot bank but adds a layer of validation before completing a deposit transaction. Specifically, the honeypot trap verifies the event is of a suspicious pattern, and only if it satisfies this condition, does the transaction complete, and the log is recorded. The lineage of these contracts is consistent with the Liskov Substitution principle, meaning that any derived contract should be able to substitute its parent contract in all ways that matter [15].

«contract» BaseLogger + logEvent(addr, action, val) event LogTransaction «contract» HoneypotBank mapping balances + deposit() payable + withdraw(amount) «contract» HoneypotTrap + deposit() override + trapCheck() = inherits

Fig. 2. Class diagram showing the Solidity inheritance hierarchy of the three smart contracts.

C. Algorithmic Design

Whenever a user performs a transaction, the system processes it using event-based logic. When a transaction is received, the contract checks whether the value exceeds the defined trap threshold. If it does, a trap event is triggered. After this check, the transaction is processed normally by the banking contract, and the account balance is updated. The base logger records [emits] all transactions by the transaction input parameters (sender details, name of

the function that triggered the event and transaction amount). All events emitted will be logged and will remain permanently as part of the blockchain ledger.
The formal data-capture function for any interaction event E is defined as:

E = (addr, fn, v, t) (1)
where addr represents the caller's Ethereum address, fn represents the function being called, v represents the transaction value in Wei, and t represents the block timestamp. A trap event is triggered according to:
$T = 1$ if $v > v_{threshold}$, else 0 (2)

D. Smart Contract Implementation
The three Smart Contracts are all deployed using Solidity v0.8.0, and below is a condensed listing of each contract.

```
// BaseLogger.sol pragma solidity ^0.8.0; contract BaseLogger { event LogTransaction( address indexed user, string action, uint value); function logEvent(address _u, string memory _a, uint _v) internal { emit LogTransaction(_u, _a, _v); } }
// HoneypotBank.sol contract HoneypotBank is BaseLogger { mapping(address => uint) public balances; function deposit() public payable { balances[msg.sender] += msg.value; logEvent(msg.sender,"Deposit",msg.value); } function withdraw(uint _amount) public { require(balances[msg.sender] >= _amount, "Insufficient balance"); balances[msg.sender] -= _amount; payable(msg.sender).transfer(_amount); logEvent(msg.sender,"Withdraw",_amount); } }
// HoneypotTrap.sol contract HoneypotTrap is HoneypotBank { event TrapTriggered( address indexed user, string reason); function trapCheck(uint _val) internal { if (_val > 10 ether) emit TrapTriggered(msg.sender, "High Value Tx Detected"); } function deposit() public payable override { trapCheck(msg.value); super.deposit(); } }
```

E. Deployment Environment
All contracts were compiled using the Solidity compiler (solc v0.8.0) and deployed on the Ganache local blockchain instance. Ganache allows for 10 test accounts that are pre-funded and have a configurable block mining interval to allow for deterministic and repeatable testing. The deployment script was written with Hardhat, allowing for programmatic contract deployments and automated test runs. There is no need for public network connectivity during execution.

User Honeypot Trap Honeypot Bank Base Logger Blockchain Ledger tx request forward log call emit event
Fig. 3. Data flow diagram showing transaction processing and event logging across contract modules.

IV. RESULTS AND DISCUSSION

A. Experimental Setup
The experiments took place on a Ubuntu 22.04 LTS workstation with an Intel Core i5 (3.4GHz) and 8GB of RAM. The smart contracts were compiled using solc v0.8.19, and the environment used for deployment was Ganache v7.9.1. A total of 150 simulated transactions, made up of 60 deposits, 50 withdrawals and 40 interactions that triggered the honeypot traps, were made. At the beginning of the experiment, each test account was assigned 100 ETH.
B. Functional Validation
After functional validation of all modules before integration testing, as a summary of results, the BaseLogger emitted a LogTransaction event as expected for each invocation of logEvent(). The HoneypotBank successfully maintained per-account accounts through deposit and withdrawal operations, by rejecting underfunded attempts on withdrawals. During testing, we observed that the HoneypotTrap correctly flagged all high-value deposits that exceeded 10 ETH.

TABLE I. FUNCTIONAL TEST CASE SUMMARY

Test Case	Operations	Pass	Fail	Rate
Deposit (normal)	60	60	0	100%
Withdrawal (valid)	35	35	0	100%
Withdrawal (invalid)	15	15	0	100%
Trap Trigger (>10 ETH)	40	40	0	100%
Event Logging	150	150	0	100%

C. Interaction Pattern Analysis
From the logged event data, we observed that the 150 events show a clear distribution across three types of interaction. Deposit transactions made up 40% of

overall transactions, Withdrawals accounted for 33%, and Trap-Touch Transactions made up 27%. Further analysis of these Trap-Touch Transactions indicated that 85% were for values above 10 ETH, suggesting that simulated attackers tried to initiate a high-risk transaction/interaction as opposed to low-risk transactions. The diagram presented in Figure 4 presents the interaction type distribution as documented over the course of the experimentation.

0 20 40 60 80 60 35 15 40 Deposit Withdraw (Valid) Withdraw (Invalid) Trap Events No. of Transactions Transaction type distribution (n=150)

Fig. 4. Distribution of interaction types recorded across 150 simulated transactions.

D. Gas Consumption Analysis

Although no actual gas fees will be represented due to the local environment, gas consumption records will indicate computational overhead. Table II lists the measured amount of gas required for each of the four operations. As indicated previously, the amount of gas consumed by the BaseLogger was approximately 12% of the total gas consumption of the deposit operation. It can therefore be considered an acceptable logging solution for a monitoring-based application.

TABLE II. GAS CONSUMPTION PER OPERATION

Operation	Gas Used	Logging Overhead
Contract Deployment	623,401	—
Deposit (normal)	47,820	~5,640 (12%)
Withdrawal (valid)	52,310	~5,640 (11%)
Trap + Deposit	64,270	~17,090 (27%)
Trap (only, flagged)	28,930	~11,450 (40%)

E. Behavioural Insights

The analysis of event logs from simulations allowed for several behavioural observations to be made. First, simulated attackers sometimes have multiple attempts at a high-value deposit (the term "reentrancy attack" implies that the attacker is probing to see how responsive the contract will be on multiple occasions). Secondly, as more people made deposits into the contract, the number of monitoring traps (as represented by the increase in trap events from one simulation round to the next) was also increasing. Thirdly, time-wise, the attempts to withdraw (which were unsuccessful) clustered around high-value deposits and probably indicate that the attacker rolled the deposit in order to create an inflated account balance before trying to withdraw it. These observations show that the honeypot is useful for understanding attacker behaviour.

F. Comparison with Existing Approaches

The comparison of the system proposed in this paper to many existing methods or systems is listed in Table III.

TABLE III. COMPARISON WITH EXISTING APPROACHES

Feature	Mythril	Public HP	Proposed
Runtime Monitoring	No	Partial	Yes
Financial Risk	None	High	None
Modular Design	N/A	No	Yes
Structured Logging	No	No	Yes
Academic Suitability	Yes	No	Yes
Behavioral Analysis	No	Limited	Yes

V. CONCLUSION AND FUTURE WORK

In this project, we developed a Bank-Logger Honeypot System using Ethereum based on the Solidity inheritance concept. The system helps solve some key limitations of the current security solutions with real-time monitoring, logging and free local deployment. The inheritance of BaseLogger, HoneypotBank and HoneypotTrap shows that we can apply the object-oriented design to the blockchain development for reusable, modular and maintainable smart contract designs.

We tested the system with 150 transactions and observed that it handled all attack scenarios correctly and captured important interaction patterns. Cost analysis showed that the logging is cheap, less than

12% on average. We also compared our system with existing approaches, static analysis and honeypots deployed in the public network, and it demonstrated that the proposed system was better for educational and research purposes.

The future work will include. First, applying machine learning techniques to the event logs data set for profiling and detection of attacks. Second, the trap list needs to include more attack patterns (e.g. simulated reentrancy and int-overflow traps). Third, build a web dashboard for visualising the event logs in near real time, for more interactive analysis of the event logs and interaction patterns. Finally, deployment of the system on a private Ethereum test network (e.g. Sepolia, Hardhat Network) to more closely simulate network interactions and avoid the public Ethereum mainnet.

ACKNOWLEDGMENT

We sincerely thank the Department of Computer Science and Engineering, Lendi Institute of Engineering and Technology, for providing the facilities and support required for this project. We also appreciate the reviewers for their helpful suggestions in improving this paper.

REFERENCES

[1] V. Buterin, "A next-generation smart contract and decentralized application platform," Ethereum White Paper, 2014.

[2] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in Proc. POST 2017, LNCS, vol. 10204, pp. 164–186, 2017.

[3] L. Spitzner, Honeypots: Tracking Hackers. Addison-Wesley, 2002.

[4] C. Dannen, Introducing Ethereum and Solidity. Apress, 2017.

[5] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of Ethereum smart contracts," in Proc. 1st Int. Workshop Emerging Trends Software Eng. Blockchain, 2018, pp. 9–16.

[6] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in Proc. ACM CCS, 2016, pp. 254–269.

[7] P. Daian, "Analysis of the DAO exploit," Hacking, Distributed, 2016. [Online]. Available: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[8] B. Mueller, "Smashing Ethereum smart contracts for fun and real profit," in Proc. 9th HITB Security Conf., Amsterdam, 2018.

[9] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in Proc. IEEE/ACM 2nd Int. Workshop Emerging Trends Software Eng. Blockchain, 2019, pp. 8–15.

[10] C. Provos and T. Holz, Virtual Honeypots: From Botnet Tracking to Intrusion Detection. Addison-Wesley, 2007.

[11] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in Proc. NDSS, 2019.

[12] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in Ethereum smart contracts," in Proc. 28th USENIX Security Symp., 2019, pp. 1591–1607.

[13] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," Future Gener. Comput. Syst., vol. 107, pp. 841–853, Jun. 2020.

[14] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on Ethereum smart contract vulnerabilities: A survey," arXiv, 2019. [Online]. Available: https://arxiv.org/abs/1908.08605

[15] B. Meyer, Object-Oriented Software Construction, 2nd ed. Prentice Hall, 1997.