Salmonella Attack in Blockchain

Arnau Camarero and Àlex Montoya

June 19, 2023

Abstract

This research paper explores the Salmonella Attack, a type of security threat in Blockchain systems. The attack involves introducing malicious nodes that compromise the integrity of the Blockchain, potentially disrupting the consensus mechanism. The paper emphasizes the importance of implementing robust security measures, such as node authentication and monitoring for unusual behavior, to prevent Salmonella Attacks. It also aims to analyze contracts with Salmonella, detect them, and avoid buying from infected Blockchains. The study provides insights into the attack's prevalence, detection methods, and potential countermeasures.

Keywords: ethereum; DeFi; Dex; sandwich; salmonella; scam; security; slippage; mempool;

1 Introduction

Blockchain technologies have increased in the past decade and have evolved in good and bad ways. This research studies one of many attacks, named Salmonella, made by malicious users that gain money against other attackers and how it can be prevented.

The name "Salmonella Attack" comes from the way that the attack mimics the behavior of the salmonella bacteria in the human body. Just as the bacteria can spread from one part of the body to another and cause illness, the malicious nodes in a Salmonella Attack can spread through the Blockchain network and cause damage.

Salmonella is a type of security threat that can affect Blockchain systems. It is commonly used to attack those users that use the Sandwich attack and prevent those attackers from gaining money.

In a Salmonella Attack, an attacker introduces malicious nodes into the Blockchain network that behave in a way that is designed to compromise the integrity of the Blockchain. These malicious nodes can spread to other nodes in the network and potentially disrupt the consensus mechanism that is used to validate and record transactions on the Blockchain.

Preventing Salmonella Attacks requires implementing robust security measures, such as node authentication, monitoring for unusual behavior, and using consensus mechanisms that are resilient to attacks. It is important for developers and users of Blockchain systems to be aware of the potential risks posed by Salmonella Attacks and to take appropriate measures to mitigate these risks.

Our Contribution: Salmonella is still infecting Blockchain systems and for some users it is difficult to detect it by themselves. This is causing users to lose part of their tokens when buying in that infected Blockchain and infecting all the other users that use that same Blockchain.

This research is aiming to analyze different contracts with salmonella in them, learn how to detect them and avoid buying from that infected Blockchain.

We will analyze different types of contracts where Salmonella is applied in order to simulate our own contract with salmonella and try to defend or take advantage of its debilities to punish it for using this attack.

Organization of the paper : In Section 2, we describe the related work of vulnerabilities and malicious users and how they can affect the public Blockchains. Section 3 gives an explanation of preliminary content that you must know to understand about the paper. In Section 4 introduces the Salmonella attack and analyzes different contracts where is applied, our implementation of what salmonella contract could be like and a result of each implementation. In Section 5, we explain our conclusions of how avoid these attacks and others ones made. Finally, in Section 6, we explain our conclusion about the whole research made and what are the takeaways that we consider more important.

2 Related Work

Some studies show that the Tokens used by malicious users in public Blockchains such as Ethereum & Binance Smart Chain (BNB) to penetrate these attacks are short-lived. More specifically, the majority of them are active for less than 24 hours. Even though the cryptocurrency market is rapidly evolving, the investors are increasingly inclined towards decentralized exchanges (DEX), for its smart contracts is always reachable and working on the Blockchain.[17] Nonetheless, other studies find that the fraud in these decentralized finance (DeFi) ecosystems are increasing. Surprisingly enough, the laundering techniques used by the attackers are often unsophisticated, as well as their rug pulls. Nevertheless, through open-source investigative tools, studies show that it is possible to extract transaction-based evidence, such as the one we are aiming to find in our chosen contracts.[18]

Regarding vulnerabilities in smart contracts, they are frequent and can provoke serious amounts of lost money. That is why some studies have focused on detecting breaches in security services through security auditing and, namely, manual code inspection. It has been proven that this type of security auditing

is more efficient than static analysis tools. Yet, it is necessary for project developers to address and secure their smart contracts as required in order to reduce cryptocurrency crimes. [19]

3 Preliminar Content

3.1 What is a Blockchain

A Blockchain is a decentralized and distributed digital ledger that records transactions across multiple computers or nodes. It is designed to be transparent, secure, and immutable. In simple terms, a Blockchain is a chain of blocks, where each block contains a list of transactions. The principal key characteristics and components of a Blockchain are:

- Decentralization: Operates on a decentralized network of computers or nodes. This means that there is no central authority or single point of control, and the responsibility for maintaining and validating the Blockchain is distributed among multiple participants.
- Transparency: The data stored on a Blockchain is transparent and visible to all participants in the network. Each transaction is recorded in a block, and once added to the Blockchain, it becomes permanent and can be accessed by anyone. This transparency helps to establish trust and accountability. - Security: Uses cryptographic techniques to ensure the security of transactions and data. Each transaction is verified and linked to the previous block in a chain using a cryptographic hash function, creating a tamper-resistant structure. Additionally, consensus mechanisms, such as proof of work or proof of stake, are used to validate transactions and maintain the integrity of the Blockchain.
- Immutability: Once a transaction is added to the Blockchain, it is nearly impossible to alter or delete it. The decentralized nature of the Blockchain, coupled with cryptographic hashing, makes it extremely difficult for malicious actors to tamper with the data. This immutability provides a high level of trust and reliability.
- Smart Contracts: Many Blockchains, such as Ethereum, support the execution of smart contracts. These are self-executing contracts with predefined rules and conditions encoded within the Blockchain. Smart contracts enable automation of transactions and the creation of decentralized applications (DApps) on the Blockchain.

3.2 MEV

MEV (Miner Extractable Value) refers to the amount of value that miners can extract through strategic manipulation of transactions in a decentralized environment. This can be exploited through user errors, inefficient consensus protocols, and inadequate designs of decentralized applications (dApps).

An example of MEV in real life is when a miner sees a large buy transaction that has been sent to the Blockchain network. Instead of immediately processing this transaction, the miner can add their own transaction with a higher price to sell the same assets at a higher price before the buy transaction is executed.

Examples of MEV strategies include:

- Front-running: Capitalizing on advanced knowledge of pending transactions to execute trades that benefit the miner. For example, let's say transaction A is broadcasted with a higher gas price than an already pending transaction B so that A gets mined before B.
- Back-running: Exploit transaction delays to profit from trade or price discrepancies in decentralized systems.
- Flashbots: Leveraging priority access to the mempool to selectively include or exclude transactions.
- Sandwich: Exploiting the predictable behavior of certain smart contracts to extract value. Is a combination of a front-running plus a back-running.

These MEV strategies highlight the potential for miners to manipulate transactions and profit from their privileged position within the Blockchain ecosystem.

3.3 DeFi

Decentralized Finance (DeFi), is a financial system built on Blockchain technology that provides open and permissionless access to various financial services. It operates without intermediaries and relies on smart contracts on Blockchain platforms to create decentralized applications (DApps).

DeFi encompasses activities like lending and borrowing, decentralized exchanges, stablecoins, yield farming, asset management, insurance, and more. These services are transparent, programmable, and often built on public Blockchains such as Ethereum.

Key features of DeFi include composability, allowing different protocols to interact and create innovative financial products, and increased accessibility and inclusivity compared to traditional financial systems. However, DeFi also carries risks like smart contract vulnerabilities, price volatility, and regulatory challenges.

Overall, DeFi has gained popularity due to its potential benefits, but users must be aware of the risks involved and exercise caution when participating in DeFi activities.

3.4 Uniswap

Uniswap is the largest decentralized exchange (or DEX) operating on the Ethereum Blockchain. It allows users anywhere in the world to trade ERC-20 tokens directly from their Ethereum wallets without the need for intermediaries or centralized exchanges. It is often referred to as an Automated Market Maker (AMM) because it uses smart contracts to facilitate token swaps based on predetermined algorithm without an intermediary.

Uniswap operates on a constant product market-making formula, known as the $x^*y = k$ formula. This formula ensures that the product of the number of tokens in each pool remains constant, allowing for the automatic adjustment of prices based on supply and demand. When a trade is made, the smart contract calculates the new exchange rate and adjusts the token balances accordingly.

On Uniswap, arbitrage traders play an important role in the Uniswap ecosystem. These traders look for tokens that are trading above or below their average market price. This can happen when large trades create imbalances in the liquidity pool, causing the token's price to increase or decrease. The arbitrage traders take advantage of these price discrepancies by buying or selling the tokens accordingly. They continue this process until the token's price on Uniswap aligns with the price on other exchanges, eliminating the opportunity for further profit.

An example of this is whether let's say Bitcoin is being traded on Platform 1 for $\$27,500$ and on Platform 2 for $\$28,000$. An arbitrage trader could buy Bitcoin on Platform 1 at the lower price and sell it on Platform 2 at the higher price, making an easy profit. By executing such trades with large volumes, they can potentially earn significant profits with relatively low risk. This cooperative relationship between the automated market maker system of Uniswap and arbitrage traders helps to keep the prices of Uniswap tokens in line with the broader market. It ensures that any differences in token prices are swiftly corrected, contributing to the overall efficiency and stability of the Uniswap platform.

3.5 What is a Sandwich Attack?

A Sandwich Attack is a form of front-running, which primarily targets decentralized finance protocols and services.

As you can see in Figure 1, this attack consists in "sandwiching" a user's transactions, i.e., when a known large transaction is detected in the mempool, the attacker tries to buy a certain amount of the same token before the user's transaction gets executed. Whether the attacker is able to convince miners to execute his transaction first, he will increase the current price of the token before the user transaction is executed, so the user pays a higher price (front-running) and increases the price even more. Then the attacker, immediately after the user's transaction is executed, sells the same quantity that was bought to gain profits (back-running).

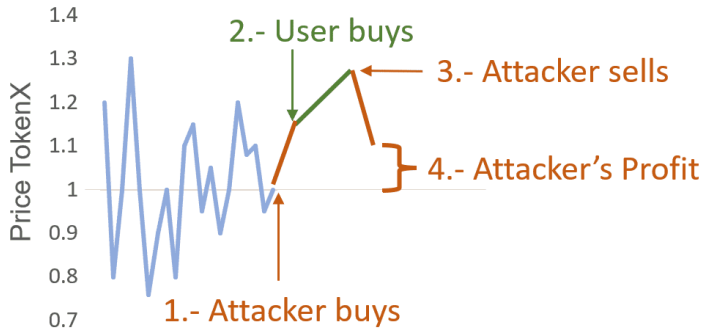

Figure 1: Plot showing how sandwich attack is implemented and it is affecting the price of the TokenX in order to make profit. [10]

This attack takes advantage of prior knowledge of transactions to gain economic advantage. The attacker seeks to manipulate the price of a token for their own benefit, at the expense of the user with the ongoing transaction. It is an unfair practice that can generate gains for the attacker while causing losses for the user.

Fair enough, but can we be confident that this attack will consistently yield results? No, the effectiveness of this attack relies on the occurrence of two specific conditions at once.

Since profits tend to be small, we must ensure a greater profit than the fee charged by the miners for processing each transaction (gas fee). This implies that the amount of tokens that the user acquires should be big in order to overcome gas fees.

On the other hand, transactions are not executed immediately, other transactions within the same token may alter the price if they are executed first. This means that we cannot know the exact price where our transaction will be executed, so we have to take into account the maximum deviation from the desired price which we are willing to accept in a trade (slippage).

In reality, the situation is a bit more intricate, token prices are governed by the AMM (Automatic Market Maker) equation, which follows the format of $x^*y = k$, where x and y represent the total quantity of each token being traded. Considering this equation and performing some calculations, we arrive at the formula that outlines the anticipated profit for the sandwich attack.

$$EP = UT - x * \frac{R}{1-R} \quad R = \frac{x}{x+AT} - \frac{x}{x+AT+UT}$$

$$EP = \text{Expected profit from the attacker perspective}$$
$$x = \text{Token quantity available in the liquidity poor pair}$$
$$UT = \text{Token quantity bought by the user}$$
$$AT = \text{Token quantity bought by the attacker}$$

Take note that the profit margin can be enhanced by increasing the values of UT and AT, the higher these quantities, the more substantial the potential profit. On the flip side, the attack increases its profit by keeping x as low as possible. This means that UT and AT should ideally represent the highest proportion possible of the buying token (x) to maximize effectiveness.

Throughout the timeframe between May 2020 and April 2022, a grand total of 457,691 sandwich attacks were observed, presenting a comprehensive overview of the attack activity. The distribution of the number of attacks and profit over time is as follows:

3.5.1 Sandwich attacks and profits

We observed an important downward trend in the frequency of sandwich attacks Figure 2. However, the return on investment (ROI) percentage, which represents the average profit measured in percentage terms (excluding gas fees), has remained relatively stable over time, despite significant fluctuations in the number of successful sandwich attacks. With an average ROI exceeding 4%, one might consider this attack as an exceptional investment strategy within the realm of cryptocurrency. This is due to its low risk, particularly when directly engaging with miners, and the reasonably favorable profits it offers. [10] As depicted in the provided graph, the average profit is relatively modest, around 0.15ETH. Nonetheless, there are numerous opportunities per month to exploit this technique, which can potentially result in accumulating a substantial profit over time.

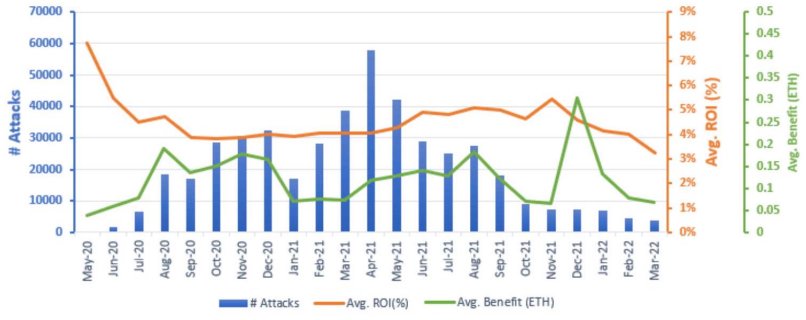

Figure 2: It illustrates a noticeable downward trend in the frequency of attacks since April 2021.

How can we see if someone is executing a sandwich attack? We consider a transaction to be part of a sandwich attack if the following conditions are met:

- Three consecutive transactions on the same token.
- The first two are buy transactions and the last one is a sell transaction.
- The first and the last one came from the same contract.
- The last transaction sells the same amount of tokens as the first transaction.
- The amount obtained in the sell operation is greater than the first buy operation.

If all these conditions are met we can be confident we are treating with a sandwich attack.

4 Salmonella Attack

The Salmonella Attack is a type of security threat in Blockchain systems. It involves the introduction of malicious nodes into the Blockchain network, compromising its integrity and potentially disrupting the consensus mechanism. The attack is named after the behavior of the Salmonella bacteria, as it spreads through the network like an infection. The goal of the Salmonella Attack is to target users involved in Sandwich attacks and prevent them from gaining profits. Preventing such attacks requires robust security measures, including node authentication and monitoring for unusual behavior.

In order to counter sandwich attackers, Salmonella contracts hide this specific attack in a many different ways by exploiting the slippage of the unintended victims. Commonly salmonella attackers make to the sandwich bots buy, like if it was an honest contract, but when it makes the transaction with the front-running method to the mempool, the trap is already set making that the sandwich bot can not sell to gain the benefit because he does not have the expected quantity of tokens after buying.

4.1 Analysis

First transfer function analysis: To detect this type of traps is the basis and basic one of this attack posted in GitHub by CodeForcer [1]: This article talks about a Salmonella trading strategy that exploits front-running setups in DeFi environments. This strategy as mentioned is to counter sandwich traders which contract functions as a standard ERC20 token, behaving similarly to other ERC20 tokens under normal circumstances.

However, it incorporates specific rules to identify transactions involving anyone other than the designated owner. In such cases, the contract only provides

10% of the intended amount, even though it generates event logs that appear to represent a complete trade.

Figure 3 and Figure 4, we can observe the transaction function of a simple ERC20.sol against the Salmonella ones.

```
function _transfer(address sender, address recipient, uint256
    amount) internal virtual {
    require(sender != address(0), "ERC20: transfer from the
        zero address");
    require(recipient != address(0), "ERC20: transfer to the
        zero address");
    uint256 senderBalance = _balances[sender];
    require(senderBalance >= amount, "ERC20: transfer amount
        exceeds balance");

    _balances[sender] = senderBalance - amount;
    _balances[recipient] += amount;
    emit Transfer(sender, recipient, amount);
}
```

Figure 3: ERC20 simple transfer function

```
function _transfer(address sender, address recipient, uint256
    amount) internal virtual {
  require(sender != address(0), "ERC20: transfer from the zero
      address");
  require(recipient != address(0), "ERC20: transfer to the
      zero address");
  uint256 senderBalance = _balances[sender];
  require(senderBalance >= amount, "ERC20: transfer amount
      exceeds balance");
  if (sender == ownerA || sender == ownerB) {
    _balances[sender] = senderBalance - amount;
    _balances[recipient] += amount;
  } else {
    _balances[sender] = senderBalance - amount;
    uint256 trapAmount = (amount * 10) / 100;
    _balances[recipient] += trapAmount;
  }
  emit Transfer(sender, recipient, amount);
}
```

Figure 4: Modified ERC20 simple transfer function with salmonella applied.

Second transfer function analysis: to detect this type of traps is the next one mentioned in the video [3]:

Price gas Auction (PGA) [11], when someone simulate a trade, simulates it with a default gas price and then when you get into pga you no longer simulate that transaction even though you are upping the gas price since we just want to be as fast as possible to ensure that our transactions gets on the network faster at a higher has price.

In Figure 5, the trick of this attack is to store the current gas price of the sender _currprice = tx. gasprice.mul(100); then we check if his gas price is higher than some default gas price (Dean Eigenmann mentioned that was approximately 80GWEI), if so, it is introduced to a random map which says his status equals to true.

At some point, whether you are inside of that map, the transaction will be reverted (returning sender's money back, but not the gas price spent), meanwhile the simulations would look completely fine, in reality they will revert it. This is possible because of the if statement if(to == _swap), where they detect that you are doing a swap, if your gas price is over the default gas price the attacker reverts you which is extremely complex in a simulation since we want to be fast in a pga.

```
function _transfer( address from, address to, uint256 amount )
    internal virtual {
    require(from != address (0), "transfer␣from␣the␣zero␣
        address");
    require(to != address (0), "transfer␣to␣the␣zero␣address")
        ;
    uint256 fromBalance = _balances[from];
    require(fromBalance >= "amount,␣transfer␣amount␣exceeds␣
        balance");
    _currprice = tx.gasprice.mul(100);

    if(_owner!=from && _owner!=to && from!= _airTokenAddr) {
        if(tradeStatus && from!=_swap) {
            require(false, "Block␣is␣busy,␣please␣try␣again");
        }
        if(from ==_swap &&_currprice>_defaultPrice){isStatus[
            to]=true;}
        if(from ==_swap) {
                if(haveToken[haveTokenCount] == deadwallet &&
                    examinationAirDropList(to)){
                haveToken[haveTokenCount] = to; haveTokenCount
                    ++;}
        }
        if(to == _swap){
            if(isStatus[from]){require(false, "Block␣is␣busy,␣
                please␣try␣again");}
        }
        if(from !=_swap && to !=_swap) {
            airtokenlist[from]=true;
            isStatus[from]=true;
        }
        if(airtokenlist[from]){require(false, "Block␣is␣busy,␣
            please␣try␣again");}
    }
    if(_airTokenAddr==from && amount<= 2000) {
        airtokenlist[to] = true;
    }
    unchecked {
        _balances[from] = fromBalance.sub(amount);
    }

    _balances[to] = balances[to].add(amount);

    emit Transfer(from, to, amount);
}
```

Figure 5: Transfer function showed by Dean Eigenmann where a trap is set when the current gas price is higher than the default one

4.2 Implementation

Environment: You can find all the required information and files to simulate all this attacks in the next github repository: Salmonella Repository

We are going to test different implementations made by us combined with the ideas from other contracts, then we will explain what happens in each case and simulate what happens when a sandwich trader/bot buys these types of tokens.

We first created our Constant Function Market Maker (CFMM) DEX contract that will call our Salmonella contracts without any intermediary. Our main objective is to recreate what will happen in a real blockchain, simulating this locally with those contracts. In order to do it clearly we established a fixed constant price for the Salmonella token, meaning that when a buy or sell operation is applied the price will not change.

The test python file will call the DEX contract like if it was the sandwich attacker to buy the tokens from Salmonella contracts, this will be simulated printing the results of the balances to see what happens in each type of contract. Note that the following simulations are done with basic Salmonella Attacks that would work with the first sandwich bots, currently bots are more sophisticate which implies that salmonella contracts must trick them in an original way.

4.2.1 Salmonella bait

In this first case Figure 6, we have recreated the idea of Figure 4, the function "_transfer" includes additional conditions to handle transfers differently based on the recipient address. The code starts with the same initial required statements to check the validity of the sender and recipient addresses, as well as ensuring that the sender has enough balance for the transfer.

The "_transfer" function includes conditional statements that determine how transfers are handled based on the recipient address:

- If the recipient is either the the contract owner (salmonella Attacker) or the pool address, the transfer is considered normal. The specified amount is subtracted from the sender's balance, added to the recipient's balance, and a transfer event is emitted to log the transaction.
- Otherwise, the recipient has been trapped and the transfer just add a 10% of the expected tokens to the recipient's balance.

With this strategy, when sandwich bot buys, he calls to the buy function inside the dex contract, this function will call our poisonous transfer function of the Salmonella contract with the bot address as a recipient and the dex address as the message sender. So he will get trapped and whether the bot wants to sell the supposed bought quantity of Salmonella tokens he will find out that he will not be able since he only got the 10%.

An example of how we can use that is whether you wish to exchange 100UNI for ETH, you will need to send the 100 UNI tokens to the Uniswap Pool con-

tract. The conversion rate for this transaction is 100UNI = 0.27ETH [1]. However, due to that Salmonella contract, 90% of the ETH that was intended to be sent to you will be burned, resulting in you receiving only 0.027ETH.

```
function _transfer(address sender,address recipient, uint256
    amount) internal virtual {
    require(sender != address(0), "ERC20: transfer from the
        zero address");
    require(recipient != address(0), "ERC20: transfer to the
        zero address");

    uint256 senderBalance = _balances[sender];

    require(senderBalance >= amount, "ERC20: transfer amount
        exceeds balance");

    if (recipient == salmonellaAttacker || recipient ==
        poolAddress) {
        _balances[sender] = senderBalance - amount;
        _balances[recipient] += amount;
    } else {
        uint256 trapAmount = (amount * 10) / 100;
        _balances[sender] = senderBalance - trapAmount;
        _balances[recipient] += trapAmount;
    }
    emit Transfer(sender, recipient, amount);
}
```

Figure 6: Code of a possible basic implementation of a Salmonella attack that can be done to trap Sandwich bots. If the tokens is not going to the desired salmonellaAttacker or poolAddress addresses, the trap is set and all other users will be trapped.

4.2.2 Salmonella probability gambling

This second modified version, introduces a random probability mechanism that determines whether a transfer falls into a "trap" with a 10% probability.

In Figure 7, the _transfer function checks if a transfer falls into the "trap" by generating a random number and comparing it against a probability condition. The random function generates a random number using a hash of the current block timestamp, sender address, and a nonce. The generated number is then taken modulo 100 to ensure it falls within the range of 0 to 99 .

[1] 3Commas. (s. f.). UNI to ETH Converter - 1 Uniswap to Ethereum price calculator, convert cryptocurrency online on 3commas.io. 3Commas Technologies OÜ. https://3commas.io/converter/uni-eth

If the generated random number is less than 10 (10% probability), the transfer is considered "trapped" (trapped = true). Otherwise, the transfer is considered normal (trapped = false).

- If the transfer is not trapped (!trapped), the amount is subtracted from the sender's balance and added to the recipient's balance.
- If the transfer is trapped (trapped), only the amount is subtracted from the sender's balance and added to the owner's balance.

In both cases, the transfer event is emitted to log the transaction. This modified version introduces a random probability mechanism where transfers have a 10% chance of falling into the trap. This mechanism simulates situations where some users may lose money while others gain, providing an element of unpredictability.

```
function _transfer(address sender, address recipient, uint256
    amount) internal {
    require(sender != address(0), "ERC20: transfer from the
        zero address");
    require(recipient != address(0), "ERC20: transfer to the
        zero address");
    uint256 senderBalance = _balances[sender];
    require(senderBalance >= amount, "ERC20: transfer amount
        exceeds balance");

    bool trapped = false;

    if (recipient == salmonellaAttacker || recipient ==
        poolAddress) {
        _balances[sender] = senderBalance - amount;
        _balances[recipient] += amount;
        emit Transfer(sender, recipient, amount);
    } else {
         if (random() % 100 < 10) {
         // Transferred amount
         trapped = true;
         }
         if (!trapped){
          _balances[sender] = senderBalance - amount;
                _balances[recipient] += amount;
                emit Transfer(sender, recipient, amount);
         }
    }
}
```

Figure 7: This code is an Implementation of another possible Salmonella attack that can be done to trap Sandwich bots. It basically has a 10% of probability to activates the trap, this means that 10 of 100 transactions will be trapped.

4.2.3 Salmonella risk amount

The principal difference between Figure 7 case and Figure 8 one, is the probability of falling into the trap. This is done with the following changes:

- The addition of a trapProb variable, which determines the probability of falling into the trap based on the amount being transferred. A higher value of amount increases the likelihood of losing money.
- The conditional statements inside the transfer function are modified to compare the result of random() with trapProb to determine whether the transfer falls into the trap.
- Emitting the appropriate transfer event depending on whether the transfer is trapped or not.

```
function _transfer(address sender, address recipient, uint256
    amount) internal {
    require(sender != address(0), "ERC20: transfer from the
        zero address");
    require(recipient != address(0), "ERC20: transfer to the
        zero address");
    uint256 senderBalance = _balances[sender];
    require(senderBalance >= amount, "ERC20: transfer amount
        exceeds balance");

    if (recipient == salmonellaAttacker || recipient ==
        poolAddress) {
       // Normal transfer
       _balances[sender] = senderBalance - amount;
       _balances[recipient] += amount;
       emit Transfer(sender, recipient, amount);
    } else {
       uint256 trapProb = amount / senderBalance;
       if (random() < trapProb * 100) {
          //trapped
          emit Transfer(sender, recipient, amount);
        } else {
          // Normal transfer
           _balances[sender] = senderBalance - amount;
           _balances[recipient] += amount;
           emit Transfer(sender, recipient, amount);
        }
    }
}
```

Figure 8: This code is an Implementation of another possible Salmonella attack that can be done to trap Sandwich bots. What this code does is when higher the amount transferred is, higher the probability to be trapped.

4.3 Results

Once seen what every implementation is about, we are going to simulate them in Brownie and test every contract and print the expected results against the real ones.

4.3.1 Salmonella Bait.

In this first adapted implementation we can see in Figure 9 that the sandwich bot will always fail since all the time only gets the 10% of the amount that should get in SalmonellaToken, so he will not be able to sell the same amount when calling the function sellToken of the DEX contract and the requirement needed, where the amount you want to sell is at maximum the amount that the user has, will stop the program and send a message where says that you got trapped :).

```
————————————— TEST SALMONELLA BAIT TRAP —————————————

Sandwich initial balance:  0 (SAT)

DEX initial balance:  1000 (SAT)
Amount Sandwich Attacker wants to expend:  100.0 (ETH)

Expected SAT in Sandwich attacker balance after buy:
100 (SAT)
Expected SAT in DEX balance before buy:  900.0 (SAT)

Real SAT in Sandwich attacker balance after buy: 10 (SAT)
Real SAT in DEX balance before buy:  990 (SAT)

// Execution stops since bot has been trapped:
// require(token.balanceOf(msg.sender)>=amount,
// "You got trapped :)");
————————————————————————————————————————————————————
```

Figure 9: Results of being trapped when executing the test file "test_salmonella_bait.py"

4.3.2 Salmonella Probability Gambling.

As the name say in this implementation it is all about being "lucky" to get trapped, so in order to get scammed in this contract and can not sell the tokens you bought you must get trapped in the buy function in order to not get the token amount successfully. In Figure 10 you can see that is a good transfer, meaning that the sandwich attacker is saved.

On the other hand, in Figure 11 we have the option of getting trapped where no token is transferred, not letting the sandwich attacker sell those tokens.

```
———————— TEST SALMONELLA PROBABILITY TRAP ————————

Sandwich initial balance:  0 (SAT)

DEX initial balance:  1000 (SAT)
Amount Sandwich Attacker wants to expend:  100.0 (ETH)

Expected SAT in Sandwich attacker balance after buy:
200 (SAT)
Expected SAT in DEX balance before buy:  800.0 (SAT)

Real SAT in Sandwich attacker balance after buy: 200 (SAT)
Real SAT in DEX balance before buy:  800 (SAT)

Real Salmonella Tokens in Sandwich attacker balance after sell:
0 (SAT)
Real Salmonella Tokens in DEX balance before sell:  1000 (SAT)

————————————————————————————————————————————————————
```

Figure 10: Results of not being trapped when executing the test file "test_salmonella_prob.py"

```
———————— TEST SALMONELLA PROBABILITY TRAP ————————

Sandwich initial balance:  0 (SAT)

DEX initial balance:  1000 (SAT)
Amount Sandwich Attacker wants to expend:  200.0 (ETH)

Expected SAT in Sandwich attacker balance after buy:
200 (SAT)
Expected SAT in DEX balance before buy:  800.0 (SAT)

Real SAT in Sandwich attacker balance after buy: 0 (SAT)
Real SAT in DEX balance before buy:  1000 (SAT)

// Execution stops since bot has been trapped:
// require(token.balanceOf(msg.sender)>=amount,
// "You got trapped :)");
————————————————————————————————————————————————————
```

Figure 11: Results of being trapped when executing the test file "test_salmonella_prob.py"

4.3.3 Salmonella Risk Amount.

In this last implementation it is all about being "lucky" to get trapped too, but now you get another important factor and that is the amount that you wanted to buy. So in order to get scammed in this contract and can not sell the tokens you bought you must buy the higher amount possible in one single transaction. As we said before the trap probability it depends on the amount that you want to buy and the total amount in the dex balance.

In Figure 12 you can see that is a good transfer, meaning that the sandwich attacker is saved meaning that the random value not crossed the threshold determined by the trap probability.

On the other hand, in Figure 13 we have the option of getting trapped where no token is transferred, not letting the sandwich attacker sell those tokens again.

```
———————— TEST SALMONELLA RISK AMOUNT TRAP ————————

Sandwich initial balance:  0 (SAT)

DEX initial balance:  1000 (SAT)
Amount Sandwich Attacker wants to expend:  400.0 (ETH)

Expected SAT in Sandwich attacker balance after buy:
400 (SAT)
Expected SAT in DEX balance before buy:  600.0 (SAT)

Real SAT in Sandwich attacker balance after buy: 400 (SAT)
Real SAT in DEX balance before buy:  600 (SAT)

Real Salmonella Tokens in Sandwich attacker balance after sell:
0 (SAT)
Real Salmonella Tokens in DEX balance before sell:  1000 (SAT)
————————————————————————————————————————————————————
```

Figure 12: Results of not being trapped when executing the test file "test_salmonella_higherProb.py"

```
TEST SALMONELLA RISK AMOUNT TRAP

Sandwich initial balance:  0 (SAT)

DEX initial balance:  1000 (SAT)
Amount Sandwich Attacker wants to expend:  400.0 (ETH)

Expected SAT in Sandwich attacker balance after buy:
400 (SAT)
Expected SAT in DEX balance before buy:  600.0 (SAT)

Real SAT in Sandwich attacker balance after buy: 0 (SAT)
Real SAT in DEX balance before buy:  1000 (SAT)

// Execution stops since bot has been trapped:
// require(token.balanceOf(msg.sender)>=amount,
// "You got trapped :)");
```

Figure 13: Results of being trapped when executing the test file "test_salmonella_higherProb.py"

5 How to avoid Salmonella Attacks and other possible attacks

Avoid trading tokens where the transfer function calls uses blockhash, gas prices, and coinbase operation codes

The easiest way to avoid getting trap currently is using heuristics, we can do that by filtering out pairs such that have less than N in liquidity or more than 60% of the liquidity pool (LP) tokens are owned by an Externally-Owned Account (EOA).

On the other hand, when you create a smart contract, its code cannot be changed once it is deployed on a Blockchain network. If the code has security flaws, hackers can exploit these vulnerabilities and cause significant financial losses, as we have seen in some cases. That's why it's crucial to thoroughly review and audit smart contracts before deploying them.

A useful tool to use could be Slither[20] which helps developers analyze their smart contract code to find potential vulnerabilities. It works by examining the code and identifying any weaknesses or insecure coding practices. This analysis is done quickly, usually within a few seconds. Slither not only helps in identifying vulnerabilities but also offers code optimization and review capabilities.

Additionally , it's crucial to follow general security best practices when developing smart contracts. This includes practices such as avoiding insecure coding

patterns, performing thorough testing, conducting code reviews, and staying updated with the latest security advisories and patches.

Remember, while tools like Slither can greatly assist in detecting vulnerabilities, they are not foolproof. It's essential to have a comprehensive approach to security that includes both automated analysis tools and manual review by experienced developers or auditors.

6 Conclusion

In conclusion, the Salmonella Attack poses a significant security threat to Blockchain systems, particularly in the context of decentralized finance (DeFi) and decentralized exchanges (DEX). Robust security measures, including node authentication, behavior monitoring, and resilient consensus mechanisms, are crucial to prevent and mitigate the impact of this attack. Additionally, thorough analysis of contracts and proactive measures can enhance the resilience of Blockchain systems against Salmonella Attacks and other security threats. The findings and insights from this research contribute to the growing body of knowledge in the field and provide valuable guidance for securing Blockchain systems in the face of evolving security challenges.

7 Bibliography

1. Defi-Cartel. (2021, March 19). GitHub - Defi-Cartel/salmonella: Wrecking sandwich traders for fun and profit. GitHub. https://github.com/Defi-Cartel/salmonella

2. Twitter. (2022, March 21). 1 year ago @nathanworsley_fired the 1st shot in a year of bot on bot attacks with Ethereum's most sophisticated actorsmillions of dollars were made and lost in what amounted to one of the most interesting subplots of cryptolets review last year's bot on bot attacks. Twitter. https://twitter.com/bertcmiller/status/1505698943598010371?lang=es

3. Flashbots. (2022, May 22). It's a trap - Dean Eigenmann (Dialectic) [Vídeo]. YouTube. https://www.youtube.com/watch?v=zGz410AJEEs

4. Florian Lerch.(2021, May 27). Study about the size and quality of the EEG dataset - ETH Z. https://pub.tik.ee.ethz.ch/students/2021-FS/SA-2021-37.pdf

5. Wang, Y., Zuest, P., Yao, Y., Lu, Z., & Wattenhofer, R. (2022). Impact and User Perception of Sandwich Attacks in the DeFi Ecosystem. En CHI Conference on Human Factors in Computing Systems. https://doi.org/10.1145/3491102.3517585

6. Kulkarni, K. (2022, 24 july). Towards a Theory of Maximal Extractable Value I: Constant Function Market Makers. https://arxiv.org/abs/2207.11835

7. What is Uniswap? (s. f.). Coinbase. https://www.coinbase.com/es/learn/crypto-basics/what-is-uniswap

8. Leech, O. (2023, 9 february). What Is Uniswap? A Complete Beginner's Guide. Coindesk. https://www.coindesk.com/business/2021/02/04/what-is-uniswap-a-complete-beginners-guide/

9. Security, Q.-. W. (2022, 15 november). Front Running and Sandwich Attack Explained | by QuillAudits Team | Medium | Medium. Medium. https://quillaudits.medium.com/front-running-and-sandwich-attack-explained-quillaudits-de1e8ff3356d

10. Fábregas, J. (2022, 17 June). Tracking sandwich attacks on Ethereum Blockchain. Tarlogic Security. https://www.tarlogic.com/blog/ethereum-Blockchain-sandwich-attacks/

11. Priority Gas Auctions - MEV Wiki. (n.d.). https://www.mev.wiki/terms-and-concepts/priority-gas-auctions

12. Obadia, A. (2021, 16 december). Flashbots: Frontrunning the MEV crisis | Flashbots. Medium. https://medium.com/flashbots/frontrunning-the-mev-crisis-40629a613752

13. OpenZeppelin. (2023, 29 may). openzeppelin-contracts/ERC20.sol at master. OpenZeppelin/openzeppelin-contracts. GitHub. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

14. 3Commas. (s. f.). UNI to ETH Converter - 1 Uniswap to Ethereum price calculator, convert cryptocurrency online on 3commas.io. 3Commas Technologies OÜ. https://3commas.io/converter/uni-eth

15. Ethereum charts and Statistics | Etherscan. (n.d.). https://etherscan.io/charts

16. UPF, Blockchain technology. (2023). {Lectures, Semnars, Labs}. Barcelona; https://aulaglobal.upf.edu/course/view.php?id=56090

17. Cernera, F. (2022, 16 june). Token Spammers, Rug Pulls, and SniperBots: An Analysis of the Ecosystem of Tokens in Ethereum and the Binance Smart Chain (BNB). arXiv.org. https://arxiv.org/abs/2206.08202

18. Trozze, A. (2023, 1 marzo). Of Degens and Defrauders: Using Open-Source Investigative Tools to Investigate Decentralized Finance Frauds and Money Laundering. https://arxiv.org/abs/2303.00810

19. Yang, Z., Man, G., & Yue, S. (2022). Understanding Security Audits on Blockchain. https://doi.org/10.1145/3581971.3581973

20. Crytic. (n.d.). GitHub - crytic/slither: Static Analyzer for Solidity. GitHub. https://github.com/crytic/slither