itworldcanada.com

Understanding Cybersecurity Management in DeFi (UCM-DeFi) – Smart Contracts and DeFi Security and Threats (Article 5) - IT World Canada

Sepideh HajiHosseinKhani and Arash Habibi Lashkari

20-25 minutes

Smart contracts have transformed how legal agreements are managed and carried out, but they come with their own share of potential flaws and security risks. These shortcomings make smart contracts susceptible to hacking, which can lead to substantial financial losses. For instance, security weaknesses in smart contracts can be manipulated to illicitly extract funds. A considerable number of attacks target smart contracts relative to other layers and components. For example, in 2016, DAO was victim of an attacker who manipulated a smart contract bug to repeatedly siphon funds, leading to an approximately US$50 million cryptocurrency loss for investors.

Understanding Cybersecurity Management in DeFi (UCM-DeFi), a five-article series, aims to discuss decentralized finance and explore a range of cybersecurity issues that impact DeFi and blockchain-based financial solutions. The articles in this series are

based on the recent book titled *Understanding Cybersecurity Management for DeFi*, published by Springer this year. This fifth and last article discusses significant smart contract vulnerabilities and risks that pose serious challenges for their creators. We'll also cover a few attack scenario examples to demonstrate the very real nature of these threats. Continue reading to learn more about these vulnerabilities, and the strategies to prevent smart contract security issues.

The previous four articles in this series are available here:

Understanding Cybersecurity Management in DeFi (UCM-DeFi) – The Origin of Modern Decentralized Finance (Article 1)

Understanding Cybersecurity Management in DeFi (UCM-DeFi) – Introduction to Smart Contracts and DeFi (Article 2)

Understanding Cybersecurity Management in DeFi (UCM-DeFi) – DeFi Platforms (Article 3)

Understanding Cybersecurity Management in DeFi (UCM-DeFi) – Blockchain Security (Article 4)

Contents

1. Arithmetic Bugs

Arithmetic bugs in smart contracts occur during arithmetic operations on integers, including situations like overflow, underflow, and division by zero. In overflow, a value exceeds the integer limit, while underflow happens when a value is smaller than the integer boundary. These operations involve both positive and negative integers.

In most programming languages, when an arithmetic operation exceeds the upper limit (boundary) of what an integer data type can store, it results in an exception or error. This situation is typically considered an “out-of-bounds” behavior. However, Ethereum precisely defines such behaviors due to the Ethereum Virtual Machine’s support for modulo 2^256 arithmetic. Essentially, instead of creating an error, operations that exceed this boundary “wrap around” because of this modulo operation.

In the context of division, most programming languages would throw an error when you attempt to divide any number by zero. However, EVM behaves differently; it simply results in zero. These peculiarities of the EVM can lead to arithmetic bugs.

For clarity, consider a simple example – an Ethereum smart contract has a fixed size incremented by 8 bits. The maximum number from this combination is 2^8 (256), ranging from 0 to 255. Adding 1 to the maximum value causes an overflow, and adding -1 to the minimum causes an underflow.

In later versions of Solidity (>0.4.0), the compiler introduces a feature where certain operations would indeed trigger exceptions. When an exception is triggered, all changes made in the current call (and any calls it made) are reversed, providing a mechanism to handle these situations.

2. Re-entrancy Attack

Re-entrancy attacks pose a serious threat to smart contracts. In such attacks, a contract (A) calls another contract (B), which in turn calls back contract A, all within a single transaction. Notably, contract B is external to the blockchain. This sequence, known as legitimate re-entrancy, is common in contract execution.

For example, contract A asks contract B to withdraw a certain amount of money. Contract B follows this instruction and sends the funds to contract A's account using a callback or fallback function. In Ethereum, transferring Ether happens via function calls. Hence, contract B calls back contract A to fulfill the instruction.

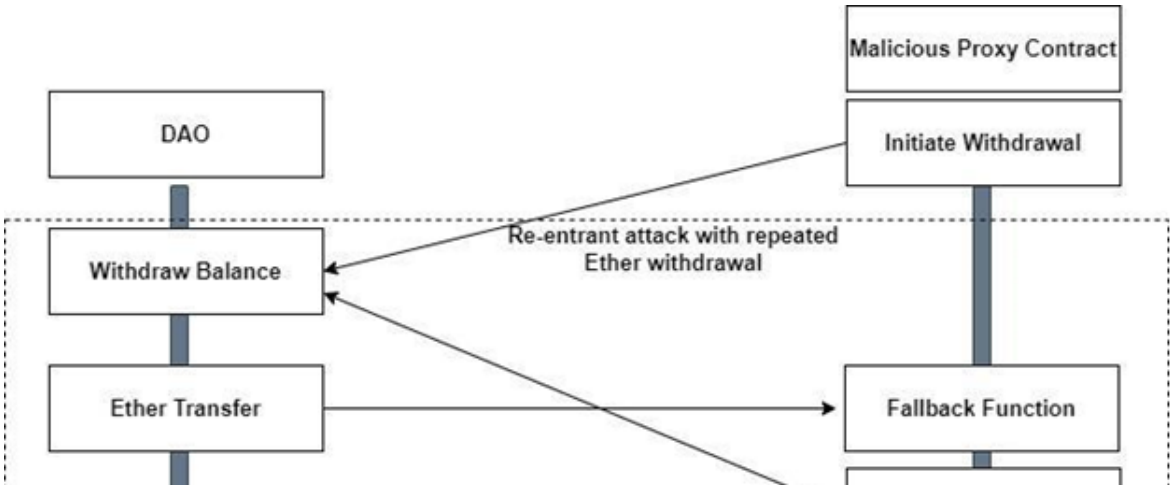
In a malicious re-entrancy or re-entrancy attack, a contract operates on an inconsistent internal state due to unexpected calling. For instance, a contract executes a control flow based on the victim contract's internal state. The internal state gets updated only after the external call returns, leaving the contract in an inconsistent state.

To elaborate, a procedure is re-entrant if its execution can be paused and restarted, and both instances run without errors. This characteristic can result in severe vulnerabilities for smart contracts. The DAO hack is a notable example.

Consider a victim contract with a withdrawal function. This function checks whether a calling contract can withdraw an amount. If so, it transfers the specified sum to the calling contract and updates the internal state. A malicious contract can exploit this by calling the victim contract to withdraw again before the internal state is updated. This situation results in multiple withdrawals before the state gets updated.

Here's how a step-by-step re-entrancy attack might occur:

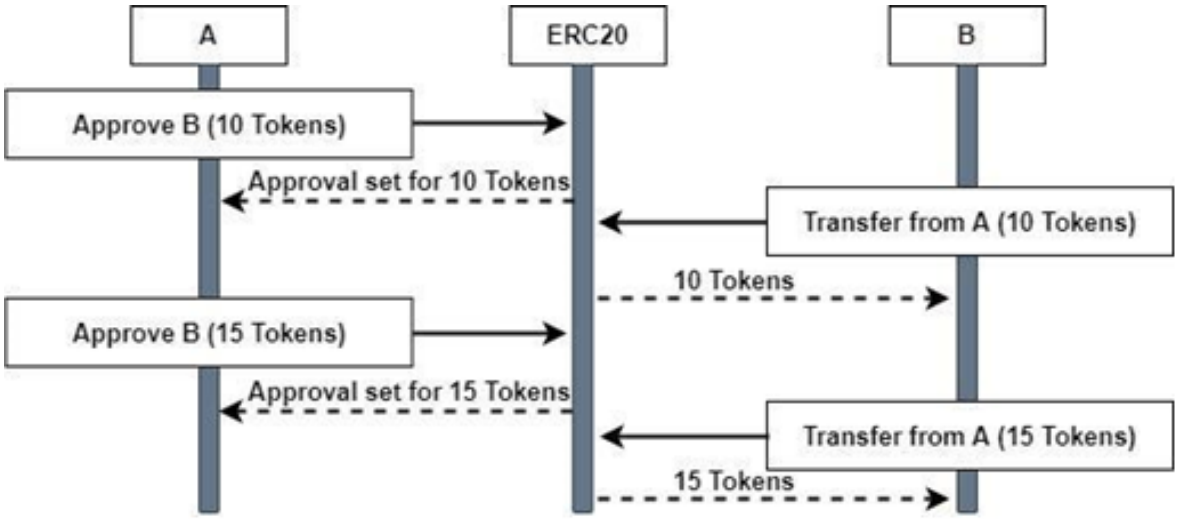
1. The proxy contract requests a withdrawal.
2. The victim contract's transfer to the proxy contract triggers the fallback function.
3. The proxy contract asks for another withdrawal.
4. The victim contract's transfer to the proxy contract again triggers the fallback function.
5. This process repeats without updating the balance or throwing an exception unless the transfer fails.



3. Race Conditions

Race conditions arise due to a lack of synchronization in transactions within Solidity and Ethereum smart contracts. A common instance of potential race conditions can be found in the ERC20 standard, which outlines the APIs used in smart contracts.

To illustrate, consider two users, A and B. A wants to allow B to withdraw 10 tokens from A's wallet as payment for a smart contract code developed by B. Assume B successfully negotiates a bonus of 5 tokens for excellent work, totaling 15 tokens. However, before A can send the approved 15 tokens, B initiates a transfer function to withdraw the original 10 tokens. Later, B receives approval for and obtains the additional 15 tokens, resulting in a total of 25 tokens.



This situation, known as a double withdrawal exploit, arises from race conditions and a lack of synchronization. It's noteworthy that re-entrancy attacks are also a type of race condition attack.

To avoid such circumstances, it's advised that token owners reset token allowances to zero before defining a new value. This mitigates the risk of exploitation by taking advantage of race conditions in the ERC20 standard.

4. Unhandled Exceptions

Unhandled exceptions pose a vulnerability in smart contracts, arising in various ways and potentially anywhere within the contract, such as within a loop or a function. Exception handling is critical because errors might not propagate across the call-stack, although this can be dependent on the specific features of the target function. External exceptions in a loop can even pave the way for denial-of-service attacks.

Different smart contract platforms manage exceptions in unique ways. For instance, Solidity has two methods for handling exceptions:

- By directly referencing the callee's contract instance or using the transfer() function, in which case the exception is escalated, and the entire transaction is reverted. Given transactions are atomic, reversing the entire transaction is a primary safety measure.
- When employing one of the four low-level methods, namely call, staticcall, delegatecall, and send, only a false value is returned to the calling contract in case of an exception. This nuanced difference in response helps maintain the security of the overall smart contract ecosystem.

5. Using a Weak Random Generator

Random number generation is essential in computer science for

various applications, including deciding winners in games. However, generating a random number in a deterministic environment like Ethereum can be challenging. Solidity does provide a function, blockhash(uint blockNumber), which generates a ‘0’ when blockNumber is more than 256.

Time-dependent functions are used in smart contracts for synchronization or to block transactions, such as timestamps, block numbers, gas limits, and coinbase. Despite this, some experts advise against using generated random numbers for making crucial decisions, due to their inherent entropy or randomness. Pseudo random number generators (PRNG) can produce weak random numbers which, if used to determine an important contract state, can be misused by attackers to multiply their wins within a winning block.

To prevent these exploitations, the source of randomness, or entropy, should be external to the blockchain. One potential solution could involve changing the trust model to a group of participants or a centralized entity, which could serve as a randomness oracle.

6. Timestamp Dependency

Timestamp dependency is a critical feature of the Ethereum Virtual Machine, often used for transaction synchronization in smart contracts over random number generators. However, this dependency doesn’t offer any information about the environment, such as the host operating system, IP address, or time, which can be derived from the timestamp field in a block’s metadata. Regrettably, this field can be manipulated as the block miner can

insert any timestamp without verification from other network nodes.

This malleability can lead to malicious attacks. For instance, miners can tamper with environment variables to gain profits. Consider a lottery system distributing prizes based on a manipulable function. Suppose a variable determining the winner produces an even or odd number. A miner could adjust the timestamp to their advantage, such as altering it to their local time when the block was created.

Generally, a contract can withstand a 30-second variance and still preserve its integrity. But, the potential for manipulation is more significant if a malicious miner tweaks crucial blockchain components leading to substantial impact.

In such cases, using random numbers to decide outcomes might be preferred. Nevertheless, random numbers aren’t a complete substitute for timestamp dependency due to the latter’s synchronization advantages.

7. Transaction-Ordering Dependence and Front Running

Transaction-ordering attacks are a type of race condition attack that exploits the miner-determined order of transaction processing in Ethereum. These attacks occur when malicious miners manipulate the transaction order to prioritize their transactions, leaving legitimate transactions in a pending state.

For instance, imagine a smart contract rewards the first correct answer to a problem. Alice solves the problem and submits her answer with a standard gas price. Bob sees her answer and

submits the same answer but with a higher gas price. If Bob's transaction is processed first, he'll receive the reward, leaving Alice empty-handed, despite being the original solver.

One way to prevent this type of attack is by using the commit-reveal hash scheme. In this scheme, instead of directly submitting the answer, the solver submits the hash of the answer, which is uninterpretable to others. The contract then stores the hash along with the user's address. To claim the reward, the solver must submit the answer, address, and a unique number (salt), which, when hashed, should match the previously stored hash. If a match is found, the reward goes to the claimant.

8. Vulnerable Libraries

Libraries in smart contracts provide reusable functionalities, such as data structures and token contract interfaces. However, they carry inherent security risks. If a library is vulnerable and multiple smart contracts use it, the vulnerability will affect all these contracts. Furthermore, once deployed, a library can't be patched, and many client contracts lack versioning capabilities, preventing a vulnerable library from being updated.

One prominent example of library vulnerability is the 2017 Parity multi-signature wallet hacks, where a vulnerability allowed unauthorized transfer of funds.

The primary source of library vulnerabilities is their statefulness. If libraries were stateless, only the state of the client contracts would change when a library is called, reducing the potential for vulnerabilities.

9. Wrong Initial Assumptions

Incorrect initial assumptions in contract logic often lead to unexpected outcomes. For example, a contract may change a transaction's state based on the received funds. If the funds are below a threshold, the code executes; if not, the transaction reverts.

In the following code snippet, the condition checks whether the total received funds are greater than or equal to the contract balance. If true, the code executes; if not, an exception is thrown.

```
function()
{
   require(ActiveSale);
   fundRaised = fundRaised.add(msg.value);
   require(fundRaised >= this.balance);
   …
}
```

However, if the contract owner mistakenly assumes the contract balance will always be greater than or equal to the received funds, it will consistently throw an exception regardless of other factors. This highlights the critical importance of correct assumptions in smart contract programming.

10. Denial of Service

Denial of Service (DoS) attacks are designed to prevent legitimate users from accessing or using smart contracts, either temporarily or indefinitely. Within a blockchain context, there are three primary types of DoS attacks:

1. Unexpected Revert

   In this type of attack, a smart contract allows a user to place a bid. If a higher bid comes in, the contract refunds the previous bid. However, the attack exploits the contract by causing unexpected reverts when higher bids are received. This vulnerability often stems from inadequate exception handling within conditional and iterative statements.

   To mitigate the impact of this vulnerability, external calls initiated by the callee contract should be placed within separate transactions.

2. Block Gas Limit

   This attack occurs when a transaction exceeds the maximum available gas limit, resulting in transaction failure. If this happens during a refund process, it halts execution and the refunds become stuck indefinitely.

   An attacker can engineer this situation by using a loop that continuously increments a variable without checking the upper limit for that variable's value. In the worst-case scenario, the transaction is permanently blocked, preventing additional transactions.

3. Block Stuffing

   In a block stuffing attack, an attacker fills multiple blocks in the blockchain to prevent other transactions from being included in the blocks. This is achieved when the attacker uses a high gas price for transactions, ensuring that only their transactions are included

in the blocks. An example of a block stuffing attack was seen in the gambling app Fomo3D.

Failed External Calls

In addition to these three types, DoS attacks can occur due to failed external calls, either accidental or deliberate. The damage done by these calls can be minimized by isolating such calls into their own transaction that can be initiated by the recipient of the call. This is particularly relevant for payments where it's better to let users withdraw funds rather than pushing funds to them automatically. This is known as the pull payment model.

11. Flash Loan Attacks

Flash loan attacks pose a significant risk in the world of decentralized finance (DeFi). These attacks leverage the features of flash loans, a unique financial instrument that allows users to borrow substantial amounts of cryptocurrency without any collateral as long as the loan is paid back within the same transaction.

Attackers exploit these features by borrowing vast amounts of funds, manipulating the price of a crypto asset on one exchange, and then reselling it to another user for a profit. The entire process happens very quickly, allowing the attacker to repeat it multiple times before detection and often resulting in substantial theft of cryptocurrency.

One of the reasons flash loan attacks are so common is because of price discrepancies in the same asset across different DeFi platforms. These discrepancies create an opportunity for arbitrage, where an asset can be bought for a lower price on one platform

and sold for a higher price on another.

However, mitigating these attacks is challenging for several reasons. First, it is virtually impossible to patch all existing vulnerabilities due to the complex nature of smart contracts. Second, the fast pace of technological development often means that security considerations may be overlooked. And third, once attackers understand how the contract code operates, they can easily manipulate it to their advantage.

To prevent flash loan attacks, several measures can be taken. These include incorporating robust security features into the design of DeFi platforms, deploying advanced security tools for regular audits and anomaly detection, and using decentralized Oracles for accurate and tamper-proof pricing information. Despite these efforts, the battle against flash loan attacks is ongoing, with new strategies and defenses continually being developed.

12. Vampire Attacks

A “vampire attack” in the world of smart contracts is a strategy where one Decentralized Finance (DeFi) protocol seeks to draw users, investors, and liquidity from another DeFi protocol by offering improved rates or incentives. It essentially attempts to “drain” the resources of competing protocols.

A well-known example of a vampire attack is the case of SushiSwap. This project managed to attract over $1 billion of liquidity in less than a week after its launch. SushiSwap did this by forking Uniswap’s code, then launching a vampire attack. It enticed users with its own tokens, aptly named “Sushi,” which were issued and distributed to participants. As these tokens gained traction, the

protocol siphoned off significant amounts of liquidity from Uniswap.

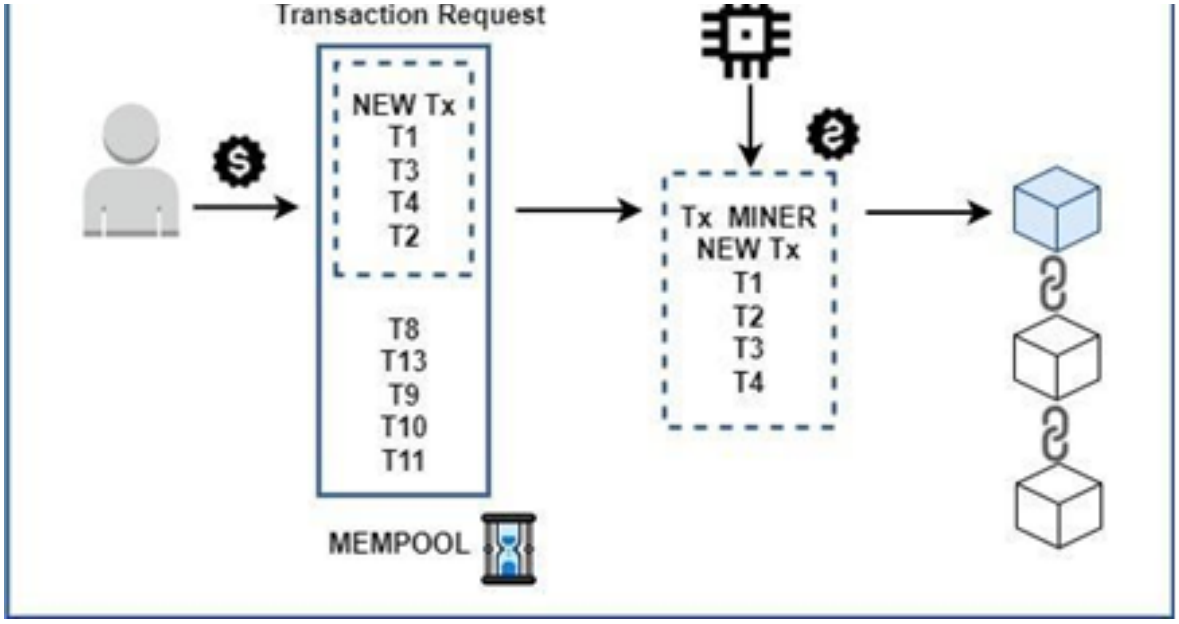
The final stage of the vampire attack involves migrating these tokens from the original platform (in this case, Uniswap) to the new platform (SushiSwap). Hence the name “vampire attack,” as it drains the lifeblood (liquidity) from one protocol to another.

To counteract such attacks, existing DeFi protocols can implement strategies like liquidity reservation offers, which provide additional benefits to their users and encourage them to stay with the existing protocol. It’s a challenge of balancing competitive rates and incentives while ensuring the long-term sustainability of the protocol.

13. Minimal Extractable Volume (MEV)

Maximal Extractable Value (MEV) is a measure of the maximum profit a miner or validator can make by reordering, including, or excluding transactions within a block. Originally known as “miner extractable value” in the context of proof-of-work (PoW), it was renamed to encompass a broader scope, as the process can be controlled by anyone involved in block creation, not just miners.

In essence, MEV quantifies the financial benefit that can be gained through transaction prioritization. Miners, incentivized by the potential of additional revenue, aim to include transactions that offer the highest fees. This allows them to extract value beyond the standard block rewards and gas fees. However, it’s not just miners who exploit this mechanism; network participants, often called “searchers,” also utilize complex algorithms or automated bots to identify profitable transaction opportunities within the blockchain.



While MEV has led to increased utilization of blockchains, attracting participants willing to pay higher gas fees, it also has downsides. High-volume trading, for example, can lead to network congestion, negatively affecting user experience. Common examples of MEV opportunities include decentralized exchanges (DEX), liquidations, and sandwich trading.

Preventing or mitigating the effects of MEV is a complex issue in blockchain protocol design and is a topic of ongoing research in the field.

14. Final Thoughts

The article outlines numerous crucial vulnerabilities and threats in smart contracts that present significant challenges to security experts, designers, and developers. These vulnerabilities can be grouped into categories such as inherent issues, owner’s errors, and unhandled problems.

Common vulnerabilities like arithmetic bugs and re-entrancy attacks have caused significant damage in the past, while others, like race conditions, are inherent in the smart contract standard

itself. Faulty initial assumptions often result from contract owners' mistakes. Additionally, there are vulnerabilities that cannot be remedied by designers due to the lack of versioning support in DeFi platforms.

It's crucial to understand and address these vulnerabilities to ensure the safety and efficiency of smart contracts, thereby fostering a more secure and reliable environment for blockchain technology.