Contents

Introduction

Blockchain technology has revolutionized the way we manage data and transactions, introducing a decentralized paradigm that eliminates the need for intermediaries. At the heart of this innovation is a distributed, secure and immutable ledger that allows multiple parties to collaborate in an environment of trust, ensuring transparency and the integrity of information. This approach has opened up new opportunities in many sectors, from finance to supply chains, from health care to the public sector.

Solana's blockchain represents one of the most interesting developments in this landscape. Designed to offer high scalability and high performance, Solana is distinguished by its ability to process thousands of transactions per second, thanks to innovations such as the Proof of History (PoH) mechanism, which efficiently synchronizes and sorts events. This architecture enables not only extremely low transaction costs but also low latency, which are key elements for performance-critical applications.

Despite its many advantages, Solana, like any platform technology, is not without vulnerabilities. The complexities inherent in the code, especially in environments with a high volume of transactions and complex operations, can lead to errors and flaws that can be exploited by malicious parties. This paper aims to analyze in detail the main vulnerabilities found in the Solana blockchain, highlighting the most common issues, providing code examples for each, explaining how they work, how they can be exploited, and finally, the best strategies to mitigate them.

The information in this document is based on vulnerability analyses reported at https://defisec.info/solana _top _vulnerabilities.

Using a structured and in-depth approach, we are going to examine vulnerabilities, starting with classic problems such as the 'overflow/underflow' of integers, through more complex issues such as memory safety in Rust and input validation .

Smart Contract

Smart contracts are self-executing programs that reside on the blockchain and automate the 'execution of agreements and business logic, without the intervention of third parties. They operate in a transparent and immutable manner. They operate in a transparent and unchanging manner , ensuring that the established rules are strictly adhered to. Once distributed on the blockchain , smart contracts cannot be modified, ensuring a high level of reliability but also requiring meticulous attention during development and review.

In the blockchain of Solana , smart contracts are usually developed in languages such as Rust , which takes advantage of the security and performance features inherent in . Their architecture allows them to handle a high volume of transactions and transactions in real time , making them essential for the creation of decentralized applications(dApps) with high performance.

Below, a basic example of smart contract in Rust for Solana:

```
use solana_program::{
    account_info::{next_account_info, AccountInfo},
    entrypoint,
    entrypoint::ProgramResult,
    pubkey::Pubkey,
};

entrypoint!(process_instruction);

fn process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
```

```
) -> ProgramResult {
    // This is where the logic of the contract is handled
    // For example, process `instruction_data` and interact with the passed accounts.
    Ok(())
}
```

This example illustrates the typical structure of a smart contract in Solana: the function `process_instruction` serves as the input, where logic is handled based on the data input and the accounts involved. It is critical to integrate controls security and input validation to avoid vulnerabilities that could be exploited by malicious actors.

Vulnerabilities

Integer Overflow/Underflow

Description

Integer overflow and **underflow** occur when a numerical value exceeds its binary representation limits. In Solana , smart contracts are mainly developed in **Rust**, a language that, by default, does not perform checks on integer overflows in **release mode**. This can lead to unexpected and potentially dangerous.

- **Integer Overflow**: occurs when a number exceeds the maximum value that can be represented by an integer.
- **Integer Underflow**: occurs when a number falls below the minimum representable value.

This vulnerability can be exploited to manipulate numeric values critical, such as account balances or amounts of tokens transferred.

Example

```
fn transfer_funds(mut balance: u64, amount: u64) -> u64 {
    balance += amount; // Possible overflow
    balance
}
```

If `balance` is close to the maximum representable value (`u64::MAX`), add `amount` can cause an overflow, causing the value to return to **zero or to a very low value** , creating an exploitable flaw.

Exploitation

An attacker could execute a transaction with a high value per amount, causing an overflow and manipulating balances to their own advantage. For example:

1. A malicious user could send a transaction with a deliberately large amount.
2. If the contract does not handle the overflow correctly, the balance could become **suddenly very low or reset to zero** .
3. This could allow the attacker to **evade limits or with draw funds illicitly**.

Solution

A simple solution is to use Rust' s functions that safely handle safely arithmetic operations, such as `checked_add()`:

```rust
fn transfer_funds(mut balance: u64, amount: u64) -> Option<u64> {
    balance = balance.checked_add(amount)?; // Check overflow
    Some(balance)
}
```

In this case, if an overflow occurs, the function will return `None` instead of an incorrect value, preventing the bug.

Conclusion

Overflow and underflow of integers are common and dangerous vulnerabilities that can lead to loss of funds or manipulation of data on Solana. Using the functions of Rust for arithmetic checking and carefully validating inputs, the developers can mitigate this risk effectively.

Memory Safety in Rust

Description

Memory management is a critical aspect in the security of the smart contracts on Solana, which are primarily developed in **Rust** . Although Rust is known for its **system of ownership and borrowing** , which prevents many errors in memory management , some vulnerabilities may still emerge due to the misuse of low-level APIs or unsafe operations.

The main issues include:

- **Use-After-Free**: accessing deallocated memory.
- **Null Pointer Dereference**: attempt to dereference a null pointer.
- **Use of uninitialized memory**: use of uninitialized data that may contain arbitrary values.
- **Double Free**: deallocation of already released memory, causing unpredictable behaviors.
- **Buffer Overflow**: Writing outside the bounds of an array, overwriting adjacent data.

If exploited, these vulnerabilities can lead to crashes, corruption data and even execution of malicious code

Example

Example of **Use-After-Free** in a Rust contract for Solana:

```rust
use std::ptr;

fn unsafe_access() {
    let mut data = vec![10, 20, 30];
    let ptr = data.as_mut_ptr();

    // Memory deallocation before use
    drop(data);

    unsafe {
        // Access to deallocated memory
        println!("Value: {}", *ptr);
    }
}
```

In this case, the pointer `ptr` points to memory that has already been released, leading to unpredictable behavior.

Exploitation

An attacker could exploit this vulnerability to:

1. **Gaining access to sensitive data** if deallocated memory Is reused with unprotected data.
2. **Manipulate data** before they are used by the contract,creating exploit scenarios.
3. **Crash the smart contract**, making the service unusable.

Solution

The main solution is to **avoid unsafe code** and let Rust manage the memory safely.

Correction with ownership and borrowing

```
fn safe_access() {
    let data = vec![10, 20, 30];

    // Safe Borrowing
    let value = data[0];

    println!("Value: {}", value);
}
```

Here we avoid using manual pointers, eliminating the risk of **use-after-free** .

Correction for buffer overflow

If an array is read incorrectly:

```
fn unsafe_read(index: usize) -> u8 {
    let data = vec![1, 2, 3];
    data[index] // Panic if index is out of bounds
}
```

A safe solution uses the `get()` which returns `None` if the index is incorrect:

```
fn safe_read(index: usize) -> Option<u8> {
    let data = vec![1, 2, 3];
    data.get(index).copied()
}
```

Conclusion

Solana is based on Rust precisely to provide greater security in memory management, but the use of **unsafe code** or incorrect operations can introduce serious vulnerabilities. Developers should avoid dangerous APIs and take advantage of Rust's **ownership and borrowing system** of Rust to prevent bugs memory-related.

Correct Execution of the Authorization

Description

One of the most common vulnerabilities in Solana's smart contracts is the **absence of adequate controls on authorization of transactions** . This happens when a contract:

- Does not properly verify **ownership of the accounts** involved.

- It does not check **whether the 'account performing the transaction is authorized signer** .
- Accepts input without validating the calling account or instructions passed.

The absence of these controls can lead to exploits such as **theft of funds, unauthorized modification of critical data, or the execution of unintended actions**.

A famous example of this vulnerability is the **Wormhole Bridge exploit**, in which the attacker exploited a lack of validation to mint arbitrary tokens without authorization.

Example

Here is an example of a smart contract that transfers funds without verifying the ownership of the account:

```
fn transfer_funds(
    from: &AccountInfo,
    to: &AccountInfo,
    amount: u64
) -> ProgramResult {
    let mut from_balance = from.lamports();

    if from_balance < amount {
        return Err(ProgramError::InsufficientFunds);
    }

    **from.lamports.borrow_mut() -= amount;
    **to.lamports.borrow_mut() += amount;

    Ok(())
}
```

What is the problem?

- The contract **does not check whether `from` is actually the sender's account**.
- An attacker could **indicate another account as the sender** and embezzle funds from an unsuspecting user.

Exploitation

1. An attacker constructs a transaction by indicating as `from` an account of another person.
2. If the contract does not control the ownership of the 'account, the transaction is executed successfully.
3. Funds are transferred from victim's account to the attacker's account without authorization.

Solution

To avoid this vulnerability, it is critical to **verify that the sender account has signed the transaction** .

Example of correct code:

```
fn secure_transfer_funds(
    accounts: &[AccountInfo],
    amount: u64
) -> ProgramResult {
```

```
    let account_iter = &mut accounts.iter();
    let from = next_account_info(account_iter)?;
    let to = next_account_info(account_iter)?;

    // Verifies that the sender has signed the transaction
    if !from.is_signer {
        return Err(ProgramError::MissingRequiredSignature);
    }

    let mut from_balance = from.lamports();

    if from_balance < amount {
        return Err(ProgramError::InsufficientFunds);
    }

    **from.lamports.borrow_mut() -= amount;
    **to.lamports.borrow_mut() += amount;

    Ok(())
}
```

Why is it safe?

- Check if `from` signed the transaction (`from.is_signer`).
- Prevents transfers from unauthorized accounts .
- Protects users from theft and illegal manipulation.

Conclusion

Mismanagement of authorization can lead to critical exploits such as theft of funds or alteration of sensitive data. Implementing strict controls over account ownership and the transaction signatures is essential to ensure the security of the smart contracts on Solana.

Depth of Cross-Contract Call

Description

In Solana's execution system , calls between contracts arelimited to a maximum depth of 4 levels. This means that one smart contract can invoke another contract, which in turn can invoke a third, and so on, but cannot exceed this limit.

This restriction exists to ensure high performance and prevent **reentrancy** attacks or excessive consumption of computational. However , depth limitation can cause problems when developers implement logic that depends on multiple calls between contracts, without considering this threshold.

If a contract relies on a chain of executions exceeding the limit allowed, the last call will fail, leading to malfunctions unforeseen and possible loss of funds.

Example

```
fn call_another_contract(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8]
```

```
) -> ProgramResult {
    let account_iter = &mut accounts.iter();
    let called_program = next_account_info(account_iter)?;

    let instruction = Instruction::new_with_bytes(*called_program.key, instruction_data,
    ↪ vec![]);
    invoke(&instruction, accounts)?;

    Ok(())
}
```

What is the problem?

If this function is invoked repeatedly all within a sequence of contracts interdependent, could exceed the depth limit imposed by Solana, causing an error in the execution of the last call.

Exploitation

1. A contract can be designed to perform multiple operations concatenated among multiple smart contracts
2. If the execution exceeds the depth limit, the last call fails unexpectedly.
3. This can lead to the loss of funds or the malfunctioning of complex contracts, such as decentralized lending protocols or automated swap systems.

Solution

To mitigate this problem, it is necessary:

- Reduce the number of calls between contracts and merge logic intoa single contract when possible.
- Check the number of invocations and stop execution beforereaching the limit.
- Use alternative strategies such as asynchronous handling of operations or the use of **events and off-chain callbacks** for complex processes.

Example of corrected code:

```
fn call_with_depth_check(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
    current_depth: u8
) -> ProgramResult {
    if current_depth >= 4 {
        return Err(ProgramError::InvalidInstructionData);
    }

    let account_iter = &mut accounts.iter();
    let called_program = next_account_info(account_iter)?;

    let instruction = Instruction::new_with_bytes(*called_program.key, instruction_data,
    ↪ vec![]);
    invoke(&instruction, accounts)?;
```

```
    Ok(())
}
```

Why is it safe?

- Checks that the number of calls does not exceed the maximum allowed.
- Avoids unexpected failure of contract performance.
- Allows developers to better manage call logic between contracts.

Conclusion

The cross-contract call depth limit in Solana is an important restriction to ensure efficiency and safety, but it can cause problems if not managed properly. Developers must design contracts with this limitation in mind and adopt solutions that avoid execution errors.

Reentrancy Attacks

Description

Reentrancy attacks occur when a contract is called again before its previous execution is completed. This type of vulnerability is known for its impact devastating on Ethereum, as demonstrated by the **DAO hack exploit**, but it can also manifest on the Solana blockchain, albeit in a limited way.

On Solana, reentrancy risks are reduced due to its **atomic transaction** model and static memory management. However, some attacks of reentrancy may still occur when a contract invokes indirectly another program that manipulates shared states without adequate controls.

If an attacker manages to exploit such a vulnerability,it can perform the same operation multiple times before the status is updated correctly, leading to problems such as double withdrawal of funds.

Example

```
fn withdraw_funds(
    accounts: &[AccountInfo],
    amount: u64
) -> ProgramResult {
    let account_iter = &mut accounts.iter();
    let user_account = next_account_info(account_iter)?;
    let vault_account = next_account_info(account_iter)?;

    if **vault_account.lamports.borrow() < amount {
        return Err(ProgramError::InsufficientFunds);
    }

    // Transfer of funds before updating the status
    **vault_account.lamports.borrow_mut() -= amount;
    **user_account.lamports.borrow_mut() += amount;

    Ok(())
}
```

What is the problem?

- The contract **transfers funds before updating the status** .
- If an attacker manages to recall the contract before the previous transaction is fully finalized, it can with draw more funds than it should.
- This vulnerability is possible when the contract calls another program that then calls the initial contract again.

Exploitation

1. An attacker sends a withdrawal request.
2. Before the transaction is finalized, the contract is recalled again through another program.
3. The account balance has not yet been updated, so the contract allows a second withdrawal.
4. The attacker repeats the process several times, taking away more funds than available.

Solution

To prevent reentrancy attacks, it is critical **update contract status before transferring funds** .

Example of corrected code:

```
use solana_program::{
    account_info::{next_account_info, AccountInfo},
    entrypoint::ProgramResult,
    program_error::ProgramError,
};

fn secure_withdraw_funds(
    accounts: &[AccountInfo],
    amount: u64
) -> ProgramResult {
    let account_iter = &mut accounts.iter();
    let user_account = next_account_info(account_iter)?;
    let vault_account = next_account_info(account_iter)?;

    // We create a temporary state variable to keep track of the balance
    let mut vault_balance = **vault_account.lamports.borrow();

    if vault_balance < amount {
        return Err(ProgramError::InsufficientFunds);
    }

    // We update the status before transferring funds
    vault_balance -= amount;

    // Now we can apply the new balance to the vault
    **vault_account.lamports.borrow_mut() = vault_balance;

    // Carry out the transfer of funds
    **user_account.lamports.borrow_mut() += amount;

    Ok(())
}
```

Why is it safe?

- The safe balance is updated before transferring funds. We use a temporary state variable (`vault_balance`) to calculate the new balance before actually changing it.
- If the contract is called up again, it will already find the updated balance. This prevents fraudulent multiple withdrawals before the previous transaction is completed.
- Eliminates the vulnerability of reentrancy. Since the state is updated before each transfer, an attacker cannot perform recursive calls to steal more funds than necessary.

Conclusion

Although Solana 's blockchain has inherent limitations that reduce the risk of reentrancy attacks compared to Ethereum, these can still occur in complex scenarios. Developers must ensure that the status is always updated before any transfer of funds and take preventive measures to avoid recalls unanticipated.

Errors in Logic and Arithmetic

Description

Errors in logic and arithmetic in smart contracts can lead to critical vulnerabilities, causing malfunctions or enabling exploits by attackers.

These errors often result from:

- **Wrong conditions in logic controls** (for example, verification improper permissions or inputs).
- **Inaccurate mathematical calculations**, such as divisions by zero or unintentional rounding.
- **Incorrect handling of numerical limits**, such as overflow and underflow.
- **Failure to handle inconsistent intermediate states** , which can lead to unexpected behavior.

These problems are among the most common causes of exploits in the blockchain, as they can allow attackers to manipulate the contract outputs or perform unauthorized operations.

Example

```
fn calculate_reward(stake_amount: u64, multiplier: u64) -> u64 {
    stake_amount * multiplier / 100
}
```

What is the problem?

- If `multiplier` is 0, a division by zero could be obtained in some contexts.
- If the value of `stake_amount * multiplier`exceeds the limit `u64`, an **integer overflow** occurs .
- Division can lead to unwanted rounding, causing discrepancies in payments.

Exploitation

1. A malicious user could manipulate multiplier to geta negative value or zero, causing erroneous results.

2. If the value of `stake_amount * multiplier` is higher than the maximum for `u64`, overflow could occur, with unforeseen consequences .
3. Users may receive incorrect rewards or perform calculations incorrectly in the contract, causing financial losses.

Solution

To avoid these errors, you must always validate the input parameters and use safe functions for arithmetic calculations.

Example of corrected code:

```
fn calculate_safe_reward(stake_amount: u64, multiplier: u64) -> Result<u64, ProgramError>
 ↪ {
    if multiplier == 0 {
        return Err(ProgramError::InvalidInstructionData);
    }

    let reward = stake_amount
        .checked_mul(multiplier)
        .and_then(|res| res.checked_div(100))
        .ok_or(ProgramError::ArithmeticOverflow)?;

    Ok(reward)
}
```

Why is it safe?

- **Check that the multiplier is not zero** to avoid divisions by zero.
- **Use `checked_mul` and `checked_div`**, functions that prevent arithmetic overflows and underflows by returning `None` in case of an error.
- **Returns a checked error** instead of generating unpredictable results.

Conclusion

Errors in logic and arithmetic are insidious vulnerabilities that can compromise the integrity of a contract. Implement rigorous input controls and use secure functions for mathematical operations is essential to avoid exploits and ensure the correct behavior of the smart contract.

Computational Units Limit

Description

In Solana's blockchain , each transaction must comply with a **maximum limit of computational units (CUs)** , currently set at **48 million CUs per transaction** . This limit is imposed to maintain the 'efficiency of the network and prevent single transactions monopolizing resources.

If a smart contract exceeds this limit, the transaction is **canceled** , even if partial transactions have already been executed. This can lead to problematic situations, such as lost of execution opportunities or inconsistent states in the contract.

Developers must therefore optimize use of computational resources, avoiding unnecessary loops or inefficient operations that could cause CU to exceed the limit.

Example

```
fn inefficient_loop(accounts: &[AccountInfo]) -> ProgramResult {
    let mut total: u64 = 0;

    for _ in 0..1_000_000 {  // Loop eccessivo
        total += 1;
    }

    msg!("Final total: {}", total);
    Ok(())
}
```

What is the problem?

- The `for` loop performs **1 million iterations** , consuming an **excessive number of CU** .
- If the transaction exceeds the **limit of 48 million CU** , the transaction fails.
- In case of partial execution, the contract may be in an **insubstantial state** .

Exploitation

1. An attacker could design transactions with highly computational to force the failure of executions.
2. Poorly designed smart contracts could fail in situations of **high computational load** , causing disruptions and operational vulnerabilities.
3. In **DEX** or financial protocols , exceeding the limit could prevent the completion of crucial transactions, generating economic losses.

Solution

To avoid exceeding the CU limit, developers must **optimize the code** and **reduce intensive calculations** .

Example of corrected code:

```
fn optimized_computation(accounts: &[AccountInfo]) -> ProgramResult {
    let total = 1_000_000;  // Calcolo statico, senza loop inutile

    msg!("Final total: {}", total);
    Ok(())
}
```

Why is it safe?

- Eliminates the **inefficient loop** , drastically reducing the CU consumption.
- The calculation is **deterministic and predictable** , avoiding risks of exceeding the limit.
- Reduces **execution time** , improving the efficiency of the contract.

Conclusion

The limit of **computational units** is a restriction fundamental in Solana, designed to keep the network performant and secure. However , inefficient use of resources can cause the failure of

transactions and compromise the operation of a smart contract. It is essential to optimize the code to reduce CU consumption and ensure the reliability of execution.

Dependencies with Vulnerabilities

Description

One of the most common mistakes in smart contract development is the **use of third-party dependencies without proper verification**. The libraries and frameworks used in programs on Solana **may contain bugs, known exploits or outdated code** , making the contracts vulnerable to attacks.

Vulnerabilities in addictions can result from:

- **Outdated versions** that contain security flaws that are not yet fixed.
- **Unverified dependencies** that might include malicious code.
- **Malicious code injection** by attackers who compromise a popular library.
- **Overreliance on third-party libraries** , without independent auditing .

Example

Let's imagine a contract that uses an outdated version of the library `spl-token`, The standard framework for token management on Solana:

```
[dependencies]
spl-token = "3.1.1"  # Outdated and vulnerable version
```

What is the problem?

- Version 3.1.1 of the library `spl-token` contains **known security bugs**.
- An attacker could exploit a vulnerability in the code of the library to manipulate transactions or gain unauthorized access to funds.
- The contract code becomes dependent on a library **no longer supported** , exposing it to future attacks.

Exploitation

1. An attacker examines the dependencies of a contract to identify **vulnerable versions** .
2. If a library contains a known flaw, it can create malicious transactions to exploit it.
3. In some cases, an attacker could also **take control of the library** , inserting malicious code that is automatically downloaded by unsuspecting developers.

Solution

To protect a contract from vulnerable dependencies, it is essential:

- **Keep libraries updated** to the latest and safest version .
- **Verify vulnerabilities** in dependencies before integrating them.
- **Avoid unnecessary dependencies** by reducing the attack surface .

Example of corrected code:

```
[dependencies]
spl-token = "4.0.0"  # Updated and secure version
```

Why is it safe?

- Version 4.0.0 includes security **fixes for known problems** in the previous version.
- Regular updates ensure that the contract uses the most secure code available.
- The use of a secure version **reduces the risk of attacks related to compromised dependencies** .

Conclusion

The use of **vulnerable dependencies** poses a threat significant to smart contracts on Solana. Regularly update libraries, verify their security, and limit dependence on third-party code are essential practices for preventing exploits and ensure the integrity of the contract.

Conclusion

Security in Solana's blockchain represents a fundamental challenge for developers and users. The vulnerabilities analyzed demonstrate how programming errors, structural limitations of the network and poor development practices can expose smart contracts and decentralized applications to significant risks.

From **integer overflow** to **lack of authorization**, from **vulnerable dependencies** to the **limited depth of cross-contract calls**, each highlighted vulnerability can have critical consequences, including fund losses, contract compromises, and attacks on DeFi protocols.

To mitigate these risks , developers must adopt a **security-oriented mindset** , implementing good development practices, conducting extensive testing and subjecting the code to regular security audits . Some of the key strategies include:

- **Use updated versions of libraries** and avoid unverified dependencies.
- **Implement strict input and authorization controls** to prevent exploits.
- **Avoid logical and arithmetical errors** by thoroughly testing each function.
- **Properly manage network limitations** such as resource consumption and cross-contract call depth.

The Solana blockchain offers advantages in terms of speed and efficiency, but only through **careful management of the security** it will be possible to ensure the stability and the reliability of the ecosystem. The future of blockchain depends on the ability of developers to write secure and resilient code, minimizing risks and improving development practices on an ongoing basis.