verichains

SECURITY AUDIT OF

MULTICHAIN SOLANA ROUTER SMART CONTRACT

Multichain

Public Report

Oct 10, 2022

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Solana	A decentralized blockchain built to enable scalable, user-friendly apps for the world.
SOL	A cryptocurrency whose blockchain is generated by the Solana platform.
Lamport	A fractional native token with the value of 0.000000001 sol.
Program	An app interacts with a Solana cluster by sending it transactions with one or more instructions. The Solana runtime passes those instructions to program.
Instruction	The smallest contiguous unit of execution logic in a program.
Cross-program invocation (CPI)	A call from one smart contract program to another.
Anchor	A framework for Solana's Sealevel runtime providing several convenient developer tools for writing smart contracts.
DAO	A digital Decentralized Autonomous Organization and a form of investor-directed venture capital fund.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Oct 10, 2022. We would like to thank the Multichain for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Multichain Solana Router Smart Contract. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified a small vulnerable issue in the smart contracts code.

Multichain fixed the code, according to Verichains's private report, in commit `4814903744b497da3f5239aec150669aeb5a7b96`.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY 5

1.1. About Multichain Solana Router Smart Contract 5

1.2. Audit scope 5

1.3. Audit methodology 5

1.4. Disclaimer 6

2. AUDIT RESULT 7

2.1. Overview 7

2.2. Findings 7

2.2.1. Lose management of router account if `new` is zero in `apply_mpc` LOW 7

2.2.2. Unnecessary `bump_seed` check when initialize INFORMATIVE 9

3. VERSION HISTORY 11

1. MANAGEMENT SUMMARY

1.1. About Multichain Solana Router Smart Contract

Multichain is the ultimate Router for web3. It is an infrastructure developed for arbitrary cross-chain interactions.

Multichain was born as Anyswap on the 20th July 2020 to service the clear needs of different and diverse blockchains to communicate with each other. Each blockchain has its own unique services that it provides, its own community and its own development ecosystem.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the Multichain Solana Router Smart Contract. It was conducted on commit `91f46c1b7cb0e3bc3b76aa120fcbaac8ed1de1ee` from git repository link: *https://github.com/anyswap/router-solana-contract*.

The latest version of the following file was made available in the course of the review:

SHA256 Sum	File
`9dc28f57b64f9fad6c7b91def4d8c56c7b70e073c898da9813721dfd2f573b95`	`router/src/lib.rs`

1.3. Audit methodology

Our security audit process for Solana smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the Solana smart contract:

- Arithmetic Overflow and Underflow
- Signer checks
- Ownership checks
- Rent exemption checks
- Account confusions
- Bump seed canonicalization
- Closing account
- Signed invocation of unverified programs

- Numerical precision errors
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The Multichain Solana Router Smart Contract was written in `Rust` programming language and `Anchor` framework. It's a router in Solana chain which helps swap token from other chains to Solana or vice versa.

The contract provides users with the ability to send assets to other chains by:

- Transfer SPL tokens/SOL to the router.
- Burn SPL tokens which Multichain is responsible for minting on Solana chain.

The contract then emits a message which will be processed by Multichain and routers in other chains to finish this swap. Please note that this processing is not in our audit scope.

It also provides users with the ability to receive assets from other chains by:

- `mpc` (authority account to manage the router) triggers contract to transfer SPL tokens/SOL to the right receiver.
- `mpc` triggers contract to mint corresponding SLP tokens to the right receiver.

2.2. Findings

During the audit process, the audit team found a small vulnerable issue in the given version of Multichain Solana Router Smart Contract.

Multichain fixed the code, according to Verichains's private report, in commit `4814903744b497da3f5239aec150669aeb5a7b96`.

2.2.1. Lose management of router account if `new` is zero in `apply_mpc` LOW

If the current `mpc` `apply_mpc` with zero public key by mistake, the new `mpc` will be set to zero, and we will lose the management.

```
/// Set pending manage account of pda account `router_account`
pub fn change_mpc(ctx: Context<ChangeMPC>, new: Pubkey) -> Result<()> {
    ctx.accounts.router_account.pending_mpc = new;

    Ok(())
}

/// Change manage account of pda account `router_account`
pub fn apply_mpc(ctx: Context<ChangeMPC>, new: Pubkey) -> Result<()> {
    require!(
            ctx.accounts.router_account.pending_mpc == new,
            RouterError::ApplyWrongAccount
        );
    let old_mpc = ctx.accounts.router_account.mpc;
    ctx.accounts.router_account.mpc = new;
    ctx.accounts.router_account.pending_mpc = Pubkey::zeroed();
    msg!(&format!(
            "ApplyNewMpc transfer from {} to {}",
            old_mpc, new
        ));

    Ok(())
}
```

RECOMMENDATION

Adding requirement that `new` parameter must not equal zero in `apply_mpc`.

```
/// Set pending manage account of pda account `router_account`
pub fn change_mpc(ctx: Context<ChangeMPC>, new: Pubkey) -> Result<()> {
    ctx.accounts.router_account.pending_mpc = new;

    Ok(())
}

/// Change manage account of pda account `router_account`
pub fn apply_mpc(ctx: Context<ChangeMPC>, new: Pubkey) -> Result<()> {
    require!(
            ctx.accounts.router_account.pending_mpc == new,
            RouterError::ApplyWrongAccount
        );
    require!(
        new != Pubkey::zeroed(),
        RouterError::ApplyWrongAccount
    );
    let old_mpc = ctx.accounts.router_account.mpc;
    ctx.accounts.router_account.mpc = new;
    ctx.accounts.router_account.pending_mpc = Pubkey::zeroed();
    msg!(&format!(
            "ApplyNewMpc transfer from {} to {}",
            old_mpc, new
        ));

    Ok(())
}
```

UPDATES

- *Oct 10, 2022*: This issue has been acknowledged and fixed by the Multichain team in commit `4814903744b497da3f5239aec150669aeb5a7b96`.

2.2.2. Unnecessary `bump_seed` check when initialize INFORMATIVE

`Anchor` framework uses the canonical bump by default, so we don't need to calculate and send `bump_seed` to validate again. We can also pre-calculate the canonical bump seed and store it in a constant to avoid cost for storing `bump` and `find_program_address`.

```
pub struct Initialize<'info> {
    #[account(mut)]
    pub initializer: Signer<'info>,
    #[account(
    init,
    seeds = [ROUTER_PDA_SEEDS.as_ref()],
    // bump = bump_seed,
    bump,
    space = 8 + 32 + 1 + 32 + 1,
    payer = initializer,

    )]
    pub router_account: Account<'info, RouterAccount>,
    /// CHECK: `mpc` is the authority account to manage `router_account`
    pub mpc: AccountInfo<'info>,
    pub system_program: Program<'info, System>,
}

pub fn initialize(ctx: Context<Initialize>, bump_seed: u8) -> Result<()> {
    let (_pda, bump) = Pubkey::find_program_address(&[&ROUTER_PDA_SEEDS[..]],
ctx.program_id);
    require!(bump == bump_seed, RouterError::InvalidArgument);

    ctx.accounts.router_account.mpc = *ctx.accounts.mpc.key;
    ctx.accounts.router_account.bump = bump;
    ctx.accounts.router_account.enable_swap_trade = true;
    ctx.accounts.router_account.pending_mpc = Pubkey::zeroed();

    Ok(())
}
```

RECOMMENDATION

Removing unnecessary `bump_seed` parameter.

```
pub fn initialize(ctx: Context<Initialize>,) -> Result<()> {
    let (_pda, bump) = Pubkey::find_program_address(&[&ROUTER_PDA_SEEDS[..]],
ctx.program_id);

    ctx.accounts.router_account.mpc = *ctx.accounts.mpc.key;
    ctx.accounts.router_account.bump = bump;
    ctx.accounts.router_account.enable_swap_trade = true;
    ctx.accounts.router_account.pending_mpc = Pubkey::zeroed();
    Ok(())
}
```

UPDATES

- *Oct 10, 2022*: This issue has been acknowledged by the Multichain team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Oct 07, 2022*	Private Report	Verichains Lab
1.1	*Oct 10, 2022*	Public Report	Verichains Lab

Table 2. Report versions history