SECURITY AUDIT REPORT

for

WarpGate FUN

Prepared By: Xiaomi Huang

PeckShield

December 8, 2024

Document Properties

Client	WarpGate
Title	Security Audit Report
Target	WarpGate FUN
Version	1.0-rc2
Author	Daisy Cao
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0-rc2	December 8, 2024	Daisy Cao	Release Candidate #2
1.0-rc1	October 10, 2024	Daisy Cao	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the WarpGate FUN contract, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About WarpGate FUN

WarpGate FUN is a platform for people to launch tokens on Aptos. The contracts support users to create and trade tokens instantly. Once the bonding process ends, liquidity will be added to liquidSwap by the protocol admin. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of WarpGate FUN

Item	Description
Name	WarpGate
Type	Aptos
Language	Move
Audit Method	Whitebox
Latest Audit Report	December 8, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/hatchy-fun/hatchy.fun-aptos-contract.git (1e017ed)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/hatchy-fun/hatchy.fun-aptos-contract.git (TBD)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Hatchy.fun` implementations. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	4	■■■■
Low	2	■■
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 4 medium-severity vulnerabilities, and 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible Pool Creation Failure in createPool()	Business Logic	TBD
PVE-002	Low	Revisited Function Visibility	Business Logic	TBD
PVE-003	Medium	Lack of Coin Type Validation in mint()	Business Logic	TBD
PVE-004	Low	Suggested fee_address Validation in register_pool()	Business Logic	TBD
PVE-005	Medium	Lack of external Function for withdraw_fee	Business Logic	TBD
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	TBD

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Possible Pool Creation Failure in createPool()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `createPool()`
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

In `WarpGate FUN`, `create_pool()` function is used to register pool and add initial liquidity. During this process, the consistency of the token type order must be ensured. In the process of examining the related pool creation logic, we notice the token type order validation can be improved.

In the following, we show the code snippet of the related `create_pool()` and `register_pool()` functions. In `register_pool()`, the token type order may be adjusted to specific order i.e., `<AptosCoin, CoinType>` (line 85). However, the `add_liquidity` function does not perform any such adjustment. It directly attempts to add liquidity using the token order `<CoinType, AptosCoin>` (line 39). This mismatch can cause the entire `create_pool` operation to fail. Therefore, the consistency of the token type order must be ensured.

```
32     public entry fun createPool<CoinType>(sender: &signer) acquires Config {
33         let sender_addr = signer::address_of(sender);
34         assert!(exists<Config>(sender_addr), ERR_NO_CONFIG);
35         let config = borrow_global_mut<Config>(sender_addr);
36 
37         // init(sender);
38         interface::register_pool<CoinType, AptosCoin>(sender);
39         interface::add_liquidity<CoinType, AptosCoin>(sender,
40             config.total_supply, config.total_supply,
41             0, 0
42         );
43     }
```

Listing 3.1: `create_pool()`

```
    public fun register_pool<X, Y>(account: &signer) {
        assert!(coin::is_coin_initialized<X>(), ERR_NOT_COIN);
        assert!(coin::is_coin_initialized<Y>(), ERR_NOT_COIN);

        create_state<X>(account);
        if (is_order<X, Y>()) {
            implements::register_pool<X, Y>(account);
        } else {
            implements::register_pool<Y, X>(account);
        };
    }
```

Listing 3.2: `register_pool()`

Recommendation The consistency of the token type order must be ensured in above mentioned functions.

Status TBD

3.2 Revisited Function Visibility

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `set_state()`
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

In `Hatchy.fun`, `set_state()` function is used for changing the token reserve. By design, it is invoked by other functions when the token reserve changed i.e. `buy_token()`.

However, it comes to our attention that the function is permissionless and the visibility is public, which means it can be invoked by anyone to set the token reserve. To elaborate, we show below the related code snippet with the `set_state()` function (line 100).

```
100     public fun set_state<CoinType>() acquires PoolState, Config {
101         let coin_addr = coin_address<CoinType>();
102         let state = borrow_global_mut<PoolState<CoinType>>(coin_addr);
103         let config = borrow_global_mut<Config>(@PumpDeployer);
104
105         if (is_order<CoinType, AptosCoin>()) {
106             let (reserve_x, reserve_y) = implements::get_reserves_size<CoinType,
                    AptosCoin>();
107             reserve_x = reserve_x/*  - config.liquidswap_token_value */;
108             reserve_y = reserve_y + config.virtual_apt_value;
109             state.reserve_x = reserve_x;
```

```
            state.reserve_y = reserve_y;
        } else {
            let (reserve_y, reserve_x) = implements::get_reserves_size<AptosCoin,
                CoinType>();
            reserve_x = reserve_x/*  - config.liquidswap_token_value */;
            reserve_y = reserve_y + config.virtual_apt_value;
            state.reserve_x = reserve_x;
            state.reserve_y = reserve_y;
        };
    }
```

Listing 3.3: `set_state()`

Recommendation Change the visibility of above-mentioned routine.

Status TBD

3.3 Lack of Coin Type Validation in mint()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `mint()`
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

As mentioned in Section 3.1, The consistency of the token type order in the pool must be ensured. In the process of examining the related mint logic, we notice the implementation can be improved to better validate the token type.

In the following, we show the code snippet of the related `mint()` routine. It assumes that the token type of `coin_y` is `Aptos`, and add `virtual_apt_value` directly (line 190). Therefore, it may lead to incorrect calculations.

```
180     public(friend) fun mint<X, Y>(
181         coin_x: Coin<X>,
182         coin_y: Coin<Y>,
183     ): Coin<LP<X, Y>>  acquires LiquidityPool, Config {
184         let pool_address = pool_address();
185         assert!(exists<LiquidityPool<X, Y>>(pool_address), ERR_POOL_DOES_NOT_EXIST);
186 
187         let config = borrow_global_mut<Config>(@PumpDeployer);
188 
189         let x_provided_val = coin::value<X>(&coin_x);
190         let y_provided_val = coin::value<Y>(&coin_y) + config.virtual_apt_value;
191 
```

```
        let lp_coins_total = option::extract(&mut coin::supply<LP<X, Y>>());
        let provided_liq = if (0 == lp_coins_total) {
            let initial_liq = math::sqrt(x_provided_val) * math::sqrt(y_provided_val);
            assert!(initial_liq > MINIMAL_LIQUIDITY, ERR_LIQUID_NOT_ENOUGH);
            initial_liq - MINIMAL_LIQUIDITY
        } else {
            let (reserve_x, reserve_y) = get_reserves_size<X, Y>();
            let x_liq = (lp_coins_total as u128) * (x_provided_val as u128) / (reserve_x
                 as u128);
            let y_liq = (lp_coins_total as u128) * (y_provided_val as u128) / (reserve_y
                 as u128);
            if (x_liq < y_liq) {
                assert!(x_liq < (U64_MAX as u128), ERR_UINT_OVERFLOW);
                (x_liq as u64)
            } else {
                assert!(y_liq < (U64_MAX as u128), ERR_UINT_OVERFLOW);
                (y_liq as u64)
            }
        };

        let pool = borrow_global_mut<LiquidityPool<X, Y>>(pool_address);
        coin::merge(&mut pool.coin_x, coin_x);
        coin::merge(&mut pool.coin_y, coin_y);

        // assert!(coin::value(&pool.coin_x) < MAX_POOL_VALUE, ERR_POOL_FULL);
        assert!(coin::value(&pool.coin_y) < MAX_POOL_VALUE, ERR_POOL_FULL);

        event::added_event<X, Y>(pool_address, x_provided_val, y_provided_val,
            provided_liq);
        update_oracle<X, Y>(pool_address, pool);

        let lp_coins = coin::mint<LP<X, Y>>(provided_liq, &pool.lp_mint_cap);

        lp_coins
    }
```

Listing 3.4: `mint()`

Recommendation Validate the token type in `mint()` function.

Status TBD

3.4 Suggested fee_address Validation in register_pool()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `register_pool()`
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

In the `Aptos` chain, there is a design principle that requires a user to proactively register to receive a token before the user can receive the token. While reviewing the `register_pool()` related logic, we notice the token register logic can be improved.

To elaborate, we show below the related code snippet of the `register_pool()` routine. The routine attempts to register tokens `X` and `Y` for the `fee_address` using the `fee_account` (line 256). However, since the `fee_address` and `fee_account` could belong to different entities, this registration may not correct.

```
240     public(friend) fun register_pool<X, Y>(
241         account: &signer
242     ) acquires Config {
243         let pool_account = pool_account();
244         let pool_address = signer::address_of(&pool_account);
245         let fee_account = fee_account();
246         let fee_address = beneficiary();
247 
248         assert!(!exists<LiquidityPool<X, Y>>(pool_address), ERR_POOL_EXISTS_FOR_PAIR);
249 
250         let (lp_name, lp_symbol) = generate_lp_name_and_symbol<X, Y>();
251 
252         let (lp_burn_cap, lp_freeze_cap, lp_mint_cap) =
253             coin::initialize<LP<X, Y>>(&pool_account, lp_name, lp_symbol, 8, true);
254         coin::destroy_freeze_cap(lp_freeze_cap);
255 
256         if (!coin::is_account_registered<X>(fee_address)) {
257             coin::register<X>(&fee_account)
258         };
259 
260         if (!coin::is_account_registered<Y>(fee_address)) {
261             coin::register<Y>(&fee_account)
262         };
263 
264         let pool = LiquidityPool<X, Y> {
265             coin_x: coin::zero<X>(),
266             coin_y: coin::zero<Y>(),
267             timestamp: 0,
268             x_cumulative: 0,
```

```
            y_cumulative: 0,
            lp_mint_cap,
            lp_burn_cap,
        };
        move_to(&pool_account, pool);

        event::created_event<X, Y>(pool_address, signer::address_of(account));
    }
```

Listing 3.5: `register_pool()`

Recommendation Validate the `fee_address` and `fee_account` are same entities.

Status TBD

3.5 Lack of external Function for withdraw_fee

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `withdraw_fee()`
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

In `WarpGate FUN`, the `withdraw_fee()` function is intended to withdraw the protocol's fees, but its visibility is set to friend, which means it can only be called by other functions within the same module or from friend modules. However, there is currently no function that calls it.

To elaborate, we show below the related code snippet of the `withdraw_fee()` routine. It is inaccessible to external entities that might need to trigger fee withdrawals (line 320). Therefore, we recommend an entry function should be implemented that can call the `withdraw_fee()` routine, providing external access while maintaining proper control over the fee withdrawal process.

```
320     public(friend) fun withdraw_fee<Coin>(
321     account: address
322 ) acquires Config {
323     let fee_account = fee_account();
324     let fee_address = signer::address_of(&fee_account);
325
326     let total = coin::balance<Coin>(fee_address);
327     coin::transfer<Coin>(&fee_account, account, total);
328
329     event::withdrew_event<Coin>(pool_address(), total)
330 }
```

Listing 3.6: `withdraw_fee()`

Recommendation Add an entry function for the `withdraw_fee()` while maintaining proper control.

Status TBD

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In `WarpGate FUN`, there is a privileged account, i.e., `@PumpDeployer`. This account plays a critical role in governing and regulating the system-wide operations (e.g., create configuration, add liquidity etc.). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `setParams()` handler as an example and show the representative functions potentially affected by the privileges of the `@PumpDeployer` account.

```
    public entry fun setParams(sender: &signer, feeRecipient: address, feeBasisPoints:
        u64,
            comp_real_apt_amount: u64, comp_fee_apt_amount: u64, comp_self_apt_amount:
                u64,
            virtual_apt_value: u64, liquidswap_token_value: u64, total_supply: u64)
                acquires Config {

        let sender_addr = signer::address_of(sender);
        // check if sender is admin
        assert!(sender_addr == @PumpDeployer, ERR_SENDER_NOT_ADMIN);

        interface::update_swap(sender, feeRecipient, feeBasisPoints, virtual_apt_value,
            liquidswap_token_value);

        // update config
        assert!(exists<Config>(sender_addr), ERR_NO_CONFIG);
        let config = borrow_global_mut<Config>(sender_addr);
        config.total_supply = total_supply;
        config.liquidswap_tokens = liquidswap_token_value;
        config.virtual_apt_value = virtual_apt_value;
        config.comp_real_apt_amount = comp_real_apt_amount;
        config.comp_fee_apt_amount = comp_fee_apt_amount;
        config.comp_self_apt_amount = comp_self_apt_amount;
    }
```

Listing 3.7: `setParams()`

We understand the need of the privileged functions for proper `WarpGate FUN` operations, but at the same time the extra power to the `@PumpDeployer` may also be a counter-party risk to the `WarpGate FUN` contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to `WarpGate FUN` explicit to `WarpGate FUN` contract users.

Status TBD

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Hatchy.fun Aptos` protocol, which allows users to to launch tokens on `Aptos`. The contracts support users to create and trade tokens instantly. Once the bonding process ends, liquidity will be added to `liquidSwap` by the protocol admin. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[3] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[4] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[5] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[6] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[7] PeckShield. PeckShield Inc. https://www.peckshield.com.