BlockSec

Security Audit Report for Permaswap

Date: Aug 29, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.08.29 10:05:27 +08'00'

Contents

Report Manifest

Item	Description
Client	EverFinance
Target	Permaswap

Version History

Version	Date	Description
1.0	Aug 29, 2022	First Release

Chapter 1 Introduction

1.1 About Target Code

Information	Description
Language	Golang
Approach	Semi-automatic and manual verification

The repositories that are audited in this report include the following ones.

Repo Name	Github URL
perma	https://github.com/everFinance/perma

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following. Our audit report is responsible for only the initial version (Version 1), as well as new codes (in the following versions) to fix issues in the audit report.

Project		Commit SHA
perma	Version 1	908e4983bb7f17e3df66621ed50237c2ca3dcb3a

Note that, we did **NOT** audit code that the project depends on but is not included in the repository.

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Access control
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization

∗ Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [1] and Common Weakness Enumeration [2]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while the impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

		Likelihood	
		High	*Low*
Impact	*High*	High	Medium
	Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client but has not been confirmed yet.
- **Confirmed** The item has been recognized by the client but has not been fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[1] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[2] https://cwe.mitre.org/

Chapter 2 Methodology

Before presenting the audit findings, we discuss the audit methodology, including the threat model of Permaswap and detailed techniques used during the audit in this chapter.

2.1 Threat Model

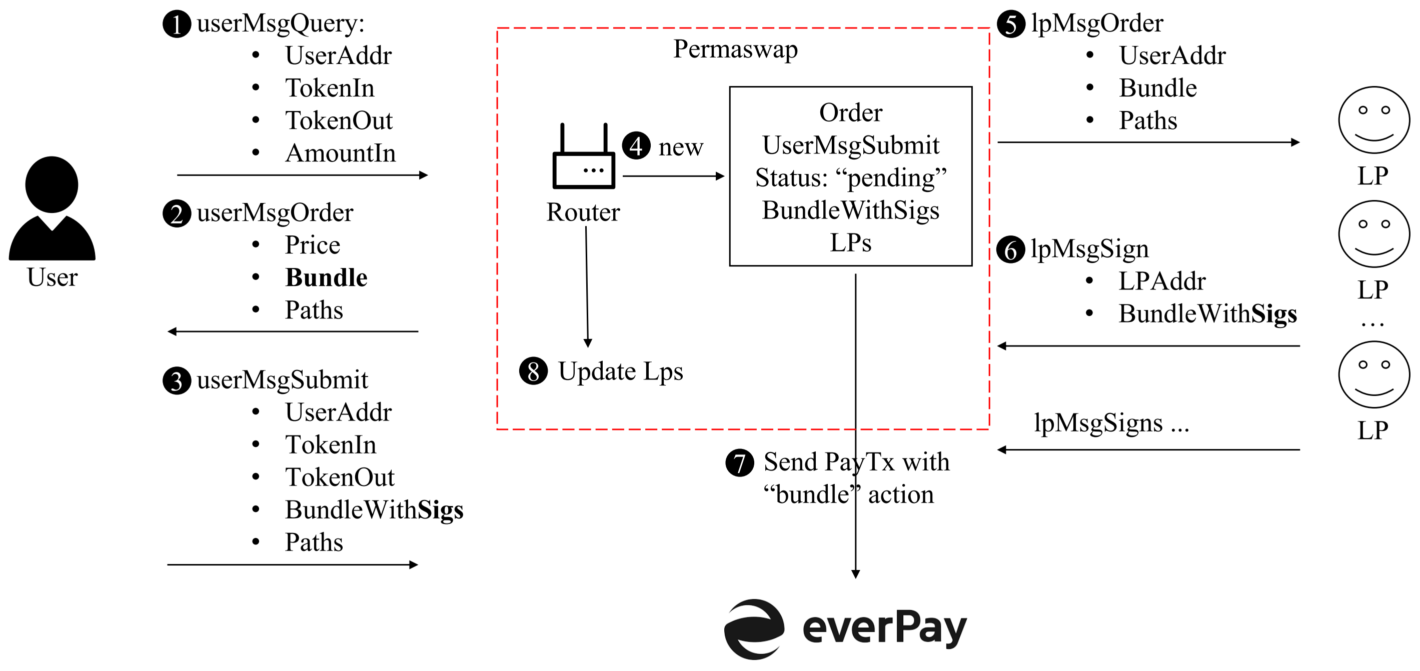

Figure 2.1: How a swap is made by Permaswap

Permaswap is an order maker that integrates the sliding price design of UniswapV3 [1] to provide a more flexible trade making service.

As shown in the Figure 2.1, a swap involves multiple interactions between a user (who fills orders) and Permaswap and between some LPs (who push orders) and Permaswap. **During the audit, we assume the user and LPs could be malicious, which means all messages sent to Permaswap are untrusted.** We enumerate the attack surface based on this assumption.

2.2 Differential Testing

Permaswap re-writes the core logic of UniswapV3 (Solidity) in Golang. Although they are different projects and provide different services, their behaves should be the same given the same inputs.

During the audit, we performed a differential testing between Permaswap and UniswapV3. Specifically, we feed a same test case (e.g., performing a swap) to both projects and compare the result. If the result (e.g., the number of swapped tokens) is the same (or within a threshold of error due to the precision loss - usually the difference of the number of swapped token is $+1$), we think the implementation of Permaswap is consistent with UniswapV3.

[1] http://uniswap.org/whitepaper-v3.pdf

During the testing, we target the `PoolSwap` function of Permaswap and `exactInput` function of UniswapV3. Our testing generated $672,405$ test cases that can be run on UniswapV3. Among them, 635,207 test cases generated the exact same result, and 37,198 test cases generated different result, which is within the scope of differences due to precision loss (the difference of the number of swapped token is $+1$ due to precision loss).

Total	Same	+1
672,405	635,207	37,198

Chapter 3 Findings

In total, we find **two** potential issues. We have **one** recommendation.

- High Risk: 0
- Medium Risk: 0
- Low Risk: 2
- Recommendations: 1
- Notes: 1

ID	Severity	Description	Category	Status
1	Low	DoS attack from LPs	DeFi Security	Confirmed
2	Low	DoS attack from users	DeFi Security	Confirmed
3	-	Remove useless code	Recommendation	Acknowledged
4	-	Discussion about how swap bundles are packed on Arweave chain	Notes	

The details are provided in the following sections.

3.1 DeFi Security

3.1.1 DoS attack from LPs

Severity Low

Status Confirmed

Introduced by `Version 1`

Description An LP in Permaswap is a user who wants to sell cryptocurrency and pushes orders. According to the design of Permaswap, an LP can perform the DoS attack via two methods:

1. Since Permaswap does not set a tick for the low price and high price of each order, there can be an infinite number of different orders theoretically. Particularly, Permaswap iterates all orders to generate a price for each buy query from potential users (who want to buy cryptocurrency). Note that, differing from an AMM, pushing an order (named "adding liquidity" in code) in Permaswap does not require the LP to deposit money in Permaswap. Therefore, a malicious LP can launch a DoS attack by pushing a huge number of different orders, which can waste the project's computing and make Permaswap slow to respond.
2. A swap may involve a lot of LPs. The completion of a swap requires all LPs to sign the swap. If an LP does not sign the swap, then the swap fails. Therefore, a malicious LP can theoretically prevent all swaps from completing by pushing enough orders and not signing all swaps.

Since the two DoS attacks discussed above have no profit for the attacker, we label the item as a low-risk issue according to its low likelihood.

Impact Permaswap may suffer from DoS attack.

Suggestion For the first DoS attack, we recommend the project to follow the tick design from UniswapV3, which can technically eliminate the possibility of the DoS attack. For the second one, we recommend the project punish LPs who do not sign swaps.

Feedback from the Project An LP is required to stake some cryptocurrencies, if an LP rejects to sign swaps or performs other malicious behaviors, we will forfeit his cryptocurrencies.

3.1.2 DoS attack from users

Severity Low

Status Confirmed

Introduced by `Version 1`

Description If a swap is verified by Permaswap and signed by the involved user and LPs, all transfers embedded in the swap will be sent to everPay. According to the design of Permaswap, if a swap passes the verification, then the involved orders (pushed by LPs) will be removed temporarily. After completing the swap, these orders will be pushed back and updated.

We notice that Permaswap does not verify if the user has enough cryptocurrencies to perform the swap. If a malicious user has no enough cryptocurrencies and submits a lot of legal swaps, then all orders in Permaswap will be invisible to normal users, which causes a DoS attack.

Impact Permaswap may suffer from DoS attack.

Suggestion In the verification on a submitted swap, please check the balance of the involved user.

3.2 Additional Recommendation

3.2.1 Remove useless code

Status Acknowledged

Introduced by `Version 1`

Description The variable `isRemoved` is useless.

```go
isRemoved := map[string]bool{}
for _, path := range msg.Paths {

  // don not remove lp twice
  _, ok := lps[path.LpID]
  if ok {
    continue
  }

  lp, err := r.core.RemoveLiquidityByID(path.LpID)

  // make sure only remove one liquidity once
  if isRemoved[path.LpID] {
    continue
  }

  if err != nil {
    log.Warn("move router core to order core failed(RemoveLiquidityByID)", "err", err)
    continue
  }

  lps[lp.ID()] = lp
}
```

Listing 3.1: perma/router/user.go

Impact NA.

Suggestion Remove code in line 129 and 141-143.

3.3 Notes

3.3.1 Discussion about how swap bundles are packed on Arweave chain

Introduced by `Version 1`

Description By design, when a new watchman joins everPay, he fetches and applies all historical PayTxs from Arweave chain to catch up the world state. However, based on the code `everSDK.Bundle(......`, we guess that PayTxs embedded with swap bundles may not be posted on Arweave chain but be sent to everPay directly. If so, a new watchman can not fetch all historical PayTxs from Arweave chain and catch up the world state.

```go
func (o *Order) submitToEver() {
    if o.dryRun || o.router.sdk == nil {
        o.EverHash = "0x0000000000000000000000000000000000000000000000000000000000000000"
        o.Status = schema.OrderStatusSuccess
        return
    }

    everSDK := o.router.sdk
    var err error
    var everTx *paySchema.Transaction
    for {
        if o.Bundle.Expiration < time.Now().Unix() {
            o.Status = schema.OrderStatusExpired
            return
        }
        // need retry
        if everTx, err = everSDK.Bundle("ETH", everSDK.AccId, big.NewInt(0), *o.Bundle); err ==
             nil {
            break
        } else {
            time.Sleep(1 * time.Second)
            log.Error("sumbit to everPay failed", "err", err)
        }
    }
```

Listing 3.2: router/order.go/submitToEver

Feedback from the Project We have a program named coordinator that performs verification and simulated execution on PayTxs like a watchman. After that, coordinator **posts PayTxs on Arweave chain**. Note that, Permaswap can fetch an early execution result from coordinator to check if the swap bundles can be applied by watchmen. The above is the process after executing the code `everSDK.Bundle(...`.