SMART CONTRACT AUDIT REPORT

for

FLAMINGO

Prepared By: Shuxiao Wang

Hangzhou, China

Dec. 22, 2020

Document Properties

Client	Flamingo
Title	Smart Contract Audit Report
Target	Flamingo Perp
Version	1.0
Author	Edward Lo
Auditors	Edward Lo, Ruiyi Zhang
Reviewed by	Shuxiao Wang, Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	Dec. 22, 2020	Edward Lo	Final Release
1.0-rc2	Sep. 25, 2020	Edward Lo	Release Candidate #2
1.0-rc1	Sep. 20, 2020	Edward Lo	Release Candidate #1

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the **Flamingo Perp** smart contract, we outline in this 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 branch of Flamingo Perp smart contract 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 Flamingo Perp Contract

Flamingo Perp is a vAMM-based perpetual contract exchange for virtually any underlying assets with infinite liquidity. Similar to Swap, traders can trade perpetual contracts using the same CPMM model with 10x long or short leverage. Funding rate is introduced to ensure the contract price converging with the actual price. Price feed will be provided through Flamingo's oracle contract. As mentioned earlier, Flamingo Perp assumes a trusted oracle with timely market price feeds and the oracle itself is not part of this audit.

The basic information of Flamingo Perp smart contract is as follows:

Table 1.1: Basic Information of Flamingo Perp smart contract

Item	Description
Issuer	Flamingo
Website	https://flamingo.finance/
Type	Neo Smart Contract
Platform	C#
Audit Method	Whitebox
Latest Audit Report	Dec. 22, 2020

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

this audit:

- https://github.com/flamingo-finance/flamingo-contract-perp (57dd375)

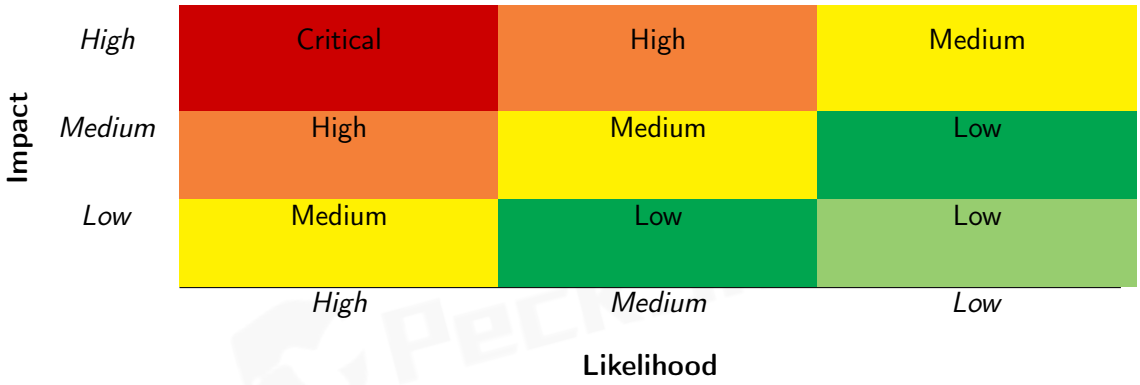
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/flamingo-finance/flamingo-contract-perp (06a0fd4)

1.2 About PeckShield

PeckShield Inc. [6] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [5]:

- 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.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Basic Coding Bugs Checks
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
	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

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 classified into four categories accordingly, 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 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) [4], 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.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 Logic	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 Flamingo Perp implementation. 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	1	■
High	1	■
Medium	1	■
Low	3	■■■
Informational	1	■
Total	7	

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 1 critical-severity vulnerability, 1 high-severity vulnerability, 1 medium-severity vulnerability, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Type	Status
PVE-001	Medium	Missed Amount Check in buyFromWhitelisted()	Business Logic	Fixed
PVE-002	Low	Lack of Scale Precision Check in scaleXY()	Business Logic	Fixed
PVE-003	Informational	Erroneous Description in the setMaintenanceMarginRate()	Inaccurate Comments	Fixed
PVE-004	Low	Lack of Pool Fee Rate Precision Check in setPoolFeeRate()	Business Logic	Confirmed
PVE-005	Low	Lack of Minimum Lot Size Check in setLotSize()	Business Logic	Fixed
PVE-006	Critical	Missed VAMM Permission Checks in Trade()	Business Logics	Fixed
PVE-007	High	Missed Liquidator Permission Checks in liquidateAndSettlement()	Business Logics	Fixed

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 Chapter 3 for details.

3 Detailed Results

3.1 Missed Amount Check in buyFromWhitelisted()

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: flamingo-contract-amm/AMM.cs
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

Similar to Perpetual Protocol, Flamingo Perp allows traders to speculate on the future price of a given asset by buying (going long) or selling (going short) perpetual futures contracts. In Flamingo Perp, the flamingo-contract-amm/AMM contract provides interfaces for users to open long / short positions. Function buy() would be called when a user wants to open a long position.

```
143 public static BigInteger buy(byte[] trader, BigInteger amount, BigInteger limitPrice,
        BigInteger deadline)
144 {
145     Assert(Runtime.CheckWitness(trader), "CheckWitness failed");
146     Assert(amount > 0, "amount should > 0");
147     return buyFrom(trader, amount, limitPrice, deadline);
148 }
```

Listing 3.1: flamingo-contract-amm/AMM.cs

```
103 private static BigInteger buyFrom(byte[] trader, BigInteger amount, BigInteger
        limitPrice, BigInteger deadline)
104 {
105     byte[] perpScriptHash = getPerpContractScriptHash();
106
107     BigInteger price = getPrice(amount); // price has decimals of 8
108     Assert(limitPrice >= price, "price limited");
109     Assert(Runtime.Time <= deadline, "deadline exceeded");
110
111     BigInteger value = price * amount; // amount has decimals of 8
```

```
112     BigInteger fee = value * getPoolFeeRate() / RateDecimal;  // value has decimals of
            16
113
114     BigInteger fee2Dev = fee / 2;
115
116     byte[] dev = getDevAddress();
117     Assert(dev.Length == 20, "devAddress.Length should == 20");
118     bool success = ((Func<string, object[], bool>)perpScriptHash.ToDelegate())("
            TransferCashBalance", new object[] { trader, dev, fee2Dev });
119     Assert(success, "transfer to dev failed");
120
121     BigInteger fee2Insurance = fee - fee2Dev;
122     bool res = ((Func<string, object[], bool>)perpScriptHash.ToDelegate())("
            TransferCashBalance", new object[] { trader, perpScriptHash, fee2Insurance });
123     Assert(res, "transferCashBalance failed");
124
125
126     BigInteger opened = ((Func<string, object[], BigInteger>)perpScriptHash.ToDelegate()
            )("Trade", new object[] { trader, SideType.LONG, price, amount });
127
128     forceFunding();
129     // TODO: BuyFrom event helps debug and show the variables, can be deleted later
130     BuyFrom(new object[] { trader, amount, price, value, opened, fee, fee2Dev,
            fee2Insurance });
131     return opened;
132 }
```

Listing 3.2: flamingo-contract-amm/AMM.cs

As shown in the above code snippet, the function `buy()` will check the open position amount > 0, then transfer control to the function `buyFrom()`. Within the routine, it will calculate the position price and related fee (line 107-112), and execute the trade eventually (line 126).

```
135 public static BigInteger buyFromWhitelisted(byte[] auth, byte[] trader, BigInteger
        amount, BigInteger limitPrice, BigInteger deadline)
136 {
137     Assert(Runtime.CheckWitness(auth), "auth checkwitness failed");
138     Assert(isAuthorized(auth), "not authorized");
139     return buyFrom(trader, amount, limitPrice, deadline);
140 }
```

Listing 3.3: flamingo-contract-amm/AMM.cs

On the other hand, authorized users can trade on anyone's behalf by `buyFromWhitelisted()` .`buyFromWhitelisted()` will check the caller is authorized (lines 137-138). Then the flow is transferred to `buyFrom()`, same as `buy()`. However, it misses the amount sanity check. If an authorized user accidentally give a negative amount, it might cause severe damage to the trading system.

The same issue is also applicable to `sellFromWhitelisted()`.

Recommendation Add sanity checks in `buyFromWhitelisted()` and `sellFromWhitelisted()` to ensure `amount` > 0.

Status The issue has been fixed by this commit: `06a0fd4`.

3.2 Lack of Scale Precision Check in scaleXY()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `flamingo-contract-amm/AMM.cs`
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

Similar to Uniswap, Flamingo Perp utilizes automated market maker (AMM) design (constant product curve). But rather than rely on liquidity providers to determine the curve of a given market, Flamingo Perp can programmatically set and update the parameters of the virtual AMM (x*y=k). In Flamingo Perp, the `flamingo-contract-amm/AMM` contract provides interfaces for admin to adjust the ratio of the token pair. Function `scaleXY()` is responsible for handling the adjustment.

```
30  public static bool scaleXY(BigInteger ratio)
31  {
32      Assert(Runtime.CheckWitness(getAdmin()), "not allowed to scaleXY");
33      XY xy = Xy;
34      Assert(xy.x != 0 && xy.y != 0, "x and y not initialized");
35      Xy = new XY
36      {
37          x = xy.x * ratio / Ten2Power18,
38          y = xy.y * ratio / Ten2Power18,
39      };
41      XY newXy = Xy;
42      Assert(newXy.x / Ten2Power8 > 0 && newXy.y / Ten2Power8 > 0, "prevent wrong ratio")
            ;
44      ScaleXY(xy.x, xy.y, newXy.x, newXy.y);
45      return true;
46  }
```

Listing 3.4: flamingo-contract-amm/AMM.cs

The passed in `ratio` is used for scaling calculation (lines 36-39), and it's assumed to be 10^18 precision, as written in the Flamingo Perp specification. However, there is no sanity check to guarantee the passed in `ratio` precision follows the specification. It might cause the confusion if the admin accidentally passes in a ratio which doesn't match the precision.

Recommendation Add a sanity check for `ratio` to guarantee its precision is 10^18.

Status The issue has been fixed by this commit: `f5f5e86`.

3.3 Erroneous Description in the setMaintenanceMarginRate()

- ID: PVE-003
- Severity: Informational
- Likelihood: High
- Impact: N/A
- Target: flamingo-contract-perp/Governance.cs
- Category: Inaccurate Comments [1]
- CWE subcategory: CWE-1116 [1]

Description

In Flamingo Perp, the flamingo-contract-perp/Governance contract provides interfaces for admin to adjust several rates used in Flamingo Perp:

- Initial Margin Rate: the initial margin rate required for the account to open long or short positions
- Maintenance Margin Rate: the margin rate for the account to continue trading
- Liquidation Penalty Rate / Penalty Fund Rate / etc

The function setMaintenanceMarginRate() is responsible for setting the maintenance margin rate used in Flamingo Perp.

```
64 public static bool setMaintenanceMarginRate(BigInteger value)
65 {
66     checkAdmin();
67     Assert(value > 0, "MaintenanceMarginRate should > 0");
68     Assert(value < getInitialMarginRate(), "MaintenanceMarginRate should <
           initialMarginRate");
69     Assert(value > getLiquidationPenaltyRate(), "MaintenanceMarginRate should <
           liquidationPenaltyRate");
70     Assert(value > getPenaltyFundRate(), "MaintenanceMarginRate should < PenaltyFundRate
           ");
71     Storage.Put(MaintenanceMarginRatePrefix, value);
72     return true;
73 }
```

Listing 3.5: flamingo-contract-perp/Governance.cs

The function setMaintenanceMarginRate() can only be called by admin (line 66). There are some other sanity checks, e.g., the maintenance margin rate should less than the initial margin rate, and should bigger than the penalty fund rate and liquidation penalty rate.

The check logic is correct, however, the descriptions when assert occurs are erroneous (lines 69-70)

Recommendation Correct the corresponding descriptions.

Status The issue has been fixed by this commit: `f5f5e86`.

3.4 Lack of Pool Fee Rate Precision Check in setPoolFeeRate()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `flamingo-contract-amm/Governance.cs`
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

As we introduced in Section 3.3, the `flamingo-contract-amm/Governance` contract provides interfaces for the admin to set different parameters and rates used in Flamingo Perp. The pool fee rate is the rate for the transaction fees, which are composed of fees for insurance and Flamingo Perp team. `setPoolFeeRate()` is responsible for setting the pool fee rate.

```
113 public static bool setPoolFeeRate(BigInteger poolFeeRate)
114 {
115     Assert(Runtime.CheckWitness(getAdmin()), "not admin");
116     Storage.Put(PoolFeeRateKey, poolFeeRate);
117     SetPoolFeeRate(poolFeeRate);
118     return true;
119 }
```

Listing 3.6: flamingo-contract-amm/Governance.cs

The function `setPoolFeeRate()` can only be called by admin (line 115). Then the passed in `poolFeeRate` will be saved in the storage and used as the pool fee rate. However, there is no sanity check to make sure the precision of `poolFeeRate` follows the specification, e.g., 10^5.

Recommendation Add a sanity check for `poolFeeRate` to guarantee its precision is 10^5.

Status The issue has been confirmed.

3.5 Lack of Minimum Lot Size Check in setLotSize()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: flamingo-contract-perp/Collateral.cs
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

As we introduced in Section 3.1, Flamingo Perp allows traders to speculate on the future price of a given asset by buying (going long) or selling (going short) perpetual futures contracts. In Flamingo Perp, the flamingo-contract-perp/Collateral contract provides interfaces for user / admin to deposit / withdraw / transfer or set different parameters used in Flamingo Perp for the admin. In addition, within the impletation of Flamingo Perp, the calculation is assumed to be 10^18 precision, as written in the specification. In particular, since the precision of originToken is less than 10^18, the function setLotSize() is responsible for setting the minimum property unit used in Flamingo Perp.

```
111 public static bool setLotSize(BigInteger lotSize)
112 {
113     checkAdmin();
114     Storage.Put(lotSizePrefix, lotSize);
115     return true;
116 }
```

Listing 3.7: flamingo-contract-perp/Collateral.cs

The function setLotSize() can only be called by admin (line 113). Then the passed in lotSize will be saved in the storage and used as the lot size. However, there is no sanity check to make sure the precision of lotSize follows the specification, e.g., multiple of 10.

Recommendation Add a sanity check for lotSize to guarantee its precision is multiple of 10.

Status The issue has been fixed by this commit: f5f5e86.

3.6 Missed VAMM Permission Checks in Trade()

- ID: PVE-006
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `Perp.cs`
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

As we introduced in Section 3.1, Flamingo Perp allows traders to speculate on the future price of a given asset by buying (going long) or selling (going short) perpetual futures contracts. The function `buy()` would be called when a user wants to open a long position. It will calculate the position price and related fee, and execute the function `trade` eventually.

```
95  if (method == "Trade")
96  {
97      //if (ExecutionEngine.CallingScriptHash != getVAMM()) return false;
98      var trader = getMarginAccount((byte[])args[0]);
99      SideType side = (bool)args[1] ? SideType.LONG : SideType.SHORT;
100     Runtime.Notify("trade start");
101     return trade(trader, side, (BigInteger)args[2], (BigInteger)args[3]);
102 }
```

Listing 3.8: Perp.cs

However, the function `Trade()` doesn't check the caller is vAMM (line 97). Specifically, an attacker can modify the state of MarginAccounts easily, which causes severe damage to the trading system.

Recommendation Add permission check on `Trade()` as follows:

```
95  if (method == "Trade")
96  {
97      if (ExecutionEngine.CallingScriptHash != getVAMM()) return false;
98      var trader = getMarginAccount((byte[])args[0]);
99      SideType side = (bool)args[1] ? SideType.LONG : SideType.SHORT;
100     Runtime.Notify("trade start");
101     return trade(trader, side, (BigInteger)args[2], (BigInteger)args[3]);
102 }
```

Listing 3.9: Perp.cs

Status The issue has been fixed by this commit: `3f2c986`.

3.7 Missed Liquidator Permission Checks in liquidateAndSettlement()

- ID: PVE-007
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Perp.cs`
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

As we introduced in Section 3.3, the function `setMaintenanceMarginRate()` is responsible for setting the maintenance margin rate. In Flamingo Perp, the `flamingo-contract-amm/Perp` contract provides interfaces for liquidator to liquidate unsafe accounts' asset. During the liquidation process, the liquidatorAddress and traderAddress are passed into `liquidateAndSettlement` function. As shown in the code snippets below, there is no permission check in the function `liquidateAndSettlement()` to make sure the caller is `liquidator`.

```
281 public static BigInteger liquidateAndSettlement(byte[] account, byte[] liquidator)
282 {
283     Assert(getPerpStatus() != 3, "wrong perpetual status");
284     MarginAccount marginAccount = getMarginAccount(account);
285     var currentMarkPrice = markPrice();
286     Assert(currentMarkPrice > 0, "price should > 0");
287     Assert(!isSafeImplementation(marginAccount, currentMarkPrice), "safe account");
288     BigInteger liquidateionAmount = calculateLiquidateAmount(marginAccount,
            currentMarkPrice);
289     BigInteger opened = liquidateNsettlementImplementation(liquidator, account,
            currentMarkPrice, liquidateionAmount);
290     MarginAccount liquidateAccount = getMarginAccount(liquidator);
291     if (opened > 0)
292     {
293         Assert(availableMarginWithPrice(liquidateAccount, currentMarkPrice) >= 0, "
                liquidator margin");
294     }
295     else
296     {
297         Assert(isSafeImplementation(liquidateAccount, currentMarkPrice), "liquidator
                unsafe");
298     }
299     liquidation(liquidator, account, liquidateionAmount, currentMarkPrice);
300     return currentMarkPrice * liquidateionAmount;
301 }
```

Listing 3.10: Perp.cs

Recommendation Add permission check on `liquidateAndSettlement()` as follows:

```
281 public static BigInteger liquidateAndSettlement(byte[] account, byte[] liquidator)
282 {
283     Assert(Runtime.CheckWitness(liquidator), "checkwitness failed");
284     Assert(getPerpStatus() != 3, "wrong perpetual status");
285     MarginAccount marginAccount = getMarginAccount(account);
286     var currentMarkPrice = markPrice();
287     Assert(currentMarkPrice > 0, "price should > 0");
288     Assert(!isSafeImplementation(marginAccount, currentMarkPrice), "safe account");
289     BigInteger liquidateionAmount = calculateLiquidateAmount(marginAccount,
            currentMarkPrice);
290     BigInteger opened = liquidateNsettlementImplementation(liquidator, account,
            currentMarkPrice, liquidateionAmount);
291     MarginAccount liquidateAccount = getMarginAccount(liquidator);
292     if (opened > 0)
293     {
294         Assert(availableMarginWithPrice(liquidateAccount, currentMarkPrice) >= 0, "
                liquidator margin");
295     }
296     else
297     {
298         Assert(isSafeImplementation(liquidateAccount, currentMarkPrice), "liquidator
                unsafe");
299     }
300     liquidation(liquidator, account, liquidateionAmount, currentMarkPrice);
301     return currentMarkPrice * liquidateionAmount;
302 }
```

Listing 3.11: Perp.cs

Status The related code snippet has been removed by this commit: `f5f5e86`.

4 Conclusion

In this audit, we have analyzed the Flamingo Perp design and implementation. The Flamingo Perp is a vAMM-based perpetual contract exchange for virtually any underlying assets with infinite liquidity. During the audit, we noticed that the current code base is well organized and those identified issues are promptly confirmed and fixed.

As a precaution, 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-1116: Inaccurate Comments. https://cwe.mitre.org/data/definitions/1116.html.

[2] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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